
Pedro Augusto  Ulisses Andrade  Adjailson Freire(POV)

Adoradores do Xandão

# Contents

# 1 Utils

## 1.1 Files

```bash
#!/bin/bash

for c in {a..f}; do
    cp temp.cpp "$c.cpp"
    echo "$c" > "$c.txt"
    if [ "$c" = "$letter" ]; then
        break
    fi
done
```

## 1.2 Limites

```
// LIMITES DE REPRESENTACAO DE DADOS

        tipo      | bits |          minimo .. maximo          | precisao decim.
------------------+------+------------------------------------+----------------
char              |    8 |                0 .. 127            |        2
signed char       |    8 |             -128 .. 127            |        2
unsigned char     |    8 |                0 .. 255            |        2
short             |   16 |          -32.768 .. 32.767         |        4
unsigned short    |   16 |                0 .. 65.535         |        4
int               |   32 |       -2 x 10^9 .. 2 x 10^9        |        9
unsigned int      |   32 |                0 .. 4 x 10^9       |        9
int64_t           |   64 |      -9 x 10^18 .. 9 x 10^18       |       18
uint64_t          |   64 |                0 .. 18 x 10^18     |       19
float             |   32 |    1.2 x 10^-38 .. 3.4 x 10^38     |      6-9
double            |   64 |   2.2 x 10^-308 .. 1.8 x 10^308    |     15-17
long double       |   80 | 3.4 x 10^-4932 .. 1.1 x 10^4932    |     18-19
BigInt/Dec(java)      1 x 10^-2147483648 .. 1 x 10^2147483647 | 0

// LIMITES DE MEMORIA

1MB = 1,048,576 bool
1MB =   524,288 char
1MB =   262,144 int32_t
1MB =   131,072 int64_t
1MB =    65,536 float
1MB =    32,768 double
1MB =    16,384 long double
1MB =    16,384 BigInteger / BigDecimal

// ESTOURAR TEMPO

imput size     | complexidade para 1 s
---------------+---------------------
 [10,11]       | O(n!), O(n^6)
 [17,19]       | O(2^n * n^2)
 [18, 22]      | O(2^n * n)
 [24,26]       | O(2^n)
 ... 100       | O(n^4)
 ... 450       | O(n^3)
 ... 1500      | O(n^2.5)
 ... 2500      | O(n^2 * log n)
 ... 10^4      | O(n^2)
 ... 2*10^5    | O(n^1.5)
 ... 4.5*10^6  | O(n log n)
 ... 10^7      | O(n log log n)
 ... 10^8      | O(n), O(log n), O(1)


// FATORIAL

12! =                479.001.600 [limite do (u)int]
20! = 2.432.902.008.176.640.000 [limite do (u)int64_t]
```

## 1.3 Makefile

```
CXX = g++
CPXXFLAGS = -fsanitize=address,undefined -fno-omit-frame-pointer -g -Wall
    -Wshadow -std=c++17 -Wno-unused-result -Wno-sign-compare -Wno-char-
    subscripts #-fuse-ld=gold

q:
    cp temp.cpp $(f).cpp
    touch $(f).txt
    code $(f).txt
    code $(f).cpp
    clear
compile:
    g++ -g $(f).cpp $(CXXFLAGS) -o $(f)
exe:
    ./$(f) < $(f).txt

runc: compile
runci: compile exe

clearexe:
    find . -maxdepth 1 -type f -executable -exec rm {} +
cleartxt:
    find . -type f -name "*.txt" -exec rm -f {} \;
clear: clearexe cleartxt
    clear
```

## 1.4 Template Cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

#define _ std::ios::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
#define all(a)        a.begin(), a.end()
#define int           long long int
#define double        long double
#define vi            vector<int>
#define pii           pair<int,int>
#define endl          "\n"
#define print_v(a)    for(auto x : a)cout<<x<<" ";cout<<endl
#define print_vp(a)   for(auto x : a)cout<<x.first<<" "<<x.second<< endl
#define f(i,s,e)      for(int i=s;i<e;i++)
```

```cpp
#define rf(i,e,s)    for(int i=e-1;i>=s;i--)
#define CEIL(a, b)   ((a) + (b - 1))/b
#define TRUNC(x, n)  floor(x * pow(10, n))/pow(10, n)
#define ROUND(x, n)  round(x * pow(10, n))/pow(10, n)
#define dbg(x) cout << #x << " = " << x << " ";
#define dbgl(x) cout << #x << " = " << x << endl;

const int INF = 1e9;    // 2^31-1
const int LLINF = 4e18; // 2^63-1
const double EPS = 1e-9;
const int MAX = 1e6+10; // 10^6 + 10

void solve() {


}

int32_t main() { _

    clock_t z = clock();
    int t = 1; // cin >> t;
    while (t--) {
        solve();
    }
    cerr << fixed << "Run Time : " << ((double)(clock() - z) /
    CLOCKS_PER_SEC) << endl;
    return 0;
}
```

## 1.5 Template Python

```python
import sys
import math
import bisect
from sys import stdin,stdout
from math import gcd,floor,sqrt,log
from collections import defaultdict as dd
from bisect import bisect_left as bl,bisect_right as br

sys.setrecursionlimit(100000000)

inp    =lambda: int(input())
strng  =lambda: input().strip()
jn     =lambda x,l: x.join(map(str,l))
strl   =lambda: list(input().strip())
mul    =lambda: map(int,input().strip().split())
mulf   =lambda: map(float,input().strip().split())
seq    =lambda: list(map(int,input().strip().split()))

ceil   =lambda x: int(x) if(x==int(x)) else int(x)+1
ceildiv=lambda x,d: x//d if(x%d==0) else x//d+1

flush  =lambda: stdout.flush()
stdstr =lambda: stdin.readline()
stdint =lambda: int(stdin.readline())
stdpr  =lambda x: stdout.write(str(x))

mod=1000000007

#main code

a = None
b = None
lista = None

def ident(*args):
    if len(args) == 1:
        return args[0]
    return args


def parsin(*, l=1, vpl=1, s=" "):
    if l == 1:
        if vpl == 1: return ident(input())
        else: return list(map(ident, input().split(s)))
    else:
        if vpl == 1: return [ident(input()) for _ in range(l)]
        else: return [list(map(ident, input().split(s))) for _ in range(l)
    ]


def solve():
    pass

# if __name__ == '__main__':
def main():
    st = clk()

    escolha = "in"
    #escolha = "num"

    match escolha:
        case "in":
            # êl infinitas linhas agrupadas de 2 em 2
            # pra infinitos valores em 1 linha pode armazenar em uma lista
            while True:
                global a, b
                try: a, b = input().split()
                except (EOFError): break #permite ler todas as linahs
    dentro do .txt
                except (ValueError): pass # consegue ler éat linhas em
    branco
                else:
                    a, b = int(a), int(b)
                solve()

        case "num":
            global lista
            # int l; cin >> l; while(l--){for(i=0; i<vpl; i++)}
            # retorna listas com inputs de cada linha
            # leia l linhas com vpl valores em cada uma delas
                # caseo seja mais de uma linha, retorna lista com listas
    de inputs
            lista = parsin(l=2, vpl=5)
```

```
            solve()

    sys.stderr.write(f"Run Time : {(clk() - st):.6f} seconds\n")

main()
```

# 2 Informações

## 2.1 Bitmask

```
int n = 11, ans = 0, k = 3;

// Operacoes com bits
ans = n & k; // AND bit a bit
ans = n | k; // OR bit a bit
ans = n ^ k; // XOR bit a bit
ans = ~n;    // NOT bit a bit

// Operacoes com 2^k em O(1)
ans = n << k; // ans = n * 2^k
ans = n >> k; // ans = n / 2^k

int j;

// Ativa j-esimo bit (0-based)
ans |= (1<<j);

// Desativa j-esimo bit (0-based)
ans &= (1<<j);

// Inverte j-esimo bit (0-based)
ans ^= (1<<j);

// checar se j-esimo bit esta ativo (0-based)
ans = n & (1<<j);

// Pegar valor do bit menos significativo | Retorna o maior divisor
ans = n & -n;

// Ligar todos on n bits
ans = (1<<n) - 1;

// Contar quantos 1's tem no binario de n
ans = __builtin_popcount(n);

// Contar quantos 0's tem no final do binario de n
ans = __builtin_ctz(n);
```

## 2.2 Priority Queue

```
// HEAP CRESCENTE {5,4,3,2,1}
priority_queue<int> pq; // max heap
    // maior elemento:
    pq.top();

// HEAP DECRESCENTE {1,2,3,4,5}
priority_queue<int, vector<int>, greater<int>> pq; // min heap
    // menor elemento:
    pq.top();

// REMOVER ELEMENTO
// Complexidade: O(n)
// Retorno: true se existe, false se ãno existe
pq.remove(x);

// INSERIR ELEMENTO
// Complexidade: O(log(n))
pq.push(x);

// REMOVER TOP
// Complexidade: O(log(n))
pq.pop();

// TAMANHO
// Complexidade: O(1)
pq.size();

// VAZIO
// Complexidade: O(1)
pq.empty();

// LIMPAR
// Complexidade: O(n)
pq.clear();

// ITERAR
// Complexidade: O(n)
for (auto x : pq) {}

// çãOrdenao por çãfuno customizada passada por parametro ao criar a pq
// Complexidade: O(n log(n))
auto cmp = [](int a, int b) { return a > b; };
priority_queue<int, vector<int>, decltype(cmp)> pq(cmp);
```

## 2.3 Set

```
set<int> st;

// Complexidade: O(log(n))
st.insert(x);
st.erase(x);
st.find(x);
st.erase(st.find(x));


// Complexidade: O(1)
st.size();
st.empty();

// Complexidade: O(n)
st.clear();
for (auto x : st) {}
```

```
              |  priority_queue |        set       |
--------------+-----------------+------------------+---------
op            | call    | compl | call    | compl  | melhor
--------------+---------+-------+---------+--------+---------
insert        | push    | log(n)| insert  | log(n) | pq
erase_menor   | pop     | log(n)| erase   | log(n) | pq
get_menor     | top     |    1  | begin   |   1    | set
get_maior     |   -     |    -  | rbegin  |   1    | set
erase_number  | remove  |    n  | erase   | log(n) | set
find_number   |   -     |    -  | find    | log(n) | set
find_>=       |   -     |    -  | lower   | log(n) | set
find_<=       |   -     |    -  | upper   | log(n) | set
iterate       | for     |    n  | for     |    n   | set
--------------+---------+-------+---------+--------+---------
```

## 2.4 Sort

```
vector<int> v;
    // Sort Crescente:
    sort(v.begin(), v.end());
    sort(all(v));

    // Sort Decrescente:
    sort(v.rbegin(), v.rend());
    sort(all(v), greater<int>());

    // Sort por uma çãfuno:
    auto cmp = [](int a, int b) { return a > b; }; // { 2, 3, 1 } -> { 3, 2, 1 }
    auto cmp = [](int a, int b) { return a < b; }; // { 2, 3, 1 } -> { 1, 2, 3 }
    sort(v.begin(), v.end(), cmp);
    sort(all(v), cmp);

    // Sort por uma çãfuno (çãcomparao de pares):
    auto cmp = [](pair<int, int> a, pair<int, int> b) { return a.second > b.second; };

    // Sort parcial:
    partial_sort(v.begin(), v.begin() + n, v.end()); // sorta com n menos elementos
    partial_sort(v.rbegin(), v.rbegin() + n, s.rend()) // sorta com n maiores elementos

    // SORT VS SET
    * para um input com elementos distintos, sort é mais árpido que set
```

## 2.5 String

```
// INICIALIZAR
string s; // string vazia
string s (n, c); // n ócpias de c
string s (s); // ócpia de s
string s (s, i, n); // ócpia de s[i..i+n-1]

// SUBSTRING
// Complexidade: O(n)
s.substr(i, n); // substring de s[i..i+n-1]
s.substr(i, j - i + 1); // substring de s[i..j]

// TAMANHO
// Complexidade: O(1)
s.size(); // tamanho da string
s.empty(); // true se vazia, false se ãno vazia

// MODIFICAR
// Complexidade: O(n)
s.push_back(c); // adiciona c no final
s.pop_back(); // remove o último
s += t; // concatena t no final
s.insert(i, t); // insere t a partir da çãposio i
s.erase(i, n); // remove n caracteres a partir da çãposio i
s.replace(i, n, t); // substitui n caracteres a partir da çãposio i por t
s.swap(t); // troca o úcontedo com t

// COMPARAR
// Complexidade: O(n)
s == t; // igualdade
s != t; // çdiferena
s < t; // menor que
s > t; // maior que
s <= t; // menor ou igual
s >= t; // maior ou igual

// BUSCA
// Complexidade: O(n)
s.find(t); // çãposio da primeira êocorrncia de t, ou string::npos se ãno existe
s.rfind(t); // çãposio da última êocorrncia de t, ou string::npos se ãno existe
s.find_first_of(t); // çãposio da primeira êocorrncia de um caractere de t, ou string::npos se ãno existe
s.find_last_of(t); // çãposio da última êocorrncia de um caractere de t, ou string::npos se ãno existe
s.find_first_not_of(t); // çãposio do primeiro caractere que ãno áest em t, ou string::npos se ãno existe
s.find_last_not_of(t); // çãposio do último caractere que ãno áest em t, ou string::npos se ãno existe

// SUBSTITUIR
// Complexidade: O(n)
s.replace(i, n, t); // substitui n caracteres a partir da çãposio i por t
s.replace(s.begin() + i, s.begin() + i + n, t.begin(), t.end()); // substitui n caracteres a partir da çãposio i por t
s.replace(s.begin() + i, s.begin() + i + n, t); // substitui n caracteres a partir da çãposio i por t
s.replace(s.begin() + i, s.begin() + i + n, n, c); // substitui n caracteres a partir da çãposio i por n ócpias de c
```

## 2.6 Vector

```
// INICIALIZAR
```

```
vector<int> v (n); // n ócpias de 0
vector<int> v (n, v); // n ócpias de v

// PUSH_BACK
// Complexidade: O(1) amortizado (O(n) se realocar)
v.push_back(x);

// REMOVER
// Complexidade: O(n)
v.erase(v.begin() + i);

// INSERIR
// Complexidade: O(n)
v.insert(v.begin() + i, x);

// ORDENAR
// Complexidade: O(n log(n))
sort(v.begin(), v.end());
sort(all(v));

// BUSCA BINARIA
// Complexidade: O(log(n))
// Retorno: true se existe, false se ãno existe
binary_search(v.begin(), v.end(), x);

// FIND
// Complexidade: O(n)
// Retorno: iterador para o elemento, v.end() se ãno existe
find(v.begin(), v.end(), x);

// CONTAR
// Complexidade: O(n)
// Retorno: únmero de êocorrncias
count(v.begin(), v.end(), x);
```

# 3 .vscode

# 4 Combinatoria

## 4.1 @ Factorial

```
// Calcula o fatorial de um únmero n
// Complexidade: O(n)

int factdp[20];

int fact(int n) {
    if (n < 2) return 1;
    if (factdp[n] != 0) return factdp[n];
    return factdp[n] = n * fact(n - 1);
}
```

## 4.2 @ Tabela

```
// Sequencia de p elementos de um total de n

ORDEM \ REPETIC |           COM             |          SEM
----------------+---------------------------+---------------------
IMPORTA         | ARRANJO COM REPETICAO     | ARRANJO SIMPLES
NAO             | COMBINACAO COM REPETICAO  | COMBINACAO SIMPLES
```

## 4.3 Arranjo Com Repeticao

```
int arranjoComRepeticao(int p, int n) {
    return pow(n, p);
}
```

## 4.4 Arranjo Simples

```
int arranjoSimples(int p, int n) {
    return fact(n) / fact(n - p);
}
```

## 4.5 Catalan

```
const int MAX_N = 100010;
const int p = 1e9+7; // p is a prime > MAX_N

int mod(int a, int m) {
  return ((a%m) + m) % m;
}

int inv(int a) {
    return modPow(a, p-2, p);
}

int modPow(int b, int p, int m) {
  if (p == 0) return 1;
  int ans = modPow(b, p/2, m);
  ans = mod(ans*ans, m);
  if (p&1) ans = mod(ans*b, m);
  return ans;
}

int Cat[MAX_N];

void solve() {
    Cat[0] = 1;
    for (int n = 0; n < MAX_N-1; ++n)              // O(MAX_N * log p)
        Cat[n+1] = ((4*n+2)%p * Cat[n]%p * inv(n+2)) % p;
    cout << Cat[100000] << "\n";                   // the answer is
    945729344
}
```

## 4.6 Combinacao Com Repeticao

```
int combinacaoComRepeticao(int p, int n) {
    return fact(n + p - 1) / (fact(p) * fact(n - 1));
}
```

## 4.7 Combinacao Simples

```
// Description: Calcula o valor de comb(n, k) % p, onde p é um primo > n.
// Complexidade: O(n)
const int MAX_N = 100010;
const int p = 1e9+7; // p is a prime > MAX_N

int mod(int a, int m) {
  return ((a%m) + m) % m;
}

int modPow(int b, int p, int m) {
  if (p == 0) return 1;
  int ans = modPow(b, p/2, m);
  ans = mod(ans*ans, m);
  if (p&1) ans = mod(ans*b, m);
  return ans;
}

int inv(int a) {
  return modPow(a, p-2, p);
}

int fact[MAX_N];

int comb(int n, int k) {
  if (n < k) return 0;
  return (((fact[n] * inv(fact[k])) % p) * inv(fact[n-k])) % p;
}

void solve() {
  fact[0] = 1;
  for (int i = 1; i < MAX_N; ++i)
  fact[i] = (fact[i-1]*i) % p;
  cout << comb(3, 3) << "\n";
}
```

## 4.8 Permutacao Circular

```
// Permutacao objetos em posicao simetrica em um circulo

int permutacaoCircular(int n) {
    return fact(n - 1);
}
```

## 4.9 Permutacao Com Repeticao

```
// Trocar elementos de lugar quando ha termos repetidos (ANAGRAMA)
int permutacaoComRepeticao(string s) {
    int n = s.size();
    int ans = fact(n);
    map<char, int> freq;
    for (char c : s) {
        freq[c]++;
    }
    for (auto [c, f] : freq) {
        ans /= fact(f);
    }
    return ans;
}
```

## 4.10 Permutacao Simples

```
// Agrupamentos distintos entre si pela ordem (FILA)
// Diferenca do arranjo: usa todos os elementos para o calculo
// SEM repeticao

int permutacaoSimples(int n) {
    return fact(n);
}
```

# 5 DP

## 5.1 Dp

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_gm = 30; // up to 20 garments at most and 20 models/garment
const int MAX_M = 210; // maximum budget is 200

int M, C, price[MAX_gm][MAX_gm]; // price[g (<= 20)][k (<= 20)]
int memo[MAX_gm][MAX_M];         // TOP-DOWN: dp table [g (< 20)][money (<= 200)]

int dp(int g, int money) {

    if (money < 0) return -1e9;
    if (g == C) return M - money;
    if (memo[g][money] != -1)
        return memo[g][money]; // avaliar linha g com dinheiro money (cada caso pensavel)
    int ans = -1;
    for (int k = 1; k <= price[g][0]; ++k)
        ans = max(ans, dp(g + 1, money - price[g][k]));
    return memo[g][money] = ans;
}

int main() {
    int TC;
    scanf("%d", &TC);
    while (TC--)
    {
        scanf("%d %d", &M, &C);
        for (int g = 0; g < C; ++g)
        {
            scanf("%d", &price[g][0]); // store k in price[g][0]
            for (int k = 1; k <= price[g][0]; ++k)
                scanf("%d", &price[g][k]);
        }
        memset(memo, -1, sizeof memo); // TOP-DOWN: init memo
```

```
        if (dp(0, M) < 0)
            printf("no solution\n"); // start the top-down DP
        else
            printf("%d\n", dp(0, M));
    }
    return 0;
}
```

## 5.2 Livro Caixa

```
// Description: Problema do livro caixa: retorna a expressao que resulta em f
// Complexidade: O(n*f)
// Explicacao: parecido com mochila => dados n numeros, dita se existe uma expressao que resulta em f, dado que cada numero pode ser positivo, negativo ou nao utilizado
/*
5 7
1 2 3 4 5
=> ?+??+
*/

int f=1, n=1, entrada[MAX];
map<pii, bool> memo;
bool positivo[MAX], negativo[MAX];

bool dp(int id, int soma) {

    if(id == n) return soma == f;
    if(memo.count({soma, id})) return memo[{soma,id}];

    bool pos = dp(id+1, soma+entrada[id]);
    bool neg = dp(id+1, soma-entrada[id]);

    if(pos and !neg) positivo[id] = true;
    else if(!pos and neg) negativo[id] = true;
    else if(pos and neg) positivo[id] = negativo[id] = true;
    return memo[{soma,id}] = (pos or neg);
}

void solve() {

    cin>> n >> f;
    memo.clear();

    f(i,0,n) {
        positivo[i] = negativo[i] = false;
        cin >> entrada[i];
    }

    bool ans = dp(0,0);

    if(!ans) cout << "*";
    else {
        f(i,0,n) {
            bool pos = positivo[i], neg = negativo[i];
            if(pos and neg) cout << "?";
            else if(pos) cout << "+";
            else cout << "-";
        }
    }
}
```

## 5.3 Mochila

```
// Description: Problema da mochila 0-1: retorna o valor maximo que pode ser carregado
// Complexidade: O(n*capacidade)

const int MAX_QNT_OBJETOS = 60; // 50 + 10
const int MAX_PESO_OBJETO = 1010; // 1000 + 10

int n, memo[MAX_QNT_OBJETOS][MAX_PESO_OBJETO];
vi valor, peso;

int mochila(int id, int remW) {
    if ((id == n) || (remW == 0)) return 0;
    int &ans = memo[id][remW];
    if (ans != -1) return ans;
    if (peso[id] > remW) return ans = mochila(id+1, remW);
    return ans = max(mochila(id+1, remW), valor[id]+mochila(id+1, remW-peso[id]));
}

void solve() {

    memset(memo, -1, sizeof memo);

    int capacidadeMochila; cin >> capacidadeMochila;

    f(i,0,capacidadeMochila) { memo[0][i] = 0; } // testar com e sem essa linha

    cin >> n;

    valor.assign(n, 0);
    peso.assign(n, 0);

    f(i,0,n) {
      cin >> peso[i] >> valor[i];
    }

    cout << mochila(0, capacidadeMochila) << endl;
}
```

## 5.4 Mochila Eduardo

```
// Description: çãImplementao da mochila com çãreconstruo de çãsoluo
// Complexidade: O(n*capacidade)

int t[100][100];// Defina os tamanhos confome seu problema, pode usar vector

```

```cpp
unordered_set<int> selecionados; // conjunto dos indices do itens que
    ãsero selecionados
int numItens;

// ps = pesos, vals = valores, fiz isso por legibilidade
int knapsack(int i, int cap, int ps[], int vals[]) {
    if(cap < 0) return -0x3f3f3f3f;
    if(i == numItens) return 0;
    if(t[i][cap] > -1) return t[i][cap];

    int x = knapsack(i + 1, cap, ps, vals);
    int y = knapsack(i + 1, cap - ps[i], ps, vals) + vals[i];
    return t[i][cap] = max(x, y);
}

// ps = pesos, vals = valores, fiz isso por legibilidade
void retrieve(int i, int cap, int ps[], int vals[]) {
    if(i == numItens) return;

    if(cap >= ps[i]) { // Dividi o if para legibilidade
        if(knapsack(i + 1, cap, ps, vals) < knapsack(i + 1, cap - ps[i],
    ps, vals) + vals[i]){
            selecionados.insert(i);
            return retrieve(i + 1, cap - ps[i], ps, vals);
        }
    }

    return retrieve(i + 1, cap, ps, vals);
}

int main() {
    memset(t, -1, sizeof t);

    int capacidade = 6;
    int pesos[] = {5, 4, 2}, valores[] = {500, 300, 250};
    numItens = 3;

    cout << knapsack(0, capacidade, pesos, valores) << endl;

    retrieve(0, 6, pesos, valores);
    for(auto i : selecionados) cout << i << ' ';
    cout << endl;
}
```

# 6 Estruturas

## 6.1 Bittree

```cpp
#include <bits/stdc++.h>
using namespace std;

/*       n --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1] --> Input array for which prefix sum is evaluated. */

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(vector<int>& BITree, int index) {
    int sum = 0;
    index = index + 1;
    while (index>0) {
        sum += BITree[index];
        index -= index & (-index);
    }
    return sum;
}

void updateBIT(vector<int>& BITree, int n, int index, int val) {
    index = index + 1;

    while (index <= n) {
        BITree[index] += val;
        index += index & (-index);
    }
}

vector<int> constructBITree(vector<int>& arr, int n) {
    vector<int> BITree(n+1, 0);

    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    return BITree;
}

void solve() {
    vector<int> freq = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = freq.size();
    vector<int> BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is"<< getSum(BITree, 5);
    // Let use test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); // BIT[4] = 6

    cout << "\nSum of elements in arr[0..5] after update is "
        << getSum(BITree, 5);
}

int main() {
    solve();
    return 0;
}
```

## 6.2 Fenwick Tree

```cpp
// BIT com update em range (Binary Indexed Tree)
//
// Operacoes 0-based
// query(l, r) retorna a soma de v[l..r]
// update(l, r, x) soma x em v[l..r]
//
// Complexidades:
```

```
// build - O(n)
// query - O(log(n))
// update - O(log(n))
namespace bit {
    int bit[2][MAX+2];
    int n;

    void build(int n2, vector<int>& v) {
        n = n2;
        for (int i = 1; i <= n; i++)
            bit[1][min(n+1, i+(i&-i))] += bit[1][i] += v[i];
    }
    int get(int x, int i) {
        int ret = 0;
        for (; i; i -= i&-i) ret += bit[x][i];
        return ret;
    }
    void add(int x, int i, int val) {
        for (; i <= n; i += i&-i) bit[x][i] += val;
    }
    int get2(int p) {
        return get(0, p) * p + get(1, p);
    }
    int query(int l, int r) { // zero-based
        return get2(r+1) - get2(l);
    }
    void update(int l, int r, int x) {
        add(0, l+1, x), add(0, r+2, -x);
        add(1, l+1, -x*l), add(1, r+2, x*(r+1));
    }
};

void solve() {

    vector<int> v {0,1,2,3,4,5}; // v[0] eh inutilizada
    bit::build(v.size(), v);

    int a = 0, b = 3;
    bit::query(a, b); // v[a] + v[a+1] + ... + v[b] = 6  | 1+2+3 = 6 |
    zero-based
    bit::update(a, b, 2); // v[a...b] += 2 | zero-based
}
```

## 6.3 Seg Tree

```
// SegTree
//
// Recursiva com Lazy Propagation
// Query: soma do range [a, b]
// Update: soma x em cada elemento do range [a, b]
// Pode usar a seguinte funcao para indexar os nohs:
// f(l, r) = (l+r)|(l!=r), usando 2N de memoria
//
// Complexidades:
// build - O(n)
// query - O(log(n))
// update - O(log(n))

#include<bits/stdc++.h>
using namespace std;

const int MAX = 1e5+10;

namespace SegTree {
    int seg[4*MAX], lazy[4*MAX];
    int n, *v;

    int op(int a, int b) { return a + b; }

    int build(int p=1, int l=0, int r=n-1) {
        lazy[p] = 0;
        if (l == r) return seg[p] = v[l];
        int m = (l+r)/2;
        return seg[p] = op(build(2*p, l, m), build(2*p+1, m+1, r));
    }
    void build(int n2, int* v2) {
        n = n2, v = v2;
        build();
    }
    void prop(int p, int l, int r) {
        seg[p] += lazy[p]*(r-l+1);
        if (l != r) lazy[2*p] += lazy[p], lazy[2*p+1] += lazy[p];
        lazy[p] = 0;
    }
    int query(int a, int b, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) return seg[p];
        if (b < l or r < a) return 0;
        int m = (l+r)/2;
        return op(query(a, b, 2*p, l, m), query(a, b, 2*p+1, m+1, r));
    }
    int update(int a, int b, int x, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (a <= l and r <= b) {
            lazy[p] += x;
            prop(p, l, r);
            return seg[p];
        }
        if (b < l or r < a) return seg[p];
        int m = (l+r)/2;
        return seg[p] = op(update(a, b, x, 2*p, l, m), update(a, b, x, 2*p
    +1, m+1, r));
    }

    // Se tiver uma seg de max, da pra descobrir em O(log(n))
    // o primeiro e ultimo elemento >= val numa range:

    // primeira posicao >= val em [a, b] (ou -1 se nao tem)
    int get_left(int a, int b, int val, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (b < l or r < a or seg[p] < val) return -1;
        if (r == l) return l;
        int m = (l+r)/2;
        int x = get_left(a, b, val, 2*p, l, m);
```

```
        if (x != -1) return x;
        return get_left(a, b, val, 2*p+1, m+1, r);
    }

    // ultima posicao >= val em [a, b] (ou -1 se nao tem)
    int get_right(int a, int b, int val, int p=1, int l=0, int r=n-1) {
        prop(p, l, r);
        if (b < l or r < a or seg[p] < val) return -1;
        if (r == l) return l;
        int m = (l+r)/2;
        int x = get_right(a, b, val, 2*p+1, m+1, r);
        if (x != -1) return x;
        return get_right(a, b, val, 2*p, l, m);
    }

    // Se tiver uma seg de soma sobre um array nao negativo v, da pra
    // descobrir em O(log(n)) o maior j tal que v[i]+v[i+1]+...+v[j-1] <
    val
    int lower_bound(int i, int& val, int p, int l, int r) {
        prop(p, l, r);
        if (r < i) return n;
        if (i <= l and seg[p] < val) {
            val -= seg[p];
            return n;
        }
        if (l == r) return l;
        int m = (l+r)/2;
        int x = lower_bound(i, val, 2*p, l, m);
        if (x != n) return x;
        return lower_bound(i, val, 2*p+1, m+1, r);
    }
};

void solve() {
    int n = 10;
    int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    SegTree::build(n, v);

    cout << SegTree::query(0, 9) << endl; // seg[0] + seg[1] + ... + seg
    [9] = 55
    SegTree::update(0, 9, 1); // seg[0,...,9] += 1
}
```

## 6.4 Sparse Table Disjunta

```
// Description: Sparse Table Disjunta para soma de intervalos
// Complexity Temporal: O(n log n) para construir e O(1) para consultar
// Complexidade Espacial: O(n log n)

#include <bits/stdc++.h>
using namespace std;

#define MAX 100010
#define MAX2 20 // log(MAX)

namespace SparseTable {
    int m[MAX2][2*MAX], n, v[2*MAX];
    int op(int a, int b) { return a + b; }
    void build(int n2, int* v2) {
        n = n2;
        for (int i = 0; i < n; i++) v[i] = v2[i];
        while (n&(n-1)) n++;
        for (int j = 0; (1<<j) < n; j++) {
            int len = 1<<j;
            for (int c = len; c < n; c += 2*len) {
                m[j][c] = v[c], m[j][c-1] = v[c-1];
                for (int i = c+1; i <  c+len; i++) m[j][i] = op(m[j][i-1],
     v[i]);
                for (int i = c-2; i >= c-len; i--) m[j][i] = op(v[i], m[j
    ][i+1]);
            }
        }
    }
    int query(int l, int r) {
        if (l == r) return v[l];
        int j = __builtin_clz(1) - __builtin_clz(l^r);
        return op(m[j][l], m[j][r]);
    }
}

void solve() {
    int n = 9;
    int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    SparseTable::build(n, v);
    cout << SparseTable::query(0, n-1) << endl; // sparse[0] + sparse[1] +
     ... + sparse[n-1] = 45
}
```

## 6.5 Tabuleiro

```
// Description: Estrutura que simula um tabuleiro M x N, sem realmente
    criar uma matriz
//              Permite atribuir valores a linhas e colunas, e consultar a
     çãposio mais frequente
// Complexidade Atribuir: O(log(N))
// Complexidade Consulta: O(log(N))
// Complexidade verificar frequencia geral: O(N * log(N))
#define MAX_VAL 5 // maior valor que pode ser adicionado na matriz + 1

class BinTree {
    protected:
        vector<int> mBin;
    public:
        explicit BinTree(int n) { mBin = vector(n + 1, 0); }

        void add(int p, const int val) {
            for (auto size = mBin.size(); p < size; p += p & -p)
                mBin[p] += val;
        }

        int query(int p) {
            int sumToP {0};
            for (; p > 0; p -= p & -p)
                sumToP += mBin[p];
```

```
            return sumToP;
        }
};

class ReverseBinTree : public BinTree {
    public:
        explicit ReverseBinTree(int n) : BinTree(n) {};

        void add(int p, const int val) {
            BinTree::add(static_cast<int>(mBin.size()) - p, val);
        }

        int query(int p) {
            return BinTree::query(static_cast<int>(mBin.size()) - p);
        }
};

class Tabuleiro {
    public:
        explicit Tabuleiro(const int m, const int n, const int q) : mM(m), mN(n), mQ(q) {
            mLinhas = vector<pair<int, int8_t>>(m, {0, 0});
            mColunas = vector<pair<int, int8_t>>(n, {0, 0});

            mAtribuicoesLinhas = vector(MAX_VAL, ReverseBinTree(mQ)); // aARvore[51]
            mAtribuicoesColunas = vector(MAX_VAL, ReverseBinTree(mQ));
        }

        void atribuirLinha(const int x, const int8_t r) {
            mAtribuirFileira(x, r, mLinhas, mAtribuicoesLinhas);
        }

        void atribuirColuna(const int x, const int8_t r) {
            mAtribuirFileira(x, r, mColunas, mAtribuicoesColunas);
        }

        int maxPosLinha(const int x) {
            return mMaxPosFileira(x, mLinhas, mAtribuicoesColunas, mM);
        }

        int maxPosColuna(const int x) {
            return mMaxPosFileira(x, mColunas, mAtribuicoesLinhas, mN);
        }

        vector<int> frequenciaElementos() {

            vector<int> frequenciaGlobal(MAX_VAL, 0);
            for(int i=0; i<mM; i++) {
                vector<int> curr = frequenciaElementos(i, mAtribuicoesColunas);
                for(int j=0; j<MAX_VAL; j++)
                    frequenciaGlobal[j] += curr[j];
            }
            return frequenciaGlobal;
        }

    private:
        int mM, mN, mQ, mMoment {0};

        vector<ReverseBinTree> mAtribuicoesLinhas, mAtribuicoesColunas;
        vector<pair<int, int8_t>> mLinhas, mColunas;

        void mAtribuirFileira(const int x, const int8_t r, vector<pair<int, int8_t>>& fileiras,
                              vector<ReverseBinTree>& atribuicoes) {
            if (auto& [oldQ, oldR] = fileiras[x]; oldQ)
                atribuicoes[oldR].add(oldQ, -1);

            const int currentMoment = ++mMoment;
            fileiras[x].first = currentMoment;
            fileiras[x].second = r;
            atribuicoes[r].add(currentMoment, 1);
        }

        int mMaxPosFileira(const int x, const vector<pair<int, int8_t>>& fileiras, vector<ReverseBinTree>& atribuicoesPerpendiculares, const int& currM) const {
            auto [momentoAtribuicaoFileira, rFileira] = fileiras[x];

            vector<int> fileiraFrequencia(MAX_VAL, 0);
            fileiraFrequencia[rFileira] = currM;

            for (int8_t r {0}; r < MAX_VAL; ++r) {
                const int frequenciaR = atribuicoesPerpendiculares[r].query(momentoAtribuicaoFileira + 1);
                fileiraFrequencia[rFileira] -= frequenciaR;
                fileiraFrequencia[r] += frequenciaR;
            }

            return MAX_VAL - 1 - (max_element(fileiraFrequencia.crbegin(), fileiraFrequencia.crend()) - fileiraFrequencia.crbegin());
        }

        vector<int> frequenciaElementos(int x, vector<ReverseBinTree>& atribuicoesPerpendiculares) const {

            vector<int> fileiraFrequencia(MAX_VAL, 0);

            auto [momentoAtribuicaoFileira, rFileira] = mLinhas[x];

            fileiraFrequencia[rFileira] = mN;

            for (int8_t r {0}; r < MAX_VAL; ++r) {
                const int frequenciaR = atribuicoesPerpendiculares[r].query(momentoAtribuicaoFileira + 1);
                fileiraFrequencia[rFileira] -= frequenciaR;
                fileiraFrequencia[r] += frequenciaR;
            }

            return fileiraFrequencia;
        }

};
```

```

void solve() {

    int L, C, q; cin >> L >> C >> q;

    Tabuleiro tabuleiro(L, C, q);

    int linha = 0, coluna = 0, valor = 10; // linha e coluna sao 0 based
    tabuleiro.atribuirLinha(linha, static_cast<int8_t>(valor)); // f(i,0,C) matriz[linha][i] = valor
    tabuleiro.atribuirColuna(coluna, static_cast<int8_t>(valor)); // f(i,0,L) matriz[i][coluna] = valor

    // Freuencia de todos os elementos, de 0 a MAX_VAL-1
    vector<int> frequenciaGeral = tabuleiro.frequenciaElementos();

    int a = tabuleiro.maxPosLinha(linha); // retorna a posicao do elemento mais frequente na linha
    int b = tabuleiro.maxPosColuna(coluna); // retorna a posicao do elemento mais frequente na coluna
}
```

## 6.6 Union Find

```
// Description: Union-Find (Disjoint Set Union)

typedef vector<int> vi;

struct UnionFind {
    vi p, rank, setSize;
    int numSets;
    UnionFind(int N) {
        p.assign(N, 0);
        for (int i = 0; i < N; ++i)
            p[i] = i;
        rank.assign(N, 0);
        setSize.assign(N, 1);
        numSets = N;
    }

    // Retorna o numero de sets disjuntos (separados)
    int numDisjointSets() { return numSets; }
    // Retorna o tamanho do set que contem o elemento i
    int sizeOfSet(int i) { return setSize[find(i)]; }

    int find(int i) { return (p[i] == i) ? i : (p[i] = find(p[i])); }
    bool same(int i, int j) { return find(i) == find(j); }
    void uni(int i, int j) {
        if (same(i, j))
            return;
        int x = find(i), y = find(j);
        if (rank[x] > rank[y])
            swap(x, y);
        p[x] = y;
        if (rank[x] == rank[y])
            ++rank[y];
        setSize[y] += setSize[x];
        --numSets;
    }
};

void solve() {
    int n; cin >> n;
    UnionFind UF(n);
    UF.uni(0, 1);
}
```

# 7 Geometria

## 7.1 3D - Distancia Entre 2 Poliedros

```
// Description: Calcula a menor distancia entre dois poliedros convexos
// Complexidade: O(n^2 * m^2), onde n e m sao o numero de vertices dos poliedros
// OBS: apenas testado para tetraedros

const double EPS = 1e-9;
const double INF = 1e50;

int cmpD(double a, double b = 0.0) { return a+EPS < b ? -1 : a-EPS > b; }

struct Point {
    double x, y, z;
    Point(double a=0.0,double b=0.0,double c=0.0){x=a,y=b,z=c;}
    Point operator+(const Point &P) const {return Point(x+P.x,y+P.y,z+P.z);}
    Point operator-(const Point &P) const {return Point(x-P.x,y-P.y,z-P.z);}
    Point operator*(double c) const {return Point(x*c,y*c,z*c);}
    Point operator/(double c) const {return Point(x/c,y/c,z/c);}
    double operator!() const {return sqrt(x*x+y*y+z*z);} // modulo
};

double produtoEscalar(Point A, Point B) { return A.x*B.x + A.y*B.y + A.z*B.z; }

Point produtoVetorial(Point A, Point B) { return Point(A.y*B.z-A.z*B.y, A.z*B.x-A.x*B.z, A.x*B.y-A.y*B.x); }

Point projWEmV(Point W, Point V) { return V * produtoEscalar(W,V) / produtoEscalar(V,V); }

// check if segments AB and CD have an intersection
bool checkIfSegmentsIntercept(Point A, Point B, Point C, Point D) {
    return cmpD(produtoEscalar(produtoVetorial(A-B,C-B),produtoVetorial(A-B,D-B))) <= 0 &&
    cmpD(produtoEscalar(produtoVetorial(C-D,A-D),produtoVetorial(C-D,B-D))) <= 0;
}

// distance between point P and segment AB
double dist_Point_seg(Point P, Point A, Point B) {
    Point PP = A + projWEmV(P-A,B-A);
```

```
    if (cmpD(!(A-PP)+!(PP-B),!(A-B)) == 0) return !(P-PP);//distance Point-line!
    return min(!(P-A),!(P-B));
}

// distance between segments AB and CD
double dist_seg_seg(Point A, Point B, Point C, Point D) {
    Point E = projWEmV(A-D,produtoVetorial(B-A,D-C));
    if (checkIfSegmentsIntercept(A,B,C+E,D+E)) return !E;
    return min( min( dist_Point_seg(A,C,D), dist_Point_seg(B,C,D) ),
    min( dist_Point_seg(C,A,B), dist_Point_seg(D,A,B) ) );
}

// distance between point P and triangle ABC
double dist_Point_tri(Point P, Point A, Point B, Point C) {
    Point N = produtoVetorial(A-C,B-C);
    Point PP = P + projWEmV(C-P,N);
    Point V1 = produtoVetorial(PP-A,B-A);
    Point V2 = produtoVetorial(PP-B,C-B);
    Point V3 = produtoVetorial(PP-C,A-C);
    if (cmpD(produtoEscalar(V1,V2)) >= 0 && cmpD(produtoEscalar(V1,V3)) >= 0 && cmpD(produtoEscalar(V2,V3)) >= 0)
    return !(PP-P); // distance Point-plane!
    return min(dist_Point_seg(P,A,B),min(dist_Point_seg(P,A,C),
    dist_Point_seg(P,B,C)));
}

// Calcula a menor distancia entre dois poliedros
// Complexidade: O(sz1^2 * sz2^2), onde sz1 e sz2 sao o numero de vertices dos poliedros
double distanciaPoliedroPoliedro(Point T1[], int sz1, Point T2[], int sz2) {

    double ans = INF;

    // itera por todos os pares de arestas dos dois tetraedros, e calcula a distancia entre os segmentos gerados por estes pares
    for (int i=0; i < sz1; i++) // arestas -> arestas
        for (int j=i+1; j < sz1; j++)
            for (int ii=0; ii < sz2; ii++)
                for (int jj=ii+1; jj < sz2; jj++)
                    ans = min( ans, dist_seg_seg(T1[i],T1[j],T2[ii],T2[jj]) );

    // itera por todos os pontos de um tetraedro e calcula a distancia entre estes pontos e as faces triangulares do outro tetraedro
    for (int i=0; i < sz1; i++)
        for (int j=i+1; j < sz1; j++)
            for (int k=j+1; k < sz1; k++)
                for (int x=0; x < sz2; x++)
                    ans = min( ans, dist_Point_tri(T1[x],T2[i],T2[j],T2[k]) ),
                    ans = min( ans, dist_Point_tri(T2[x],T1[i],T1[j],T1[k]) );
    return ans;
}

void solve() {
    Point poliedro1[5], poliedro2[5];

    f(i,0,4) { auto& [x,y,z] = poliedro1[i]; scanf("%lf %lf %lf", &x, &y, &z); }
    f(i,0,4) { auto& [x,y,z] = poliedro2[i]; scanf("%lf %lf %lf", &x, &y, &z); }

    printf("%.2lf\n",distanciaPoliedroPoliedro(poliedro1, 4, poliedro2, 4));
}
```

## 7.2 Andrew

```
// Nome: Convex Hull - Andrew's Monotone Chain
// Description: Calcula o perimetro do menor poligono convexo que contem todos os pontos
// Complexidade: O(n logn)

int produto_vetoril(pair<int,int> a,pair<int,int> b,pair<int,int> novo){
    return (b.first - a.first)*(novo.second-b.second) -(b.second - a.second)*(novo.first - b.first);
}
double distancia(pair<int,int> a,pair<int,int> b){
    return sqrt(pow((a.first - b.first), 2) + pow((a.second - b.second), 2));
}

double andrew(pair<int,int> pontos[], int n) {

    vector<pair<int,int>> hull;

    pair<int,int> ponto;
    int k=0;
    f(i,0,n) {

        while(k>=2 and produto_vetoril(hull[k-2],hull[k-1],pontos[i]) <= 0) {
            hull.pop_back();
            k--;
        }
        hull.push_back(pontos[i]);
        k++;
    }

    for(int i=n-1, tam = k+1 ; i>=0 ; i--) {
        while(k>=tam && produto_vetoril(hull[k-2],hull[k-1],pontos[i])<=0)
    {
            hull.pop_back();
            k--;
        }
        hull.push_back(pontos[i]);
        k++;
    }

    double perimetro = 0;

```

```
    f(i,1,hull.size()) {
        perimetro += distancia(hull[i-1],hull[i]);
    }

    return perimetro;
}

void solve() {

    int n; scanf("%lld",&n);
    pair<int,int> pontos[n];

    for(auto& [x, y] : pontos)
        scanf("%lld %lld",&x,&y);

    sort(pontos,pontos+n);

    double perimetro = andrew(pontos,n);
}
```

## 7.3 Circulo

```
#include <bits/stdc++.h>
#include "ponto.cpp"
using namespace std;

// Retorna se o ponto p esta dentro, fora ou na circunferencia de centro c e raio r
int insideCircle(const point_i &p, const point_i &c, int r) {
    int dx = p.x-c.x, dy = p.y-c.y;
    int Euc = dx*dx + dy*dy, rSq = r*r; // all integer
    return Euc < rSq ? 1 : (Euc == rSq ? 0 : -1); // in/border/out
}

// Determina o centro e raio de um circulo que passa por 3 pontos
bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x-p2.x) * (p1.x-p2.x) +
              (p1.y-p2.y) * (p1.y-p2.y);
  double det = r*r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x+p2.x) * 0.5 + (p1.y-p2.y) * h;
  c.y = (p1.y+p2.y) * 0.5 + (p2.x-p1.x) * h;
  return true;
}
```

## 7.4 Closestpair Otimizado

```

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};
// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a,  *p2 = (Point *)b;
    return (p1->x != p2->x) ? (p1->x - p2->x) : (p1->y - p2->y);
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a,   *p2 = (Point *)b;
    return (p1->y != p2->y) ? (p1->y - p2->y) : (p1->x - p2->x);
}
// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}
// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n){
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}
// A utility function to find a minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}
// A utility function to find the distance between the closest points of
// strip of a given size. All points in strip[] are sorted according to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d){
    float min = d;  // Initialize the minimum distance as d
    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}
// A recursive function to find the smallest distance. The array Px contains
// all points sorted according to x coordinates and Py contains all points
// sorted according to y coordinates
float closestUtil(Point Px[], Point Py[], int n){
    // If there are 2 or 3 points, then use brute force
```

```
    if (n <= 3)
        return bruteForce(Px, n);
    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];
    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point Pyl[mid];   // y sorted points on left of vertical line
    Point Pyr[n-mid];  // y sorted points on right of vertical line
    int li = 0, ri = 0;  // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
      if ((Py[i].x < midPoint.x || (Py[i].x == midPoint.x && Py[i].y < midPoint.y)) && li<mid)
          Pyl[li++] = Py[i];
      else
          Pyr[ri++] = Py[i];
    }
    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(Px, Pyl, mid);
    float dr = closestUtil(Px + mid, Pyr, n-mid);
    // Find the smaller of two distances
    float d = min(dl, dr);
    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j] = Py[i], j++;
    // Find the closest points in strip.  Return the minimum of d and closest
    // distance is strip[]
    return stripClosest(strip, j, d);
}
// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n){
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }
    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);
    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(Px, Py, n);
}

int main(){
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

## 7.5 Geometricosgerai

```
#include <bits/stdc++.h>
using namespace std;

struct Point {
    double x, y;
};
//checa se dado ponto esta dentro de um poligno.
//tempo O(NxM) sendo N=numero de pontos do poligno, M= a quantia de pontos que vc quer checar
bool point_in_polygon(Point point, vector<Point> polygon){
    int num_vertices = polygon.size();
    double x = point.x, y = point.y;
    bool inside = false;
    Point p1 = polygon[0], p2;
    for (int i = 1; i <= num_vertices; i++) {
        p2 = polygon[i % num_vertices];
        if (y > min(p1.y, p2.y)) {
            if (y <= max(p1.y, p2.y)) {
                if (x <= max(p1.x, p2.x)) {
                    double x_intersection
                        = (y - p1.y) * (p2.x - p1.x)
                              / (p2.y - p1.y)
                          + p1.x;
                    if (p1.x == p2.x
                        || x <= x_intersection) {
                        inside = !inside;
                    }
                }
            }
        }
        p1 = p2;
    }
    return inside;
}
//dado N pontos ordenados, encontre a area do poligno
double polygonArea(vector<pair<double,double>> vec )
{
    // Initialize area
    double area = 0.0;
    // Calculate value of shoelace formula
    int j = vec.size() - 1;
    for (int i = 0; i < vec.size(); i++)
    {
        area += (vec[j].first + vec[i].first) * (vec[j].second - vec[i].second);
        j = i;  // j is previous vertex to i
    }
    // Return absolute value
    return abs(area / 2.0);
}

//encontrar area de intersecao entre dois circulos
```

```
//(x,y)posicao do centro + raio
long long int intersectionArea(long double X1, long double Y1,
                               long double R1, long double X2,
                               long double Y2, long double R2){
    long double Pi = 3.14;
    long double d, alpha, beta, a1, a2;
    long long int ans;

    // Calculate the euclidean distance
    // between the two points
    d = sqrt((X2 - X1) * (X2 - X1) + (Y2 - Y1) * (Y2 - Y1));

    if (d > R1 + R2)
        ans = 0;

    else if (d <= (R1 - R2) && R1 >= R2)
        ans = floor(Pi * R2 * R2);

    else if (d <= (R2 - R1) && R2 >= R1)
        ans = floor(Pi * R1 * R1);

    else {
        alpha = acos((R1 * R1 + d * d - R2 * R2)
                     / (2 * R1 * d))
                * 2;
        beta = acos((R2 * R2 + d * d - R1 * R1)
                    / (2 * R2 * d))
               * 2;
        a1 = 0.5 * beta * R2 * R2
             - 0.5 * R2 * R2 * sin(beta);
        a2 = 0.5 * alpha * R1 * R1
             - 0.5 * R1 * R1 * sin(alpha);
        ans = floor(a1 + a2);
    }

    return ans;
}
```

## 7.6 Leis

```
// Lei dos Cossenos: a^2 = b^2 + c^2 - 2bc*cos(A)
// Lei dos Senos: a/sen(A) = b/sen(B) = c/sen(C) = 2R
// Pitagoras: a^2 = b^2 + c^2
```

## 7.7 Linha

```
#include <bits/stdc++.h>
#include "ponto.cpp"
using namespace std;

// const int EPS = 1e-9;

struct line { double a, b, c; }; // ax + by + c = 0

// Gera a equacao da reta que passa por 2 pontos
void pointsToLine(point p1, point p2, line &l) {
    if (fabs(p1.x-p2.x) < EPS)
        l = {1.0, 0.0, -p1.x};
    else {
        double a = -(double)(p1.y-p2.y) / (p1.x-p2.x);
        l = {a, 1.0, -(double)(a*p1.x) - p1.y};
    }
}

// Gera a equacao da reta que passa por um ponto e tem inclinacao m
void pointSlopeToLine(point p, double m, line &l) { // m < Inf
    l = {m, 1.0, -((m * p.x) + p.y)};
}

// Checa se 2 retas sao paralelas
bool areParallel(line l1, line l2) {
    return (fabs(l1.a-l2.a) < EPS) and (fabs(l1.b-l2.b) < EPS);
}

// Checa se 2 retas sao iguais
bool areSame(line l1, line l2) {
    return areParallel(l1, l2) and (fabs(l1.c-l2.c) < EPS);
}

// Retorna se 2 retas se intersectam e o ponto de interseccao (referencia)
bool areIntersect(line l1, line l2, point &p) {
    if (areParallel(l1, l2)) return false;

    p.x = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);
    if (fabs(l1.b) > EPS) p.y = -(l1.a*p.x + l1.c);
    else                  p.y = -(l2.a*p.x + l2.c);
    return true;
}
```

## 7.8 Maior Poligono Convexo

```
#include <bits/stdc++.h>
using namespace std;

const double EPS = 1e-9;

double DEG_to_RAD(double d) { return d*M_PI / 180.0; }

double RAD_to_DEG(double r) { return r*180.0 / M_PI; }

struct point {
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
    bool operator == (point other) const {
        return (fabs(x-other.x) < EPS && (fabs(y-other.y) < EPS));
    }

    bool operator <(const point &p) const {
        return x < p.x || (abs(x-p.x) < EPS && y < p.y);
    }
};

```

```
struct vec {
    double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVec(point a, point b) { return vec(b.x-a.x, b.y-a.y); }

double dist(point p1, point p2) { return hypot(p1.x-p2.x, p1.y-p2.y); }

// returns the perimeter of polygon P, which is the sum of Euclidian distances of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size()-1; ++i)
        ans += dist(P[i], P[i+1]);
    return ans;
}

// returns the area of polygon P
double area(const vector<point> &P) {
    double ans = 0.0;
    for (int i = 0; i < (int)P.size()-1; ++i)
        ans += (P[i].x*P[i+1].y - P[i+1].x*P[i].y);
    return fabs(ans)/2.0;
}

double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }
double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }

// returns angle aob in rad
double angle(point a, point o, point b) {
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }

// returns the area of polygon P, which is half the cross products of vectors defined by edge endpoints
double area_alternative(const vector<point> &P) {
    double ans = 0.0; point O(0.0, 0.0);
    for (int i = 0; i < (int)P.size()-1; ++i)
        ans += cross(toVec(O, P[i]), toVec(O, P[i+1]));
    return fabs(ans)/2.0;
}

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) { return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) { return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn
// while examining all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int n = (int)P.size();
  // a point/sz=2 or a line/sz=3 is not convex
  if (n <= 3) return false;
  bool firstTurn = ccw(P[0], P[1], P[2]);         // remember one result,
  for (int i = 1; i < n-1; ++i)                   // compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]) != firstTurn)
      return false;                               // different -> concave
  return true;                                    // otherwise -> convex
}

// returns 1/0/-1 if point p is inside/on (vertex/edge)/outside of
// either convex/concave polygon P
int insidePolygon(point pt, const vector<point> &P) {
  int n = (int)P.size();
  if (n <= 3) return -1;                          // avoid point or line
  bool on_polygon = false;
  for (int i = 0; i < n-1; ++i)                   // on vertex/edge?
    if (fabs(dist(P[i], pt) + dist(pt, P[i+1]) - dist(P[i], P[i+1])) < EPS)
      on_polygon = true;
  if (on_polygon) return 0;                       // pt is on polygon
  double sum = 0.0;                               // first = last point
  for (int i = 0; i < n-1; ++i) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]);             // left turn/ccw
    else
      sum -= angle(P[i], pt, P[i+1]);             // right turn/cw
  }
  return fabs(sum) > M_PI ? 1 : -1;               // 360d->in, 0d->out
}

// compute the intersection point between line segment p-q and line A-B
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y-A.y, b = A.x-B.x, c = B.x*A.y - A.x*B.y;
  double u = fabs(a*p.x + b*p.y + c);
  double v = fabs(a*q.x + b*q.y + c);
  return point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

// cuts polygon Q along the line formed by point A->point B (order matters)
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point A, point B, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); ++i) {
    double left1 = cross(toVec(A, B), toVec(A, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(A, B), toVec(A, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);          // Q[i] is on the left
    if (left1*left2 < -EPS)                       // crosses line AB
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], A, B));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());                       // wrap around
  return P;
}
```

```

vector<point> CH_Graham(vector<point> &Pts) {    // overall O(n log n)
  vector<point> P(Pts);                          // copy all points
  int n = (int)P.size();
  if (n <= 3) {                                  // point/line/triangle
    if (!(P[0] == P[n-1])) P.push_back(P[0]);    // corner case
    return P;                                    // the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = min_element(P.begin(), P.end())-P.begin();
  swap(P[0], P[P0]);                             // swap P[P0] with P[0]

  // second, sort points by angle around P0, O(n log n) for this sort
  sort(++P.begin(), P.end(), [&](point a, point b) {
    return ccw(P[0], a, b);                      // use P[0] as the pivot
  });

  // third, the ccw tests, although complex, it is just O(n)
  vector<point> S({P[n-1], P[0], P[1]});         // initial S
  int i = 2;                                     // then, we check the
    rest
  while (i < n) {                                // n > 3, O(n)
    int j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i]))                 // CCW turn
      S.push_back(P[i++]);                       // accept this point
    else                                         // CW turn
      S.pop_back();                              // pop until a CCW turn
  }
  return S;                                      // return the result
}

vector<point> CH_Andrew(vector<point> &Pts) {    // overall O(n log n)
  int n = Pts.size(), k = 0;
  vector<point> H(2*n);
  sort(Pts.begin(), Pts.end());                  // sort the points by x/y
  for (int i = 0; i < n; ++i) {                  // build lower hull
    while ((k >= 2) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  for (int i = n-2, t = k+1; i >= 0; --i) {       // build upper hull
    while ((k >= t) && !ccw(H[k-2], H[k-1], Pts[i])) --k;
    H[k++] = Pts[i];
  }
  H.resize(k);
  return H;
}

int main() {
  // 6(+1) points, entered in counter clockwise order, 0-based indexing
  vector<point> P;
  P.emplace_back(1, 1);                          // P0
  P.emplace_back(3, 3);                          // P1
  P.emplace_back(9, 1);                          // P2
  P.emplace_back(12, 4);                         // P3
  P.emplace_back(9, 7);                          // P4
  P.emplace_back(1, 7);                          // P5
  P.push_back(P[0]);                             // loop back, P6 = P0

  printf("Perimeter = %.2lf\n", perimeter(P));   // 31.64
  printf("Area = %.2lf\n", area(P));             // 49.00
  printf("Area = %.2lf\n", area_alternative(P)); // also 49.00
  printf("Is convex = %d\n", isConvex(P));       // 0 (false)

  point p_out(3, 2); // outside this (concave) polygon
  printf("P_out is inside = %d\n", insidePolygon(p_out, P)); // -1
  printf("P1 is inside = %d\n", insidePolygon(P[1], P)); // 0
  point p_on(5, 7); // on this (concave) polygon
  printf("P_on is inside = %d\n", insidePolygon(p_on, P)); // 0
  point p_in(3, 4); // inside this (concave) polygon
  printf("P_in is inside = %d\n", insidePolygon(p_in, P)); // 1

  P = cutPolygon(P[2], P[4], P);
  printf("Perimeter = %.2lf\n", perimeter(P));   // smaller now, 29.15
  printf("Area = %.2lf\n", area(P));             // 40.00

  P = CH_Graham(P);                              // now this is a
    rectangle
  printf("Perimeter = %.2lf\n", perimeter(P));   // precisely 28.00
  printf("Area = %.2lf\n", area(P));             // precisely 48.00
  printf("Is convex = %d\n", isConvex(P));       // true
  printf("P_out is inside = %d\n", insidePolygon(p_out, P)); // 1
  printf("P_in is inside = %d\n", insidePolygon(p_in, P)); // 1

  return 0;
}
```

## 7.9 Minkowski Sum

```
// Nome: Minkowski Sum
// Complexidade: O(n + m)

struct pt{
    long long x, y;
    pt operator + (const pt & p) const {
        return pt{x + p.x, y + p.y};
    }
    pt operator - (const pt & p) const {
        return pt{x - p.x, y - p.y};
    }
    long long cross(const pt & p) const {
        return x * p.y - y * p.x;
    }
};

void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y == P[pos].y && P[i].x < P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
}

```

```cpp
vector<pt> minkowski(vector<pt> P, vector<pt> Q){
    // the first vertex must be the lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while(i < P.size() - 2 || j < Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] - P[i]).cross(Q[j + 1] - Q[j]);
        if(cross >= 0 && i < P.size() - 2)
            ++i;
        if(cross <= 0 && j < Q.size() - 2)
            ++j;
    }
    return result;
}
```

## 7.10 Ponto

```cpp
#include <bits/stdc++.h>
using namespace std;
const int EPS = 1e-9;
// Ponto 2D
struct point_i {
    int x, y;
    point_i() { x = y = 0; }
    point_i(int _x, int _y) : x(_x), y(_y) {}
};

// Ponto 2D com precisao
struct point {
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}

    bool operator < (point other) const {
        if (fabs(x-other.x) > EPS)
            return x < other.x;
        return y < other.y;
    }

    bool operator == (const point &other) const {
        return (fabs(x-other.x) < EPS) and (fabs(y-other.y) < EPS);
    }
};

// Distancia entre 2 pontos
double dist(const point &p1, const point &p2) {
    return hypot(p1.x-p2.x, p1.y-p2.y);
}

double DEG_to_RAD(double d) { return d*M_PI / 180.0; }
double RAD_to_DEG(double r) { return r*180.0 / M_PI; }

// Rotaciona o ponto p em theta graus em sentido anti-horario em relacao a origem (0, 0)
point rotate(const point &p, double theta) {
    double rad = DEG_to_RAD(theta);
    return point(p.x*cos(rad) - p.y*sin(rad),
                 p.x*sin(rad) + p.y*cos(rad));
}
```

## 7.11 Triangulos

```cpp
#include <bits/stdc++.h>
#include "vetor.cpp"
#include "linha.cpp"

using namespace std;

// Condicao Existencia
bool existeTriangulo(double a, double b, double c) {
  return (a+b > c) && (a+c > b) && (b+c > a);
}

// Area de um triangulo de lados a, b e c
int area(int a, int b, int c) {
    if (!existeTriangulo(a, b, c)) return 0;
    double s = (a+b+c)/2.0;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca;
}

double perimeter(point a, point b, point c) {
    return dist(a, b) + dist(b, c) + dist(c, a);
}

// ====== CIRCULO INSCRITO ======

// Retorna raio de um circulo inscrito em um triangulo de lados a, b e c
double rInCircle(double ab, double bc, double ca) {
    return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca));
}
double rInCircle(point a, point b, point c) {
    return rInCircle(dist(a, b), dist(b, c), dist(c, a));
}

// Calcula o centro e o raio do circulo inscrito em um triangulo dados seus pontos
bool inCircle(point p1, point p2, point p3, point &ctr, double &r) {
    r = rInCircle(p1, p2, p3);
    if (fabs(r) < EPS) return false;
    line l1, l2;
    double ratio = dist(p1, p2) / dist(p1, p3);
    point p = translate(p2, scale(toVec(p2, p3), ratio / (1+ratio)));
```

```
    pointsToLine(p1, p, l1);
    ratio = dist(p2, p1) / dist(p2, p3);
    p = translate(p1, scale(toVec(p1, p3), ratio / (1+ratio)));
    pointsToLine(p2, p, l2);
    areIntersect(l1, l2, ctr);
    return true;
}

// ====== CIRCULO CIRCUNSCRITO ======

double rCircumCircle(double ab, double bc, double ca) {
    return ab * bc * ca / (4.0 * area(ab, bc, ca));
}
double rCircumCircle(point a, point b, point c) {
    return rCircumCircle(dist(a, b), dist(b, c), dist(c, a));
}
```

## 7.12 Vetor

```
#include <bits/stdc++.h>
#include "ponto.cpp"
using namespace std;

struct vec {
    double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {}
};

    double dot(vec a, vec b) { return (a.x*b.x + a.y*b.y); }
    double norm_sq(vec v) { return v.x*v.x + v.y*v.y; }
    double cross(vec a, vec b) { return a.x*b.y - a.y*b.x; }

// Converte 2 pontos em um vetor
vec toVec(const point &a, const point &b) {
    return vec(b.x-a.x, b.y-a.y);
}

// Soma 2 vetores
vec scale(const vec &v, double s) {
    return vec(v.x*s, v.y*s);
}
// Resultado do ponto p + vetor v
point translate(const point &p, const vec &v) {
    return point(p.x+v.x, p.y+v.y);
}

// Angulo entre 2 vetores (produto escalar) em radianos
double angle(const point &a, const point &o, const point &b) {
    vec oa = toVec(o, a), ob = toVec(o, b);
    return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}

// Retorna se o ponto r esta a esquerda da linha pq (counter-clockwise)
bool ccw(point p, point q, point r) {
return cross(toVec(p, q), toVec(p, r)) > EPS;
}

// Retorna se sao colineares
bool collinear(point p, point q, point r) {
    return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}

// Distancia ponto-linha
double distToLine(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    c = translate(a, scale(ab, u));
    return dist(p, c);
}

// Distancia ponto p - segmento ab
double distToLineSegment(point p, point a, point b, point &c) {
    vec ap = toVec(a, p), ab = toVec(a, b);
    double u = dot(ap, ab) / norm_sq(ab);
    if (u < 0.0) { // closer to a
        c = point(a.x, a.y);
        return dist(p, a); // dist p to a
    }
    if (u > 1.0) { // closer to b
        c = point(b.x, b.y);
        return dist(p, b); // dist p to b
    }
    return distToLine(p, a, b, c); // use distToLine
}
```

# 8 Grafos

## 8.1 Bfs - Matriz

```
// Description: BFS para uma matriz (n x m)
// Complexidade: O(n * m)

vector<vi> mat;
vector<vector<bool>> vis;
vector<pair<int,int>> mov = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
int l, c;

bool valid(int x, int y) {
    return (0 <= x and x < l and 0 <= y and y < c and !vis[x][y] /*and mat
    [x][y]*/);
}

void bfs(int i, int j) {

    queue<pair<int,int>> q; q.push({i, j});

    while(!q.empty()) {

        auto [u, v] = q.front(); q.pop();
        vis[u][v] = true;

        for(auto [x, y]: mov) {
            if(valid(u+x, v+y)) {
```

```
                q.push({u+x,v+y});
                vis[u+x][v+y] = true;
            }
        }
    }
}

void solve() {
    cin >> l >> c;
    mat.resize(l, vi(c));
    vis.resize(l, vector<bool>(c, false));
    /*preenche matriz*/
    bfs(0,0);
}
```

## 8.2 Bfs - Por Niveis

```
// Description: Encontrar distancia entre S e outros pontos em que pontos estao agrupados (terminais)
// EXTRA: BFS diferenciado para armazenar distancias sem VIS

int n;
vi dist;
vector<vi> niveisDoNode, itensDoNivel;

void bfs(int s) {

    queue<pair<int, int>> q; q.push({s, 0});

    while (!q.empty()) {
        auto [v, dis] = q.front(); q.pop();

        for(auto nivel : niveisDoNode[v]) {
            for(auto u : itensDoNivel[nivel]) {
                if (dist[u] == 0) {
                    q.push({u, dis+1});
                    dist[u] = dis + 1;
                }
            }
        }
    }
}

void solve() {

    int n, ed; cin >> n >> ed;
    dist.clear(), itensDoNivel.clear(), niveisDoNode.clear();
    itensDoNivel.resize(n);

    f(i,0,ed) {
        int q; cin >> q;
        while(q--) {
            int v; cin >> v;
            niveisDoNode[v].push_back(i);
            itensDoNivel[i].push_back(v);
        }
    }

    bfs(0);
}
```

## 8.3 Bfs - String

```
// Description: BFS para listas de adjacencia por nivel
// Complexidade: O(V + E)

int n;
unordered_map<string, int> dist;
unordered_map<string, vector<int>> niveisDoNode;
vector<vector<string>> itensDoNivel;

void bfs(string s) {

    queue<pair<string, int>> q; q.push({s, 0});

    while (!q.empty()) {
        auto [v, dis] = q.front(); q.pop();

        for(auto linha : niveisDoNode[v]) {
            for(auto u : itensDoNivel[linha]) {
                if (dist[u] == 0) {
                    q.push({u, dis+1});
                    dist[u] = dis + 1;
                }
            }
        }
    }
}

void solve() {

    int n, ed; cin >> n >> ed;
    dist.clear(), itensDoNivel.clear(), niveisDoNode.clear();
    itensDoNivel.resize(n);

    f(i,0,ed) {
        int q; cin >> q;
        while(q--) {
            string str; cin >> str;
            niveisDoNode[str].push_back(i);
            itensDoNivel[i].push_back(str);
        }
    }

    string src; cin >> src;
    bfs(src);
}
```

## 8.4 Bfs - Tradicional

```
// BFS com informacoes adicionais sobre a distancia e o pai de cada vertice
// Complexidade: O(V + E), onde V eh o numero de vertices e E o numero de areqas
```

```

int n;
vector<bool> vis;
vector<int> d, p;
vector<vector<int>> adj;

void bfs(int s) {

    queue<int> q; q.push(s);
    vis[s] = true, d[s] = 0, p[s] = -1;

    while (!q.empty()) {
        int v = q.front(); q.pop();
        vis[v] = true;

        for (int u : adj[v]) {
            if (!vis[u]) {
                vis[u] = true;
                q.push(u);
                // d[u] = d[v] + 1;
                // p[u] = v;
            }
        }
    }
}

void solve() {
    cin >> n;
    adj.resize(n); d.resize(n, -1);
    vis.resize(n); p.resize(n, -1);

    for (int i = 0; i < n; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    bfs(0);
}

// OBS: Pode ser usado para encontrar o menor caminho entre dois vertices em um grafo sem pesos
```

## 8.5 Dfs

```
vector<int> adj[MAXN], parent;
int visited[MAXN];

// DFS com informacoes adicionais sobre o pai de cada vertice
// Complexidade: O(V + E), onde V eh o numero de vertices e E o numero de areqas
void dfs(int p) {
    memset(visited, 0, sizeof visited);
    stack<int> st;
    st.push(p);

    while (!st.empty()) {
        int v = st.top(); st.pop();

        if (visited[v]) continue;
        visited[v] = true;

        for (int u : adj[v]) {
            if (!visited[u]) {
                parent[u] = v;
                st.push(u);
            }
        }
    }
}

// DFS com informacoes adicionais sobre o pai de cada vertice
// Complexidade: O(V + E), onde V eh o numero de vertices e E o numero de areqas
void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) {
            parent[u] = v;
            dfs(u);
        }
    }
}

void solve() {
    int n; cin >> n;
    for (int i = 0; i < n; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs(0);
}
```

## 8.6 Articulation

```
// Description: encontra os pontos de çãarticulao de um grafo
// Complexidade: O(V+E)

const int MAX = 410;

vector  <int> adj[MAX];

void APUtil(int u, bool visited[], int disc[], int low[], int& time, int parent, bool isAP[]) {
    int children = 0;

    visited[u] = true;

    disc[u] = low[u] = ++time;

    for (auto v : adj[u]) {
        if (!visited[v]) {
            children++;
```

```
            APUtil(v, visited, disc, low, time, u, isAP);

            low[u] = min(low[u], low[v]);

            if (parent != -1 && low[v] >= disc[u])
                isAP[u] = true;
        }

        else if (v != parent)
            low[u] = min(low[u], disc[v]);
    }

    if (parent == -1 && children > 1)
        isAP[u] = true;
}

void AP(int V) {
    int disc[V] = { 0 };
    int low[V];
    bool visited[V] = { false };
    bool isAP[V] = { false };
    int time = 0, par = -1;

    for (int u = 0; u < V; u++)
        if (!visited[u])
            APUtil(u, visited, disc, low, time, par, isAP);

    bool printed = false;

    for (int u = 0; u < V; u++) {
        if (isAP[u] == true) {
            cout << u+1 << " ";
            printed = true;
        }
    }

    if (!printed) cout << "nenhum" << endl;
    else cout << endl;
}

void solve() {

    int n, ed; cin >> n >> ed;

    for(int i = 0; i < n; i++)
        adj[i].clear();

    while(ed--) {
        int a, b; cin >> a >> b; a--, b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    AP(n);
}
```

## 8.7 Bipartido

```
// Description: Determina se um grafo eh bipartido ou nao
// Complexidade: O(V+E)

vector<vi> AL;

bool bipartido(int n) {

    int s = 0;
    queue<int> q; q.push(s);

    vi color(n, INF); color[s] = 0;
    bool ans = true;
    while (!q.empty() && ans) {
        int u = q.front(); q.pop();

        for (auto &v : AL[u]) {
            if (color[v] == INF) {
                color[v] = 1 - color[u];
                q.push(v);
            }
            else if (color[v] == color[u]) {
                ans = false;
                break;
            }
        }
    }

    return ans;
}

void solve() {

    int n, edg; cin >> n >> edg;
    AL.resize(n, vi());

    while(edg--) {
        int a, b; cin >> a >> b;
        AL[a].push_back(b);
        AL[b].push_back(a);
    }

    cout << bipartido(n) << endl;
}
```

## 8.8 Caminho Minimo - @Tabela

```
Criterio |   BFS (V + E) | Dijkstra (E*log V) | Bellman-Ford (V*E) | Floyd-Warshall (V^3)
---------+---------------+--------------------+--------------------+---------
Max Size | V + E <= 100M | V + E <= 1M        | V * E <= 100M      | V <= 450
Sem-Peso | CRIA          | Ok                 | Ruim               | Ruim no geral
```

```
Peso      | WA             | Melhor             | Ok                 | Ruim no geral
Peso Neg  | WA             | Modificado Ok      | Ok                 | Ruim no geral
Neg-Cic   | Nao Detecta    | Nao Detecta        | Detecta            | Detecta
Grafo Peq | WA se peso     | Overkill           | Overkill           | Melhor
```

## 8.9 Caminho Minimo - Bellman Ford

```cpp
// Description: Encontra menor caminho em grafos com pesos negativos
/* Complexidade:
    Conexo: O(VE)
    Desconexo: O(EV^2)
*/
// Classe: Single Source Shortest Path (SSSP)

vector<tuple<int,int,int>> edg; // edge: u, v, w
vi dist;

int bellman_ford(int n, int src) {
    dist.assign(n+1, INT_MAX);

    f(i,0,n+2) {
        for(auto& [u, v, w] : edg) {
            if(dist[u] != INT_MAX and dist[v] > w + dist[u])
                dist[v] = dist[u] + w;
        }
    }

    // Possivel checar ciclos negativos (ciclo de peso total negativo)
    for(auto& [u, v, w] : edg) {
        if(dist[u] != INT_MAX and dist[v] > w + dist[u])
            return 1;
    }

    return 0;
}

int main() {

    int n, edges; cin >> n >> edges;
    f(i,0,edges) {
        int u, v, w; cin >> u >> v >> w;
        edg.push_back({u, v, w});
    }
    bellman_ford(n, 1);
}
```

## 8.10 Caminho Minimo - Checar I J (In)Diretamente Conectados

```cpp
// Description: Verifica se o vertice i esta diretamente conectado ao vertice j
// Complexity: O(n^3)

const int INF = 1e9;
const int MAX_V = 450;
int adj[MAX_V][MAX_V];

void transitive_closure(int n) {

    for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        adj[i][j] |= (adj[i][k] & adj[k][j]);
}

void solve() {

    int n, ed; cin >> n >> ed;
    f(u,0,n) {
        f(v,0,n) {
            adj[u][v] = INF;
        }
        adj[u][u] = 0;
    }

    f(i,0,ed) {
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
    }

    transitive_closure(n);

    int i = 0, j = 0; cin >> i >> j;
    cout << (adj[i][j] == INF ? "Nao" : "Sim") << endl;
}
```

## 8.11 Caminho Minimo - Diametro Do Grafo

```cpp
// Description: Encontra o diametro de um grafo
//  => maximum shortest path between any two vertices
// Complexidade: O(n^3)

int adj[MAX_V][MAX_V];

int diameter(int n) {
    int ans = 0;
    f(u,0,n) {
        f(v,0,n) {
            if (adj[u][v] != INF) {
                ans = max(ans, adj[u][v]);
            }
        }
    }
    return ans;
}

void floyd_warshall(int n) {

    for (int k = 0; k < n; ++k)
```

```
    for (int u = 0; u < n; ++u)
    for (int v = 0; v < n; ++v)
        adj[u][v] = min(adj[u][v], adj[u][k]+adj[k][v]);
}

void solve() {

    int n, ed; cin >> n >> ed;
    f(u,0,n) {
        f(v,0,n) {
            adj[u][v] = INF;
        }
        adj[u][u] = 0;
    }

    f(i,0,ed) {
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
    }

    floyd_warshall(n);
    cout << diameter(n) << endl;
}
```

## 8.12 Caminho Minimo - Dijkstra

```
// Description: Algoritmo de Dijkstra para caminho ímnimo em grafos.
// Complexity: O(E log V)
// Classe: Single Source Shortest Path (SSSP)

vi dist;
vector<vector<pii>> adj;

void dijkstra(int s) {

    dist[s] = 0; // se eventualmente puder voltar pra ca, tipo ciclo | salesman | remover essa linha

    priority_queue<pii, vector<pii>, greater<pii>> pq; pq.push({0, s});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();

        if (d > dist[u]) continue;

        // if(u == s and dist[u] < INF) break; | pra quando tiver que fazer um ciclo

        for (auto &[v, w] : adj[u]) {
            if (dist[u] + w >= dist[v]) continue;
            dist[v] = dist[u]+w;
            pq.push({dist[v], v});
        }
    }
}

void solve() {

    int n, ed; cin >> n >> ed;
    adj.assign(n, vector<pii>());
    dist.assign(n, INF); // INF = 1e9

    while (ed--) {
        int u, v, w; cin >> u >> v >> w;
        adj[u].emplace_back(v, w);
    }

    int s; cin >> s;
    dijkstra(s);
}
```

## 8.13 Caminho Minimo - Floyd Warshall

```
// Description: Caminho minimo entre todos os pares de vertices em um grafo
// Complexity: O(n^3)
// Classe: All Pairs Shortest Path (APSP)

const int INF = 1e9;
const int MAX_V = 450;
int adj[MAX_V][MAX_V];

void printAnswer(int n) {
    for (int u = 0; u < n; ++u)
    for (int v = 0; v < n; ++v)
    cout << "APSP("<<u<<", "<<v<<") = " << adj[u][v] << endl;
}

void prepareParent() {
    f(i,0,n) {
        f(j,0,n) {
            p[i][j] = i;
        }
    }

    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (adj[i][k] + adj[k][j] < adj[i][j]) {
                    adj[i][j] = adj[i][k]+adj[k][j];
                    p[i][j] = p[k][j];
                }
}

vi restorePath(int u, int v) {

    if (adj[u][v] == INF) return {};
    vi path;
    for (; v != u; v = p[u][v]) {
        if (v == -1) return {};
        path.push_back(v);
    }
    path.push_back(u);
    reverse(path.begin(), path.end());
```

```
    return path;
}

void floyd_warshall(int n) {

    for (int k = 0; k < n; ++k)
    for (int u = 0; u < n; ++u)
    for (int v = 0; v < n; ++v)
        adj[u][v] = min(adj[u][v], adj[u][k]+adj[k][v]);
}

void solve() {

    int n, ed; cin >> n >> ed;
    f(u,0,n) {
        f(v,0,n) {
            adj[u][v] = INF;
        }
        adj[u][u] = 0;
    }

    f(i,0,ed) {
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
    }

    floyd_warshall(n);

    // prepareParent();
    // vi path = restorePath(0, 3);
}
```

## 8.14 Caminho Minimo - Minimax

```
// Description: MiniMax problem: encontrar o menor caminho mais longo entre todos os pares de vertices em um grafo
// Complexity: O(n^3)

const int INF = 1e9;
const int MAX_V = 450;
int adj[MAX_V][MAX_V];

void miniMax(int n) {
    for (int k = 0; k < V; ++k)
    for (int i = 0; i < V; ++i) // reverse min and max
    for (int j = 0; j < V; ++j) // for MaxiMin problem
    AM[i][j] = min(AM[i][j], max(AM[i][k], AM[k][j]));
}

void solve() {

    int n, ed; cin >> n >> ed;
    f(u,0,n) {
        f(v,0,n) {
            adj[u][v] = INF;
        }
        adj[u][u] = 0;
    }

    f(i,0,ed) {
        int u, v, w; cin >> u >> v >> w;
        adj[u][v] = w;
    }

    transitive_closure(n);

    int i = 0, j = 0; cin >> i >> j;
    cout << (adj[i][j] == INF ? "Nao" : "Sim") << endl;
}
```

## 8.15 Cycle Check

```
// Descriptionn: Checa se um grafo direcionado possui ciclos e imprime os tipos de arestas.
// Complexidade: O(V + E)

vector<vector<pii>> adj;
vi dfs_num, dfs_parent;

void cycleCheck(int u) {
    dfs_num[u] = -2;
    for (auto &[v, w] : adj[u]) {
        if (dfs_num[v] == -1) {
            dfs_parent[v] = u;
            cycleCheck(v);
        }
        else if (dfs_num[v] == -2) {
            if (v == dfs_parent[u])
                cout << " Bidirectional Edge (" << u << ", " << v << ")-(" << v << ", " << u << ")\n";
            else
                cout << "Back Edge (" << u << ", " << v << ") (Cycle)\n";
        }
        else if (dfs_num[v] == -3)
            cout << "  Forward/Cross Edge (" << u << ", " << v << ")\n";
    }
    dfs_num[u] = -3;
}

void solve() {
    int n, ed; cin >> n >> ed;
    adj.assign(ed, vector<pii>());

    for (int i = 0; i < ed; ++i) {
        int u, v, w; cin >> u >> v >> w;
        adj[u].emplace_back(v, w);
    }

    cout << "Graph Edges Property Check\n";
    dfs_num.assign(ed, -1);
    dfs_parent.assign(ed, -1);
    for (int u = 0; u < n; ++u)
        if (dfs_num[u] == -1)
        cycleCheck(u);
```

```
}
```

## 8.16 Encontrar Ciclo

```
// Description: Encontrar ciclo em grafo nao direcionado
// Complexidade: O(n + m)

int n;
vector<vector<int>> adj;
vector<bool> vis;
vector<int> p;
int cycle_start, cycle_end;

bool dfs(int v, int par) {
    vis[v] = true;
    for (int u : adj[v]) {
        if(u == par) continue;
        if(vis[u]) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
        p[u] = v;
        if(dfs(u, p[u]))
            return true;
    }
    return false;
}

vector<int> find_cycle() {
    cycle_start = -1;

    for (int v = 0; v < n; v++)
        if (!vis[v] and dfs(v, p[v]))
            break;

    if (cycle_start == -1) return {};

    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = p[v])
        cycle.push_back(v);
    cycle.push_back(cycle_start);
    return cycle;
}

void solve() {
    int edg; cin >> n >> edg;
    adj.assign(n, vector<int>());
    vis.assign(n, false), p.assign(n, -1);
    while(edg--) {
        int a, b; cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    vector<int> ans = find_cycle();
}
```

## 8.17 Euler Tree

```
// Descricao: Encontra a euler tree de um grafo
// Complexidade: O(n)
vector<vector<int>> adj(MAX);
vector<int> vis(MAX, 0);
vector<int> euTree(MAX);

void eulerTree(int u, int &index) {
    vis[u] = 1;
    euTree[index++] = u;
    for (auto it : adj[u]) {
        if (!vis[it]) {
            eulerTree(it, index);
            euTree[index++] = u;
        }
    }
}

void solve() {

    f(i,0,n-1) {
        int a, b; cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    int index = 0; eulerTree(1, index);
}
```

## 8.18 Isomorfia

```
// Descricao: Verifica se dois grafos sao isomorfos (possuem a mesma
    estrutura)
// Complexidade: O(n^2)

struct Node {
    unordered_map<int, int> edges;
    int grau;
};

Node adj1[MAX], adj2[MAX];

bool verify(int a, int b, int origemA, int origemB) {

    Node& node1 = adj1[a], node2 = adj2[b];

    if (node1.grau != node2.grau)
        return false;

    int n = node1.grau;
    bool usedA[n], usedB[n];

    f(i,0,n) {
        usedA[i] = (node1.edges[i] == origemA);
        usedB[i] = (node2.edges[i] == origemB);
```

```
    }

    f(i,0,n) {

        if (usedA[i]) continue;

        f(j,0,n) {
            if (usedB[j]) continue;

            if (verify(node1.edges[i], node2.edges[j], a, b)) {
                usedA[i] = usedB[j] = true;
                break;
            }
        }
        if (!usedA[i])
            return false;
    }
    return true;
}

bool areIsomorphic(int n) {

    f(i,0,n) {
        if (verify(0, i, -1, -1))
            return true;
    }

    return false;
}

void solve(int n) {

    f(i,0,n) {
        adj1[i].edges.clear(), adj2[i].edges.clear();
        adj1[i].grau = 0, adj2[i].grau = 0;
    }

    f(i,0,n-1) {
        int a, b; cin >> a >> b; a--, b--;
        adj1[a].edges[adj1[a].grau++] = b;
        adj1[b].edges[adj1[b].grau++] = a;
    }

    f(i,0,n-1) {
        int a, b; cin >> a >> b; a--, b--;
        adj2[a].edges[adj2[a].grau++] = b;
        adj2[b].edges[adj2[b].grau++] = a;
    }

    bool ans = areIsomorphic(n);
}
```

## 8.19 Kosaraju

```
// Description: Encontra o numero de componentes fortemente conexas em um grafo direcionado
// Complexidade: O(V + E)

int dfsNumberCounter, numSCC;
vector<vii> adj, adj_t;
vi dfs_num, dfs_low, S, visited;
stack<int> St;

void kosarajuUtil(int u, int pass) {
    dfs_num[u] = 1;
    vii &neighbor = (pass == 1) ? adj[u] : adj_t[u];
    for (auto &[v, w] : neighbor)
        if (dfs_num[v] == -1)
        kosarajuUtil(v, pass);
    S.push_back(u);
}

bool kosaraju(int n) {

    S.clear();
    dfs_num.assign(n, -1);

    f(u,0,n) {
        if (dfs_num[u] == -1)
            kosarajuUtil(u, 1);
    }

    int numSCC = 0;
    dfs_num.assign(n, -1);
    f(i,n-1,-1) {
        if (dfs_num[S[i]] == -1)
            numSCC++, kosarajuUtil(S[i], 2);
    }

    return numSCC == 1;
}

void solve() {

    int n, ed; cin >> n >> ed;
    adj.assign(n, vii());
    adj_t.assign(n, vii());

    while (ed--) {
        int u, v, w; cin >> u >> v >> w;
        AL[u].emplace_back(v, 1);
        adj_t[v].emplace_back(u, 1);
    }

    // Printa se o grafo eh fortemente conexo
    cout << kosaraju(n) << endl;

    // Printa o numero de componentes fortemente conexas
    cout << numSCC << endl;

    // Printa os vertices de cada componente fortemente conexa
    f(i,0,n){
        if (dfs_num[i] == -1) cout << i << ": " << "Nao visitado" << endl;
        else cout << i << ": " << dfs_num[i] << endl;
```

```
    }
}
```

## 8.20 Kruskal

```
// DEscricao: Encontra a arvore geradora minima de um grafo
// Complexidade: O(E log V)

vector<int> id, sz;

int find(int a){ // O(a(N)) amortizado
    return id[a] = (id[a] == a ? a : find(id[a]));
}

void uni(int a, int b) { // O(a(N)) amortizado
    a = find(a), b = find(b);
    if(a == b) return;

    if(sz[a] > sz[b]) swap(a,b);
    id[a] = b, sz[b] += sz[a];
}

pair<int, vector<tuple<int, int, int>>> kruskal(vector<tuple<int, int, int>>& edg) {

    sort(edg.begin(), edg.end()); // Minimum Spanning Tree

    int cost = 0;
    vector<tuple<int, int, int>> mst; // opcional
    for (auto [w,x,y] : edg) if (find(x) != find(y)) {
        mst.emplace_back(w, x, y); // opcional
        cost += w;
        uni(x,y);
    }
    return {cost, mst};
}

void solve() {

    int n, ed;

    id.resize(n); iota(all(id), 0);
    sz.resize(n, -1);
    vector<tuple<int, int, int>> edg;

    f(i,0,ed) {
        int a, b, w; cin >> a >> b >> w;
        edg.push_back({w, a, b});
    }

    auto [cost, mst] = kruskal(edg);
}

// VARIANTES

// Maximum Spanning Tree: sort(edg.rbegin(), edg.rend());

/* 'Minimum' Spanning Subgraph:
    - Algumas arestas ja foram adicionadas (maior prioridade - Questao das rodovias)
    - Arestas que nao foram adicionadas (menor prioridade - ferrovias)
    -> kruskal(rodovias); kruskal(ferrovias);
*/

/* Minimum Spanning Forest:
    - Queremos uma floresta com k componentes
    -> kruskal(edg); if(mst.sizer() == k) break;
*/

/* MiniMax
    - Encontrar menor caminho entre dous vertices com maior quantidade de arestas
    -> kruskal(edg); dijsktra(mst);
*/

/* Second Best MST
    - Encontrar a segunda melhor arvore geradora minima
    -> kruskal(edg);
    -> flag mst[i] = 1;
    -> sort(cmp(edg.flag != -1)) => da prioridade para outras arestas
*/
```

## 8.21 Labirinto

```
// Verifica se eh possivel sair de um labirinto
// Complexidade: O(4^(n*m))

vector<pair<int,int>> mov = {{1,0}, {0,1}, {-1,0}, {0,-1}};
vector<vector<int>> labirinto, sol;
vector<vector<bool>> visited;
int L, C;

bool valid(const int& x, const int& y) {
    return x >= 0 and x < L and y >= 0 and y < C and labirinto[x][y] != 0 and !visited[x][y];
}

bool condicaoSaida(const int& x, const int& y) {
    return labirinto[x][y] == 2;
}

bool search(const int& x, const int& y) {

    if(!valid(x, y))
        return false;

    if(condicaoSaida(x,y)) {
        sol[x][y] = 2;
        return true;
    }

    sol[x][y] = 1;
    visited[x][y] = true;

```

```
    for(auto [dx, dy] : mov)
        if(search(x+dx, y+dy))
            return true;

    sol[x][y] = 0;
    return false;
}

int main() {

    labirinto = {
        {1, 0, 0, 0},
        {1, 1, 0, 0},
        {0, 1, 0, 0},
        {1, 1, 1, 2}
    };

    L = labirinto.size(), C = labirinto[0].size();
    sol.resize(L, vector<int>(C, 0));
    visited.resize(L, vector<bool>(C, false));

    cout << search(0, 0) << endl;
}
```

## 8.22 Pontos Articulacao

```
// Description: Encontra os pontos de çãarticulao de um grafo ãno direcionado
// Complexidade: O(V*(V+E))

int V;
vector<vi> adj;
vi ans;

void dfs(vector<bool>& vis, int i, int curr) {
    vis[curr] = 1;
    for (auto x : adj[curr]) {
        if (x != i) {
            if (!vis[x]) {
                dfs(vis, i, x);
            }
        }
    }
}

void AP() {

    f(i,1,V+1) {
        int components = 0;
        vector<bool> vis(V + 1, 0);
        f(j,1, V+1) {
            if (j != i) {
                if (!vis[j]) {
                    components++;
                    dfs(vis, i, j);
                }
            }
        }
        if (components > 1) {
            ans.push_back(i);
        }
    }
}

void solve() {

    V = n;
    adj.clear(), ans.clear();
    adj.resize(V+1);

    while(edg--) {
        int a, b; cin >> a >> b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    AP();

    // Vertices articulacao: ans
}
```

## 8.23 Prufer Code To Tree

```
bool vis [MAX];
vector<int> adj[MAX];
int freq[MAX];

void dfs (int a) {
    vis[a] = true;
    cout << "(" << a;
    for (const auto& p : adj[a]) {
        if (!vis[p]) {
            cout << " ";
            dfs(p);
        }
    }

    cout << ")";
}

// Description: Dado um ódigo de Prufer, construir a árvore correspondente, prenchendo a lista de êadjacncia
// Complexidade: O(V^2)
void pruferCodeToTree(queue<int>& q, int V) {

    f(j,1,V) {
        f(i,1,V+1) {
            if (freq[i] == 0) {

                int front = q.front(); q.pop();

                freq[i] = -1; // mark as visited
                freq[front]--; // decrease the frequency of the front element
```

```
                adj[front].push_back(i);
                adj[i].push_back(front);

                break;
            }
        }
    }
}

void solve(string s) {

    int testNum = s[0];

    if(!('0' <= testNum and testNum <= '9')) {
        cout << "(1)" << endl;
        return;
    }

    memset(freq, 0, sizeof(freq));
    memset(vis, 0, sizeof(vis));
    for (int i = 0; i < MAX; i++) adj[i].clear(); //

    stringstream ss(s);
    int v;

    queue<int> q;
    while (ss >> v) {
        freq[v]++;
        q.push(v);
    }

    int V = q.back(); // quantidade de vertices

    pruferCodeToTree(q, V);

    dfs(V);

    cout << endl;
}
```

## 8.24 Successor Graph

```
// Encontra sucessor de um vertice dentro de um grafo direcionado
// Pre calcular: O(nlogn)
// Consulta: O(logn)

vector<vector<int>> adj;

int succ(int x, int u) {
    if(k == 1) return adj[x][0];
    return succ(succ(x, k/2), k/2);
}
```

## 8.25 Topological Sort

```
// Description: Retorna ordenacao topologica de adj, e vazio se nao for DAG
// Complexidade: O(V+E)
// Explicacao: usado para ordenar vercies de um DAG de forma que para cada aresta direcionada uv, o évrtice u aparece antes do évrtice v na ordenacao

#define MAXN 50010

int grauEntrada[MAXN];
vi adj[MAXN];

vi topologicalSort(int n) {

    priority_queue<int, vi, greater<int>> pq;

    f(i,0,n) {
        if(!grauEntrada[i])
            pq.push(i);
    }

    vi ans;

    while (!pq.empty()) {
        int node = pq.top(); pq.pop();

        for(auto x : adj[node]) {
            grauEntrada[x]--;
            if (!grauEntrada[x])
                pq.push(x);
        }

        ans.push_back(node);
    }

    return ans.size() == n ? ans : vi();
}

void solve() {

    int n, ed; cin >> n >> ed;

    memset(grauEntrada, 0, sizeof grauEntrada);

    while(ed--) {
        int a, b; cin >> a >> b;
        grauEntrada[b]++;
        adj[a].push_back(b);
    }

    vi ans = topologicalSort(n);
}
```

# 9 Grafos Especiais

## 9.1 Arvore - @Info

```
Arvore (NDAG):

* Definicao
    - écontm V vertices e V-1 arestas (E = V-1)
    - todo algoritmo O(V+E) numa arvore eh O(V)
    - nao direcionado
    - sem ciclo
    - conexa
    - um unico caminho para todo par de vertices

* Aplicacoes

    -> TREE TRAVERSAL
        pre-order(v):           in-order(v):            post-order(v):
            visit(v)                in-order(left(v))       post-order(left(v))
            pre-order(left(v))      visit(v)                post-order(right(v))
            pre-order(right(v))     in-order(right(v))      visit(v)

    -> Pontos de Articulacao / Pontes
        - todo vertice eh ponto de articulacao

    -> Single Source Shortest Path (SSSP)
        - O(V) para achar o caminho minimo de um vertice para todos os outros
        - BFS ou DFS funcionam, mesmo com pesos

    -> All Pairs Shortest Path (APSP)
        - O(V^2) para achar o caminho minimo de todos para todos
        - V * SSSP

    -> Diametro
        - greatest 'shortest path length' between any pair of vertices
        - 2 * SSSP:
            1. BFS/DFS de qualquer vertice
            2. BFS/DFS do vertice mais distante => diametro = maior distancia

    -> Lowest Common Ancestor (LCA)
        - O(V) para achar o LCA de 2 vertices
        - O(V) para pre-processar
```

## 9.2 Bipartido - @Info

```
Grafo Bipartido

* Definicao
    - vertices podem ser divididos em 2 conjuntos disjuntos
    - todas as arestas conectam vertices de conjuntos diferentes
    - nao ha arestas entre vertices do mesmo conjunto
    - nao ha ciclos de tamanho impar
    > EX: arvores sao bipartidas

* Aplicacoes
```

## 9.3 Dag - @Info

```
Grafo Direcionado Aciclico (DAG):
* Definicao
    - tem direcao
    - nao tem ciclos
    - problemas com ele => usar DP (estados da DP = vertices DAG)
    - so tem um topological sort
* Aplicacoes
    - Single Source (Shortest / Longest) Path na DAG => O(V + E)
    - Numero de caminhos entre 2 vertices => O(V + E)
    - Numero de caminhos de um vertice para todos os outros => O(V + E)
    - DP de 'minimizacao', 'maximizacao', 'contar algo' => menor | maior | contar numero de caminhos na recursao de DP na DAG
* Exemplos
    - mochila
    - troco
```

## 9.4 Dag - Sslp

```
// Description: Finds SSLP (Single Source Longest Path) in a directed acyclic graph.
// Complexity: O(V + E)
// OBS: Not tested
vector<vector<pair<int,int>>> adj;

vector<int> dagLongestPath(int s, int n) {

    vector<int> topsort = topologicalSort();
    vector<int> dist(n, INT_MIN);
    dist[s] = 0;

    for (int i = 0; i < n; i++) {
        int nodeIndex = topsort[i];
        if (dist[nodeIndex] != INT_MIN) {
            auto adjacentEdges = adj[nodeIndex];
            for (auto [u, w] : adjacentEdges) {
                int newDist = dist[nodeIndex] + w;
                if (dist[u] == INT_MIN) dist[u] = newDist;
                else dist[u] = max(dist[u], newDist);
            }
        }
    }

    return dist;
}
```

## 9.5 Dag - Sssp

```
// Description: Encontra SSSP (Single Source Shortest Path) em um grafo íacclico direcionado.
// Complexity: O(V + E)
```

```cpp
// OBS: Nao testado
vector<vector<pair<int,int>>> adj;

vector<int> dagShortestPath(int s, int n) {

    vector<int> topsort = topologicalSort();
    vector<int> dist(n, INT_MAX);
    dist[s] = 0;

    for (int i = 0; i < n; i++) {
        int nodeIndex = topsort[i];
        if (dist[nodeIndex] != nodeIndex) {
            auto adjacentEdges = adj[nodeIndex];
            for (auto [u, w] : adjacentEdges) {
                int newDist = dist[nodeIndex] + w;
                if (dist[u] == INT_MAX) dist[u] = newDist;
                else dist[u] = min(dist[u], newDist);
            }
        }
    }

    return dist;
}
```

## 9.6 Dag - Fishmonger

```cpp
// Given the number of cities 3 <= n <= 50, available time 1 <= t <= 1000, and two n x n matrices (one gives travel times and another gives tolls between two cities), choose a route from the port city (vertex 0) in such a way that the fishmonger has to pay as little tolls as possible to arrive at the market city (vertex n-1) within a certain time t

// Cada estado eh um vertice da DAG (node, tempoRestante)

pii dp(int cur, int t_left) {
    if (t_left < 0) return {INF, INF};
    if (cur == n-1) return {0, 0};
    if (memo[cur][t_left] != {-1, -1}) return memo[cur][t_left];
    pii ans = {INF, INF};
    for (int X = 0; X < n; ++X)
    if (cur != X) {
        auto &[tollpaid, timeneeded] = dp(X, t_left-travelTime[cur][X]);
        if (tollpaid+toll[cur][X] < ans.first) {
        ans.first = tollpaid+toll[cur][X];
        ans.second = timeneeded+travelTime[cur][X];
        }
    }
    return memo[cur][t_left] = ans;
}
```

## 9.7 Dag - Numero De Caminhos 2 Vertices

```cpp
// Description: Encontra o únmero de caminhos entre dois évrtices em um grafo íacclico direcionado.
// Complexity: O(V + E)

const int MAXN = 1e5 + 5;

int dp[MAXN],
int mod = 1e9 + 7, n;
vector<vector<int>> adj;

int countPaths(int s, int d) {
    if (s == d) return 1;
    if (dp[s] != -1) return dp[s];

    int c = 0;
    for (int& neigh : adj[s]) {
        int x = countPaths(neigh, d);
        if (x != -1)
            c = (c % mod + x % mod) % mod;
    }
    return (dp[s] = (c == 0) ? -1 : c);
}

int countPossiblePaths(int s, int d) {
    memset(dp, -1, sizeof dp);
    int c = countPaths(s, d);
    if (c == -1) return 0;
    return c;
}

void solve() {
    int n, ed; cin >> n >> ed;
    adj.resize(n);

    for (int i = 0; i < ed; i++) {
        int u, v; cin >> u >> v;
        adj[u].push_back(v);
    }

    int src, end; cin >> src >> end; // 0-based
    cout << countPossiblePaths(src, end) << endl;
}
```

## 9.8 Eulerian - @Info

```
Eulerian Graph:

* Eulerian Path (Eulerian Tour):
    - caminho que atravessa grafo apenas 1 vez
    - Grafo Nao direcionado: tem um se e somente se tiver 0 ou 2 vertices de grau impar
    - Grafo Direcionado: tem um se e somente se
        1. todos os vertices tiverem o mesmo numero de arestas entrando e saindo
        2. eh 'conexo' (considerando arestas bidirecionadas)
* Definicao
    - nao direcionado
    - conexo
    - grau de todos os vertices par
```

### 9.9 Eulerian - Euler Path

```cpp
// Description: Encontra um caminho euleriano em um grafo direcionado
// Complexidade: O(E)
// OBS: testar com bidirecionado / encontrar versao que aceita bidirecionado

int N;
vector<vi> adj;
vi hierholzer(int s) {
    vi ans, idx(N, 0), st;
    st.push_back(s);
    while (!st.empty()) {
        int u = st.back();
        if (idx[u] < (int)adj[u].size()) {
            st.push_back(adj[u][idx[u]]);
            ++idx[u];
        }
        else {
            ans.push_back(u);
            st.pop_back();
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

# 10 Matematica

## 10.1 Casas

```cpp
// Descriptiuon: Conta quantas casas decimais certo numero tem

int casas(double a) {
    return (int)floor(1 + log10((double)a))
}
```

## 10.2 Ciclo Em Funcao

```cpp
// Description: Encontra o tamanho do ciclo de uma çãfuno f(x) = (Z*x + I) % M a partir de um valor inicial L.
// Complexidade: O(lambda + mu) | lambda = tamanho do ciclo | mu = tamanho do prefixo antes do ciclo.
// Return: pair<int, int> = {mu, lambda} | mu = tamanho do prefixo antes do ciclo | lambda = tamanho do ciclo.
// Parameters: x0 = valor inicial para encontrar o ciclo.
int f(int x); // f(x) do problema

pii floydCycleFinding(int x0) {
    int t = f(x0), h = f(f(x0));
    while (t != h) { t = f(t); h = f(f(h)); }
    int mu = 0; h = x0;
    while (t != h) { t = f(t); h = f(h); ++mu; }
    int lambda = 1; h = f(t);
    while (t != h) { h = f(h); ++lambda; }
    return {mu, lambda};
}
```

## 10.3 Contar Quanti Solucoes Eq 2 Variaveis

```cpp
// Description: Dada uma equacao de 2 variaveis, calcula quantas combinacoes {x,y}
// inteiras que resolvem essa equacao
// Complexidade: O(sqrt(c))
// y = numerador / denominador
int numerador(int x) { return c - x; } // expressao do numerador
int denominador(int x) { return 2 * x + 1; } // expressao do denominador

int count2VariableIntegerEquationAnswers() {

    unordered_set<pair<int,int>, PairHash> ans; int lim = sqrt(c);
    for(int i=1; i<= lim; i++) {
        if (numerador(i) % denominador(i) == 0) {
            int x = i, y = numerador(i) / denominador(i);
            if(!ans.count({x,y}) and !ans.count({y,x}))
                ans.insert({x,y});
        }
    }

    return ans.size();
}
```

## 10.4 Conversao De Bases

```cpp
// Converter um decimal (10) para base n [2, 8, 10, 16]
// Complexidade: O(log n)
char charForDigit(int digit) {
    if (digit > 9) return digit + 87;
    return digit + 48;
}

string decimalToBase(int n, int base = 10) {
    if (not n) return "0";
    stringstream ss;
    for (int i = n; i > 0; i /= base) {
        ss << charForDigit(i % base);
    }
    string s = ss.str();
    reverse(s.begin(), s.end());
    return s;
}

// Converter um numero de base [2, 8, 10, 16] para decimal (10)
// Complexidade: O(n)
int intForDigit(char digit) {
    int intDigit = digit - 48;
    if (intDigit > 9) return digit - 87;
    return intDigit;
}

int baseToDecimal(const string& n, int base = 10) {
```

```
    int result = 0;
    int basePow =1;
    for (auto it = n.rbegin(); it != n.rend(); ++it, basePow *= base)
        result += intForDigit(*it) * basePow;
    return result;
}
```

## 10.5 Decimal Para Fracao

```
// Converte um decimal para fracao irredutivel
// Complexidade: O(log n)
pair<int, int> toFraction(double n, unsigned p) {
    const int tenP = pow(10, p);
    const int t = (int) (n * tenP);
    const int rMdc = mdc(t, tenP);
    return {t / rMdc, tenP / rMdc};
}
```

## 10.6 Dois Primos Somam Num

```
// Description: Verifica se dois numeros primos somam um numero n.
// Complexity: O(sqrt(n))
bool twoNumsSumPrime(int n) {

    if(n % 2 == 0) return true;
    return isPrime(n-2);
}
```

## 10.7 Factorial

```
unordered_map<int, int> memo;

// Factorial
// Complexidade: O(n), onde n eh o numero a ser fatorado
int factorial(int n) {
    if (n == 0 || n == 1)  return 1;
    if (memo.find(n) != memo.end()) return memo[n];
    return memo[n] = n * factorial(n - 1);
}
```

## 10.8 Fast Exponentiation

```
const int mod = 1e9 + 7;

// Fast Exponentiation: retorna a^b % mod
// Quando usar: quando precisar calcular a^b % mod
int fexp(int a, int b)
{
    int ans = 1;
    while (b)
    {
        if (b & 1)
            ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}
```

## 10.9 Fast Fibonacci

```
#include <bits/stdc++.h>
using namespace std;

#define _ std::ios::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
#define all(a)        a.begin(), a.end()
#define int           long long int
#define double        long double
#define vi            vector<int>
#define pii           pair<int,int>
#define endl          "\n"
#define print_v(a)    for(auto x : a)cout<<x<<" ";cout<<endl
#define print_vp(a)   for(auto x : a)cout<<x.first<<" "<<x.second<< endl
#define f(i,s,e)      for(int i=s;i<e;i++)
#define rf(i,e,s)     for(int i=e-1;i>=s;i--)
#define CEIL(a, b)    ((a) + (b - 1))/b
#define TRUNC(x, n)   floor(x * pow(10, n))/pow(10, n)
#define ROUND(x, n)   round(x * pow(10, n))/pow(10, n)
#define dbg(x) cout << #x << " = " << x << " ";
#define dbgl(x) cout << #x << " = " << x << endl;
using namespace std;

string decimal_to_bin(int n) {
    string bin = bitset<sizeof(int) * 8>(n).to_string();
    auto loc = bin.find('1');
    // remove leading zeros
    if (loc != string::npos)
        return bin.substr(loc);
    return "0";
}

int fastfib(int n) {
    string bin_of_n = decimal_to_bin(n);

    int f[] = { 0, 1 };

    for (auto b : bin_of_n) {
        int f2i1 = f[1] * f[1] + f[0] * f[0];
        int f2i = f[0] * (2 * f[1] - f[0]);

        if (b == '0') {
            f[0] = f2i;
            f[1] = f2i1;
        } else {
            f[0] = f2i1;
            f[1] = f2i1 + f2i;
        }
    }

    return f[0];
}

```

```
int main() {
    int n = 13;
    int fib = fastfib(n);
    cout << "F(" << n << ") = " << fib << "\n";
}
```

## 10.10 Fatorial Grande

```
static BigInteger[] dp = new BigInteger[1000000];

public static BigInteger factorialDP(BigInteger n) {
    dp[0] = BigInteger.ONE;
    for (int i = 1; i <= n.intValue(); i++) {
        dp[i] = dp[i - 1].multiply(BigInteger.valueOf(i));
    }
    return dp[n.intValue()];
}
```

## 10.11 Fibonacci Modulo

```
long pisano(long m)
{
    long prev = 0;
    long curr = 1;
    long res = 0;

    for(int i = 0; i < m * m; i++)
    {
        long temp = 0;
        temp = curr;
        curr = (prev + curr) % m;
        prev = temp;

        if (prev == 0 && curr == 1)
            res = i + 1;
    }
    return res;
}

// Calculate Fn mod m
long fibonacciModulo(long n, long m)
{

    // Getting the period
    long pisanoPeriod = pisano(m);

    n = n % pisanoPeriod;

    long prev = 0;
    long curr = 1;

    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;

    for(int i = 0; i < n - 1; i++)
    {
        long temp = 0;
        temp = curr;
        curr = (prev + curr) % m;
        prev = temp;
    }
    return curr % m;
}
```

## 10.12 Mmc Mdc - Euclides Extendido

```
// Description: Retorna mdc(a, b) e referencia inteiros x, y t.q ax + by =
    mdc(a, b).
// Complexidade: O(log(min(a, b)))

int extEuclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a/b;
        tie(a, b) = tuple(b, a%b);
        tie(x, xx) = tuple(xx, x-q*xx);
        tie(y, yy) = tuple(yy, y-q*yy);
    }
    return a;
}
```

## 10.13 Mmc Mdc - Mdc

```
// Description: Calcula o mdc de dois numeros inteiros.
// Complexidade: O(logn) onde n eh o maior numero
int mdc(int a, int b) {
    for (int r = a % b; r; a = b, b = r, r = a % b);
    return b;
}
```

## 10.14 Mmc Mdc - Mdc Multiplo

```
// Description: Calcula o MDC de um vetor de inteiros.
// Complexidade: O(nlogn) onde n eh o tamanho do vetor
int mdc_many(vector<int> arr) {
   int result = arr[0];

   for (int& num : arr) {
       result = mdc(num, result);

       if(result == 1) return 1;
   }
   return result;
}
```

## 10.15 Mmc Mdc - Mmc

```
// Description: Calcula o mmc de dois únmeros inteiros.
```

```
// Complexidade: O(logn) onde n eh o maior numero
int mmc(int a, int b) {
    return a / mdc(a, b) * b;
}
```

## 10.16 Mmc Mdc - Mmc Multiplo

```
// Description: Calcula o mmc de um vetor de inteiros.
// Complexidade: O(nlogn) onde n eh o tamanho do vetor
int mmc_many(vector<int> arr)
{
    int result = arr[0];

    for (int &num : arr)
        result = (num * result / mdc(num, result));
    return result;
}
```

## 10.17 Modulo - @Info

```
SOMA
(a + b) % m = ((a % m) + (b % m)) % mÇÃ

SUBTRAO
(a - b) % m = ((a % m) - (b % m) + m) % mÇÃ

MULTIPLICAO
(a * b) % m = ((a % m) * (b % m)) % mÃ

DIVISO
(a / b) % m       = (a * b^-1) % m
// se m eh primo = ((a % m) * (b^(m-2) % m)) % m.
// else          = (a * modInverse(b, m)) % m

POTENCIA
(a ^ b) % m = ((a % m) ^ b) % m = modPow(a, b, m)
```

## 10.18 Modulo - Divisao E Potencia Mod M

```
// Retorna a % m (garante que o resultado é positivo)
int mod(int a, int m) {
    return ((a%m) + m) % m;
}

// Description: retorna b^(-1) mod m, ou -1 se ãno existir.
// Complexidade: O(log(min(b, m)))
int modInverse(int b, int m) {
    int x, y;
    int d = extEuclid(b, m, x, y);
    if (d != 1) return -1;
    return mod(x, m);
}

// Description: retorna b^p mod m
// Complexidade: O(log(p))
int modPow(int b, int p, int m) {
  if (p == 0) return 1;
  int ans = modPow(b, p/2, m);
  ans = mod(ans*ans, m);
  if (p&1) ans = mod(ans*b, m);
  return ans;
}
```

## 10.19 Modulo - Fibonacci Modulo

```
// Descricao: Calcula o n-esimo numero de Fibonacci modulo P
// Complexidade: O(log(n))

int mostSignificantBitPosition(int n) {
    int msb_position = 63;
    while (!((1 << (msb_position-1) & n)) && msb_position >= 0)
        msb_position--;
    return msb_position;
}

int fib (int n, int P) {

    int msb_position = mostSignificantBitPosition(n);

    int a=0, b=1;

    for (int i=msb_position; i>=0;--i) {
        int d = (a%P) * ((b%P)*2 - (a%P) + P),
            e = (a%P) * (a%P) + (b%P)*(b%P);
        a = d % P;
        b = e % P;

        if (((n >> i) & 1) != 0) {
            int c = (a + b) % P;
            a = b;
            b = c;
        }
    }
    return a;
}
```

## 10.20 N Fibonacci

```
int dp[MAX];

int fibonacciDP(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (dp[n] != -1) return dp[n];
    return dp[n] = fibonacciDP(n-1) + fibonacciDP(n-2);
}

int nFibonacci(int minus, int times, int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (dp[n] != -1) return dp[n];
    int aux = 0;
```

```
    for(int i=0; i<times; i++) {
        aux += nFibonacci(minus, times, n-minus);
    }
}
```

## 10.21 Numeros Grandes

```
public static void BbigInteger() {

    BigInteger a = BigInteger.valueOf(1000000000);
                a = new BigInteger("10000000000");

    // çõOperaes com inteiros grandes
    BigInteger  arit = a.add(a);
                arit = a.subtract(a);
                arit = a.multiply(a);
                arit = a.divide(a);
                arit = a.mod(a);

    // çãComparao
    boolean bool = a.equals(a);
            bool = a.compareTo(a) > 0;
            bool = a.compareTo(a) < 0;
            bool = a.compareTo(a) >= 0;
            bool = a.compareTo(a) <= 0;

    // ãConverso para string
    String m = a.toString();

    // ãConverso para inteiro
    int     _int = a.intValue();
    long   _long = a.longValue();
    double _doub = a.doubleValue();

    // êPotncia
    BigInteger _pot = a.pow(10);
    BigInteger _sqr = a.sqrt();

}

public static void BigDecimal() {

    BigDecimal a = new BigDecimal("10000000000");
                a = new BigDecimal("10000000000.0000000000");
                a = BigDecimal.valueOf(1000000000, 10);


    // çõOperaes com reais grandes
    BigDecimal arit = a.add(a);
                arit = a.subtract(a);
                arit = a.multiply(a);
                arit = a.divide(a);
                arit = a.remainder(a);

    // çãComparao
    boolean bool = a.equals(a);
            bool = a.compareTo(a) > 0;
            bool = a.compareTo(a) < 0;
            bool = a.compareTo(a) >= 0;
            bool = a.compareTo(a) <= 0;

    // ãConverso para string
    String m = a.toString();

    // ãConverso para inteiro
    int     _int = a.intValue();
    long   _long = a.longValue();
    double _doub = a.doubleValue();

    // êPotncia
    BigDecimal _pot = a.pow(10);
}
```

## 10.22 Primos - Divisores De N - Listar

```
// Description: Retorna o numero de divisores de N
// Complexidade: O(log(N))
// Exemplo: numDiv(60) = 12 {1,2,3,4,5,6,10,12,15,20,30,60}
int numDiv(int N) {
    int ans = 1;
    for (int i = 0; i < p.size() and p[i]*p[i] <= N; ++i) {
        int power = 0;
        while (N%p[i] == 0) { N /= p[i]; ++power; }
        ans *= power+1;
    }
    return (N != 1) ? 2*ans : ans;
}
```

## 10.23 Primos - Divisores De N - Somar

```
// Description: Retorna a soma dos divisores de N
// Complexidade: O(log(N))
// Exemplo: sumDiv(60) = 168 : 1+2+3+4+5+6+10+12+15+20+30+60
int sumDiv(int N) {
    int ans = 1;
    for (int i = 0; i < p.size() and p[i]*p[i] <= N; ++i) {
        int multiplier = p[i], total = 1;
        while (N%p[i] == 0) {
            N /= p[i];
            total += multiplier;
            multiplier *= p[i];
        }
        ans *= total;
    }
    if (N != 1) ans *= (N+1);
    return ans;
}
```

## 10.24 Primos - Fatores Primos - Contar Diferentes

```
// Description: Retorna o numero de fatores primos diferentes de N
// Complexidade: O(sqrt(N))
// Exemplo: numDiffPF(60) = 3 {2, 3, 5}
```

```

int numDiffPF(int N) {
    int ans = 0;
    for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i) {
        if (N%p[i] == 0) ++ans;                         // count this prime
    factor
        while (N%p[i] == 0) N /= p[i];                  // only once
    }
    if (N != 1) ++ans;
    return ans;
}
```

## 10.25 Primos - Fatores Primos - Listar

```
// Fatora um únmero em seus fatores primos
// Complexidade: O(sqrt(n))
// Ex: factorize(1200) = {2: 4, 3: 1, 5: 2}

map<int, int> factorize(int n) {
    map<int, int> factorsOfN;
    int lpf = 2;

    while (n != 1) {
        lpf = lowestPrimeFactor(n, lpf);
        factorsOfN[lpf] = 1;
        n /= lpf;
        while (not (n % lpf)) {
            factorsOfN[lpf]++;
            n /= lpf;
        }
    }

    return factorsOfN;
}
```

## 10.26 Primos - Fatores Primos - Somar

```
// Description: Retorna a soma dos fatores primos de N
// Complexidade: O(log(N))
// Exemplo: sumPF(60) = sumPF(2^2 * 3^1 * 5^1) = 2 + 2 + 3 + 5 = 12

int sumPF(int N) {
    int ans = 0;
    for (int i = 0; i < p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) { N /= p[i]; ans += p[i]; }
    if (N != 1) ans += N;
    return ans;
}
```

## 10.27 Primos - Is Prime

```
// Descricao: Funcao que verifica se um numero n eh primo.
// Complexidade: O(sqrt(n))
bool isPrime(int n) {
    return n > 1 and lowestPrimeFactor(n) == n;
}
```

## 10.28 Primos - Lowest Prime Factor

```
// Description: Funcao auxiliar que retorna o menor fator primo de n.
// Complexidade: O(sqrt(n))

int lowestPrimeFactor(int n, int startPrime = 2) {
    if (startPrime <= 3) {
        if (not (n & 1)) return 2;
        if (not (n % 3)) return 3;
        startPrime = 5;
    }

    for (int i = startPrime; i * i <= n; i += (i + 1) % 6 ? 4 : 2)
        if (not (n % i))
            return i;
    return n;
}
```

## 10.29 Primos - Miller Rabin

```
// Teste de primalidade de Miller-Rabin
// Complexidade: O(k*log^3(n)), onde k eh o numero de testes e n eh o
    numero a ser testado
// Descicao: Testa se um numero eh primo com uma probabilidade de erro de
    1/4^k

int mul(int a, int b, int m) {
    int ret = a*b - int((long double)1/m*a*b+0.5)*m;
    return ret < 0 ? ret+m : ret;
}

int pow(int x, int y, int m) {
    if (!y) return 1;
    int ans = pow(mul(x, x, m), y/2, m);
    return y%2 ? mul(x, ans, m) : ans;
}

bool prime(int n) {
    if (n < 2) return 0;
    if (n <= 3) return 1;
    if (n % 2 == 0) return 0;
    int r = __builtin_ctzint(n - 1), d = n >> r;

    // com esses primos, o teste funciona garantido para n <= 2^64
    // funciona para n <= 3*10^24 com os primos ate 41
    for (int a : {2, 325, 9375, 28178, 450775, 9780504, 795265022}) {
        int x = pow(a, d, n);
        if (x == 1 or x == n - 1 or a % n == 0) continue;

        for (int j = 0; j < r - 1; j++) {
            x = mul(x, x, n);
            if (x == n - 1) break;
        }
        if (x != n - 1) return 0;
    }
    return 1;
}
```

## 10.30 Primos - Numero Fatores Primos De N

```
// Description: Retorna o numero de fatores primos de N
// Complexidade: O(log(N))

vi p; // vetor de primos p (sieve(10000000))

int numPF(int N) {
    int ans = 0;
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i)
        while (N%p[i] == 0) { N /= p[i]; ++ans; }
    return ans + (N != 1);
}
```

## 10.31 Primos - Primo Grande

```
// Description: verificar se um numero > 9e18 eh primo
public static boolean isProbablePrime(BigInteger num, int certainty) {
    return num.isProbablePrime(certainty);
}
```

## 10.32 Primos - Primos Relativos De N

```
// Description: Conta o numero de primos relativos de N
// Complexidade: O(log(N))
// Exemplo: countPrimosRelativos(60) = 16
    {1,7,11,13,17,19,23,29,31,37,41,43,47,49,53,59}
// Definicao: Dois numeros sao primos relativos se o mdc entre eles eh 1

int countPrimosRelativos(int N) {
    int ans = N;
    for (int i = 0; i < (int)p.size() && p[i]*p[i] <= N; ++i) {
        if (N%p[i] == 0) ans -= ans/p[i];
        while (N%p[i] == 0) N /= p[i];
    }
    if (N != 1) ans -= ans/N;
    return ans;
}
```

## 10.33 Primos - Sieve

```
// Description: Gera todos os primos do intervalo [1,lim]
// Complexidade: O(n log log n)

int _sieve_size;
bitset<10000010> bs;
vi p;

void sieve(int lim) {
    _sieve_size = lim+1;
    bs.set();
    bs[0] = bs[1] = 0;
    f(i,2,_sieve_size) {
        if (bs[i]) {
            for (int j = i*i; j < _sieve_size; j += i) bs[j] = 0;
            p.push_back(i);
        }
    }
}
```

## 10.34 Primos - Sieve Linear

```
// Sieve de Eratosthenes com linear sieve
// Encontra todos os únmeros primos no intervalo [2, N]
// Complexidade: O(N)

vector<int> sieve(const int N) {

    vector<int> lp(N + 1); // lp[i] = menor fator primo de i
    vector<int> pr;

    for (int i = 2; i <= N; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            pr.push_back(i);
        }
        for (int j = 0; i * pr[j] <= N; ++j) {
            lp[i * pr[j]] = pr[j];
            if (pr[j] == lp[i])
                break;
        }
    }

    return pr;
}
```

## 10.35 Tabela Verdade

```
// Gerar tabela verdade de uma ãexpresso booleana
// Complexidade: O(2^n)

vector<vector<int>> tabelaVerdade;
int indexTabela = 0;

void backtracking(int posicao, vector<int>& conj_bool) {

    if(posicao == conj_bool.size()) { // Se chegou ao fim da BST
        for(size_t i=0; i<conj_bool.size(); i++) {
            tabelaVerdade[indexTabela].push_back(conj_bool[i]);
        }
        indexTabela++;

    } else {
        conj_bool[posicao] = 1;
        backtracking(posicao+1,conj_bool);
        conj_bool[posicao] = 0;
        backtracking(posicao+1,conj_bool);
    }
}

int main() {
```

```
    int n = 3;

    vector<int> linhaBool (n, false);
    tabelaVerdade.resize(pow(2,n));

    backtracking(0,linhaBool);
}
```

# 11 Matriz

## 11.1 Fibonacci Matricial

```

#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

ll MOD;

const int MAX_N = 2;

struct Matrix { ll mat[MAX_N][MAX_N]; };

ll mod(ll a, ll m) { return ((a%m)+m) % m; }

Matrix matMul(Matrix a, Matrix b) {
    Matrix ans;
    for (int i = 0; i < MAX_N; ++i)
        for (int j = 0; j < MAX_N; ++j)
        ans.mat[i][j] = 0;
    for (int i = 0; i < MAX_N; ++i)
        for (int k = 0; k < MAX_N; ++k) {
        if (a.mat[i][k] == 0) continue;
        for (int j = 0; j < MAX_N; ++j) {
            ans.mat[i][j] += mod(a.mat[i][k], MOD) * mod(b.mat[k][j], MOD)
    ;
            ans.mat[i][j] = mod(ans.mat[i][j], MOD);
        }
        }
    return ans;
    }

Matrix matPow(Matrix base, int p) {
    Matrix ans;
    for (int i = 0; i < MAX_N; ++i)
        for (int j = 0; j < MAX_N; ++j)
        ans.mat[i][j] = (i == j);
    while (p) {
        if (p&1)
        ans = matMul(ans, base);
        base = matMul(base, base);
        p >>= 1;
    }
    return ans;
}

int main() {
    int n, m;
    while (scanf("%d %d", &n, &m) == 2) {
        Matrix ans;
        ans.mat[0][0] = 1;  ans.mat[0][1] = 1;
        ans.mat[1][0] = 1;  ans.mat[1][1] = 0;
        MOD = 1LL << m;
        ans = matPow(ans, n);
        printf("%lld\n", ans.mat[0][1]);
    }
    return 0;
}
```

## 11.2 Maior Retangulo Binario Em Matriz

```
// Description: Encontra o maior âretngulo ábinrio em uma matriz.
// Time: O(n*m)
// Space: O(n*m)
tuple<int, int, int> maximalRectangle(vector<vector<int>>& mat) {
    int r = mat.size();
    if(r == 0) return {0, 0, 0};
    int c = mat[0].size();

    vector<vector<int>> dp(r+1, vector<int>(c));

    int mx = 0;
    int area = 0, height = 0, length = 0;
    for(int i=1; i<r; ++i) {
        int leftBound = -1;
        stack<int> st;
        vector<int> left(c);

        for(int j=0; j<c; ++j) {
            if(mat[i][j] == 1) {
                mat[i][j] = 1+mat[i-1][j];
                while(!st.empty() and mat[i][st.top()] >= mat[i][j])
                    st.pop();

                int val = leftBound;
                if(!st.empty())
                    val = max(val, st.top());

                left[j] = val;
            } else {
                leftBound = j;
                left[j] = 0;
            }
            st.push(j);
        }
        while(!st.empty()) st.pop();

        int rightBound = c;
        for(int j=c-1; j>=0; j--) {
            if(mat[i][j] != 0) {
```

```
                while(!st.empty() and mat[i][st.top()] >= mat[i][j])
                    st.pop();

                int val = rightBound;
                if(!st.empty())
                    val = min(val, st.top());

                dp[i][j] = (mat[i][j]) * (((val-1)-(left[j]+1)+1));
                if (dp[i][j] > mx) {
                    mx = dp[i][j];
                    area = mx;
                    height = mat[i][j];
                    length = (val-1)-(left[j]+1)+1;
                }
                st.push(j);
            } else {
                dp[i][j] = 0;
                rightBound = j;
            }
        }
    }

    return {area, height, length};
}
    int r = mat.size();
    if(r == 0) return make_tuple(0, 0, 0);
    int c = mat[0].size();

    vector<vector<int>> dp(r+1, vector<int>(c));

    int mx = 0;
    int area = 0, height = 0, length = 0;
    for(int i=1; i<r; ++i) {
        int leftBound = -1;
        stack<int> st;
        vector<int> left(c);

        for(int j=0; j<c; ++j) {
            if(mat[i][j] == 1) {
                mat[i][j] = 1+mat[i-1][j];
                while(!st.empty() and mat[i][st.top()] >= mat[i][j])
                    st.pop();

                int val = leftBound;
                if(!st.empty())
                    val = max(val, st.top());

                left[j] = val;
            } else {
                leftBound = j;
                left[j] = 0;
            }
            st.push(j);
        }
        while(!st.empty()) st.pop();

        int rightBound = c;
        for(int j=c-1; j>=0; j--) {
            if(mat[i][j] != 0) {

                while(!st.empty() and mat[i][st.top()] >= mat[i][j])
                    st.pop();

                int val = rightBound;
                if(!st.empty())
                    val = min(val, st.top());

                dp[i][j] = (mat[i][j]+1) * (((val-1)-(left[j]+1)+1)+1);
                if (dp[i][j] > mx) {
                    mx = dp[i][j];
                    area = mx;
                    height = mat[i][j];
                    length = (val-1)-(left[j]+1)+1;
                }
                st.push(j);
            } else {
                dp[i][j] = 0;
                rightBound = j;
            }
        }
    }

    return make_tuple(area, height, length);
}
```

## 11.3 Maxsubmatrixsum

```
// Description: Calcula a maior soma de uma submatriz MxN de uma matriz lxc
// Complexidade: O(l*c)

const int MAX = 1010; // 10^6 + 10

int mat[MAX][MAX];

int maxSubmatrixSum(int l, int c, int M, int N) {
    int dp[l+1][c+1];

    f(i,0,l+1) {
        dp[i][0] = 0;
        dp[0][i] = 0;
    }

    f(i,1,l+1) {
        f(j,1,c+1) {
            dp[i][j] =   dp[i-1][j]
                       + dp[i][j-1]
                       - dp[i-1][j-1]
                       + mat[i][j];
        }
    }

    int ans = 0;
```

```
    f(i,M,l+1) {
        f(j,N,c+1) {
            int ponto =
                  dp[i][j]
                - dp[i-M][j]
                - dp[i][j-N]
                + dp[i-M][j-N];
            ans = max(ans, ponto);
        }
    }
    return ans;
}

void solve() {
    int l, c, M, N; cin >> l >> c >> M >> N;

    f(i,1,l+1) {
        f(j,1,c+1) {
            cin >> mat[i][j];
        }
    }

    int ans = maxSubmatrixSum(l, c, M, N);

    cout << ans << endl;
}
```

## 11.4 Max 2D Range Sum

```
// Maximum Sum
// O(n^3) 1D DP + greedy (Kadane's) solution, 0.000s in UVa

#include <bits/stdc++.h>
using namespace std;

#define f(i,s,e)     for(int i=s;i<e;i++)
#define MAX_n 110

int A[MAX_n][MAX_n];

int maxMatrixSum(vector<vector<int>> mat) {

    int n = mat.size();
    int m = mat[0].size();

    f(i,0,n) {
        f(j,0,m) {
            if (j > 0)
                mat[i][j] += mat[i][j - 1];
        }
    }

    int maxSum = INT_MIN;
    f(l,0,m)  {
        f(r,l,m) {
            vector<int> sum(n, 0);
            f(row,0,n) {
                sum[row] = mat[row][r] - (l > 0 ? mat[row][l - 1] : 0);
            }
            int maxSubRect = sum[0];
            f(i,1,n) {
                if (sum[i - 1] > 0)
                    sum[i] += sum[i - 1];
                maxSubRect = max(maxSubRect, sum[i]);
            }
            maxSum = max(maxSum, maxSubRect);
        }
    }

    return maxSum;
}
```

## 11.5 Potencia Matriz

```
// Description: Calcula a potencia de uma matriz quadrada A elevada a um
    expoente n

int MOD;
const int MAX_N = 2;

struct Matrix { int mat[MAX_N][MAX_N]; };

int mod(int a, int m) { return ((a%m)+m) % m; }

Matrix matMul(Matrix a, Matrix b) {
    Matrix ans;
    for (int i = 0; i < MAX_N; ++i)
    for (int j = 0; j < MAX_N; ++j)
    ans.mat[i][j] = 0;

    for (int i = 0; i < MAX_N; ++i)
        for (int k = 0; k < MAX_N; ++k) {
            if (a.mat[i][k] == 0) continue;
            for (int j = 0; j < MAX_N; ++j) {
                ans.mat[i][j] += mod(a.mat[i][k], MOD) * mod(b.mat[k][j],
    MOD);
                ans.mat[i][j] = mod(ans.mat[i][j], MOD);
                }
        }
    return ans;
}

Matrix matPow(Matrix base, int p) {
    Matrix ans;
    for (int i = 0; i < MAX_N; ++i)
    for (int j = 0; j < MAX_N; ++j)
    ans.mat[i][j] = (i == j);
    while (p) {
        if (p&1)
        ans = matMul(ans, base);
        base = matMul(base, base);
        p >>= 1;
    }
    return ans;
```

```
}

void solve() {

}
```

## 11.6 Verifica Se E Quadrado Magico

```
// Description: Verifica se uma matriz é um quadrado ámgico.
// Complexidade: O(n^2)

int isMagicSquare(vector<vi> mat, int n) {
    int i=0,j=0;
    int sumd1 = 0, sumd2=0;
    f(i,0,n) {
        sumd1 += mat[i][i];
        sumd2 += mat[i][n-1-i];
    }
    if(sumd1!=sumd2) return 0;

    int ans = 0;

    f(i,0,n) {
        int rowSum = 0, colSum = 0;
        f(j,0,n) {
            rowSum += mat[i][j];
            colSum += mat[j][i];
        }
        if (rowSum != colSum || colSum != sumd1) return 0;
        ans = rowSum;
    }
    return ans;
}
```

## 11.7 Verificer Se Retangulo Cabe Em Matriz Binaria

```
// Description: Verifica se um retangulo C X L cabe em uma matriz binaria N X M
// çComplexidade: O(N*M)
// OBS: comprimParaAltura[i] = maior comprimento de âretngulo de 1's com altura i que caiba na matriz
void histogram(int alturasHistograma[], int colunas, int comprimParaAltura[]) {
    int stack_top, width;
    stack<int> st;

    int i = 0;
    while (i < colunas) {
        if (st.empty() || alturasHistograma[st.top()] <= alturasHistograma[i]) {
            st.push(i++);
        } else {
            stack_top = alturasHistograma[st.top()];
            st.pop();
            width = i;

            if (!st.empty())
                width = i - st.top() - 1;

            if (comprimParaAltura[stack_top] < width)
                comprimParaAltura[stack_top] = width;
        }
    }

    while (!st.empty()) {
        stack_top = alturasHistograma[st.top()];
        st.pop();
        width = i;

        if (!st.empty())
            width = i - st.top() - 1;

        if (comprimParaAltura[stack_top] < width)
            comprimParaAltura[stack_top] = width;
    }
}

bool fits(int c, int l, int comprimParaAltura[], int maxRectSize) {
    return (c <= maxRectSize and l <= comprimParaAltura[c]) or (l <= maxRectSize and c <= comprimParaAltura[l]);
}

void solve() {

    int n, m; cin >> n >> m; // dimensioes da matriz

    int mat[n][m]; memset(mat, 0, sizeof(mat));

    char str[m];
    f(i,0,n) {
        cin >> str;
        f(j,0,m) {
            if (str[j] == '.')
                mat[i][j] = 1;
        }
    }

    int maxRectSize = min((int)500, max(n, m)); // ãdimenso maxima do retangulo (max(comprimentoMaximo, larguraMaxima))

    int comprimParaAltura[maxRectSize + 1];
    memset(comprimParaAltura, -1, sizeof(comprimParaAltura));

    int histogramaAux[m]; memset(histogramaAux, 0, sizeof(histogramaAux));

    f(i,0,n) {
        f(j,0,m) {
            histogramaAux[j] = (mat[i][j] ? 1 + histogramaAux[j] : 0);
        }
        histogram(histogramaAux, m, comprimParaAltura);
    }

    int comprimentoRetangulo, larguraRetangulo; cin >>
```

```
    comprimentoRetangulo >> larguraRetangulo;

    if(fits(comprimentoRetangulo, larguraRetangulo, comprimParaAltura, maxRectSize)) {
        /* retangulo de comprimento comprimentoRetangulo e largura larguraRetangulo cabe na matriz */
    }
}
```

# 12 Strings

## 12.1 Kmp

```
#include <bits/stdc++.h>

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of pat[] in txt[]
void KMPSearch(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while ((N - i) >= (M - j)) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Driver code
int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

## 12.2 Aro Corasick

```
// C++ program for implementation of Aho Corasick algorithm
// for string matching
using namespace std;
#include <bits/stdc++.h>

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 500;

// Maximum number of characters in input alphabet
const int MAXC = 26;
```

```
12
13  // OUTPUT FUNCTION IS IMPLEMENTED USING out[]
14  // Bit i in this mask is one if the word with index i
15  // appears when the machine enters this state.
16  int out[MAXS];
17
18  // FAILURE FUNCTION IS IMPLEMENTED USING f[]
19  int f[MAXS];
20
21  // GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
22  int g[MAXS][MAXC];
23
24  // Builds the string matching machine.
25  // arr - array of words. The index of each keyword is important:
26  //       "out[state] & (1 << i)" is > 0 if we just found word[i]
27  //       in the text.
28  // Returns the number of states that the built machine has.
29  // States are numbered 0 up to the return value - 1, inclusive.
30  int buildMatchingMachine(string arr[], int k)
31  {
32      // Initialize all values in output function as 0.
33      memset(out, 0, sizeof out);
34
35      // Initialize all values in goto function as -1.
36      memset(g, -1, sizeof g);
37
38      // Initially, we just have the 0 state
39      int states = 1;
40
41      // Construct values for goto function, i.e., fill g[][]
42      // This is same as building a Trie for arr[]
43      for (int i = 0; i < k; ++i)
44      {
45          const string &word = arr[i];
46          int currentState = 0;
47
48          // Insert all characters of current word in arr[]
49          for (int j = 0; j < word.size(); ++j)
50          {
51              int ch = word[j] - 'a';
52
53              // Allocate a new node (create a new state) if a
54              // node for ch doesn't exist.
55              if (g[currentState][ch] == -1)
56                  g[currentState][ch] = states++;
57
58              currentState = g[currentState][ch];
59          }
60
61          // Add current word in output function
62          out[currentState] |= (1 << i);
63      }
64
65      // For all characters which don't have an edge from
66      // root (or state 0) in Trie, add a goto edge to state
67      // 0 itself
68      for (int ch = 0; ch < MAXC; ++ch)
69          if (g[0][ch] == -1)
70              g[0][ch] = 0;
71
72      // Now, let's build the failure function
73
74      // Initialize values in fail function
75      memset(f, -1, sizeof f);
76
77      // Failure function is computed in breadth first order
78      // using a queue
79      queue<int> q;
80
81      // Iterate over every possible input
82      for (int ch = 0; ch < MAXC; ++ch)
83      {
84          // All nodes of depth 1 have failure function value
85          // as 0. For example, in above diagram we move to 0
86          // from states 1 and 3.
87          if (g[0][ch] != 0)
88          {
89              f[g[0][ch]] = 0;
90              q.push(g[0][ch]);
91          }
92      }
93
94      // Now queue has states 1 and 3
95      while (q.size())
96      {
97          // Remove the front state from queue
98          int state = q.front();
99          q.pop();
100
101         // For the removed state, find failure function for
102         // all those characters for which goto function is
103         // not defined.
104         for (int ch = 0; ch <= MAXC; ++ch)
105         {
106             // If goto function is defined for character 'ch'
107             // and 'state'
108             if (g[state][ch] != -1)
109             {
110                 // Find failure state of removed state
111                 int failure = f[state];
112
113                 // Find the deepest node labeled by proper
114                 // suffix of string from root to current
115                 // state.
116                 while (g[failure][ch] == -1)
117                     failure = f[failure];
118
119                 failure = g[failure][ch];
120                 f[g[state][ch]] = failure;
121
122                 // Merge output values
123                 out[g[state][ch]] |= out[failure];
124
125                 // Insert the next level node (of Trie) in Queue
```

```
                q.push(g[state][ch]);
            }
        }
    }

    return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.
// currentState - The current state of the machine. Must be between
//           0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine.
int findNextState(int currentState, char nextInput)
{
    int answer = currentState;
    int ch = nextInput - 'a';

    // If goto is not defined, use failure function
    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

// This function finds all occurrences of all array words
// in text.
void searchWords(string arr[], int k, string text)
{
    // Preprocess patterns.
    // Build machine with goto, failure and output functions
    buildMatchingMachine(arr, k);

    // Initialize current state
    int currentState = 0;

    // Traverse the text through the built machine to find
    // all occurrences of words in arr[]
    for (int i = 0; i < text.size(); ++i)
    {
        currentState = findNextState(currentState, text[i]);

        // If match not found, move to next state
        if (out[currentState] == 0)
            continue;

        // Match found, print all matching words of arr[]
        // using output function.
        for (int j = 0; j < k; ++j)
        {
            if (out[currentState] & (1 << j))
            {
                cout << "Word " << arr[j] << " appears from "
                     << i - arr[j].size() + 1 << " to " << i << endl;
            }
        }
    }
}

// Driver program to test above
int main()
{
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int k = sizeof(arr)/sizeof(arr[0]);

    searchWords(arr, k, text);

    return 0;
}
```

## 12.3 Calculadora Posfixo

```
// Description: Calculadora de expressoes posfixas
// Complexidade: O(n)
int posfixo(string s) {
    stack<int> st;
    for (char c : s) {
        if (isdigit(c)) {
            st.push(c - '0');
        } else {
            int b = st.top(); st.pop();
            int a = st.top(); st.pop();
            if (c == '+') st.push(a + b);
            if (c == '-') st.push(a - b);
            if (c == '*') st.push(a * b);
            if (c == '/') st.push(a / b);
        }
    }
    return st.top();
}
```

## 12.4 Chaves Colchetes Parenteses

```
// Description: Verifica se s tem uma êsequncia valida de {}, [] e ()
// Complexidade: O(n)
bool brackets(string s) {
    stack<char> st;

    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else {
            if (st.empty()) return false;
            if (c == ')' and st.top() != '(') return false;
            if (c == ']' and st.top() != '[') return false;
            if (c == '}' and st.top() != '{') return false;
            st.pop();
        }
    }

    return st.empty();
}
```

## 12.5 Infixo Para Posfixo

```cpp
// Description: Converte uma expressao matematica infixa para posfixa
// Complexidade: O(n)
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

char associativity(char c) {
    if (c == '^')
        return 'R';
    return 'L';
}

string infixToPostfix(string s) {
    stack<char> st;
    string result;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;

        else if (c == '(')
            st.push('(');

        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop(); // Pop '('
        }

        else {
            while (!st.empty() && prec(s[i]) < prec(st.top()) ||
                   !st.empty() && prec(s[i]) == prec(st.top()) &&
                   associativity(s[i]) == 'L') {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    return result;
}
```

## 12.6 Is Subsequence

```cpp
// Description: Verifica se a string s eh subsequencia da string t
// Complexidade Temporal: O(n)
// Complexidade Espacial: O(n)
bool isSubsequence(string& s, string& t) {
    queue<char> q;
    int cnt = 0;
    for (int i = 0; i < t.size(); i++) {
        q.push(t[i]);
    }
    int i = 0;
    while (!q.empty()) {
        if (s[i] == q.front()) {
            cnt++;
            i++;
        }
        q.pop();
    }

    return cnt == s.size();
}
```

## 12.7 Levenshtein

```cpp
#include <bits/stdc++.h>
using namespace std;
//a âdistncia Levenshtein ou âdistncia de çãedio entre dois "strings" é dada
//pelo únmero ímnimo de çõoperaes ánecessrias para transformar um string no outro.
//Entendemos por "çõoperaes" a çãinsero, çãdeleo ou çãsubstituio de um ácarcter.
int levenshteinDist(string word1, string word2) {
    int size1 = word1.size();
    int size2 = word2.size();
    int verif[size1 + 1][size2 + 1]; // Verification matrix i.e. 2D array which will store the calculated distance.

    // If one of the words has zero length, the distance is equal to the size of the other word.
    if (size1 == 0)
        return size2;
    if (size2 == 0)
        return size1;

    // Sets the first row and the first column of the verification matrix with the numerical order from 0 to the length of each word.
    for (int i = 0; i <= size1; i++)
        verif[i][0] = i;
    for (int j = 0; j <= size2; j++)
```

```
        verif[0][j] = j;

    // Verification step / matrix filling.
    for (int i = 1; i <= size1; i++) {
        for (int j = 1; j <= size2; j++) {
            // Sets the modification cost.
            // 0 means no modification (i.e. equal letters) and 1 means that a modification is needed (i.e. unequal letters).
            int cost = (word2[j - 1] == word1[i - 1]) ? 0 : 1;

            // Sets the current position of the matrix as the minimum value between a (deletion), b (insertion) and c (substitution).
            // a = the upper adjacent value plus 1: verif[i - 1][j] + 1
            // b = the left adjacent value plus 1: verif[i][j - 1] + 1
            // c = the upper left adjacent value plus the modification cost: verif[i - 1][j - 1] + cost
            verif[i][j] = min(
                min(verif[i - 1][j] + 1, verif[i][j - 1] + 1),
                verif[i - 1][j - 1] + cost
            );
        }
    }

    // The last position of the matrix will contain the Levenshtein distance.
    return verif[size1][size2];
}

int main() {
    string word1, word2;

    cout << "Please input the first word: " << endl;
    cin >> word1;
    cout << "Please input the second word: " << endl;
    cin >> word2;

    cout << "The number of modifications needed in order to make one word equal to the other is: " << levenshteinDist(word1, word2) << endl;

    system("pause");
    return 0;
}
```

## 12.8 Lexico E Sintatico

```
unordered_map<char, int> precedence = {{'|', 1}, {'.', 2}, {'>', 3}, {'<', 3}, {'=', 3}, {'#', 3}, {'+', 4}, {'-', 4}, {'*', 5}, {'/', 5}, {'^', 6}};
string ops = "+-*/^><=#.|";

bool isOp(char a) {
    return ops.find(a) != string::npos;
}

bool checkIfSyntaxError(const string &str) {
    int len = str.size();
    stack<char> pilha;
    for (int i = 0; i < len; i++) {
        if (isOp(str[i])) {
            if (i + 1 == len || i == 0) return true;
            if (!isalnum(str[i - 1]) and str[i - 1] != ')') return true;
            if (!isalnum(str[i + 1]) and str[i + 1] != '(') return true;
            while (!pilha.empty() and pilha.top() != '(' and precedence[pilha.top()] >= precedence[str[i]]) {
                pilha.pop();
            }
            pilha.push(str[i]);
        }
        else if (str[i] == '(') {
            if (i > 0 and (isalnum(str[i - 1]) || str[i - 1] == ')')) return true;
            pilha.push('(');
        }
        else if (str[i] == ')') {
            if (i > 0 and str[i - 1] == '(') return true;
            if (pilha.empty()) {
                return true;
            }
            while (!pilha.empty() and pilha.top() != '(') {
                pilha.pop();
                if (pilha.empty()) {
                    return true;
                }
            }
            if (!pilha.empty()) pilha.pop();
        }
        else if (isalnum(str[i])) {
            if (i > 0 and isalnum(str[i - 1]) and !isOp(str[i - 1]) and str[i - 1] != '(') return true;
        }
    }
    while (!pilha.empty()) {
        if (pilha.top() == '(') {
            return true;
        }
        pilha.pop();
    }
    return false;
}

bool checkIfLexicalError(const string &str) {
    for (char ch : str)
        if (!isalnum(ch) and !isOp(ch) and ch != '(' and ch != ')')
            return true;
    return false;
}

void solve() {
    string str; cin >> str;
    if (checkIfLexicalError(str))
        cout << "Lexical Error!" << endl;
    else if (checkIfSyntaxError(str))
        cout << "Syntax Error!" << endl;
}
```

## 12.9 Lexicograficamente Minima

```
// Descricao: Retorna a menor rotacao lexicografica de uma string.
// Complexidade: O(n * log(n)) onde n eh o tamanho da string
string minLexRotation(string str) {
    int n = str.length();

    string arr[n], concat = str + str;

    for (int i = 0; i < n; i++)
        arr[i] = concat.substr(i, n);

    sort(arr, arr+n);

    return arr[0];
}
```

## 12.10 Longest Common Substring

```
// Description: Encontra o comprimento da maior usbstring em comum entre 2 strings
// Complexidade Temporal: O(n * m)
// Complexidade Espacial: O(min(m,n))
int LCSubStr(string s, string t, int n, int m)
{
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    int ans = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s[i - 1] == t[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if (dp[i][j] > ans)
                    ans = dp[i][j];
            }
            else
                dp[i][j] = 0;
        }
    }
    return ans;
}

void solve() {
    string x, y; cin >> x >> y;
    cout << LCSubStr(x, y, x.size(), y.size()) << endl;
}
```

## 12.11 Lower Upper

```
// Description: çãFuno que transforma uma string em lowercase.
// Complexidade: O(n) onde n é o tamanho da string.
string to_lower(string a) {
   for (int i=0;i<(int)a.size();++i)
      if (a[i]>='A' && a[i]<='Z')
         a[i]+='a'-'A';
   return a;
}

// para checar se é lowercase: islower(c);

// Description: çãFuno que transforma uma string em uppercase.
// Complexidade: O(n) onde n é o tamanho da string.
string to_upper(string a) {
   for (int i=0;i<(int)a.size();++i)
      if (a[i]>='a' && a[i]<='z')
         a[i]-='a'-'A';
   return a;
}

// para checar se e uppercase: isupper(c);
```

## 12.12 Numeros E Char

```
char num_to_char(int num) { // 0 -> '0'
    return num + '0';
}

int char_to_num(char c) { // '0' -> 0
    return c - '0';
}

char int_to_ascii(int num) { // 97 -> 'a'
    return num;
}

int ascii_to_int(char c) { // 'a' -> 97
    return c;
}
```

## 12.13 Ocorrencias

```
// Description: çãFuno que retorna um vetor com as çõposies de todas as êocorrncias de uma substring em uma string.
// Complexidade: O(n * m) onde n é o tamanho da string e m é o tamanho da substring.
vector<int> ocorrencias(string str,string sub){
    vector<int> ret;
    int index = str.find(sub);
    while(index!=-1){
        ret.push_back(index);
        index = str.find(sub,index+1);
    }

    return ret;
}
```

## 12.14 Palindromo

```
// Descricao: Funcao que verifica se uma string eh um palindromo.
// Complexidade: O(n) onde n eh o tamanho da string.
bool isPalindrome(string str) {
    for (int i = 0; i < str.length() / 2; i++) {
```

```
        if (str[i] != str[str.length() - i - 1]) {
            return false;
        }
    }
    return true;
}
```

## 12.15 Permutacao

```
// Funcao para gerar todas as permutacoes de uma string
// Complexidade: O(n!)

void permute(string& s, int l, int r) {
    if (l == r)
        permutacoes.push_back(s);
    else {
        for (int i = l; i <= r; i++) {
            swap(s[l], s[i]);
            permute(s, l+1, r);
            swap(s[l], s[i]);
        }
    }
}

int main() {

    string str = "ABC";
    int n = str.length();
    permute(str, 0, n-1);
}
```

## 12.16 Remove Acento

```
// Descricao: Funcao que remove acentos de uma string.
// Complexidade: O(n * m) onde n eh o tamanho da string e m eh o tamanho do alfabeto com acento.
string removeAcentro(string str) {

    string comAcento = "áéíóúâêôãõà";
    string semAcento = "aeiouaeoaoa";

    for(int i = 0; i < str.size(); i++){
        for(int j = 0; j < comAcento.size(); j++){
            if(str[i] == comAcento[j]){
                str[i] = semAcento[j];
                break;
            }
        }
    }

    return str;
}
```

## 12.17 Split Cria

```
// Descricao: Funcao que divide uma string em um vetor de strings.
// Complexidade: O(n * m) onde n eh o tamanho da string e m eh o tamanho do delimitador.
vector<string> split(string s, string del = " ") {
    vector<string> retorno;
    int start, end = -1*del.size();
    do {
        start = end + del.size();
        end = s.find(del, start);
        retorno.push_back(s.substr(start, end - start));
    } while (end != -1);
    return retorno;
}
```

## 12.18 String Hashing

```
#include <bits/stdc++.h>
using namespace std;

struct Hash {
    const int p1 = 31, m1 = 1e9 + 7;
    const int p2 = 37, m2 = 1e9 + 9;
    int hash1 = 0, hash2 = 0;
    Hash(const string& s) {
        compute_hash1(s);
        compute_hash2(s);
    }

    void compute_hash1(const string& s) {
        long p_pow = 1;
        for(char ch: s) {
            hash1 = (hash1 + (ch + 1 - 'a') * p_pow) % m1;
            p_pow = (p_pow * p1) % m1;
        }
    }

    void compute_hash2(const string& s) {
        long p_pow = 1;
        for(char ch: s) {
            hash2 = (hash2 + (ch + 1 - 'a') * p_pow) % m2;
            p_pow = (p_pow * p2) % m2;
        }
    }

    // For two strings to be equal
    // they must have same hash1 and hash2
    bool operator==(const Hash& other) {
        return (hash1 == other.hash1 && hash2 == other.hash2);
    }
};

int main() {
    const string s = "geeksforgeeks";
    Hash h(s);
    cout << "Hash values of " << s << " are: ";
    cout << "(" << h.hash1 << ", " << h.hash2 << ")" << '\n';
    return 0;
```

```
}
```

# 13 Vector

## 13.1 Contar Menores Elementos A Direita

```
// Description: Conta quantos elementos menores que o elemento atual existem a direita do mesmo.
// Complexity: O(nlogn)

const int MAX = 100010; // Tamanho maximo do array de entrada

int ansArr[MAX]; // Array que armazena a resposta

void merge(pair<int, int> a[], int start, int mid, int end) {
    pair<int, int> f[mid - start + 1], s[end - mid];

    int n = mid - start + 1;
    int m = end - mid;

    for(int i = start; i <= mid; i++)
        f[i - start] = a[i];
    for(int i = mid + 1; i <= end; i++)
        s[i - mid - 1] = a[i];

    int i = 0, j = 0, k = start;
    int cnt = 0;

    while(i < n and j < m) {
        if (f[i].second <= s[j].second) {
            ansArr[f[i].first] += cnt;
            a[k++] = f[i++];
        } else {
            cnt++;
            a[k++] = s[j++];
        }
    }

    while(i < n) {
        ansArr[f[i].first] += cnt;
        a[k++] = f[i++];
    }

    while(j < m) {
        a[k++] = s[j++];
    }
}

void mergesort(pair<int, int> item[], int low, int high) {
    if (low >= high) return;

    int mid = (low + high) / 2;
    mergesort(item, low, mid);
    mergesort(item, mid + 1, high);
    merge(item, low, mid, high);
}

void solve() {
    int n; cin >> n;
    int arr[n];
    f(i,0,n) { cin >> arr[i]; }

    pair<int, int> a[n];
    memset(ansArr, 0, sizeof(ansArr));

    f(i,0,n) {
        a[i].second = arr[i];
        a[i].first = i;
    }

    mergesort(a, 0, n - 1);

    int ans = 0;
    f(i,0,n) { ans += ansArr[i]; }
    cout << ans << endl;
}
```

## 13.2 Contar Subarrays Somam K

```
// Descricao: Conta quantos subarrays de um vetor tem soma igual a k
// Complexidade: O(n)
int contarSomaSubarray(vector<int>& v, int k) {
    unordered_map<int, int> prevSum; // map to store the previous sum

    int ret = 0, currentSum = 0;

    for(int& num : v) {
        currentSum += num;

        if (currentSum == k) ret++; /// Se a soma atual for igual a k, encontramos um subarray

        if (prevSum.find(currentSum - k) != prevSum.end()) // se subarray com soma (currentSum - k) existir, sabe que [0:n] eh um subarray com soma k
            ret += (prevSum[currentSum - k]);

        prevSum[currentSum]++;
    }

    return ret;
}
```

## 13.3 Elemento Mais Frequente

```
#include <bits/stdc++.h>
using namespace std;

// Encontra o unico elemento mais frequente em um vetor
// Complexidade: O(n)
int maxFreq1(vector<int> v) {
    int res = 0;
```

```
    int count = 1;

    for(int i = 1; i < v.size(); i++) {

        if(v[i] == v[res])
            count++;
        else
            count--;

        if(count == 0) {
            res = i;
            count = 1;
        }
    }

    return v[res];
}

// Encontra os elemento mais frequente em um vetor
// Complexidade: O(n)
vector<int> maxFreqn(vector<int> v)
{
    unordered_map<int, int> hash;
    for (int i = 0; i < v.size(); i++)
        hash[v[i]]++;

    int max_count = 0, res = -1;
    for (auto i : hash) {
        if (max_count < i.second) {
            res = i.first;
            max_count = i.second;
        }
    }

    vector<int> ans;
    for (auto i : hash) {
        if (max_count == i.second) {
            ans.push_back(i.first);
        }
    }

    return ans;
}
```

## 13.4 K Maior Elemento

```
// Description: Encontra o ké-simo maior elemento de um vetor
// Complexidade: O(n)

int Partition(vector<int>& A, int l, int r) {
    int p = A[l];
    int m = l;
    for (int k = l+1; k <= r; ++k) {
        if (A[k] < p) {
            ++m;
            swap(A[k], A[m]);
        }
    }
    swap(A[l], A[m]);
    return m;
}

int RandPartition(vector<int>& A, int l, int r) {
    int p = l + rand() % (r-l+1);
    swap(A[l], A[p]);
    return Partition(A, l, r);
}

int QuickSelect(vector<int>& A, int l, int r, int k) {
    if (l == r) return A[l];
    int q = RandPartition(A, l, r);
    if (q+1 == k)
        return A[q];
    else if (q+1 > k)
        return QuickSelect(A, l, q-1, k);
    else
        return QuickSelect(A, q+1, r, k);
}

void solve() {
    vector<int> A = { 2, 8, 7, 1, 5, 4, 6, 3 };
    int k = 1;
    cout << QuickSelect(A, 0, A.size()-1, k) << endl;
}
```

## 13.5 Longest Common Subsequence

```
// Description: Encontra o tamanho da maior subsequencia comum entre duas
    strings
// Complexidade Temporal: O(n*m)
// Complexidade Espacial: O(m)
// Exemplo: "ABCDG" e "ADEB" => 2 ("AB")

int longestCommonSubsequence(string& text1, string& text2) {
    int n = text1.size();
    int m = text2.size();

    vector<int> prev(m + 1, 0), cur(m + 1, 0);

    for (int idx2 = 0; idx2 < m + 1; idx2++)
        cur[idx2] = 0;

    for (int idx1 = 1; idx1 < n + 1; idx1++) {
        for (int idx2 = 1; idx2 < m + 1; idx2++) {
            if (text1[idx1 - 1] == text2[idx2 - 1])
                cur[idx2] = 1 + prev[idx2 - 1];

            else
                cur[idx2]
                    = 0 + max(cur[idx2 - 1], prev[idx2]);
        }
        prev = cur;
    }

```

```
    return cur[m];
}
```

## 13.6 Maior Retangulo Em Histograma

```
// Calcula area do maior retangulo em um histograma
// Complexidade: O(n)
int maxHistogramRect(const vector<int>& hist) {
    stack<int> s;
    int n = hist.size();

    int ans = 0, tp, area_with_top;

    int i = 0;
    while (i < n) {

        if (s.empty() || hist[s.top()] <= hist[i])
            s.push(i++);

        else {
            tp = s.top(); s.pop();

            area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);

            if (ans < area_with_top)
                ans = area_with_top;
        }
    }

    while (!s.empty()) {
        tp = s.top();  s.pop();
        area_with_top = hist[tp] * (s.empty() ? i : i - s.top() - 1);

        if (ans < area_with_top)
            ans = area_with_top;
    }

    return ans;
}

void solve() {
    vector<int> hist = { 6, 2, 5, 4, 5, 1, 6 };
    cout << maxHistogramRect(hist) << endl;
}
```

## 13.7 Maior Sequencia Subsequente

```
// Maior sequencia subsequente
// {6, 2, 5, 1, 7, 4, 8, 3} => {2, 5, 7, 8}

int maiorCrescente(vector<int> v) {
    vector<int> lenght(v.size());
    for(int k=0; k<v.size(); k++) {
        lenght[k] = ;
        for(int i=0; i<k; i++) {
            if(v[i] < v[k]) {
                lenght[i] = max(lenght[k], lenght[i]+1)
            }
        }
    }
    return lenght.back();
}
```

## 13.8 Maior Subsequencia Comum

```
int s1[MAXN], s2[MAXN], tab[MAXN][MAXN];

// Description: Retorna o tamanho da maior êsubsequncia comum entre s1 e
    s2
// Complexidade: O(n*m)
int lcs(int a, int b){

    if(tab[a][b]>=0) return tab[a][b];
    if(a==0 or b==0) return tab[a][b]=0;
    if(s1[a]==s2[b]) return 1 + lcs(a-1, b-1);
    return tab[a][b] = max(lcs(a-1, b), lcs(a, b-1));
}

void solve() {

    s1 = {1, 3, 2, 5, 4, 2, 3, 4, 5};
    s2 = {1, 2, 3, 4, 5};
    int n = s1.size(), m = s2.size();
    memset(tab, -1, sizeof(tab));
    cout << lcs(n, m) << endl; // 5
}
```

## 13.9 Maior Subsequência Crescente

```
// Retorna o tamanho da maior êsubsequncia crescente de v
// Complexidade: O(n log(n))
int maiorSubCrescSize(vector<int> &v) {

    vector<int> pilha;
    for (int i = 0; i < v.size(); i++) {
        auto it = lower_bound(pilha.begin(), pilha.end(), v[i]);
        if (it == pilha.end())
            pilha.push_back(v[i]);
        else
            *it = v[i];
    }

    return pilha.size();
}

// Retorna a maior êsubsequncia crescente de v
// Complexidade: O(n log(n))
vector<int> maiorSubCresc(vector<int> &v) {

    vector<int> pilha, resp;
    int pos[MAXN], pai[MAXN];
    for (int i = 0; i < v.size(); i++) {
```

```
        auto it = lower_bound(pilha.begin(), pilha.end(), v[i]);
        int p = it - pilha.begin();
        if (it == pilha.end())
            pilha.PB(v[i]);
        else
            *it = x;
        pos[p] = i;
        if (p == 0)
            pai[i] = -1; // seu pai áser -1
        else
            pai[i] = pos[p - 1];
    }

    int p = pos[pilha.size() - 1];
    while (p >= 0) {
        resp.PB(v[p]);
        p = pai[p];
    }
    reverse(resp.begin(), resp.end());

    return resp;
}

void solve() {
    vector<int> v = {1, 3, 2, 5, 4, 2, 3, 4, 5};
    cout << maiorSubCrescSize(v) << endl // 5
    /*************************************/
    vector<int> ans = maiorSubCresc(v); // {1,2,3,4,5}
}
```

## 13.10 Maior Triangulo Em Histograma

```
// Calcula o maior âtringulo em um histograma
// Complexidade: O(n)
int maiorTrianguloEmHistograma(const vector<int>& histograma) {

    int n = histograma.size();
    vector<int> esquerda(n), direita(n);

    esquerda[0] = 1;
    f(i,1,n) {
        esquerda[i] = min(histograma[i], esquerda[i - 1] + 1);
    }

    direita[n - 1] = 1;
    rf(i,n-1,0) {
        direita[i] = min(histograma[i], direita[i + 1] + 1);
    }

    int ans = 0;
    f(i,0,n) {
        ans = max(ans, min(esquerda[i], direita[i]));
    }

    return ans;

}
```

## 13.11 Remove Repetitive

```
// Remove repetitive elements from a vector
// Complexity: O(n)
vector<int> removeRepetitive(const vector<int>& vec) {

    unordered_set<int> s;
    s.reserve(vec.size());

    vector<int> ans;

    for (int num : vec) {
        if (s.insert(num).second)
            v.push_back(num);
    }

    return ans;
}

void solve() {
    vector<int> v = {1, 3, 2, 5, 4, 2, 3, 4, 5};
    vector<int> ans = removeRepetitive(v); // {1, 3, 2, 5, 4}
}
```

## 13.12 Soma Maxima Sequencial

```
// Description: Soma maxima sequencial de um vetor
// Complexidade: O(n)
int max_sum(vector<int> s) {

    int ans = 0, maior = 0;

    for(int i = 0; i < s.size(); i++) {
        maior = max(0,maior+s[i]);
        ans = max(resp,maior);
    }

    return ans;
}

void solve() {
    vector<int> v = {1,-3,5,-1,2,-1};
    cout << max_sum(v) << endl; // 6 = {5,-1,2}
}
```

## 13.13 Subset Sum

```
// Description: Verifica se algum subset dentro do array soma igual a sum
// Complexidade Temporal: O(sum * n)
// Complexidade Espacial: O(sum * n)

bool isSubsetSum(vi set, int n, int sum) {
    bool subset[n + 1][sum + 1];

    for (int i = 0; i <= n; i++)
        subset[i][0] = true;
```

```
    for (int i = 1; i <= sum; i++)
        subset[0][i] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1])
                subset[i][j] = subset[i - 1][j];
            if (j >= set[i - 1])
                subset[i][j]
                    = subset[i - 1][j]
                      || subset[i - 1][j - set[i - 1]];
        }
    }

    return subset[n][sum];
}
```

### 13.14 Troco

```
// Description: Retorna o menor únmero de moedas para formar um valor n
// Complexidade: O(n*m)
vector<int> troco(vector<int> coins, int n) {
    int first[n];
    value[0] = 0;
    for(int x=1; x<=n; x++) {
        value[x] = INF;
        for(auto c : coins) {
            if(x-c => 0 and value[x-c] + 1 < value[x]) {
                value[x] = value[x-c]+1;
                first[x] = c;
            }
        }
    }

    vector<int> ans;
    while(n>0) {
        ans.push_back(first[n]);
        n -= first[n];
    }
    return ans;
}

void solve() {
    vector<int> coins = {1, 3, 4};
    vector<int> ans = troco(coins, 6); // {3,3}
}
```

## 14 Outros

### 14.1 Dp

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_gm = 30; // up to 20 garments at most and 20 models/garment
const int MAX_M = 210; // maximum budget is 200

int M, C, price[MAX_gm][MAX_gm]; // price[g (<= 20)][k (<= 20)]
int memo[MAX_gm][MAX_M];         // TOP-DOWN: dp table [g (< 20)][money
    (<= 200)]

int dp(int g, int money) {

    if (money < 0) return -1e9;
    if (g == C) return M - money;
    if (memo[g][money] != -1)
        return memo[g][money]; // avaliar linha g com dinheiro money (cada
     caso pensavel)
    int ans = -1;
    for (int k = 1; k <= price[g][0]; ++k)
        ans = max(ans, dp(g + 1, money - price[g][k]));
    return memo[g][money] = ans;
}

int main() {
    int TC;
    scanf("%d", &TC);
    while (TC--)
    {
        scanf("%d %d", &M, &C);
        for (int g = 0; g < C; ++g)
        {
            scanf("%d", &price[g][0]); // store k in price[g][0]
            for (int k = 1; k <= price[g][0]; ++k)
                scanf("%d", &price[g][k]);
        }
        memset(memo, -1, sizeof memo); // TOP-DOWN: init memo
        if (dp(0, M) < 0)
            printf("no solution\n"); // start the top-down DP
        else
            printf("%d\n", dp(0, M));
    }
    return 0;
}
```

### 14.2 Binario

```
// Descicao: conversao de decimal para binario
// Complexidade: O(logn) onde n eh o numero decimal
string decimal_to_binary(int dec) {
    string binary = "";
    while (dec > 0) {
        int bit = dec % 2;
        binary = to_string(bit) + binary;
        dec /= 2;
    }
    return binary;
}

// Descicao: conversao de binario para decimal
// Complexidade: O(logn) onde n eh o numero binario
```

```cpp
int binary_to_decimal(string binary) {
    int dec = 0;
    int power = 0;
    for (int i = binary.length() - 1; i >= 0; i--) {
        int bit = binary[i] - '0';
        dec += bit * pow(2, power);
        power++;
    }
    return dec;
}
```

## 14.3 Binary Search

```cpp
// Description: çãImplementao do algoritmo de busca ábinria.
// Complexidade: O(logn) onde n eh o tamanho do vetor
int BinarySearch(<vector>int arr, int x){
    int k = 0;
    int n = arr.size();

    for (int b = n/2; b >= 1; b /= 2) {
        while (k+b < n && arr[k+b] <= x) k += b;
    }
    if (arr[k] == x) {
        return k;
    }
}
```

## 14.4 Fibonacci

```cpp
vector<int> memo(MAX, -1);

// Descricao: Funcao que retorna o n-esimo termo da sequencia de Fibonacci
     utilizando programacao dinamica.
// Complexidade: O(n) onde n eh o termo desejado
int fibPD(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fibPD(n - 1) + fibPD(n - 2);
}
```

## 14.5 Horario

```cpp
// Descricao: Funcoes para converter entre horas e segundos.
// Complexidade: O(1)
int cts(int h, int m, int s) {
    int total = (h * 3600) + (m * 60) + s;
    return total;
}

tuple<int, int, int> cth(int total_seconds) {
    int h = total_seconds / 3600;
    int m = (total_seconds % 3600) / 60;
    int s = total_seconds % 60;
    return make_tuple(h, m, s);
}
```

## 14.6 Intervalos

```cpp
// Conta quantos intervalos nao tem overlap ([a,b] e [b,c] nao geram)

bool cmp(const pair<int,int>& p1, const pair<int,int>& p2) {
    if(p1.second != p2.second) return p1.second < p2.second;
    return p1.first < p2.first;
}

int countNonOverlappingIntervals(vector<pair<int,int>> intervals) {
    sort(all(intervals), cmp);
    int firstTermino = intervals[0].second;
    int ans = 1;
    f(i,1,intervals.size()) {
        if(intervals[i].first >= firstTermino) {
            ans++;
            firstTermino = intervals[i].second;
        }
    }

    return ans;
}
```