Material Maratona de Programação

NPcomp - Nucleo de Programação Competitiva

2024

Conteúdo

Capítulo 1

Teoria e algoritmos gerais

1.1 C++ STL

1.1.1 Vector

Estrutura que armazena itens como em arrays, porém permite a manipulação dinâmica de seu tamanho. Pode ser usada para manipular lista de elementos e modelar matrizes.

Essa estrutura está presente na STL, e pode ser usado a partir das definições.

```
1 // Exemplo de definicao
2 // n: tamanho inicial, v: valor inicial
3 vector<int> vec;
4 vector<int> vec(n,v);
5 vector<vector<int>> vec(n, vector<int>(n,v));
```

Lista de funções

FUNC **begin()** $O(n)$
Retorna um iterador para o primeiro elemento do vetor.

FUNC **end() 1**
Retorna um iterador para o fim do vetor, posição após o último elemento.

FUNC **rbegin() 1**
Retorna um iterador reverso para o início do vetor.

FUNC **rend() 1**
Retorna um iterador reverso para o fim do vetor.

FUNC **size() 1**
Retorna o tamanho atual do vetor.

FUNC **assign(size_t n, T val)** $O(n)$
Redefine o vetor com o novo tamanho e valor, respectivos.

FUNC **resize(size_t n)** $O(n)$
Altera o tamanho do vetor.

FUNC **empty() 1**
Verifica se o vetor está vazio.

FUNC **front() 1**
Acessa o primeiro elemento do vetor.

FUNC **back() 1**
Acessa o último elemento do vetor.

FUNC **push_back(T val) 1**
Adiciona um elemento ao final do vetor.

FUNC **pop_back() 1**
Remove o último elemento do vetor.

FUNC **insert(iterator position, T val)** $O(n)$
Insere elementos antes da posição especificada.

FUNC **erase(iterator position)** $O(n)$
Remove um único elemento da posição especificada.

FUNC **clear()** $O(n)$
Remove todos os elementos do vetor.

1.1.2 Set

Estrutura que armazena itens de maneira ordenada, com valores únicos. Os elementos são acessados por meio de iteradores, não é possivel indexar. Pode ser usada para manipular conjuntos, identificar elementos únicos, ordenar os elementos.

Essa estrutura está presente na STL, e pode ser usado a partir das definições.

```
1 // Exemplo de definicao
2 set<int> st;
```

Lista de funções

Func **begin()** $O(1)$
Retorna um iterador para o primeiro elemento do set.

Func **end()** $O(1)$
Retorna um iterador para o fim do set, posição após o último elemento.

Func **rbegin()** $O(1)$
Retorna um iterador reverso para o início do set.

Func **rend()** $O(1)$
Retorna um iterador reverso para o fim do set.

Func **size()** $O(1)$
Retorna o tamanho atual do set.

Func **empty()** $O(1)$
Verifica se o set está vazio.

Func **insert(T val)** $O(n)$
Insere elemento ao set.

Func **erase(iterator pos)** $O(n)$
Remove um único elemento da posição especificada.

Func **clear()** $O(n)$
Remove todos os elementos do set.

Func **find()** $O(n)$
Remove todos os elementos do set.

Func **clear()** $O(n)$
Remove todos os elementos do set.

Func **find(T val) logn**
Retorna um iterador para o elemento com o valor especificado, ou end() se não encontrado.

Func **count(T val)** $O(logn)$
Conta o número de elementos com o valor especificado.

Func **lower_bound(T val)** $O(logn)$
Retorna um iterador para o primeiro elemento igual ou maior que o valor especificado.

Func **upper_bound(T val)** $O(logn)$
Retorna um iterador para o primeiro elemento maior que o valor especificado.

Func **equal_range(T val)** $O(logn)$
Retorna um par de iteradores, lower_bound e upper_bound.

1.1.3 Stack

Estrutura que armazena itens de maneira empilhada, seguindo o conceito LIFO, no qual o último a ser inserido é o primeiro a sair. Os elementos são acessados por meio de desempilhamentos, não é possivel indexar. Pode ser usada para modelar problemas, como busca em profundidade em grafos.

Essa estrutura está presente na STL, e pode ser usado a partir das definições.

```
1 // Exemplo de definicao
2 stack<int> stk;
```

Lista de funções

Func **size()** $O(1)$
Retorna o tamanho atual da stack.

Func **empty()** $O(1)$
Verifica se o set está vazio.

Func **push(T val)** $O(1)$
Insere elemento ao stack.

Func **top()** $O(n)$
Retorna o próximo elemento.

Func **pop()** $O(n)$
Remove o próximo elemento.

1.1.4 Queue

Estrutura que armazena itens de maneira enfileirada, seguindo o conceito FIFO, no qual o primeiro a ser inserido é o primeiro a sair. Os elementos são acessados por meio de desenfileiramentos, não é possivel indexar. Pode ser usada para modelar problemas, como busca em largura em grafo.

Essa estrutura está presente na STL, e pode ser usado a partir das definições.

```
1 // Exemplo de definicao
2 queue<int> q;
```

Lista de funções

FUNC **size()** $O(1)$
Retorna o tamanho atual da stack.

FUNC **empty()** $O(1)$
Verifica se o set está vazio.

FUNC **push(T val)** $O(1)$
Insere elemento ao stack.

FUNC **front()** $O(n)$
Retorna o primeiro elemento.

FUNC **back()** $O(n)$
Retorna o ultimo elemento.

FUNC **pop()** $O(n)$
Remove o próximo elemento.

1.1.5 Priority Queue

Estrutura que mantem o maior ou menor elemento na primeira posição, por padrão segue o conceito de Max-heap, mas pode ser alterada para ser uma Min-heap. Os elementos são acessados por meio de desenfileiramento, não é possivel indexar. Pode ser usada para obter o maior ou menor elemento de maneira eficiente, caso essa operação se repita.

Essa estrutura está presente na STL, e pode ser usado a partir das definições.

```
1 // Exemplo de definicao
2 priority_queue<int> pq; //Max-Heap
3 priority_queue<int, vector<int>, greater<int>> pq; //Min-Heap
```

LISTA DE FUNÇÕES

FUNC **size()** $O(1)$
Retorna o tamanho atual da priority-queue.

FUNC **empty()** $O(1)$
Verifica se a priority-queue está vazia.

FUNC **push(T val)** $O(1)$
Insere elemento a priority-queue.

FUNC **top()** $O(n)$
Retorna o primeiro elemento.

FUNC **pop()** $O(n)$
Remove o elemento do topo.

1.2 Disjoint Set Union - DSU

Essa estrutura é focada na criação e união de conjuntos. As operações básicas e suas complexidadessão:

- `make_set` : cria um novo conjunto $O(1)$
- `union_sets` : une dois conjuntos $O(logn)$
- `find_set` : retorna o elemento representativo do conjunto de determinado elemento $O(logn)$

A complexidade das operações pode ser otimizada. No código abaixo, a função `find_set` usa um otimização conhecida como *path compression*, e a união é feita por *rank* (o tamanho de cada conjunto). Na prática, essas otimizações deixam as consultas com ordem próxima à constante.

Implementação

```
1  void make_set(int v) {
2      parent[v] = v;
3      size[v] = 1;
4  }
5
6  int find_set(int v) {
7      if (v == parent[v]) return v;
8      return parent[v] = find_set(parent[v]);
9  }
10
11 void union_sets(int a, int b) {
12     a = find_set(a);
13     b = find_set(b);
14     if (a != b) {
15         if (size[a] < size[b])
16             swap(a, b);
17         parent[b] = a;
18         size[a] += size[b];
19     }
20 }
```

Capítulo 2

Strings

2.0.1 Longest Preffix also Suffix

Função utilizada para encontrar o maior prefixo que também seja um sufixo em uma string. Pode ser usado para implementar algoritmos de casamento de caracteres, KMP, que usam o array LPS, para manter a complexidade linear.

Essa função pode ser implementada atraves de um array de tamanho N, referente a string analisada, para cada posição ficará indicado, o maior sufixo que termina na posicao i, que também é prefixo.

Lista de funções

Func **construct(string s)** $O(n)$
Constroi o array LPS para a string 's'.

```
1  int lps[N];
2
3  void construct(string word) {
4      int n = str.size();
5      for (int i = 1, j = 0; i < n; i++) {
6          while(j > 0 && str[i] != str[j]) j = lps[j-1];
7          if(str[i] == str[j]) j++;
8          lps[i] = j;
9      }
10 }
```

2.0.2 Z array

Função utilizada para encontrar o maior prefixo que também seja uma substring. Pode ser usado para implementar algoritmos de casamento de caracteres, que usam o array Z, para manter a complexidade linear.

Essa função pode ser implementada atraves de um array de tamanho N, referente a string analisada, para cada

posição ficará indicado, a maior substring que começa na posicao i, que também é prefixo.

Lista de funções

Func **construct(string s)** $O(n)$
Constroi o array Z para a string 's'.

```
1  int z[N];
2
3  void construct(string str) {
4      int n = str.size();
5      for (int i = 1, l = 0, r = 0; i < n; i++) {
6          if(i <= r) z[i] = min(z[i-l], r-i+1);
7          while(i + z[i] < n && str[z[i]] == str[i + z[i]]) z[i]++;
8          if(i + z[i] - 1 > r) {
9              l = i;
10             r = i + z[i] - 1;
11         }
12     }
13 }
```

2.0.3 String Hash

Uma string hash é uma função que associa uma string a um número inteiro. Ela é frequentemente usada em algoritmos de casamento de padrões, pois permite verificar se duas strings são iguais em tempo constante, usando o hash.

Para calcular o hash de uma string, podemos usar a fórmula:

$$H(s) = \sum_{i=0}^{n-1} s[i] * p^i \pmod m$$

Onde s é a string, n é o tamanho da string, p é um número primo e m é um número inteiro.

Essa estrutura é implementada através de um array de tamanho N, que guarda os valores do hash de todos os prefixos da string, e um array para armazenar as potencias do número primo. Com essa estrutura podemos verificar se duas strings são iguais em tempo constante.

Lista de funções

FUNC **build_pow()** $O(n)$
Pre-calcula as potências do número primo usado no hash.

FUNC **build_hash(string s)** $O(n)$
Calcula o hash de todos os prefixos da string 's'.

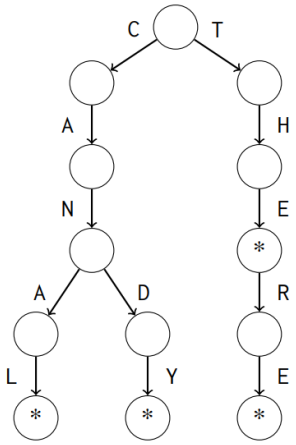
FUNC **get_hash(int l, int r)** $O(1)$
Calcula o hash da substring de 's' que começa na posição 'l' e termina na posição 'r'.

```
1  const ll phash1 = 257;
2  const ll mod = 1e9 + 7;
3
4  ll powh1[mxN] = {1}; // Pre-calcula as potencias da hash
5  ll valh1[mxN]; //Pre calcula o hash de todos os prefixos
6
7  void build_pow() {
8      for (int i = 1; i < n; i++) {
9          powh1[i] = (powh1[i-1]*phash1) % mod;
10     }
11 }
12
13 void build_hash(string str) {
14     valh1[0] = str[0];
15     for (int i = 0; i < n; i++) {
16         valh1[i+1] = (valh1[i] + str[i]*powh1[i]) % mod;
17     }
18
19 }
20
21 ll get_hash(int l, int r) {
22
23     ll h1 = (valh1[r]-valh1[l] + mod) % mod;
24     h1 = (h1*powh1[n-l-1])% mod;
25
26     return h1;
27
28 }
```

2.0.4 Trie

Estrutura que mantém um conjunto de strings, na forma de árvore. De forma que se duas strings possuem um prefixo em comum, elas possuem o mesmo caminho na arvore. Pode ser bastante útil quando é necessário lidar com múltiplas palavras, para identificar se uma palavra pertence ao conjunto, ou de quantas maneiras diferentes ela pode ser formada dado o conjunto.

Na figura abaixo temos o exemplo do conjunto CANDY, CANAL, THE, THERE, os asteriscos indicam o fim de uma string.



Essa estrutura pode ser utilizada através de uma matriz contendo N (tamanho da maior string) linhas, M (tamanho do alfabeto) colunas, uma variavel para contabilizar o numero de nós, e um array booleano para indicar o fim das strings.

Lista de funções

Func **insert(string s)** $O(n)$
Insere a string 's' na arvore trie.

Func **search(string s)** $O(n)$
Verifica se a string 's' pertence ao conjunto.

```
1  int trie[N][M];
2  int node_count;
3  bool stop[N];
4
5  void insert(string word) {
6      int node = 0;
7      for(auto c: word) {
8          if(trie[node][c-'a'] == 0) trie[node][c-'a'] = ++node_count;
9          node = trie[node][c-'a'];
10     }
11     stop[node] = true;
12 }
13
14 bool search(string word) {
15     int node = 0;
16     for(auto c: word) {
17         if(trie[node][c-'a'] == 0) return False;
18         node = trie[node][c-'a'];
19     }
20     return stop[node];
21 }
```

2.0.5 Aho-Corasick

O algoritmo Aho-Corasick é uma estrutura de dados eficiente para encontrar todas as ocorrências de um conjunto de palavras (padrões) em um texto. Ele combina a ideia de uma árvore Trie com a função de falha do algoritmo Knuth-Morris-Pratt (KMP), permitindo a busca de múltiplos padrões simultaneamente, com uma complexidade de tempo linear em relação ao tamanho do texto e ao número total de caracteres nos padrões.

Como funciona O Aho-Corasick constrói um automato finito determinístico (AFD) a partir do conjunto de palavras. O AFD possui as seguintes propriedades:

- Cada nó representa um prefixo de uma das palavras.
- As arestas representam transições entre prefixos com base em um caractere.
- Cada nó terminal representa uma palavra completa.
- A função de sufixo (link) conecta um nó a outro nó que representa o maior sufixo próprio do prefixo correspondente.
- A função de transição (go) calcula o próximo estado no AFD dado um estado atual e um caractere de entrada.

Observação Para otimizar o codigo usamos de lazy programing, ou seja apenas realizamos os calculos quando estes são necessarios.

Lista de funções

Func **insert(string s, int idx)** $O(n)$
Insere a palavra 's' no automato e associa o índice 'idx' a ela.

Func **link(int u)** $O(n)$
Calcula a função de sufixo proprio para o nó 'u'.

Func **go(int u, int c)** $O(n)$
Calcula a função de transição para o nó 'u' com o caractere 'c'

Func **exi(int u)** $O(n)$
Encontra o próximo nó terminal na cadeia de links a partir do nó 'u'.

Func **process(string s)** $O(n)$
Processa o texto 's' e encontra as ocorrencias das palavras.

```
1  struct node {
2      int nxt[mxK]; //Define qual o proximo no dado o caracter
3      int term=0; //Define se corresponde a um no terminal
4      int p; //Define quem e o no pai
5      int pc; //Define com qual caracter viemos do pai
6      int link = -1; //Define qual no da arvore corresponde ao maior sufixo proprio
7      int go[mxK]; //Possui todos as transicoes do automato
8      int exi = -1; //Leva direto para o proximo vertice na cadeia de links que eh terminal
9      int occ = 0; //Guarda quantas ocorrencias desse no do automato no texto
10     vector<int> idx; //Representa os indices de palavras que terminam nesse no
11     node(int _p = 0, int _pc = 0) : p(_p), pc(_pc){
```

```
12          memset(nxt, -1, sizeof(nxt));
13          memset(go, -1, sizeof(go));
14      }
15  };
16
17  vector<node> aca; //Aho-Corasick Automaton
18  int occ[mxN];
19  void insert(string s, int idx) {
20      int i, u = 0;
21      for(auto si: s) {
22          int c = to_i(si);
23          if(aca[u].nxt[c] == -1) {
24              aca[u].nxt[c] = aca.size();
25              aca.emplace_back(u, c);
26          }
27          u = aca[u].nxt[c];
28      }
29      aca[u].term = 1;
30      aca[u].idx.push_back(idx);
31  }
32
33  int go(int u, int c);
34
35  int link(int u) {
36      if(aca[u].link != -1) return aca[u].link;
37      return aca[u].link = (u == 0 || aca[u].p == 0) ? 0 : go(link(aca[u].p), aca[u].pc);
38  }
39
40  int go(int u, int c) {
41      if(aca[u].go[c] != -1) return aca[u].go[c]; // O estado do automato ja existe
42      if(aca[u].nxt[c] != -1) return aca[u].go[c] = aca[u].nxt[c]; // O estado do automato ja ex
43
44      return aca[u].go[c] = u == 0 ? 0 : go(link(u), c);
45  }
46
47  int exi(int u) {
48
49      int v = link(u);
50      if(aca[u].exi != -1) return aca[u].exi;
51      return aca[u].exi = (v == 0 || aca[v].term) ? v : exi(v);
52  }
53
54  void process(string s) {
55      int i, u = 0;
56      for(i = 0; s[i]; i++) {
57          int c = to_i(s[i]);
58          u = go(u, c); //Verifica o proximo estado no automato
59          for (int v = u; v; v = exi(v)) {
60              aca[v].occ++;
61          }
62      }
63      for (u = 0; u < (int)aca.size(); u++)
64          for (auto& idx: aca[u].idx)
65              occ[idx] += aca[u].occ;
```

```
66 }
```

2.0.6 Suffix Array and LCP

Essa estrutura é utilizada para armazenar todos os sufixos de uma string, ordenados em ordem lexicográfica. O array LCP (Longest Common Prefix) guarda o comprimento do maior prefixo comum entre cada sufixo e seu antecessor no array de sufixos.

Essa estrutura pode ser implementada por um array de tamanho N, referente a string analisada, para cada posição ficará indicado, a posição do inicio do sufixo no array de sufixos.

Lista de funções

Func **build_sa(string s)** $O(nlogn)$
Constroi o array de sufixos para a string 's'.

Func **build_lcp(string s)** $O(n)$
Constroi o array LCP para a string 's'.

Func **countingSort(int k)** $O(n)$
Função auxiliar para realizar o radix sort.

```
1  ll lcp[mxN]; //Armazena o lcp, maior prefixo em comum entre a string e seu antecedente imediat
2  ll ra[mxN], tmpra[mxN]; // rank array e temporario
3  ll sa[mxN], tmpsa[mxN]; // suffixo array e temporario
4  ll phi[mxN]; // Indica que eh o anterior no suffix array
5  ll c[mxN]; // Para o counting/radix sort
6  ll plcp[mxN]; // Constroi o lcp permutado
7  ll n; // Tamanho da string concatenada
8  ll ans[mxN];
9
10 void countingSort(int k) {
11     int mxi = max(300LL, n);
12     memset(c, 0, sizeof(c));
13     for (int i = 0; i < n; i++)
14         c[_ra(i+k)]++;
15     for (int i = 0, sum = 0; i < mxi; i++) {
16         int t = c[i]; c[i] = sum; sum += t;
17     }
18     for (int i = 0; i < n; i++)
19         tmpsa[c[_ra(sa[i]+k)]++] = sa[i];
20     for (int i = 0; i < n; i++)
21         sa[i] = tmpsa[i];
22 }
23
24 void build_lcp(string t) {
25
26     phi[sa[0]] = -1;
27     for (int i = 1; i < n; i++) phi[sa[i]] = sa[i-1];
28     for (int i = 0, l = 0; i < n; i++) {
```

```
29          if(phi[i] == -1) {plcp[i] = 0; continue;}
30          while(t[i+l] == t[phi[i] + l]) l++;
31          plcp[i] = l--;
32          l = max(0,l);
33      }
34      for (int i = 0; i < n; i++) lcp[i] = plcp[sa[i]];
35  }
36
37  void build_sa(string t) {
38
39      for (int i = 0; i < n; i++) { sa[i] = i; ra[i] = t[i];}
40      for (int k = 1; k < n; k <<= 1) {
41          countingSort(k);
42          countingSort(0);
43          int r; // re-ranking
44          tmpra[sa[0]] = r = 0;
45          for (int i = 1; i < n; i++) {
46              if(ra[sa[i]] == ra[sa[i-1]] && _ra(sa[i] + k) == _ra(sa[i-1] + k))
47                  tmpra[sa[i]] = r;
48              else
49                  tmpra[sa[i]] = ++r;
50          }
51          for (int i = 0; i < n; i++) ra[i] = tmpra[i];
52          if (ra[sa[n-1]] == n-1) break;
53      }
54
55  }
```

Capítulo 3

Programação dinâmica

3.0.1 Max consecutive sum

Encontra a maior soma de elementos consecutivos em um vetor. Por padrão, aceita resultados negativos, mas isso pode ser alterado facilmente na condição **if**. Por exemplo, para aceitar apenas resultados maiores que 0, é possível fazer, nesse ponto, $sum = max(a[i], 0)$.

Lista de funções

Func **max_consecutive_sum(vector a, int N)** $O(n)$
Retorna a maior soma de elementos consecutivos.

```
1
2           int max_consecutive_sum(int a[], int N) {
3               int sum = 0, max_sum = INT_MIN;
4               for(int i = 0; i < N; i++) {
5                   if(sum < 0) sum = a[i];
6                   else sum += a[i];
7                   if(max_sum < sum) max_sum = sum;
8               }
9               return max_sum;
10          }
```

3.0.2 Soma maxima sem adjacente

Dado um array [], retorne a *soma máxima sem selecionar números adjacentes*.

Lista de funções

FUNC **FindMaxSum(vector arr, int n)** $O(n)$
Retorna a soma máxima sem selecionar números adjacentes.

```
1
2         int FindMaxSum(vector<int> arr, int n){
3             int incl = arr[0];
4             int excl = 0;
5             int excl_new;
6             int i;
7
8             for (i = 1; i < n; i++){
9                 /* current max excluding i */
10                excl_new = (incl > excl) ? incl : excl;
11
12                /* current max including i */
13                incl = excl + arr[i];
14                excl = excl_new;
15            }
16
17            /* return max of incl and excl */
18            return ((incl > excl) ? incl : excl);
19        }
```

3.0.3 Fibonacci

Dado um número n, imprime o n-ésimo número de Fibonacci.

LISTA DE FUNÇÕES

FUNC **fib(int n)** $O(n)$
Retorna o n-ésimo número de Fibonacci.

```
1
2         int fib(int n){
3             int a = 0, b = 1, c, i;
4             if( n == 0)
5                 return a;
6             for(i = 2; i <= n; i++){
7                c = a + b;
8                a = b;
9                b = c;
10            }
11            return b;
12        }
```

3.0.4 Coin change problem

Dada uma matriz inteira de moedas[ ] de tamanho N representando diferentes tipos de moeda e uma soma inteira, a tarefa é encontrar o número de maneiras de fazer a soma usando diferentes combinações de moedas[].

Lista de funções

Func **count(vector coins, int n, int sum)** $O(N * Sum)$
Retorna o número de maneiras de fazer a soma usando diferentes combinações de moedas.

```
1
2          int count(int coins[], int n, int sum){
3              int table[sum + 1];
4
5              memset(table, 0, sizeof(table));
6
7              table[0] = 1;
8
9              for (int i = 0; i < n; i++)
10                 for (int j = coins[i]; j <= sum; j++)
11                     table[j] += table[j - coins[i]];
12             return table[sum];
13         }
```

3.0.5 Minimizing Coins

Dada uma matriz inteira de moedas[ ] de tamanho N representando diferentes tipos de moeda e uma soma inteira. Sua tarefa é encontrar o menor número de moedas capaz de produzir essa soma.

Lista de funções

Func **minCoins(vector¡ll¿ coins, int m, int V)** $O(N * Sum)$
Retorna o menor número de moedas capas de fazer a soma.

```
1  // m is size of coins array (number of different coins)
2  int minCoins(vector<ll> coins, int m, int V){
3      // table[i] will be storing the minimum number of coins
4      // required for i value.  So table[V] will have result
5      int table[V + 1];
6
7      // Base case (If given value V is 0)
8      table[0] = 0;
9
10     // Initialize all table values as Infinite
11     for (int i = 1; i <= V; i++)
12         table[i] = INT_MAX;
13
14     // Compute minimum coins required for all
15     // values from 1 to V
```

```
16      for (int i = 1; i <= V; i++) {
17          // Go through all coins smaller than i
18          for (int j = 0; j < m; j++)
19              if (coins[j] <= i) {
20                  int sub_res = table[i - coins[j]];
21                  if (sub_res != INT_MAX
22                      && sub_res + 1 < table[i])
23                      table[i] = sub_res + 1;
24              }
25      }
26
27      if (table[V] == INT_MAX)
28          return -1;
29
30      return table[V];
31  }
```

3.0.6 Subset sum

Dado um conjunto de inteiros não negativos e uma soma de valores, a tarefa é verificar se existe um subconjunto do conjunto dado cuja soma seja igual à soma dada.

Lista de funções

FUNC **subsetSum(vector a, int n, int sum)** $O(N * Sum)$
Retorna se a soma existe.

```
1
2       //init table with -1
3       //int tab[2000][2000];
4       vector<vector<int>> tab(2000, vector<int>(2000, -1));
5
6       int subsetSum(int a[], int n, int sum){
7           if (sum == 0)
8               return 1;
9
10          if (n <= 0)
11              return 0;
12
13          if (tab[n - 1][sum] != -1)
14              return tab[n - 1][sum];
15
16          if (a[n - 1] > sum)
17              return tab[n - 1][sum] = subsetSum(a, n - 1, sum);
18          else{
19              return tab[n - 1][sum] = subsetSum(a, n - 1, sum) ||
20                          subsetSum(a, n - 1, sum - a[n - 1]);
21          }
22      }
```

3.0.7 Subset sum

Dado um conjunto de inteiros não negativos e uma soma de valores, a tarefa é retornaar o número de subsets cuja soma seja igual à soma dada.

Lista de funções

Func **countSubsetSum(vector a, int N, int k)** $O(N * Sum)$
Retorna a quantidade de subset com a soma igual a k.

```
1
2          int countSubsetSum(int *a, int N, int k) {
3              int l[k+1],c[k+1];
4
5              memset(l,0,sizeof(l));
6              memset(c,0,sizeof(c));
7              l[0] = c[0] = 1;
8
9              for(int i = 0; i < N; i++) {
10                 for(int j = 0; j <= k; j++) {
11                     if(j < a[i]) continue;
12                     c[j] += l[j-a[i]];
13                 }
14                 memcpy(l,c,sizeof(c));
15             }
16
17             return c[k];
18         }
```

3.0.8 Distância de edição (Levenshtein)

Este algoritmo é usado para calcular a distância de edição entre duas strings. A distância de edição é o número mínimo de operações necessárias para transformar uma string em outra. As operações permitidas são inserção, exclusão e substituição de caracteres.

Lista de funções

Func **levenshteinTwoMatrixRows(string str1, string str2)** $O(n * m)$
retorna a distancia de edição.

```
1  int levenshteinTwoMatrixRows(const string& str1, const string& str2){
2  int m = str1.length();
```

```
3  int n = str2.length();
4
5  vector<int> prevRow(n + 1, 0);
6  vector<int> currRow(n + 1, 0);
7
8  for (int j = 0; j <= n; j++) prevRow[j] = j;
9
10
11 for (int i = 1; i <= m; i++){
12     currRow[0] = i;
13
14     for (int j = 1; j <= n; j++){
15         if (str1[i - 1] == str2[j - 1]){
16             currRow[j] = prevRow[j - 1];
17         }
18         else{
19             currRow[j] = 1
20                         + min(
21
22                             // Insert
23                             currRow[j - 1],
24                             min(
25
26                                 // Remove
27                                 prevRow[j],
28
29                                 // Replace
30                                 prevRow[j - 1]));
31         }
32     }
33
34     prevRow = currRow;
35 }
36
37 return currRow[n];
```

3.0.9 Longest Common Subsequence (LCS)

Dadas duas strings, S1 e S2 , a tarefa é encontrar o comprimento da subsequência mais longa presente em ambas as strings.

Lista de funções

FUNC **lcs(string X, string Y, int m, int n)** $O(n * m)$
retorna o comprimento subsequência mais longa presente em ambas as strings.

```
1
2          int lcs(string X, string Y, int m, int n){
3              int L[m + 1][n + 1];
```

```
4
5               for (int i = 0; i <= m; i++) {
6                   for (int j = 0; j <= n; j++) {
7                       if (i == 0 || j == 0)
8                           L[i][j] = 0;
9
10                      else if (X[i - 1] == Y[j - 1])
11                          L[i][j] = L[i - 1][j - 1] + 1;
12
13                      else
14                          L[i][j] = max(L[i - 1][j], L[i][j - 1]);
15                  }
16              }
17
18              return L[m][n];
19          }
```

3.0.10 Longest Increasing Subsequence (LIS)

Dada uma matriz arr[] de tamanho N , a tarefa é encontrar o comprimento da Subsequência Crescente Mais Longa (LIS), ou seja, a subsequência mais longa possível na qual os elementos da subsequência são classificados em ordem crescente.

Lista de funções

Func **lis(int arr[], int n)** $O(n^2)$
retorna o tamanho da maior subsequência crescente.

Func **lengthOfLIS(vector nums)** $O(nlogn)$
retorna o tamanho da maior subsequência crescente.

```
1
2           int lis(int arr[], int n){
3               int lis[n];
4
5               lis[0] = 1;
6
7               for (int i = 1; i < n; i++) {
8                   lis[i] = 1;
9                   for (int j = 0; j < i; j++)
10                      if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
11                          lis[i] = lis[j] + 1;
12              }
13
14              // Return maximum value in lis[]
15              return *max_element(lis, lis + n);
16          }
17
18          int lengthOfLIS(vector<int>& nums){
```

```
19
20              // Binary search approach
21              int n = nums.size();
22              vector<int> ans;
23
24              // Initialize the answer vector with the
25              // first element of nums
26              ans.push_back(nums[0]);
27
28              for (int i = 1; i < n; i++) {
29                  if (nums[i] > ans.back()) {
30
31                      // If the current number is greater
32                      // than the last element of the answer
33                      // vector, it means we have found a
34                      // longer increasing subsequence.
35                      // Hence, we append the current number
36                      // to the answer vector.
37                      ans.push_back(nums[i]);
38                  }
39                  else {
40
41                      // If the current number is not
42                      // greater than the last element of
43                      // the answer vector, we perform
44                      // a binary search to find the smallest
45                      // element in the answer vector that
46                      // is greater than or equal to the
47                      // current number.
48
49                      // The lower_bound function returns
50                      // an iterator pointing to the first
51                      // element that is not less than
52                      // the current number.
53                      int low = lower_bound(ans.begin(), ans.end(),
54                                            nums[i])
55                                - ans.begin();
56
57                      // We update the element at the
58                      // found position with the current number.
59                      // By doing this, we are maintaining
60                      // a sorted order in the answer vector.
61                      ans[low] = nums[i];
62                  }
63              }
64
65              // The length of the answer vector
66              // represents the length of the
67              // longest increasing subsequence.
68              return ans.size();
69          }
```

3.0.11 Mochila binaria

Recebemos N itens onde cada item tem algum peso e lucro associado a ele. Também recebemos uma sacola com capacidade W, [isto é, a sacola pode conter no máximo o peso W]. O objetivo é colocar os itens na sacola de forma que a soma dos lucros associados a eles seja a máxima possível.

Lista de funções

Func **knapSack(int W, int wt[], int val[]** $O(n^2)$
retorna o lucro maximo.

```
1  int knapSack(int W, int wt[], int val[], int n){
2      int dp[W + 1];
3      memset(dp, 0, sizeof(dp));
4
5      for (int i = 1; i < n + 1; i++) {
6          for (int w = W; w >= 0; w--) {
7
8              if (wt[i - 1] <= w)
9
10                 dp[w] = max(dp[w],
11                             dp[w - wt[i - 1]] + val[i - 1]);
12         }
13     }
14     return dp[W];
15 }
```

3.0.12 Mochila binaria com itens infinitos

Dado um peso de mochila W e um conjunto de n itens com certo valor val_i e peso wt_i , precisamos calcular a quantidade máxima que poderia compor exatamente essa quantidade. Diferença do problema classico é que aqui podemos usar um número ilimitado de instâncias de um item.

Lista de funções

Func **unboundedKnapsack(int W, int n, int val[], int wt[])** $O(W * N)$
retorna o lucro maximo.

```
1  int unboundedKnapsack(int W, int n, int val[], int wt[]){
2      // dp[i] is going to store maximum value
3      // with knapsack capacity i.
4      int dp[W+1];
5      memset(dp, 0, sizeof dp);
6
7      // Fill dp[] using above recursive formula
8      for (int i=0; i<=W; i++)
9        for (int j=0; j<n; j++)
10          if (wt[j] <= i)
```

```
11              dp[i] = max(dp[i], dp[i-wt[j]] + val[j]);
12
13      return dp[W];
14  }
```

3.0.13 Square submatrix of 1s

Dada uma matriz binária, descubra a submatriz quadrada de tamanho máximo com todos os 1s.

Lista de funções

FUNC **printMaxSubSquare(bool M[R][C])** $O(n * m)$
printa a maior matriz de 1.

```
1
2           void printMaxSubSquare(bool M[R][C])
3           {
4               int S[2][C], Max = 0;
5
6               // set all elements of S to 0 first
7               memset(S, 0, sizeof(S));
8
9               // Construct the entries
10              for (int i = 0; i < R; i++)
11                  for (int j = 0; j < C; j++) {
12
13                      // Compute the entry at the current position
14                      int Entrie = M[i][j];
15                      if (Entrie)
16                          if (j)
17                              Entrie
18                                  = 1
19                                    + min(S[1][j - 1],
20                                          min(S[0][j - 1], S[1][j]));
21
22                      // Save the last entry and add the new one
23                      S[0][j] = S[1][j];
24                      S[1][j] = Entrie;
25
26                      // Keep track of the max square length
27                      Max = max(Max, Entrie);
28                  }
29
30              // Print the square
31              cout << "Maximum␣size␣sub-matrix␣is:␣\n";
32              for (int i = 0; i < Max; i++, cout << '\n')
33                  for (int j = 0; j < Max; j++)
34                      cout << "1␣";
35          }
```

3.0.14 submatriz de soma maxima

Dada uma matriz 2D, encontre a submatriz de soma máxima nela.

Lista de funções

Func **findMaxSum(int M[][]) $O(c * c * r)$**
Retorna a a submatriz de soma maxima.

Func **kadane(int* arr, int* start, int* finish, int n) $O(n)$**
encontrar a soma do subarray contíguo com a maior soma.

```
1  #define ROW 4
2  #define COL 5
3
4  int kadane(int* arr, int* start, int* finish, int n)
5  {
6      int sum = 0, maxSum = INT_MIN, i;
7
8      *finish = -1;
9
10     int local_start = 0;
11
12     for (i = 0; i < n; ++i)
13     {
14         sum += arr[i];
15         if (sum < 0)
16         {
17             sum = 0;
18             local_start = i + 1;
19         }
20         else if (sum > maxSum)
21         {
22             maxSum = sum;
23             *start = local_start;
24             *finish = i;
25         }
26     }
27
28     if (*finish != -1)
29         return maxSum;
30
31     maxSum = arr[0];
32     *start = *finish = 0;
33
34     for (i = 1; i < n; i++)
35     {
36         if (arr[i] > maxSum)
```

```
37          {
38              maxSum = arr[i];
39              *start = *finish = i;
40          }
41      }
42      return maxSum;
43  }
44
45  void findMaxSum(int M[][COL])
46  {
47      // Variables to store the final output
48      int maxSum = INT_MIN,
49                   finalLeft,
50                   finalRight,
51                   finalTop,
52                   finalBottom;
53
54      int left, right, i;
55      int temp[ROW], sum, start, finish;
56
57      // Set the left column
58      for (left = 0; left < COL; ++left) {
59          // Initialize all elements of temp as 0
60          memset(temp, 0, sizeof(temp));
61
62          for (right = left; right < COL; ++right) {
63
64              for (i = 0; i < ROW; ++i)
65                  temp[i] += M[i][right];
66
67              sum = kadane(temp, &start, &finish, ROW);
68
69              if (sum > maxSum) {
70                  maxSum = sum;
71                  finalLeft = left;
72                  finalRight = right;
73                  finalTop = start;
74                  finalBottom = finish;
75              }
76          }
77      }
78
79      // Print final values
80      cout << "(Top, Left) ("
81           << finalTop << ", "
82           << finalLeft
83           << ")" << endl;
84      cout << "(Bottom, Right) ("
85           << finalBottom << ", "
86           << finalRight << ")" << endl;
87      cout << "Max sum is: " << maxSum << endl;
88  }
```

3.0.15 Max histogram rectangle

O objetivo é encontrar a maior área retangular formada pelas colunas de um histograma.

Lista de funções

Func **maxHist(vector line)** $O(n)$
retorna a maior área.

```
1  int maxHist(const vector<int> &line){
2    stack<int> result;
3    uint max_area = 0;
4    uint area = 0;
5    int top;
6
7    auto popmult = [&](int i){
8      top = line[result.top()];
9      result.pop();
10     area = top * i;
11     if(!result.empty())
12       area = top * (i - result.top() - 1);
13     max_area = max(area, max_area);
14   };
15
16   uint i = 0;
17   while(i < line.size()){
18     if(result.empty() || line[result.top()] <= line[i])
19       result.push(i++);
20     else popmult(i);
21   }
22
23   while(!result.empty()) popmult(i);
24
25   return max_area;
26 }
```

3.0.16 Minimum insertions to form a palindrome

Dada a string str , a tarefa é encontrar o número mínimo de caracteres a serem inseridos para convertê-la em um palíndromo.

Lista de funções

Func **findMinInsertionsDP(char str[], int n)** $O(n^2)$
Retorna o menor número de inserções.

```
int findMinInsertionsDP(char str[], int n){
    // Create a table of size n*n. table[i][j]
    // will store minimum number of insertions
    // needed to convert str[i..j] to a palindrome.
    int table[n][n], l, h, gap;

    // Initialize all table entries as 0
    memset(table, 0, sizeof(table));

    // Fill the table
    for (gap = 1; gap < n; ++gap)
        for (l = 0, h = gap; h < n; ++l, ++h)
            table[l][h] = (str[l] == str[h])?
                          table[l + 1][h - 1] :
                          (min(table[l][h - 1],
                               table[l + 1][h]) + 1);

    // Return minimum number of insertions
    // for str[0..n-1]
    return table[0][n - 1];
}
```

3.0.17 Weight Job scheduling

Dado N trabalhos, onde cada trabalho é representado pelos seguintes três elementos: Tempo de Início, Tempo de Término, Lucro ou Valor Associado (¿= 0) Encontre o subconjunto de trabalhos que maximize o lucro de forma que nenhum dos trabalhos no subconjunto se sobreponha.

Lista de funções

Func **findMaxProfit(Job arr[], int n)** $O(n^2)$
sretorna o maior lucro.

```

        struct Job {
            int start, finish, profit;
        };

        // A utility function that is used for sorting events
        // according to finish time
        bool jobComparator(Job s1, Job s2)
        {
            return (s1.finish < s2.finish);
        }

        // Find the latest job (in sorted array) that doesn't
        // conflict with the job[i]
        int latestNonConflict(Job arr[], int i)
        {
            for (int j = i - 1; j >= 0; j--) {
```

```
18                  if (arr[j].finish <= arr[i].start)
19                      return j;
20              }
21              return -1;
22          }
23
24          // The main function that returns the maximum possible
25          // profit from given array of jobs
26          int findMaxProfit(Job arr[], int n)
27          {
28              // Sort jobs according to finish time
29              sort(arr, arr + n, jobComparator);
30
31              // Create an array to store solutions of subproblems.
32              // table[i] stores the profit for jobs till arr[i]
33              // (including arr[i])
34              int* table = new int[n];
35              table[0] = arr[0].profit;
36
37              // Fill entries in M[] using recursive property
38              for (int i = 1; i < n; i++) {
39                  // Find profit including the current job
40                  int inclProf = arr[i].profit;
41                  int l = latestNonConflict(arr, i);
42                  if (l != -1)
43                      inclProf += table[l];
44
45                  // Store maximum of including and excluding
46                  table[i] = max(inclProf, table[i - 1]);
47              }
48
49              // Store result and free dynamic memory allocated for
50              // table[]
51              int result = table[n - 1];
52              delete[] table;
53
54              return result;
55          }
```

3.0.18 Particionamento palindrômico

Dada uma string, será considerado um particonamento palindrômico uma divisão que gere apenas substrings palindrômicas. O objetivo é, então, encontrar o número mínimo de cortes necessário para realizar o particionamento palindrômico de uma string.

Lista de funções

FUNC **minCut(string s)** $O(n^2)$
indica o número mínimo de cortes para o particionamento em palíndromo.

FUNC **generatePalindrome(string s, vector¡vector¡bool¿¿ pal)** $O(n^2)$
gera todas as substrings palindrômicas possíveis.

```
1
2         bool generatePalindrome(string& s, vector<vector<bool>>& pal) {
3         int n = s.size();
4         for (int i = 0; i < n; i++) pal[i][i] = true;
5         for (int len = 2; len <= n; len++) {
6             for (int i = 0; i <= n - len; i++) {
7                 int j = i + len - 1;
8                 if (s[i] == s[j] && (len == 2 || pal[i + 1][j - 1]))
9                     pal[i][j] = true;
10            }
11        }
12    }
13
14    int minCut(string s) {
15        if (s.empty()) return 0;
16        int n = s.size();
17        vector<vector<bool>> pal(n, vector<bool>(n, false));
18        generate_palindrome(s, pal);
19        vector<int> minCutDp(n, INT_MAX);
20        minCutDp[0] = 0;
21
22        for (int i = 1; i < n; i++) {
23            if (pal[0][i]) minCutDp[i] = 0;
24            else {
25                for (int j = i; j >= 1; j--)
26                    if (pal[j][i]) {
27                        if (minCutDp[j - 1] + 1 < minCutDp[i])
28                            minCutDp[i] = minCutDp[j - 1] + 1;
29                    }
30            }
31        }
32
33        return minCutDp[n - 1];
34    }
```

3.0.19 Cutting a rod

Dada uma haste de comprimento n polegadas e uma matriz de preços que inclua os preços de todas as peças de tamanho menor que n. Determine o valor máximo obtido cortando a haste e vendendo os pedaços.

LISTA DE FUNÇÕES

FUNC **cutRod(int price[], int n)** $O(n^2)$
retorna o máximo obtido cortando a haste e vendendo os pedaços.

```
1  int cutRod(int price[], int n){
2     int val[n+1];
3     val[0] = 0;
4     int i, j;
5
6     for (i = 1; i<=n; i++)
7     {
8         int max_val = INT_MIN;
9         for (j = 0; j < i; j++)
10          max_val = max(max_val, price[j] + val[i-j-1]);
11        val[i] = max_val;
12    }
13
14    return val[n];
15 }
```

3.0.20 Min jumps to end

Dado um array arr[] onde cada elemento representa o número máximo de etapas que podem ser feitas a partir desse índice. A tarefa é encontrar o número mínimo de saltos para chegar ao final do array a partir do índice 0 . Se o fim não for alcançável, retorne -1.

LISTA DE FUNÇÕES

FUNC **minJumps(int arr[], int n)** $O(n^2)$
printa o número minimo de saltos.

```
1  int minJumps(int arr[], int n){
2      if (n <= 1) return 0;
3      if (arr[0] == 0) return -1;
4
5      int maxReach = arr[0];
6      int steps = arr[0];
7      int jumps = 1;
8      for (int i = 1; i < n; i++) {
9          if (i == n - 1)
10             return jumps;
11         maxReach = max(maxReach,
12         i + arr[i]);
13         steps--;
14         if (steps == 0) {
15             jumps++;
16             if (i >= maxReach)
17                 return -1;
```

```
18                steps = maxReach - i;
19            }
20        }
21        return -1;
22    }
```

Capítulo 4

Grafos

4.1 Caminho mínimo

Os seguintes algoritmos determinam a menor distância entre qualquer par de vértice do grafo. Resumidamente, as especificidades de cada um podem ser dadas como:

- **Dijkstra :** Encontra o menor caminho de um vértice origem para todos os outros. Tem ordem de complexidade $O(V \log E)$ e não produz resultados corretos na presença de arestas com peso negativo.
- **Bellman Ford :** Também encontra o menor caminho de um vértice para todos os outros. É tolerante à presença de arestas de custo negativo e consegue identificar a existência de ciclos negativos, mas possui complexidade $O(VE)$.
- **Floyd-Wharsall :** Encontra o menor caminho entre todos os pares de vértices do grafo. Também funciona na presença de arestas de custo negativo e pode ser usado para identificar ciclos negativos. Sua complexidade é $O(V^3)$.

4.1.1 Dijkstra

```
1  int n;
2  vector<vector<pair<int,long long>>> adj;
3  vector<bool> done;
4  vector<long long> dist;
5  vector<int> par;
6
7  void dijkstra(int src)
8  {
9      priority_queue<pair<long long, int>> pq;
10
11     dist.assign(n,LLONG_MAX);
12     done.assign(n,false);
13     par.assign(n,-1);
14     dist[src] = 0;
15     pq.push({0,src});
```

```
16
17      while(!pq.empty()) {
18          int u = pq.top().second;
19          pq.pop();
20          if(done[u]) continue;
21          done[u] = true;
22          for(auto edge : adj[u]) {
23               int v = edge.first;
24               long long w = edge.second;
25               if(dist[v] == LLONG_MAX || dist[u]+w < dist[v]) {
26                  dist[v] = dist[u]+w;
27                  pq.push({-dist[v],v});
28                  par[v] = u;
29               }
30          }
31      }
32  }
```

4.1.2 Bellman-Ford

```
1  int n;
2  vector<array<int,3>> edges;
3  vector<long long> dist;
4
5  bool bellman(int src)
6  {
7      bool changed;
8
9      dist.assign(n,LLONG_MAX);
10     dist[src] = 0;
11
12     for(int i = 0; i < n; ++i) {
13         changed = 0;
14         for(auto e : edges) {
15             if(dist[e[0]] == LLONG_MAX) continue;
16             if(dist[e[1]] > dist[e[0]] + e[2]) {
17                 dist[e[1]] = dist[e[0]] + e[2];
18                 changed = 1;
19             }
20         }
21     }
22
23     return !changed;
24 }
```

4.1.3 Floyd-Wharsall

```
1  int n;
2  vector<vector<long long>> cost;
3
4  void floyd()
```

```
5  {
6      for(int i = 0; i < n; ++i) {
7          for(int j = 0; j < n; ++j) {
8              for(int k = 0; k < n; ++k) {
9                  if(cost[i][k] == LLONG_MAX || cost[k][j] == LLONG_MAX) continue;
10                 cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j]);
11             }
12         }
13     }
14 }
```

4.2 Ordenação topológica

A ordenação topológica de um grafo é uma permutação dos vértices que respeita a ordem de depêndencia imposta pelas arestas. Sendo ord_v o índice de um vértice v na ordenação, $\forall (u, v) \in E,\ ord_u < ord_v$.

```
1  int n;
2  vector<vector<int>> adj;
3  vector<bool> visited;
4  vector<int> order;
5
6  void dfs(int v) {
7      visited[v] = true;
8      for (int u : adj[v]) {
9          if(!visited[u]) dfs(u);
10     }
11     order.push_back(v);
12 }
13
14 void topological_sort() {
15     visited.assign(n, false);
16     order.clear();
17     for (int i = 0; i < n; ++i) {
18         if (!visited[i]) dfs(i);
19     }
20     reverse(order.begin(), order.end());
21 }
```

4.3 Outros

4.3.1 Heurística de Warnsdorff

Essa heurística é primeiramente uma otimização para a resolução do problema de **passeio do cavalo**. Nesse problema, o cavalo é colocado no tabuleiro vazio e, seguindo as regras do jogo, precisa passar por todas as casas exatamente uma vez em movimentos consecutivos.

A solução possui complexidade exponencial, mas é possível otimizar sua execução usando a seguinte estratégia: cada casa possui um **grau** associado, e, a cada movimento, tenta-se achar uma solução a partir da casa vizinha (atingível por um movimento válido do cavalo) de menor grau.

O código dessa heurística é mostrado na resolução 8.3.9.6.

Capítulo 5

Árvores

5.1 Segment Tree

Essa estrutura é construída para fazer consultas em intervalos de listas de forma eficiente. Dessa forma, tarefas realizadas trivialmente em $O(n)$ podem ser desenvolvidas em $O(logn)$. Para isso, é usada uma árvore: cada nó representa um intervalo da lista, possui a propriedade almejada pré-computada e aponta para outros dois nós, cada um correspondente a uma metade do intervalo.

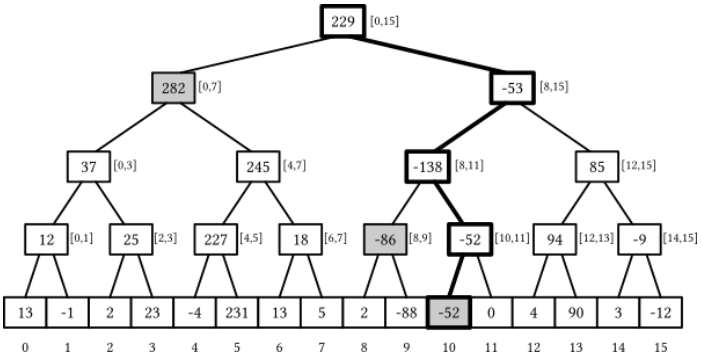


Figura 5.1: Exemplo de segtree para um vetor de números

É claro que a eficiência real da estrutura depende da implementação. Algumas implementações usam recursão para manipular a árvore, e isso pode aumentar o valor da função de complexidade. A implementação abaixo simula a árvore a partir de um array, que precisa ter tamanho $2n$.

A estrutura deve ser iniciada com um valor padrão (o elemento neutro da operação que será realizada), e basta chamar a função *segtree_update* com cada posição e valor do vetor base para construir a árvore.

Implementação

```
1  int n;
2  int segtree[2*MAXN];
3
```

```
4  void segtree_update(int k, int x)
5  {
6      k += n;
7      segtree[k] = x;
8      while(k >>= 1) {
9          segtree[k] = segtree[2*k]+segtree[2*k+1]);
10     }
11 }
12
13 int segtree_query(int l, int r)
14 {
15     l += n, r += n;
16     int ans = 0;
17     while(l <= r) {
18         if(l&1) ans += segtree[l++]);
19         if(~r&1) ans += segtree[r--]);
20         l >>= 1, r >>= 1;
21     }
22     return ans;
23 }
24
25 void init()
26 {
27     memset(segtree,0,sizeof(segtree));
28 }
```

5.2 Heavy Light Decomposition

Essa técnica é voltada para consultar eficientemente propriedades de caminhos de uma árvore. Mais especificamente, busca responder perguntas da forma *entre os vértices a e b, qual o valor da propriedade x.*

A base do algoritmo para HLD é a divisão da árvore em segmentos. As arestas que formam cada seguimento são pesadas (*heavy*), enquanto as outras, que ligam diferentes seguimentos, são leves (*light*).

Idealmente, cada seguimento é representado por uma SegmentTree, apesar de que é possível utilizar apenas uma estrutura desse tipo desde que vértices de um mesmo seguimento sejam colocados em posições adjacentes. Para realizar uma consulta, então, é possível fazer computações de um segumento usando a SegmentTree e utilizar o conceito de *lowest common ancestor (LCA)* para encontrar o ponto de encontro entre os seguimentos de dois vértices.

A complexidade de atualizações e consultas podem variar de $O(logn)$ a $O(log^2 n)$, dependendo da modelagem do problema. Apesar disso, a implementação, principalmente da SegmentTree pode gerar grande impacto na função real de complexidade.

O código abaixo exemplifica a busca pelo maior elemento em determinado seguimento. Essa especificidade é determinada apenas pelas funções relacionadas à SegmentTree, que pode ser modificada facilmente. As demais funções são genéricas.

Implementação

```
1  int n;
2  vector<int> adj[MAXN];
3  int parent[MAXN], depth[MAXN], heavy[MAXN];
4  int head[MAXN], pos[MAXN];
```

```
5  int v[MAXN], segtree[2*MAXN];
6  int cur_pos;
7
8  void segtree_update(int k, int x)
9  {
10     k += n;
11     segtree[k] = x;
12     while(k >>= 1) {
13         segtree[k] = max(segtree[2*k],segtree[2*k+1]);
14     }
15 }
16
17 int segtree_query(int l, int r)
18 {
19     l += n, r += n;
20     int ans = 0;
21     while(l <= r) {
22         if(l&1) ans = max(ans,segtree[l++]);
23         if(~r&1) ans = max(ans,segtree[r--]);
24         l >>= 1, r >>= 1;
25     }
26     return ans;
27 }
28
29 int dfs(int v) {
30     int size = 1;
31     int max_c_size = 0;
32     for (int c : adj[v]) {
33         if(c == parent[v]) continue;
34         parent[c] = v;
35         depth[c] = depth[v] + 1;
36         int c_size = dfs(c);
37         size += c_size;
38         if (c_size > max_c_size) {
39             max_c_size = c_size;
40             heavy[v] = c;
41         }
42     }
43     return size;
44 }
45
46 void decompose(int i, int h) {
47     head[i] = h;
48     pos[i] = cur_pos++;
49     segtree_update(pos[i],v[i]);
50     if(heavy[i] != -1) decompose(heavy[i], h);
51     for(int c : adj[i]) {
52         if(c == parent[i] || c == heavy[i]) continue;
53         decompose(c, c);
54     }
55 }
56
57 int query(int a, int b) {
58     int res = 0;
59     while(head[a] != head[b]) {
```

```
60          if (depth[head[a]] > depth[head[b]]) swap(a, b);
61          int cur_heavy_path_max = segtree_query(pos[head[b]], pos[b]);
62          res = max(res, cur_heavy_path_max);
63          b = parent[head[b]];
64      }
65      if (depth[a] > depth[b]) swap(a, b);
66      int last_heavy_path_max = segtree_query(pos[a], pos[b]);
67      res = max(res, last_heavy_path_max);
68      return res;
69  }
70
71  void init() {
72      memset(heavy,UCHAR_MAX,sizeof(heavy));
73      memset(segtree,0,sizeof(segtree));
74      cur_pos = 0;
75      dfs(0);
76      decompose(0,0);
77  }
```

5.3 Fenwick Tree - BIT

É uma estrutura que proporciona, eficientemente, atualização de elementos e o cálculo de **operações com prefixos** ao longo de um vetor de itens. Suas aplicações são parecidas com aplicações de Segment Trees, mas sua implementação

Implementação

O código abaixo implementa uma BIT para a soma dos elementos.

5.4 MergeSort Tree

É uma estrutura usada para responder consultas como ”quantos elementos distintos estão presentes no intervalo $[a, b]$”. Basicamente, é uma Segment Tree que armazena conjuntos ordenados em cada nó e realiza a operação de união desses conjuntos.

A complexidade da estrutura pode ser alta: a operação de união de dois conjuntos de tamanhos n e m tem complexidade $min(n, m)log(max(n, m))$.

Implementação

```
1  int n;
2  set<int> tree[2*MAXN];
3
4  set<int> merge(set<int> a,
5                 set<int> b)
6  {
7      if(a.size() < b.size()) swap(a,b);
8      for(int v : b) a.insert(v);
```

```
 9      return a;
10  }
11
12  void update(int k, int x)
13  {
14      k += n;
15      tree[k] = {x};
16      while(k >>= 1) {
17          tree[k] = merge(tree[2*k], tree[2*k+1]);
18      }
19  }
20
21  int query(int l, int r)
22  {
23      set<int> ans;
24      l += n, r += n;
25      while(l <= r) {
26          if(l&1) ans = merge(ans, tree[l++]);
27          if(~r&1) ans = merge(ans, tree[r--]);
28          l >>= 1, r >>= 1;
29      }
30      return ans.size();
31  }
```

Capítulo 6

Matemática

6.1 Números primos

Um número é considerado **primo** se, e somente se, seus únicos divisores forem ele mesmo e 1. Todo número pode ser escrito como uma multiplicação de fatores primos, o que pode ser útil para representar e operar números grandes.

Existem muitas conjecturas envolvendo números primos. A maioria das pessoas acredita que essas conjecturas são verdadeiras, mas ninguém foi capaz de prová-las. Por exemplo, as seguintes conjecturas são famosas:

- **Conjectura de Goldbach:** Cada número inteiro par $n > 2$ pode ser representado como uma soma $n = a + b$ de forma que tanto a quanto b são primos.
- **Conjectura dos primos gêmeos:** Existe um número infinito de pares da forma $\{p, p + 2\}$, onde tanto p quanto $p + 2$ são primos.
- **Conjectura de Legendre:** Sempre existe um primo entre os números n^2 e $(n+1)^2$, onde n é qualquer número inteiro positivo.

O crivo de Eratóstenes é um algoritmo de pré-processamento que constrói um array com o qual podemos verificar eficientemente se um número dado entre $2 \ldots n$ é primo e, se não for, encontrar um fator primo do número. O algoritmo constrói um array *sieve* cujas posições $2, 3, \ldots, n$ são usadas. O valor $sieve[k] = 0$ significa que k é primo, e o valor $sieve[k] \neq 0$ significa que k não é primo e um dos seus fatores primos é $sieve[k]$. O algoritmo itera através dos números $2 \ldots n$ um por um. Sempre que um novo primo x é encontrado, o algoritmo registra que os múltiplos de x ($2x, 3x, 4x, \ldots$) não são primos, pois o número x os divide.

Lista de funções

Func **fp(ll n)** $O(\sqrt{n})$
retorna se um número n é primo ou não.

FUNC **sieve(ll upperbound)** $O(N \log \log N)$
Cria um vetor de números primos até 10^7 (esse limite pode ser aumentado, mas é suficiente para a maior parte dos problemas). A função não possui retorno, e as variáveis mais úteis são definidas globalmente (a fim de que possam ser utilizadas por outras funções).

FUNC **isPrime(ll N)** $O(\sqrt{N} \log \sqrt{N})$
Determina se um número N é primo. É restrita a $N \leq F^2$, em que F é o último primo na lista construída por $sieve$. Sua complexidade é $O(\sqrt{N} \log \sqrt{N})$. Contudo, para primos pequenos, a execução é $O(1)$.

```
1
2       bool fp(ll n){
3           if(n<2) return false;
4           for(ll i=2; i*i<=n; i++){
5               if(n%i == 0) return false;
6           }
7           return true;
8       }
9
10      ll sieve_size;          // Generator's limit
11      bitset<10000010> bs;    // Each bit tells if a number is prime
12      vector<int> primes;     // Prime list
13
14      void sieve(ll upperbound) {
15          sieve_size = upperbound + 1;
16          bs.set();
17          bs[0] = bs[1] = 0;
18          for (ll i = 2; i <= sieve_size; i++)
19              if (bs[i]) {
20                  for (ll j = i * i; j <= sieve_size; j += i) bs[j] = 0;
21                  primes.push_back((int)i);
22              }
23      }
24
25      bool isPrime(ll N) {    // Rely on sieve prime list
26          if (N <= sieve_size) return bs[N];
27          for(int p: primes)
28              if (N % p == 0) return false;
29          return true;
30      }
```

6.2 Fatores primos

Todo número pode ser escrito como uma multiplicação de fatores primos. Para todo número $n > 1$, existe uma fatoração prima única: $n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$. Outras Propriedades:

- **Número de Divisores**: Divisores de um número são números inteiros que dividem esse número de forma exata. O número de divisores de um número é dado por: $\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1)$;

- **Soma de divisores**: A soma dos divisores de um número é dada por: $\sigma(n) = \prod_{i=1}^{k}(1 + p_i + \ldots + p_{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1}-1}{p_i-1}$;
- **Produto de divisores**: O produto dos divisores de um número é dado por: $\mu(n) = n^{\frac{\tau(n)}{2}}$
- **MDC**: Segundo o Algoritmo de Euclides o MDC é definido como gcd(a,b) $= \begin{cases} a & \text{se } b = 0 \\ \gcd(b, a \mod b) & \text{se } b \neq 0 \end{cases}$
- **MMC**: MMC$(a, b) = \frac{ab}{\text{MDC}(a,b)}$
- **Coprimos**: Dois inteiros a e b são coprimos se $gcd(a, b) = 1$.
- **Números de coprimos**: Uma maneira eficiente de contar os coprimos de um número é usar a função Totiente de Euler, $\phi(N) = N \times \prod_{PF}(1 - \frac{1}{PF})$.

Lista de funções

Func **primeFactors(ll N)** $O(\sqrt{N} \log \sqrt{N})$
retorna um vetor com os fatores primos de um número.

Func **numDiv(ll N)** $O(\sqrt{N} \log \sqrt{N})$
retorna o número de divisores de n.

Func **sumDiv(ll N)** $O(\sqrt{N} \log \sqrt{N})$
retorna a soma dos divisores de n.

Func **multDiv(ll N)** $O(\sqrt{N} \log \sqrt{N})$
retorna a multiplicação dos divisores de n.

Func **numDIffPF(ll N)** $O(N \log N)$
Calcula a quantidade de fatores primos diferentes para $0 \leq j \leq N$ e armazena os valores no vetor $numDiffPF$.

Func **MDC(ll N)** $O(\log n)$
retorna o mdc de a e b.

Func **MMC(ll N)** $O(\log n)$
retorna o mmc de a e b

Func **EulerPhi(ll N)** $O(\sqrt{N} \log \sqrt{N})$
Retorna a quantidade de coprimos de N

```
1
2       // Rodar sieve antes
3
4       vector<int> primeFactors(ll N){
5           vector<int> factors;
6           ll PF_idx = 0;
7           ll PF = primes[PF_idx];
8           while (PF * PF <= N) {
```

```
9               while (N % PF == 0) {
10                  N /= PF;
11                  factors.push_back(PF);
12              }
13              PF = primes[++PF_idx];
14          }
15          if (N != 1) factors.push_back(N);
16          return factors;
17      }
18
19      ll numDiv(ll N) {
20          ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
21          while (PF * PF <= N) {
22              ll power = 0;
23              while (N % PF == 0) { N /= PF; power++; }
24              ans *= (power + 1);
25              PF = primes[++PF_idx];
26          }
27          if (N != 1) ans *= 2;
28          return ans;
29      }
30
31      ll sumDiv(ll N) {
32          ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
33          while (PF * PF <= N) {
34              ll power = 0;
35              while (N % PF == 0) { N /= PF; power++; }
36              ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
37              PF = primes[++PF_idx];
38          }
39          if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);
40          return ans;
41      }
42
43      ll multDiv(ll N) {
44          ll t = numDiv(N)/2;
45          return pow(N, t);
46      }
47
48      int numDiffPF[MAXN];
49
50      void numDIffPF() {
51          memset(numDiffPF, 0, sizeof(numDiffPF));
52          for (int i = 2; i < MAX_N; i++) {
53              if(numDiffPF[i] != 0) continue;
54              for (int j = i; j < MAX_N; j += i)
55                  numDiffPF[j]++;
56          }
57      }
58
59      int mdc(int a, int b) { return b == 0 ? a : mdc(b, a % b); }
60
61      int mmc(int a, int b) { return a * (b / mdc(a, b)); }
62
63      ll EulerPhi(ll N) {
```

```
64          ll PF_idx = 0, PF = primes[PF_idx], ans = N; // start from ans = N
65          while (PF * PF <= N) {
66              if (N % PF == 0) ans -= ans / PF; // only count unique factor
67              while (N % PF == 0) N /= PF;
68              PF = primes[++PF_idx];
69          }
70          if (N != 1) ans -= ans / N; // last factor
71          return ans;
72      }
```

6.3 Algoritmo de Euclides extendido: Resolvendo equações Diofantinas

---

O objetivo é resolver equações da forma $ax + by = c$, em que a, b, c são constantes conhecidas, e deve-se encontrar $\{x, y\}$ que solucionem a equação.

Para isso, é utilizado $d = MMC(a, b)$.Se d não é divisor de c, então são existem soluções inteiras. Caso contrário, existem várias soluções, e o algoritmo de Euclides extendido consegue encontrar a primeira delas $\{x_0, y_0\}$. A partir disso, outras soluções podem ser encontradas a partir de

$$x_n = x_0 + (b/d)n \qquad\qquad y_n = y_0 - (a/d)n$$

Lista de funções

FUNC **extendedEuclid(int a, int b)** $O(\log n)$
retorna o X e o Y que satisfazem a equação.

```
1  void extendedEuclid(int a, int b) {
2      if (b == 0) {
3          x = 1;
4          y = 0;
5          d = a;
6          return;
7      }
8      extendedEuclid(b, a % b); // similar as the original gcd
9      int x1 = y;
10     int y1 = x - (a / b) * y;
11     x = x1;
12     y = y1;
13 }
```

6.4 Aritmética modular

Em aritmética modular, o conjunto de números é limitado de forma que apenas os números $0, 1, 2, \ldots, m-1$ são usados, onde m é uma constante. Cada número x é representado pelo número $x \mod m$: o resto da divisão de x por m. Por exemplo, se $m = 17$, então 75 é representado por $75 \mod 17 = 7$. Propriedades:

- $(x + y) \mod m = (x \mod m + y \mod m) \mod m$
- $(x - y) \mod m = (x \mod m - y \mod m) \mod m$
- $(x \cdot y) \mod m = (x \mod m \cdot y \mod m) \mod m$
- $x^n \mod m = (x \mod m)^n \mod m$
- $x^{y^z} \mod m = x^{(y^z \mod (m-1))} \mod m$

6.4.1 Exponenciação modular

A necessidade de calcular eficientemente o valor de $x^n \mod m$ é frequente. Isso pode ser feito em tempo $O(\log n)$ usando a seguinte recursão:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \cdot x^{n/2} & \text{if } n \text{ é par} \\ x^{n-1} \cdot x & \text{if } n \text{ é ímpar} \end{cases}$$

É importante destacar que no caso de n ser par, o valor de $x^{n/2}$ é calculado apenas uma vez. Isso garante que a complexidade de tempo do algoritmo seja $O(\log n)$, porque n é sempre dividido ao meio quando é par.

Lista de funções

FUNC **modpow(int x, int n, int m)** $O(\log n)$
exponenciação modular, recursiva e iterativa.

```
1  int modpow(int x, int n, int m) {
2      if (n == 0) return 1%m;
3      long long u = modpow(x,n/2,m);
4      u = (u*u)%m;
5      if (n%2 == 1) u = (u*x)%m;
6      return u;
7  }
8
9  ll modpow(ll x, unsigned ll y, ll mod){
10     ll res=1; x=x%mod;
11     while(y>0){
12         if (y&1) res= (res*x)%mod; y=y>>1; x=(x*x)%mod;
13     }
14     return res;
15 }
```

6.4.2 Teorema de Fermat e teorema de Euler

O teorema de Fermat afirma que $x^{m-1} \mod m = 1$ quando m é primo e x e m são coprimos. Isso também resulta em $x^k \mod m = x^{k \mod (m-1)} \mod m$.

Mais geralmente, o teorema de Euler afirma que $x^{\phi(m)} \mod m = 1$ quando x e m são coprimos. O teorema de Fermat segue do teorema de Euler, pois se m é primo, então $\phi(m) = m - 1$.

6.4.3 Inversão modular

O inverso de x modulo m é um número x^{-1} tal que $xx^{-1} \mod m = 1$. No entanto, um inverso modular nem sempre existe. Por exemplo, se $x = 2$ e $m = 4$, a equação $xx^{-1} \mod m = 1$ não pode ser resolvida, porque todos os múltiplos de 2 são pares e o resto nunca pode ser 1 quando $m = 4$. Descobre-se que o valor de $x^{-1} \mod m$ pode ser calculado exatamente quando x e m são coprimos.

Se um inverso modular existir, ele pode ser calculado usando a fórmula: $x^{-1} = x^{\phi(m)-1}$

Se m é primo, a fórmula se torna: $x^{-1} = x^{m-2}$

6.5 Análise combinatória

Problemas que envolvem análise combinatória geralmente serão acompanhados de ideias como **"total de possibilidades"**, **"quantos casos"** e a definição de alguma **sequência**, por exemplo. A análise de combinações é útil, então, para evitar altos custos de computação, já que algoritmos de força bruta são, na maioria dos casos, inviáveis.

- **Permutação:** Indica de quantas maneiras é possível reordenar items de uma sequência.
  Seja n o total de itens, m o número de elementos repetidos e $r_i > 1$ a quantidade de repetições do item i. Então
  - *Permutação simples:* $P(n) = n!$
  - *Permutação com repetição*: $P_r(n)^{r_1,\dots,r_m} = n!\ /\ (r_1! \times \dots \times r_m!)$
  - *Permutação circular*: $P_c(n) = (n-1)!$
- **Arranjo:** Indica de quantas maneiras é possível formar uma sequência distinta de k itens (nesse caso, a ordem importa).
  - *Arranjo:* $A(n,k) = n!\ /\ (n-k)!$
- **Combinação:** Indica de quantas maneiras é possível formar conjuntos distintos de k itens (portanto, a ordem não importa).
  - *Combinação:* $C(n,k) = n!\ /\ k!(n-k)!$
  - **Uma observação importante é que $C(n,k) = C(n,n-k)$, e isso pode ser útil para reduzir a computação quando $k > (n-k)$.**

Lista de funções

Func **fatorialEinverso(ll maxV)** $O(n \log n)$
constroi um vetor de fat com o fatorial de 1 a N modulo MOD e um vetor inve com o 1/fat(n) %MOD. A complexidade de criar apenas o vetor fatorial é $(O(n))$, logo é interessante criar o vetor de inversos apenas se o problema requerer diversas contas de combinações.

FUNC **comb(ll a, ll b)** $O(1)$
Retorna a combinação $C(a, b)\%MOD$

FUNC **arr(ll a, ll b)** $O(1)$
Retorna o Arranjo $A(a, b)\%MOD$

```
1
2  // pode ser maior q 1e6+10
3  // array com o fatorial dos n meros \% MOD
4  ll fat[(int)1e6+10];
5  // array com 1/fat dos n meros \%MOD
6  ll inve[(int)1e6+10];
7
8  void fatorialEinverso(ll maxV=1e6){
9      fat[0] = 1;
10     inve[0] = 1;
11     for(int i=1; i<=maxV; i++){
12         fat[i] = (fat[i-1]*i)\%MOD;
13         inve[i] = modpow(fat[i],MOD-2,MOD);
14     }
15 }
16
17 ll comb(ll a, ll b){
18     return fat[a]*inve[b]\%MOD*inve[a-b]\%MOD;
19 }
20
21 ll arr(ll a, ll b){
22     return fat[a]*inve[a-b]\%MOD;
23 }
```

6.5.1 Coeficientes binomiais (e triângulo de Pascal)

Esses são os coeficientes que acompanham a expansão em potências de um **binômio de Newton**, $(x + y)^n$. Uma utilidade desses coeficientes é o cálculo eficiente de combinações associadas a um mesmo conjunto, isso porque, para $k \geq 0$, o coeficiente binomial a_k é igual a $C(n, k)$.

Por exemplo, $(x + y)^3 = 1x^3 + 3x^2y + 3xy^2 + y^3$ e $\{1, 3, 3, 1\}$ são as combinações $C(3, k)$ para $k = \{0, 1, 2, 3\}$ respectivamente.

LISTA DE FUNÇÕES

FUNC **comb(ll n, ll k, int init = 1)** $O(n^2)$
Retorna diretamente a combinação $C(n, k)$. É um DP *top-down* para calcular os coeficientes binomiais e é util para cálculo de combinações para uma quantidade moderada de tamanhos k

FUNC **pascals(ll n)** $O(n^2)$
Calcula o triângulo de Pascal, armazenado na matriz triangular C. Para acessar uma combinação $C(n, k)$, basta acessar $C[n][k]$. Basicamente, é uma versão *bottom-up* do algoritmo anterior e é útil quando o cálculo envolver quase ou todo k.

```
1  // implementa o diferente para os codigos de combina o
2  vector<vector<ll>> C;
3
4  ll comb(ll n, ll k, int init = 1) {
5      if(init) C = vector<vector<ll>>(n+1,vector<ll>(n+1,INT_MIN));
6      if(C[n][k] != INT_MIN) return C[n][k];
7      if(k == 0 || k == n) return 1;
8      C[n][k] = comb(n-1,k-1,0) + comb(n-1,k,0);
9      return C[n][k];
10 }
11
12 void pascals(ll n) {
13     C = vector<vector<ll>>(n+1,vector<ll>(n+1));
14     for(ll i = 0; i <= n; i++) {
15         C[i][0] = C[i][i] = 1;
16         for(ll j = 1; j < i; j++)
17             C[i][j] = C[i-1][j-1] + C[i-1][j];
18     }
19 }
```

6.5.2 Boxes and balls

”Boxes and balls” é um modelo bastante útil, onde contamos as maneiras de colocar k bolas em n caixas. Vamos considerar três cenários:

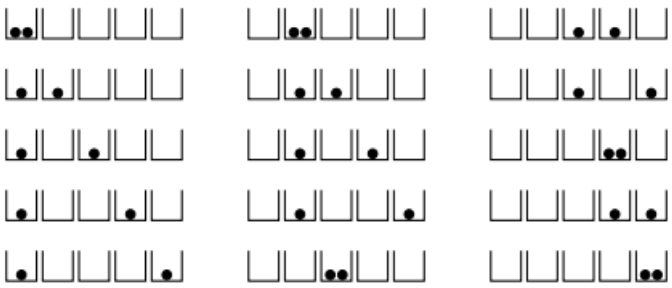


Figura 6.1: Figura do Cenário 2

1. **Cenário 1**: Cada caixa pode conter no máximo uma bola. Por exemplo, quando $n = 5$ e $k = 2$, existem 10 soluções. Neste cenário, a resposta é diretamente o coeficiente binomial $\binom{n}{k}$.
2. **Cenário 2**: Uma caixa pode conter várias bolas. Por exemplo, quando $n = 5$ e $k = 2$, existem 15 soluções.

   O processo de colocar as bolas nas caixas pode ser representado como uma sequência que consiste nos símbolos ”o”e ”$\rightarrow$”. Inicialmente, assumimos que estamos na caixa mais à esquerda. O símbolo ”o”significa que colocamos uma bola na caixa atual, e o símbolo ”$\rightarrow$”significa que nos movemos para a próxima caixa à direita. Usando esta notação, cada solução é uma sequência que contém k vezes o símbolo ”o”e $n - 1$ vezes o símbolo ”$\rightarrow$”. Por exemplo, a solução do canto superior direito na imagem acima corresponde à sequência ”$\rightarrow \rightarrow$ o $\rightarrow$ o $\rightarrow$”. Assim, o número de soluções é dado por $\binom{k+n-1}{k}$.
3. **Cenário 3**: Cada caixa pode conter no máximo uma bola, e além disso, duas caixas adjacentes não podem ambas conter uma bola. Por exemplo, quando $n = 5$ e $k = 2$, existem 6 soluções. Neste cenário, podemos assumir que k bolas são inicialmente colocadas nas caixas e há uma caixa vazia entre cada duas caixas adjacentes. A

tarefa restante é escolher as posições para as caixas vazias restantes. Existem $n - 2k + 1$ tais caixas e $k + 1$ posições para elas. Assim, usando a fórmula do cenário 2, o número de soluções é dado por $\binom{n-k+1}{n-2k+1}$.

6.5.3 Coeficientes multinomiais

O coeficiente multinomial $\binom{n}{k_1,k_2,\ldots,k_m}$ é igual a $\frac{n!}{k_1!k_2!\cdots k_m!}$, e representa o número de maneiras que podemos dividir n elementos em subconjuntos de tamanhos $k_1, k_2, ..., k_m$, onde $k_1 + k_2 + \cdots + k_m = n$. Os coeficientes multinomiais podem ser vistos como uma generalização dos coeficientes binomiais; se $m = 2$, a fórmula acima corresponde à fórmula do coeficiente binomial.

6.5.4 Números de Catalan

Os número de Catalan são definidos pela fórmula

$$Cat(n) = \frac{C(2n, n)}{(n + 1)} \tag{6.1}$$

Novamente, o cálculo dessa função envolve fatoriais, o que pode ser um empecilho para o cálculo. Portanto, quando for necessário computar $Cat(n)$ para vários valores de n, é útil utilizar uma abordagem *DP bottom-up*, visto que $Cat(0) = 1$, e

$$Cat(m) = \frac{4m^2 - 2m}{m^2 + m} \cdot Cat(m - 1) \tag{6.2}$$

Outra formula para se calcular o número de catalan:

$$Cat(n) = \frac{1}{n + 1}\binom{2n}{n} \tag{6.3}$$

Números de Catalan podem ser utilizados para resolver problemas de combinação que envolvem a ”divisão”de um conjunto. Por exemplo, é possível calcular

- O número de subárvores de uma árvore de n vértices.
- O número de expresões que contém n parênteses devidamente pareados.
- De quantas maneiras é possível colocar parênteses em uma expressão de tamanho $n + 1$.
- De quantas formas é possível traingular um polígono convexo com $n + 2$ lados.
- O número de **caminhos monótonos** que em uma matriz $n \times n$. Um caminho é monótono quando atravessa a matriz unindo o canto inferior esquerdo ao canto superior direito sem passar acima da diagonal principal e obedecendo sempre passos inteiros na horizontal ou vertical.

Lista de funções

FUNC **genCat() $O(n)$**
Calcula $Cat(m)$ $0 \leq m \leq n$ utilizando programação dinâmica e armazena os valores no array CAT. Útil quando é preciso calcular muitos valores.

FUNC **Cat(n)** $O(n^2)$
Retorna $Cat(n)$ utilizando a fórmula de combinação. Válida para cálculos isolados.

```
CAT = [0]*MAXN

def genCat():
    CAT[0] = 1;
    for m in range(1,MAXN):
        CAT[m] = CAT[m-1] * (4*m*m - 2*m)/(m*m + m)

def Cat(n):
    if n == 0: return 1
    return C(2n,n) / (n+1) # C - funcao de combinacao
```

6.6 Inclusão-exclusão

Inclusão-exclusão é uma técnica que pode ser usada para contar o tamanho de uma união de conjuntos quando os tamanhos das interseções são conhecidos, e vice-versa.

- $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$
- $|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|$

6.7 Teoria dos jogos

Problemas que envolvem jogos geralmente são compostos por um jogo que

- é imparcial, ou seja, ambos os jogadores seguem as mesmas regras de jogada
- é acíclico, ou seja, finito

A base para resolução desse tipo de problema é o **jogo de Nim**. Nele, existem várias pilhas, cada uma com uma determinada quantidade de elementos. Os jogadores jogam otimamente e alternam suas jogadas e, em cada uma delas, devem retirar uma quantidade $q \geq 1$ de elementos de alguma pilha. O jogador que ficar impossibilitado de jogar (ou seja, aquele que chegar uma jogada em que o número de elementos restantes é 0) perde o jogo.

Teorema : Seja a_i a quantidade total de elementos na pilha i em um determinado estado do jogo. Para n pilhas a operação $NIM = a_0 \otimes a_1 \otimes ... \otimes a_{n-1}$ define se o jogador correspondente a esse estado é capaz de ganhar o jogo.

- $NIM > 0 \quad \longrightarrow \quad vitoria$
- $NIM == 0 \quad \longrightarrow \quad derrota$

Contudo, nem todos os jogos (principalmente os mais complexos), podem ser reduzidos ao problemas de pilhas descrito acima. Nesse caso, é comum a necessidade de uso dos **números de Grundy**. Seu cálculo é feito a partir de

$$Grundy(n) = mex\{Grundy(s_1), Grundy(s_2), ..., Grundy(s_m)\} \tag{6.4}$$

Nessa operação, mex representa o mínimo excludente do conjunto, ou seja, o menor inteiro não negativo (inclui 0) que não pertence ao conjunto. Além disso, s_i representa um estado de jogo que pode ser alcançado a partir do estado n.

Teorema : Dado $Grundy(n)$, é possível determinar a condição de vitória do jogador da seguinte forma:

- $Groundy(n) > 0 \quad \longrightarrow \quad vitoria$
- $Groundy(n) == 0 \quad \longrightarrow \quad derrota$

É preciso ressaltar que, diferentemente do cálculo de NIM, $Grundy(n)$ não deve ser usado diretamente para jogos compostos, ou seja, jogos que envolvem vários conjuntos(pilhas, por exemplo) de elementos. Nesse caso é preciso usar o **Teorema de Sprague-Grundy**.

Teorema de Sprague-Grundy : Dado o estado de um jogo composto por n conjuntos, é possível determinar $SPRAGUE = \overline{Groundy(1) \otimes Groundy(2) \otimes .. \otimes Groundy(n)}$, e, de forma parecida com os casos anteriores,

- $SPRAGUE > 0 \quad \longrightarrow \quad vitoria$
- $SPRAGUE == 0 \quad \longrightarrow \quad derrota$

Como é possível perceber, função de envolvem números de $grundy$ podem ser computacionalmente custosas, tanto pela recursão envolvida quanto pela necessidade de busca em conjuntos (base da função mex). Normalmente, algoritmos para resolver esse tipo de problema precisam ser baseados em programação dinâmica, sendo sua forma dependente de cada jogo e situação.

6.7.1 Exemplo: Jogo da velha em uma dimensão

Nesse jogo, existe um tabuleiro de dimensões $1 \times n$. Ganha o primeiro jogador que completar uma sequência de 3 'X's, ou seja, além de marcar para si mesmos, os jogadores também podem facilitar o jogo do adversário (já que o símbolo de marcação é o mesmo). Portanto, o objetivo é determinar, a partir de um estado de jogo dado, se o jogador atual pode ganhar o jogo.

ENTRADA :
```
5
.....
5
..X..
6
X.X.X.
12
............
0
```

SAIDA :
```
S
N
S
N
```

```
#include <bits/stdc++.h>
#define MAX 10001
```

```
#define g(x) (x > 0 ? grundy[x] : 0)

int grundy[MAX];
char flag[MAX], str[MAX];

void fill_grundy() {
    memset(flag,0,MAX);
    int n,i,j,tmp,mflag;
    for(n = 0; n < MAX; n++) {
        if(n < 3) {
            grundy[n] = n > 0;
            continue;
        }
        for(i = n - 3, j = -2, mflag = 0;
            i >= j; i--, j++) {
            tmp = g(i)^g(j);
            flag[tmp] = 1;
            if(tmp > mflag) mflag = tmp;
        }
        for(j = 0; flag[j]; j++);
        grundy[n] = j;
        memset(flag,0,mflag+1);
    }
}


int main() {
    fill_grundy();
    int n,i,j;
    while(scanf("%d ",&n), n) {
        scanf("%s ", str);
        for(i = 0; i < n-2 &&
            ((str[i] == 'X') + (str[i+1] == 'X')
             + (str[i+2] == 'X')) < 2; i++);
        if(i < n-2) { printf("S\n"); continue; }

        for(i = 0; i < n; i++) {
            if(str[i] != 'X') continue;
            for(j = -2; j < 3; j++)
                if(str[i+j] == '.') str[i+j] = 'x';
        }

        int c = 0, ans = 0;
        for(i = 0; i <= n; i++)
            if(i < n && str[i] == '.') c++;
            else {
                ans ^= grundy[c];
                c = 0;
            }

        printf("%s\n", ans ? "S" : "N");
```

```
    }
    return 0;
}
```

6.8 Teoremas e Fórmulas

- **Desarranjo**: o número $der(n)$ de permutações de n elementos em que nenhum dos elementos fica na posição original é dado por: $der(n) = (n-1)(der(n-1) + der(n-2))$, onde $der(0) = 1$ e $der(1) = 0$.
- **Fórmula de Euler para poliedros convexos**: $V - E + F = 2$, onde F é o número de faces.
- **Círculo de Moser**: o número de peças em que um círculo pode ser divido por cordas ligadas a n pontos tais que não se tem 3 cordas internamente concorrentes é dada por: $g(n) = \binom{n}{4} + \binom{n}{2} + 1 = \frac{1}{24}(n^4 - 6n^3 + 23n^2 - 18n + 24)$.
- **Teorema de Pick**: se I é o número de pontos inteiros dentro de um polígono, A a área do polígono e b o número de pontos inteiros na borda, então $A = i + b/2 - 1$.
- **Teorema de Zeckendorf**: qualquer inteiro positivo pode ser representado pela soma de números de Fibonacci que não inclua dois números consecutivos. Para achar essa soma, usar o algoritmo guloso, sempre procurando o maior número de fibonacci menor que o número.
- **Teorema de Wilson**: um número n é primo se e somente se $(n-1)! \equiv -1 \mod n$
- **Teorema de Euler**: se a e b forem coprimos entre si, então $a^{\phi(b)} \equiv 1 \mod b$ ou $a^n \equiv a^{n\%\phi(b)} \mod b$.
- **Teorema Pequeno de Fermat**: se p é um número primo, então, para qualquer inteiro a, $a^p - a$ é múltiplo de p. Ou seja, $a^p \equiv a \mod p$ ou $a^{p-1} \equiv 1 \mod p$.
- **Último Teorema de Fermat**: para qualquer inteiro $n > 2$, a equação $x^n + y^n = z^n$ não possui soluções inteiras.
- **Teorema de Lagrange**: qualquer inteiro positivo pode ser escrito como a soma de 4 quadrados perfeitos.
- **Conjectura de Goldbach**: qualquer par $n > 2$ pode ser escrito como a soma de dois primos (testada até 4×10^{18}).
- **Conjectura dos primos gêmeos**: existem infinitos primos p tal que $p + 2$ também é primo.
- **Conjectura de Legendre**: para todo n inteiro positivo, existe um primo entre n^2 e $(n+1)^2$.
- **divisibilidade por 11**: subtraia os elementos de índice impar dos de par se resultado for divisível por 11 ou 0 o número original também será divisível por 11.

Capítulo 7

Geometria

7.0.1 Estruturas básicas

Conjunto de funções para lidar com vetores, pontos e geometria computacional em geral.

O código utiliza a estrutura ‘point‘ para representar pontos no plano cartesiano. A estrutura ‘vec‘ representa um vetor e possui métodos para calcular o produto escalar, o produto vetorial e o ângulo entre dois vetores.

Lista de funções

FUNC **point(double _x, double _y)** $O(1)$
Construtor da estrutura ‘point‘. Recebe as coordenadas x e y do ponto.

FUNC **dist(point p1, point p2)** $O(1)$
Calcula a distância entre dois pontos.

FUNC **vec(double _x, double _y)** $O(1)$
Construtor da estrutura ‘vec‘. Recebe as coordenadas x e y do vetor.

FUNC **vec(point a, point b)** $O(1)$
Construtor da estrutura ‘vec‘. Recebe dois pontos e calcula o vetor que conecta os dois pontos.

FUNC **dot(vec a, vec b)** $O(1)$
Calcula o produto escalar de dois vetores.

FUNC **cross(vec a, vec b)** $O(1)$
Calcula o produto vetorial de dois vetores.

FUNC **ccw(point a, point b, point c)** $O(1)$
Calcula o sentido da rotação do triângulo formado pelos pontos a, b e c.

FUNC **angle(point a, point b, point c)** $O(1)$
Calcula o ângulo entre as linhas ab e ac.

FUNC **insegment(point p1, point p2, point p3)** $O(1)$
Verifica se o ponto p3 está no segmento de reta definido pelos pontos p1 e p2.

FUNC **inpolygon(vector¡point¿ poly, point p)** $O(n)$
Verifica se o ponto p está dentro do polígono definido pelo vetor de pontos ‘poly‘.

FUNC **area(vector¡point¿ poly)** $O(n)$
Calcula a área do polígono definido pelo vetor de pontos ‘poly‘.

FUNC **lineintersection(point p[])** $O(1)$
Verifica se as linhas se interceptam.

```
1  struct point {
2      double x, y;
3      point(){}
4      point(double \_x, double \_y) : x(\_x), y(\_y) {}
5      bool operator <(point other) const {
6          if(fabs(x - other.x) > eps)
7              return x < other.x;
8          return y < other.y;
9      }
10     bool operator ==(point other) const {
11         return (fabs(x - other.x) < eps && fabs(y - other.y) < EPS);
12     }
13 };
14
15 double dist(point p1, point p2) {
```

```
16      return hypot(p1.x - p2.x, p1.y - p2.y);
17  }
18
19  struct vec {
20      double x, y;
21      vec(){}
22      vec(double \_x, double \_y) : x(\_x), y(\_y) {}
23      vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
24  };
25
26  double dot(vec a, vec b) { return a.x*b.x + a.y*b.y; }
27  double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }
28  double ccw(point a, point b, point c) { return cross(vec(a,b), vec(a,c));}
29
30  double angle(point a, point b, point c) {
31      vec u(b,a), v(a,c);
32      return acos(dot(u,v) / sqrt(dot(u,u) * dot(v,v)));
33  }
34
35  bool insegment(point p1, point p2, point p3){
36      return (min(p1.x, p2.x) <= p3.x && p3.x <= max(p1.x, p2.x)) &&
37             (min(p1.y, p2.y) <= p3.y && p3.y <= max(p1.y, p2.y));
38  }
39
40  int inpolygon(vector<point> poly, point p) {
41      int n = poly.size();
42      bool inside = false;
43      for (int i = 0, j = n - 1; i < n; j = i++) {
44          if (((poly[i].y > p.y) != (poly[j].y > p.y)) &&
45              (p.x < (poly[j].x - poly[i].x) * (p.y - poly[i].y) / (poly[j].y - poly[i].y) + pol
46              inside = !inside;
47          }
48      }
49      return inside;
50  }
51
52  double area(vector<point> poly) {
53      int n = poly.size();
54      double area = 0.0;
55      for (int i = 0, j = n - 1; i < n; j = i++) {
56          area += (poly[j].x * poly[i].y - poly[i].x * poly[j].y);
57      }
58      return abs(area) / 2.0;
59  }
60
61  bool lineintersection(point p[]) {
62
63      ll or1 = ccw(p[0], p[1], p[2]);
64      ll or2 = ccw(p[0], p[1], p[3]);
65      ll or3 = ccw(p[2], p[3], p[0]);
66      ll or4 = ccw(p[2], p[3], p[1]);
67
68      if(or1 != or2 && or3 != or4) return 1;
69      else if(or1 == 0 && insegment(p[0], p[1], p[2])) return 1;
```

```
70      else if(or2 == 0 && insegment(p[0], p[1], p[3])) return 1;
71      else if(or3 == 0 && insegment(p[2], p[3], p[0])) return 1;
72      else if(or4 == 0 && insegment(p[2], p[3], p[1])) return 1;
73
74      return 0;
75  }
```

7.0.2 Convex Hull

Algoritmo que retorna o conjunto de pontos que formam a menor área convexa que engloba todos os pontos de um conjunto. O algoritmo mais comum é o "Graham Scan", que utiliza um ponto com a menor coordenada y como pivô, e ordena os demais pontos em sentido anti-horário, em relação ao pivô. Após ordenar os pontos, percorre cada um deles e realiza uma verificação para identificar quais pontos devem ser incluidos no convex hull. O convex hull é importante para problemas que envolvem encontrar o contorno de um conjunto de pontos, como por exemplo: otimização de rotas, encontrar a menor distância entre dois conjuntos de pontos, ou encontrar a menor área de um conjunto de pontos.

Essa estrutura pode ser implementada atraves do algoritmo Graham Scan, que tem complexidade de O(n log n). O algoritmo funciona da seguinte forma:

1. Encontra o ponto com menor coordenada y (pivô). 2. Ordena os pontos em sentido anti-horário em relação ao pivô. 3. Percorre cada ponto, verificando se a orientação do triângulo formado pelo ponto atual, o ponto anterior e o ponto seguinte, é anti-horária. Se for, o ponto anterior deve ser removido da lista de pontos do convex hull. 4. Para garantir a inclusão de pontos colineares, repetimos o algoritmo no sentido horario

Lista de funções

FUNC **hull()** $O(nlogn)$
Calcula o convex hull de um conjunto de pontos.

```
1  int np, nup, ndown;
2  point p[mxn], up[mxn], down[mxn], pivot;
3  set<point> convex;
4  // up representa o feixe no sentido anti-horario e down representa o feixe no sentido horario
5  // p eh o vetor de pont os disponiveis que sera ordenado para conseguir o feixo
6
7  bool operator<(point a, point b) { // Para o set<point>
8      return a.y < b.y || (a.y == b.y && a.x < b.x);
9  }
10
11 bool cmp1(point a, point b) {
12     vec u(pivot, a), v(pivot, b);
13     if(cross(u,v) == 0) return dot(u,u) < dot(v,v);
14     return cross(u,v) > 0;
15 }
16
17 bool cmp2(point a, point b) {
18     vec u(pivot, a), v(pivot, b);
19     if(cross(u,v) == 0) return dot(u,u) > dot(v,v);
20     return cross(u,v) > 0;
```

```
21 }
22
23 void hull() {
24
25     sort(p + 1, p + np); //Ordenamos os pontos para encontrar o pivo
26     pivot = p[0];
27
28     // Feixo anti-horario
29     sort(p + 1, p + np, cmp1);
30     p[np] = up[0] = p[0];
31     up[1] = p[1];
32     nup = 1; // nup representa o topo da pilha
33     for(int i = 2; i <= np; i++) {
34         while(cross(vec(up[nup], p[i]),
35                     vec(up[nup-1], up[nup])) > 0) nup--; //Desempilha enquanto o ponto anterio
36         up[++nup] = p[i];
37     }
38
39     // Feixo horario
40     sort(p + 1, p + np, cmp2);
41     down[0] = p[0];
42     down[1] = p[np-1];
43     ndown = 1;
44     for(int i = np-2; i >= 0; i--) {
45         while(cross(vec(down[ndown], p[i]),
46                     vec(down[ndown-1], down[ndown])) < 0) ndown--; // desempilhamos enquanto o
47         down[++ndown] = p[i];
48     }
49
50     // Inserimos no conjunto do feixo
51     for (int i = 0; i < nup; i++)
52         convex.insert(up[i]);
53     for (int i = 0; i < ndown; i++)
54         convex.insert(down[i]);
55 }
56
57 int main() {
58
59     int n; cin >> n;
60     set<point> st;
61     for(int i = 0; i < n; i++){
62         ll x, y; cin >> x >> y;
63         st.insert(point(x, y));
64     }
65     np = 0;
66     for (point pi: st) p[np++] = pi;
67
68     hull();
69
70     cout << convex.size() << endl;
71     for(point pi: convex)
72         cout << pi.x << "␣" << pi.y << endl;
73 }
```

Capítulo 8

Resoluções

8.1 Introductory Problems

8.1.0.1 Weird Algorithm

Objetivo

Você recebe um número n e deve simular o funcionamento do algoritmo. Se n for par divida por 2, se n é impar multiplique por 3 e some 1, faça isso até o número chegar a 1.

Estratégia

simule o algoritmo. Use long long.

Código

```
1  int main(){
2
3      unsigned long long n;
4
5      cin >> n;
6
7      cout << n;
8      while(n!=1){
9          if(n!=1) cout << "␣";
10         if(n%2!=0) n = 3*n+1;
11         else n = n/2;
12         cout << n;
13     }
14     cout << "\n";
15
16     return 0;
17 }
```

8.1.0.2 Missing Number

Objetivo

Você recebe todos os número de 1 a n menos um deles, ache o faltante.

Estratégia

Percorrer o array.

Código

```
1  int main(){
2
3      int n, a;
4      cin >> n;
5      vector<int> numeros(n+1, 0);
6      while(cin >> a){
7          numeros[a] = a;
8      }
9      for(int i=1; i<=n; i++){
10         if(!numeros[i]) cout << i << "\n";
11     }
12
13     return 0;
14 }
```

8.1.0.3 Repetitions

Objetivo

Você recebe uma string e quer saber a maior cadeia formada por caracteres iguais.

Estratégia

Percorrer o array encontrando o tamanho das cadeias formadas por apenas caracteres iguais.

Código

```
1
2  int main(){
3
4      int max_glob=0, max_loc=0;
5      char caracter, att=0;
6
7      while(cin >> caracter){
8          if(caracter != att){
9              max_glob = max(max_loc, max_glob);
10             max_loc = 0;
11             att = caracter;
12         }
13         max_loc++;
```

```
14      }
15
16      cout << max(max_loc, max_glob) << "\n";
17      return 0;
18  }
```

8.1.0.4 Increasing Array

Objetivo

Você recebe uma lista e quer que cada elemento seja pelo menos tão grande quanto o número anterior. cada movimento pode aumentar um número em 1, retorne o menor número de movimentos.

Estratégia

Percorra o vetor e, caso o elemento atual seja menor que o elemento anterior, aumente o elemento atual até que ele seja igual ao anterior e adicione a diferença à resposta.

Código

```
1
2  int main(){
3
4      long long n, att, ant, ans=0;
5      cin >> n;
6
7      cin >> ant;
8      while (cin >> att){
9          if(ant>att) ans += (ant - att);
10         ant = max(ant, att);
11     }
12
13     cout << ans << "\n";
14     return 0;
15 }
```

8.1.0.5 Permutations

Objetivo

Uma permutação de inteiros de 1 a n é chamado de bonito se não houver elementos adjacentes cuja diferença seja 1. Recebe um número n e retorne uma permutação bonita ou 'NO SOLUTION'

Estratégia

Para $n < 3$ retorne 'NO SOLUTION', se $n = 4$ retorne 2413, se $n >= 5$ printe os números impares e depois os pares.

Código

```
1
2  int main(){
3
```

```
4       int n, j=1;
5       cin >> n;
6
7       if(n == 2 || n == 3){
8           cout << "NO␣SOLUTION\n";
9           return 0;
10      }
11
12      if(n==4){
13          cout << "2␣4␣1␣3\n";
14          return 0;
15      }
16
17      for(int i=0; i<n; i++){
18          if(i) cout << "␣";
19
20          cout << j;
21          j += 2;
22
23          if(j>n) j = 2;
24      }
25
26      cout << "\n";
27      return 0;
28  }
```

8.1.0.6 Number Spiral

Objetivo

Você recebe varios a e b, linha e coluna de uma espiral numérica, e deve retornar o número da posição a b da matriz. [[1,2,9],[4,3,8],[5,6,7]].

Estratégia

Se $a > b$: se a for par retorne $(a * a) - (b - 1)$. se I for impar retorne $((a * a) - ((2 * a) - 2)) + (b - 1)$.

Se não: se b for impar retorne $b * b - (a - 1)$. Se não $((b * b) - ((2 * b) - 2)) + (a - 1)$.

Código

```
1   int main(){
2
3       unsigned long long n, a, b;
4       cin >> n;
5
6       while(cin >> a >> b){
7           if(a>b){
8               if(a%2!=0){
9                   cout << ((a*a) - ((2*a)-2)) + (b-1);
10              }
11              else{
12                  cout << a*a - (b-1);
13              }
```

```
14          }
15          else{
16              if(b%2==0){
17                  cout << ((b*b) - ((2*b)-2)) + (a-1);
18              }
19              else{
20                  cout << b*b - (a-1);
21              }
22          }
23          cout << "\n";
24      }
25
26      return 0;
27  }
```

8.1.0.7 Two Knights

Objetivo

Sua tarefa é contar $k = 1, 2, ..., n$ o número de maneiras pelas quais dois cavalos podem ser colocados em um kXk tabuleiro de xadrez para que não se ataquem.

Estratégia

Retorne: Número total de maneiras que os cavalos podem ser posicionados $((K^2 * (K^2 - 1))/2)$ - Número de possições onde eles se atacam $(4 * (K - 1) * (k - 2))$.

Código

```
1  int main(){
2
3      unsigned long long n;
4      cin >> n;
5      for(unsigned long long i=1; i<=n; i++){
6          cout << (((i*i*i*i)-((i*i)-1))/2) - 4*((i-1)*(i-2)) << "\n";
7      }
8
9      return 0;
10 }
```

8.1.0.8 Two Sets

Objetivo

Você recebe um número n e quer dividir todos os números de $1an$ em 2 conjuntos de soma igual.

Estratégia

Seja S a soma de todos os valores do array e caso a S for impar retorne NO. Se não, passe pelo vetor achando elementos cuja a soma é igual a $S/2$.

Código

```
int main(){

    long long n, som;
    vector<int> a, b;
    cin >> n;

    som = (n*(n+1))/2;

    if(som%2==1){
        cout << "NO\n";
        return 0;
    }

    som /= 2;
    for(int i=n; i>0; i--){
        if(som-i>=0){
            a.push_back(i);
            som -= i;
        }
        else{
            b.push_back(i);
        }
    }

    sort(a.begin(), a.end());
    sort(b.begin(), b.end());

    cout << "YES\n";
    cout << a.size() << "\n";
    for(auto elemento: a) cout << elemento << "␣";
    cout << "\n";
    cout << b.size() << "\n";
    for(auto elemento: b) cout << elemento << "␣";
    cout << "\n";

    return 0;
```

8.1.0.9 Bit Strings

Objetivo

Você recebe um número n e deve retornar o número de sequencias de binarias de comprimento n.

Estratégia

retorne 2^n.

Código

```
#define mod 1000000007

int main(){
```

```
4
5       long long n, resp=1;
6       cin >> n;
7
8       for(int i=0; i<n; i++){
9           resp = (resp*2)%mod;
10      }
11
12      cout << resp << "\n";
13
14      return 0;
15  }
```

8.1.0.10 Trailing Zeros

Objetivo

Você recebe um número n e deve retornar o número de zeros a direita do número $n!$.

Estratégia

O número de 0 a direita de $n!$ é o mesmo que o número de potencias de 5 do número.

Código

```
1   long long findTrailingZeros(long long n)
2   {
3       if (n < 0)
4           return -1;
5
6       long long count = 0;
7
8       for (long long i = 5; n / i >= 1; i *= 5)
9           count += n / i;
10
11      return count;
12  }
13
14  int main(){
15
16      long long n;
17      cin >> n;
18      cout << findTrailingZeros(n) << "\n";
19
20      return 0;
21  }
```

8.1.0.11 Coin Piles

Objetivo

Dado o número de elementos nas pilhas A e B, e podendo ou retirar um elemento da pilha A e 2 elementos da pilha B ou 1 elemento da pilha B e 2 da pilha A. Diga se da para esvaziar as 2 pilhas.

Estratégia

Para conseguir, A+B deve ser multiplo de 3 e $min(A, B) * 2 >= max(A, B)$.

Código

```
1  int main(){
2
3      long long t, a, a_aux, b, b_aux, m;
4      cin >> t;
5
6      while(t--){
7          cin >> a >> b;
8
9          if(((a+b)%3==0) && (min(a, b) * 2 >= max(a, b))) cout << "YES\n";
10         else cout << "NO\n";
11     }
12
13     return 0;
14 }
```

8.1.0.12 CPalindrome Reorder

Objetivo

Dado uma string reordene-a para formar um palindromo.

Estratégia

caso tenha mais de uma letra em quantidade impar, não da para reordenar. Caso contrário apenas reordene.

Código

```
1  int main(){
2      int conterpar = 0, sz=0, l, r;
3      char caracter;
4      vector<pair<int,char>> palavra;
5
6      for(int i='A'; i<='Z'; i++) palavra.push_back(make_pair(0,char(i)));
7
8      while(cin >> caracter){
9          palavra[caracter-'A'].first+=1;
10         sz++;
11     }
12
13     vector<char> palindromo(sz);
14
15     sort(palavra.rbegin(), palavra.rend());
16
```

```
17      l=0, r=sz-1;
18      for(auto par: palavra){
19          if(par.first == 0) break;
20          if(par.first%2==1){
21              if(conterpar < 2){
22                  conterpar++;
23                  if(sz%2==0){
24                      cout << "NO␣SOLUTION\n";
25                      return 0;
26                  }
27                  palindromo[ceil(sz/2)] = par.second;
28                  par.first--;
29              }
30              else{
31                  cout << "NO␣SOLUTION\n";
32                  return 0;
33              }
34          }
35
36          for(int i=0; i<par.first; i+=2){
37              palindromo[l] = par.second;
38              palindromo[r] = par.second;
39              l++;
40              r--;
41          }
42      }
43
44      for(auto caracter: palindromo){
45          cout << caracter;
46      }
47      cout << "\n";
48      return 0;
49  }
```

8.1.0.13 Gray Code

Objetivo

Um código Gray é uma lista de todos 2^n cadeias de bits de comprimenton, onde quaisquer duas strings sucessivas diferem em exatamente um bit. Sua tarefa é criar um código Gray para um determinado comprimento n.

Estratégia

Gere o codigo de gray.

Código

```
1  void generateGrayarr(int n)
2  {
3          if (n <= 0)
4                  return;
5
6          vector<string> arr;
7
```

```
8           arr.push_back("0");
9           arr.push_back("1");
10
11          int i, j;
12          for (i = 2; i < (1<<n); i = i<<1){
13
14                  for (j = i-1 ; j >= 0 ; j--)
15                          arr.push_back(arr[j]);
16
17                  for (j = 0 ; j < i ; j++)
18                          arr[j] = "0" + arr[j];
19
20                  for (j = i ; j < 2*i ; j++)
21                          arr[j] = "1" + arr[j];
22          }
23
24          for (i = 0 ; i < arr.size() ; i++ ) cout << arr[i] << "\n";
25  }
26
27  int main()
28  {
29      int n;
30      cin >> n;
31      generateGrayarr(n);
32
33          return 0;
34  }
```

8.1.0.14 Creating Strings

Objetivo

Dada uma string, sua tarefa é gerar todas as diferentes strings que podem ser criadas usando seus caracteres.

Estratégia

Use next_permutation.

Código

```
1   int main(){
2
3       string palavra;
4       vector<string> ans;
5       cin >> palavra;
6
7       sort(palavra.begin(), palavra.end());
8
9       do {
10          ans.push_back(palavra);
11      } while(next_permutation(palavra.begin(), palavra.end()));
12
13      cout << ans.size() << "\n";
14      for(auto p: ans) cout << p << "\n";
```

```
15
16      return 0;
17  }
```

8.1.0.15 Apple Division

Objetivo

Você recebe um array e deve dividi-lo em 2 grupos de forma a minimizar a diferença entre a soma dos elementos dos grupos.

Estratégia

O tamanho maximo do arrsy é 20, logo testar todas as possibilidades é 2^{20}.

Código

```
1  long long n, som=0, min_dif=LONG_LONG_MAX;
2  vector<long long> numeros(20);
3
4  void calcula(int i, long long soma){
5
6      if(i==n){
7          min_dif = min(min_dif, abs(((som-soma)-soma)));
8          return;
9      }
10
11     calcula(i+1, soma+numeros[i]);
12     calcula(i+1, soma);
13 }
14
15 int main(){
16
17     cin >> n;
18
19     for(int i=0; i<n; i++){
20         cin >> numeros[i];
21         som +=numeros[i];
22     }
23
24     calcula(0, 0);
25
26     cout << min_dif << "\n";
27
28 }
```

8.1.0.16 Chessboard and Queens

Objetivo

Dado um tabuleiro 8x8 com obstaculos retorne o número de possibilidades para colocar 8 damas sem que elas se ameassem.

Estratégia

Backtracking. Para cada dama teste todas as possibilidades que ela pode ficar em sua linha, mate um ramo quando a rainha ameassar outra.

Código

```
int ans=0;
vector<vector<char>> matriz(8, vector<char> (8, '.'));

int valid(int i_a, int j_a){
    int i, j;

    if(matriz[i_a][j_a] == '*') return 0;

    for(int a=0; a<8; a++)
        if(matriz[a][j_a]=='1' || matriz[i_a][a]=='1') return 0;

    i=i_a; j=j_a;
    while(i < 8 && j < 8 ){
        if(matriz[i][j]=='1') return 0;
        i++; j++;
    }
    i=i_a; j=j_a;
    while(i < 8 && j >= 0 ){
        if(matriz[i][j]=='1') return 0;
        i++; j--;
    }
    i=i_a; j=j_a;
    while(i >= 0 && j >= 0){
        if(matriz[i][j]=='1') return 0;
        i--; j--;
    }
    i=i_a; j=j_a;
    while(i >= 0 && j < 8){
        if(matriz[i][j]=='1') return 0;
        i--; j++;
    }

    return 1;
}

void solve(int j){

    if(j==8){
        ans++;
        return;
    }

    for(int i=0; i<8; i++){
        if(!valid(i, j)) continue;;

        matriz[i][j] = '1';
        solve(j+1);
        matriz[i][j] = '.';
    }
```

```
50
51  }
52
53  int main(){
54
55      for(int i=0; i<8; i++)
56          for(int j=0; j<8; j++)
57              cin >> matriz[i][j];
58
59      solve(0);
60
61      cout << ans << "\n";
62
63      return 0;
64  }
```

8.2 Dynamic Programming

8.2.0.1 Number of paths in a matrix with obstacles

Objetivo

É dada uma matriz, em que obstáculos são representados com o número 1, e posições livres com 0. Sabendo que só é possível cominhar para baixo e para a direita, determinar a quantidade de caminhos possíveis do canto superior esquerdo ao canto inferior direito.

Estratégia

Cada posição só pode ser acessada a partir de sua adjacente superior e sua adjacente esquerda. Assim, a quantidade de caminhos até uma posição é determinada pela soma dos caminhos de suas posições adjacentes. As primeiras linha e coluna são preenchidas com 1 até que se encontre um obstáculo, e as demais seguem a relação mencionada. $O(n^2)$

Código

```
1
2  int paths(const vector<vector<int>> &A){
3      int R = A.size();
4      int C = A[0].size();
5          vector<vector<int>> paths(R, vector<int>(C, 0));
6
7      paths[0][0] = !A[0][0];
8      FOR(i,1,R){
9          if(A[i][0]) break;
10         paths[i][0] = paths[i-1][0];
11     }
12     FOR(j,1,C){
13         if(A[0][j]) break;
14         paths[0][j] = paths[0][j-1];
```

```
15      }
16
17      FOR(i,1,R)
18          FOR(j,1,C)
19              if(!A[i][j]) paths[i][j] = paths[i-1][j] + paths[i][j-1];
20
21          return paths[R-1][C-1];
22  }
```

8.2.0.2 Uma mesa de doces

Objetivo

Uma mesa de doces é representada por uma matriz. Cada célula pode ter um valor maior ou igual a 0 (que indica quantos doces ela possui). Enquanto houver doces na mesa, é possível retirá-los. Porém, se os doces de uma posição forem retirados, todos os doces das linhas superior e inferior a ela desaparecem, assim como os de seus adjacentes laterais. O **objetivo** é descobrir a quantidade máxima de doces que pode ser obtida a partir de uma mesa qualquer.

Estratégia

Fundamentalmente, a matriz é reduzida a uma coluna, que é reduzida ao resultado. Para isso, dado que duas células adjacentes são mutuamente exclusivas, é feita uma escolha entre elas (sendo a primeira célula auxiliada pela não adjacente mais próxima na linha ou coluna). $O(m * n)$

Código

```
1
2   /* In original problem, the largest dimension is 10^5 */
3   #define MAX 100001
4
5   uint n, m;
6   uint dp[MAX];
7   uint row[MAX];
8   uint col[MAX];
9
10  uint reduce(uint i, uint j, uint *t){
11    if(i >= j) return 0;
12    if(dp[i]) return dp[i];
13    return dp[i] = max(t[i] + reduce(i+2,j,t), reduce(i+1,j,t));
14  }
15
16  void solve(){
17    while(true){
18      scanf("␣%d␣%d␣", &m, &n);
19      if(!m) break;
20      FOR(i,0,m){
21        FOR(j,0,n) scanf("␣%d␣", &row[j]);
22        fill(dp, dp+n, 0);
23        col[i] = reduce(0,n,row);
24      }
25      fill(dp,dp+m,0);
26      printf("%d\n", reduce(0,m,col));
27    }
28  }
```

8.2.0.3 Word break problem

Objetivo

Dada uma string de entrada e um dicionário de palavras, descubra se a string de entrada pode ser segmentada em uma sequência separada por espaços de palavras do dicionário.

Estratégia

TODO $O(n^2)$

Código

```
1
2  int dictionaryContains(string word)
3  {
4      string dictionary[]
5          = { "mobile", "samsung",  "sam",  "sung", "man",
6              "mango",  "icecream", "and",  "go",   "i",
7              "like",   "ice",      "cream" };
8      int size = sizeof(dictionary) / sizeof(dictionary[0]);
9      for (int i = 0; i < size; i++)
10         if (dictionary[i].compare(word) == 0)
11             return true;
12     return false;
13 }
14
15 // Returns true if string can be segmented into space
16 // separated words, otherwise returns false
17 bool wordBreak(string s)
18 {
19     int n = s.size();
20     if (n == 0)
21         return true;
22
23     vector<bool> dp(n + 1, 0); // Initialize all values
24
25     vector<int> matched_index;
26     matched_index.push_back(-1);
27
28     for (int i = 0; i < n; i++) {
29         int msize = matched_index.size();
30
31         int f = 0;
32
33         for (int j = msize - 1; j >= 0; j--) {
34
35             string sb = s.substr(matched_index[j] + 1,
36                                  i - matched_index[j]);
37
38             if (dictionaryContains(sb)) {
39                 f = 1;
40                 break;
41             }
```

```
42          }
43
44          if (f == 1) {
45              dp[i] = 1;
46              matched_index.push_back(i);
47          }
48      }
49      return dp[n - 1];
50  }
```

8.2.0.4 Word break problem

Objetivo

A seguir está uma descrição da instância deste famoso quebra-cabeça envolvendo N = 2 ovos e um prédio com K = 36 andares. Suponha que desejamos saber de quais andares em um prédio de 36 andares é seguro jogar ovos e quais farão com que os ovos quebrem ao pousar.

Estratégia

TODO $O(n^2)$

Código

```
1
2  int minTrials(int n, int k){
3      vector<vector<int> > dp(k + 1, vector<int>(n + 1, 0));
4      int m = 0; // Number of moves
5      while (dp[m][n] < k) {
6          m++;
7          for (int x = 1; x <= n; x++) {
8              dp[m][x] = 1 + dp[m - 1][x - 1] + dp[m - 1][x];
9          }
10     }
11     return m;
12 }
```

8.2.0.5 Empilhando infinitas barras

Objetivo

É preciso construir uma torre com n barras. Essa torre precisa ser estável, ou seja, se uma barra está em cima de outra, o tamanho da barra inferior é maior ou igual ao tamanho da barra superior. Temos à disposição infinitas barras de todos os tamanhos $0 \leq k \leq m$, e é preciso descobrir o número total de possibilidades de construção.

Estratégia

TODO $O(n^2)$

Código

```
1
2  int tileStacking(int n, int m, int k) {
3      int dp[N][N];
4      int presum[N][N];
5      memset(dp, 0, sizeof dp);
6      memset(presum, 0, sizeof presum);
7
8      for (int i = 1; i < n + 1; i++) {
9          dp[0][i] = 0;
10         presum[0][i] = 1;
11     }
12
13     for (int i = 0; i < m + 1; i++)
14         presum[i][0] = dp[i][0] = 1;
15
16     for (int i = 1; i < m + 1; i++) {
17         for (int j = 1; j < n + 1; j++) {
18             dp[i][j] = presum[i - 1][j];
19             if (j > k) {
20                 dp[i][j] -= presum[i - 1][j - k - 1];
21             }
22         }
23         for (int j = 1; j < n + 1; j++)
24             presum[i][j] = dp[i][j] + presum[i][j - 1];
25     }
26
27     return dp[m][n];
28 }
```

8.2.0.6 Atravesando pessoas em uma ponte

Objetivo

Dado um array de inteiros positivos distintos que denotam o tempo de travessia de 'n' pessoas. Essas 'n' pessoas estão paradas em um lado da ponte. A ponte pode suportar no máximo duas pessoas por vez. Quando duas pessoas atravessam a ponte, elas devem se mover no ritmo da pessoa mais lenta. Encontre o tempo total mínimo em que todas as pessoas podem atravessar a ponte.

Estratégia

TODO $O(n^2)$

Código

```
1
2  int dp[1 << 20][2];
3
4  int findMinTime(int leftmask, bool turn, int arr[], int& n) {
5      if (!leftmask) return 0;
6      int& res = dp[leftmask][turn];
7      if (~res) return res;
8      int rightmask = ((1 << n) - 1) ^ leftmask;
```

```
9
10      if (turn == 1) {
11          int minRow = INT_MAX, person;
12          for (int i = 0; i < n; ++i)
13              if (rightmask & (1 << i)) {
14                  if (minRow > arr[i]) {
15                      person = i;
16                      minRow = arr[i];
17                  }
18              }
19          res = arr[person]
20                + findMinTime(leftmask | (1 << person),
21                              turn ^ 1, arr, n);
22      }
23      else {
24          if (__builtin_popcount(leftmask) == 1) {
25              for (int i = 0; i < n; ++i)
26                  if (leftmask & (1 << i)) {
27                      res = arr[i];
28                      break;
29                  }
30          }
31          else {
32              res = INT_MAX;
33              for (int i = 0; i < n; ++i) {
34                  if (!(leftmask & (1 << i))) continue;
35                  for (int j = i + 1; j < n; ++j)
36                      if (leftmask & (1 << j)) {
37                          int val = max(arr[i], arr[j]);
38                          val += findMinTime(
39                              leftmask ^ (1 << i) ^ (1 << j),
40                              turn ^ 1, arr, n);
41                          res = min(res, val);
42                      }
43              }
44          }
45      }
46      return res;
47  }
48
49  int findTime(int arr[], int n)
50  {
51      int mask = (1 << n) - 1;
52      memset(dp, -1, sizeof(dp));
53      return findMinTime(mask, 0, arr, n);
54  }
```

8.2.0.7 Number of paths in a matrix with obstacles

Objetivo

Sua tarefa é contar o número de maneiras de construir a soma n jogando um dado uma ou mais vezes. Cada lançamento produz um resultado entre 1 e 6.

Estratégia

Exemplo para quando a soma é 5:

- Dado da um e compute o número de maneiras de obter o número 4
- Dado da 2 e compute o número de maneiras de obter o número 3
- Dado da 3 e compute o número de maneiras de obter o número 2
- Dado da 4 e compute o número de maneiras de colocar o último (que é apenas um)
- Dado da 5 e está feito

Portanto, se c é nossa função, podemos dizer o seguinte:

$$c(5) = 1 + c(4) + c(3) + c(2) + c(1)$$

$$c(4) = 1 + c(3) + c(2) + c(1)$$

$$c(3) = 1 + c(2) + c(1)$$

$$c(2) = 1 + c(1)$$

$$c(1) = c(0) = 1$$

Veja o número de vezes que temos que computar os mesmos valores repetidamente (por exemplo, $c(2)$ ou $c(3)$). Em vez disso, podemos usar programação dinâmica para nos ajudar. Se tivermos um array de tamanho N para armazenar cada um dos valores, só precisamos computá-los uma vez. Depois de computá-los, nós os armazenamos e acessamos os valores em tempo constante.

Código

```
1  int DP[MAXN];
2
3  int conput(int left){
4      if(DP[left] != 0){
5          return DP[left];
6      }
7      for(int i=1; i<=6; i++){
8          if(left-i >= 0){
9              DP[left] += conput(left - i);
10             DP[left] %= MOD;
11         }
12     }
13     return DP[left];
14 }
15
16 int main(){
17
18     long long n;
19     memset(DP, 0, sizeof(DP));
20     DP[0] = 1;
21
22     cin >> n;
23
24     cout << conput(n) << "\n";
25
26     return 0;
27 }
```

8.2.0.8 Minimizing Coins

Objetivo

Considere um sistema monetário consistindo de n moedas. Cada moeda tem um valor inteiro positivo. Sua tarefa é produzir uma soma de dinheiro x usando as moedas disponíveis de tal forma que o número de moedas seja mínimo. Por exemplo, se as moedas são $\{1, 5, 7\}$ e a soma desejada é 11, uma solução ótima é 5+5+1, que requer 3 moedas.

Estratégia

Use Minimizing Coins.

Código

```
// m is size of coins array (number of different coins)
int minCoins(vector<ll> coins, int m, int V)
{
    // table[i] will be storing the minimum number of coins
    // required for i value.  So table[V] will have result
    int table[V + 1];

    // Base case (If given value V is 0)
    table[0] = 0;

    // Initialize all table values as Infinite
    for (int i = 1; i <= V; i++)
        table[i] = INT_MAX;

    // Compute minimum coins required for all
    // values from 1 to V
    for (int i = 1; i <= V; i++) {
        // Go through all coins smaller than i
        for (int j = 0; j < m; j++)
            if (coins[j] <= i) {
                int sub_res = table[i - coins[j]];
                if (sub_res != INT_MAX
                    && sub_res + 1 < table[i])
                    table[i] = sub_res + 1;
            }
    }

    if (table[V] == INT_MAX)
        return -1;

    return table[V];
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    ll n, x;
    cin >> n >> x;
    vector<ll> coins(n);
```

```
41
42      for(ll i=0; i<n; i++) cin >> coins[i];
43
44          ll res = minCoins(coins, coins.size(), x);
45
46          cout << res << "\n";
47
48          return 0;
49  }
```

8.2.0.9 Coin Combinations I

Objetivo

Considere um sistema monetário consistindo de n moedas. Cada moeda tem um valor inteiro positivo. Sua tarefa é calcular o número de maneiras distintas de produzir uma soma de dinheiro x usando as moedas disponíveis.

Estratégia

Use Coin change problem.

Código

```
1   int DP[MAXN];
2   int n;
3
4   int main(){
5       ios::sync_with_stdio(false);
6       cin.tie(NULL);
7
8       ll x;
9       cin >> n >> x;
10      int NUM[n];
11      for(int i=0; i<n; i++) cin >> NUM[i];
12
13      // memset(DP, 0, sizeof(DP));
14      DP[0] = 1;
15
16      for(int i=1; i<=x; i++)
17          for(int j=0; j<n; j++)
18              if(i - NUM[j] >=0){
19                  DP[i] += DP[i-NUM[j]];
20                  DP[i] %= MOD;
21              }
22
23      cout << DP[x] << "\n";
24
25      return 0;
26  }
```

8.2.0.10 Coin Combinations II

Objetivo

Considere um sistema monetário consistindo de n moedas. Cada moeda tem um valor inteiro positivo. Sua tarefa é calcular o número de maneiras ordenadas distintas de produzir uma soma de dinheiro x usando as moedas disponíveis.

Estratégia

Variação do Coin change problem onde trocamos a ordem dos for, iterando primeiro por cada moeda e depois por cada valor.

Código

```
int DP[MAXN];
int n;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    ll x;
    cin >> n >> x;
    int NUM[n];
    for(int i=0; i<n; i++) cin >> NUM[i];

    // memset(DP, 0, sizeof(DP));
    DP[0] = 1;

    for(int j=0; j<n; j++)
        for(int i=1; i<=x; i++)
            if(i - NUM[j] >=0){
                DP[i] += DP[i-NUM[j]];
                DP[i] %= MOD;
            }

    cout << DP[x] << "\n";

    return 0;
}
```

8.2.0.11 Removing Digits

Objetivo

Você recebe um número inteiro n. Em cada etapa, você pode subtrair um dos dígitos do número. Quantos passos são necessários para tornar o número igual a 0?

Estratégia

construa a DP de 0 até N, onde a DP[N] é o número de operações para deixar N = 0.

DP[0] = 0; DP[N] = DP[N-MAXDIGIT(N)] + 1

Código

```
int main(){

    int n, max_d;
    cin >> n;
    int DP[n+1];
    DP[0] = 0;

    for(int i=1; i<=n; i++){
        max_d = 0;
        for(auto numero: to_string(i))
            if(max_d<(numero-'0')) max_d = numero-'0';
        DP[i] = DP[i-max_d]+1;
    }

    cout << DP[n] << "\n";

    return 0;
}
```

8.2.0.12 Grid Paths

Objetivo

Considere uma grade $n \times n$ cujos quadrados podem conter armadilhas. Não é permitido mover-se para um quadrado com armadilha. Sua tarefa é calcular o número de caminhos do quadrado superior esquerdo ao quadrado inferior direito. Você só pode se mover para a direita ou para baixo.

Estratégia

começãndo de DP[n][n]=1 faça, DP[i][j] = DP[i+1][j]+DP[i][j+1] se Matriz[i][j] não for um obstaculo.

Código

```
int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    ll n;
    cin >> n ;
    vector<vector<bool>> matriz(n,vector<bool>(n));
    vector<vector<int>> DP(n+1,vector<int>(n+1, 0));

    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            char a; cin >> a;
            if(a == '.') matriz[i][j] = true;
            else matriz[i][j] = false;
        }
    }

    if(!matriz[n-1][n-1]){
        cout << "0\n";
```

```
20          return 0;
21      }
22
23      for(int j=n-1; j>=0; j--){
24          for(int i=n-1; i>=0; i--){
25              if(i==n-1 && j==n-1){ DP[i][j] = 1; continue; }
26
27              if(matriz[i][j]){
28                  DP[i][j] = DP[i][j+1] + DP[i+1][j];
29                  DP[i][j] %= MOD;
30              }
31          }
32      }
33
34      cout << DP[0][0] << "\n";
35
36      return 0;
37  }
```

8.2.0.13 Book Shop

Objetivo

Você está em uma livraria que vende n livros diferentes. Você conhece o preço e o número de páginas de cada livro. Você decidiu que o preço total de suas compras será no máximo x. Qual é o número máximo de páginas que você pode comprar? Você pode comprar cada livro no máximo uma vez.

Estratégia

Problema da mochila

Código

```
1   int knapSack(int W, int wt[], int val[], int n) {
2       // Making and initializing dp array
3       int dp[W + 1];
4       memset(dp, 0, sizeof(dp));
5
6       for (int i = 1; i < n + 1; i++) {
7           for (int w = W; w >= 0; w--) {
8
9               if (wt[i - 1] <= w)
10
11                  // Finding the maximum value
12                  dp[w] = max(dp[w],
13                              dp[w - wt[i - 1]] + val[i - 1]);
14          }
15      }
16      // Returning the maximum value of knapsack
17      return dp[W];
18  }
19
20  // Driver code
21  int main(){
```

```
22      int n, W;
23      cin >> n >> W;
24
25      int profit[n], weight[n];
26      for(int i=0; i<n; i++) cin >> weight[i];
27      for(int i=0; i<n; i++) cin >> profit[i];
28
29      cout << knapSack(W, weight, profit, n) << "\n";
30      return 0;
31  }
```

8.2.0.14 Array Description

Objetivo

Você sabe que uma matriz possui n inteiros entre 1 e m, e a diferença absoluta entre dois valores adjacentes é no máximo 1. Dada uma descrição da matriz onde alguns valores podem ser desconhecidos, sua tarefa é contar o número de matrizes que correspondem à descrição.

Estratégia

Mantenha uma matriz dp[][], de modo que dp$[i][j]$ armazene o número de maneiras de ter arr$[i] = j$.

Inicialmente, vamos nos concentrar no primeiro índice, $i = 0$. Portanto, existem 2 valores possíveis para arr$[0]$.

Se arr$[0] = 0$, então podemos ter arr$[0]$ como qualquer valor de 1 a M. Portanto, para todos os valores j de 1 a M, só há uma maneira de ter arr$[i] = j$. Portanto, para todos os valores de j de 1 a M, dp$[0][j] = 1$.

Se arr$[0] \neq 0$, então significa que arr$[0]$ já tem um valor e não podemos colocar nenhum outro valor no primeiro índice. Portanto, temos dp$[0]$[arr$[0]] = 1$.

Agora, para todos os índices i de 1 a N, novamente existem dois valores possíveis de arr$[i]$.

Se arr$[i] = 0$, então o número de maneiras pelas quais podemos ter arr$[i] =$ val é a soma do número de maneiras pelas quais podemos ter arr$[i–1] =$ val–1, arr$[i - 1] =$ val e arr$[i–1] =$ val $+ 1$. Portanto, para todos os valores de j de 1 a M, dp$[i][j] =$ dp$[i–1][j–1] +$ dp$[i–1][j] +$ dp$[i–1][j + 1]$.

Se arr$[i] \neq 0$, então significa que arr$[i]$ já tem um valor e não podemos colocar nenhum outro valor no índice i. Portanto, o número de maneiras será a soma do número de maneiras pelas quais podemos ter arr$[i–1] =$ arr$[i]$–1, arr$[i - 1] =$ arr$[i]$ e arr$[i–1] =$ arr$[i] + 1$. Portanto, temos dp$[i]$[arr$[i]]$ = dp$[i–1]$[arr$[i]$–1] + dp$[i–1]$[arr$[i]] +$ dp$[i–1]$[arr$[i] + 1]$. A resposta final será a soma do número de maneiras pelas quais podemos colocar qualquer valor j no último índice, ou seja, a soma de dp$[N–1][j]$ para todos j de 1 a M.

Código

```
1  // function to find number of possible arrays that satisfy
2  // the description
3  ll getans(vector<ll>& arr, ll N, ll M,
4            vector<vector<ll> >& dp)
5  {
6      // Iterate over all indices from 0 to N - 1
7      for (int i = 0; i < N; i++) {
8          if (i == 0) {
9              // If we are at the first index and the value is
10             // unknown
```

```
11              if (arr[i] == 0) {
12                  // There is only one way to put any value at
13                  // the first index
14                  for (int val = 1; val <= M; val++)
15                      dp[i][val] = 1;
16              }
17              else {
18                  // If we are at the first index and the
19                  // value is fixed, there is only one way to
20                  // put the value arr[0] at 0th index
21                  dp[i][arr[i]] = 1;
22              }
23          }
24          else {
25              // If we are not at the first index and the
26              // value is unknown, then for each val from 1 to
27              // M, the number of ways to put val are equal to
28              // the number of ways we can put val-1, val,
29              // val+1 at the previous index
30              if (arr[i] == 0) {
31                  for (int val = 1; val <= M; val++) {
32                      dp[i][val] = (dp[i - 1][val - 1]
33                                    + dp[i - 1][val]
34                                    + dp[i - 1][val + 1])
35                                    % mod;
36                  }
37              }
38              // If we are not at the first index and the
39              // value is known, then the number of ways to
40              // put arr[i] are equal to the number of ways we
41              // can put arr[i]-1, arr[i], arr[i]+1 at the
42              // previous index
43              else {
44                  dp[i][arr[i]] = (dp[i - 1][arr[i] - 1]
45                                  + dp[i - 1][arr[i]]
46                                  + dp[i - 1][arr[i] + 1]) % mod;
47              }
48          }
49      }
50
51      // Variable to store the final answer
52      ll ans = 0;
53
54      // Sum the number of ways of putting any value in the
55      // last index to get the final answer
56      for (int val = 1; val <= M; val++) {
57          ans = (ans + dp[N - 1][val]) % mod;
58      }
59
60      return ans;
61  }
62  int main()
63  {
64      // Sample Input
65      ll N = 3, M = 5;
```

```
66      vector<ll> arr = {2, 0, 2};
67
68      // dp[][] array such that dp[i][j] stores the number of
69      // arrays that have arr[i] = j
70      vector<vector<ll> > dp(N, vector<ll>(M + 2, 0));
71      cout << getans(arr, N, M, dp) << endl;
72  }
```

8.2.0.15 Edit Distance

Objetivo

A distância de edição entre duas strings é o número mínimo de operações necessárias para transformar uma string na outra.

As operações permitidas são:

- Adicionar um caractere à string.
- Remover um caractere da string.
- Substituir um caractere na string.

Por exemplo, a distância de edição entre LOVE e MOVIE é 2, porque você pode primeiro substituir L por M e depois adicionar I.

Sua tarefa é calcular a distância de edição entre duas strings.

Estratégia

Use Levenshtein.

Código

```
1  int levenshteinTwoMatrixRows(const string& str1, const string& str2){
2          int m = str1.length();
3          int n = str2.length();
4
5          vector<int> prevRow(n + 1, 0);
6          vector<int> currRow(n + 1, 0);
7
8          for (int j = 0; j <= n; j++) prevRow[j] = j;
9
10
11         for (int i = 1; i <= m; i++){
12                 currRow[0] = i;
13
14                 for (int j = 1; j <= n; j++){
15                         if (str1[i - 1] == str2[j - 1]){
16                 currRow[j] = prevRow[j - 1];
17             }
18                         else{
19                                 currRow[j] = 1
20                                                         + min(
21
22                                                                 // Insert
```

```
23                                                                  currRow[j - 1],
24                                                                  min(
25
26                                                                          // Remove
27                                                                          prevRow[j],
28
29                                                                          // Replace
30                                                                          prevRow[j - 1]));
31                              }
32                  }
33
34                  prevRow = currRow;
35          }
36
37          return currRow[n];
38  }
39
40  // Drivers code
41  int main()
42  {
43          string str1, str2;
44      cin >> str1 >> str2;
45
46          int distance = levenshteinTwoMatrixRows(str1, str2);
47          cout << distance << "\n";
48          return 0;
49  }
```

8.2.0.16 Rectangle Cutting

Objetivo

Dado um retângulo de tamanho $a \times b$, sua tarefa é cortá-lo em quadrados. Em cada movimento, você pode selecionar um retângulo e cortá-lo em dois retângulos de forma que todos os comprimentos dos lados permaneçam inteiros. Qual é o número mínimo possível de movimentos?

Estratégia

Mantenha um array dp[][] tal que dp$[i][j]$ armazene o número mínimo de cortes necessários para cortar um retângulo $(i \times j)$ em quadrados. Para um retângulo $(i \times j)$, podem haver dois casos:

Caso 1: Se $i == j$, então o retângulo já é um quadrado. Então, dp$[i][j] = 0$.

Caso 2: Se $i \neq j$, então precisamos fazer o corte horizontalmente ou verticalmente. Se fizermos o corte horizontalmente, podemos cortar em qualquer posição $1, 2, \ldots, i-1$. Se cortarmos na posição k, então ficamos com duas peças de tamanhos $k \times j$ e $(i-k) \times j$. Agora, podemos consultar o número de movimentos para reduzir essas peças a quadrados no array dp[][]. Podemos iterar sobre todos os valores possíveis de k e encontrar o corte que requer o número mínimo de movimentos, isto é,

$$\text{dp}[i][j] = \min(\text{dp}[i][j], \text{dp}[i][k] + \text{dp}[i][j-k] + 1)$$

De forma semelhante, se fizermos o corte verticalmente, podemos cortar em qualquer posição $1, 2, \ldots, j-1$. Se cortarmos na posição k, então ficamos com duas peças de tamanhos $i \times k$ e $i \times (j-k)$. Agora, podemos consultar o

número de movimentos para reduzir essas peças a quadrados no array dp[][]. Podemos iterar sobre todos os valores possíveis de k e encontrar o corte que requer o número mínimo de movimentos, isto é,

$$\text{dp}[i][j] = \min(\text{dp}[i][j], \text{dp}[k][j] + \text{dp}[i-k][j] + 1)$$

Após considerar todos os cortes possíveis, dp$[A][B]$ armazena a resposta final.

Código

```
int  main(){

    int a, b;
    cin >> a >> b;
    vector<vector<int>> dp(a+1, vector<int>(b+1, 0));

    for(int i=1; i<=a; i++){
        for(int j=1; j<=b; j++){
            if(i==j){
                dp[i][j] == 0;
                continue;
            }

            int min_e = INT_MAX;
            for(int k=1; k<i; k++) min_e = min(dp[i-k][j]+dp[k][j], min_e);
            for(int k=1; k<j; k++) min_e = min(dp[i][j-k]+dp[i][k], min_e);

            dp[i][j] = 1+min_e;
        }
    }

    cout  << dp[a][b] << "\n";
}
```

8.2.0.17 Money Sums

Objetivo

Você tem n moedas com determinados valores. Sua tarefa é encontrar todas as quantias de dinheiro que você pode criar usando essas moedas.

Estratégia

Variação do Minimizing Coins em 2d

Código

```
bool DP[102][MAXN];
int n;

int main(){
    ios::sync_with_stdio(false);
    cin.tie(NULL);

```

```
8       ll sum=0, ans=0;
9       cin >> n;
10      int NUM[n+1];
11      for(int i=1; i<=n; i++){
12          cin >> NUM[i];
13          sum += NUM[i];
14      }
15
16      // memset(DP, 0, sizeof(DP));
17      DP[0][0] = true;
18
19      for(int i=1; i<=n; i++){ //para cada moeda (i
20                               //tem a quantidadede moedas)
21          for(int j=0; j<=sum; j++){ //para cada soma
22              DP[i][j] = DP[i-1][j];
23
24              if(j-NUM[i]>=0 && DP[i-1][j-NUM[i]]){
25                  DP[i][j] = 1;
26              }
27
28              if(i==n && DP[i][j] && j){
29                  ans++;
30                  DP[101][j] = true;
31              }
32          }
33      }
34
35      cout << ans << "\n";
36      for(int j=1; j<=sum; j++){
37          if(DP[101][j]){
38              cout << j << "␣";
39          }
40      }
41
42      cout << "\n";
43
44      return 0;
45  }
```

8.2.0.18 Removal Game

Objetivo

Há uma lista de n números e dois jogadores que jogam alternadamente. Em cada jogada, um jogador remove o primeiro ou o último número da lista, e sua pontuação aumenta por esse número. Ambos os jogadores tentam maximizar suas pontuações. Qual é a pontuação máxima possível para o primeiro jogador quando ambos jogam de forma otimizada?

Estratégia

Código

```
1   int main(){
2
```

```
3       ll n;
4       cin >> n;
5
6       vector<ll> num(n), sum(n+1, 0);
7       vector<vector<ll>> dp(n, vector<ll>(n, 0));
8       for(int i=0;i<n;i++){
9           cin >> num[i];
10          sum[i+1]= sum[i]+num[i];
11          dp[i][i] = num[i];
12      }
13
14      for(int i=n-1; i>=0; i--){
15          for(int j=i+1; j<n; j++){
16
17              ll select1=0, select2=0, total=sum[j+1]-sum[i];
18
19              if(i+1<=j) select1 = total - dp[i+1][j];
20              if(i<=j-1) select2 = total - dp[i][j-1];
21
22              dp[i][j] = max(select1, select2);
23          }
24      }
25
26      cout << dp[0][n-1] << "\n";
27
28      return 0;
29  }
```

8.2.0.19 Two Sets II

Objetivo

Sua tarefa é contar o número de maneiras que os números $1, 2, \ldots, n$ podem ser divididos em dois conjuntos de soma igual.

Estratégia

Modificação do money_sums. considera cada numero uma moeda.

Código

```
1   int DP[501][MAXN];
2   int n;
3
4   int main(){
5       ios::sync_with_stdio(false);
6       cin.tie(NULL);
7
8       ll x=0;
9       cin >> n;
10      ll NUM[n];
11      for(ll i=0; i<n; i++){
12          NUM[i] = i+1;
13          x+=i+1;
```

```
14      }
15
16      if(x\%2==1){
17          cout << 0 << "\n";
18          return 0;
19      }
20
21      x = x/2;
22      DP[0][0] = 1;
23      // for(ll i=0; i<=n; i++) DP[i][0] = 1;
24
25      for(ll j=1; j<=n; j++)
26          for(ll i=0; i<=x; i++){
27              DP[j][i] = DP[j-1][i];
28              if(i - NUM[j-1] >=0){
29                  DP[j][i] += DP[j-1][i-NUM[j-1]];
30                  DP[j][i] \%= MOD;
31              }
32          }
33
34      if(DP[n][x]\%2 == 1) DP[n][x] += MOD; //Pra nao dar merda na arritimetica modular
35      cout << DP[n][x]/2 << "\n";
36
37      return 0;
38  }
```

8.2.0.20 Increasing Subsequence

Objetivo

Você recebe um array contendo n números inteiros. Sua tarefa é determinar a subsequência crescente mais longa no array, ou seja, a subsequência mais longa onde cada elemento é maior que o anterior. Uma subsequência é uma sequência que pode ser derivada do array deletando alguns elementos sem alterar a ordem dos elementos restantes.

Estratégia

LIS em $\log n$

Código

```
1   #define INF 1e9
2
3   using namespace std;
4
5   int main(){
6
7       ll n, num, att;
8       cin >> n;
9
10      vector<ll> dp(n+1, INF);
11      dp[0] = -INF;
12
13      for(int i=0; i<n; i++){
14          cin >> num;
```

```
15          att = upper_bound(dp.begin(), dp.end(), num) - dp.begin();
16          if(num>dp[att-1] && num<dp[att]) dp[att]=num;
17      }
18
19      for(int i=n; i>0; i--){
20          if(dp[i]!=INF){
21              cout << i << "\n";
22              break;
23          }
24      }
25
26
27      return 0;
28  }
```

8.2.0.21 Projects

Objetivo

Há n projetos que você pode participar. Para cada projeto, você sabe os dias de início e término e a quantia de dinheiro que você receberia como recompensa. Você só pode participar de um projeto por dia. Qual é a quantia máxima de dinheiro que você pode ganhar?

Estratégia

Embora os dias possam chegar até 10^9, só nos importamos com os dias em que começamos ou terminamos um projeto. Então, antes de fazer qualquer outra coisa, comprimimos todos os dias para seu índice entre todos os dias interessantes (ou seja, dias correspondentes a a_i ou $b_i + 1$ para algum i). Dessa forma, os dias variam de 0 a menos de $2n \leq 4 \cdot 10^5$.

dp$[i]$ = quantia máxima de dinheiro que podemos ganhar antes do dia i.

No dia i, talvez não fizemos nada, então ganhamos o que ganhamos no dia $i-1$, ou seja, dp$[i-1]$. Caso contrário, acabamos de terminar algum projeto. Ganhamos algum dinheiro realizando o projeto, e usamos dp[início do projeto] para saber quanto dinheiro poderíamos ter ganhado antes de começar o projeto. Percorremos todos os projetos terminando pouco antes do dia i e escolhemos o melhor.

A complexidade é $O(n \cdot \log n)$, o log vem da compressão dos dias.

Código

```
1
2   int minTrials(int n, int k){
3       vector<vector<int> > dp(k + 1, vector<int>(n + 1, 0));
4       int m = 0; // Number of moves
5       while (dp[m][n] < k) {
6           m++;
7           for (int x = 1; x <= n; x++) {
8               dp[m][x] = 1 + dp[m - 1][x - 1] + dp[m - 1][x];
9           }
10      }
11      return m;
12  }
```

8.2.0.22 Elevator Rides

Objetivo

Há n pessoas que querem chegar ao topo de um prédio que possui apenas um elevador. Você sabe o peso de cada pessoa e o peso máximo permitido no elevador. Qual é o número mínimo de viagens de elevador?

Estratégia

variação da mochila binaria, bin-packing

Código

```
int main(){

    ll n, x, limit;
    cin >> n >> x;
    vector<ll> pessoa(n);
    for(int i=0;i<n;i++) cin >> pessoa[i];

    limit = 1 << n;
    vector<pair<ll, ll>> dp(limit);
    dp[0] = {1, 0};

    for(int mask=1; mask<limit; mask++){
        pair<ll, ll> bestResult = {INT_MAX, INT_MAX};

        for(int i=0; i<n; i++){
            if(((1<<i)&mask) == 0) continue;

            auto res = dp[(1<<i)^mask];

            if(res.second + pessoa[i] <= x){
                res.second += pessoa[i];
            }
            else{
                res.first++;
                res.second = pessoa[i];
            }

            bestResult = min(bestResult, res);
        }

        dp[mask] = bestResult;
    }

    cout << dp[limit-1].first << "\n";

    return 0;
}
```

8.2.0.23 Counting Numbers

Objetivo

Sua tarefa é contar o número de inteiros entre a e b onde nenhum dos dígitos adjacentes é igual.

Estratégia

A ideia é usar Digit DP para resolver este problema. O array dp dp[n][prev_digit][leading_zero][tight] representa a contagem de números válidos (cujos dígitos adjacentes não são iguais) formados pelos n dígitos.

Os parâmetros podem ser descritos da seguinte maneira:

- **curr**: Representa a posição atual ou dígito sendo processado.
- **prev_digit**: Representa o dígito anterior na posição anterior.
- **leading_zero**: Indica se há um zero à esquerda no número atual (1 se houver um zero à esquerda, 0 caso contrário).
- **tight**: Indica se o número formado até agora é restrito (1 se for restrito, 0 caso contrário).

Durante a transição, o código itera através dos dígitos possíveis para a posição atual, verificando sua validade ao garantir que diferem do dígito anterior e, se aplicável, permitindo zeros consecutivos à esquerda. O estado é então atualizado chamando recursivamente a função `mem` para a próxima posição, ajustando flags como `leading_zero` e `tight` de acordo. A contagem de números válidos é acumulada considerando diferentes escolhas de dígitos, e a memoização é aplicada para otimizar cálculos repetitivos.

A resposta final será a diferença entre a contagem de números válidos de $[0, b]$ e $[0, a-1]$.

Código

```
ll dp[19];

ll solve(ll num){
    if(num == -1) return 0;

    string numero = to\_string(num);
    ll ans=1, sz = numero.size();

    dp[0] = 1;
    for(int i=1; i<sz;i++){
        dp[i] = dp[i-1]*9;
        ans += dp[i];
    }

    // cout << "numero real: " << numero  << " sz: " << sz << "\n";

    for(int i=0; i<sz; i++){

        ll algarismo = numero[i]-'0';
        if(i && numero[i-1]>=numero[i]) algarismo++;

        ans += (algarismo-1)*dp[sz-i-1];
        if(i && numero[i]==numero[i-1]) return ans;
    }
    ans++;
    // cout  << "\nans: "<<ans<<"\n\n";

    return ans;
```

```
29  }
30
31  int main(){
32
33      ll menor, maior;
34      cin >> menor >> maior;
35
36      cout << solve(maior) - solve(menor-1) << "\n";
37
38      return 0;
39  }
```

8.3 Grafos

8.3.1 Caminho mínimo

8.3.1.1 Caminho mínimo sem pesos

Objetivo

A rede de Syrjälä possui n computadores e m conexões. Sua tarefa é determinar se Uolevi pode enviar uma mensagem para Maija e, se for possível, qual é o número mínimo de computadores em tal rota.

Estratégia

Basta utilizar o algoritmo de BFS, que encontrará o menor caminho (em número de arestas) até o vértice final se for possível alcançá-lo.

Código

```
1   int n, m;
2   vector<int> parent;
3   vector<vector<int>> ng;
4
5   BEGIN
6   {
7       cin >> n >> m;
8       vector<vector<int>> ng(n+1, vector<int>());
9       vector<int> parent(n+1,0);
10
11      FOR(i,0,m) {
12          int u, v;
13          cin >> u >> v;
14          ng[v].push_back(u);
15          ng[u].push_back(v);
16      }
17
```

```
18      queue<int> Q;
19      int u;
20
21      Q.push(1);
22      while(Q.size()) {
23          u = Q.front(); Q.pop();
24          if(u == n) break;
25          for(int v : ng[u]) {
26              if(parent[v]) continue;
27              parent[v] = u;
28              Q.push(v);
29          }
30      }
31
32      if(u != n) {
33          cout << "IMPOSSIBLE" << endl;
34      }
35      else {
36          vector<int> path;
37          while(u != 1) {
38              path.push_back(u);
39              u = parent[u];
40          }
41          path.push_back(1);
42          cout << path.size() << endl;
43          for(auto it = path.rbegin(); it != path.rend(); it++)
44              cout << *it << '␣';
45          cout << endl;
46      }
47  }
48  END
```

8.3.1.2 Distância em arestas - quais pontos estão mais próximos da saída?

Objetivo

Você e alguns monstros estão em um labirinto. Ao dar um passo em qualquer direção no labirinto, cada monstro pode simultaneamente dar um passo também. Seu objetivo é alcançar uma das praças da borda sem nunca compartilhar uma praça com um monstro.

Sua tarefa é descobrir se seu objetivo é possível e, se for, imprimir um caminho que você possa seguir. Seu plano deve funcionar em qualquer situação; mesmo que os monstros saibam seu caminho antecipadamente.

Estratégia

A ideia principal é iniciar várias BFSs no grafo. Se, em alguma delas, o tempo de chegada do personagem em uma saída for menor que o tempo de chegada de algum monstro, o problema está resolvido.

Para reconstruir o caminho, o pai de cada vértice é indicado pela direção correspondente.

Código

```
int n, m, player;
vector<char> symbol, parent;
vector<int> discovered;
queue<int> Q;

void init()
{
    symbol = vector<char>(n*m);
    discovered = vector<int>(n*m,INT_MAX);
}

void queue_add(int v)
{
    discovered[v] = 0;
    Q.push(v);
}

int discover(bool monster)
{
    bool possible = false;
    parent.assign(n*m,0);
    int out;

    while(Q.size()) {
        int cur = Q.front();
        Q.pop();

        if(!monster && (
            cur % m == 0
            || cur % m == m-1
            || cur / m == 0
            || cur / m == n-1
        )) return cur;

        vector<pair<char,int>> ng;
        if(cur % m) ng.push_back({'L',cur-1});
        if(cur % m < m-1) ng.push_back({'R',cur+1});
        if(cur / m < n-1) ng.push_back({'D',cur+m});
        if(cur / n) ng.push_back({'U',cur-m});

        for(auto p: ng) {
            char dir = p.first;
            out = p.second;

            if(symbol[out] != '.' || out < 0 || out >= n*m
               || discovered[out] <= discovered[cur]+1) continue;

            parent[out] = dir;
            discovered[out] = discovered[cur]+1;
            Q.push(out);
        }
    }

    out:
    if(possible) return out;
```

```
56      return -1;
57  }
58
59
60  BEGIN
61  {
62      cin >> n >> m;
63      init();
64
65      FOR(i,0,n)
66          FOR(j,0,m) {
67              int v = m*i+j;
68              cin >> symbol[v];
69
70              switch(symbol[v]) {
71                  case '#': continue;
72                  case 'A': player = v; break;
73                  case 'M': queue_add(v);
74              }
75          }
76
77      discover(true);
78      queue_add(player);
79      int exit = discover(false);
80
81      if(exit == -1) cout << "NO\n";
82      else {
83          cout << "YES\n";
84          vector<char> path;
85
86          while(exit != player) {
87              path.push_back(parent[exit]);
88              switch (parent[exit]) {
89                  case 'L': exit += 1; break;
90                  case 'R': exit -= 1; break;
91                  case 'U': exit += m; break;
92                  case 'D': exit -= m; break;
93              }
94          }
95
96          cout << path.size() << endl;
97          for(auto it = path.rbegin(); it != path.rend(); it++)
98              cout << *it;
99          cout << endl;
100     }
101
102 }
103 END
```

8.3.1.3 Caminho mínimo de um para todos

Objetivo

Existem n cidades e m conexões de voo entre elas. Sua tarefa é determinar o comprimento da rota mais curta de Syrjälä para cada cidade.

Estratégia

Algoritmo de Dijkstra.

Código

```
template<class T> using min_heap =
    priority_queue<T, vector<T>, greater<T>>;

size_t n,m;
vector<vector<pair<int,long>>> adj;
min_heap<pair<long,int>> heap;
vector<long> dist;
vector<bool> closed;

BEGIN
{
    cin >> n >> m;
    adj.resize(n+1);
    dist.assign(n+1,LONG_MAX);
    closed.assign(n+1,false);
    FOR(i,0,m) {
        int a,b,c;
        cin >> a >> b >> c;
        adj[a].push_back({b,c});
    }

    dist[1] = 0;
    heap.push({0,1});
    while(heap.size()) {
        int u = heap.top().second;
        heap.pop();

        if(closed[u]) continue;
        closed[u] = true;

        for(auto edge : adj[u]) {
            int v = edge.first;
            long cost = edge.second;
            if(dist[v] > dist[u] + cost) {
                dist[v] = dist[u] + cost;
                heap.push({dist[v],v});
            }
        }
    }

    FOR(i,1,n+1) cout << dist[i] << '␣';
    cout << endl;
}
END
```

8.3.1.4 Caminho mínimo de todos para todos

Objetivo

Existem n cidades e m estradas entre elas. Sua tarefa é processar q consultas onde você tem que determinar o comprimento da rota mais curta entre duas cidades dadas.

Estratégia

Algoritmo Floyd Wharshall.

Código

```
BEGIN
{
    size_t n, m, q;
    cin >> n >> m >> q;

    vector<vector<long>> D(n+1,vector<long>(n+1,LONG_MAX));

    FOR(i,1,n+1) D[i][i] = 0;
    FOR(i,0,m) {
        int u, v, w;
        cin >> u >> v >> w;
        if(D[u][v] < w) continue;
        D[u][v] = D[v][u] = w;
    }

    FOR(k,1,n+1) FOR(i,1,n+1) FOR(j,1,n+1) {
        if(D[i][k] == LONG_MAX || D[k][j] == LONG_MAX) continue;
        D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
    }

    int a, b;
    FOR(i,0,q) {
        cin >> a >> b;
        cout << (D[a][b] == LONG_MAX ? -1 : D[a][b]) << endl;
    }


}
END
```

8.3.1.5 Caminho mínimo com redução de peso de uma aresta

Objetivo

Sua tarefa é encontrar a rota de voo de preço mínimo de Syrjälä para Metsälä. Você tem um cupom de desconto, com o qual pode reduzir pela metade o preço de um único voo durante a rota. No entanto, você só pode usar o cupom uma vez.

Quando você usa o cupom de desconto para um voo cujo preço é x, seu preço se torna $\lfloor x/2 \rfloor$.

Estratégia

- Calcular o caminho mínimo de 1 a todos os vértices.
- Calcular o caminho mínimo de n a todos os vértices.
- Para cada aresta (u, v), verificar $min_c(1, u) + \frac{w(u,v)}{2} + min_c(v, n)$, sendo o resultado o menor valor encontrado.

Código

```
template<class T> using min_heap =
    priority_queue<T, vector<T>, greater<T>>;

using adj_list = vector<vector<pair<int,int>>>;

size_t n,m;
adj_list child, parent;

vector<long> dijkstra(int src, const adj_list &adj)
{
    min_heap<pair<long,int>> heap;
    vector<long> dist(n+1,LONG_MAX);
    vector<bool> closed(n+1,false);

    dist[src] = 0;
    heap.push({0,src});
    while(heap.size()) {
        int u = heap.top().second;
        heap.pop();

        if(closed[u]) continue;
        closed[u] = true;

        for(auto edge : adj[u]) {
            int v = edge.first;
            long cost = edge.second;
            if(dist[v] > dist[u] + cost) {
                dist[v] = dist[u] + cost;
                heap.push({dist[v],v});
            }
        }
    }

    return dist;
}


BEGIN
{
    cin >> n >> m;
    child.resize(n+1);
    parent.resize(n+1);

    FOR(i,0,m) {
```

```
45          int a,b,c;
46          cin >> a >> b >> c;
47          child[a].push_back({b,c});
48          parent[b].push_back({a,c});
49      }
50
51      vector<long> src_dist = dijkstra(1,child);
52      vector<long> end_dist = dijkstra(n,parent);
53      long ans = LONG_MAX;
54
55      FOR(u,1,n+1) {
56          for(auto edge : child[u]) {
57              int v = edge.first;
58              long cost = edge.second;
59              if(src_dist[u] != LONG_MAX && end_dist[v] != LONG_MAX) {
60                  ans = min(ans, src_dist[u] + end_dist[v] + cost/2);
61              }
62          }
63      }
64
65      cout << ans << endl;
66  }
67  END
```

8.3.1.6 k caminhos mínimos

Objetivo

Sua tarefa é encontrar as k rotas aéreas mais curtas de Syrjälä para Metsälä. Uma rota pode visitar a mesma cidade várias vezes.

Observe que pode haver várias rotas com o mesmo preço e todas elas devem ser consideradas.

Estratégia

A base da solução é o algoritmo de Dijkstra. Contudo, os vértices só são fechados depois de analisados k vezes (e isso inclui o vértice de destino, ou seja, podem existir k caminhos chegando até ele). Isso permite que várias rotas sejam construídos simultaneamente na fila de prioridades, que garante que as k menores serão definidos antes dos demais.

Código

```
1   template<class T> using min_heap =
2       priority_queue<T, vector<T>, greater<T>>;
3   template<class T> using max_heap = priority_queue<T>;
4
5   BEGIN
6   {
7       int n,m,k;
8
9       cin >> n >> m >> k;
10
```

```
11      vector<vector<pair<int,long>>> adj(n+1);
12      vector<max_heap<long>> dist(n+1);
13
14      FOR(i,0,m) {
15          int a,b,c;
16          cin >> a >> b >> c;
17          adj[a].push_back({b,c});
18      }
19
20      min_heap<pair<long,int>> queue;
21      vector<int> vis(n+1,0);
22
23      queue.push({0,1});
24      while(queue.size() && vis[n] < k) {
25          int u = queue.top().second;
26          long du = queue.top().first;
27          queue.pop();
28
29          vis[u]++;
30          if(u == n) cout << du << '␣';
31          if(vis[u] <= k) {
32              for(auto edge : adj[u]) {
33                  int v = edge.first;
34                  int dv = edge.second;
35                  queue.push({du+dv,v});
36              }
37          }
38      }
39      cout << endl;
40
41  }
42  END
```

8.3.1.7 Caminho máximo

Objetivo

Você joga um jogo composto por n quartos e m túneis. Sua pontuação inicial é 0, e cada túnel aumenta sua pontuação em x, onde x pode ser tanto positivo quanto negativo. Você pode passar por um túnel várias vezes.

Sua tarefa é caminhar do quarto 1 até o quarto n. Qual é a pontuação máxima que você pode obter?

Estratégia

Todas as arestas têm sua pontuação multiplicada por -1, o que permite o uso do algoritmo de Bellman Ford para encontrar o ”custo mínimo”(que, com o peso oposto, será máximo).

Código

```
1  size_t n, m;
2  vector<vector<int>> parent;
```

```
3  vector<array<int,3>> edges;
4  vector<bool> changed, visited;
5  vector<long> d;
6
7  bool check(int v)
8  {
9      visited[v] = true;
10
11     if(changed[v]) return false;
12     for(int p : parent[v]) {
13         if(visited[p]) continue;
14         if(!check(p)) return false;
15     }
16     return true;
17 }
18
19 BEGIN
20 {
21     cin >> n >> m;
22     parent.assign(n+1,vector<int>());
23     visited.assign(n+1,false);
24     d.assign(n+1, INF);
25
26     FOR(i,0,m) {
27         int a,b,x;
28         cin >> a >> b >> x;
29         edges.push_back({a,b,-x});
30         parent[b].push_back(a);
31     }
32
33     bool any;
34     d[1] = 0;
35     FOR(i,0,n) {
36         changed.assign(n+1,0);
37         for(auto e : edges) {
38             if(d[e[0]] == INF) continue;
39             if(d[e[1]] > d[e[0]] + e[2]) {
40                 d[e[1]] = d[e[0]] + e[2];
41                 changed[e[1]] = 1;
42             }
43         }
44     }
45
46
47     if(!check(n)) d[n] = 1;
48
49     cout << -d[n] << endl;
50 }
51 END
```

8.3.1.8 Custo mínimo em rotas diferentes

Objetivo

Você vai viajar de Syrjälä para Lehmälä de avião. Você gostaria de encontrar respostas para as seguintes perguntas:

- qual é o preço mínimo de tal rota?
- quantas rotas de preço mínimo existem? (módulo $10^9 + 7$)
- qual é o número mínimo de voos em uma rota de preço mínimo?
- qual é o número máximo de voos em uma rota de preço mínimo?

Solução

A base da solução é o algoritmo de Dijkstra. Contudo, a cada vértice observado, além da possível atualização da fila de prioridades, é preciso atualizar os dados pedidos pelo enunciado.

Exemplo

```
template<typename T> using min_heap =
    priority_queue<T, vector<T>, greater<T>>;
using edge = pair<long,int>;

int n, m;
vector<edge> adj[MAXN];
min_heap<edge> proc;
vector<int>
    fcount(MAXN),
    min_hops(MAXN),
    max_hops(MAXN);
vector<long> dist(MAXN,LONG_MAX);
vector<bool> vis(MAXN,false);

void explore()
{
    proc.push({0,1});
    fcount[1] = 1;
    dist[1] = 0;

    while(proc.size()) {
        int u = proc.top().second;
        proc.pop();

        if(vis[u]) continue;
        vis[u] = true;

        for(edge e : adj[u]) {
            long dv = e.first;
            int v = e.second;

            if(dist[u]+dv < dist[v]) {
                dist[v] = dist[u]+dv;
                fcount[v] = fcount[u];
                min_hops[v] = min_hops[u]+1;
                max_hops[v] = max_hops[u]+1;

                proc.push({dist[v],v});
```

```
39              }
40              else if(dist[u]+dv == dist[v]) {
41                  fcount[v] = (fcount[v]+fcount[u])%MOD;
42                  min_hops[v] = min(min_hops[v], min_hops[u]+1);
43                  max_hops[v] = max(max_hops[v], max_hops[u]+1);
44              }
45          }
46      }
47  }
48
49  BEGIN
50  {
51      cin >> n >> m;
52      FOR(i,0,m) {
53          int a,b,c;
54          cin >> a >> b >> c;
55          adj[a].push_back({c,b});
56      }
57
58      explore();
59
60      cout << dist[n] << '␣' << fcount[n] << '␣'
61           << min_hops[n] << '␣' << max_hops[n] << endl;
62  }
63  END
```

8.3.2 Árvore geradora mínima

8.3.2.1 Conexões com o menor custo total

Objetivo

Há n cidades e m estradas entre elas. Infelizmente, a condição das estradas está tão ruim que elas não podem ser usadas. Sua tarefa é reparar algumas das estradas para que haja uma rota decente entre quaisquer duas cidades. Para cada estrada, você conhece o custo de reparação, e você deve encontrar uma solução onde o custo total seja o menor possível.

Estratégia

Basta aplica um algoritmo de MST.

Código

```
1  #define MAXN (int)(2e5+1)
2
3  template<class T> using min_heap =
4      priority_queue<T, vector<T>, greater<T>>;
5  using edge = pair<long,int>;
6
```

```
7  vector<bool> vis(MAXN,false);
8  vector<edge> adj[MAXN];
9  int n, m;
10
11 long solve() {
12     min_heap<edge> pq;
13     long min_cost = 0;
14
15     pq.push({0,1});
16     while(pq.size()) {
17         int u = pq.top().second;
18         long cost = pq.top().first;
19         pq.pop();
20
21         if(vis[u]) continue;
22
23         vis[u] = true;
24         min_cost += cost;
25         for(edge e: adj[u]) {
26             int v = e.second;
27             long dv = e.first;
28             pq.push({dv,v});
29         }
30     }
31
32     FOR(u,1,n+1) {
33         if(vis[u]) continue;
34         return 0;
35     }
36     return min_cost;
37 }
38
39
40 BEGIN
41 {
42     cin >> n >> m;
43     FOR(i,0,m) {
44         int a, b, c;
45         cin >> a >> b >> c;
46         adj[a].push_back({c,b});
47         adj[b].push_back({c,a});
48     }
49
50     long ans = solve();
51     if(ans) cout << ans << endl;
52     else cout << "IMPOSSIBLE\n";
53 }
54 END
```

8.3.3 Ciclos

8.3.3.1 Encontrando um ciclo com tamanho determinado

Objetivo

Byteland possui n cidades e m estradas entre elas. Sua tarefa é projetar uma viagem circular que começa em uma cidade, passa por duas ou mais outras cidades e finalmente retorna à cidade de partida. Todas as cidades intermediárias na rota devem ser distintas.

Estratégia

A resolução é baseada em DFS, com o diferencial de que os vértices são marcados por nível. Quando é encontrado um vértice já visitado (cujo nível já foi designado) e a diferença de níveis for maior ou iguala 2, um par resposta foi encontrado. A partir disso, basta reconstruir o caminho com base no vetor de antecessores atualizado pela própria busca.

Código

```
int u, v;
vector<vector<int>> G;
vector<int> level;
vector<int> parent;
bool possible = false;

pair<int,int> dfs(int i, int l = 1)
{
    level[i] = l;
    for(int v: G[i]) {
        if(level[v]) {
            if(level[i] - level[v] >= 2) {
                return make_pair(i,v);
            }
            continue;
        }
        parent[v] = i;
        pair<int,int> ans = dfs(v,l+1);
        if(ans != make_pair(0,0)) return ans;
    }
    return make_pair(0,0);
}

void solve(int src)
{
    if(parent[src]) return;

    vector<int> path;
    pair<int,int> ans = dfs(src);

    if(ans == make_pair(0,0)) return;

    path.push_back(ans.second);
    while(ans.first != ans.second) {
        path.push_back(ans.first);
        ans.first = parent[ans.first];
    }
```

```
38      path.push_back(ans.second);
39
40      cout << path.size() << endl;
41      for(int i: path) cout << i << '␣';
42      cout << endl;
43
44      possible = true;
45  }
46
47  BEGIN
48  {
49      int n, m;
50      cin >> n >> m;
51
52      G = vector<vector<int>>(n+1,vector<int>());
53      level = vector<int>(n+1,0);
54      parent = vector<int>(n+1,0);
55      FOR(i,0,m) {
56          cin >> u >> v;
57          G[v].push_back(u);
58          G[u].push_back(v);
59      }
60
61      FOR(i,0,n) {
62          if(possible) break;
63          solve(i);
64      }
65      if(!possible) cout << "IMPOSSIBLE\n";
66  }
67  END
```

8.3.3.2 Encontrando ciclo negativo

Objetivo

Você recebe um grafo direcionado e sua tarefa é descobrir se ele contém um ciclo negativo, e também fornecer um exemplo desse ciclo.

Estratégia

Basta usar o algoritmo de Bellman Ford. Se após $n-1$ iterações ainda há modificações de peso, então existe um ciclo, e basta usar o vetor de pais para reconstruí-lo.

Código

```
1  size_t n,m;
2  vector<vector<int>> adj;
3  vector<tuple<int,int,long>> edges;
4  vector<int> parent;
5  vector<long> dist;
6  vector<bool> vis;
```

```
 7
 8  int bellman()
 9  {
10      parent.assign(n+1,-1);
11      int x;
12
13      FOR(i,0,n) {
14          x = -1;
15          for(auto edge : edges) {
16              int u = get<0>(edge);
17              int v = get<1>(edge);
18              int cost = get<2>(edge);
19
20              if(dist[u] == LONG_MAX) continue;
21              if(dist[v] > dist[u] + cost) {
22                  dist[v] = dist[u]+cost;
23                  parent[v] = u;
24                  x = v;
25              }
26          }
27      }
28
29      return x;
30  }
31
32  void dfs(int u)
33  {
34      vis[u] = true;
35      for(int v: adj[u]) {
36          if(vis[v]) continue;
37          dfs(v);
38      }
39  }
40
41  BEGIN
42  {
43      cin >> n >> m;
44      adj.resize(n+1);
45      parent.resize(n+1);
46      dist.assign(n+1,LONG_MAX);
47      vis.assign(n+1,false);
48
49      FOR(i,0,m) {
50          int a,b,c;
51          cin >> a >> b >> c;
52          edges.push_back({a,b,c});
53          adj[a].push_back(b);
54      }
55
56      FOR(u,1,n+1) {
57          if(vis[u]) continue;
58          dist[u] = 0;
59          dfs(u);
60      }
61
```

```
62      int x = bellman();
63      if(x == -1) cout << "NO\n";
64      else {
65          cout << "YES\n";
66          int y = x;
67          FOR(i,0,n) y = parent[y];
68
69          vector<int> path;
70          for(int u = y;; u = parent[u]) {
71              path.push_back(u);
72              if(u == y && path.size() > 1) break;
73          }
74          reverse(ALL(path));
75
76          for(int u: path) cout << u << '␣';
77          cout << endl;
78      }
79  }
80  END
```

8.3.3.3 Encontrando um ciclo

Objetivo

Byteland possui n cidades e m conexões de voo. Sua tarefa é projetar uma viagem circular que começa em uma cidade, passa por uma ou mais outras cidades e finalmente retorna à cidade de partida. Cada cidade intermediária na rota deve ser distinta.

Estratégia

A resolução é baseada em DFS. São mantidos dois vetores, *visitados* e *habilitados*. Se é encontrado um vértice que já foi visitado, então o ciclo foi encontrado, bastando reconstruí-lo com base no pai de cada vértice. O segundo vetor é mantido para evitar que exista confusão entre buscas diferentes, sendo um vértice desabilitado apenas quando todo seu processamento em uma busca é totalmente terminado.

Código

```
1   int n, m, k;
2   vector<vector<int>> adj;
3   vector<bool> vis, able;
4   vector<int> p;
5
6   int dfs(int u)
7   {
8       vis[u] = true;
9       int x;
10      for(int v : adj[u]) {
11          if(!able[v]) continue;
12          p[v] = u;
13          if(vis[v]) return v;
14          if((x = dfs(v)) != -1) return x;
```

```
15      }
16      able[u] = false;
17      return -1;
18  }
19
20  BEGIN
21  {
22      cin >> n >> m;
23      adj.resize(n+1);
24      vis.assign(n+1,false);
25      able.assign(n+1,true);
26      p.resize(n+1);
27
28      FOR(i,0,m) {
29          int a, b;
30          cin >> a >> b;
31          adj[a].push_back(b);
32      }
33
34      int x;
35      FOR(v,0,n) {
36          if(!able[v]) continue;
37          if((x = dfs(v)) == -1) continue;
38
39          vector<int> path {x};
40          for(int y = x;
41              path.size() == 1 || y != x;
42              path.push_back(y = p[y]));
43          reverse(ALL(path));
44
45          cout << path.size() << endl;
46          for(int v : path) cout << v << '␣';
47          cout << endl;
48
49          return 0;
50      }
51
52      cout << "IMPOSSIBLE\n";
53  }
54  END
```

8.3.3.4 Tamanho de ciclos

Objetivo

Você está jogando um jogo composto por n planetas. Cada planeta possui um teletransportador para outro planeta (ou para o próprio planeta). Você começa em um planeta e, em seguida, viaja através dos teletransportadores até chegar a um planeta que você já visitou anteriormente. Sua tarefa é calcular para cada planeta o número de teletransportações que haveria se você começasse naquele planeta.

Estratégia

Para encontrar a solução de forma eficiente (evitando a execução de várias buscas), são desenvolvidas algumas rotinas:

1. os ciclos são encontrados, e cada vértice é ligado a algum deles
2. é determinado o tamanho de cada ciclo
3. para a resposta, tem-se dois casos:
    - **o vértice está no próprio ciclo**: a resposta é justamente o tamanho do ciclo
    - **o vértice está em uma árvore anexa a um ciclo**: a resposta é a soma da distância até o ciclo mais o tamanho dele mesmo

Código

```
#define MAXN (int)(2e5+1)

int n,q;
vector<bool> vis(MAXN,false);
static vector<int>
    succ(MAXN),
    cycle(MAXN),
    cycle_len(MAXN),
    ans(MAXN);

int find_cycle(int u) {
    if(vis[u]) {
        if(cycle[u]) return cycle[u];
        return cycle[u] = u;
    }
    vis[u] = true;
    return cycle[u] = find_cycle(succ[u]);
}

void mark_cycle(int u, int src, int l = 0) {
    if(l && u == src) return;
    cycle_len[src]++;
    vis[u] = true;
    cycle[u] = src;
    mark_cycle(succ[u],src,l+1);
}

int get_ans(int u) {
    if(ans[u]) return ans[u];
    if(vis[u]) return ans[u] = cycle_len[cycle[u]];
    return ans[u] = 1 + get_ans(succ[u]);
}

BEGIN
{
    cin >> n;
    FOR(u,1,n+1) cin >> succ[u];

    FOR(u,1,n+1) find_cycle(u);
    vis.assign(n+1,false);
```

```
41      FOR(u,1,n+1) {
42          int c = cycle[u];
43          if(vis[c]) continue;
44          mark_cycle(c,c);
45      }
46      FOR(u,1,n+1) get_ans(u);
47
48      FOR(u,1,n+1) cout << ans[u] << '␣';
49      cout << endl;
50  }
51  END
```

8.3.3.5 Ciclo euleriano não direcionado

Objetivo

Sua tarefa é entregar correspondência aos habitantes de uma cidade. Por esse motivo, você deseja encontrar uma rota cujo ponto de partida e de chegada seja a agência dos correios e que passe por cada rua exatamente uma vez.

Estratégia

Algoritmo clássico para encontrar um ciclo Euleriano.

Código

```
1   int n, m;
2   queue<pair<int,int>> edge[MAXN];
3   vector<int> deg(MAXN,0), path;
4   vector<bool> vis(2*MAXN,0);
5
6   void dfs(int u)
7   {
8       while(!edge[u].empty()) {
9           auto e = edge[u].front();
10          edge[u].pop();
11          if(!vis[e.second]) {
12              vis[e.second] = 1;
13              dfs(e.first);
14          }
15      }
16      path.push_back(u);
17  }
18
19  BEGIN
20  {
21      cin >> n >> m;
22      FOR(i,0,m) {
23          int a, b;
24          cin >> a >> b;
25          edge[a].push({b,i});
26          edge[b].push({a,i});
```

```
27          ++deg[a]; ++deg[b];
28      }
29
30      FOR(u,1,n+1) {
31          if(!(deg[u]%2)) continue;
32          cout << "IMPOSSIBLE\n";
33          return 0;
34      }
35
36      dfs(1);
37      if(path.size() != m+1) {
38          cout << "IMPOSSIBLE\n";
39          return 0;
40      }
41
42      for(int u: path) cout << u << '␣';
43      cout << endl;
44  }
45  END
```

8.3.3.6 Ciclo euleriano direcionado

Objetivo

Um jogo tem níveis n e m teleportes entre eles. Você vence o jogo se mover do nível 1 para o nível n usando cada teleporte exatamente uma vez. Isso é possível? Se sim, qual é uma sequência possível de teleportes.

Estratégia

A base da solução é encontrar um ciclo Euleriano. Contudo, é preciso que exista uma conexão extra entre os vértices 1 e n, já que o ciclo retorna sempre para 1. Isso tem as seguintes implicações.

- todos os vértices precisam ter seu grau de saída igual a seu grau de entrada
- **exceção 1:** o vértice 1 pode ter grau de entrada menor
- **exceção 2:** o vértice n pode ter grau de entrada maior

Código

```
1   int n, m;
2   queue<pair<int,int>> adj[MAXN];
3   vector<bool> vis(2*MAXN);
4   vector<int>
5       in(MAXN,0),
6       out(MAXN,0),
7       path;
8
9   void dfs(int u)
10  {
11      while(!adj[u].empty()) {
12          auto e = adj[u].front();
```

```
13          adj[u].pop();
14          if(!vis[e.second]) {
15              vis[e.second] = 1;
16              dfs(e.first);
17          }
18      }
19      path.push_back(u);
20  }
21
22  void quit()
23  {
24      cout << "IMPOSSIBLE\n";
25      exit(0);
26  }
27
28  BEGIN
29  {
30      cin >> n >> m;
31      FOR(i,0,m) {
32          int a, b;
33          cin >> a >> b;
34          adj[a].push({b,i});
35          ++out[a]; ++in[b];
36      }
37
38      if(in[n]-1 != out[n]) quit();
39      if(in[1]+1 != out[1]) quit();
40      FOR(u,2,n) if(in[u] != out[u]) quit();
41
42      dfs(1);
43
44      if(path.size() != m+1 || path[0] != n) quit();
45      reverse(ALL(path));
46      for(int u : path) cout << u << '␣';
47      cout << endl;
48  }
49  END
```

8.3.4 Componentes

8.3.4.1 Componentes conectados

Objetivo

Você recebe um mapa de um edifício e sua tarefa é contar o número de seus quartos. O tamanho do mapa é $n \times m$ quadrados, e cada quadrado é ou piso ou parede. Você pode caminhar para a esquerda, direita, para cima e para baixo através dos quadrados de piso.

Estratégia

Como o grafo é **não direcionado**, basta usar o algoritmo de DFS, que encontra os componentes conectados no grafo (mapa).

Código

```
vector<vector<char>> build;
vector<vector<bool>> visited;

void expand(int i, int j)
{
    if(build[i][j] == '#' || visited[i][j]) return;
    visited[i][j] = 1;
    expand(i+1, j);
    expand(i,j+1);
    expand(i-1,j);
    expand(i,j-1);
}

BEGIN
{
    int n, m;
    cin >> n >> m;

    build = vector<vector<char>>(n+2, vector<char>(m+2, '#'));
    visited = vector<vector<bool>>(n+2, vector<bool>(m+2, 0));
    FOR(i,1,n+1) FOR(j,1,m+1) cin >> build[i][j];

    int rooms = 0;
    FOR(i,1,n+1) {
        FOR(j,1,m+1) {
            if(visited[i][j] || build[i][j] == '#') continue;
            expand(i,j);
            rooms++;
        }
    }

    cout << rooms << endl;
}
END
```

8.3.4.2 Grafo bipartido

Objetivo

Há n alunos na classe de Uolevi e m amizades entre eles. Sua tarefa é dividir os alunos em dois times de modo que nenhum dois alunos no mesmo time sejam amigos. Você pode escolher livremente os tamanhos dos times.

Estratégia

A resolução é baseada em DFS. A cada chamada, determino um time para um vértice e o time contrário para todos os seus vizinhos. Se, em algum ponto, isso não for possível (um vizinho já tem time e esse time é igual ao do vértice atual), não é possível resolver o problema.

Código

```
int u, v;
vector<vector<int>> G;
vector<bool> team;
vector<bool> vis;
bool impossible = false;

void dfs(int u) {
    vis[u] = true;
    for(int v : G[u]) {
        if(vis[v]) {
            if(team[v] == team[u]) {
                impossible = true;
                return;
            }
            continue;
        }
        team[v] = !team[u];
        dfs(v);
    }
}

BEGIN
{
    int n, m;
    cin >> n >> m;

    G = vector<vector<int>>(n+1, vector<int>());
    team = vector<bool>(n+1,0);
    vis = vector<bool>(n+1,0);
    FOR(i,0,m) {
        cin >> u >> v;
        G[v].push_back(u);
        G[u].push_back(v);
    }

    FOR(i,0,n) dfs(i);

    if(impossible) cout << "IMPOSSIBLE\n";
    else {
        FOR(i,1,n+1) cout << team[i]+1 << '␣';
        cout << endl;
    }
}
END
```

8.3.4.3 Conectando um grafo

Objetivo

Byteland possui n cidades e m estradas entre elas. O objetivo é construir novas estradas de modo que haja uma rota entre qualquer duas cidades. Sua tarefa é determinar o número mínimo de estradas necessárias e também quais estradas devem ser construídas.

Estratégia

O grafo não é direcionado. Assim, basta usar o algoritmo de DFS para encontrar as componentes conexas, armazenando um vértice de cada componente. Feito isso, basta selecionar um dos vértices armazenadados e construir arestas entre ele e os restantes.

Código

```
1  int n, m;
2  vector<bool> vis;
3  vector<vector<int>> ng;
4
5  void dfs(int v)
6  {
7      if(vis[v]) return;
8      vis[v] = 1;
9      for(int u: ng[v]) dfs(u);
10 }
11
12 BEGIN
13 {
14     cin >> n >> m;
15     vis = vector<bool>(n+1);
16     ng = vector<vector<int>>(n+1, vector<int>());
17
18     int u, v;
19     FOR(i,0,m) {
20         cin >> u >> v;
21         ng[v].push_back(u);
22         ng[u].push_back(v);
23     }
24
25     vector<int> comp;
26     FOR(i,1,n+1) {
27         if(vis[i]) continue;
28         comp.push_back(i);
29         dfs(i);
30     }
31
32     cout << comp.size()-1 << endl;
33     FOR(i,1,comp.size()) {
34         cout << comp[0] << '␣'
35              << comp[i] << endl;
36     }
37 }
38 END
```

8.3.4.4 Construindo componentes iterativamente

Objetivo

Há n cidades e inicialmente não há estradas entre elas. No entanto, a cada dia uma nova estrada será construída, e haverá um total de m estradas. Um componente é um grupo de cidades onde há uma rota entre quaisquer duas cidades usando as estradas. Após cada dia, sua tarefa é encontrar o número de componentes e o tamanho do maior componente.

Estratégia

A solução é baseada em DSU. Basta manter um contador para o número de componentes (que será decrementado sempre que for feita uma união de componentes) e uma fila de prioridades para determinar o eficientemente o tamanho do maior componente.

Código

```
#define MAXN (int)(2e5+1)

int n, m, sets;
vector<int> par(MAXN), sz(MAXN,1);
priority_queue<int> pq;

int get_set(int a) {
    if(par[a] == a) return a;
    return par[a] = get_set(par[a]);
}

void union_sets(int a, int b) {
    a = get_set(a);
    b = get_set(b);
    if(a != b) {
        par[b] = a;
        sz[a] += sz[b];
        pq.push(sz[a]);
        --sets;
    }
}

BEGIN
{
    cin >> n >> m;

    sets = n;
    FOR(u,1,n+1) par[u] = u;
    FOR(i,0,m) {
        int a,b;
        cin >> a >> b;
        union_sets(a,b);
        cout << sets << "␣" << pq.top() << endl;
    }
}
END
```

8.3.4.5 Grafo fortemente conexo

Objetivo

Há n cidades e m conexões de voo. Sua tarefa é verificar se você pode viajar de qualquer cidade para qualquer outra cidade usando os voos disponíveis. Imprima 'SIM' se todas as rotas forem possíveis, e 'NÃO' caso contrário. Neste último caso, também imprima duas cidades a e b tais que você não consiga viajar da cidade a para a cidade b.

Estratégia

A solução é baseada em duas DFS, uma feita no grafo original e outra feito no grafo reverso. Ambas devem ser iniciadas no mesmo vértice. Se um vértice v não é encontrado em alguma das buscas, então existe pelo menos um par de cidades $(1, v)$ que não possui ligação. Caso contrário, todos os vértices conseguem se alcançar de alguma forma - por próprio intermédio do vértice inicial.

Código

```
#define MAXN (int)(2e5+1)

int n, m;
vector<int> adj[MAXN], par[MAXN];
vector<bool> from(MAXN,0), to(MAXN,0);

void dfs(int u, vector<bool> &vis, const vector<int> *rel)
{
    vis[u] = true;
    for(int v : rel[u]) {
        if(vis[v]) continue;
        dfs(v,vis,rel);
    }
}

BEGIN
{
    cin >> n >> m;
    FOR(i,0,m) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        par[b].push_back(a);
    }

    dfs(1,to,adj);
    dfs(1,from,par);

    pair<int,int> ans {0,0};
    FOR(u,1,n+1) {
        if(to[u] && from[u]) continue;
        if(!to[u]) ans = {1,u};
        else if(!from[u]) ans = {u,1};
    }

    if(!ans.first) cout << "YES\n";
    else cout << "NO\n"
```

```
38          << ans.first << "␣"
39          << ans.second << endl;
40  }
41  END
```

8.3.4.6 Componentes fortemente conexos

Objetivo

Um jogo tem n planetas, conectados por m teletransportadores. Dois planetas a e b pertencem ao mesmo reino exatamente quando há uma rota tanto de a para b quanto de b para a. Sua tarefa é determinar para cada planeta seu reino.

Estratégia

Algoritmo clássico para componentes fortemente conexos.

Código

```
1   #define MAXN (int)(2e5+1)
2
3   int n, m;
4   vector<int>
5       adj[MAXN],
6       par[MAXN],
7       kingdom(MAXN);
8   vector<bool> vis(MAXN,0);
9   vector<pair<int,int>> bridges;
10
11  vector<int> order;
12  void build_order(int u) {
13      vis[u] = true;
14      for(int v : adj[u]) {
15          if(vis[v]) continue;
16          build_order(v);
17      }
18      order.push_back(u);
19  }
20
21  int king = 0;
22  void find_kingdom(int u) {
23      kingdom[u] = king;
24      for(int v: par[u]) {
25          if(kingdom[v]) continue;
26          find_kingdom(v);
27      }
28  }
29
30  BEGIN
31  {
32      cin >> n >> m;
```

```
33      FOR(i,0,m) {
34          int a, b;
35          cin >> a >> b;
36          adj[a].push_back(b);
37          par[b].push_back(a);
38      }
39
40      FOR(u,1,n+1) {
41          if(vis[u]) continue;
42          build_order(u);
43      }
44      FORIT_R(order) {
45          if(kingdom[*it]) continue;
46          ++king;
47          find_kingdom(*it);
48      }
49
50      cout << king << endl;
51      FOR(u,1,n+1) cout << kingdom[u] << '␣';
52      cout << endl;
53  }
54  END
```

8.3.4.7 Percorrendo componentes

Objetivo

Um jogo possui n salas e m túneis entre elas. Cada sala tem uma certa quantidade de moedas. Qual é o número máximo de moedas que você pode coletar ao se mover pelos túneis quando pode escolher livremente sua sala inicial e final?

Estratégia

Como não há limitação de repetição de vértices e arestas, é preciso notar que

- quando em um componente fortemente conexo, todas as moedas dele podem ser coletadas
- é preciso saber qual o melhor caminho entre esses componentes, a fim de que a soma total de moedas seja máxima

A solução, então, consiste em determinar os componentes (algoritmo clássico) e usar DP para determinar a soma possível dado o início em cada componente.

Código

```
1  int n, m;
2  vector<int>
3      order, comps,
4      adj[MAXN], par[MAXN],
5      comp_adj[MAXN],
6      comp(MAXN,0),
7      val(MAXN);
```

```
8  vector<long>
9      comp_val(MAXN,0),
10     dp(MAXN,0);
11 vector<bool> vis(MAXN,0);
12
13 void discover(int u) {
14     if(vis[u]) return;
15     vis[u] = true;
16     for(int v : adj[u]) discover(v);
17     order.push_back(u);
18 }
19
20 void scc(int u, int c) {
21     if(comp[u]) return;
22     comp[u] = c;
23     comp_val[c] += val[u];
24     for(int v : par[u]) {
25         if(comp[v]) continue;
26         scc(v,c);
27     }
28 }
29
30 long calc(int c) {
31     if(dp[c]) return dp[c];
32     dp[c] = comp_val[c];
33     for(int j : comp_adj[c]) dp[c] = max(dp[c], calc(j) + comp_val[c]);
34     return dp[c];
35 }
36
37 BEGIN
38 {
39     cin >> n >> m;
40     FOR(u,1,n+1) cin >> val[u];
41     FOR(i,0,m) {
42         int a, b;
43         cin >> a >> b;
44         adj[a].push_back(b);
45         par[b].push_back(a);
46     }
47
48     FOR(u,1,n+1) discover(u);
49     FOR(i,1,n+1) {
50         int u = order[n-i];
51         if(!comp[u]) {
52             scc(u,u);
53             comps.push_back(u);
54         }
55     }
56     FOR(u,1,n+1) {
57         for(int j : adj[u]) {
58             if(comp[u] == comp[j]) continue;
59             comp_adj[comp[j]].push_back(comp[u]);
60         }
61     }
62
```

```
63      long ans = 0;
64      for(int c : comps) {
65          ans = max(ans, calc(c));
66      }
67      cout << ans << endl;
68  }
```

8.3.5 Ordenação

8.3.5.1 Schedule

Objetivo

Você tem que completar n cursos. Existem m requisitos do tipo "o curso a deve ser completado antes do curso b". Sua tarefa é encontrar uma ordem na qual você pode completar os cursos.

Solução

O problema pode ser resolvido por meio de ordenação topológica.

Exemplo

```
1   BEGIN
2   {
3       int n, m, a, b;
4       cin >> n >> m;
5
6       vector<int> parent[n+1];
7       vector<int> child(n+1,0);
8
9       FOR(i,0,m) {
10          cin >> a >> b;
11          parent[a].push_back(b);
12          ++child[b];
13      }
14
15      queue<int> pq;
16      vector<int> order;
17      vector<bool> vis(n+1,false);
18      int c = 0;
19
20      FOR(i,1,n+1) if(!child[i]) pq.push(i);
21      while(pq.size()) {
22          int u = pq.front();
23          pq.pop();
24
25          c += !vis[u];
26          vis[u] = true;
27
```

```
28          order.push_back(u);
29          for(int v : parent[u]) {
30              if(!child[v]) goto impossible;
31              child[v]--;
32              if(!child[v]) pq.push(v);
33          }
34      }
35      if(c < n) goto impossible;
36
37      for(auto it = order.begin();
38          it != order.end();
39          cout << *it << '␣', it++);
40      cout << endl;
41      return 0;
42
43      impossible:
44      cout << "IMPOSSIBLE\n";
45  }
46  END
```

8.3.5.2 Rota com mais arestas

Objetivo

Uolevi ganhou um concurso, e o prêmio é uma viagem de avião gratuita que pode consistir em um ou mais voos por cidades. Claro, Uolevi quer escolher uma viagem que tenha o maior número possível de cidades.

Uolevi quer voar de Syrjälä para Lehmälä de modo que ele visite o número máximo de cidades. Você recebeu a lista de voos possíveis e sabe que não há ciclos direcionados na rede de voos.

Solução

O problema pode ser resolvido por meio de ordenação topológica. O maior diferencial é a necessidade de retirar a influência de caminhos que não se iniciam no vértice 1. Para isso, basta executar uma DFS a partir de 1: se um vértice não foi visitado, retira-se qualquer dependência ligada a ele.

Exemplo

```
1  vector<bool> vis(MAXN,0);
2  vector<int>
3      dep(MAXN,0),
4      level(MAXN,0),
5      par(MAXN),
6      adj[MAXN],
7      path;
8  queue<int> q;
9  int n, m;
10
11 void dfs(int u) {
12     vis[u] = true;
13     for(int v: adj[u]) {
```

```
14          if(vis[v]) continue;
15          dfs(v);
16      }
17  }
18
19  BEGIN
20  {
21      cin >> n >> m;
22      FOR(i,0,m) {
23          int a,b;
24          cin >> a >> b;
25          adj[a].push_back(b);
26          ++dep[b];
27      }
28
29      dfs(1);
30      FOR(u,1,n+1) {
31          if(vis[u]) continue;
32          for(int v: adj[u]) --dep[v];
33      }
34
35      q.push(1);
36      while(q.size()) {
37          int u = q.front(); q.pop();
38          for(int v : adj[u]) {
39              --dep[v];
40              if(level[v] <= level[u]) {
41                  level[v] = level[u]+1;
42                  par[v] = u;
43              }
44              if(!dep[v]) q.push(v);
45          }
46      }
47
48      int x = n;
49      while(x != 1 && par[x]) {
50          path.push_back(x);
51          x = par[x];
52      }
53
54      if(x != 1) cout << "IMPOSSIBLE\n";
55      else {
56          path.push_back(1);
57          cout << path.size() << endl;
58          FORIT_R(path) cout << *it << '␣';
59          cout << endl;
60      }
61  }
62  END
```

8.3.6 Fluxo

8.3.6.1 Fluxo em rede

Objetivo

Considerando uma rede consistindo de n computadores e m conexões. Cada conexão especifica a velocidade com que um computador pode enviar dados para outro computador. Kotivalo quer baixar alguns dados de um servidor. Qual é a velocidade máxima com que ele pode fazer isso, usando as conexões na rede?

Estratégia

Algoritmo de fluxo máximo clássico.

Código

```
int n, m;
vector<vector<int64_t>> capacity(MAXN, vector<int64_t>(MAXN, 0));
vector<int> adj[MAXN];
vector<int> parent(MAXN);

int64_t bfs(int s, int t) {
    queue<pair<int,int64_t>> Q;

    fill(parent.begin(), parent.end(), 0);
    Q.push({s, INT64_MAX});

    while(Q.size()) {
        int u = Q.front().first;
        int64_t flow = Q.front().second;
        Q.pop();

        if(u == t) return flow;

        for(int v : adj[u]) {
            if(parent[v] || !capacity[u][v]) continue;

            int64_t new_flow = min(flow, capacity[u][v]);
            parent[v] = u;
            Q.push({v, new_flow});
        }
    }

    return 0;
}

BEGIN
{
    cin >> n >> m;
    FOR(i,0,m) {
        int a,b;
        uint64_t c;
        cin >> a >> b >> c;
        adj[a].push_back(b);
        adj[b].push_back(a);
        capacity[a][b] += c;
```

```
41      }
42
43      int64_t flow = 0;
44      int64_t new_flow;
45
46      while(new_flow = bfs(1,n)) {
47          int u,v;
48
49          u = n;
50          flow += new_flow;
51          while(u != 1) {
52              v  = parent[u];
53              capacity[u][v] += new_flow;
54              capacity[v][u] -= new_flow;
55              u = v;
56          }
57      }
58
59      cout << flow << endl;
60  }
61  END
```

8.3.6.2 Corte de fluxo

Objetivo

Kaaleppi acabou de assaltar um banco e agora está indo para o porto. No entanto, a polícia quer impedi-lo fechando algumas ruas da cidade. Qual é o número mínimo de ruas que devem ser fechadas para que não haja rota entre o banco e o porto?

Estratégia

A solução envolve um corte de fluxo no grafo. Para determiná-la, é preciso usar um algoritmo de fluxo máximo. Após a execução, o grafo será composto por dois componentes, divididos pelas arestas sobrecarregadas pelo fluxo. A resposta do problema, então, é o conjunto de arestas que ligam esses componentes.

OBS: A determinação dos componentes é feita na própria DFS e verificada pelo vetor de $visitados$.

Código

```
1  int n, m, a, b;
2  long g[MAXN][MAXN], w[MAXN][MAXN];
3  int vis[MAXN];
4  int par[MAXN];
5  vector<pair<int,int>> ans;
6
7  bool stuck()
8  {
9      memset(vis,0,sizeof(vis));
10     queue<int> Q;
11
```

```
12      vis[1] = 1;
13      Q.push(1);
14      while(Q.size()) {
15          int u = Q.front();
16          Q.pop();
17          FOR(v,1,n+1) {
18              if(!w[u][v] || vis[v]) continue;
19              vis[v] = 1;
20              par[v] = u;
21              Q.push(v);
22          }
23      }
24
25      return !vis[n];
26  }
27
28  BEGIN
29  {
30      cin >> n >> m;
31      FOR(i,0,m) {
32          cin >> a >> b;
33          ++g[a][b]; ++w[a][b];
34          ++g[b][a]; ++w[b][a];
35      }
36
37      int v;
38      while(!stuck()) {
39          long flow = 1e18;
40
41          for(v = n; v != 1; v = par[v]) {
42              flow = min(flow, w[par[v]][v]);
43          }
44
45          for(v = n; v != 1; v = par[v]) {
46              w[par[v]][v] -= flow;
47              w[v][par[v]] += flow;
48          }
49      }
50      stuck();
51
52      FOR(u,1,n+1) FOR(v,1,n+1) {
53          if(vis[u] && !vis[v] && g[u][v])
54              ans.emplace_back(u,v);
55      }
56
57      cout << ans.size() << endl;
58      for(auto p: ans) cout << p.first << '␣' << p.second << endl;
59  }
60  END
```

8.3.7 Consultas

8.3.7.1 k-ésimo antecessor

Objetivo

Você está jogando um jogo composto por n planetas. Cada planeta possui um teletransportador para outro planeta (ou para o próprio planeta). Sua tarefa é processar q consultas do seguinte formato: quando você começar no planeta x e viajar através de k teletransportadores, para qual planeta você chegará?

Estratégia

Para atender cada consulta eficientemente, é preciso usar **caminhamento binário**. Para cada vértice, são armazenados seus predecessores de grau 2^l $0 \leq l \leq log_2 n$. Assim, dado qualquer distância k, é possível encontrar o predecessor correspondente por meio de cada bit em $(k)_2$.

Código

```
#define MAXN (int)(2e5+1)
#define MAXJ 30

int par[MAXN][MAXJ];
int n, q, x, k;

int go(int x, int k) {
    FOR(j,0,MAXJ) {
        if(k & (1 << j)) x = par[x][j];
    }
    return x;
}

BEGIN
{
    cin >> n >> q;
    FOR(u,1,n+1) {
        cin >> par[u][0];
    }

    FOR(j,1,MAXJ)
        FOR(i,1,n+1) {
            int next = par[i][j-1];
            par[i][j] = par[next][j-1];
        }

    while(q--) {
        cin >> x >> k;
        cout << go(x,k) << endl;
    }
}
END
```

8.3.7.2 Distância entre *a* e *b*

Objetivo

Você está jogando um jogo composto por n planetas. Cada planeta possui um teletransportador para outro planeta (ou para o próprio planeta). Você precisa processar q consultas no seguinte formato: Você está agora no planeta a e quer chegar ao planeta b. Qual é o número mínimo de teletransportações necessárias?

Estratégia

A solução é construída em torno da existência de ciclos no grafo. Por meio de uma DFS, associa-se a cada vértice uma distância: se o vértice está em um ciclo, ela indica o número de passos para se alcançar o nó raiz c do ciclo (o que fica definido pela DFS; caso contrário, o vértice está em uma árvore anexa a um ciclo, e a distância é referente até a conexão c com esse ciclo.

Temos, assim, três casos distintos:

- ***b* é antecessor de *a* :** Nesse caso, a resposta é dada por $dist(b) - dist(a)$
- ***c* conecta *b* e *a* :** Nesse caso, a resposta é dada por $dist(a) - dist(b) + dist(c)$
- **não existe caminho de *a* para *b***: A resposta é -1

Para descobrir c de forma eficiente, é possível usar a mesma estratégia de caminhamento de a resolução 8.3.7.1.

Código

```
#define MAXN (int)(2e5+1)
#define MAXD 18

int n,q,ans;
int a,b,c;
int par[MAXN][MAXD];
static vector<bool> vis(MAXN);
static vector<int> len(MAXN);

int lift(int u, int d) {
    for(int i = 0; d > 0; d >>= 1, i++)
        if(d & 1) u = par[u][i];
    return u;
}

void dfs(int u) {
    if(vis[u]) return;
    vis[u] = true;
    dfs(par[u][0]);
    len[u] = len[par[u][0]]+1;
}

BEGIN
{
    cin >> n >> q;
    FOR(u,1,n+1) cin >> par[u][0];

    FOR(d,1,MAXD)
        FOR(u,1,n+1) {
            par[u][d] = par[par[u][d-1]][d-1];
        }

```

```
33      FOR(u,1,n+1) {
34          if(vis[u]) continue;
35          dfs(u);
36      }
37
38      while(q--) {
39          cin >> a >> b;
40          c = lift(a,len[a]);
41
42          if(lift(a, len[a]-len[b]) == b) ans = len[a]-len[b];
43          else if(lift(c, len[c]-len[b]) == b) ans = len[a]+len[c]-len[b];
44          else ans = -1;
45
46          cout << ans << endl;
47      }
48  }
49  END
```

8.3.8 Emparelhamento

8.3.8.1 Ciclo euleriano não direcionado

Objetivo

Existem n meninos e m meninas em uma escola. Na próxima semana, um baile escolar será organizado. Um par de dança consiste em um menino e uma menina, e há k pares potenciais. Sua tarefa é descobrir o número máximo de pares de dança e mostrar como esse número pode ser alcançado.

Estratégia

Emparelhamento máximo - aplicação direta do algoritmo de Kuhn.

Código

```
1  int n, m, k, a, b;
2  vector<vector<int>> g;
3  vector<int> mt;
4  vector<bool> used;
5  vector<pair<int,int>> ans;
6
7  bool try_kuhn(int u) {
8      if (used[u]) return false;
9      used[u] = true;
10     for (int v : g[u]) {
11         if (!mt[v] || try_kuhn(mt[v])) {
12             mt[v] = u;
13             return true;
14         }
15     }
```

```
16      return false;
17  }
18
19  BEGIN
20  {
21      cin >> n >> m >> k;
22      g.resize(n+1);
23      mt.assign(m+1, 0);
24
25      FOR(i,0,k) {
26          cin >> a >> b;
27          g[a].push_back(b);
28      }
29
30      FOR(v,1,n+1) {
31          used.assign(n, false);
32          try_kuhn(v);
33      }
34
35
36      FOR(i,1,m+1) {
37          if(!mt[i]) continue;
38          ans.emplace_back(mt[i], i);
39      }
40
41      cout << ans.size() << endl;
42      for(auto p : ans) {
43          cout << p.first << "␣" << p.second << endl;
44      }
45  }
46  END
```

8.3.9 Outros

8.3.9.1 Caminho em um tabuleiro

Objetivo

Você recebe um mapa de um labirinto e sua tarefa é encontrar um caminho do início ao fim. Você pode caminhar para a esquerda, direita, para cima e para baixo.

Estratégia

Qualquer estratégia de caminhamento pode ser usada, bastando armazenar os pais designados a cada vértice durante a construção do caminho.

Código

```
unordered_map<char,pair<int,int>> dir {
    {'L', {0,-1}},
    {'U', {-1,0}},
    {'R', {0,1}},
    {'D', {1,0}}
};

BEGIN
{
    int n, m;
    cin >> n >> m;

    vector<vector<char>> build(n+2, vector<char>(m+2, '#'));
    vector<vector<char>> parent(n+2, vector<char>(m+2));
    vector<vector<bool>> visited(n+2, vector<bool>(m+2, 0));
    pair<int,int> start, end;
    FOR(i,1,n+1) {
        FOR(j,1,m+1) {
            cin >> build[i][j];
            if(build[i][j] == 'A') start = {i,j};
            if(build[i][j] == 'B') end = {i,j};
        }
    }

    queue<pair<int,int>> Q;
    pair<int,int> cur;
    Q.push(start);

    while(Q.size()) {
        cur = Q.front(); Q.pop();
        if(visited[cur.first][cur.second]) continue;
        visited[cur.first][cur.second] = 1;

        if(cur == end) break;
        for(auto item : dir) {
            pair<int,int> next {
                cur.first + item.second.first,
                cur.second + item.second.second
            };
            if(build[next.first][next.second] == '#'
               || visited[next.first][next.second]) continue;
            parent[next.first][next.second] = item.first;
            Q.push(next);
        }
    }

    if(cur != end) {
        cout << "NO" << endl;
    }
    else {
        cout << "YES" << endl;
        vector<char> path;
        while(cur != start) {
            char p = parent[cur.first][cur.second];
            path.push_back(p);
```

```
56              cur.first -= dir[p].first;
57              cur.second -= dir[p].second;
58          }
59          cout << path.size() << endl;
60          for(auto it = path.rbegin(); it != path.rend(); it++) cout << *it;
61          cout << endl;
62      }
63  }
64  END
```

8.3.9.2 Contando rotas

Objetivo

Um jogo tem níveis, conectados por m teleportadores, e sua tarefa é ir do nível 1 ao nível n. O jogo foi projetado de forma que não existam ciclos direcionados no gráfico subjacente. De quantas maneiras você pode completar o jogo?

Solução

Basta utilizar o algoritmo de DFS e trabalhar com um vetor ans, sendo que

$$ans[n] = 1$$
$$ans[u] = \sum ans[v] \quad \forall v \in adj[u]$$

Exemplo

```
1   vector<bool> vis(MAXN,false);
2   vector<long> reach(MAXN,0);
3   vector<int> adj[MAXN];
4   int n, m;
5
6   void dfs(int u) {
7       vis[u] = true;
8       for(int v: adj[u]) {
9           if(!vis[v]) dfs(v);
10          reach[u] += reach[v];
11          reach[u] %= MOD;
12      }
13  }
14
15  BEGIN
16  {
17      cin >> n >> m;
18      FOR(i,0,m) {
19          int a,b;
20          cin >> a >> b;
21          adj[a].push_back(b);
22      }
23
24      reach[n] = 1;
```

```
25      dfs(1);
26      cout << reach[1] << endl;
27  }
28  END
```

8.3.9.3 Caminho de distância *x*

Objetivo

Você está jogando um jogo com n planetas e m teletransportes entre eles. Todos os teletransportes funcionam em ambas as direções e conectam dois planetas diferentes. Você começa o jogo em um planeta específico e, a cada turno, se move através de um teletransporte para um planeta diferente. Você pode assumir que todos os planetas estão conectados por teletransportes. Sua tarefa é responder a q consultas do tipo: "é possível começar o jogo no planeta a e terminar no planeta b após x turnos?"

Estratégia

Dado x, basta encontrar qualquer caminho de tamanho l entre a e b tal que $l \leq x$ e $l \pmod 2 = x \pmod 2$. Isso porque a repetição de vértices no caminho é permitida e, uma vez em b, é possível percorrer uma aresta um número par de vezes até alcançar a distância x necessária.

Para fazer essa verificação, basta usar uma BFS. Na fila, é colocado, além de cada vértice, a paridade de um determinado caminho que vai até ele. A busca, então, tenta encontrar um caminho par e um caminho ímpar até cada vértice.

Código

```
1  vector<int> adj[MAXN];
2  int ans[MAXN][MAXN][2];
3  bool vis[MAXN][2];
4  int n, m, q;
5  int a, b, x;
6
7  BEGIN
8  {
9      cin >> n >> m >> q;
10     FOR(i,0,m) {
11         cin >> a >> b;
12         adj[a].push_back(b);
13         adj[b].push_back(a);
14     }
15     memset(ans, SCHAR_MAX, sizeof(ans));
16
17     FOR(i,1,n+1) {
18         queue<pair<int,int>> Q;
19
20         memset(vis,0,sizeof(vis));
21         ans[i][i][0] = 0;
22         vis[i][0] = 1;
23         Q.push({i,0});
24
```

```
25          while(Q.size()) {
26              int u = Q.front().first;
27              int len = Q.front().second;
28              Q.pop();
29
30              for(int v : adj[u]) {
31                  if(vis[v][!(len%2)]) continue;
32                  vis[v][!(len%2)] = 1;
33                  ans[i][v][!(len%2)] = len+1;
34                  Q.push({v,len+1});
35              }
36          }
37      }
38
39      while(q--) {
40          cin >> a >> b >> x;
41          cout << (ans[a][b][x%2] <= x ? "YES" : "NO") << endl;
42      }
43  }
44  END
```

8.3.9.4 Satisfabilidade

Objetivo

A família de Uolevi vai pedir uma pizza grande e comê-la juntos. Um total de n membros da família se juntarão ao pedido, e há m coberturas possíveis. A pizza pode ter qualquer número de coberturas. Cada membro da família dá dois desejos relacionados às coberturas da pizza. Os desejos estão na forma "cobertura x é boa/ruim". Sua tarefa é escolher as coberturas de forma que pelo menos um desejo de cada pessoa se torne verdadeiro (uma cobertura boa está incluída na pizza ou uma cobertura ruim não está incluída).

Para cada cobertura, vou imprimir um símbolo "+"se estiver incluído e -"se não estiver incluído. Eu posso escolher qualquer solução válida. Se não houver soluções válidas, vou imprimir "IMPOSSIBLE".

Estratégia

Algoritmo 2-SAT clássico.

Código

```
1  vector<int> adj[2*MAXN], adj_t[2*MAXN];
2  vector<int> comp(2*MAXN,-1), order;
3  vector<bool> used(2*MAXN,0), ass(MAXN,0);
4  bool possible = true;
5  int k = 0;
6  int n, m;
7  int a, b;
8  char x, y;
9
10 void dfs1(int u)
11 {
```

```
12      if(used[u]) return;
13      used[u] = true;
14      for(int v : adj[u]) dfs1(v);
15      order.push_back(u);
16  }
17
18  void dfs2(int u)
19  {
20      if(comp[u] != -1) return;
21      comp[u] = k;
22      for(int v : adj_t[u])
23          dfs2(v);
24  }
25
26  void print(int u)
27  {
28      cout << (u%2 ? "- " : "+ ") << u/2 << " ";
29  }
30
31  BEGIN
32  {
33      cin >> n >> m;
34      FOR(i,0,n) {
35          cin >> x >> a >> y >> b;
36          a = 2*(a-1)^(x=='-');
37          b = 2*(b-1)^(y=='-');
38
39          adj[a^1].push_back(b);
40          adj[b^1].push_back(a);
41          adj_t[a].push_back(b^1);
42          adj_t[b].push_back(a^1);
43      }
44
45      m *= 2;
46      FOR(u,0,m) dfs1(u);
47      FOR(i,1,m+1) {
48          int v = order[m-i];
49          if(comp[v] == -1) {
50              dfs2(v);
51              ++k;
52          }
53      }
54
55      FOR(i,0,m) {
56          if(comp[i] == comp[i+1]) {
57              possible = false;
58              break;
59          }
60          ass[i/2] = comp[i] > comp[i+1];
61          ++i;
62      }
63
64      if(!possible) cout << "IMPOSSIBLE\n";
65      else {
66          FOR(i,0,m) cout << (ass[i++/2] ? '+' : '-') << ' ';
```

```
67          cout << endl;
68      }
69  }
70  END
```

8.3.9.5 A sequência de De Bruijn

Objetivo

Sua tarefa é construir uma string de bits de comprimento mínimo que contenha todas as substrings possíveis de comprimento n. Por exemplo, quando n = 2, a string 00110 é uma solução válida, porque suas substrings de comprimento 2 são 00, 01, 10 e 11.

Estratégia

Algoritmo clássico baseado em DFS. **A complexidade é alta**.

Código

```
1   unordered_set<string> vis;
2   vector<int> edges;
3
4   void dfs(string node, string a)
5   {
6       FOR(i,0,a.size()) {
7           string s = node + a[i];
8           if(vis.find(s) == vis.end()) {
9               vis.insert(s);
10              dfs(s.substr(1), a);
11              edges.push_back(i);
12          }
13      }
14  }
15
16  string bruijin(int n, string a)
17  {
18      string node (n-1, a[0]);
19      dfs(node, a);
20
21      string ans;
22      FOR(i,0,pow(a.size(),n)) ans += a[edges[i]];
23      ans += node;
24
25      return ans;
26  }
27
28  BEGIN
29  {
30      int n; cin >> n;
31      cout << bruijin(n, "01") << endl;
32  }
```

```
33 END
```

8.3.9.6 Caminho do cavalo

Objetivo

Dada uma posição inicial de um cavalo em um tabuleiro de xadrez 8×8, sua tarefa é encontrar uma sequência de movimentos de modo que ele visite cada casa exatamente uma vez. Em cada movimento, o cavalo pode mover-se duas casas horizontalmente e uma verticalmente, ou uma casa horizontalmente e duas verticalmente.

Estratégia

A base da solução é o algoritmo força bruta. Contudo, é preciso utilizar a **otmização de Warnsdorff**: a cada passo, o cavalo sempre se desloca para a posição adjacente (de acordo com os movimentos possíveis) de **menor grau** - sendo o grau de uma posição a quantidade de posições livres adjacentes a ela.

Código

```
1  int degree(int i, int j) {
2      int en = 0;
3      for(auto p : adj) {
4          if(!VALID(i+p.first) || !VALID(j+p.second)) continue;
5          en += !ans[i][j];
6      }
7      return en;
8  }
9
10 bool move(int i, int j, int c = 1) {
11     ans[i][j] = c;
12
13     if(c == 64) return true;
14
15     vector<array<int,3>> nb;
16
17     for(auto p: adj) {
18         if(!VALID(i+p.first) || !VALID(j+p.second) || ans[i+p.first][j+p.second])
19             continue;
20         int cur_deg = degree(i+p.first, j+p.second);
21         nb.push_back({ cur_deg, i+p.first, j+p.second });
22     }
23
24     sort(nb.begin(), nb.end());
25
26     for(array<int,3> next : nb) {
27         if(move(next[1], next[2], c+1)) return true;
28     }
29
30     ans[i][j] = 0;
31     return false;
32 }
33
```

```
34 BEGIN
35 {
36     cin >> x >> y;
37     --y; --x;
38     move(y,x);
39
40     FOR(a,0,8) {
41         FOR(b,0,8) cout << ans[a][b] << '␣';
42         cout << endl;
43     }
44 }
45 END
```

8.4 Consultas em intervalos

8.4.1 Intervalos estáticos

8.4.1.1 Soma estática

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas da seguinte forma: qual é a soma dos valores no intervalo $[a, b]$?

Estratégia

Basta usar uma soma de prefixos e a diferença entre os valores referentes às posições b e $a - 1$.

Código

```
1  int n, q, a, b;
2  vector<long> sum;
3
4  BEGIN
5  {
6      cin >> n >> q;
7      sum.resize(n+1,0);
8      FOR(i,1,n+1) {
9          cin >> sum[i];
10         sum[i] += sum[i-1];
11     }
12
13     while(q--) {
14         cin >> a >> b;
15         cout << (sum[b]-sum[a-1]) << endl;
```

```
16      }
17  }
18  END
```

8.4.1.2 Mínimo estático

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas da seguinte forma: qual é o menor valor no intervalo $[a, b]$?

Estratégia

Basta usar uma SegmentTree.

Código

```
1  int n;
2  vector<int> segtree(2*MAXN,INT_MAX);
3
4  void update(int k, int x)
5  {
6      k += n;
7      segtree[k] = x;
8      while(k >>= 1) {
9          segtree[k] = min(segtree[2*k],segtree[2*k+1]);
10     }
11 }
12
13 int query(int l, int r)
14 {
15     l += n, r += n;
16     int ans = INT_MAX;
17     while(l <= r) {
18         if(l&1) ans = min(ans,segtree[l++]);
19         if(~r&1) ans = min(ans,segtree[r--]);
20         l >>= 1, r >>= 1;
21     }
22     return ans;
23 }
24
25 int main()
26 {
27     int q;
28     cin >> n >> q;
29     for(int i = 0; i < n; ++i) {
30         int x;
31         cin >> x;
32         update(i,x);
33     }
34
```

```
35      for(int i = 0; i < q; ++i) {
36          int a, b;
37          cin >> a >> b;
38          cout << query(a-1,b-1) << endl;
39      }
40
41      return 0;
42  }
```

8.4.1.3 Soma estática

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas da seguinte forma: qual é o valor da operação xor entre os valores do intervalo $[a, b]$?

Estratégia

A operação xor é inversível, então basta fazer um vetor com resultados acumulados ao longo do vetor entrada e responder cada consulta com base nas posições b e $a - 1$.

Código

```
1   int n, q, a, b;
2   vector<int> x;
3
4   int main()
5   {
6       cin >> n >> q;
7       x.assign(n+1,0);
8       for(int i = 0; i < n; ++i) {
9           cin >> x[i+1];
10          x[i+1] ^= x[i];
11      }
12      while(q--) {
13          cin >> a >> b;
14          cout << (x[b]^x[a-1]) << endl;
15      }
16  }
```

8.4.1.4 Soma em intervalos 2D

Objetivo

É dado a você um grid de $n \times n$ representando o mapa de uma floresta. Cada quadrado está vazio ou contém uma árvore. O quadrado no canto superior esquerdo tem coordenadas $(1, 1)$, e o quadrado no canto inferior direito tem coordenadas (n, n).

Sua tarefa é processar q consultas da seguinte forma: quantas árvores estão dentro de um determinado retângulo na floresta?

Estratégia

Basta usar uma soma de prefixos $2D$. Uma matriz armazena quantas àrvores existem no retângulo formado por $(0, 0)$ e uma posição (i, j), bastando fazer a remoção dos intervalos indesejados para obter a soma em um retângulo $(a, b) - (i, j)$

Código

```
int n,q,p[4];
vector<vector<long>> dp;
char c;

BEGIN
{
    cin >> n >> q;
    dp.assign(n+1, vector<long>(n+1,0));
    FOR(i,1,n+1) FOR(j,1,n+1) {
        cin >> c;
        dp[i][j] = dp[i-1][j] + dp[i][j-1]
            - dp[i-1][j-1] + (c == '*');
    }

    while(q--) {
        cin >> p[0] >> p[1] >> p[2] >> p[3];
        long ans = dp[p[2]][p[3]]
            - dp[p[2]][p[1]-1]
            - dp[p[0]-1][p[3]]
            + dp[p[0]-1][p[1]-1];
        cout << ans << endl;
    }
}
END
```

8.4.2 Intervalos dinâmicos

8.4.2.1 Soma com atualização

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas dos seguintes tipos:

- Atualizar o valor na posição k para u.
- Qual é a soma dos valores no intervalo $[a, b]$?

Estratégia

Basta usar uma SegmentTree.

Código

```
int n;
vector<long long> segtree(2*MAXN,0);

void update(int k, int x)
{
    k += n;
    segtree[k] = x;
    while(k >>= 1) {
        segtree[k] = segtree[2*k] + segtree[2*k+1];
    }
}

long long query(int l, int r)
{
    l += n, r += n;
    long long ans = 0LL;
    while(l <= r) {
        if(l&1) ans += segtree[l++];
        if(~r&1) ans += segtree[r--];
        l >>= 1, r >>= 1;
    }
    return ans;
}

int main()
{
    int q;
    cin >> n >> q;
    for(int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        update(i,x);
    }

    for(int i = 0; i < q; ++i) {
        int a, b, c;
        cin >> a >> b >> c;
        if(a == 1) update(b-1,c);
        else cout << query(b-1,c-1) << endl;
    }

    return 0;
}
```

8.4.2.2 Mínimo estático

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas dos seguintes tipos:

- Atualizar o valor na posição k para u.
- Qual o menor valor no intervalo $[a, b]$?

Estratégia

Basta usar uma SegmentTree.

Código

```
1  int n, q;
2  vector<int> segtree(2*MAXN,INT_MAX);
3
4  void update(int k, int x)
5  {
6      k += n;
7      segtree[k] = x;
8      while(k >>= 1) {
9          segtree[k] = min(segtree[2*k],segtree[2*k+1]);
10     }
11 }
12
13 int query(int l, int r)
14 {
15     l += n, r += n;
16     int ans = INT_MAX;
17     while(l <= r) {
18         if(l&1) ans = min(ans,segtree[l++]);
19         if(~r&1) ans = min(ans,segtree[r--]);
20         l >>= 1, r >>= 1;
21     }
22     return ans;
23 }
24
25 int main()
26 {
27     cin >> n >> q;
28     for(int i = 0; i < n; ++i) {
29         int x;
30         cin >> x;
31         update(i,x);
32     }
33
34     for(int i = 0; i < q; ++i) {
35         int a, b, c;
36         cin >> a >> b >> c;
37         if(a == 1) update(b-1,c);
38         else cout << query(b-1,c-1) << endl;
39     }
40
41     return 0;
42 }
```

8.4.2.3 Incrementando um intervalo

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas dos seguintes tipos:

- Incrementar cada valor do intervalo $[a, b]$ com o valor u
- Qual o valor da posição k?

Estratégia

Basta usar uma SegmentTree. Porém, os nós folha representam o valor armazenado em cada posição do vetor, e os nós internos armazenam a atualização indicada nas operações de tipo 1.

Para obter a resposta de uma consulta, basta percorrer o caminho do nó raiz até o nó folha desejado e somar as atualizações encontradas ao longo do percurso.

Essa técnica é um caso simples de Lazy Propagation.

Código

```
long T[4*MAXN];
int val[MAXN];
int n, q, k, a, b, c;

void build(int tl, int tr, int v = 1) {
    if(tl == tr) T[v] = val[tr];
    else {
        int tm = (tl+tr)/2;
        build(tl, tm, 2*v);
        build(tm+1, tr, 2*v+1);
        T[v] = 0;
    }
}

void update(int a, int l, int r,
            int tl = 0,
            int tr = n-1,
            int v = 1)
{
    if(l > r) return;
    if(l == tl && r == tr) T[v] += a;
    else {
        int tm = (tl+tr)/2;
        update(a, l, min(r,tm), tl, tm, 2*v);
        update(a, max(l,tm+1), r, tm+1, tr, 2*v+1);
    }
}

long ask(int k,
         int tl = 0,
         int tr = n-1,
         int v = 1) {
    if(tl == tr) return T[v];
```

```
34      else {
35          int tm = (tl+tr)/2;
36          return T[v] + (k <= tm
37              ? ask(k, tl, tm, 2*v)
38              : ask(k, tm+1, tr, 2*v+1));
39      }
40  }
41
42  BEGIN
43  {
44      cin >> n >> q;
45      FOR(i,0,n) cin >> val[i];
46      build(0,n-1);
47
48      FOR(i,0,q) {
49          cin >> k >> a;
50          if(k == 1) {
51              cin >> b >> c;
52              update(c, --a, --b);
53          }
54          else cout << ask(--a) << endl;
55      }
56  }
57  END
```

8.4.2.4 Primeiro valor maior ou igual

Objetivo

Existem n hotéis em uma rua. Para cada hotel, você sabe o número de quartos disponíveis. Sua tarefa é atribuir quartos de hotel para grupos de turistas. Todos os membros de um grupo querem ficar no mesmo hotel.

Os grupos virão até você um após o outro, e você sabe para cada grupo o número de quartos que ele precisa. Você sempre atribui um grupo ao primeiro hotel que tiver quartos suficientes. Após isso, o número de quartos disponíveis no hotel diminui.

Estratégia

A base da solução é uma SegmentTree de máximos. Cada consulta deve percorrer a árvore, priorizando nós da esquerda (referentes aos valores mais à esquerda do vetor base), até encontrar algum valor maior ou igual a x.

Código

```
1  int n,m,r,pos;
2  vector<int> h(MAXN);
3  int tree[4*MAXN];
4
5  void build_tree(int tl = 1,
6                  int tr = n,
7                  int v = 1)
8  {
```

```
9       if(tl == tr) tree[v] = h[tl];
10      else {
11          int tm = (tl+tr)/2;
12          build_tree(tl,tm,2*v);
13          build_tree(tm+1,tr,2*v+1);
14          tree[v] = max(tree[2*v],tree[2*v+1]);
15      }
16  }
17
18  void update(int i, int a,
19              int tl = 1,
20              int tr = n,
21              int v = 1)
22  {
23      if(tl > tr) return;
24      if(tl == tr) {
25          tree[v] = a;
26          h[tl] = a;
27      }
28      else {
29          int tm = (tr+tl)/2;
30          if(i <= tm) update(i,a,tl,tm,2*v);
31          else update(i,a,tm+1,tr,2*v+1);
32          tree[v] = max(tree[2*v],tree[2*v+1]);
33      }
34  }
35
36  int search(int x)
37  {
38      int v = 1;
39      int l = 1;
40      int r = n;
41
42      while(l < r) {
43          int m = (l+r)/2;
44          if(tree[2*v] < x) {
45              v = 2*v+1;
46              l = m + 1;
47          }
48          else {
49              v = 2*v;
50              r = m;
51          }
52      }
53
54      return l * (tree[v] >= x);
55  }
56
57  BEGIN
58  {
59      cin >> n >> m;
60      FOR(i,1,n+1) cin >> h[i];
61      build_tree();
62
63      while(m--) {
```

```
64          cin >> r;
65          pos = search(r);
66          if(pos) update(pos,h[pos]-r);
67          cout << pos << '␣';
68      }
69      cout << endl;
70  }
71  END
```

8.4.2.5 Removendo elementos

Objetivo

Você recebe uma lista composta por n inteiros. Sua tarefa é remover elementos da lista em posições dadas e reportar os elementos removidos.

Estratégia

A base da solução é uma SegmentTree que armazena a quantidade de elementos restantes em cada seguimento do vetor. Dessa forma, é possível caminhar na árvore de acordo com a posição da consulta, retornando o nó folha encontrado e atualizando a contagem de elementos nos seguimentos de que ele faz parte.

Código

```
1   int n,m,pos;
2   vector<int> x(MAXN);
3   int tree[4*MAXN];
4   map<int,int> mp;
5
6   void build_tree(int tl = 1,
7                   int tr = n,
8                   int v = 1)
9   {
10      if(tl == tr) {
11          tree[v] = 1;
12          mp[v] = x[tl];
13      }
14      else {
15          int tm = (tl+tr)/2;
16          build_tree(tl,tm,2*v);
17          build_tree(tm+1,tr,2*v+1);
18          tree[v] = tree[2*v] + tree[2*v+1];
19      }
20  }
21
22  int query(int i,
23            int tl = 1,
24            int tr = n,
25            int v = 1)
26  {
27      if(tl == tr) {
```

```
28          tree[v] = 0;
29          return x[tl];
30      }
31      else {
32          int tm = (tl+tr)/2;
33          int ans = i > tree[2*v]
34              ? query(i-tree[2*v], tm+1, tr, 2*v+1)
35              : query(i, tl, tm, 2*v);
36          tree[v] = tree[2*v] + tree[2*v+1];
37          return ans;
38      }
39  }
40
41  BEGIN
42  {
43      cin >> n;
44      FOR(i,1,n+1) cin >> x[i];
45      build_tree();
46
47      FOR(i,0,n) {
48          cin >> pos;
49          cout << query(pos) << '␣';
50      }
51      cout << endl;
52  }
53  END
```

8.4.2.6 Atualizando e contado elementos

Objetivo

Uma empresa tem n funcionários com certos salários. Sua tarefa é acompanhar os salários e processar consultas.

Estratégia

A base da solução é um OrderedSet. Sabendo a posição de cada valor, basta localizar os limites do intervalo da consulta e retornar sua diferença.

Código

```
1   template<class T> using oset =
2       tree<T, null_type, less<T>, rb_tree_tag,
3       tree_order_statistics_node_update>;
4
5   int n, q;
6   oset<pair<int,int>> st;
7   vector<int> p(MAXN);
8   char type;
9   int a, b;
10
11  BEGIN
```

```
12  {
13      cin >> n >> q;
14      FOR(i,1,n+1) {
15          cin >> p[i];
16          st.insert({p[i],i});
17      }
18
19      while(q--) {
20          cin >> type >> a >> b;
21          if(type == '?') {
22              cout << st.order_of_key({b,INT_MAX})
23                  - st.order_of_key({a,INT_MIN}) << endl;
24          }
25          else {
26              st.erase({p[a],a});
27              st.insert({p[a] = b,a});
28          }
29      }
30  }
31  END
```

8.4.2.7 Soma de prefixos

Objetivo

Dado um array de nn inteiros, sua tarefa é processar qq consultas dos seguintes tipos:

- Atualizar o valor na posição k para u.
- Qual é a soma máxima de prefixo no intervalo $[a, b]$?

Estratégia

A base da solução é uma SegmentTree. Cada nó armazena a soma de seu intervalo e seu maior prefixo. Para unir dois nós, basta notar que

$$sum_v = sum_{2v} + sum_{2*v+1}$$
$$prefix_v = max(prefix_{2v}, sum_{2v} + prefix_{2v+1}$$

Código

```
1  pair<long,long> T[4*MAXN];
2  int val[MAXN];
3  int n, q, k, a, b;
4
5  // first = prefix
6  // second = sum
7
8  void build(int tl = 0,
9             int tr = n-1,
10            int v = 1)
```

```
11 {
12     if(tl == tr) T[v] = { max(val[tr],0), val[tr] };
13     else {
14         int tm = (tl+tr)/2;
15         build(tl, tm, 2*v);
16         build(tm+1, tr, 2*v+1);
17         T[v] = {
18             max(T[2*v].first, T[2*v].second + T[2*v+1].first),
19             T[2*v].second + T[2*v+1].second
20         };
21     }
22 }
23
24 void change(int i, int a,
25             int tl = 0,
26             int tr = n-1,
27             int v = 1)
28 {
29     if(tl > tr) return;
30     if(tl == tr) T[v] = { max(a,0), a };
31     else {
32         int tm = (tr+tl)/2;
33         if(i <= tm) change(i, a, tl, tm, 2*v);
34         else change(i, a, tm+1, tr, 2*v+1);
35         T[v] = {
36             max(T[2*v].first, T[2*v].second + T[2*v+1].first),
37             T[2*v].second + T[2*v+1].second
38         };
39     }
40 }
41
42 pair<long,long> ask(int l, int r,
43                     int tl = 0,
44                     int tr = n-1,
45                     int v = 1)
46 {
47     if(l > r) return {0,0};
48     if(l == tl && r == tr) return T[v];
49     else {
50         int tm = (tl+tr)/2;
51         pair<long,long> left = ask(l, min(r,tm), tl, tm, 2*v);
52         pair<long,long> right = ask(max(l,tm+1), r, tm+1, tr, 2*v+1);
53         return {
54             max(left.first, left.second + right.first),
55             left.second + right.second
56         };
57     }
58 }
59
60 BEGIN
61 {
62     cin >> n >> q;
63     FOR(i,0,n) cin >> val[i];
64     build();
65
```

```
66      FOR(i,0,q) {
67          cin >> k >> a >> b;
68          if(k == 1) change(--a, b);
69          else cout << ask(--a,--b).first << endl;
70      }
71  }
72  END
```

8.4.2.8 Custo mínimo direcionado

Objetivo

Existem n edifícios em uma rua, numerados de $1, 2, \ldots, n$. Cada edifício tem uma pizzaria e um apartamento. O preço da pizza no edifício k é p_k. Se você pedir uma pizza do edifício a para o edifício b, o preço (com entrega) é $p_a + |a - b|$.

Sua tarefa é processar dois tipos de consultas:

- O preço da pizza p_k no edifício k se torna x.
- Você está no edifício k e quer pedir uma pizza. Qual é o preço mínimo?

Estratégia

A base da solução são duas SegmentTrees. Uma armazena o preço mínimo à direita, e a outra, à esquerda. Portanto, basta fazer a busca do valor mínimo nas duas, e as atualizações necessárias.

Código

```
1  int down[4*MAXN];
2  int up[4*MAXN];
3
4  int val[MAXN];
5  int n, q, t, k, x;
6
7  void build(int tl = 1,
8             int tr = n,
9             int v = 1)
10 {
11     if(tl == tr) {
12         down[v] = val[tl]-tl;
13         up[v] = val[tl]+tl;
14     }
15     else {
16         int tm = (tl+tr)/2;
17         build(tl, tm, 2*v);
18         build(tm+1, tr, 2*v+1);
19         down[v] = min(down[2*v], down[2*v+1]);
20         up[v] = min(up[2*v], up[2*v+1]);
21     }
22 }
23
```

```
24 void update(int i, int a,
25              int tl = 1,
26              int tr = n,
27              int v = 1)
28 {
29     if(tl > tr) return;
30     if(tl == tr) {
31         down[v] = a-tl;
32         up[v] = a+tl;
33     }
34     else {
35         int tm = (tr+tl)/2;
36         if(i <= tm) update(i, a, tl, tm, 2*v);
37         else update(i, a, tm+1, tr, 2*v+1);
38         down[v] = min(down[2*v], down[2*v+1]);
39         up[v] = min(up[2*v], up[2*v+1]);
40     }
41 }
42
43 long down_query(int l, int r,
44                 int tl = 1,
45                 int tr = n,
46                 int v = 1)
47 {
48     if(l > r) return INT_MAX;
49     if(l == tl && r == tr) return down[v];
50     else {
51         int tm = (tl+tr)/2;
52         return min(
53             down_query(l, min(r,tm), tl, tm, 2*v),
54             down_query(max(l,tm+1), r, tm+1, tr, 2*v+1)
55         );
56     }
57 }
58
59 long up_query(int l, int r,
60               int tl = 1,
61               int tr = n,
62               int v = 1)
63 {
64     if(l > r) return INT_MAX;
65     if(l == tl && r == tr) return up[v];
66     else {
67         int tm = (tl+tr)/2;
68         return min(
69             up_query(l, min(r,tm), tl, tm, 2*v),
70             up_query(max(l,tm+1), r, tm+1, tr, 2*v+1)
71         );
72     }
73 }
74
75 BEGIN
76 {
77     cin >> n >> q;
78     FOR(i,1,n+1) cin >> val[i];
```

```
79      build();
80
81      FOR(i,0,q) {
82          cin >> t;
83          if(t == 1) {
84              cin >> k >> x;
85              update(k,x);
86          }
87          else {
88              cin >> k;
89              cout << min(
90                  down_query(1,k)+k,
91                  up_query(k,n)-k
92              ) << endl;
93          }
94      }
95  }
96  END
```

8.4.2.9 Soma máxima de um subarray

Objetivo

Existe um array consistindo de n inteiros. Alguns valores do array serão atualizados, e após cada atualização, sua tarefa é relatar a soma máxima de subarray no array.

Estratégia

A base da solução é uma SegmentTree. É preciso armazenar em cada nó a soma total, o maior sufixo, o maior prefixo e a soma máxima de um subarray. A junção de nós é feita de forma intuitiva com base nesses valores.

Código

```
1
2  long seg[4*MAXN][4];
3  long seg_size, x;
4  int n, q, k;
5
6  // Node data
7  // 0 : max subarray sum
8  // 1 : max prefix sum
9  // 2 : max suffix sum
10 // 3 : total sum
11
12 void update(int k, long x)
13 {
14   k += seg_size;
15   FOR(i,0,3) seg[k][i] = max(x,0LL);
16   seg[k][3] = x;
17   while(k >>= 1 > 0) {
18     seg[k][0] = max({seg[2*k][0],seg[2*k+1][0],seg[2*k][2]+seg[2*k+1][1],0LL});
```

```
19      seg[k][1] = max({seg[2*k][1],seg[2*k][3]+seg[2*k+1][1],0LL});
20      seg[k][2] = max({seg[2*k][2]+seg[2*k+1][3],seg[2*k+1][2],0LL});
21      seg[k][3] = seg[2*k][3]+seg[2*k+1][3];
22    }
23  }
24
25  BEGIN
26  {
27    cin >> n >> q;
28    seg_size = 1 << (int)ceil(log2(n));
29
30   FOR(i,0,n) {
31      cin >> x;
32      update(i,x);
33    }
34
35    while(q--) {
36      cin >> k >> x;
37      update(k-1,x);
38      cout << seg[1][0] << endl;
39    }
40  }
41  END
```

8.4.2.10 Valores distintos

Objetivo

Dado um array de n inteiros, sua tarefa é processar q consultas dos seguintes tipos:

- Incrementar cada valor do intervalo $[a, b]$ com o valor u
- Qual o valor da posição k?

Estratégia

A solução é baseada em uma *BIT*. As consultas são processadas da direita para a esquerda, e, nessa ordem, a última ocorrência de um valor soma 1 à contagem da árvore. Sempre que um valor já estiver presente nela, é somado -1 para retirar sua influência e somado 1 referente à nova posição.

Para armazenar a última ocorrência de um valor, é usado um Map.

Código

```
1  int fen[MAXN], x[MAXN], sol[MAXN];
2  vector<pair<int,int>> queries[MAXN];
3  map<int,int> last_index;
4  int n, q, a, b;
5
6  void update(int k, int x)
7  {
8      while(k < n) {
```

```
9           fen[k] += x;
10          k |= (k+1);
11      }
12  }
13
14  int query(int b)
15  {
16      int ans = 0;
17      while(b >= 0) {
18          ans += fen[b];
19          b = (b&(b+1))-1;
20      }
21      return ans;
22  }
23
24  BEGIN
25  {
26      cin >> n >> q;
27      FOR(i,0,n) cin >> x[i];
28      FOR(i,0,q) {
29          cin >> a >> b;
30          queries[a-1].push_back({b-1,i});
31      }
32
33      FOR(k,1,n+1) {
34          if(last_index.count(x[n-k])) {
35              update(last_index[x[n-k]],-1);
36          }
37          last_index[x[n-k]] = n-k;
38          update(n-k,1);
39
40          for(auto qr : queries[n-k]) {
41              sol[qr.second] = query(qr.first);
42          }
43      }
44
45      FOR(i,0,q) cout << sol[i] << endl;
46  }
47  END
```

8.4.2.11 Valores distintos

Objetivo

Você tem um array que consiste em n inteiros. Os elementos do array são indexados de $1, 2, \ldots, n$. Você pode modificar o array usando a seguinte operação: escolher um elemento do array e aumentar seu valor em um.

Sua tarefa é processar q consultas da seguinte forma: ao considerarmos um subarray da posição a até a posição b, qual é o número mínimo de operações após as quais o subarray se torna crescente?

Um array é crescente se cada elemento for maior ou igual ao elemento anterior.

Estratégia

A solução é baseada na influência de um número na transformação de seus sucessores. Suponha que temos um elemento de valor x no índice i, e o próximo elemento estritamente maior que x está no índice j. Se não houver próximo elemento maior, deixe j ser igual a $n + 1$.

A contribuição de x será então

$$x \cdot (j - 1) - \sum_{k=i}^{j-1} arr[k] \tag{8.1}$$

A partir disso, iteramos o array na ordem inversa. Os elementos máximos alcançados durante a iteração são armazenados em uma pilha, sendo cada um deles responsáveis por uma contribuiçao que segue a fórmula acima desde que estjam totalmente compreendidos por uma consulta. Além disso, elementos que ficarem fora desse intervalo têm sua contribuição calculada manualmente.

Para verificar essa última condição, dado que os valores estarão consequentemente ordenados na pilha, basta usar uma busca binária para definir o intervalo compreendido pela consulta. Para deixar a soma de contribuições eficiente, é possível usar uma *BIT*, e, para calcular eficientemente o somatório da fórmula, basta manter uma soma de prefixos.

Código

```
1  int fen[MAXN], x[MAXN], sol[MAXN];
2  vector<pair<int,int>> queries[MAXN];
3  map<int,int> last_index;
4  int n, q, a, b;
5
6  void update(int k, int x)
7  {
8      while(k < n) {
9          fen[k] += x;
10         k |= (k+1);
11     }
12 }
13
14 int query(int b)
15 {
16     int ans = 0;
17     while(b >= 0) {
18         ans += fen[b];
19         b = (b&(b+1))-1;
20     }
21     return ans;
22 }
23
24 BEGIN
25 {
26     cin >> n >> q;
27     FOR(i,0,n) cin >> x[i];
28     FOR(i,0,q) {
29         cin >> a >> b;
30         queries[a-1].push_back({b-1,i});
31     }
32
33     FOR(k,1,n+1) {
34         if(last_index.count(x[n-k])) {
```

```
35              update(last_index[x[n-k]],-1);
36          }
37          last_index[x[n-k]] = n-k;
38          update(n-k,1);
39
40          for(auto qr : queries[n-k]) {
41              sol[qr.second] = query(qr.first);
42          }
43      }
44
45      FOR(i,0,q) cout << sol[i] << endl;
46  }
47  END
```

8.5 Matemática

8.5.0.1 Josephus Queries

Objetivo

Imagine um jogo onde há n crianças (numeradas de 1, 2, ..., n) em círculo. Durante o jogo, a cada dois segundos, uma criança é retirada do círculo, até que não sobre nenhuma. Sua tarefa é processar q perguntas no formato: "quando há n crianças, qual é a k-ésima criança que será retirada?"

Estratégia

A ideia é utilizar uma função recursiva, denotada por 'calculateRemovedChild()', para solucionar o problema.

Caso base: Se existir apenas uma criança restante, esta será a criança removida. **Caso recursivo**: Se o número da rodada, 'kthChild', for menor ou igual à metade do número total de crianças, então calcula-se qual criança será removida nessa rodada. Se 'kthChild' for maior que a metade do número total de crianças, chama-se a função recursivamente com metade do número total de crianças e ajusta-se 'kthChild' de acordo.

Código

```
1
2  ll solve(ll n, ll k){
3      if(n==1) return 1;
4
5      if(k<=(n+1)/2){
6          if(2*k>n) return (2*k)%n;
7          return 2*k;
8      }
9      ll temp=solve(n/2,k-(n+1)/2);
10     if(n%2==1) return 2*temp+1;
11     return 2*temp-1;
12 }
```

```
13
14 int main(){
15
16     ll q, n, k;
17     cin >> q;
18
19     while(cin >> n >> k){
20         cout << solve(n, k) << "\n";
21     }
22
23     return 0;
24 }
```

8.5.0.2 Exponentiation

Objetivo

Sua tarefa é calcular de forma eficiente os valores $a^{b^b} \mod (10^9 + 7)$.

Note que nesta tarefa assumimos que $0^0 = 1$.

Estratégia

Use ModPow

Código

```
1
2 using namespace std;
3
4 ll solve(ll n, ll k){
5     if(k==0) return 1;
6
7     ll mult = (n*n)%MOD;
8
9     if(k%2 == 1) return (n*solve(mult, (k-1)/2))%MOD;
10    return (solve(mult, k/2))%MOD;
11 }
12
13 int main(){
14
15     ll q, n, k;
16     cin >> q;
17
18     while(cin >> n >> k){
19         cout << solve(n, k) << "\n";
20     }
21
22     return 0;
23 }
```

8.5.0.3 Exponentiation II

Objetivo

Sua tarefa é calcular de forma eficiente os valores $a^{b^c} \mod (10^9 + 7)$.

Note que nesta tarefa assumimos que $0^0 = 1$.

Estratégia

ModPow

Código

```
1
2  ll solve(ll n, ll k, ll mod){
3      if(k==0) return 1;
4
5      ll mult = (n*n)%mod;
6
7      if(k%2 == 1) return (n*solve(mult, (k-1)/2, mod))%mod;
8      return (solve(mult, k/2, mod))%mod;
9  }
10
11 int main(){
12
13     ll q, n, k, j;
14     cin >> q;
15
16     while(cin >> n >> k >> j){
17         cout << solve(n, solve(k, j, MOD-1), MOD) << "\n";
18     }
19
20     return 0;
21 }
```

8.5.0.4 Counting Divisors

Objetivo

Dado um número inteiro, sua tarefa é relatar para cada número inteiro o número de seus divisores. Por exemplo, se $x = 18$, a resposta correta é 6 porque seus divisores são $1, 2, 3, 6, 9, 18$.

Estratégia

número de divisores da sessão de matematia

Código

```
1
2  ll sieve_size; // Limite de gera  o
3  bitset<10000010> bs; // Cada bit mostra se um n mero    primo ou n o
4  vector<int> primes; // Lista de primos
5
6
7  void sieve(ll upperbound) {
```

```
8       // O(n loglog n) ~= O(n)
9       sieve_size = upperbound + 1;
10      bs.set();
11      bs[0] = bs[1] = 0;
12      for (ll i = 2; i <= sieve_size; i++)
13          if (bs[i]) {
14              for (ll j = i * i; j <= sieve_size; j += i) bs[j] = 0;
15              primes.push_back((int)i);
16          }
17  }
18
19  ll numDiv(ll N) {
20      ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
21      while (PF * PF <= N) {
22          ll power = 0;
23          while (N % PF == 0) { N /= PF; power++; }
24          ans *= (power + 1);
25          PF = primes[++PF_idx];
26      }
27      if (N != 1) ans *= 2;
28      return ans;
29  }
30
31  int main(){
32      sieve(1000000);
33
34      int n, x;
35      cin >> n;
36
37      while(cin >> x){
38          cout << numDiv(x) << "\n";
39      }
40
41      return 0;
42  }
```

8.5.0.5 Common Divisors

Objetivo

Você recebe uma série de n inteiros positivos. Sua tarefa é encontrar dois números inteiros tais que seu máximo divisor comum seja o maior possível.

Estratégia

Fazer o GCD par a par da TLE. Crie um array D[N] onde D[i] indica de quantos números da entrada o número i é divisor, ache os divisores para cada número da entrada e por fim percorra D e printe o maior elemento em que D ¿ 1.

Código

```
1
2  int divisores[MAXN+1];
3
4  int mdc(int a, int b) { return b == 0 ? a : mdc(b, a % b); }
```

```
5  int mmc(int a, int b) { return a * (b / mdc(a, b)); }
6
7  void obterDivisores(int n){
8      for(int i=1; i*i <= n; i++){
9          if(n%i==0){
10             divisores[i]++;
11             if(i != n/i) divisores[n/i]++;
12         }
13     }
14 }
15
16 int main(){
17
18     ios_base::sync_with_stdio(0);
19         cin.tie(0);
20
21     int n, x, max_v=0;
22     cin >> n;
23
24     while(cin >> x){
25         max_v = max(max_v, x);
26         obterDivisores(x);
27     }
28
29     for(int i=max_v; i>=1; i--){
30         if(divisores[i]>=2){
31             cout << i << "\n";
32             break;
33         }
34     }
35
36     return 0;
37 }
```

8.5.0.6 Sum of Divisors

Objetivo

Deixar $\sigma(n)$ denotar a soma dos divisores de um número inteiro n. Por exemplo, $\sigma(12) = 1+2+3+4+6+12 = 28$. Sua tarefa é calcular a soma $\sum_{i=1}^{n} \sigma(i) \mod (10^9+7)$.

Estratégia

Para cada divisor i, o número de vezes que ele ocorre de 1 a n é $\lfloor \frac{n}{i} \rfloor$. Assim, a soma a ser calculada é equivalente a $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor i$.

Para que isso seja computado em $O(\sqrt{n})$, observamos que há apenas $O(\sqrt{n})$ valores distintos de $\lfloor \frac{n}{i} \rfloor$.

Dividimos nossas respostas em 2 partes: 1. Para a primeira parte, $i \leq \sqrt{n}$, simplesmente encontraremos a soma dos termos de 1 a $\sqrt{n}$. Isso pode ser feito em $O(\sqrt{n})$. 2. Para a segunda parte, $\sqrt{n} < i \leq n$. Embora este intervalo seja da ordem de $O(n)$, aqui notamos que $\lfloor \frac{n}{i} \rfloor$ assume valores apenas de 1 a $\sqrt{n}$. Então, encontraremos os pontos onde o valor de $\lfloor \frac{n}{i} \rfloor$ muda, mantendo contadores à esquerda e à direita, e usaremos a soma da PA para calcular a resposta.

Código

```
1
2  ll total_sum(ll i, ll f){
3      // soma de p.a. o TWO_MOD_INV    pelo dividir por 2
4      return ((((f-i+1)%MOD) * ((i + f) % MOD) % MOD) * TWO_MOD_INV % MOD);
5  }
6
7  ll sumOfAllDiv(ll n){
8      ll att=1, ans=0;
9
10     while(att <= n){
11         // att    o divisor
12         ll q = n/att; //q    quantas vezes essedivisor aparece
13
14         ll att_n = (n/q); // aqui a gente vai para o
15                           // proximo divisor que aparece uma
16                          //  quantidade de vezes diferente de att
17
18         ans = (ans+q*total_sum(att, att_n))%MOD;
19
20         att = att_n + 1;
21     }
22
23     return ans;
24 }
25
26 int main(){
27
28     ll n;
29     cin >> n;
30
31     cout << sumOfAllDiv(n) << "\n";
32
33     return 0;
34 }
```

8.5.0.7 textotextotexto

Objetivo

Dado um número inteiro, sua tarefa é encontrar o número, a soma e o produto de seus divisores. Como exemplo, consideremos o número 12:

O número de divisores é 6 (eles são 1, 2, 3, 4, 6, 12) A soma dos divisores é $1 + 2 + 3 + 4 + 6 + 12 = 28$ O produto dos divisores é $1 \cdot 2 \cdot 3 \cdot 4 \cdot 6 \cdot 12 = 1728$

Como o número de entrada pode ser grande, ele é dado como uma fatoração em primos.

Ele te da na entrada os fatores e quantas vezes eles aparecem.

Estratégia

Formulas da sessão de matematica

Código

```
1
2  ll exponentiation_mod(ll n, ll k, ll mod){
3      if(k==0) return 1;
4
5      ll mult = (n*n)%mod;
6
7      if(k%2 == 1) return (n*exponentiation_mod(mult, (k-1)/2, mod))%mod;
8      return (exponentiation_mod(mult, k/2, mod))%mod;
9  }
10
11
12 int main(){
13
14     ll n,p, a, num=1, num_div=1, sum_div=1, div_prod=1, div_cnt2=1;
15     cin >> n;
16
17     while(cin >> p >> a){
18         num = (num*exponentiation_mod(p,a,MOD))%MOD;
19         num_div = (num_div * (a+1))%MOD;
20
21         sum_div = sum_div*(
22                                (
23                                    (
24                                        exponentiation_mod(p, a+1, MOD)-1)%MOD
25                                    )*
26                                (
27                                    exponentiation_mod(p-1, MOD-2, MOD)
28                                )%MOD
29                            )%MOD;
30
31         div_prod = exponentiation_mod(div_prod, a + 1, MOD) *
32                             exponentiation_mod(exponentiation_mod(p, (a * (a + 1) / 2), MOD),
33                                                div_cnt2, MOD) % MOD;
34         div_cnt2 = div_cnt2 * (a + 1) % (MOD - 1);
35     }
36
37     cout << num_div << "␣" << sum_div << "␣" << div_prod << "\n";
38
39     return 0;
40 }
```

8.5.0.8 Prime Multiples

Objetivo

Você recebe k números primos distintos $a_1, a_2, \ldots, a_k$ e um número inteiro n. Sua tarefa é calcular quantos dos primeiros n números positivos são divisíveis por pelo menos um dos números primos dados.

Estratégia

inclusão exclusão.

Código

```
1
2  ll solve(ll n, vector<ll> p){
3      ll sum = 0;
4      vector<ll> ans(21,0);
5      for(ll mask=1; mask<(1<<p.size()); mask++){
6          ll mult=1, bit=0, sai=0;
7          for(int i=0; i<(ll)p.size(); i++){
8              if(mask&(1<<i)){
9                  bit++;
10                 if(mult > n/p[i]+1){
11                     sai=1;
12                     break;
13                 }
14                 mult*=p[i];
15             }
16         }
17         if(sai) continue;
18         ans[bit] += n/mult;
19     }
20
21     int m=-1;
22     for(int i=1; i<21; i++){
23         m*=-1;
24         sum += m*ans[i];
25     }
26
27     return sum;
28 }
29
30 int main(){
31
32     ll n, k;
33     cin >> n >> k;
34
35     vector<ll> p(k);
36     for(int i=0; i<k; i++){cin >> p[i];}
37
38     cout << solve(n, p) << "\n";
39
40     return 0;
41 }
```

8.5.0.9 Counting Coprime Pairs

Objetivo

Dada uma lista de n inteiros positivos, sua tarefa é contar o número de pares de inteiros que são coprimos (ou seja, cujo máximo divisor comum é um).

Estratégia

Vamos usar a função de Möbius $\mu(k)$ para isso. A resposta para o problema é $\sum_{k=1}^{max(x[i])} \mu(k)\binom{d(k)}{2}$. Aqui, $d(k)$ é o número de inteiros na sequência fornecida que são divisíveis por k. Vemos que esse resultado vem do princípio da inclusão-exclusão. O termo na soma corresponde à escolha de dois números que são múltiplos de k, e $\mu(k)$ decide se o adicionamos ou não. Note que, na inclusão-exclusão, não consideramos k que não são livres de quadrados, pois isso não adiciona nenhum efeito à nossa resposta. A complexidade de tempo é $O(10^6 \log(10^6))$. Verifique o código abaixo para a implementação da função de Möbius.

Código

```
1
2  ll sieve_size; // Limite de gera  o
3  bitset<10000010> bs; // Cada bit mostra se um n mero    primo ou n o
4  vector<ll> primes; // Lista de primos
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  void sieve(ll upperbound) {
10     // O(n loglog n) ~= O(n)
11     sieve_size = upperbound + 1;
12     bs.set();
13     bs[0] = bs[1] = 0;
14     for (ll i = 2; i <= sieve_size; i++)
15         if (bs[i]) {
16             for (ll j = i * i; j <= sieve_size; j += i) bs[j] = 0;
17             primes.push_back((ll)i);
18         }
19 }
20
21 vector<ll> setPrimeFactors(ll N){
22     vector<ll> factors;
23     ll PF_idx = 0;
24     ll PF = primes[PF_idx];
25     while (PF * PF <= N) {
26         int s=0;
27         while (N % PF == 0) {
28             N /= PF;
29             if(s) continue;
30             factors.push_back(PF);
31             s=1;
32         }
33         PF = primes[++PF_idx];
34     }
35     if (N != 1) factors.push_back(N);
36     return factors;
37 }
38
39
40 ll cnt[mxN];
41
42 int main(){
43     sieve(1000000);
44     ll n, a, ans=0;
45
```

```
46      cin >> n;
47
48      while(cin >> a){
49          vector<ll> v = setPrimeFactors(a);
50
51          ll nf = v.size();
52          for(ll mask=1; mask<(1<<nf); mask++){
53              ll p=1, bit=0;
54
55              for(ll i=0; i<nf; i++){
56                  if(mask&(1<<i)){
57                      bit++;
58                      p*=v[i];
59                  }
60              }
61
62              ll f=1;
63              if(bit%2==0) f=-1;
64
65              ans += f*cnt[p];
66              cnt[p]++;
67          }
68      }
69
70      cout << n*(n-1)/2 - ans << "\n";
71
72      return 0;
73  }
```

8.5.0.10 Binomial Coefficients

Objetivo

Sua tarefa é calcular n coeficientes binomiais modulo 10^9+7. Um coeficiente binomial $\binom{a}{b}$ pode ser calculado usando a fórmula $\frac{a!}{b!(a-b)!}$. Assumimos que a e b são inteiros e $0 \leq b \leq a$.

Estratégia

Coeficiente Binomial normal

Código

```
1
2   ll fat[(int)1e6+10];
3   ll inve[(int)1e6+10];
4
5   ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
6   ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
7
8   ll exp(ll x, unsigned ll y, ll p){
9       ll res=1; x=x%p;
10      while(y>0){
11          if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
```

```
12      }
13      return res;
14  }
15
16  void fatorialEinv(){
17      fat[0] = 1;
18      inve[0] = 1;
19      for(int i=1; i<=1e6; i++){
20          fat[i] = (fat[i-1]*i)%MOD;
21          inve[i] = exp(fat[i],MOD-2,MOD);
22      }
23  }
24
25  ll solve(ll a, ll b){
26      return fat[a]*inve[b]%MOD*inve[a-b]%MOD;
27  }
28
29  int main(){
30      fatorialEinv();
31
32      ll n, a, b;
33      cin >> n;
34
35      while (cin >> a >> b){
36          cout << solve(a, b) << "\n";
37      }
38
39      return 0;
40  }
```

8.5.0.11 Creating Strings II

Objetivo

Dada uma string, sua tarefa é calcular o número de diferentes strings que podem ser criadas usando seus caracteres.

Estratégia

dado a string A. a resposta vai ser —A—! dividido pelos fatoriais da quantidade que cada letra aparece.

Código

```
1
2  ll fat[(int)1e6+10];
3  ll inve[(int)1e6+10];
4  vector<int> b(26,0);
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  ll exp(ll x, unsigned ll y, ll p){
10      ll res=1; x=x%p;
11      while(y>0){
12          if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
```

```
13      }
14      return res;
15  }
16
17  void fatorialEinv(){
18      fat[0] = 1;
19      inve[0] = 1;
20      for(int i=1; i<=1e6; i++){
21          fat[i] = (fat[i-1]*i)%MOD;
22          inve[i] = exp(fat[i],MOD-2,MOD);
23      }
24  }
25
26  ll solve(ll a){
27      ll ans = fat[a];
28      for(auto n: b){
29          ans = ans*inve[n]%MOD;
30      }
31      return ans;
32  }
33
34  int main(){
35      fatorialEinv();
36
37      int a=0;
38      char c;
39
40      while(cin >> c){
41          a++;
42          b[c-'a']++;
43      }
44
45      cout << solve(a) << "\n";
46
47      return 0;
48  }
```

8.5.0.12 Distributing Apples

Objetivo

Existem n crianças e m maçãs que serão distribuídas para elas. Sua tarefa é contar o número de maneiras que isso pode ser feito. Por exemplo, se $n = 3$ e $m = 2$, existem 6 maneiras: [0,0,2], [0,1,1], [0,2,0], [1,0,1], [1,1,0] e [2,0,0].

Estratégia

Boxes and balls normal

Código

```
1
2  ll fat[(int)2e6+10];
3  ll inve[(int)2e6+10];
4  vector<int> b(26,0);
```

```
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  ll expo(ll x, unsigned ll y, ll p){
10     ll res=1; x=x%p;
11     while(y>0){
12         if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
13     }
14     return res;
15 }
16
17 void fatorialEinv(){
18     fat[0] = 1;
19     inve[0] = 1;
20     for(int i=1; i<=2e6; i++){
21         fat[i] = (fat[i-1]*i)%MOD;
22         inve[i] = expo(fat[i],MOD-2,MOD);
23     }
24 }
25
26 ll bn(ll a, ll b){
27     return fat[a]*inve[b]%MOD*inve[a-b]%MOD;
28 }
29
30 ll box_and_balls(ll a, ll b){
31     // return fat[a+b-1]*inve[a-1]%MOD*inve[b]%MOD;
32     return bn(a+b-1,b);
33 }
34
35 int main(){
36     fatorialEinv();
37
38     int n, m;
39     cin >> n >> m;
40
41     cout << box_and_balls(n, m) << "\n";
42
43     return 0;
44 }
```

8.5.0.13 Christmas Party

Objetivo

Existem n crianças em uma festa de Natal, e cada uma delas trouxe um presente. A ideia é que cada pessoa receba um presente trazido por outra pessoa. De quantas maneiras os presentes podem ser distribuídos?

Estratégia

Este é um problema padrão de encontrar deranjos em uma sequência. A fórmula recursiva é $D(n) = (D(n-1) + D(n-2))(n-1)$.

Código

```
1
2  int main()
3  {
4      ios::sync_with_stdio(0);
5      cin.tie(0);
6      cout.tie(0);
7      ll n;
8      cin>>n;
9      ll d[n+1];
10     d[1]=0;
11     d[2]=1;
12     for(ll i=3;i<=n;i++)
13     {
14         d[i]=(((d[i-1]+d[i-2])%MOD)*(i-1))%MOD;
15     }
16     cout<<d[n];
17 }
```

8.5.0.14 Bracket Sequences I

Objetivo

Sua tarefa é calcular o número de sequências de parênteses válidas de comprimento n. Por exemplo, quando $n = 6$, existem 5 sequências.

Estratégia

Numero de catalan normal.

Código

```
1
2  ll fat[(int)2e6+10];
3  ll inve[(int)2e6+10];
4  vector<int> b(26,0);
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  ll expo(ll x, unsigned ll y, ll p){
10     ll res=1; x=x%p;
11     while(y>0){
12         if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
13     }
14     return res;
15 }
16
17 void fatorialEinv(){
18     fat[0] = 1;
19     inve[0] = 1;
20     for(int i=1; i<=2e6; i++){
21         fat[i] = (fat[i-1]*i)%MOD;
```

```
22          inve[i] = expo(fat[i],MOD-2,MOD);
23      }
24  }
25
26  ll bn(ll a, ll b){
27      return fat[a]*inve[b]%MOD*inve[a-b]%MOD;
28  }
29
30  ll catalan(ll n){
31      if(n%2==1) return 0;
32      n/=2;
33      return bn(2*n,n)*expo(n+1,MOD-2,MOD)%MOD;
34  }
35
36  int main(){
37      fatorialEinv();
38
39      int n;
40      cin >> n;
41
42      cout << catalan(n) << "\n";
43
44      return 0;
45  }
```

8.5.0.15 Bracket Sequences II

Objetivo

Sua tarefa é calcular o número de sequências de parênteses válidas de comprimento n quando um prefixo da sequência é dado.

Estratégia

Número de catalan mas Cat(numero de '()' que ainda faltam + N de '(' sozinhos)

Código

```
1
2  ll fat[(int)2e6+10];
3  ll inve[(int)2e6+10];
4  vector<int> b(26,0);
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  ll expo(ll x, unsigned ll y, ll p){
10     ll res=1; x=x%p;
11     while(y>0){
12         if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
13     }
14     return res;
15 }
16
```

```
17 void fatorialEinv(){
18     fat[0] = 1;
19     inve[0] = 1;
20     for(int i=1; i<=2e6; i++){
21         fat[i] = (fat[i-1]*i)%MOD;
22         inve[i] = expo(fat[i],MOD-2,MOD);
23     }
24 }
25
26 ll bn(ll a, ll b){
27     return fat[a]*inve[b]%MOD*inve[a-b]%MOD;
28 }
29
30 ll catalan(ll m){
31     ll n = m;
32     if(n%2==1) return 0;
33     n/=2;
34
35     ll k=0, O=0;
36     char c;
37     while(cin >> c){
38         if(c=='('){
39             k++;
40             O++;
41         }else{
42             k--;
43         }
44         if(k<0) return 0;
45     }
46
47     n -= O;
48     if(k<0 || n<0 || 2*n+k<n){
49         return 0;
50     }
51
52     ll cat  = bn(2*n+k,n);
53     ll mult = (k+1)%MOD*expo(n+k+1,MOD-2,MOD)%MOD;
54
55     return cat*mult%MOD;
56     // return bn(2*n,n)*expo(n+1,MOD-2,MOD)%MOD;
57 }
58
59 int main(){
60     fatorialEinv();
61
62     int n;
63     cin >> n;
64
65     cout << catalan(n) << "\n";
66
67     return 0;
68 }
```

8.5.0.16 Counting Necklaces

Objetivo

Sua tarefa é contar o número de colares diferentes que consistem em n pérolas e cada pérola tem m cores possíveis. Dois colares são considerados diferentes se não for possível girar um deles para que eles se pareçam iguais.

Estratégia

Teoria dos grupos. Número de elementos unicos = soma das possibilidades simetricas / Número de movimento

Código

```
1
2  ll fat[(int)2e6+10];
3  ll inve[(int)2e6+10];
4  vector<int> b(26,0);
5
6  ll mdc(ll a, ll b) { return b == 0 ? a : mdc(b, a % b); }
7  ll mmc(ll a, ll b) { return a * (b / mdc(a, b));}
8
9  ll expo(ll x, unsigned ll y, ll p){
10     ll res=1; x=x%p;
11     while(y>0){
12         if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
13     }
14     return res;
15 }
16
17 void fatorialEinv(){
18     fat[0] = 1;
19     inve[0] = 1;
20     for(int i=1; i<=2e6; i++){
21         fat[i] = (fat[i-1]*i)%MOD;
22         inve[i] = expo(fat[i],MOD-2,MOD);
23     }
24 }
25
26 ll bn(ll a, ll b){
27     return fat[a]*inve[b]%MOD*inve[a-b]%MOD;
28 }
29
30 ll burnside(ll n, ll m){
31     ll ans = 0;
32
33     for(int i=0; i<n; i++){
34         ans = (ans+expo(m,mdc(i,n),MOD)*expo(n,MOD-2,MOD)%MOD)%MOD;
35     }
36
37     return ans;
38 }
39
40 int main(){
41     // fatorialEinv();
42
43     int n, m;
44     cin >> n >> m;
45
```

```
46      cout << burnside(n, m) << "\n";
47
48      return 0;
49  }
```

8.5.0.17 Counting Grids

Objetivo

Sua tarefa é contar o número de diferentes grades $n \times n$ onde cada quadrado é preto ou branco. Duas grades são consideradas diferentes se não for possível girar uma delas para que pareçam iguais.

Estratégia

Teoria dos grupos. Número de elementos unicos = soma das possibilidades simetricas / Número de movimento

Código

```
1
2  ll exp(ll x, ll y, ll md){
3      ll ans = 1;
4      x = x%md;
5      while (y > 0) {
6          if (y&1)
7              ans = ans*x%md;
8          y = y>>1;
9          x = x*x%md;
10     }
11     return ans;
12 }
13
14 int main(){
15     ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);
16
17     ll n, r0, r90, r180, ans=0;
18     cin >> n;
19
20     r0 = n*n;
21     r90 = 1;
22     r180 = 1;
23
24     if(n > 1 && n&1){
25         r90 = (((n/2+1)*(n/2))+1);
26         r180 = (((n)*(n/2))+(n/2+1));
27         // r90 = (n+3)*(n-1)/4 - (n-1)/2 + 1;
28         // r180 = (n+3)*(n-1)/2 - (n-1) + 1;
29     }else if(n!=1){
30         r90 = ((n/2)*(n/2));
31         r180 = ((n)*(n/2));
32         // r90 = n*(n+2)/4 - n/2;
33         // r180 = n*(n+2)/2 - n;
34     }
35
36     ans = (ans + exp(2, r0, MOD)) % MOD; //0 deg
```

```
37      ans = (ans + exp(2, r90, MOD)) % MOD; //90 deg
38      ans = (ans + exp(2, r180, MOD)) % MOD; //180 deg
39      ans = (ans + exp(2, r90, MOD)) % MOD; //270 deg
40      ans = (ans * exp(4, MOD-2, MOD)) % MOD; // media
41
42      cout << ans << "\n";
43      return 0;
44  }
```

8.5.0.18 Fibonacci Numbers

Objetivo

Os números de Fibonacci podem ser definidos da seguinte forma:

$F_0 = 0$ $F_1 = 1$ $F_n = F_{n-2} + F_{n-1}$

Sua tarefa é calcular o valor de F_n para um dado n.

Estratégia

Fibonacci em $\log n$ com recorrencia linear

Código

```
1   #define ll long long
2   #define MOD 1000000007
3
4
5   ll expo(ll x, unsigned ll y, ll p){
6       ll res=1; x=x%p;
7       while(y>0){
8           if (y&1) res= (res*x)%p; y=y>>1; x=(x*x)%p;
9       }
10      return res;
11  }
12
13  map<ll,ll> fib;
14  ll fibonate(ll n){
15      if(fib.count(n)) return fib[n];
16      if(n==0) return 0;
17      if(n==1||n==2) return 1;
18
19      ll k, f1, f2;
20      if(n%2==0){
21          k=n/2;
22          f1 = fibonate(k);
23          f2 = fibonate(k-1);
24
25          return fib[n]=(2*f2%MOD+f1)%MOD*f1%MOD;
26      }
27      else{
28          k=(n+1)/2;
29          f1 = fibonate(k);
30          f2 = fibonate(k-1);
```

```
31
32          return fib[n]=((f1*f1%MOD)+(f2*f2%MOD))%MOD;
33      }
34  }
35
36  int main(){
37      ios::sync_with_stdio(0);
38      cin.tie(0);
39      cout.tie(0);
40
41      ll n;
42      cin >> n;
43
44      cout << fibonate(n) << "\n";
45
46      return 0;
47  }
```

8.5.0.19 Throwing Dice

Objetivo

Sua tarefa é calcular o número de maneiras de obter uma soma n ao lançar dados. Cada lançamento resulta em um número inteiro entre 1 e 6. Por exemplo, se $n = 10$, algumas maneiras possíveis são 3+3+4, 1+4+1+4 e 1+1+6+1+1.

Estratégia

recorrencia linear.

Código

```
1
2  ll N, x[6][6], y[6][6];
3
4  void init(){
5      for(int i = 0; i < 6; i++)
6          x[0][i] = 1;
7      for(int i = 0; i < 5; i++)
8          x[i+1][i] = 1;
9      for(int i = 0; i < 6; i++)
10         y[i][i] = 1;
11 }
12
13 void mult(ll A[6][6], ll B[6][6]){
14     ll C[6][6];
15     memset(C, 0, sizeof(C));
16     for(int i = 0; i < 6; i++){
17         for(int j = 0; j < 6; j++){
18             for(int k = 0; k < 6; k++){
19                 C[i][j] += A[i][k] * B[k][j];
20                 C[i][j] %= MOD;
21             }
22         }
23     }
```

```
24      for(int i = 0; i < 6; i++)
25          for(int j = 0; j < 6; j++)
26              A[i][j] = C[i][j];
27  }
28
29  int main(){
30      init();
31
32      scanf("%lld", &N);
33      while(N){
34          if(N&1)
35              mult(y, x);
36          mult(x, x);
37          N >>= 1;
38      }
39
40      printf("%lld\n", y[0][0]);
41  }
```

8.5.0.20 Graph Paths I

Objetivo

Considere um grafo direcionado que possui n nós e m arestas. Sua tarefa é contar o número de caminhos do nó 1 para o nó n com exatamente k arestas.

Estratégia

Se elevar a matriz de adj a K potencia a posição [i][j] indicara se é possivel ir de i a j em k passos.

Código

```
1  vector<vector<ll>> mul(vector<vector<ll>> a, vector<vector<ll>> b) {
2      vector<vector<ll>> c(a.size(), vector<ll>(b[0].size()));
3      for (int i = 0; i < a.size(); i++)
4          for (int j = 0; j < b[0].size(); j++)
5              for ( int k = 0; k < a[0].size(); k++)
6                  (c[i][j] += a[i][k]*b[k][j]%MOD)%=MOD;
7      return c;
8  }
9
10
11 vector<vector<ll>> exp( vector<vector<ll>> x, int y) {
12     vector<vector<ll>> r(x.size(), vector<ll>(x.size()));
13     for ( int i = 0; i < x.size(); i++) r[i][i] = 1;
14     while (y>0){
15         if (y&1) {
16             r = mul(r,x);
17         }
18         y=y>>1;
19         x = mul(x,x);
20     }
21     return r;
22 }
```

```
23
24 int main(){
25
26     int n, m, k;
27     cin >> n >> m >> k;
28     vector<vector<ll>> grafo(n, vector<ll>(n, 0));
29
30     for(int i=0; i<m; i++){
31         ll a, b;
32         cin >> a >> b;
33         grafo[a-1][b-1]++;
34     }
35
36     grafo = exp(grafo, k);
37
38     cout  << grafo[0][n-1] << "\n";
39
40     return 0;
41 }
```

8.5.0.21 Graph Paths II

Objetivo

Considere um grafo direcionado ponderado com n nós e m arestas. Sua tarefa é calcular o comprimento do caminho mínimo do nó 1 para o nó n com exatamente k arestas.

Estratégia

Se elevar a matriz de adj de um grafo com pesos a K potencia e a posição [i][j] indicara se é possivel ir de i a j em k passos.

Código

```
1  const ll inf = 1LL<<60;
2
3  vector<vector<ll>> mul_min(vector<vector<ll>> a, vector<vector<ll>> b) {
4      vector<vector<ll>> c(a.size(), vector<ll>(b[0].size(), inf));
5      for (int i = 0; i < a.size(); i++)
6          for (int j = 0; j < b[0].size(); j++)
7              for ( int k = 0; k < a[0].size(); k++)
8                  c[i][j] = min(c[i][j], a[i][k]+b[k][j]);
9      return c;
10 }
11
12
13 vector<vector<ll>> exp( vector<vector<ll>> x, int y) {
14     vector<vector<ll>> r(x.size(), vector<ll>(x.size()));
15     ll f = 0;
16     while (y>0){
17         if (y&1) {
18             if (f) r = mul_min(r,x);
19             else r = x, f =1;
20         }
```

```
21          y=y>>1;
22          x = mul_min(x,x);
23      }
24      return r;
25  }
26
27  int main(){
28
29      int n, m, k;
30      cin >> n >> m >> k;
31      vector<vector<ll>> grafo(n, vector<ll>(n, inf));
32
33      for(int i=0; i<m; i++){
34          ll a, b, c;
35          cin >> a >> b >> c;
36          grafo[a-1][b-1] = min(grafo[a-1][b-1], c);
37      }
38
39      grafo = exp(grafo, k);
40
41      if(grafo[0][n-1] < inf) cout  << grafo[0][n-1] << "\n";
42      else  cout << "-1\n";
43
44      return 0;
45  }
```

8.5.0.22 Stick Game

Objetivo

Considere um jogo onde dois jogadores removem varetas de uma pilha. Os jogadores se revezam e o jogador que remover a última vareta vence o jogo. Um conjunto $P = \{p_1, p_2, \ldots, p_k\}$ determina os movimentos permitidos. Por exemplo, se $P = \{1, 3, 4\}$, um jogador pode remover 1, 3 ou 4 varetas. Sua tarefa é descobrir para cada número de varetas $1, 2, \ldots, n$ se o primeiro jogador tem uma posição vencedora ou perdedora.

Estratégia

Reduza a Nim, calculando o Groundy para cada número de palitos

Código

```
1   int flag[MAXV];
2
3   int main(){
4
5       ll n, k;
6       cin >>  n >> k;
7       vector<ll> mov(k, 0);
8       vector<ll> DP(n+1, 0);
9
10      for(int i=0; i<k; i++){
11          cin >> mov[i];
12      }
13
```

```
14      for(int i=1; i<=n; i++){
15          int l =0;
16          memset(flag,0,MAXV);
17          for(int j=0; j<k; j++){
18              if(i-mov[j]>=0) flag[DP[i-mov[j]]]=1;
19          }
20          for(l; flag[l]; l++);
21          DP[i] = l;
22      }
23
24      for(int i=1; i<=n; i++){
25          if(DP[i]>0) cout << "W";
26          else cout << "L";
27      }
28
29      cout << "\n";
30
31      return 0;
32  }
```

8.5.0.23 Nim Game I

Objetivo

Há n pilhas de varetas e dois jogadores que se alternam. Em cada jogada, um jogador escolhe uma pilha não vazia e remove qualquer número de varetas. O jogador que remover a última vareta vence o jogo. Sua tarefa é descobrir quem vence se ambos os jogadores jogarem de forma otimizada.

Estratégia

Nim normal, use xor para verificar quem ganha, xor=0 segundo e xor=1 é o primeiro.

Código

```
1   int main(){
2
3       ll t, n;
4       cin >> t;
5
6       while(cin >> n){
7           int a, ans=0;
8           for(int i = 0; i<n; i++){
9               cin >> a;
10              ans^=a;
11          }
12          if(ans==0) cout << "second\n";
13          else cout << "first\n";
14      }
15
16      return 0;
17  }
```

8.5.0.24 Nim Game II

Objetivo

Há n pilhas de varetas e dois jogadores que se alternam. Em cada jogada, um jogador escolhe uma pilha não vazia e remove qualquer número de varetas. O jogador que remover a última vareta vence o jogo. Sua tarefa é descobrir quem vence se ambos os jogadores jogarem de forma otimizada.

Estratégia

variação do Nim onde ao inves de fazer o xor com o tamanho da pilha fazemos com o tamanho da pilha modulo K onde k é 1+ o maior numero q pode ser removido.

Código

```
1  int main(){
2
3      ll t, n;
4      cin >> t;
5
6      while(cin >> n){
7          int a, ans=0;
8          for(int i = 0; i<n; i++){
9              cin >> a;
10             ans^=a%(4);
11         }
12         if(ans==0) cout << "second\n";
13         else cout << "first\n";
14     }
15
16 }
```

8.5.0.25 Stair Game

Objetivo

Há uma escada consistindo de n degraus, numerados de $1, 2, \ldots, n$. Inicialmente, cada degrau tem um certo número de bolas. Há dois jogadores que se alternam. Em cada jogada, um jogador escolhe um degrau k onde $k \neq 1$ e que tenha pelo menos uma bola. Então, o jogador move qualquer número de bolas do degrau k para o degrau $k-1$. O jogador que fizer o último movimento vence o jogo. Sua tarefa é descobrir quem vence o jogo quando ambos os jogadores jogam de forma otimizada. Note que, se não houver movimentos possíveis, o segundo jogador vence.

Estratégia

Nim arvore, so considera no xor as possições impares.

Código

```
1  int main(){
2
3      ll t, n;
```

```
4       cin >> t;
5
6       while(cin >> n){
7           int a, ans=0;
8           for(int i = 0; i<n; i++){
9               cin >> a;
10              if(i&1) ans^=a;
11          }
12          if(ans==0) cout << "second\n";
13          else cout << "first\n";
14      }
15
16  }
```

8.5.0.26 Grundy's Game

Objetivo

Há uma pilha de n moedas e dois jogadores que se alternam. Em cada jogada, um jogador escolhe uma pilha e a divide em duas pilhas não vazias que tenham um número diferente de moedas. O jogador que fizer a última jogada vence o jogo. Sua tarefa é descobrir quem vence se ambos os jogadores jogarem de forma otimizada.

Estratégia

Variação de Nim, achando o Grundy

Código

```
1   ll mex(vector<ll> v){
2       set<ll> s;
3       for(int i=0;i<v.size();i++)
4       {
5           s.insert(v[i]);
6       }
7       for(int i=0;i<1000001;i++)
8       {
9           if(s.count(i)==0) return i;
10      }
11  }
12
13  int main(){
14
15      ll t,n;
16      cin >> t;
17      vector<ll> DP(limit+1, 0);
18
19      for(int i=3; i<=limit; i++){
20          int l=0;
21          vector<ll> flag;
22          for(int j=1; 2*j<i; j++){
23              flag.push_back(DP[j]^DP[i-j]);
24          }
25          // for(l; flag[l]; l++);
26          DP[i] = mex(flag);
```

```
27      }
28
29      while(cin >> n){
30          if(n>=limit) cout<<"first\n";
31          else if(DP[n]) cout<<"first\n";
32          else cout<<"second\n";
33      }
34
35      return 0;
36  }
```

8.6 Strings

8.6.0.1 Contagem de combinações de palavras

Objetivo

Você recebe uma string de comprimento n e um dicionário contendo k palavras. De quantas maneiras você pode criar a string usando as palavras?

Estratégia

A solução usa programação dinâmica juntamente com uma trie. A trie armazena todas as palavras possíveis no dicionário e a programação dinâmica calcula o número de maneiras de criar a string. $dp[i]$ representa o número de maneiras de criar a substring começando no índice i. Para calcular $dp[i]$, iteramos sobre todos os prefixos da string começando no índice i e verificamos se ele existe na trie. Se existir, adicionamos $dp[j+1]$ a $dp[i]$, onde j é o índice do final do prefixo.

Código

```
1   int trie[MAXW][26];
2   int node_count;
3   bool stop[MAXW];
4
5   void insert(string word) {
6
7       int node = 0;
8       for(auto c: word) {
9           if(trie[node][c-'a'] == 0) trie[node][c-'a'] = ++node_count;
10          node = trie[node][c-'a'];
11      }
12      stop[node] = true;
13  }
14
15  int main() {
16
17      string str; cin >> str;
18      int n; cin >> n;
19      while(n--) {
```

```
20          string word; cin >> word;
21          insert(word);
22      }
23
24      int dp[str.size() + 1];
25      memset(dp, 0, sizeof(dp));
26
27      dp[str.size()] = 1;
28      for (int i = str.size()-1; i >= 0; i--) {
29          int node = 0;
30          for (int j = i; j < str.size(); j++) {
31              if(trie[node][str[j]-'a'] == 0) break;
32
33              node = trie[node][str[j]-'a'];
34
35              if(stop[node]) dp[i] = (dp[i] + dp[j+1]) % MOD;
36          }
37      }
38      cout << dp[0] << endl;
39      return 0;
40  }
```

8.6.0.2 Algoritmo KMP - Casamento de padrões

Objetivo

Dada uma string e um padrão, sua tarefa é contar o número de posições onde o padrão ocorre na string.

Estratégia

A resolução é baseada no algoritmo KMP (Knuth-Morris-Pratt), o qual utiliza o maior prefixo que também é sufixo para decidir o proximo caractere do padrão a ser comparado. Uma dica é concatenar o padrão com o texto dessa maneira podemos analisar o casamento pelo lps.

Código

```
1   int lps[MAXN]; // Armazena o maior prefixo que tamb m    sufixo
2
3   void calculateLPS(string str) {
4
5       int n = str.size();
6       for (int i = 1, j = 0; i < n; i++) {
7           while(j > 0 && str[i] != str[j]) j = lps[j-1];
8
9           if(str[i] == str[j]) j++;
10          lps[i] = j; // Armazenamos o maior prefixo que tamb m    sufixo para a posi  o i
11      }
12  }
13
14  int main() {
15
16      string text, pattern;
17      cin >> text >> pattern;
18
```

```
19      string str = pattern + '#' + text;
20      calculateLPS(str);
21
22      int ans = 0;
23      for (auto v: lps) {
24          if(v == pattern.size()) ans++; // Se o maior prefixo que tamb m    sufixo    igual ao
25      }
26      cout << ans << endl;
27
28      return 0;
29  }
```

8.6.0.3 Bordas de uma string

Objetivo

A borda de uma string é um prefixo que também é um sufixo da string, mas não de toda a string. Por exemplo, as bordas de abcababcab são abc, abcab. Sua tarefa é encontrar todos os comprimentos das bordas de uma determinada string.

Estratégia

A resolução é baseada no algoritmo KMP (Knuth-Morris-Pratt), o qual utiliza o maior prefixo que também é sufixo para decidir o proximo caractere do padrão a ser comparado. Como a borda é um prefixo que também é sufixo, podemos usar o array lps para encontrar todas as bordas.

Código

```
1   int lps[MAXN];
2
3   void calculateLPS(string str) {
4
5       int n = str.size();
6       for (int i = 1, j = 0; i < n; i++) {
7           while(j > 0 && str[i] != str[j]) j = lps[j-1];
8           if(str[i] == str[j]) j++;
9           lps[i] = j;
10      }
11  }
12
13  int main() {
14
15      string str; cin >> str;
16
17      calculateLPS(str); // Calcula o array lps
18
19      set<int> st;
20      int v = lps[str.size()-1]; // O  ltimo  valor do array lps    o tamanho da maior borda
21      while(v > 0) { // Enquanto houver bordas
22          st.insert(v); // Insere o tamanho da borda no set
23          v = lps[v-1]; // Encontra a pr xima borda
24      }
25
```

```
26      for (auto v: st) cout << v << "␣"; // Imprime os tamanhos das bordas
27      cout << endl;
28
29      return 0;
30  }
```

8.6.0.4 Z Algorithm - Encontrar todos os períodos

Objetivo

Um ponto final de uma string é um prefixo que pode ser usado para gerar a string inteira repetindo o prefixo. A última repetição pode ser parcial. Por exemplo , os períodos de abcabcasão abce abcabc. Sua tarefa é encontrar todos os comprimentos de período de uma string.

Estratégia

A resolução é baseada no algoritmo Z, que é um algoritmo de correspondência de strings que encontra todas as ocorrências de uma string dentro de outra string. O algoritmo Z constrói um array Z onde Z[i] denota o comprimento da correspondência mais longa entre o prefixo começando no índice 0 e o sufixo começando no índice i. Depois de calcular o array Z, os comprimentos de período da string podem ser encontrados iterando sobre o array Z e verificando se Z[i] + i é igual ao comprimento da string. Se for, então i é um comprimento de período da string.

Código

```
1   int z[MAXN];
2
3   void calculateZ(string str) {
4
5       int n = str.size();
6       for (int i = 1, l = 0, r = 0; i < n; i++) {
7           if(i <= r) z[i] = min(z[i-l], r-i+1);
8           while(i + z[i] < n && str[z[i]] == str[i + z[i]]) z[i]++;
9           if(i + z[i] - 1 > r) {
10              l = i;
11              r = i + z[i] - 1;
12          }
13      }
14  }
15
16  int main(){
17
18      string str; cin >> str;
19
20      calculateZ(str);
21
22      for(int i = 0; i < str.size(); i++) {
23          if(z[i] + i == str.size()) cout << i << "␣"; // Se Z[i] + i    igual ao
24          comprimento da string, ent o i    um comprimento de per odo da string
25      }
26      cout << str.size() << endl;
27
28      return 0;
29  }
```

8.6.0.5 Booth's Algorithm - Menor rotação lexicográfica

Objetivo

Uma rotação de uma string pode ser gerada movendo os caracteres um após o outro, do início ao fim. Por exemplo, as rotações de acab são acab, caba, abace baca. Sua tarefa é determinar a rotação lexicograficamente mínima de uma string.

Estratégia

A resolução é baseada no algoritmo de Booth, que é um algoritmo eficiente para encontrar a rotação lexicograficamente mínima de uma string. O algoritmo de Booth usa um array de falha para acompanhar as correspondências entre os caracteres da string e sua rotação. O array de falha é usado para encontrar a próxima posição possível para comparar os caracteres da string e sua rotação. O algoritmo de Booth itera sobre a string e sua rotação e compara os caracteres. Se os caracteres forem iguais, o algoritmo move-se para a próxima posição. Se os caracteres forem diferentes, o algoritmo usa o array de falha para encontrar a próxima posição possível para comparar os caracteres. O algoritmo continua até encontrar a rotação lexicograficamente mínima da string.

Código

```
1  string findMinRotation(string str) {
2
3      str += str; // Concatenamos a string com ela mesma para simplificar o algoritmo
4      int fail[str.size()];
5      memset(fail, -1, sizeof(fail));
6
7      int minRot = 0;
8
9      for(int i = 1; i < str.size(); i++) {
10         int j = fail[i-minRot-1];
11         while(j != -1 && str[i] != str[minRot + j + 1]) {
12             if(str[i] < str[minRot + j + 1])
13                 minRot = i - j - 1;
14             j = fail[j];
15         }
16         if(str[i] != str[minRot + j + 1]) {
17             if(str[i] < str[minRot]) minRot = i;
18             fail[i-minRot] = -1;
19         } else {
20             fail[i-minRot] = j + 1;
21         }
22     }
23
24     return str.substr(minRot, (int)str.size()/2);
25 }
26
27 int main() {
28
29     string str; cin >> str;
30
31     cout << findMinRotation(str) << endl;
32
33     return 0;
```

```
34 }
```

8.6.0.6 Manacher Algorithm - Maior Palindromo Substring

Objetivo

Dada uma string, sua tarefa é determinar a substring palindrômica mais longa da string. Por exemplo, o palíndromo mais longo de aybabtue é bab.

Estratégia

A resolução é baseada no algoritmo de Manacher, o qual utiliza do tamanho de palindromos já calculados a esquerda, para calculo dos proximos, e verifica qual será o proximo centro a ser expandido, dessa maneira construimos um array dp que armazena em cada indice o maior palindromo que pode ser formado tendo esse indice como centro. Para adequar o algoritmo a palindromos pares, inserimos caracteres invalidos entre os da string.

Código

```
1  void manacher(string str) {
2
3      // Pre-processamento para tratar palindromos pares
4      string arr;
5      for(auto c: str) {
6          arr.push_back('#');
7          arr.push_back(c);
8      }
9      arr.push_back('#');
10
11     // dp[i] : indica o maior palindromo tendo i como centro
12     vector<int> dp(arr.size());
13     int center = 0, right = 0;
14
15     for (int i = 1; i < arr.size(); i++) {
16         // Utiliza informa  o do pal ndromo espelhado para calcular o tamanho do pal ndromo
17         int mirror = 2*center - i;
18         if (i < right)
19             dp[i] = min(right - i, dp[mirror]);
20
21         // Expande o pal ndromo atual a partir do seu centro
22         while (i - dp[i] - 1 >= 0 and i + dp[i] + 1 < arr.size() and
23         arr[i - dp[i] - 1] == arr[i + dp[i] + 1])
24             dp[i]++;
25         if (i + dp[i] > right) {
26             center = i;
27             right = i + dp[i];
28         }
29     }
30     int maxLen = 0, maxCenter = 0;
31     for (int i = 0; i < arr.size(); i++) {
32         if (dp[i] > maxLen) {
33             maxLen = dp[i];
34             maxCenter = i;
35         }
```

```
36      }
37
38      // Extrai a substring palindr mica da string original
39      string palindrome = "";
40      for (int i = maxCenter - maxLen; i <= maxCenter + maxLen; i++) {
41          if (arr[i] != '#')
42              palindrome += arr[i];
43      }
44
45      cout << "Maior␣palindromo:␣" << palindrome << endl;
46  }
```

8.6.0.7 Contagem de strings com padrão definido

Objetivo

Sua tarefa é calcular o número de strings de comprimento n tendo um determinado padrão de comprimento m como sua substring. Todas as strings consistem em caracteres de A a Z.

Estratégia

A solução é baseada em programação dinâmica e KMP. Primeiro, pré-calculamos o array LPS do padrão usando o algoritmo KMP. Em seguida, usamos programação dinâmica para calcular o número de strings válidas. $dp[i][j]$ representa o número de strings de comprimento i que têm o padrão como uma substring, terminando no índice j do padrão. Para calcular $dp[i][j]$, iteramos sobre todos os caracteres possíveis e calculamos o próximo índice possível no padrão usando o array LPS. Adicionamos o número de strings que podem ser formadas com o próximo índice e o caractere atual a $dp[i][j]$.

Código

```
1   int lps[mxN];
2   ll dp[mxN][mxM];
3   bool done[mxN][mxM];
4
5   void initLPS(string str) {
6
7       int n = str.size();
8       for (int i = 1, j = 0; i < n; i++) {
9           while(j > 0 && str[i] != str[j]) j = lps[j-1];
10          if(str[i] == str[j]) j++;
11          lps[i] = j;
12      }
13  }
14
15  ll solve(int i, int n, int j, string &s) {
16      if (done[i][j]) {
17          return dp[i][j];
18      }
19
20      done[i][j] = true;
21
22      // Caso base: Chegamos ao final da string que est  sendo constru da (i == n)
23      if (i == n) {
```

```
24          // Se tamb m casamos com o padr o inteiro (j == s.size()), conte como uma string v li
25          return dp[i][j] = (j == (int)s.size() ? 1 : 0);
26      }
27
28      // Caso base: Casamos com o padr o inteiro (j == s.size())
29      if (j == (int)s.size()) {
30          // Podemos adicionar qualquer caractere do alfabeto (mxK)  s  posi  es restantes
31          return dp[i][j] = (mxK * solve(i + 1, n, j, s)) % mod;
32      }
33
34      ll ans = 0;
35      for (int k = 0; k < mxK; k++) {
36          // Calcula o novo progresso (t) ap s adicionar o caractere k (A-Z)
37          int t = j;
38          while (t > 0 && k != s[t] - 'A') {
39              t = lps[t - 1];
40          }
41          if (k == s[t] - 'A') {
42              t++;
43          }
44
45          ans = (ans + solve(i + 1, n, t, s)) % mod;
46      }
47
48      return dp[i][j] = ans;
49  }
50
51  int main()
52  {
53      int n; cin >> n;
54      string s; cin >> s;
55      initLPS(s);
56      cout << solve(0, n, 0, s) << endl;
57      return 0;
58  }
```

8.6.0.8 Segtree + Hash - Palindromos Queries

Objetivo

Você recebe uma string que consiste em n caracteres entre a–z. As posições da string são indexadas $1, 2, \ldots, n$. Sua tarefa é processar operações dos seguintes tipos: - Mude o caracter na posição k para x - Verifique se a substring no intervalo a e b é um palíndromo

Estratégia

A solução usa uma estrutura de dados de árvore de segmentos e hash de strings para realizar as operações de forma eficiente. Duas tabelas de hash são mantidas: uma para hashes de prefixo e outra para hashes de sufixo. Essas tabelas de hash são usadas para calcular o hash de qualquer substring da string em tempo $O(1)$. A árvore de segmentos é usada para atualizar os valores na tabela hash em tempo $O(logn)$ quando um caractere é alterado.

Para verificar se uma substring é um palíndromo, calculamos o hash de seu prefixo e sufixo e os comparamos. Se os hashes forem iguais, a substring é um palíndromo.

Código

```
const ll hashv = 257;
const ll mxN = 2e5 + 7;
const ll mod = 1e9 + 7;

ll powH[mxN] = {1};
ll fwdH[2*mxN];
ll bckH[2*mxN];

void updateHash(ll tableH[], int i, ll v, int n) {
    for (tableH[i += n] = v; i > 1; i >>= 1)
        tableH[i >> 1] = (tableH[i] + tableH[i ^ 1]) % mod;
}

ll queryHash(ll tableH[], int l, int r, int n) {
    ll res = 0;
    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
        if (l & 1) res = (res + tableH[l++]) % mod;
        if (r & 1) res = (res + tableH[--r]) % mod;
    }
    return res;
}

int main() {
    int n, m; cin >> n >> m;
    string str; cin >> str;

    // 1. Inicializa as pot ncias do valor base
    for (int i = 1; i < n; ++i) {
        powH[i] = (powH[i - 1] * hashv) % mod;
    }

    // 2. Inicializa as tabelas hash de prefixo e sufixo
    for (int i = 0; i < n; ++i) {
        updateHash(fwdH, i, (powH[i] * (ll)str[i]) % mod, n);
        updateHash(bckH, i, (powH[n - i - 1] * (ll)str[i]) % mod, n);
    }

    // 3. Processa as opera  es
    while (m--) {
        int op, a, b;
        char x;
        cin >> op;

        if (op == 1) {
            cin >> a >> x;
            a--;
            updateHash(fwdH, a, (powH[a] * (ll)x) % mod, n);
            updateHash(bckH, a, (powH[n - a - 1] * (ll)x) % mod, n);
        } else {
            cin >> a >> b;
            a--; b--;

            ll fwd = queryHash(fwdH, a, b, n);
            fwd = (fwd * powH[n - 1 - b]) % mod;
```

```
55              ll bck = queryHash(bckH, a, b, n);
56              bck = (bck * powH[a]) % mod;
57
58              cout << (fwd == bck ? "YES" : "NO");
59              cout << endl;
60          }
61      }
62
63      return 0;
64  }
```

8.6.0.9 Aho Corasick - Posição de padrões

Objetivo

Dada uma string e padrões, encontre para cada padrão a primeira posição (indexada em 1) onde ele aparece na string.

Estratégia

A solução usa o algoritmo Aho-Corasick para encontrar a primeira ocorrência de cada padrão na string. O algoritmo Aho-Corasick é um algoritmo de pesquisa de strings que constrói um autômato finito determinístico para pesquisar um conjunto de padrões em um texto. O autômato é construído de forma que cada caminho da raiz para um nó corresponda a um padrão no conjunto de padrões. O algoritmo então usa o autômato para pesquisar o texto e encontra todas as ocorrências dos padrões no texto.

O autômato usado é uma trie que armazena todos os padrões possíveis, juntamente com as informações necessárias para navegar no autômato e encontrar o próximo nó válido.

Observação esse codigo foi otimizado pois queremos apenas a primeira aparição de cada padrão. Função 'process' alterada.

Código

```
1
2  ------------------- Mesmo codigo do aho - corasick ---------------------------
3  void process(string s) {
4      int i, u = 0;
5      for(i = 0; s[i]; i++) {
6          int c = to_i(s[i]);
7          u = go(u, c);
8          for (int v = u; v; v = exi(v)) {
9              for(int w: aca[v].idx) pos[w] = i+1;
10             aca[v].term = 0;
11             aca[v].idx.clear();
12         }
13     }
14
15 }
16
17 int main() {
18     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
19
20     aca.clear(); aca.emplace_back(); //Limpa e Cria o no raiz
21
```

```
22      string text; cin >> text;
23      int q; cin >> q;
24      for (int i = 0; i < q; i++) {
25          string word; cin >> word;
26          sz[i] = word.size();
27          insert(word, i);
28      }
29      // Computa os estados do automato de maneira preguicosa
30      // Computa ao processar o texto
31      process(text);
32      for (int i = 0; i < q; i++)
33          cout << (pos[i] != 0 ? (pos[i] - sz[i] + 1) : -1 )<< endl;
34
35      return 0;
36  }
```

8.6.0.10 Número de Substrings Distintas

Objetivo

Conte o número de substrings distintas que aparecem em uma string.

Estratégia

A solução utiliza o conceito de suffix array e lcp array, um suffix array ordena todos os sufixos da string em ordem lexicográfica. O lcp array armazena o comprimento do maior prefixo comum entre dois sufixos consecutivos no suffix array. A intuição é que o número de substrings distintas é igual ao número total de substrings menos o número de substrings repetidas. O número de substrings repetidas pode ser calculado como a soma de todos os valores no lcp array.

Código

```
1
2   ------------------- Mesmo codigo do suffix array and lcp ----------------------------
3
4   int main() {
5
6       string str; cin >> str;
7       str = str + '$';
8       n = str.size();
9       build_sa(str);
10      build_lcp(str);
11
12      ll ans = 0;
13
14      for (int i = 0; i < n; i++) {
15          ans += lcp[i];
16      }
17
18      n--;
19      cout << n*(n+1)/2 - ans << endl; // Calcula o n mero de substrings distintas
20
21      return 0;
22  }
```

8.6.0.11 Maior Substring Repetida

Objetivo

Uma substring repetida é uma substring que ocorre em dois (ou mais) locais na string. Sua tarefa é encontrar a substring de repetição mais longa em uma determinada string.

Estratégia

A solução utiliza o conceito de suffix array e lcp array, um suffix array ordena todos os sufixos da string em ordem lexicográfica. O lcp array armazena o comprimento do maior prefixo comum entre dois sufixos consecutivos no suffix array. A intuição é que o maior valor no lcp array corresponde ao comprimento da substring de repetição mais longa, e o índice do suffix array correspondente ao maior valor no lcp array indica o início da substring de repetição mais longa.

Código

```
1
2  ---------------- Mesmo codigo do suffix array and lcp ----------------
3
4  int main() {
5      string str; cin >> str;
6      str = str + '$';
7      n = str.size();
8      build_sa(str); // Constr i o suffix array
9      build_lcp(str); // Constr i o lcp array
10
11     ll mxi = 0, idx;
12     for (int i = 0; i < n; i++) {
13         if(lcp[i] > mxi) {
14             mxi = lcp[i];
15             idx = sa[i];
16         }
17     }
18
19     if(mxi == 0) {
20         cout << -1;
21     } else {
22         string ans(str.begin()+idx, str.begin()+idx+mxi);
23         cout << ans;
24     }
25     cout << endl;
26     return 0;
27 }
```

8.6.0.12 K-ésima Menor Substring

Objetivo

Você recebe uma string de comprimento n. Se todas as suas substrings distintas são ordenadas lexicograficamente, qual é a k-ésima menor delas?

Estratégia

A solução utiliza o conceito de suffix array e lcp array, um suffix array ordena todos os sufixos da string em ordem lexicográfica. O lcp array armazena o comprimento do maior prefixo comum entre dois sufixos consecutivos no suffix array. A intuição é que, para cada sufixo no suffix array, podemos calcular o número de substrings distintas que começam com esse sufixo. Se esse número for menor que k, subtraímos esse número de k e passamos para o próximo sufixo. Caso contrário, encontramos a k-ésima menor substring começando com esse sufixo.

Código

```
1
2  --------------- Mesmo codigo do suffix array and lcp ---------------
3
4  int main() {
5
6      ll k;
7      string str;
8      cin >> str >> k;
9      str = str + '$';
10     n = str.size();
11     build_sa(str);
12     build_lcp(str);
13     n--;
14     for (ll i = 1; i <= n; i++) {
15         ll qtd = n - (sa[i]+lcp[i]); // Calcula o n mero de substrings distintas que come am
16         if(qtd < k) {
17             k -= qtd;
18         } else {
19             for (int j = sa[i]; j < sa[i] + lcp[i] + k; j++)
20                 cout << str[j];
21             cout << endl;
22             break;
23         }
24     }
25
26     return 0;
27 }
```

8.6.0.13 Número de Substrings Distintas de Cada Comprimento

Objetivo

Você recebe uma string de comprimento n. Para cada inteiro entre $1...n$ você precisa imprimir o número de substrings distintas desse comprimento.

Estratégia

A solução utiliza o conceito de suffix array e lcp array, um suffix array ordena todos os sufixos da string em ordem lexicográfica. O lcp array armazena o comprimento do maior prefixo comum entre dois sufixos consecutivos no suffix array.

A ideia é utilizar o LCP array para identificar os limites das substrings distintas. Para cada índice no suffix array, o LCP correspondente indica o comprimento da substring que é compartilhada com o sufixo anterior. Se o LCP for menor que o comprimento da substring que estamos considerando, então sabemos que a substring é distinta.

Podemos usar um array de diferença para contar as substrings distintas de cada comprimento. Para cada índice no suffix array, incrementamos o valor no índice correspondente ao comprimento da substring. Em seguida, decrementamos o valor no índice correspondente ao LCP. Em seguida, calculamos a soma cumulativa do array de diferença para obter o número de substrings distintas de cada comprimento.

Código

```
1
2  -------------------- Mesmo codigo do suffix array and lcp ---------------------
3
4  int main() {
5
6      string str; cin >> str;
7      str = str + '$';
8      n = str.size();
9      build_sa(str);
10     build_lcp(str);
11
12
13     for (int i = 0; i < n; i++) {
14         int l = lcp[i], r = sa[i];
15         ans[l+1]++;
16         ans[r+1]--;
17     }
18     for(int i = 1; i< n; i++){
19         ans[i] += ans[i - 1];
20     }
21     for (int i = 1; i < n; i++) {
22         cout << ans[i] << "␣";
23     }
24     cout << endl;
25
26     return 0;
27 }
```

8.6.0.14 Maior Sequência Comum

Objetivo

Escreva um programa que aceita como input duas strings que representam sequências de DNA, e printa a maior (s) sequência em comum na ordem lexicográfica entre eles.

Estratégia

A solução utiliza o conceito de suffix array e lcp array, um suffix array ordena todos os sufixos da string em ordem lexicográfica. O lcp array armazena o comprimento do maior prefixo comum entre dois sufixos consecutivos no suffix array. Para identificar a qual string original o sufixo pertence, um array auxiliar owner é utilizado. Com essas estruturas de dados, podemos encontrar eficientemente a maior sequência comum entre as duas strings. A intuição é encontrar o maior valor no lcp array, onde os sufixos comparados pertencem a strings originais diferentes.

Código

```
1
```

```
2  ------------------------------ Mesmo codigo do suffix array and lcp -----------------
3
4  int owner[mxN];
5
6  int main()
7  {
8      string s1, s2;
9      cin >> s1 >> s2;
10
11     string str;
12     str = s1 + '$' + s2 + '#';
13     n = str.size();
14
15     for (int i = 0; i < n; i++) owner[i] = i > s1.size();
16
17     build_sa(str);
18     build_lcp(str);
19
20     ll mxi = 0;
21     for (int i = 1; i < n; i++)
22         if(owner[sa[i]] != owner[sa[i-1]])
23             mxi = max(mxi, lcp[i]); // Encontra o maior valor no lcp array, onde os sufixos com
24
25     if(mxi == 0) {
26         cout << "No common sequence." << endl;
27     } else {
28         string prev = "";
29         for (int i = 1; i < n; i++) {
30             if(owner[sa[i]] != owner[sa[i-1]] && lcp[i] == mxi) {
31                 string ans(str.begin()+sa[i], str.begin()+sa[i]+mxi);
32                 if(ans != prev) { // Verifica se a substring j  foi printada, pois podem haver
33                     cout << ans << endl;
34                     prev = ans;
35                 }
36             }
37         }
38     }
39
40     return 0;
41 }
```

8.7 Geometria Computacional

8.7.0.1 Interseção de Segmentos de Reta

Objetivo

Existem dois segmentos de reta: o primeiro passa pelos pontos (x_1, y_1) e (x_2, y_2), e o segundo passa pelos pontos (x_3, y_3) e (x_4, y_4). Sua tarefa é determinar se os segmentos de reta se cruzam, ou seja, se eles têm pelo menos um ponto comum.

Estratégia

A solução é baseada no conceito de produto vetorial. Calculamos o produto vetorial entre a primeira reta e o ponto inicial da segunda reta, e o produto vetorial entre a primeira reta e o ponto final da segunda reta. Se os sinais desses dois produtos forem opostos, significa que a segunda reta cruza a primeira. Repetimos o processo invertendo os papéis das retas. Levamos em conta o caso de retas colineares, nesse caso verificamos se o ponto se encontra na reta.

Código

```
1  struct point {
2      ll x, y;
3      point() {}
4      point(ll _x, ll _y) : x(_x), y(_y){}
5      bool operator ==(point other) const {
6          return x == other.x && y == other.y;
7      }
8  };
9
10 bool operator<(point a, point b) { // Para o set<point>
11     return a.y < b.y || (a.y == b.y && a.x < b.x);
12 }
13
14 bool insegment(point p1, point p2, point p3){
15     return (min(p1.x, p2.x) <= p3.x && p3.x <= max(p1.x, p2.x)) &&
16            (min(p1.y, p2.y) <= p3.y && p3.y <= max(p1.y, p2.y));
17 }
18
19 struct vec {
20     ll x, y;
21     vec() {}
22     vec(ll _x, ll _y) : x(_x), y(_y) {}
23     vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
24 };
25
26 ll dot(vec u, vec v) {
27     return u.x*v.x + u.y*v.y;
28 }
29
30 ll cross(vec u, vec v) {
31     return u.x*v.y - u.y*v.x;
32 }
33
34 ll ccw(point p1, point p2, point p3) {
35
36     ll res = cross(vec(p1, p2), vec(p1, p3));
37
38     if(res > 0) return 1;
39     else if(res < 0) return 2;
40     else return 0;
41
42 }
43
44 int main() {
45
46     int n; cin >> n;
47     while(n--) {
```

```
48
49          point p[4];
50          for (int i = 0; i < 4; i++)
51              cin >> p[i].x >> p[i].y;
52
53          ll or1 = ccw(p[0], p[1], p[2]);
54          ll or2 = ccw(p[0], p[1], p[3]);
55          ll or3 = ccw(p[2], p[3], p[0]);
56          ll or4 = ccw(p[2], p[3], p[1]);
57
58          string ans = "NO";
59          if(or1 != or2 && or3 != or4) ans = "YES";
60          else if(or1 == 0 && insegment(p[0], p[1], p[2])) ans = "YES";
61          else if(or2 == 0 && insegment(p[0], p[1], p[3])) ans = "YES";
62          else if(or3 == 0 && insegment(p[2], p[3], p[0])) ans = "YES";
63          else if(or4 == 0 && insegment(p[2], p[3], p[1])) ans = "YES";
64
65          cout << ans << endl;
66      }
67
68
69      return 0;
70  }
```

8.7.0.2 Ponto à esquerda ou à direita da linha

Objetivo

Existe uma linha que passa pelos pontos $p1 = (x1, y1)$ e $p2 = (x2, y2)$. Existe também um ponto $p3 = (x3, y3)$. Sua tarefa é determinar se $p3$ está localizado à esquerda ou à direita da linha ou se ele toca a linha quando estamos olhando de $p1$ para $p2$.

Estratégia

A solução utiliza o produto vetorial para determinar a posição do ponto em relação à linha. O produto vetorial é calculado entre os vetores $p1p2$ e $p1p3$. Se o produto vetorial for positivo, o ponto $p3$ está à esquerda da linha. Se for negativo, o ponto $p3$ está à direita da linha. Se o produto vetorial for zero, o ponto $p3$ está na linha.

Código

```
1  struct point {
2      ll x, y;
3      point() {}
4      point(ll _x, ll _y) : x(_x), y(_y){}
5  };
6
7  struct vec {
8      ll x, y;
9      vec() {}
10     vec(ll _x, ll _y) : x(_x), y(_y) {}
11     vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
12 };
13
```

```
14 ll cross(vec u, vec v) {
15     return u.x*v.y - u.y*v.x;
16 }
17
18 int main() {
19
20     int n; cin >> n;
21     while(n--) {
22         point p[3];
23         for (int i = 0; i < 3; i++)
24             cin >> p[i].x >> p[i].y;
25
26         vec u(p[0], p[1]), v(p[0], p[2]);
27         if(cross(u,v) > 0) cout << "LEFT";
28         else if(cross(u,v) < 0) cout << "RIGHT";
29         else cout << "TOUCH";
30         cout << endl;
31     }
32
33     return 0;
34 }
```

8.7.0.3 Área de um Polígono

Objetivo

Sua tarefa é calcular a área de um polígono dado. O polígono consiste em n vértices $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Os vértices (x_i, y_i) e (x_{i+1}, y_{i+1}) são adjacentes para $i = 1, 2, ..., n-1$, e os vértices (x_1, y_1) e (x_n, y_n) também são adjacentes.

Estratégia

A solução utiliza a fórmula da área de um polígono dada pelos seus vértices. A fórmula é baseada na soma dos produtos cruzados de cada par de vértices adjacentes, dividida por 2. O produto cruzado de dois vetores (x_1, y_1) e (x_2, y_2) é dado por $x_1y_2 - x_2y_1$.

Código

```
1  struct point {
2      ll x, y;
3      point() {}
4      point(ll _x, ll _y) : x(_x), y(_y){}
5  };
6
7  ll area(vector<point> p){
8
9      ll area = 0;
10     for(int i = 0; i < p.size()-1; i++) {
11         area += (p[i].x*p[i+1].y - p[i+1].x*p[i].y);
12     }
13     //Nesse problema ele pedia 2*area, correto dividr por 2
14     return abs(area);
15 }
```

```
16
17 int main() {
18
19     int n; cin >> n;
20
21     vector<point> p(n);
22     for (int i = 0; i < n; i++)
23         cin >> p[i].x >> p[i].y;
24     p.emplace_back(p[0].x, p[0].y);
25
26     cout << area(p) << endl;
27
28     return 0;
29 }
```

8.7.0.4 Ponto dentro, fora ou na borda do polígono

Objetivo

Você recebe um polígono com n vértices e uma lista de m pontos. Sua tarefa é determinar para cada ponto se ele está dentro, fora ou na borda do polígono. O polígono consiste de n vértices $(x1, y1), (x2, y2), ..., (xn, yn)$. Os vértices (xi, yi) e $(xi + 1, yi + 1)$ são adjacentes para $i = 1, 2, ..., n - 1$, e os vértices $(x1, y1)$ e (xn, yn) também são adjacentes.

Estratégia

A solução utiliza da soma dos angulos do ponto em relação aos lados do poligono para determinar se ele está dentro, fora, ou na borda. Caso esteja dentro o ponto forma um angulo de 360, caso esteja fora forma um angulo de 0, e caso seja colinear esta na borda.

Código

```
1  const double pi = 2 * acos(0.);
2  const double eps = 1e-9;
3
4  struct point {
5      ll x, y;
6      point() {}
7      point(ll _x, ll _y) : x(_x), y(_y){}
8      bool operator ==(point other) const {
9          return x == other.x && y == other.y;
10     }
11 };
12
13 bool operator<(point a, point b) { // Para o set<point>
14     return a.y < b.y || (a.y == b.y && a.x < b.x);
15 }
16
17 struct vec {
18     ll x, y;
19     vec() {}
20     vec(ll _x, ll _y) : x(_x), y(_y) {}
21     vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
```

```
22 };
23
24 double dot(vec u, vec v) {
25     return u.x*v.x + u.y*v.y;
26 }
27
28 ll cross(vec u, vec v) {
29     return u.x*v.y - u.y*v.x;
30 }
31
32 ll ccw(point p1, point p2, point p3) {
33
34     ll res = cross(vec(p1, p2), vec(p1, p3));
35
36     if(res > 0) return 1;
37     else if(res < 0) return 2;
38     else return 0;
39
40 }
41
42 double angle(point p1, point p2, point p3) {
43     vec u(p2, p1), v(p2, p3);
44     return acos( dot(u,v) / sqrt(dot(u, u) * dot(v, v)));
45 }
46
47 bool insegment(point p1, point p2, point p3){
48     return (min(p1.x, p2.x) <= p3.x && p3.x <= max(p1.x, p2.x)) &&
49            (min(p1.y, p2.y) <= p3.y && p3.y <= max(p1.y, p2.y));
50 }
51
52 int main() {
53
54     int n, m; cin >> n >> m;
55
56     vector<point> p(n+1);
57     for (int i = 0; i < n; i++)
58         cin >> p[i].x >> p[i].y;
59     p[n] = p[0];
60
61     ll or1 = ccw(p[0], p[1], p[2]);
62
63     point q;
64     while(m--) {
65         cin >> q.x >> q.y;
66
67         double theta = 0;
68         bool touch = 0;
69         for (int i = 0; i < n; i++) {
70             ll dir = ccw(p[i], p[i+1], q);
71             if(dir == 0 && insegment(p[i], p[i+1], q)) touch = 1;
72             theta += ( dir == or1 ? 1. : -1.) * angle(p[i], q, p[i+1]);
73         }
74
75         if(touch) cout << "BOUNDARY";
76         else if(theta < eps) cout << "OUTSIDE";
```

```
77          else cout << "INSIDE";
78          cout << endl;
79
80      }
81      return 0;
82  }
```

8.7.0.5 Contagem de Pontos Inteiros em um Polígono

Objetivo

Dado um polígono, sua tarefa é calcular o número de pontos de grade dentro do polígono e em sua fronteira. Um ponto de grade é um ponto cujas coordenadas são inteiras. O polígono consiste em n vértices $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Os vértices (x_i, y_i) e (x_{i+1}, y_{i+1}) são adjacentes para $i = 1, 2, ..., n-1$, e os vértices (x_1, y_1) e (x_n, y_n) também são adjacentes.

Estratégia

A solução usa o algoritmo de Pick para calcular a área de um polígono dado seus vértices. O algoritmo de Pick afirma que a área de um polígono com pontos de grade como vértices é dada por:

$A = i + \frac{b}{2} - 1$

onde i é o número de pontos de grade dentro do polígono e b é o número de pontos de grade na fronteira do polígono.

Para calcular a área do polígono, a solução usa a fórmula de Gauss:

$A = \frac{1}{2}|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)|$

onde $(x_{n+1}, y_{n+1}) = (x_1, y_1)$.

Código

```
1  const double pi = 2 * acos(0.);
2  const double eps = 1e-9;
3
4  struct point {
5      ll x, y;
6      point() {}
7      point(ll _x, ll _y) : x(_x), y(_y){}
8      bool operator ==(point other) const {
9          return x == other.x && y == other.y;
10     }
11 };
12
13 bool operator<(point a, point b) { // Para o set<point>
14     return a.y < b.y || (a.y == b.y && a.x < b.x);
15 }
16
17 struct vec {
18     ll x, y;
19     vec() {}
20     vec(ll _x, ll _y) : x(_x), y(_y) {}
21     vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
22 };
```

```
23
24 ll area(vector<point> p){
25
26     ll area = 0;
27     for(int i = 0; i < (int)p.size()-1; i++) {
28         area += (p[i].x*p[i+1].y - p[i+1].x*p[i].y);
29     }
30
31     //Correto dividir por 2
32     return abs(area);
33 }
34
35 int main() {
36
37     int n; cin >> n;
38
39     vector<point> p(n+1);
40     for (int i = 0; i < n; i++)
41         cin >> p[i].x >> p[i].y;
42     p[n] = p[0];
43
44     ll a = area(p);
45
46     // Count the number of boundary points, integer space
47     ll boundary = 0;
48     for (int i = 0; i < n; i++) {
49         if(p[i].x == p[i+1].x) boundary += abs(p[i].y - p[i+1].y);
50         else if(p[i].y == p[i+1].y) boundary += abs(p[i].x - p[i+1].x);
51         else boundary += __gcd(abs(p[i].x-p[i+1].x),abs(p[i].y-p[i+1].y));
52     }
53
54     ll inside = (a + 2 - boundary) / 2;
55
56     cout << inside << "␣" << boundary << endl;
57
58     return 0;
```

8.7.0.6 SweepLine - Distancia Minima entre Pontos

Objetivo

Dado um conjunto de pontos no plano bidimensional, sua tarefa é encontrar a distância euclidiana mínima entre dois pontos distintos. A distância euclidiana dos pontos (x_1, y_1) e (x_2, y_2) é $(x_1 - x_2)^2 + (y_1 - y_2)^2$.

Estratégia

A solução utiliza o algoritmo de varredura de linha, que é um algoritmo eficiente para encontrar a distância mínima entre dois pontos em um conjunto de pontos. O algoritmo funciona ordenando os pontos por suas coordenadas x e, em seguida, iterando sobre os pontos ordenados. Para cada ponto, o algoritmo mantém um conjunto de pontos que estão dentro de uma certa distância horizontal do ponto atual. Para cada ponto no conjunto, o algoritmo calcula a distância entre o ponto atual e o ponto no conjunto e atualiza a distância mínima se necessário.

Código

```
ll dist(point p1, point p2) {
    return (p1.f - p2.f)*(p1.f - p2.f) + (p1.s - p2.s)*(p1.s - p2.s);
}

int main() {

    int np; cin >> np;

    vector<point> p(np);
    for (int i = 0; i < np; i++)
        cin >> p[i].f >> p[i].s;
    sort(p.begin(), p.end());

    ll mind = LLONG_MAX;
    set<point> status;
    status.insert({p[0].s, p[0].f});

    int j = 0;
    for (int i = 1; i < np; i++) {

        // Distance threshold
        ll dd = ceil(sqrt(mind));

        //Erase all points that are too far
        while(j < i && p[j].f < p[i].f - dd) {
            status.erase({p[j].s, p[j].f});
            j++;
        }

        auto l = status.lower_bound({p[i].s - dd, 0});
        auto r = status.upper_bound({p[i].s + dd, 0});

        for(auto it = l; it != r; it++)
            mind = min(mind, dist(p[i], {(*it).s,(*it).f}));

        status.insert({p[i].s, p[i].f});

    }

    cout << mind << endl;

    return 0;
rn 0;
}
```

8.7.0.7 Prim + Distancia entre Segmentos

Objetivo

O governo deseja transformar todas as terras com relíquias em áreas de preservação ambiental. Para facilitar o trabalho dos arqueólogos, o governo também construirá trilhas, cada uma conectando duas relíquias. O objetivo é construir o mínimo possível de trilhas para minimizar o impacto ambiental, mas garantindo que haja um caminho entre quaisquer duas relíquias. Reliquias são representadas como segmentos de reta.

Estratégia

A solução utiliza a técnica de Árvore Geradora Mínima (AGM) para encontrar o conjunto mínimo de trilhas que conecta todas as relíquias, sendo a menor distancia entre segmentos de reta.

O algoritmo de Prim funciona escolhendo um vértice inicial e, em seguida, adicionando o vértice mais próximo não conectado à AGM. Este processo é repetido até que todos os vértices estejam na AGM.

Código

```
struct point {
    double x, y;
    point() {}
    point(double _x, double _y) : x(_x), y(_y){}
};

struct seg {
    point a, b;
    seg() {}
    seg(point _a, point _b) : a(_a), b(_b) {}
};

struct vec {
    double x, y;
    vec() {}
    vec(double _x, double _y) : x(_x), y(_y) {}
    vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
};

const int mxn = 1010;
const double eps = 1e-9;

int n;
seg relic[mxn];
double dist[mxn];
bool vis[mxn];

double dot(vec u, vec v) {
    return u.x*v.x + u.y*v.y;
}

double cross(vec u, vec v) {
    return u.x*v.y - u.y*v.x;
}

double ccw(point p1, point p2, point p3) {
    return cross(vec(p1, p2), vec(p1, p3));
}

bool intersec(seg s, seg r) { // Verify if two segment intersects
    return ccw(s.a, s.b, r.a) * ccw(s.a, s.b, r.b) < -eps &&
           ccw(r.a, r.b, s.a) * ccw(r.a, r.b, s.b) < -eps;
}

vec scale(double l, vec u) { //Scale vector u by scalar l
    return vec(l*u.x, l*u.y);
```

```
47 }
48
49 point trans(point a, vec u) { // Translate point a by vector u
50     return point(a.x + u.x, a.y + u.y);
51 }
52
53 double dpp(point a, point b) { //Distance point and point
54     vec ab(a, b);
55     return sqrt(dot(ab, ab));
56 }
57
58 double dpseg(point p, seg s) { //Distance point and segment
59     vec ab(s.a, s.b), ap(s.a, p);
60     double norm = dot(ab, ab);
61     if(norm < eps) return dpp(p, s.a);
62
63     double l = dot(ab, ap) / norm; //Calculate lambda value of projection
64     if(l < -eps) return dpp(p, s.a);
65     if(l > 1.) return dpp(p, s.b);
66
67     point c = trans(s.a, scale(l, ab));
68     return dpp(c, p);
69 }
70
71 double dseg(seg s, seg r) { //Distance segment and segment
72
73     if(intersec(s, r)) return 0.;
74
75     return min(dpseg(s.a, r),
76                min(dpseg(s.b, r),
77                    min(dpseg(r.a, s), dpseg(r.b, s))));
78 }
79
80 double prim() {
81
82     double cost = 0;
83     memset(vis, 0, sizeof(vis));
84     for (int i = 0; i <= n; i++) dist[i] = 1e8 + 7;
85     dist[0] = 0;
86     while(1) {
87         int u = n;
88         for (int v = 0; v < n; v++)
89             if(!vis[v] && dist[v] < dist[u]) u = v;
90         if(u == n) break;
91         cost += dist[u];
92         vis[u] = 1;
93
94         for (int v = 0; v < n; v++)
95             if(!vis[v]) dist[v] = min(dist[v], dseg(relic[u], relic[v]));
96     }
97     return cost;
98 }
99
100 int main() {
101
```

```
102       cin >> n;
103       for (int i = 0; i < n; i++) {
104           point a, b;
105           cin >> a.x >> a.y >> b.x >> b.y;
106           relic[i] = seg(a, b);
107       }
108
109       cout << ceil(prim()) << endl;
110
111       return 0;
112   }
```

8.7.0.8 SweepLine + DP - Baloes e telhados

Objetivo

Uma das principais dificuldades de organizar uma Maratona de Programação é recolher os balões que escapam e ficam presos no teto do salão: muitas vezes o contrato com o dono do salão exige que este seja entregue limpo logo após o evento, sob pena de multa. Este ano a organização da Maratona está mais previdente: ela tem o desenho do teto do salão, e quer sua ajuda para determinar o que pode acontecer com um balão, dependendo da posição no solo onde ele é solto (isto é, se é bloqueado pelo teto ou se escapa para o exterior do salão).

O teto do salão é formado por vários planos que, vistos de lado, podem ser descritos por segmentos de reta, como mostrado na figura abaixo:

O balão pode ser considerado pontual. Quando um balão toca um segmento do teto que é horizontal, ele fica preso. Quando um balão toca um segmento que é inclinado, o balão desliza até o ponto mais alto do segmento e escapa, podendo escapar completamente do salão ou podendo tocar em mais segmentos. Não há pontos em comum entre os segmentos que formam o teto.

Por exemplo, se o balão for solto nas posições marcadas como a ou b, será bloqueado na posição de coordenadas (2, 5); se o balão for solto na posição marcada como c, será bloqueado na posição de coordenadas (6, 5); e se o balão for solto na posição marcada como d, não será bloqueado e escapará para fora do salão na posição de coordenada x = 7.

Escreva um programa que, dada a descrição do teto do salão como segmentos de reta, responde a uma série de consultas sobre a posição final de balões soltos do piso do salão.

Estratégia

A solução utiliza um algoritmo de varredura de linha para determinar a próxima interação do balão com o teto, considerando a ordem das coordenadas x dos eventos. Um array $snxt$ é utilizado para armazenar o próximo segmento de teto acima do segmento i, e $bnxt$ para armazenar o próximo segmento do balao que encontraremos na varredura. A programação dinâmica é utilizada para encontrar o destino final de cada balão, buscando o último segmento que o balão irá tocar.

Código

```
1  const int mxn = 1e6 + 7;
2
3  struct point {
4      int x, y;
5      point() {}
6      point(int _x, int _y) : x(_x), y(_y){}
7  };
```

```
8
9  struct vec {
10     ll x, y;
11     vec() {}
12     vec(int _x, int _y) : x(_x), y(_y) {}
13     vec(point a, point b) : x(b.x - a.x), y(b.y - a.y) {}
14 };
15
16 struct seg {
17     point a, b;
18     seg() {}
19     seg(point _a, point _b) : a(_a), b(_b) {}
20 };
21
22 struct event {
23     int x, type, i; // type: -1/1 if begin/end of segment or 0 if balloon
24     event() {}
25     event(int _x, int _type, int _i): x(_x), type(_type), i(_i) {}
26 };
27
28 int nroof, nbln, nev;
29 seg roof[mxn];
30 int bln[mxn];
31 event ev[mxn];
32 int snxt[mxn], bnxt[mxn];
33 int memo[mxn];
34
35 ll cross(vec u, vec v) {
36     return u.x*v.y - u.y*v.x;
37 }
38
39 bool cmp1(event e, event f) { // For sort all events
40     // Sort by x coordinate, considering if we have a start roof and a baloon the baloon come f
41     if(e.type != 0 && f.type == 0) return !cmp1(f, e);
42     if(e.x == f.x && e.type == 0 && f.type != 0)
43         return f.type == 1;
44     return e.x < f.x;
45 }
46
47 bool cmp2(int i, int j) { // For sort set status
48     // Sort in a vertical way such that roof[i] is below roof[j]
49     seg s = roof[i], r = roof[j];
50     if(s.a.x < r.a.x) return cross(vec(s.a, s.b), vec(s.a, r.a)) > 0;
51     return cross(vec(r.a, r.b), vec(r.a, s.a)) < 0;
52 }
53
54 int dp(int i) { // for snxt[i] != -1
55     // Make a path comprehension and return the one-last element
56     if(memo[i] != -1) return memo[i];
57     if(snxt[snxt[i]] == -1) return memo[i] = i;
58     return memo[i] = dp(snxt[i]);
59 }
60
61 point dest(int x, int i) { // for snxt == -1
62     // Return the end of each baloon considiering horizontal and incline roofs
```

```
63      seg s = roof[i];
64      if(s.a.y == s.b.y) return point(x, s.a.y);
65      if(s.a.y < s.b.y) return point(s.b.x, -1);
66      return point(s.a.x, -1);
67  }
68
69  // Use to process what is the next element above element for each balloon and roof
70  void sweepline() {
71
72      set<int, bool (*)(int,int)> status(cmp2);
73      for (int j = 0; j < nev; j++) {
74          int i = ev[j].i;
75          if(ev[j].type == 0) { // If balloon just verify if we have a roof above it
76              auto it = status.begin();
77              bnxt[i] = it == status.end() ? -1 : *it;
78          } else {
79              seg s = roof[i];
80              if(ev[j].type == -1) { // If start first insert and if the roof is descedent verify
81                  status.insert(i);
82                  if(s.a.y > s.b.y) {
83                      auto it = status.find(i);
84                      ++it;
85                      snxt[i] = it == status.end() ? -1 : *it;
86                  } else if(s.a.y == s.b.y) snxt[i] = -1;
87              } else { // If end and if the roof is ascedent verify if we have a roof above it, s
88                  if(s.a.y < s.b.y) {
89                      auto it = status.find(i);
90                      ++it;
91                      snxt[i] = it == status.end() ? -1 : *it;
92                  }
93                  status.erase(i);
94              }
95          }
96      }
97  }
98
99  int main() {
100
101     while(cin >> nroof >> nbln && nroof && nbln) {
102         nev = 0;
103         for (int i = 0; i < nroof; i++) {
104             point p1, p2;
105             cin >> p1.x >> p1.y >> p2.x >> p2.y;
106             roof[i] = p1.x < p2.x ? seg(p1, p2) : seg(p2, p1);
107             seg s = roof[i];
108             ev[nev++] = event(s.a.x, -1, i);
109             ev[nev++] = event(s.b.x, 1, i);
110         }
111         for (int i = 0; i < nbln; i++) {
112             int x; cin >> x;
113             bln[i] = x;
114             ev[nev++] = event(x, 0, i);
115         }
116         sort(ev, ev+nev, cmp1);
117         sweepline();
```

```
118         memset(memo, -1, sizeof(memo));
119         for (int i = 0; i < nbln; i++) {
120             if(bnxt[i] == -1) {
121                 cout << bln[i] << endl;
122                 continue;
123             }
124             int x, is;
125             if(snxt[bnxt[i]] == -1) {
126                 x = bln[i];
127                 is = bnxt[i];
128             } else {
129                 is = dp(bnxt[i]);
130                 seg s = roof[is];
131                 x = s.a.y < s.b.y ? s.b.x : s.a.x;
132                 is = snxt[is];
133             }
134             point p = dest(x, is);
135             cout << p.x;
136             if(p.y > -1) cout << "␣" << p.y;
137             cout << endl;
138         }
139     }
140
141     return 0;
142 }
```

8.8 Sorting e Searching

8.8.0.1 Contagem de valores distintos

Objetivo

Você recebe uma lista de n inteiros, e sua tarefa é calcular o número de valores distintos na lista.

Estratégia

A solução usa um conjunto (set) para armazenar os valores distintos. Para cada elemento na lista, o elemento é inserido no conjunto. Se o elemento já existe no conjunto, a inserção não é feita. Portanto, o tamanho do conjunto no final do loop é o número de valores distintos.

Código

```
1  int main() {
2
3          int n; cin >> n;
4          set<int> s;
5          for (int i = 0; i < n; i++) {
6              int x; cin >> x;
7              s.insert(x);
8          }
```

```
 9          cout << s.size() << endl;
10          return 0;
11  }
```

8.8.0.2 Alocação de Apartamentos

Objetivo

Existem n candidatos e m apartamentos disponíveis. Sua tarefa é distribuir os apartamentos de forma que o maior número possível de candidatos receba um apartamento. Cada candidato tem um tamanho de apartamento desejado e eles aceitarão qualquer apartamento cujo tamanho seja próximo o suficiente do tamanho desejado.

Estratégia

A solução utiliza ordenação e um algoritmo guloso para alocar apartamentos aos candidatos. Primeiro, ordenamos os candidatos em ordem crescente de tamanho de apartamento desejado. Em seguida, iteramos sobre os apartamentos e atribuímos cada apartamento ao primeiro candidato na lista que o aceita.

Código

```
 1  int main() {
 2
 3          ll n,m,k; cin >> n >> m >> k;
 4          int i,j;
 5          vector<ll> need(n);
 6          for (i = 0; i < n; i++) cin >> need[i];
 7          sort(need.begin(), need.end());
 8
 9          vector<ll> ap(m);
10          for (i = 0; i < m; i++) cin >> ap[i];
11          sort(ap.begin(), ap.end());
12
13          int ans = 0;
14          for(i = 0, j = 0; i < n && j < m;) {
15                  if(ap[j] >= need[i]-k && ap[j] <= need[i]+k) {
16                          i++; j++; ans++;
17                  }
18                  else if(ap[j] < need[i]-k) {
19                          j++;
20                  }
21                  else {
22                          i++;
23                  }
24          }
25
26          cout << ans << endl;
27
28          return 0;
```

8.8.0.3 Dois ponteiros - Gondolas da Roda Gigante

Objetivo

Existem n crianças que desejam ir a uma roda gigante, e sua tarefa é encontrar uma gôndola para cada criança. Cada gôndola pode ter uma ou duas crianças, e, além disso, o peso total em uma gôndola não pode exceder x. Você sabe o peso de cada criança. Qual é o número mínimo de gôndolas necessárias para as crianças?

Estratégia

A solução utiliza um algoritmo guloso para minimizar o número de gôndolas. Primeiro, ordenamos as crianças em ordem crescente de peso. Em seguida, iteramos sobre as crianças da menor para a maior, tentando emparelhar cada criança com a criança mais pesada que ainda não foi emparelhada. Se o peso total da criança atual e a criança mais pesada não exceder x, podemos emparelhá-las na mesma gôndola. Caso contrário, a criança mais pesada terá que ir em uma gôndola sozinha.

Código

```
int main() {

        ll n, x; cin >> n >> x;

        ll children[n];
        for (int i = 0; i < n; i++) cin >> children[i];
        sort(children, children + n);

        int i, j, ans = 0;
        for (i = 0, j = n-1; i <= j;) {
                if(children[j]+children[i] <= x) {
                        i++; j--;
                }
                else  {
                        j--;
                }
                ans++;
        }
        cout << ans << endl;

        return 0;
```

8.8.0.4 Multiset - Bilhetes de Concerto

Objetivo

Existem n bilhetes de concerto disponíveis, cada um com um determinado preço. Então, m clientes chegam, um após o outro. Cada cliente anuncia o preço máximo que está disposto a pagar por um bilhete, e depois disso, receberá um bilhete com o preço mais próximo possível, de modo que não exceda o preço máximo.

Estratégia

A solução utiliza um multiset para armazenar os preços dos bilhetes disponíveis. A cada cliente que chega, realizamos uma busca binária no multiset para encontrar o preço mais próximo que não excede o preço máximo que o cliente está disposto a pagar. Se encontrarmos esse preço, removemos o bilhete do multiset e adicionamos o preço do bilhete à lista de preços dos bilhetes comprados. Caso contrário, indicamos que o cliente não conseguiu comprar um bilhete.

Código

```
int main() {

        int n, m; cin >> n >> m;

        multiset<int> tickets;
        for (int i = 0; i < n; i++) {
                int p; cin >> p;
                tickets.insert(p);
        }

        while(m--) {
                int p; cin >> p;
                auto it = tickets.upper_bound(p);
                if(it == tickets.begin()) {
                        cout << -1 << endl;
                }
                else {
                        cout << *(--it) << endl;
                        tickets.erase(it);
                }
        }

        return 0;

}
```

8.8.0.5 Número Máximo de Clientes em um Restaurante

Objetivo

Você é dado os horários de chegada e saída de n clientes em um restaurante. Qual foi o número máximo de clientes no restaurante a qualquer momento?

Estratégia

A solução envolve o uso de um vetor de frequência para rastrear o número de clientes no restaurante a qualquer momento. Para cada cliente, incrementamos o vetor de frequência no horário de chegada e decrementamos no horário de saída. Em seguida, iteramos sobre o vetor de frequência e calculamos a soma máxima, que representa o número máximo de clientes no restaurante.

Código

```
int main() {

        int n; cin >> n;
        vector<pair<int,int>> rest;
        while (n--) {
                int a,b; cin >> a >> b;
                rest.push_back({a,1});
                rest.push_back({b,-1});
        }
        sort(rest.begin(), rest.end());
```

```
11          int ans = 0;
12          int preff = 0;
13          for (auto r: rest) {
14                  preff += r.second;
15                  ans = max(ans,preff);
16          }
17          cout << ans << endl;
18  }
```

8.8.0.6 Greed Scheduling - Máximo de Filmes

Objetivo

Em um festival de cinema, n filmes serão exibidos. Você sabe o horário de início e término de cada filme. Qual é o número máximo de filmes que você pode assistir completamente?

Estratégia

A solução utiliza uma abordagem gulosa, ordenando os filmes pelo horário de término. O algoritmo itera pelos filmes ordenados, adicionando o filme atual à lista de filmes que você pode assistir se o horário de início do filme atual for maior ou igual ao horário de término do último filme na lista.

Código

```
1  int main() {
2
3          int n; cin >> n;
4          vector<pair<int,int>> movies;
5          while(n--) {
6                  int a,b; cin >> a >> b;
7                  movies.push_back({b,a});
8          }
9          sort(movies.begin(), movies.end());
10
11         int ans = 0;
12         int atual = 0;
13         for(auto m: movies) {
14                 if(m.second >= atual) {
15                         ans++;
16                         atual = m.first;
17                 }
18         }
19         cout << ans << endl;
20         return 0;
21
22 }
```

8.8.0.7 Encontrando o Par com Soma X

Objetivo

Você recebe um array de n inteiros, e sua tarefa é encontrar dois valores (em posições distintas) cuja soma seja x.

Estratégia

A solução utiliza um dicionário para armazenar os valores já encontrados no array. Para cada valor $a[i]$ do array, verificamos se $x-a[i]$ está no array. Caso esteja, encontramos a soma desejada e o algoritmo termina. Caso contrário, inserimos $a[i]$ no dicionário.

Código

```
int main()
{
    int n, x;
    cin >> n >> x;

    vector<pair<int,int>> arr;
    for (int i = 0; i < n; i++){
        int v; cin >> v;
        arr.push_back({v,i+1});
    }
    sort(arr.begin(), arr.end());

    bool ans = false;
    for (auto p: arr) {
        int v = x-p.first;
        int k = 0;
        for (int b = n / 2; b >= 1; b /= 2)
            while (k + b < n && arr[k + b].first <= v)
                k += b;
        if(p.second == arr[k].second) continue;
        if(arr[k].first == v) {
            cout << p.second << " " << arr[k].second << endl;
            ans = 1;
            break;
        }
    }
    if(!ans) cout << "IMPOSSIBLE" << endl;
    return 0;

}
```

8.8.0.8 Máxima Soma de Subarrays Contínuos

Objetivo

Dado um arranjo de n inteiros, sua tarefa é encontrar a soma máxima de valores em um subarranjo contíguo e não vazio.

Estratégia

A solução utiliza o algoritmo de Kadane para encontrar a máxima soma de subarranjos. O algoritmo funciona mantendo duas variáveis: ‘sum‘ e ‘maxi‘. A variável ‘sum‘ armazena a soma atual do subarranjo, e a variável ‘maxi‘ armazena a maior soma encontrada até o momento. Para cada elemento do arranjo, o algoritmo verifica se a soma atual é menor que zero. Se for, a soma atual é redefinida para zero. Caso contrário, o elemento atual é adicionado à soma. A maior soma é atualizada se a soma atual for maior que a maior soma encontrada até o momento.

Código

```
int main() {

        int n; cin >> n;
        ll sum = 0, maxi = -INT_MAX;
        while(n--) {
                if(sum < 0) sum = 0;
                ll x; cin >> x;
                sum += x;
                maxi = max(maxi, sum);
        }
        cout << maxi << endl;

        return 0;
}
```

8.8.0.9 Custo Mínimo para Igualar Valores

Objetivo

Existem n palitos com alguns comprimentos. Sua tarefa é modificar os palitos para que cada palito tenha o mesmo comprimento. Você pode aumentar ou diminuir o comprimento de cada palito. Ambas as operações custam x, onde x é a diferença entre o novo e o comprimento original. Qual é o custo total mínimo?

Estratégia

A solução é baseada na ideia de encontrar o comprimento médio dos palitos. A soma das diferenças entre o comprimento médio e o comprimento de cada palito é o custo total mínimo. Portanto, a estratégia é ordenar os palitos e então calcular a soma das diferenças entre o comprimento médio e o comprimento de cada palito.

Código

```
int main() {

        int n; cin >> n;

        ll sticks[n];
        for(int i = 0; i < n; i++) cin >> sticks[i];
        sort(sticks, sticks+n);

        ll mid = sticks[n/2]; // Encontra o comprimento m dio
        ll ans = 0;
        for(auto p: sticks) ans += abs(mid-p); // Calcula a soma das diferen as
        cout << ans << endl;

        return 0;
}
```

8.8.0.10 A Menor Soma Inalcançavel

Objetivo

Você tem n moedas com valores inteiros positivos. Qual é a menor soma que você não pode criar usando um subconjunto das moedas?

Estratégia

A solução é baseada na ideia de que se você puder criar todas as somas até x, você também pode criar todas as somas até $x +$ menor moeda. Então, o algoritmo itera sobre as moedas em ordem crescente e tenta criar todas as somas até a soma atual. Se o algoritmo encontrar uma soma que não pode ser criada, essa é a menor soma inatingível.

Código

```
int main() {

        int n; cin >> n;

        ll coins[n];
        for (int i = 0; i < n; i++) cin >> coins[i];
        sort(coins, coins+n);

        ll ans = 1;
        for (int i = 0; i < n; i++)
                if(coins[i] <= ans) ans+= coins[i];
        cout << ans << endl;

        return 0;

}
```

8.8.0.11 Contagem de arrays em ordem crescente

Objetivo

Você recebe um array que contém cada número entre $1 \ldots n$ exatamente uma vez. Sua tarefa é coletar os números de 1 a n em ordem crescente. Em cada rodada, você percorre o array da esquerda para a direita e coleta o máximo de números possível. Qual será o número total de rodadas?

Estratégia

A solução é baseada na observação de que o número de rodadas é igual ao maior valor no array. Isso porque, na primeira rodada, você coleta todos os números que são iguais a 1. Na segunda rodada, você coleta todos os números que são iguais a 2, e assim por diante.

Código

```
int main()
{

    int n;
    cin >> n;

    vector<int> nums(n);
    for (int i = 0; i < n; i++) {
```

```
 9          int x; cin >> x;
10          nums[x-1] = i;
11      }
12
13      int ans = 1;
14      for (int i = 1; i < n; i++) {
15          ans += (nums[i] < nums[i-1]);
16      }
17      cout << ans << endl;
18
19      return 0;
20  }
```

8.8.0.12 Contagem de arrays em ordem crescente + swap

Objetivo

Você recebe um array que contém cada número entre 1…n exatamente uma vez. Sua tarefa é coletar os números de 1 a n em ordem crescente. Em cada rodada, você percorre o array da esquerda para a direita e coleta o máximo de números possível. Dado m operações que trocam dois números no array, sua tarefa é relatar o número de rodadas após cada operação.

Estratégia

A solução utiliza a técnica de simulação para coletar os números em ordem crescente. Em cada rodada, percorremos o array da esquerda para a direita. Se o número atual for igual ao número que estamos procurando, coletamos o número atual e atualizamos o número que estamos procurando. Caso contrário, ignoramos o número atual e continuamos percorrendo o array. Para calcular o número de rodadas após cada operação, basta simular as operações no array e contar o número de rodadas necessárias para coletar todos os números.

Código

```
 1  int main()
 2  {
 3
 4      int n, m; cin >> n >> m;
 5
 6      vector<int> idx(n+2,0), nums(n+1,0);
 7      for(int i = 1; i <= n; i++) {
 8          int x; cin >> x;
 9          idx[x] = i;
10          nums[i] = x;
11      }
12
13      int ans = 1;
14      for(int i = 1; i <= n; i++) {
15          ans += (idx[i] < idx[i-1]);
16      }
17
18      while(m--) {
19          int a,b; cin >> a >> b;
20          int val1 = nums[a], val2 = nums[b];
21          swap(nums[a], nums[b]);
```

```
22
23          if(idx[val1-1] <=  idx[val1] && idx[val1-1] > b) ans++;
24          if(idx[val1-1] >  idx[val1] && idx[val1-1] <= b) ans--;
25          if(idx[val1] <=  idx[val1+1] && idx[val1+1] < b) ans++;
26          if(idx[val1] >  idx[val1+1] && idx[val1+1] >= b) ans--;
27          idx[val1] = b;
28
29          if(idx[val2-1] <=  idx[val2] && idx[val2-1] > a) ans++;
30          if(idx[val2-1] >  idx[val2] && idx[val2-1] <= a) ans--;
31          if(idx[val2] <=  idx[val2+1] && idx[val2+1] < a) ans++;
32          if(idx[val2] >  idx[val2+1] && idx[val2+1] >= a) ans--;
33          idx[val2] = a;
34
35          cout << ans << endl;
36      }
37
38
39      return 0;
40
41  }
```

8.8.0.13 A Maior Sequência de Músicas Únicas

Objetivo

Você é dado uma playlist de uma estação de rádio desde sua fundação. A playlist possui um total de n músicas. Qual é a maior sequência de músicas sucessivas onde cada música é única?

Estratégia

A solução usa um dicionário (ou mapa) para manter o rastreamento da última ocorrência de cada música na lista de reprodução. Iteramos sobre a lista de reprodução e para cada música, verificamos se ela já foi tocada antes. Se sim, atualizamos o último índice da música. Se não, atualizamos o último índice da música e calculamos o comprimento da sequência atual de músicas únicas. Mantém-se o comprimento máximo da sequência de músicas únicas ao longo da iteração.

Código

```
1  int main() {
2
3          int n; cin >> n;
4          vector<int> musics(n);
5          map<int,int> idx;
6          for (int i = 0; i < n; i++) {
7                  cin >> musics[i];
8                  idx[musics[i]] = -1;
9          }
10
11         int start = 0, ans = 0;
12         for (int j = 0; j < n; j++) {
13
14                 start = max(start, idx[musics[j]] +1);
15                 ans = max(ans,j-start+1);
```

```
16                  idx[musics[j]] = j;
17          }
18
19          cout << ans << endl;
20
21          return 0;
22  }
```

8.8.0.14 Menor Numero de Torres de Cubos Empilhados

Objetivo

Você recebe n cubos em uma determinada ordem, e sua tarefa é construir torres usando eles. Quando dois cubos estão um em cima do outro, o cubo superior deve ser menor que o cubo inferior. Você deve processar os cubos na ordem dada. Você sempre pode colocar o cubo no topo de uma torre existente ou iniciar uma nova torre. Qual é o número mínimo possível de torres?

Estratégia

A solução usa um multiset para manter o tamanho dos cubos na parte superior de cada torre. Para cada novo cubo, procuramos no multiset um cubo maior que o cubo atual. Se encontrarmos, colocamos o cubo atual no topo da torre correspondente e removemos o cubo maior do multiset. Caso contrário, iniciamos uma nova torre e adicionamos o cubo atual ao multiset.

Código

```
1  int main() {
2
3          int n; cin >> n;
4          multiset<int> towers;
5          while(n--) {
6                  int k; cin >> k;
7                  auto it = towers.upper_bound(k);
8                  if(it == towers.end()) {
9                          towers.insert(k);
10                 } else {
11                         towers.erase(it);
12                         towers.insert(k);
13                 }
14         }
15         cout << towers.size() << endl;
16
17         return 0;
18 }
```

8.8.0.15 Maior trecho sem semáforos

Objetivo

Existe uma rua de comprimento x cujas posições são numeradas $0, 1, ..., x$. Inicialmente, não há semáforos, mas n conjuntos de semáforos são adicionados à rua um após o outro. Sua tarefa é calcular o comprimento da passagem mais longa sem semáforos após cada adição.

Estratégia

A solução utiliza um conjunto para armazenar as posições dos semáforos já adicionados na rua. A cada adição de um conjunto de semáforos, o código itera sobre as posições do conjunto e insere as posições no conjunto. Em seguida, o código calcula a diferença entre a posição atual do conjunto e a posição anterior do conjunto no conjunto, e atualiza o comprimento da passagem mais longa sem semáforos.

Código

```
int main () {
        ios::sync_with_stdio(false);
        cin.tie(NULL);

        int x, n; cin >> x >> n;
        set<int> traffic;
        multiset<int> gaps;
        traffic.insert(0); traffic.insert(x);
        gaps.insert(x);
        while(n--) {
                int t; cin >> t;
                auto next = traffic.upper_bound(t);
                auto prev = next; --prev;

                traffic.insert(t);
                gaps.erase(gaps.find(*next-*prev));
                gaps.insert(t-*prev);
                gaps.insert(*next-t);

                auto ans = gaps.end(); --ans;
                cout << *(ans) << "␣";
        }
        cout << endl;
        return 0;
}
```

8.8.0.16 Eliminação Circular com k Passos

Objetivo

Considere um jogo onde há n crianças (numeradas de 1 a n) em um círculo. Durante o jogo, a cada rodada, a criança no próximo slot (uma criança a cada duas) é removida do círculo até que não haja mais crianças. Em qual ordem as crianças serão removidas?

Estratégia

A solução utiliza um conceito semelhante ao algoritmo de Josephus, porém no lugar de eliminar o segundo elemento, eliminamos o k-ésimo. A principal ideia é simular o jogo e manter um vetor de booleanos para indicar quais crianças ainda estão no jogo. A cada rodada, percorremos o vetor, ignorando as crianças que já foram removidas. Depois de encontrar a k-ésima criança, a removemos e continuamos o processo até que todas as crianças tenham sido removidas.

Código

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
```

```
3
4  using namespace __gnu_pbds;
5  using namespace std;
6
7  template<class T> using oset =
8          tree<T, null_type, less<T>, rb_tree_tag,
9          tree_order_statistics_node_update>;
10
11 int main() {
12
13         int n, k = 1; cin >> n;
14         oset<int> children;
15         for (int i = 1; i <= n; i++) children.insert(i);
16
17         int idx = 0;
18         while(children.size() > 1) {
19                 idx = (idx+k)\%children.size();
20                 cout << *(children.find_by_order(idx)) << "␣";
21                 children.erase(children.find_by_order(idx));
22         }
23
24         cout << *(children.find_by_order(0)) << endl;
25
26         return 0;
```

8.8.0.17 Verificação de Conter ou Ser Contido por Outros Intervalos

---

Objetivo

Dada n intervalos, sua tarefa é determinar para cada intervalo se ele contém algum outro intervalo e se algum outro intervalo o contém. O intervalo $[a, b]$ contém o intervalo $[c, d]$ se $a \leq c$ e $d \leq b$.

Estratégia

A estratégia consiste em usar uma árvore de segmentos para armazenar os intervalos e, em seguida, realizar duas consultas para cada intervalo: uma para verificar se existe algum outro intervalo que o contém e outra para verificar se ele contém algum outro intervalo. Para otimizar as consultas, a árvore de segmentos será construída com os seguintes parâmetros:

* **Valor mínimo:** a do intervalo. * **Valor máximo:** b do intervalo. * **Operação:** $min(x, y)$.

Essa configuração garante que para cada consulta, a árvore de segmentos retorna o intervalo com a menor a que seja maior ou igual ao c da consulta. Se esse intervalo for encontrado, então ele contém o intervalo da consulta. O mesmo raciocínio se aplica à segunda consulta, com a diferença de que a operação da árvore de segmentos será $max(x, y)$, retornando o intervalo com a maior b que seja menor ou igual ao d da consulta.

Código

```
1  int main() {
2
3          int n; cin >> n;
4          vector<pair<int,int>> ranges;
5          map<pair<int,int>,int> idx;
6          for (int i = 0; i < n; i++) {
```

```
7                   int a,b; cin >> a >> b;
8                   ranges.push_back( {a, -b});
9                   idx[{a, -b}] = i;
10          }
11          sort(ranges.begin(), ranges.end());
12
13          vector<bool> contain(n,0), contained(n,0);
14          int maxb = 0;
15          for (auto p: ranges) {
16                  if(abs(p.s) <= maxb) contained[idx[p]] = 1;
17                  maxb = max(maxb, abs(p.s));
18          }
19          int minb = INT_MAX;
20          for (int i = n-1; i >= 0; i--) {
21                  if(abs(ranges[i].s) >= minb) contain[idx[ranges[i]]] = 1;
22                  minb = min(minb, abs(ranges[i].s));
23          }
24
25          for (auto i: contain) cout << i << "␣";
26          cout << endl;
27          for (auto i: contained) cout << i << "␣";
28          cout << endl;
29
30          return 0;
31  }
```

8.8.0.18 Contagem de intervalos contendo ou contidos

Objetivo

Dada uma sequência de n intervalos, sua tarefa é contar, para cada intervalo, quantos outros intervalos ele contém e quantos outros intervalos o contêm. O intervalo $[a, b]$ contém o intervalo $[c, d]$ se ac e db.

Estratégia

A solução utiliza ordered_set para realizar as consultas de contagem de intervalos. Primeiramente, ordenamos os intervalos por seus pontos finais, e depois iteramos sobre eles. Para cada intervalo, inserimos seu ponto inicial na árvore de intervalo, com o valor 1, e seu ponto final, com o valor -1. Após a inserção de todos os pontos, realizamos uma consulta para cada intervalo, obtendo a soma dos valores da árvore de intervalo no intervalo que corresponde ao intervalo atual. Esse valor corresponde ao número de outros intervalos que o intervalo atual contém. Para contar quantos outros intervalos contém o intervalo atual, realizamos a mesma operação, mas antes de cada consulta, removemos os valores correspondentes ao intervalo atual da árvore.

Código

```
1  using namespace __gnu_pbds;
2  using namespace std;
3
4  template<class T> using oset =
5          tree<T, null_type, less<T>, rb_tree_tag,
6          tree_order_statistics_node_update>;
7
8
```

```
int main() {

        int n; cin >> n;
        vector<pair<int,int>> ranges;
        map<pair<int,int>,int> idx;
        for (int i = 0; i < n; i++) {
                int a,b; cin >> a >> b;
                ranges.push_back( {a, -b});
                idx[{a, -b}] = i;
        }
        sort(ranges.begin(), ranges.end());

        vector<int> contain(n,0), contained(n,0);
        oset<pair<int,int>> st;
        for (int i = 0; i < n; i++) {
                pair<int,int> p = {abs(ranges[i].s), n-i};
                st.insert(p);
                contained[idx[ranges[i]]] = st.size() - st.order_of_key(p) - 1;
        }
        st.clear();
        for (int i = n-1; i >= 0; i--) {
                pair<int,int> p = {abs(ranges[i].s), n-i};
                st.insert(p);
                contain[idx[ranges[i]]] = st.order_of_key(p);
        }

        for (auto i: contain) cout << i << "␣";
        cout << endl;
        for (auto i: contained) cout << i << "␣";
        cout << endl;

        return 0;
}
```

8.8.0.19 Alocação de Quartos de Hotel por Chegada e Saida

Objetivo

Existe um grande hotel, e n clientes chegarão em breve. Cada cliente quer ter um quarto individual. Você sabe o dia de chegada e o dia de saída de cada cliente. Dois clientes podem ficar no mesmo quarto se o dia de saída do primeiro cliente for anterior ao dia de chegada do segundo cliente. Qual é o número mínimo de quartos necessários para acomodar todos os clientes? E como os quartos podem ser alocados?

Estratégia

A solução utiliza um algoritmo guloso para alocar os quartos. A ideia é ordenar os clientes em ordem crescente de dia de chegada e, em seguida, iterar sobre os clientes. Para cada cliente, o algoritmo verifica se há um quarto disponível. Se houver, o algoritmo aloca o quarto ao cliente. Caso contrário, o algoritmo aloca um novo quarto ao cliente.

Código

```
int main()
{
```

```
3
4       int n;
5       cin >> n;
6       vector<pair<ii,int>> customers;
7       for (int i = 0; i < n; i++) {
8           int a, b; cin >> a >> b;
9           customers.push_back({{a, b},i});
10      }
11
12      sort(customers.begin(), customers.end());
13      set<ii> st;
14      vector<int> rooms(n, 0);
15      int maxr = 0, lastr = 0;
16      for (int i = 0; i < n; i++) {
17          auto it = st.begin();
18          if (st.empty() || (*it).f >= customers[i].f.f) {
19              lastr++;
20              st.insert({customers[i].f.s, lastr});
21              rooms[customers[i].s] = lastr;
22          }
23          else {
24              int aux = (*it).s;
25              rooms[customers[i].s] = aux;
26              st.erase(it);
27              st.insert({customers[i].f.s, aux});
28          }
29          maxr = max(maxr, (int)st.size());
30      }
31      cout << maxr << endl;
32      for (auto i : rooms) cout << i << "␣";
33      cout << endl;
34
35      return 0;
36  }
```

8.8.0.20 Busca Binaria Minimização - Tempo Mínimo de Produção

---

Objetivo

Uma fábrica possui n máquinas que podem ser usadas para fazer produtos. Seu objetivo é fazer um total de t produtos. Para cada máquina, você sabe o número de segundos que ela precisa para fazer um único produto. As máquinas podem trabalhar simultaneamente, e você pode decidir livremente seu cronograma. Qual é o menor tempo necessário para fazer t produtos?

Estratégia

A solução utiliza uma estratégia de ordenação e busca binária. Primeiramente, ordenamos as máquinas pelo tempo que levam para produzir um item. Em seguida, utilizaremos a busca binária para determinar o menor tempo possível para produzir t produtos.

Para a busca binária, definimos um intervalo de tempo que representa o tempo necessário para produzir t produtos. O limite inferior do intervalo é o tempo que a máquina mais rápida leva para produzir t produtos, e o limite superior é o tempo que a máquina mais lenta leva para produzir t produtos. Em cada iteração, calculamos o tempo médio do

intervalo e verificamos se é possível produzir t produtos nesse tempo.

Para verificar se é possível produzir t produtos no tempo médio, iteramos sobre as máquinas e calculamos quantos produtos cada máquina pode produzir nesse tempo. Se a soma dos produtos produzidos por todas as máquinas for maior ou igual a t, então é possível produzir t produtos nesse tempo. Caso contrário, precisamos aumentar o tempo médio.

Código

```
vector<ll> fact;

ll calc(ll time, ll t) {

        ll total = 0;
        for(auto f: fact) {
                total += time/f;
                if(total >= t) break;
        }
        return total;
}

int main() {

        ll n, t; cin >> n >> t;

        fact.resize(n);
        for (int i = 0; i < n; i++) cin >> fact[i];

        ll ans = 1e18;
        for (ll i = ans/2; i >= 1; i /= 2)
                while(calc(ans-i, t) >= t) {
                        ans-=i;
                }
        cout << ans << endl;

        return 0;
```

8.8.0.21 Maximização de Recompensa em Tarefas

Objetivo

Você tem que processar n tarefas. Cada tarefa possui uma duração e um prazo, e você processará as tarefas em alguma ordem, uma após a outra. Sua recompensa por uma tarefa é $d - f$, onde d é seu prazo e f é seu tempo de término. (O tempo de início é 0, e você tem que processar todas as tarefas, mesmo que uma tarefa gere uma recompensa negativa.)

Qual é sua recompensa máxima se você agir de forma ótima?

Estratégia

A solução usa ordenação por prazo, seguido de um algoritmo guloso. Primeiro, ordenamos as tarefas de acordo com seus prazos. Em seguida, processamos as tarefas em ordem crescente de prazo, escolhendo sempre a tarefa com menor duração dentre as disponíveis.

Código

```
int main() {

        int n; cin >> n;
        vector<pll> tasks(n);
        for (int i = 0; i < n; i++) {
                cin >> tasks[i].f >> tasks[i].s;
        }
        sort(tasks.begin(), tasks.end());
        ll ans = 0, time = 0;
        for (auto p : tasks) {
                time += (p.f);
                ans += (p.s-time);
        }
        cout << ans << endl;

}
```

8.8.0.22 Três Valores que Somam X

Objetivo

Você recebe um array de n inteiros, e sua tarefa é encontrar três valores (em posições distintas) cuja soma seja x.

Estratégia

A solução utiliza dois loops aninhados para iterar sobre todos os trios possíveis de números no array. Utilizamos ponteiros no loop interno para otimizar.

Código

```
int main() {

    ll n, x;
    cin >> n >> x;
    vector<pll> vec(n);

    for (int i = 0; i < n; i++) {
        cin >> vec[i].f;
        vec[i].s = i;
    }
    sort(vec.begin(), vec.end());
    for (int i = 0; i < n; i++) {
        ll l = 0, r = n-1;
        while(l != r) {
            if(i != l && i != r && vec[l].f + vec[r].f + vec[i].f == x) {
                cout << vec[i].s+1 << "␣" << vec[l].s+1 << "␣" << vec[r].s+1;
                cout << endl;
                return 0;
            }
            if(vec[l].f + vec[r].f+vec[i].f < x) l++;
            else r--;
        }
```

```
23      }
24      cout << "IMPOSSIBLE" << endl;
25
26      return 0;
27  }
```

8.8.0.23 Quatro Valores que somam X

Objetivo

Você é dado um array de n inteiros, e sua tarefa é encontrar quatro valores (em posições distintas) cuja soma é x.

Estratégia

A solução usa uma técnica de hash. Primeiramente, percorremos todos os pares possíveis de números no array. Para cada par, calculamos sua soma s. Em seguida, verificamos para cada par se $x - s$ existe no array. Se existir, então encontramos a solução. Para tornar essa verificação eficiente, usamos um mapa hash.

Código

```
1  int main() {
2
3      ll n, x;
4      cin >> n >> x;
5      vector<ll> vec(n);
6      map<ll,pll> mp;
7
8      for (int i = 0; i < n; i++) cin >> vec[i];
9
10     for (int i = 0; i < n; i++)
11         for (int j = i+1; j < n; j++)
12             mp[vec[i] + vec[j]] = {i,j};
13
14     for (int i = 0; i < n; i++) {
15         for (int j = i+1; j < n; j++) {
16             ll target = x-vec[i]-vec[j];
17             if(mp.find(target) != mp.end() &&
18                mp[target].f != i && mp[target].s != j &&
19                mp[target].s != i && mp[target].f != j) {
20                 cout << i+1 << "␣" << j+1 << "␣";
21                 cout << mp[target].f+1 << "␣" << mp[target].s+1;
22                 cout << endl;
23                 return 0;
24             }
25         }
26     }
27     cout << "IMPOSSIBLE" << endl;
28
29 }
```

8.8.0.24 Contagem de Subarrays com Soma Específica

Objetivo

Dado um array de n inteiros positivos, sua tarefa é contar o número de subarrays que possuem soma x.

Estratégia

A solução usa um mapa para armazenar a soma dos prefixos do array e o número de vezes que cada soma aparece. Para cada elemento do array, calculamos a soma do prefixo até o elemento atual e, em seguida, verificamos se a soma x menos a soma do prefixo atual já existe no mapa. Se existir, adicionamos o número de vezes que essa soma aparece ao nosso contador de subarrays.

Código

```
int main() {

        int n, k; cin >> n >> k;

        map<ll,ll> mp;
        mp[0] = 1;

        ll acc = 0, ans = 0;
        while(n--) {
                ll x; cin >> x;
                acc += x;

                ans += mp[acc-k];

                mp[acc]++;
        }
        cout << ans << endl;

}
```

8.8.0.25 Contagem de Subarrays Divisiveis

Objetivo

Dado um array de n inteiros, sua tarefa é contar o número de subarrays onde a soma dos valores é divisível por n.

Estratégia

A solução utiliza a propriedade de que a soma dos valores de um subarray é igual à soma dos valores do array completo até o último elemento do subarray, menos a soma dos valores do array completo até o primeiro elemento do subarray. Podemos usar um mapa para armazenar as somas dos valores do array completo até cada índice, e contar quantas vezes cada resto da divisão por n aparece. O número de subarrays com soma divisível por n será igual ao número de pares de índices onde as somas até esses índices têm o mesmo resto da divisão por n.

Código

```
int main() {

        int n; cin >> n;
        map<ll,ll> mp;
        mp[0] = 1;
```

```
6           ll acc = 0;
7           for (int i = 0; i < n; i++) {
8                   ll x; cin >> x;
9                   acc += x;
10                  mp[(acc%n + n) % n]++;
11          }
12
13          ll ans = 0;
14          for (auto i: mp) ans += (i.s*(i.s-1)/2);
15          cout << ans << endl;
16
17          return 0;
18
19  }
```

8.8.0.26 Contagem de Subarrays com Limite de Valores Distintos

---

Objetivo

Dado um array de n inteiros, sua tarefa é calcular o número de subarrays que possuem no máximo k valores distintos.

Estratégia

A solução utiliza a técnica de janela deslizante. Mantemos uma janela que representa um subarray e mantemos um mapa para rastrear a contagem de valores distintos dentro da janela. Expandimos a janela para a direita até que o número de valores distintos exceda k. Em seguida, contraímos a janela da esquerda até que o número de valores distintos seja menor ou igual a k. Para cada posição do início da janela, contamos o número de subarrays que podem ser formados com esse início.

Código

```
1  int main() {
2
3      int n, k;
4      cin >> n >> k;
5      map<int, int> freq;
6
7      vector<int> nums(n);
8      for(int i = 0; i < n; i++)
9          cin >> nums[i];
10
11     ll j = 0, ans = 0;
12     for(int i = 0; i < n; i++) {
13
14         while(j < n && (freq.size() < k  || freq.count(nums[j]) > 0 )) {
15             freq[nums[j]]++;
16             j++;
17         }
18         ans += j-i;
19         freq[nums[i]]--;
20         if(freq[nums[i]] == 0) freq.erase(nums[i]);
21     }
22
```

```
23      cout << ans << endl;
24
25      return 0;
26  }
```

8.8.0.27 Minimização da Soma Máxima em Subarrays

Objetivo

Você recebe um array contendo n inteiros positivos.

Sua tarefa é dividir o array em k subarrays de forma que a soma máxima em um subarray seja a menor possível.

Estratégia

A solução utiliza busca binária e um algoritmo guloso.

1. **Busca Binária:** A busca binária é aplicada sobre o intervalo de possíveis somas máximas. Para cada valor médio mid, verificamos se podemos dividir o array em k subarrays com a soma máxima de cada subarray sendo no máximo mid.

2. **Algoritmo Guloso:** O algoritmo guloso é usado para verificar se podemos dividir o array em k subarrays com a soma máxima de cada subarray sendo no máximo mid. Iteramos sobre o array e adicionamos os elementos a um subarray até que a soma do subarray exceda mid. Nesse momento, criamos um novo subarray e continuamos o processo.

Código

```
1  bool check(ll maxi, int k) {
2      ll acc = 0, groups = 0;
3      for (int i = 0; i < (int)nums.size(); i++) {
4          if (nums[i] > maxi) return false;
5          if (acc + nums[i] > maxi) {
6              groups++;
7              acc = 0;
8          }
9          acc += nums[i];
10     }
11     if (acc > 0) groups++;
12     return groups <= k;
13
14
15 nt main() {
16     ll n, k;
17     cin >> n >> k;
18
19     nums.resize(n);
20     for (int i = 0; i < n; i++) cin >> nums[i];
21
22     ll r = 2e5 * 1e9, l = 1;
23     while(l < r) {
24         ll mid = (r + l)/2;
25
26         if(check(mid, k))   r = mid;
```

```
27          else l = mid + 1;
28
29      }
30
31      cout << l << endl;
32
33      return 0;
```

8.8.0.28 Custo Mínimo para Equalizar Janelas

Objetivo

Você recebe um array de n inteiros. Sua tarefa é calcular para cada janela de k elementos, da esquerda para a direita, o custo total mínimo para tornar todos os elementos iguais.

Você pode aumentar ou diminuir cada elemento com custo x, onde x é a diferença entre o novo e o valor original. O custo total é a soma desses custos.

Estratégia

A solução baseia-se em maneter um multiset da esquerda e um da direita, mantendo a soma de cada um, para cada elemento verificamosse ele ira alterar a mediana, com isso alteramos o custo, caso contrariom podemos manter a mediana.

Código

```
1  int main() {
2
3          int n, k; cin >> n >> k;
4          vector<ll> nums(n);
5          for (int i = 0; i < n; i++) cin >> nums[i];
6
7          if(k == 1) {
8                  for (int i = 0; i < n; i++) cout << 0 << "␣";
9                  cout << endl;
10                 return 0;
11         }
12
13         vector<ll> aux(nums.begin(), nums.begin()+k);
14         sort(aux.begin(), aux.end());
15
16         multiset<ll> left, right;
17         ll lsum = 0, rsum = 0, lsz = ceil(k/2.0), rsz = k/2;
18         for (int i = 0; i < k; i++) {
19                 if(i < lsz) {
20                         left.insert(aux[i]);
21                         lsum += aux[i];
22                 } else {
23                         right.insert(aux[i]);
24                         rsum += aux[i];
25                 }
26         }
27
28         int start = 0;
```

```
29          for (int i = k; i <= n; i++, start++) {
30                  ll med = *left.rbegin();
31                  cout << lsz*med - lsum + rsum - rsz*med << "␣";
32                  if(i == n) break;
33
34                  if(left.find(nums[start]) != left.end()) {
35                          left.erase(left.find(nums[start]));
36                          lsum -= nums[start];
37                          int xi = *right.begin();
38                          left.insert(xi);
39                          right.erase(right.begin());
40                          lsum += xi;
41                          rsum -= xi;
42                  }
43                  else {
44                          right.erase(right.find(nums[start]));
45                          rsum -= nums[start];
46                  }
47
48                  med = *left.rbegin();
49                  if(nums[i] > med) {
50                          right.insert(nums[i]);
51                          rsum += nums[i];
52                  }
53                  else {
54                          left.insert(nums[i]);
55                          lsum += nums[i];
56                          med = *left.rbegin();
57                          right.insert(med);
58                          left.erase(left.find(med));
59                          lsum -= med;
60                          rsum += med;
61                  }
62          }
63
64          cout << endl;
65
66          return 0;
67  }
```

8.8.0.29 Janela Deslizante - Mediana

Objetivo

Você recebe um array de n inteiros. Sua tarefa é calcular a mediana de cada janela de k elementos, da esquerda para a direita. A mediana é o elemento do meio quando os elementos são ordenados. Se o número de elementos for par, há duas medianas possíveis e assumimos que a mediana é a menor delas.

Estratégia

A solução utiliza ordered_set para armazenar os elementos em cada janela e manter a ordem deles. A cada nova janela, adicionamos o novo elemento ao multiset e removemos o elemento que está fora da janela. Em seguida, calculamos a mediana usando o find_by_order.

Código

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

template<class T> using oset =
        tree<T, null_type, less_equal<T>, rb_tree_tag,
        tree_order_statistics_node_update>;

int main() {

        int n, k; cin >> n >> k;
        vector<int> nums(n);
        for (int i = 0; i < n; i++) cin >> nums[i];

        int start = 0;
        oset<int> window;
        for(auto xi: nums) {
                window.insert(xi);

                if((int)window.size() >= k) {
                        cout << *window.find_by_order(((k - 1)/2)) << "␣";
                        auto it = window.upper_bound(nums[start]);
                        window.erase(it);

                        start++;
                }
        }

        cout << endl;

        return 0;
}
```

8.8.0.30 Gredy Scheduling - Mais pessoas

Objetivo

Em um festival de cinema, n filmes serão exibidos. O clube de cinema de Syrjälä consiste em k membros, que estarão todos presentes no festival. Você sabe o horário de início e término de cada filme. Qual é o número total máximo de filmes que os membros do clube podem assistir inteiramente se eles agirem de forma ótima?

Estratégia

A solução usa ordenação por tempo de término, seguido de um algoritmo guloso. Primeiro, ordenamos os filmes de acordo com seus tempos de término. Em seguida, processamos os filmes em ordem crescente de tempo de término, escolhendo sempre o filme com o horário de início mais cedo dentre as disponíveis, desde que ele não se sobreponha ao filme anterior. Considerando que temos mais de uma pessoa, sempre teremos k filmes para serem vistos, usamos ordered_set para manter o ultimo filme que cada um esta assistindo.

Código

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

template<class T> using oset =
        tree<T, null_type, less_equal<T>, rb_tree_tag,
        tree_order_statistics_node_update>;

int main() {

        int n, k; cin >> n >> k;
        vector<pair<int,int>> vec(n);
        for (int i = 0; i < n; i++)
                cin >> vec[i].s >> vec[i].f;
        sort(vec.begin(), vec.end());

        int ans = 0;
        oset<pair<int,int>> st;
        for(auto p: vec) {
                int order = st.order_of_key({p.s,p.f});
                if(order > 0)
                        st.erase(st.find_by_order(order-1));
                if((int)st.size() < k) {
                        st.insert({p.f,p.s});
                        ans++;
                }
        }

        cout << ans << endl;
}
```

8.8.0.31 Soma Máxima em Subarrays de Comprimento Intervalar

Objetivo

Dado um array de n inteiros, sua tarefa é encontrar a soma máxima de valores em um subarray contíguo com comprimento entre a e b.

Estratégia

A solução utiliza uma abordagem de janela deslizante para calcular a soma máxima de todos os subarrays possíveis com comprimentos entre a e b.

1. **Janela Deslizante:** Inicializamos uma janela de tamanho a e calculamos a soma dos elementos dentro dela. 2. **Iteração:** Movemos a janela para a direita, adicionando o próximo elemento e removendo o elemento mais à esquerda da janela. 3. **Atualização da Soma Máxima:** A cada passo, atualizamos a soma máxima com a soma atual da janela. 4. **Comprimentos entre a e b:** Após a janela atingir o tamanho b, reduzimos seu tamanho para a e repetimos os passos anteriores.

Código

```
int main() {

        int n, a, b; cin >> n >> a >> b;
        vector<ll> pref(n+1,0);
        for (int i = 1; i <= n; i++) {
                ll xi; cin >> xi;
                pref[i] = pref[i-1] + xi;
        }
        multiset<ll> ms;
        ll ans = -LONG_LONG_MAX;
        for (int i = a; i <= n; i++) {

                if(i > b) {
                        ms.erase(ms.find(pref[i-b-1]));
                }
                ms.insert(pref[i-a]);
                ans = max(ans, pref[i] - *ms.begin());

        }
        cout << ans << endl;

        return 0;
```

8.9 CSES 490

8.9.1 Jogo de teletransporte

Objetivo

Você está jogando um jogo com n planetas e m teletransportes entre eles. Todos os teletransportes funcionam em ambas as direções e conectam dois planetas diferentes. Você começa o jogo em um planeta específico e, a cada turno, você se move através de um teletransporte para um planeta diferente. Você pode assumir que todos os planetas estão conectados por teletransportes.

Sua tarefa é responder a q consultas da forma: "é possível começar o jogo no planeta a e acabar no planeta b após x turnos?"

Estratégia

Para solucionar o problema, basta notar que, como não há restrições de repetição de arestas, é suficiente que dado x:

- é preciso existir um caminho com tamanho maior ou igual a x de a até b.
- satisfeita a primeira condição, basta que exista um caminho com tamanho cuja paridade é igual a x. Isso porque, alcançado o destino final, é possível fazer. enquanto o caminho não alcança o tamanho x, idas e voltas em uma mesma aresta.

Código

```
vector<int> adj[MAXN];
int ans[MAXN][MAXN][2];
bool vis[MAXN][2];
int n, m, q;
int a, b, x;

BEGIN
{
    cin >> n >> m >> q;
    FOR(i,0,m) {
        cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    memset(ans, SCHAR_MAX, sizeof(ans));

    FOR(i,1,n+1) {
        queue<pair<int,int>> Q;

        memset(vis,0,sizeof(vis));
        ans[i][i][0] = 0;
        vis[i][0] = 1;
        Q.push({i,0});

        while(Q.size()) {
            int u = Q.front().first;
            int len = Q.front().second;
            Q.pop();

            for(int v : adj[u]) {
                if(vis[v][!(len%2)]) continue;
                vis[v][!(len%2)] = 1;
                ans[i][v][!(len%2)] = len+1;
                Q.push({v,len+1});
            }
        }
    }

    while(q--) {
        cin >> a >> b >> x;
        cout << (ans[a][b][x%2] <= x ? "YES" : "NO") << endl;
    }
}
END
```

8.9.2 Acampamento

Objetivo

Você chegou a um acampamento e quer encontrar um local para acampar que fique o mais longe possível de outros campistas. O acampamento pode ser representado como uma tabela, em que cada quadrado pode conter um parque

de campismo reservado ou livre. A distância entre dois quadrados (x_1, y_1) e (x_2, y_2) é calculada com a fórmula $|x_1 - x_2| + |y_1 - y_2|$.

Estratégia

Código

```
#include <bits/stdc++.h>

#define all(x) begin(x),end(x)
#define fi first
#define se second

using namespace std;
using ll = long long;
using pic = pair<int, char>;
using pii = pair<int, int>;

struct segtree {
    int l, r, m;
    segtree *lc = nullptr, *rc = nullptr;
    int v = -1e9;

    segtree(int L, int R) : l(L), r(R) {
        m = (l + r) / 2;
        if (l == r) return;
        lc = new segtree(l, m);
        rc = new segtree(m+1, r);
    }

    void point_upd(int i, int u) {
        if (l == i && i == r) {
            v = max(v, u);
            return;
        }
        if (i <= m) lc->point_upd(i, u);
        else rc->point_upd(i, u);
        v = max(lc->v, rc->v);
    }

    int range_max(int L, int R) {
        if (L > R) return -1e9;
        if (l == L && r == R) return v;
        return max(lc->range_max(L, min(R, m)), rc->range_max(max(m+1, L), R));
    }

    void clear() {
        for (auto u : {lc, rc}) if (u != nullptr) {
            u->clear();
            delete u;
        }
    }
};

void solve() {
    int n, m;
```

```
50      cin >> n >> m;
51
52      vector<array<int, 5>> pts(n+m);
53      vector<int> yv(n+m);
54      for (int i = 0; i < n; ++i) {
55          cin >> pts[i][0] >> pts[i][1];
56          yv[i] = pts[i][1];
57          pts[i][2] = i;
58          pts[i][3] = 0;
59      }
60      for (int i = n; i < n+m; ++i) {
61          cin >> pts[i][0] >> pts[i][1];
62          yv[i] = pts[i][1];
63          pts[i][2] = i-n;
64          pts[i][3] = 1;
65      }
66      sort(all(pts), [](array<int, 5> &a, array<int, 5> &b) {
67          return a[1] < b[1];
68      });
69      sort(all(yv));
70      pts[0][4] = 0;
71      for (int i = 1; i < n+m; ++i) pts[i][4] = pts[i-1][4] + (pts[i][1] != pts[i-1][1]);
72      int w = pts[n+m-1][4];
73      sort(all(pts));
74      segtree left(0, w), right(0, w);
75      vector<int> dis(m, 1e9);
76      for (auto &p : pts) {
77          if (p[3] == 0) {
78              left.point_upd(p[4], p[0] + p[1]);
79              right.point_upd(p[4], p[0] - p[1]);
80          } else {
81              dis[p[2]] = min(dis[p[2]], p[0] + p[1] - left.range_max(0, p[4]));
82              dis[p[2]] = min(dis[p[2]], p[0] - p[1] - right.range_max(p[4], w));
83          }
84      }
85      left.clear();
86      right.clear();
87      left = segtree(0, w);
88      right = segtree(0, w);
89      reverse(all(pts));
90      for (auto &p : pts) {
91          if (p[3] == 0) {
92              left.point_upd(p[4], - p[0] + p[1]);
93              right.point_upd(p[4], - p[0] - p[1]);
94          } else {
95              dis[p[2]] = min(dis[p[2]], - p[0] + p[1] - left.range_max(0, p[4]));
96              dis[p[2]] = min(dis[p[2]], - p[0] - p[1] - right.range_max(p[4], w));
97          }
98      }
99      left.clear();
100     right.clear();
101
102     cout << *max_element(all(dis)) << '\n';
103 }
104
```

```
105 int main() {
106     cin.tie(0) -> sync_with_stdio(0);
107 
108     solve();
109 
110     return 0;
111 }
```

8.9.3 Caminhos

Objetivo

Você recebe um grafo direcionado com n nós e m arestas. O gráfico não possui ciclos, o que significa que a partir de qualquer nó você não pode voltar ao mesmo nó seguindo as arestas. Determine se dois caminhos podem ser formados no gráfico de modo que cada nó do gráfico aparece exatamente em um dos caminhos. Observe que todas as arestas de o gráfico não precisa aparecer nos caminhos.

Estratégia

Código

```
 1 #include <bits/stdc++.h>
 2 
 3 #define all(x) begin(x),end(x)
 4 #define fi first
 5 #define se second
 6 
 7 using namespace std;
 8 using ll = long long;
 9 using pic = pair<int, char>;
10 using pii = pair<int, int>;
11 
12 const int N = 200200;
13 vector<int> adj[N], rev[N];
14 int has[N];
15 vector<int> cur;
16 
17 vector<int> ord;
18 bool vis[N];
19 
20 void dfs(int u) {
21     if (vis[u]) return;
22     vis[u] = 1;
23     for (int v : adj[u]) dfs(v);
24     ord.push_back(u);
25 }
26 
27 void solve() {
28     int n, m;
29     cin >> n >> m;
30 
```

```
31      for (int i = 0; i < m; ++i) {
32          int u, v;
33          cin >> u >> v;
34          adj[u].push_back(v);
35          rev[v].push_back(u);
36      }
37
38      for (int i = 1; i <= n; ++i) {
39          rev[i].push_back(n+1);
40          rev[i].push_back(n+2);
41      }
42
43      for (int i = 1; i <= n; ++i) dfs(i);
44      reverse(all(ord));
45      cur = {n+1, n+2};
46      has[n+1] = has[n+2] = 1;
47      vector<pii> evt;
48      for (int i = 1; i < n; ++i) {
49          int u = ord[i-1];
50          int v = ord[i];
51          bool insu = 0;
52          for (int j : rev[v]) insu |= has[j];
53          if (!count(all(adj[u]), v)) {
54              for (int j : cur) {
55                  has[j] = 0;
56                  evt.push_back({i, j});
57              }
58              cur.clear();
59          }
60          if (insu) {
61              cur.push_back(u);
62              has[u] = 1;
63          }
64          if (cur.empty()) {
65              cout << "NO\n";
66              return;
67          }
68      }
69
70      reverse(all(evt));
71      vector<int> p, q;
72      int a = ord[n-1];
73      int i = n-1, j = 0;
74      while (i >= 0) {
75          swap(p, q);
76          if (q.empty()) {
77              a = 0;
78              while (!has[a]) ++a;
79          } else {
80              for (int j : rev[q.back()]) if (has[j]) {
81                  a = j;
82                  break;
83              }
84          }
85          has[a] = 0;
```

```
86          while (i >= 0 && ord[i] != a) {
87              p.push_back(ord[i]);
88              while (j < evt.size() && evt[j].fi == i) {
89                  has[evt[j].se] = 1;
90                  ++j;
91              }
92              --i;
93          }
94      }
95      reverse(all(p));
96      reverse(all(q));
97      cout << "YES\n";
98      cout << p.size() << '␣';
99      for (int i : p) cout << i << '␣';
100     cout << '\n';
101     cout << q.size() << '␣';
102     for (int i : q) cout << i << '␣';
103     cout << '\n';
104 }
105
106 int main() {
107     cin.tie(0) -> sync_with_stdio(0);
108
109     solve();
110
111     return 0;
112 }
```

Capítulo 9

Maratona Mineira 2023

9.0.0.1 14-bis - Maior Subarray com Diferença de 1

Objetivo

Para melhorar as chances de decolar sua lendária aeronave, o 14-bis, Santos-Dumont precisava de uma pista de decolagem que fosse bastante reta e o mais comprida possível. Uma pista de decolagem é descrita como uma sequência de números (representando a altura de cada trecho da pista), que é considerada reta se a diferença em valor absoluto entre duas posições adjacentes nessa sequência não é maior que um (não queremos estragar o trem de pouso da nossa querida aeronave de papel com terrenos acidentados!).

Dado um mapa topográfico da área, descrito como uma matriz $N \times M$ (em que cada valor da matriz guarda a altura da posição), imprima o tamanho da maior pista de decolagem possível, sendo que as pistas de decolagem podem ser no sentido norte-sul (intervalo de uma coluna da matriz) ou leste-oeste (intervalo de uma linha da matriz).

Estratégia

A solução para esse problema consiste em percorrer cada linha e cada coluna da matriz, verificando a maior sequência de números que satisfazem a condição de ”pista reta”. A resolução é baseada no conceito de ”tamanho da pista até o momento”e ”tamanho da pista até o momento que termina no ponto atual”. A cada posição, o código compara a altura atual com a altura da posição anterior e, se a diferença for menor ou igual a 1, a pista atual é estendida. Caso contrário, o código reinicia a contagem do tamanho da pista.

Código

```
int solve(const std::vector<std::vector<int>>& a) {
    int ans = 0;

    for (int i = 0; i < a.size(); i++) {
        int r = 0;
        for (int l = 0; l < a[i].size(); l++) {
            while (r < a[i].size() and (l == r or abs(a[i][r] - a[i][r - 1]) <= 1))
                r++;
            ans = std::max(ans, r - l);
        }
    }

```

```
13      return ans;
14  }
15
16  int main() { _
17      int n, m;
18      std::cin >> n >> m;
19      std::vector<std::vector<int>> a(n, std::vector<int>(m));
20      std::vector<std::vector<int>> at(m, std::vector<int>(n));
21
22      for (int i = 0; i < n; i++) {
23          for (int j = 0; j < m; j++) {
24              std::cin >> a[i][j];
25              at[j][i] = a[i][j];
26          }
27      }
28
29      std::cout << std::max(solve(a), solve(at)) << std::endl;
30
31      return 0;
32  }
```

9.0.0.2 Árvore Mágica de Bacon - HLD

Objetivo

Árvores mágicas possuem luz própria, são enraizadas, e funcionam da seguinte forma:

* A árvore é um grafo conexo sem ciclos. * A árvore inicia-se com todos os vértices apagados. * Uma sub-árvore inteira pode se acender ou se apagar. * Um vértice interno da árvore está aceso se e somente se todos os seus filhos estão acesos.

Além da magia da luz própria, a quarta propriedade também é bem misteriosa. Note que, se a sub-árvore enraizada em v é desligada, todos os ancestrais de v ligados também devem ser desligados; por outro lado, se a sub-árvore de v é ligada, pode ser que tenhamos de ligar o pai de v, se todos seus filhos agora estiverem ligados, e assim sucessivamente.

Na 13ª Festa da Cheia de seu reinado, Bacon, O Próspero, levou seu bisneto, que viria a ser o rei Bacon, O Grafo, para ver a árvore mágica plantada no palácio. Como todo bom amante da combinatória, a pequena capivara perguntava incessantemente ao bisavô quantos vértices de uma dada sub-árvore estavam acesos. Infelizmente, nosso herói não é lá muito bom nessa área, e pediu a você, Grande Maratonista, para o ajudar a responder as perguntas do futuro monarca!

Estratégia

Código

```
1  const int MAX = 5e5+10;
2
3  namespace seg {
4          int seg[4*MAX], lazy[4*MAX];
5          int n, *v;
6
7          ll build(int p=1, int l=0, int r=n-1) {
8                  lazy[p] = -1;
9                  if (l == r) return seg[p] = v[l];
```

```
10                  int m = (l+r)/2;
11                  return seg[p] = build(2*p, l, m) + build(2*p+1, m+1, r);
12          }
13          void build(int n2, int* v2) {
14                  n = n2, v = v2;
15                  build();
16          }
17          void prop(int p, int l, int r) {
18                  if (lazy[p] == -1) return;
19                  seg[p] = lazy[p]*(r-l+1);
20                  if (l != r) lazy[2*p] = lazy[p], lazy[2*p+1] = lazy[p];
21                  lazy[p] = -1;
22          }
23          int query(int a, int b, int p=1, int l=0, int r=n-1) {
24                  prop(p, l, r);
25                  if (a <= l and r <= b) return seg[p];
26                  if (b < l or r < a) return 0;
27                  int m = (l+r)/2;
28                  return query(a, b, 2*p, l, m) + query(a, b, 2*p+1, m+1, r);
29          }
30          int update(int a, int b, int x, int p=1, int l=0, int r=n-1) {
31                  prop(p, l, r);
32                  if (a <= l and r <= b) {
33                          lazy[p] = x;
34                          prop(p, l, r);
35                          return seg[p];
36                  }
37                  if (b < l or r < a) return seg[p];
38                  int m = (l+r)/2;
39                  return seg[p] = update(a, b, x, 2*p, l, m) + update(a, b, x, 2*p+1, m+1, r);
40          }
41  };
42
43  namespace hld {
44          vector<int> g[MAX];
45          int pos[MAX], sz[MAX];
46          int peso[MAX], pai[MAX];
47          int h[MAX], v[MAX], t;
48          int at[MAX];
49
50          void build_hld(int k, int p = -1, int f = 1) {
51                  at[t] = k;
52                  v[pos[k] = t++] = peso[k]; sz[k] = 1;
53                  for (auto& i : g[k]) if (i != p) {
54                          pai[i] = k;
55                          h[i] = (i == g[k][0] ? h[k] : i);
56                          build_hld(i, k, f); sz[k] += sz[i];
57
58                          if (sz[i] > sz[g[k][0]] or g[k][0] == p) swap(i, g[k][0]);
59                  }
60                  if (p*f == -1) build_hld(h[k] = k, -1, t = 0);
61          }
62          void build(int root = 0) {
63                  t = 0;
64                  build_hld(root);
```

```
65                  pai[root] = -1;
66                  seg::build(t, v);
67          }
68          int query_path(int a, int b) {
69                  if (pos[a] < pos[b]) swap(a, b);
70
71                  if (h[a] == h[b]) return seg::query(pos[b], pos[a]);
72                  return seg::query(pos[h[a]], pos[a]) + query_path(pai[h[a]], b);
73          }
74          void update_path(int a, int b, int x) {
75                  if (pos[a] < pos[b]) swap(a, b);
76
77                  if (h[a] == h[b]) return (void)seg::update(pos[b], pos[a], x);
78                  seg::update(pos[h[a]], pos[a], x); update_path(pai[h[a]], b, x);
79          }
80          ll query_subtree(int a) {
81                  return seg::query(pos[a], pos[a]+sz[a]-1);
82          }
83          void update_subtree(int a, int x) {
84                  seg::update(pos[a], pos[a]+sz[a]-1, x);
85          }
86          void sobe(int a) {
87                  if (a == -1) return;
88                  if (seg::query(pos[h[a]], pos[h[a]]+sz[h[a]]-1) + pos[a]-pos[h[a]]+1 == sz[h[a]
89                          seg::update(pos[h[a]], pos[a], 1);
90                          return sobe(pai[h[a]]);
91                  }
92                  int l = pos[h[a]], r = pos[a]+1;
93                  while (l < r) {
94                          int m = (l+r)/2;
95                          if (seg::query(m, m+sz[at[m]]-1) + pos[a]-m+1 == sz[at[m]]) {
96                                  r = m;
97                          }
98                          else l = m+1;
99                  }
100                 seg::update(l, pos[a], 1);
101         }
102 }
103
104 int main() { _
105         int n; cin >> n;
106         for (int i = 1; i < n; i++) {
107                 int a, b; cin >> a >> b; a--, b--;
108                 hld::g[a].push_back(b);
109                 hld::g[b].push_back(a);
110         }
111         hld::build(0);
112         int q; cin >> q;
113         while (q--) {
114                 int t, a; cin >> t >> a; a--;
115                 if (t == 1 and !seg::query(hld::pos[a], hld::pos[a])) {
116                         hld::update_subtree(a, 1);
117                         hld::sobe(hld::pai[a]);
118                 }
119                 if (t == 2) {
```

```
120                         hld::update_subtree(a, 0);
121                         hld::update_path(0, a, 0);
122                 }
123                 if (t == 3) cout << hld::query_subtree(a) << endl;
124         }
125         exit(0);
126 }
```

9.0.0.3 Contorno - COnexão de Cstablecimentos sem Cruzamento de Ruas

---

Objetivo

A Avenida do Contorno é uma avenida circular de Belo Horizonte. Durante sua construção, foram instalados vários estabelecimentos ao longo dela, como várias drogarias (Araújo), várias pastelarias (Rei do Pastel), etc. Por motivos logísticos, todo par de estabelecimentos do mesmo tipo deve ser conectado diretamente por uma rua (que pode passar por dentro ou por fora da Contorno). Porém, para que a Contorno continue sendo a avenida principal, não podem existir esquinas em outros lugares da cidade (cruzamento de ruas). Além disso, não haverá túneis ou viadutos na cidade. Dados os tipos de estabelecimentos que existem em ordem ao longo da Contorno, sua tarefa é determinar se é possível construir as ruas conectando todo par de estabelecimentos de mesmo tipo, sem criar esquinas (cruzamentos de ruas) extras.

Estratégia

A estratégia é usar um grafo para resolver os conflitos entre as conexões dos estabelecimentos, consideramos todas as arestas que devem ser construidas no grafo, estes serçao nossos vertices, e para cada par de aresta que intersecta, geramos uma aresta nesse par de vertices, após podemos verificar se o grafo é bipartido, ou pode ser colorido com duas cores, dessa maneira é possivel construir todas as conexões.

Código

```
1  const int MAX = 2e3+10;
2
3  vector<int> g[MAX];
4
5  bool inter(tuple<int, int, int> a, tuple<int, int, int> b) {
6          if (get<0>(b) < get<0>(a)) swap(a, b);
7          if (get<0>(a) == get<0>(b) or get<1>(a) == get<0>(b)) return false;
8          return get<0>(b) < get<1>(a) and get<1>(a) < get<1>(b);
9  }
10
11 bool bip;
12 int c[MAX];
13
14 void dfs(int i, int cc = 1) {
15         c[i] = cc;
16         for (int j : g[i]) {
17                 if (!c[j]) dfs(j, 3 - cc);
18                 else if (c[j] == c[i]) bip = false;
19         }
20 }
21
22 int main() { _
23         int n; cin >> n;
```

```
24          vector<vector<int>> pos(n);
25          for (int i = 0; i < n; i++) {
26                  int cor; cin >> cor; cor--;
27                  pos[cor].push_back(i);
28          }
29          vector<tuple<int, int, int>> par;
30          for (int i = 0; i < n; i++) {
31                  if (pos[i].size() > 4) return cout << "N" << endl, 0;
32                  for (int j = 0; j < pos[i].size(); j++) for (int k = j+1; k < pos[i].size(); k+
33                          par.emplace_back(pos[i][j], pos[i][k], i);
34          }
35          for (int i = 0; i < par.size(); i++) for (int j = i+1; j < par.size(); j++)
36                  if (inter(par[i], par[j])) g[i].push_back(j), g[j].push_back(i);
37          bip = true;
38          for (int i = 0; i < par.size(); i++) if (!c[i]) dfs(i);
39          if (bip) cout << "S" << endl;
40          else cout << "N" << endl;
41          exit(0);
42  }
```

9.0.0.4 Dever - Menor Base para Divisão Inteira

Objetivo

Joãozinho acabou de aprender sobre divisão na escola. A professora passou o dever de casa, com N divisões que Joãozinho deve fazer, e entregar o resultado. Para cada uma delas, Joãozinho deve calcular o valor de a_i/b_i, a_i, b_i inteiros positivos e $1iN$.

Porém, Joãozinho não é muito inteligente, e ele não entendeu muito bem como fazer quando há dízimas periódicas: ele acha que, se tiver que fazer $1/3 = 0.333333...$, ele nunca mais vai conseguir terminar o dever! Joãozinho ficou desesperado.

Entretanto, Joãozinho é muito sagaz. Ele achou um furo: notou que a professora não especificou a base numérica em que os alunos devem representar o valor da divisão.

Sendo assim, ajude Joãozinho a descobrir a menor base $B2$ tal que, se escrevermos a_i/b_i na base B, para todo i, não teremos dízimas periódicas. Como a resposta pode ser muito grande, imprima o resto da divisão de B por 998244353.

Estratégia

Uma fração a/b tem representação decimal finita se e somente se o denominador b é divisível apenas por potências da base Portanto, para encontrar a menor base B que garante representação finita para todas as frações a_i/b_i, precisamos encontrar o menor número que é divisível por todos os fatores primos de b_i.

Para isso, podemos fatorar cada b_i e identificar seus fatores primos. Em seguida, precisamos encontrar o mínimo múltiplo comum (MMC) desses fatores primos. O MMC é a menor base B que garante que todas as frações a_i/b_i tenham representação finita.

Código

```
1  vector<int> fact(int n) {
2          vector<int> ret;
3          for (int i = 2; i <= n/i; i++) if (n\%i == 0) {
4                  ret.push_back(i);
```

```
5                   while (n\%i == 0) n /= i;
6           }
7           if (n > 1) ret.push_back(n);
8           return ret;
9   }
10
11  int tem[MAX];
12
13  int main() { _
14          int n; cin >> n;
15          while (n--) {
16                  int a, b; cin >> a >> b;
17                  for (int i : fact(b / gcd(a, b))) tem[i] = 1;
18          }
19          if (accumulate(tem, tem + MAX, 0ll) == 0) return cout << 2 << endl, 0;
20          ll ans = 1;
21          for (int i = 0; i < MAX; i++) if (tem[i]) ans = ans * i % MOD;
22          cout << ans << endl;
23          exit(0);
24  }
```

9.0.0.5 Pastel - Jogo de Tabuleiros

Objetivo

O Jogo possui 3 tabuleiros retangulares, divididos de forma regular em posições quadradas. Em cada tabuleiro, há uma peça e obstáculos. A peça inicialmente está na posição superior esquerda. Em um movimento, um jogador escolhe um dos 3 tabuleiros e move a peça uma quantidade não nula de espaços para baixo ou para a direita de tal forma que ela não atravesse nem pare em um obstáculo. Jogam dois jogadores alternadamente e, na vez de um jogador, se não houver movimento válido a ser feito, aquele jogador perde.

Você vai jogar O Jogo contra seu amigo. Ele vai começar, mas, logo antes, ele vai passar numa pastelaria para comer um pastel. Você, obviamente, vai trapacear (não que você seja desonesto, você apenas gosta de oferecer partidas desafiadoras para seu amigo). Você vai fazer até um movimento em cada tabuleiro antes dele voltar (seu amigo é esperto, se você fizer mais de um movimento em um tabuleiro, ele vai notar).

De quantas formas você pode fazer até um movimento em cada tabuleiro, de tal forma que, quando seu amigo voltar, ele começa a jogar O Jogo e, se ambos jogarem de forma ótima, você ganha?

Estratégia

Código

```
1   vector<vector<int>> le() {
2           int n, m; cin >> n >> m;
3           vector ret(n, vector(m, int()));
4           for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) {
5                   char c; cin >> c;
6                   if (c == 'x') ret[i][j] = -1;
7           }
8           return ret;
9   }
10
11  vector<vector<int>> trans(vector<vector<int>> v) {
```

```
12          int n = v.size(), m = v[0].size();
13          vector ret(m, vector(n, int()));
14          for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) ret[j][i] = v[i][j];
15          return ret;
16  }
17
18  vector<vector<int>> get_grundy(vector<vector<int>> v) {
19          int n = v.size(), m = v[0].size();
20          vector<set<int>> baixo(m);
21          set<int> all;
22          for (int i = 0; i <= n+m; i++) all.insert(i);
23          vector<int> tirei;
24
25          for (int i = n-1; i >= 0; i--) {
26                  for (int j = m-1; j >= 0; j--) {
27                          if (v[i][j] == -1) {
28                                  baixo[j].clear();
29                                  for (int k : tirei) all.insert(k);
30                                  tirei.clear();
31                                  continue;
32                          }
33                          auto it = all.begin();
34                          for (int k : baixo[j]) {
35                                  if (*it == k) it++;
36                                  else if (*it < k) break;
37                          }
38                          int mex = *it;
39                          v[i][j] = mex;
40                          all.erase(mex);
41                          tirei.push_back(mex);
42                          baixo[j].insert(mex);
43                  }
44                  for (int k : tirei) all.insert(k);
45                  tirei.clear();
46          }
47          return v;
48  }
49
50  vector<vector<int>> grundy(vector<vector<int>> v) {
51          if (v.size() > v[0].size()) return trans(get_grundy(trans(v)));
52          return get_grundy(v);
53  }
54
55  vector<ll> mov(vector<vector<int>> v) {
56          v = grundy(v);
57          int n = v.size(), m = v[0].size();
58          vector<ll> ret(n+m);
59          for (int i = 0; i < n; i++) {
60                  if (v[i][0] >= 0) ret[v[i][0]]++;
61                  else break;
62          }
63          for (int j = 1; j < m; j++) {
64                  if (v[0][j] >= 0) ret[v[0][j]]++;
65                  else break;
66          }
```

```
67          return ret;
68  }
69
70  template<char op, class T> vector<T> FWHT(vector<T> f, bool inv = false) {
71          int n = f.size();
72          for (int k = 0; (n-1)>>k; k++) for (int i = 0; i < n; i++) if (i>>k&1) {
73                  int j = i^(1<<k);
74                  if (op == '^') f[j] += f[i], f[i] = f[j] - 2*f[i];
75                  if (op == '|') f[i] += (inv ? -1 : 1) * f[j];
76                  if (op == '&') f[j] += (inv ? -1 : 1) * f[i];
77          }
78          if (op == '^' and inv) for (auto& i : f) i /= n;
79          return f;
80  }
81
82  vector<ll> conv(vector<ll> a, vector<ll> b) {
83          int N = max(a.size(), b.size());
84          while (N&(N-1)) N++;
85          a.resize(N), b.resize(N);
86          a = FWHT<'^'>(a), b = FWHT<'^'>(b);
87          for (int i = 0; i < a.size(); i++) a[i] = a[i] * b[i];
88          return FWHT<'^'>(a, true);
89  }
90
91  int main() { _
92          cout << conv(mov(le()), conv(mov(le()), mov(le())))[0] << endl;
93          exit(0);
94  }
```

9.0.0.6 Sapo no Pântano - PD

Objetivo

Um sapo está no pântano, na frente de um rio. No rio há N vitórias-régias, organizadas em uma sequência, da esquerda para a direita. O sapo sabe que, ao pular da vitória-régia A para a vitória-régia B, a A afunda para sempre no rio. Além disso, da i-ésima vitória-régia o sapo gasta x_i de energia para pular para a esquerda (independente do tamanho do pulo), ou y_i para a direita.

O sapo, então, por diversão, irá escolher uma das vitórias-régias, pular nela, e depois pular para todas as outras antes de sair do rio, decidindo em cada momento pular para a esquerda ou direita. Como ele tem medo de cair no rio, em cada pulo, ele sempre pula para a vitória-régia mais próxima na direção escolhida que ainda não afundou. O sapo quer afundar todas as vitórias-régia dessa maneira, e, depois, pular para fora do rio. Se o caminho do sapo termina na posição i, o custo de pular dessa última vitória-régia para fora do rio é o mínimo entre x_i e y_i.

Ajude o sapo a descobrir alguma sequência de vitórias-régia que minimiza a energia total gasta.

Estratégia

Primeiro passo, para vitoria regia, definir se é melhor para a direita ou esquerda, com isso podemos escolher um inicio de maneira que eu consiga pular todas as direções ideias de cada vitoria regia, observação para o caso das bordas. Para resolução, rodamos um algoritmo guloso começando de do x e x+1, pois em cada um terminamos em uma borda diferente, e escolhemos o melhor deles

Código

```cpp
const int INF = 0x3f3f3f3f;
const ll LINF = 0x3f3f3f3f3f3f3f3fll;

pair<ll, vector<int>> solve(vector<pair<int, int>>& v, int at) {
        vector<int> ret;
        ll val = 0;
        int l = at, r = at;
        while (true) {
                ret.push_back(at);
                if (r+1 == v.size() and l == 0) {
                        val += min(v[at].first, v[at].second);
                        break;
                }
                if (r+1 == v.size()) {
                        val += v[at].first;
                        at = --l;
                        continue;
                }
                if (l == 0) {
                        val += v[at].second;
                        at = ++r;
                        continue;
                }
                if (v[at].first < v[at].second) {
                        val += v[at].first;
                        at = --l;
                } else {
                        val += v[at].second;
                        at = ++r;
                }
        }
        return {val, ret};
}

int main() { _
        int n; cin >> n;
        vector<pair<int, int>> v(n);
        for (auto& [a, b] : v) cin >> a >> b;
        int x = 0;
        for (int i = 0; i < n; i++) if (v[i].first < v[i].second) x++;
        vector<int> ans;
        ll val = LINF;
        for (int i = max(0, x - 1); i <= min(n-1, x); i++) {
                auto [val2, ans2] = solve(v, i);
                if (val2 < val) val = val2, ans = ans2;
        }
        for (int i = 0; i < n; i++) {
                if (i) cout << "␣";
                cout << ans[i]+1;
        }
        cout << endl;
        exit(0);
}
```

9.0.0.7 Tenet - Quantidade de SubPalíndromos Reordenados

Objetivo

Emanuel estava estudando Programação Competitiva, resolvendo problemas sobre palíndromos. Um palíndromo é uma string que permanece igual quando lida de trás para frente, como arara e tenet. Além disso, Emanuel sabe que uma substring de uma string T é uma string que pode ser obtida apagando-se zero ou mais caracteres do início e/ou fim de T. Por exemplo, ara, ar e arara são substrings de arara.

Ao se deparar com o problema clássico de computar quantas substrings de uma string S são palíndromos, Emanuel se perguntou como que esse problema poderia ser resolvido se pudessemos reordenar cada substring. Ou seja, dada uma string S, quantas substrings de S (contando repetições) podem ser reordenadas para formar um palíndromo.

Cansado após implementar o problema clássico, Emanuel pede sua ajuda para implementar a variação descrita acima para ele. Para simplificar o problema para você, ele garante que a string nunca conterá vogais (a, e, i, o, u, y).

Estratégia

Para resolver o problema, podemos usar a propriedade de que uma string pode ser reordenada para formar um palíndromo se, e somente se, a quantidade de cada letra na string for par, exceto por no máximo uma letra.

Utilizando essa propriedade, podemos usar uma tabela de frequência para contar a quantidade de cada letra em cada substring. Para cada substring, verificamos se a quantidade de cada letra é par, exceto por no máximo uma letra. Se isso for verdade, a substring pode ser reordenada para formar um palíndromo e incrementamos a resposta.

Código

```
1  const int MAX = (1<<20) + 10;
2
3  int qt[MAX];
4  int val[26];
5
6  int main() { _
7          int at = 0;
8          for (char c = 'a'; c <= 'z'; c++) {
9                  if (c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u' or c == 'y') conti
10                 val[c-'a'] = at++;
11         }
12
13         int n; cin >> n;
14         string s; cin >> s;
15         qt[0] = 1;
16         int mask = 0;
17         ll ans = 0;
18         for (int i = 0; i < n; i++) {
19                 mask ^= (1<<val[s[i]-'a']);
20                 ans += qt[mask];
21                 for (int j = 0; j < 20; j++) ans += qt[mask^(1<<j)];
22                 qt[mask]++;
23         }
24         cout << ans << endl;
25         return 0;
26 }
```

9.0.0.8 Trebado - Quantidade de movimentos

Objetivo

"O andar do bêbado" é um problema muito conhecido na matemática. Nesse problema temos um bêbado iniciando na origem (0, 0) do plano 2D e, a cada momento, ele escolhe aleatóriamente uma direção, norte (N), sul (S), leste (L) ou oeste (O), e anda um metro nesta direção. Apesar do andar caótico, nesta versão original do problema é matemáticamente garantido que depois de infinitos movimentos o bêbado sempre volta para a origem pelo menos uma vez!

Apresentamos aqui um problema ligeiramente diferente, o bêbado pode fazer os movimentos:

* Norte (N): anda um metro para cima. * Sul (S): anda um metro para baixo. * Leste (E): anda um metro para a direita. * Oeste (O): anda um metro para a esquerda. * Nordeste (NE): anda um metro para cima e um metro para a direita. * Sudeste (SE): anda um metro para baixo e um metro para a direita. * Noroeste (NO): anda um metro para cima e um metro para a esqueda. * Sudoeste (SO): anda um metro para baixo e um metro para a esquerda.

Dado que o bêbado inicia na origem (0, 0), imprima de quantas formas ele pode fazer exatamente K movimentos de tal forma que após esses K movimentos ele esteja novamente em sua casa, a origem. Como o resultado pode ser grande, imprima a resposta módulo P, sendo P um primo entre 10^8 e 10^9 dado na entrada.

Estratégia

O problema pode ser resolvido analisando a quantidade de movimentos para cada direção. Observe que o bêbado precisa ter o mesmo número de movimentos para cima e para baixo, e o mesmo número de movimentos para a direita e para a esquerda para voltar para a origem.

Podemos usar combinações para contar o número de maneiras de escolher os movimentos. Sejam n o número de movimentos para cima/baixo, l o número de movimentos para a direita/esquerda e k o número total de movimentos.

* O número de maneiras de escolher $2n$ movimentos para cima/baixo em k movimentos é dado por $\binom{k}{2n}$. * O número de maneiras de escolher n movimentos para cima em $2n$ movimentos para cima/baixo é dado por $\binom{2n}{n}$. * O número de maneiras de escolher $2l$ movimentos para a direita/esquerda em k movimentos é dado por $\binom{k}{2l}$. * O número de maneiras de escolher l movimentos para a direita em $2l$ movimentos para a direita/esquerda é dado por $\binom{2l}{l}$.

No entanto, ainda precisamos levar em conta os movimentos diagonais. Para cada movimento para cima/baixo, o bêbado pode escolher fazer um movimento diagonal para direita ou esquerda. Da mesma forma, para cada movimento para a direita/esquerda, ele pode escolher fazer um movimento diagonal para cima ou para baixo.

Para cada movimento para cima/baixo, o bêbado tem 2 opções (diagonal para direita ou esquerda). Para cada movimento para a direita/esquerda, o bêbado também tem 2 opções (diagonal para cima ou para baixo). Portanto, precisamos multiplicar a resposta por 2^{2n} e 2^{2l}, onde $2n$ é o número de movimentos para cima/baixo e $2l$ é o número de movimentos para a direita/esquerda.

A resposta final será a soma de todas as combinações possíveis, iterando sobre todos os valores possíveis de n e l.

Código

```
1  int MOD;
2
3  ll fat[MAX], ifat[MAX];
4
5  ll inv(ll a, ll b) {
6          return a > 1 ? b - inv(b\%a, a)*b/a : 1;
7  }
```

```
8
9  void build() {
10         fat[0] = ifat[0] = 1;
11         for (ll i = 1; i < MAX; i++) fat[i] = i*fat[i - 1]\%MOD;
12         for (ll i = 1; i < MAX; i++) ifat[i] = inv(fat[i], MOD);
13 }
14
15 ll choose(int n, int m) {
16         if (n < m or m < 0) return 0;
17         return fat[n]*ifat[m]\%MOD*ifat[n - m]\%MOD;
18 }
19
20 int main() { _
21
22         int k; cin >> k >> MOD;
23         build();
24
25         ll ans = 0;
26         for (int n = 0; n <= k; n++) {
27                 for (int l = 0; l <= k; l++) {
28                         ll cur = choose(k, 2*n)*choose(2*n, n)\%MOD;
29                         int over = 2*l - (k - 2*n);
30                         cur = cur*choose(2*n, over)\%MOD*choose(2*l, l)\%MOD;
31                         ans = (ans + cur)\%MOD;
32                 }
33         }
34
35         cout << ans << endl;
36
37         exit(0);
38 }
```