

**INSTITUTO FEDERAL**

Sul de Minas Gerais

Campus Poços de Caldas



Notebook para as Maratonas de Programação  
Equipe SK Teletom

# Sumário

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Template</b>  | <b>6</b> |
| <b>2</b> | <b>Matemática Computacional</b>                            | <b>7</b> |
| 2.1      | Geometria Básica . . . . .                                 | 7        |
| 2.2      | Geometria Computacional . . . . .                          | 9        |
| 2.3      | Primos . . . . .   | 12       |
| 2.3.1    | Primo rápido – $O(\sqrt{n})$ . . . . .                     | 12       |
| 2.3.2    | Crivo de Erastótenes . . . . .                             | 12       |
| 2.3.3    | Fatoração de primos . . . . .                              | 12       |
| 2.3.4    | Números primos menores que 100 . . . . .                   | 13       |
| 2.4      | Algoritmos de Euclides . . . . .                           | 13       |
| 2.4.1    | Maior divisor comum – GCD . . . . .                        | 13       |
| 2.4.2    | Menor divisor comum – LCM . . . . .                        | 13       |
| 2.4.3    | Maior múltiplo comum – MMC . . . . .                       | 13       |
| 2.5      | Operadores binários . . . . .                              | 13       |
| 2.5.1    | OR nos Bits (   ) . . . . .                                | 13       |
| 2.5.2    | AND nos bits ( & ) . . . . .                               | 13       |
| 2.5.3    | XOR nos bits ( ^ ) . . . . .                               | 13       |
| 2.5.4    | Shift Esquerdo ( « ) . . . . .                             | 13       |
| 2.5.5    | Shift Direito ( » ) . . . . .                              | 14       |
| 2.6      | Manipulação de bits . . . . .                              | 14       |
| 2.7      | Checar se um dado bit está ligado . . . . .                | 14       |
| 2.8      | Extraír o bit menos signifiante . . . . .                  | 14       |
| 2.9      | Contar o número de bits iguais a 1 . . . . .               | 14       |
| 2.10     | Checar se um número é potência de 2 . . . . .              | 14       |
| 2.11     | Ligar um bit em um número . . . . .                        | 14       |
| 2.12     | Desligar o bit . . . . .                                   | 14       |
| 2.13     | Módulo de $2^i$ usando bitwise . . . . .                   | 15       |
| 2.14     | Divisão de números inteiros com resto negativo . . . . .   | 15       |
| 2.15     | Condição existência e classificação de triângulo . . . . . | 15       |
| 2.16     | Comparação entre 2 valores tipo Double . . . . .           | 16       |
| 2.17     | Arredondamento para cima . . . . .                         | 16       |
| 2.18     | Número de casas decimais de um número . . . . .            | 16       |
| 2.19     | Zerar conteúdo de um array 2d . . . . .                    | 16       |
| 2.20     | Zerar conteúdo de um array 1d . . . . .                    | 16       |
| 2.21     | Cuidado para divisão de dois floats ou double . . . . .    | 16       |
| 2.22     | Conversão inteiro para hexadecimal . . . . .               | 17       |
| 2.23     | Adicionar notação científica . . . . .                     | 17       |
| 2.24     | Adicionar casas decimais fixas . . . . .                   | 17       |
| 2.25     | Volume do cilindro . . . . .                               | 17       |
| 2.26     | Área Total . . . . .                                       | 17       |
| 2.27     | Somatório de Feynman . . . . .                             | 17       |
| 2.28     | Ponto de máximo e mínimo em uma parábola . . . . .         | 17       |
| 2.29     | Somatório de um intervalo [a,b] inclusivo . . . . .        | 17       |
| 2.30     | Distância entre 2 pontos . . . . .                         | 17       |
| 2.31     | Conversão cartesiano para polar . . . . .                  | 18       |
| 2.32     | Conversão polar para cartesiano . . . . .                  | 18       |
| 2.33     | Número de permutações de um conjunto . . . . .             | 18       |

|          |  |           |
|----------|--|-----------|
| 2.34     | Número de combinações de um conjunto . . . . .                           | 18        |
| 2.35     | Tricks do cmath . . . . .  | 18        |
| 2.36     | Máximo entre dois números . . . . .                                      | 18        |
| 2.37     | Mínimo entre dois números . . . . .                                      | 18        |
| 2.38     | $A^b \bmod p$ . . . . .  | 18        |
| 2.39     | $n! \bmod p$ . . . . .   | 19        |
| 2.40     | Digital Root . . . . .   | 19        |
| 2.40.1   | Digital Root de um número X . . . . .                                    | 19        |
| 2.40.2   | K-ésimo número com Digital Root igual a X . . . . .                      | 19        |
| <b>3</b> | <b>Strings</b> . . . . .   | <b>20</b> |
| 3.1      | Modificações . . . . .   | 20        |
| 3.1.1    | Dividir uma string de acordo com um token . . . . .                      | 20        |
| 3.1.2    | Apagar um intervalo de uma string . . . . .                              | 20        |
| 3.1.3    | Remover um caracter de toda a string . . . . .                           | 20        |
| 3.1.4    | Inverter String . . . . .  | 20        |
| 3.1.5    | Substring . . . . .  | 20        |
| 3.2      | Verificações . . . . .   | 20        |
| 3.2.1    | Verificar se uma string está vazia . . . . .                             | 20        |
| 3.2.2    | Verificar se caracter está entre [A-z] . . . . .                         | 20        |
| 3.3      | Conversões . . . . .   | 21        |
| 3.3.1    | String para int . . . . .  | 21        |
| 3.3.2    | String para long long . . . . .  | 21        |
| 3.3.3    | String para unsigned int . . . . .                                       | 21        |
| 3.3.4    | String para unsigned long long . . . . .                                 | 21        |
| 3.3.5    | Char para int . . . . .  | 21        |
| 3.3.6    | Int para String . . . . .  | 21        |
| 3.3.7    | Caracteres minúsculos . . . . .  | 21        |
| 3.3.8    | Caracteres maiúsculos . . . . .  | 21        |
| 3.4      | Apagar um intervalo de uma string . . . . .                              | 21        |
| 3.5      | Remover um caracter de toda a string . . . . .                           | 22        |
| 3.6      | Verificar se uma string está vazia . . . . .                             | 22        |
| 3.7      | Inverter String . . . . .  | 22        |
| 3.8      | Criar uma nova string a partir de um intervalo de outra string . . . . . | 22        |
| 3.9      | Verificar se caracter está entre [A-z] . . . . .                         | 22        |
| 3.10     | Busca [A-z] . . . . .  | 22        |
| 3.11     | Insert, Erase, Replace . . . . .   | 22        |
| 3.12     | String Streams . . . . .   | 22        |
| <b>4</b> | <b>Estruturas</b> . . . . .  | <b>23</b> |
| 4.1      | Verificar se elemento existe em um vetor . . . . .                       | 23        |
| 4.2      | Apagar elementos duplicados em um vetor . . . . .                        | 23        |
| 4.3      | Ordenar vector forma crescente . . . . .                                 | 23        |
| 4.4      | Ordenar vector forma crescente . . . . .                                 | 23        |
| 4.5      | Excluir primeiro elemento de um vetor . . . . .                          | 23        |
| 4.6      | Excluir último elemento de um vetor . . . . .                            | 23        |
| 4.7      | Alterar tamanho de um vector . . . . .                                   | 23        |
| 4.8      | Busca em um vetor . . . . .  | 23        |
| 4.9      | Busca em um vetor . . . . .  | 24        |
| 4.10     | Deque . . . . .  | 24        |
| 4.11     | Definição de um pair . . . . .   | 24        |

|          |  |           |
|----------|--|-----------|
| 4.12     | Leitura de um pair                                     | 24        |
| 4.13     | Utilizando pair de pair                                | 24        |
| 4.14     | Criando pair com dois valores                          | 24        |
| 4.15     | Fila   | 24        |
| 4.16     | Pilha  | 24        |
| 4.17     | SET  | 25        |
| 4.18     | Map  | 25        |
| 4.19     | For em Map   | 25        |
| 4.20     | Union-Find com PD                                      | 25        |
| 4.21     | Union-Find com PD que atualiza o tamanho dos conjuntos | 26        |
| 4.22     | Fila de prioridades                                    | 26        |
| 4.23     | Árvore de Segmentos                                    | 26        |
| 4.24     | Árvore de Indexação Binária (BIT)                      | 27        |
| 4.25     | Lazy Propagation                                       | 28        |
| 4.26     | Sort em structs  | 29        |
| <b>5</b> | <b>Grafos</b>  | <b>30</b> |
| 5.1      | Representações de um Grafo                             | 30        |
| 5.1.1    | Matriz de Adjacência                                   | 30        |
| 5.1.2    | Lista de Adjacência                                    | 30        |
| 5.2      | Lista de Arestas                                       | 32        |
| 5.3      | Algoritmos   | 32        |
| 5.3.1    | DFS(Busca em profundidade)                             | 32        |
| 5.3.2    | BFS(Busca em largura)                                  | 32        |
| 5.3.3    | Dijkstra - Caminho Mínimo entre dois pontos            | 33        |
| 5.3.4    | Kruskal - Árvore Geradora Mínima                       | 35        |
| 5.3.5    | Prim - Árvore Geradora Mínima                          | 36        |
| 5.3.6    | Ordenação Topológica                                   | 37        |
| 5.3.7    | Floyd-Warshall - Menor Caminho                         | 38        |
| 5.3.8    | LCA - Menor Ancestral Comum                            | 40        |
| 5.3.9    | Caminho Euleriano                                      | 41        |
| 5.3.10   | Grafos bipartidos                                      | 41        |
| <b>6</b> | <b>Programação dinâmica</b>                            | <b>43</b> |
| 6.1      | Problema da mochila                                    | 43        |
| 6.2      | Problema do troco                                      | 44        |
| 6.2.1    | Problema do corte de hastes                            | 44        |
| 6.2.2    | Dado valor e as moedas existe troco possível?          | 44        |
| 6.3      | Calculadora Quebrada                                   | 45        |
| 6.3.1    | Mínimo de moedas para troco                            | 47        |
| 6.4      | Contagem de inversões                                  | 47        |
| 6.5      | Maior Subsequência Comum                               | 48        |
| 6.6      | Maior Subsequência crescente                           | 49        |
| 6.7      | Soma máxima em um intervalo                            | 50        |
| 6.8      | Vertex Cover   | 50        |
| <b>7</b> | <b>Outros</b>  | <b>52</b> |
| 7.1      | Compilar programa em Java no CMD                       | 52        |
| 7.2      | Compilar programa em C++ no CMD                        | 52        |
| 7.3      | Tabela ASCII   | 52        |
| 7.4      | C++ Limits   | 53        |

|      |  |    |
|------|--|----|
| 7.5  | Estruturas de dados C++ . . . . .                                    | 54 |
| 7.6  | Conversão para números romanos . . . . .                             | 54 |
| 7.7  | Antes e depois de cristo . . . . .                                   | 54 |
| 7.8  | Problemas que envolvem horário . . . . .                             | 54 |
| 7.9  | Ano bissexto . . . . .   | 54 |
| 7.10 | Ano normal . . . . .   | 54 |
| 7.11 | Dias de cada mês . . . . .   | 54 |
| 7.12 | Número de letras no alfabeto . . . . .                               | 55 |
| 7.13 | Forma alternativa para escrita de nome para tipos de dados . . . . . | 55 |
| 7.14 | Inicializar vetor com valor predefinido . . . . .                    | 55 |
| 7.15 | Operações para modificar sequências . . . . .                        | 55 |
| 7.16 | Permutações . . . . .  | 55 |
| 7.17 | Gerar números aleatórios . . . . .                                   | 55 |
| 7.18 | Pesquisa binária . . . . .   | 55 |

# 1 Template

```
1 #include <bits/stdc++.h>
2
3 #define ll long long
4 #define ull unsigned long long
5 #define ld long double
6 #define INF 0x3f3f3f3f
7 #define f first
8 #define s second
9 #define pb push_back
10 #define mp make_pair
11 #define l length()
12 #define forn(i, n) for ( int i = 0; i < (n); ++i )
13 #define forn(x, i, n) for ( int i = (x); i < (n); ++i )
14
15 using namespace std;
16
17 int main() {
18     ios::sync_with_stdio(false); cin.tie(NULL);
19
20     return 0;
21 }
```

## 2 Matemática Computacional

### 2.1 Geometria Básica

Vamos trabalhar com Geometria Euclidiana em 2D, em especial estaremos lidando com Geometria Analítica. Começaremos supondo que os estudantes entendem o conceito de ponto, reta e polígono simples. Agora vejamos alguns teoremas importantes:

#### Soma dos ângulos internos de um triângulo:

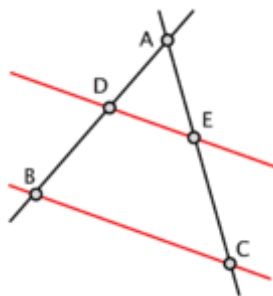
A soma dos ângulos internos de triângulo é  $180^\circ$

#### Soma dos ângulos internos de um polígono:

Vejamos que podemos triangular um polígono simples, basta escolhermos um dos pontos e ligá-los a todos os outros vemos então que teremos  $N - 2$  triângulos, logo a soma dos ângulos internos será  $(N-2) 180$  graus.

#### Teorema de Tales:

Quando duas retas transversais cortam um feixe de retas paralelas, as medidas dos segmentos delimitados nas transversais são proporcionais. Por exemplo, usando a figura abaixo:



Então pelo teorema de Tales temos que:

$$\frac{AD}{AB} = \frac{AE}{AC}$$

$$\frac{AD}{AE} = \frac{AB}{AC}$$

$$\frac{AD}{DB} = \frac{AE}{EC}$$

#### Teorema de Pitágoras:

Um triângulo é retângulo se e somente se a soma dos quadrados de seus catetos (lados menores) for igual ao quadrado de sua hipotenusa (lado maior).

Na geometria analítica, nós consideramos que nossas figuras estão em um plano com dois eixos ortogonais que se cruzam na origem, esses eixos nos permitem definir coordenadas para os

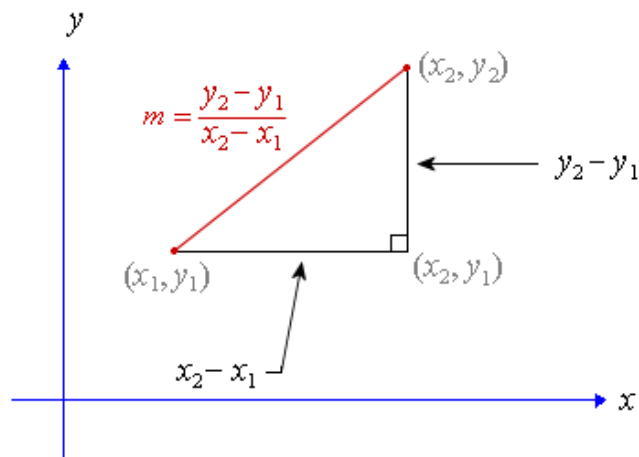
pontos dos planos e transformar um problema de geometria em um problema de álgebra, que computadores conseguem resolver.

**Ponto:**

ponto em geometria analítica é apenas um par de números, suas coordenadas, uma horizontal e uma vertical.

**Distância entre dois pontos:**

distância entre dois pontos em geometria analítica pode facilmente ser descoberta usando o teorema de Pitágoras. Seja o primeiro ponto P1 (de coordenadas  $x_1$  e  $y_1$ ) e o segundo P2 (de coordenadas  $x_2$  e  $y_2$ ), então vemos que se construirmos um ponto P3 de coordenadas  $x_2$  e  $y_1$ , teremos um triângulo retângulo, cuja hipotenusa é a distância entre P1 e P2, como mostra a figura abaixo:



Assim temos que a distância será:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

**Reta:**

Uma reta pode ser representada de várias formas, seguem aqui duas delas:

$$a \cdot x + b = y$$

$$a \cdot x + b \cdot y = c$$

A primeira nos permite escrever uma coordenada dos pontos na reta em função da outra, sendo que  $a$  é chamado de coeficiente angular da reta (podemos ver facilmente que ele é a tangente do ângulo que a reta faz com o eixo horizontal), note que  $a$  e  $b$  são únicos. Já a segunda tem infinitas triplas  $a$ ,  $b$  e  $c$  possíveis, porém se fixarmos o valor de um dos 3 os outros dois estão determinados, além disso essa forma tem a vantagem de nos permitir criar retas verticais, pois na forma anterior tais retas teriam  $a = \text{infinito}$ .

**Círculo**

O círculo é o lugar geométrico dos pontos equidistantes a um dado ponto, chamamos esse ponto de centro e essa distância de raio. Dessa forma, vemos que se as coordenadas do centro são  $x_c$  e  $y_c$ , então todos os pontos obedecem a seguinte equação:



$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

## 2.2 Geometria Computacional

Essa classe de problemas em geral eles contêm códigos bastante complicados e podem ter certos problemas que geralmente não encontramos em outros tipos de questão, como por exemplo problemas com a precisão do float. Trabalharemos em 2D.

### Ponto e Vetor:

Pontos são geralmente representados por dois números reais, e são **análogos a um vetor indo da origem para onde o ponto fica**. Assim podemos criar um ponto de 3 formas: Pair:

```
1 #include <bits/stdc++.h>
2 #define x first
3 #define y second
4
5 using namespace std;
6
7 typedef pair <double, double> point;
8
9 point sum(point a, point b){
10     point ret;
11     ret.x = a.x + b.x;
12     ret.y = a.y + b.y;
13     return ret;
14 }
15
16 point neg(point a){
17     point ret;
18     ret.x = -a.x;
19     ret.y = -a.y;
20     return ret;
21 }
```

### Objeto:

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 struct point{
6     double x, double y;
7     point(){}
8     point(double _x, double _y){
9         x = _x;
10        y = _y;
11    }
12    point operator+(const point &oth){
13        return point(x + oth.x, y + oth.y);
14    }
15    point operator-(const point &oth){
16        return point(x - oth.x, y - oth.y);
17    }
18 };
```

### Complex:

```
1 #include <bits/stdc++.h>
2 #define x real()
3 #define y imag()
4
5 using namespace std;
6 typedef complex <double> point;
```

O tipo complex de C++ já tem soma, subtração e multiplicação definidos para ele, porém deve-se tomar cuidado com esse tipo, pois o complex de int não é bem definido na std e vai variar com o compilador.

Agora exploremos algumas funções do complex antes de avançarmos:

**real(p):** Retorna a parte real do número complexo p.

**imag(p):** Retorna a parte imaginária do número complexo.

**abs(p):** Retorna o comprimento do vetor (o valor absoluto de p)

**sin(p), cos(p), tan(p):** São as funções trigonométricas no nosso número complexo.

**arg(p):** Diz o ângulo que o vetor faz com a horizontal.

**conj(p):** Retorna o conjugado do número complexo

### Linhas:

São simplesmente pares de pontos (não importando como você definiu o seu ponto).

### Círculo

Um círculo pode ser definido por seu centro e seu raio, desta forma temos que podemos definir um círculo como um par de ponto e double.

Vejamos agora funções úteis nos problemas de geometria:

**Produto Escalar** O produto escalar é um dos dois tipos de produtos entre dois vetores e tem boas aplicações como veremos adiante. Vale lembrar que:

$$\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos(\angle AB)$$

```
1 double dot(point a, point b){
2   return a.x*b.x + a.y*b.y;
3 }
```

Caso você tenha implementado o ponto com a complex também podemos definir o produto escalar da seguinte forma:

```
1 double dot(point a, point b){
2   return (a*conj(b)).x;
3 }
```

Supondo a e b não nulos, temos que o produto escalar deles vai ser menor que zero se eles tiverem um ângulo maior que 90° entre eles, igual a 0 se forem perpendiculares e maior que zero se formarem um ângulo agudo.

### Produto vetorial:

O produto vetorial geralmente tomaria dois vetores e nos retornaria um terceiro, porém aqui apenas nos retornaremos com a magnitude do vetor retornado. Matematicamente temos:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \sin(\angle AB)$$

E o código para calcular o produto vetorial é:

```
1 double cross(point a, point b){
2   return a.x*b.y - a.y*b.x;
3 }
```

Ou ainda, se você estiver usando complex:

```
1 double cross(point a, point b){
2   return (a*conj(b)).y;
3 }
```

O produto vetorial nos dá a área do paralelogramo com lados a e b (com sinal) e nos permite saber se o ângulo entre a e b é menor que 180 (se a área for menor que 0) , igual a 180 (se a área for igual a 0, no caso os vetores são paralelos), ou maior que 180 (se a área for maior que 180). Agora por fim vejamos como essas funções nos permitem calcular quantias geometricamente importantes:

### Distância entre dois pontos:

A distância entre dois pontos é simplesmente o módulo do vetor que liga esses pontos, dessa forma basta subtrairmos um ponto do outro e retornarmos o módulo da resultante:

```
1 double dist(point a, point b){
2     point c = a - b;
3     return sqrt(c.x*c.x + c.y*c.y);
4 }
```

Usando a complex podemos escrever:

```
1 double dist(point a, point b){
2     return abs(a - b);
3 }
```

### Distância entre ponto e reta:

A distância de um ponto para uma linha é igual a distância do ponto a um ponto qualquer da linha vezes o ângulo que esse vetor faz com a linha, assim podemos usar o produto vetorial para conseguir essa distância:

```
1 double dist(point a, line b){
2     double crs = cross(point(a - b.first), point(b.second - b.first));
3     return abs(crs/dist(b.first, b.second));
4 }
```

### Área do Polígono

Uma fórmula conhecida para a área de polígonos é a shoelace formula (muito usada em geometria analítica). Assim, sendo um polígono um vector de pontos ordenados tais que dois pontos adjacentes são uma aresta:

```
1 double area(vector <point> p){
2     double ret = 0;
3     for(int i = 2; i < p.size(); ++i){
4         ret += cross(p[i] - p[0], p[i - 1] - p[0]) / 2;
5     }
6     return abs(ret);
7 }
```

### CCW

A última função interessante que veremos toma 3 números e retorna se eles formam um ângulo convexo ou côncavo.

```
1 double ccw(point a, point b, point c){
2     double ret = cross(b - a, c - b);
3     return ret < 0;
4 }
```

Note que em alguns juízes, e na OBI, muitas vezes erros de precisão podem levar um algoritmo correto a receber um **WA** (resposta errada), nesse caso não se deve usar a **complex** e sim um **pair** de **long long int** ou uma **struct**, e todas as operações que envolverem igualar duas frações,  $a/b$  e  $c/d$ , devem ser checados do seguinte modo:

```
1 if(a*d == b*c){
2     //seu código aqui
3 }
```

## 2.3 Primos

### 2.3.1 Primo rápido – $O(\sqrt{n})$

```
1 bool e_primo(int x) {
2     if (x == 1) return 0;
3     //note que se o número for 2 ele não entra no loop, comportamento desejado
4     for (int i = 2; i*i <= x; ++i) {
5         if (x % i == 0) { //se o resto de x por i for 0, então i divide x
6             return 0;
7         }
8     }
9     return 1;
10 }
```

### 2.3.2 Crivo de Erastótenes

Dados  $N$  e  $Q$ , com ambos menores que  $10^6$ , teremos  $Q$  inteiros  $a$ , menores que  $N$ , e devemos responder para cada um deles se ele é primo.

```
1 // dados N e Q, com ambos menores que 10^6, teremos Q inteiros a, menores que N, e devemos responder
  para cada um deles se ele é primo
2
3 bool e_composto[1000010];
4
5 void crivo(int n) {
6     // 1 não composto, mas o vetor na verdade guarda os números que não são primos
7     e_composto[1] = 1;
8     for (int i = 2; i <= n; ++i) {
9         if (!e_composto[i]) {
10             for (int j = 2; j*i <= n; ++j) {
11                 e_composto[i*j] = 1;
12             }
13         }
14     }
15     return;
16 }
17
18 int main() {
19     int N, Q, a;
20     cin >> N >> Q;
21     crivo(N); // Complexidade  $O(n \cdot \log(\log(n)))$ 
22     for (int i = 0; i < Q; ++i) { // Complexidade  $O(Q)$ 
23         cin >> a;
24         // Se composto falso, então primo, caso contrário composto.
25         cout << !e_composto[a] << "\n";
26     }
27     return 0;
28 }
29 }
```

### 2.3.3 Fatoração de primos

A fatoração de números primos transforma um número grande em um produto de primos. Por exemplo, 5733, a fatoração pode transformá-lo em  $3 \times 3 \times 7 \times 7 \times 13$ .

```
1 vector<int> factors(int n) {
2     vector<int> f;
3     for (int x = 2; x*x <= n; x++) {
4         while (n%x == 0) {
5             f.push_back(x);
6             n /= x;
7         }
8     }
9     if (n > 1) f.push_back(n);
10    return f;
11 }
```

### 2.3.4 Números primos menores que 100

// there are 25 numbers  
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,  
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

## 2.4 Algoritmos de Euclides

### 2.4.1 Maior divisor comum – GCD

```
1 int gcd(int a, int b)
2 {
3     if (b==0) return a;
4     else return gcd(b, a%b);
5 }
```

### 2.4.2 Menor divisor comum – LCM

```
1 int lcm(int a, int b)
2 {
3     return a*b/gcd(a,b);
4 }
```

### 2.4.3 Maior múltiplo comum – MMC

```
1 int mdc(int a, int b){
2     return (b == 0 ? a : mdc(b, a%b)); //b == 0 ? Caso sim, retorne a, caso não, retorne mdc(b, a%b)
3 }
4
5 int mmc = (a*b)/mdc(a,b);
```

## 2.5 Operadores binários

### 2.5.1 OR nos Bits ( | )

```
1 // a = 10010; b = 01110; a|b = 11110
```

### 2.5.2 AND nos bits (&)

```
1 // a = 10110; b = 10011; a&b = 10010
```

### 2.5.3 XOR nos bits (^)

```
1 // a = 10110; b = 10011; a^b = 00101
```

### 2.5.4 Shift Esquerdo ( << )

```
1 // a = 1; a = a << 8; a = 256, que em binário é 100000000
```

### 2.5.5 Shift Direito ( » )

```
1 // b = 260; b >>= 3; b = 32, que em binário é 100000
```

## 2.6 Manipulação de bits

### 2.7 Checar se um dado bit está ligado

```
1 // Note que uma potência de 2 tem sempre apenas um bit igual a 1, dessa forma se queremos saber se o
  // bit i está igual a 1, precisamos apenas verificar se o and dele e de 2^i é diferente de 0.
2
3 bool is_set(int x, int i) {
4     bool ret = ((x & (1 << i)) != 0);
5     return ret;
6 }
```

### 2.8 Extrair o bit menos significativo

```
1 // O bit menos significativo de um número menor bit de um número igual a 1, chamamos ele de lsb.
  // Exemplo:
2 // lsb(20) = lsb(10100) = 100 = 4
3
4 int lsb(int x) {
5     return x & -x;
6 }
```

### 2.9 Contar o número de bits iguais a 1

```
1 int count_bits(int x) {
2     int ret = 0;
3     while (x != 0) {
4         ++ret;
5         x = x & -x;
6     }
7     return ret;
8 }
```

### 2.10 Checar se um número é potência de 2

```
1 bool is_power_of_two(int x) {
2     if (x == 0) return 0;
3     return ((x & (x - 1)) == 0);
4 }
```

### 2.11 Ligar um bit em um número

```
1 // É bem simples, basta o número receber ele or 2 elevado ao bit que queremos setar
2 int x, i;
3 cin >> x >> i;
4 x |= (1 << i);
```

### 2.12 Desligar o bit

```

1 int x, i;
2 cin >> x >> i;
3 x |= (1 << i); // Primeiro eu ligo o bit, caso ele esteja desligado
4 x ^= (1 << i); // Depois desligo o bit

```

## 2.13 Módulo de $2^i$ usando bitwise

Para realizar o módulo de  $2^i$ , ( 2, 4, 8, 16 ...)

```

1 n % 2^i = n & (2^i - 1)

```

## 2.14 Divisão de números inteiros com resto negativo

Caso seja necessário dividir números inteiros com resto um resto que possivelmente negativo

```

1 int a, b, c;
2 int q, r;
3
4 cin >> a >> b;
5
6 q = a / b;
7 r = a % b;
8
9 if (r < 0) {
10     int c, d;
11     c = (a < 0) ? a * -1 : a;
12     d = (b < 0) ? b * -1 : b;
13
14     q = (c + d) / d;
15     r = (c - (q * d)) * -1;
16
17     q = (a * b > 0) ? q : q * -1;
18 }

```

## 2.15 Condição existência e classificação de triângulo

Para um triângulo existir, as três condições devem ser satisfeitas.

```

1 int max(int a, int b) {
2     return (a > b) ? a : b;
3 }
4 int min(int a, int b) {
5     return (a < b) ? a : b;
6 }
7
8 cin >> a >> b >> c;
9
10 // x > y > z
11 int x, y, z;
12
13 x = max(a, max(b, c));
14 z = min(a, min(b, c));
15 // a+b+c = soma total | -x - z = total - (maior+menor)
16 y = a + b + c - x - z;
17
18 if (x < (y + z)) {
19     if (x == y && y == z) {
20         cout << "Valido-Equilatero" << endl;
21     }
22     else if (x != y && x != z && y != z) {
23         cout << "Valido-Escaleno" << endl;
24     }
25     else {
26         cout << "Valido-Isoceles" << endl;
27     }
28     // pitagoras x=y+z
29     if ((x*x) == ((y*y) + (z*z))) {
30         cout << "Retangulo: S" << endl;
31     }
32     else {
33         cout << "Retangulo: N" << endl;
34     }
35 }

```

```

36 else {
37     cout << "Invalido" << endl;
38 }

```

## 2.16 Comparação entre 2 valores tipo Double

A comparação entre dois doubles pode retornar valores indejados. Por isso uma função especial para comparação pode ser necessária.

```

1 bool comparaDouble(double val1, double val2, string cmp) {
2     if (cmp == "=") {
3         return fabs(val1 - val2) < EPSILON;
4     }
5     else if (cmp == "<=") {
6         if (fabs(val1 - val2) < EPSILON) {
7             return true;
8         }
9         else {
10            return val1 <= val2;
11        }
12    }
13    else if (cmp == ">=") {
14        if (fabs(val1 - val2) < EPSILON) {
15            return true;
16        }
17        else {
18            return val1 >= val2;
19        }
20    }
21 }

```

## 2.17 Arredondamento para cima

```

1 ceil(numero)

```

## 2.18 Número de casas decimais de um número

```

1 ceil(log10(numero+1))

```

## 2.19 Zerar conteúdo de um array 2d

```

1 memset(array, 0, sizeof(array[0][0]) * n * n)

```

## 2.20 Zerar conteúdo de um array 1d

```

1 memset(array, 0, sizeof(array))

```

## 2.21 Cuidado para divisão de dois floats ou double

```

1 // 1/6=0
2 // 1.0/6.0=0,1666667

```



## 2.22 Conversão inteiro para hexadecimal

```
1 cout << hex << v  
2 // Considerando v um inteiro
```

## 2.23 Adicionar notação científica

```
1 cout << scientific => 5e+2;
```

## 2.24 Adicionar casas decimais fixas

```
1 cout << fixed << setprecision(2); => 5.00;
```

## 2.25 Volume do cilindro

$$\pi * r^2 * h$$

## 2.26 Área Total

$$A = Ab + Al = 2 * \pi * r * (2 + h)$$

$$Ab = 2 * \pi * r^2$$

$$Al = 2 * \pi * r * h$$

## 2.27 Somatório de Feynman

Para saber quantos quadrados diferentes existem em um quadriculado de N x N quadrados  
 $(n * (n + 1) * ((2 * n) + 1)) / 6$

## 2.28 Ponto de máximo e mínimo em uma parábola

Se  $a < 0$ , ponto de máximo

Se  $a > 0$ , ponto de mínimo

$$x = \frac{-b}{2a} \quad (1)$$

$$x = \frac{\Delta}{4a} \quad (2)$$

## 2.29 Somatório de um intervalo [a,b] inclusivo

$$((a + b) * (b - a + 1)) / 2$$

## 2.30 Distância entre 2 pontos

$$\sqrt{\text{pow}((xf - xi), 2) + \text{pow}((yf - yi), 2)}$$

### 2.31 Conversão cartesiano para polar

$$r = \sqrt{a^2 + b^2}$$

$$\Phi = \operatorname{tg}^{-1} b/a$$

### 2.32 Conversão polar para cartesiano

$$a = r \cos$$

$$b = r \sin$$

### 2.33 Número de permutações de um conjunto

```
1 // dados um grupo de 4 pessoas, de quantas formas podemos colocá-los em fila?
2 // P(n, k) = n!/(n-k)!
3 // k = número de elementos para permuta; n = número total de elementos
```

### 2.34 Número de combinações de um conjunto

```
1 // x = n!/(n-k)!k!
```

### 2.35 Tricks do cmath

```
1 // Quando um número for muito grande usar powl ao invés de pow. powl terá mais precisão
2 powl(a, b)
3 (int)round(p, (1.0/n)) // nth raiz de p
```

### 2.36 Máximo entre dois números

```
1 int max(int a, int b) { return a>b ? a:b; }
```

### 2.37 Mínimo entre dois números

```
1 int min(int a, int b) { return a<b ? a:b; }
```

### 2.38 $A^b \bmod p$

```
1 long powmod(long base, long exp, long modulus) {
2     base %= modulus;
3     long result = 1;
4
5     while (exp > 0) {
6         if (exp & 1) result = (result * base) % modulus;
7         base = (base * base) % modulus;
8         exp >>= 1;
9     }
10
11     return result;
12 }
```

## 2.39 $n! \bmod p$

```
1 int factmod (int n, int p) {
2     long long res = 1;
3     while (n > 1) {
4         res = (res * powmod (p-1, n/p, p)) % p;
5         for (int i=2; i<=n%p; ++i)
6             res=(res*i) %p;
7         n /= p;
8     }
9     return int (res % p);
10 }
```

## 2.40 Digital Root

Vamos chamar o digital root de  $x$  como  $S(x)$ .  $S(5) = 5$ ,  $S(38) = S(3 + 8 = 11) = S(1 + 1 = 2) = 2$ ,  $S(10) = S(1 + 0 = 1) = 1$ .

### 2.40.1 Digital Root de um número $X$

```
1 digitalRoot = x % 9
2 digitalRoot = ((x-1) % 9) + 1
```

### 2.40.2 $K$ -ésimo número com Digital Root igual a $X$

```
1  $x_k = ((k - 1) * 9) + x$ 
```

## 3 Strings

### 3.1 Modificações

#### 3.1.1 Dividir uma string de acordo com um token

```
1 std::string s = "scott>=tiger>=mushroom";
2 std::string delimiter = ">=";
3 size_t pos = 0;
4 std::string token;
5 while ((pos = s.find(delimiter)) != std::string::npos) {
6     token = s.substr(0, pos);
7     std::cout << token << std::endl;
8     s.erase(0, pos + delimiter.length());
9 }
10 std::cout << s << std::endl;
```

#### 3.1.2 Apagar um intervalo de uma string

```
1 n.erase(pos_inicio, pos_fim);
2 // pos_inicio e pos_fim são inteiros que representam posições
```

#### 3.1.3 Remover um caracter de toda a string

```
1 n.erase(remove(n.begin(), n.end(), caracter_a_ser_removido), n.end());
2 // caracter_a_ser_removido representa uma variável do tipo char, com o caracter ser removido da string
```

#### 3.1.4 Inverter String

```
1 reverse(str1.begin(), str1.end());
```

#### 3.1.5 Substring

```
1 substr(pos, len) // (Posição inicial, Quantidade de caracteres para pegar)
2 string str1 = line.substr(0, meio);
3 string str1 = line.substr(1); // Pega tudo do primeiro caracter em diante
```

### 3.2 Verificações

#### 3.2.1 Verificar se uma string está vazia

```
1 n.empty() // Retorna true ou false
```

#### 3.2.2 Verificar se caracter está entre [A-z]

```
1 (line[i] >= 65 && line[i] <= 90) || (line[i] >= 97 && line[i] <= 122)
```

## 3.3 Conversões

### 3.3.1 String para int

```
1 stoi(string, 0, 10)
```

### 3.3.2 String para long long

```
1 stoll(string, 0, 10)
```

### 3.3.3 String para unsigned int

```
1 stoul(string, 0, 10)
```

### 3.3.4 String para unsigned long long

```
1 stoull(string, 0, 10)
```

### 3.3.5 Char para int

```
1 var_char - 48 ou ((var_char - '0') % 48)
```

### 3.3.6 Int para String

```
1 int a = 10;  
2 stringstream ss;  
3 ss << a;  
4 string str = ss.str();
```

### 3.3.7 Caracteres minúsculos

```
1 tolower(char)
```

### 3.3.8 Caracteres maiúsculos

```
1 toupper(char)
```

## 3.4 Apagar um intervalo de uma string

```
1 n.erase(pos_início, pos_fim);  
2 // pos_início e pos_fim são inteiros que representam posições
```

### 3.5 Remover um caracter de toda a string

```
1 n.erase(remove(n.begin(), n.end(), caracter_a_ser_removido), n.end());
2 // caracter_a_ser_removido representa uma variável do tipo char, com o caracter ser removido da string
```

### 3.6 Verificar se uma string está vazia

```
1 n.empty() // Retorna true ou false
```

### 3.7 Inverter String

```
1 reverse(str1.begin(), str1.end());
```

### 3.8 Criar uma nova string a partir de um intervalo de outra string

```
1 substr(pos, len) // (Posição inicial, Quantidade de caracteres para pegar)
2 string str1 = line.substr(0, meio);
3 string str1 = line.substr(1); // Pega tudo do primeiro caracter em diante
```

### 3.9 Verificar se caracter está entre [A-z]

```
1 (line[i] >= 65 && line[i] <= 90) || (line[i] >= 97 && line[i] <= 122)
```

### 3.10 Busca [A-z]

```
1 unsigned int find(const string &s2, unsigned int pos1 = 0);
2 unsigned int rfind(const string &s2, unsigned int pos1 = end);
3 unsigned int find_first_of(const string &s2, unsigned int pos1 = 0);
4 unsigned int find_last_of(const string &s2, unsigned int pos1 = end);
5 unsigned int find_first_not_of(const string &s2, unsigned int pos1 = 0);
6 unsigned int find_last_not_of(const string &s2, unsigned int pos1 = end);
```

### 3.11 Insert, Erase, Replace

```
1 string& insert(unsigned int pos1, const string &s2);
2 string& insert(unsigned int pos1, unsigned int repetitions, char c);
3 string& erase(unsigned int pos = 0, unsigned int len = npos);
4 string& replace(unsigned int pos1, unsigned int len1, const string &s2);
5 string& replace(unsigned int pos1, unsigned int len1, unsigned int repetitions, char c);
```

### 3.12 String Streams

```
1 stringstream s1;
2 int i = 22;
3 s1 << "Hello world! " << i;
4 cout << s1.str() << endl;
```

## 4 Estruturas

### 4.1 Verificar se elemento existe em um vetor

```
1 // O código abaixo verifica se o número 1 existe no vetor uniao (retorna true se existe, falso se não existe)
2 find(uniao.begin(), uniao.end(), 1) != uniao.end()
```

### 4.2 Apagar elementos duplicados em um vetor

```
1 sort(uniao.begin(), uniao.end());
2 uniao.erase(unique(uniao.begin(), uniao.end()), uniao.end());
```

### 4.3 Ordenar vector forma crescente

```
1 sort(notas.begin(), notas.end());
```

### 4.4 Ordenar vector forma crescente

```
1 sort(p.begin(), p.end(), greater<int>());
```

### 4.5 Excluir primeiro elemento de um vetor

```
1 notas.erase(notas.begin());
```

### 4.6 Excluir último elemento de um vetor

```
1 notas.pop_back();
```

### 4.7 Alterar tamanho de um vector

```
1 V.resize(10); //Muda o tamanho do vector V para 10.
```

### 4.8 Busca em um vetor

```
1 iterator find(iterator first, iterator last, const T &value);
2 iterator find_if(iterator first, iterator last, const T &value, TestFunction test);
3 bool binary_search(iterator first, iterator last, const T &value);
4 bool binary_search(iterator first, iterator last, const T &value, LessThanOrEqualFunction comp);
```

## 4.9 Busca em um vetor

```
1 iterator find(iterator first, iterator last, const T &value);
2 iterator find_if(iterator first, iterator last, const T &value, TestFunction test);
3 bool binary_search(iterator first, iterator last, const T &value);
4 bool binary_search(iterator first, iterator last, const T &value, LessThanOrEqualFunction comp);
```

## 4.10 Deque

Deque array dinâmico que funciona como vector mas, tem os métodos `push_front()` e `pop_front()`.

## 4.11 Definição de um pair

```
1 pair<string, int> P
```

## 4.12 Leitura de um pair

```
1 cin>>P.first>>P.second
```

## 4.13 Utilizando pair de pair

```
1 pair<string, pair<double, double>> P; //Cria uma variável pair
2 P.first = "Joao"; //Nome de um aluno
3 P.second.first = 8.2; //Primeira nota do aluno
4 P.second.second = 10; //Segunda nota do aluno
```

## 4.14 Criando pair com dois valores

```
1 make_pair(a,b)
```

## 4.15 Fila

```
1 queue<int> fila; // Declaração da fila
2 fila.push(10); // Adicionando um elemento ao final da fila
3 fila.pop(); // Retira o primeiro elemento
4 fila.front(); // Retorna o primeiro elemento da fila
5 fila.empty(); // Verifica se a fila está vazia
```

## 4.16 Pilha

```
1 stack<int> pilha; // Declaração da pilha
2 pilha.push(10); // Adicionado um elemento ao topo da pilha
3 pilha.pop(); // Retira o elemento do topo da pilha
4 pilha.top(); // Retorna o elemento do topo da pilha
5 pilha.empty(); // Verifica se a pilha está vazia
```



## 4.17 SET

```
1 // busca, inserção e exclusão em complexidade O(log n)
2 // Mantém os elementos ordenados e não permite elementos duplicados
3 set<int> S; // Declaração do SET
4 S.insert(10); // Adiciona um elemento
5 if(S.find(3) != S.end()) // Se 3 está no conjunto
6 S.erase(10); // Apaga o elemento do SET
7 // clear(): Apaga todos os elementos.
8 // size(): Retorna a quantidade de elementos.
9 // begin(): Retorna um ponteiro para o início do set
10 // end(): Retorna um ponteiro para o final do set
```

## 4.18 Map

```
1 // Map é uma variação da estrutura set e sua implementação também é feita utilizando Red- Black
  Trees. A principal diferença entre um set e um map é o segundo armazena os conjuntos chave,
  valor e o primeiro apenas chave.
2 map<string, int> M; // Declaração
3 M.insert(make_pair("Alana", 10)); //Inserimos uma variável do tipo pair diretamente no map
4 M["Alana"] = 10; // Relacionando o valor 10 à chave "Alana"
5 if(M.find("Alana") != M.end()) //Se a chave "Alana" foi inserida no map
6 cout<<M["Alana"]<<"\n"; //Imprime o valor da chave "Alana", no caso, o valor 10.
7 M.erase("Alana"); //Apaga o elemento que possui a chave "Alana" do map
8 // clear(): Apaga todos os elementos.
9 // size(): Retorna a quantidade de elementos.
10 // begin(): Retorna um ponteiro para o início do map
11 // end(): Retorna um ponteiro para o final do map
```

## 4.19 For em Map

```
1 for (map<string, int>::iterator it=M.begin(); it!=M.end(); ++it){
2     cout << "(" << it->first << ", " << it->second << ") ";
3 }
```

## 4.20 Union-Find com PD

```
1 #define MAXN 100100
2
3 using namespace std;
4
5 int pai[MAXN], peso[MAXN];
6
7 // funções do Union-Find otimizadas
8 int find(int x){
9
10     if(pai[x]==x){
11         return x;
12     }
13     return pai[x]=find(pai[x]);
14 }
15
16 void join(int x, int y){
17
18     x=find(x);
19     y=find(y);
20
21     if(x==y){
22         return;
23     }
24
25     if(peso[x]<peso[y]){
26         pai[x]=y;
27     }
28     if(peso[x]>peso[y]){
29         pai[y]=x;
30     }
31
32     if(peso[x]==peso[y]){
33
34         pai[x]=y;
```

```

35     peso[y]++;
36 }
37 }
38 for(int i = 0; i < n; i++){
39     pai[i] = i;
40 }
41 memset(peso, 1, sizeof peso);

```

## 4.21 Union-Find com PD que atualiza o tamanho dos conjuntos

```

1 // declaro as variáveis que vou precisar
2 int pai[MAXN], peso[MAXN], qtd[MAXN];
3
4 // função join otimizada que atualiza o tamanho dos conjuntos
5 void join(int x, int y){ // ela recebe os elementos a serem unidos
6
7     // como só vamos unir os patriarcas só precisamos trabalhar com eles
8     // e fazemos caa um dos elementos receber o valor de sue patriarca
9     x=find(x);
10    y=find(y);
11
12    // agora x e y são os patriarcas dos conjuntos que desejamos unir
13
14    // se eles forem o mesmo patriarca
15    if(x==y){
16        return; // então os conjuntos já estão unidos e retornamos
17    }
18
19    // se o patriarca x tiver peso menor que o patriarca y
20    if(peso[x]<peso[y]){
21        pai[x]=y; // faço x ser filho do y
22        qtd[y]+=qtd[x]; // e atualizo o número de descendentes de y
23    }
24
25    // se o patriarca y tiver peso menor que o patriarca x
26    if(peso[x]>peso[y]){
27
28        pai[y]=x; // faço y ser filho de x
29        qtd[x]+=qtd[y]; // e a tualizo o número de descendentes de x
30    }
31
32    // se os dois tiverem o mesmo peso
33    if(peso[x]==peso[y]){
34
35        // escolho um para ser o novo patriarca
36        pai[x]=y;
37
38        // e guardo que seu peso aumentará um uma unidade
39        peso[y]++;
40
41        // e atualizo o seu número de descendentes
42        qtd[y]+=qtd[x];
43    }
44 }
45
46 for(int i = 0; i < n; i++){
47     pai[i] = i;
48 }

```

## 4.22 Fila de prioridades

```

1 priority_queue< pair<int, string> > pokemon;
2 pokemon.push(make_pair(poder, nome));
3 pokemon.top();
4 pokemon.pop();

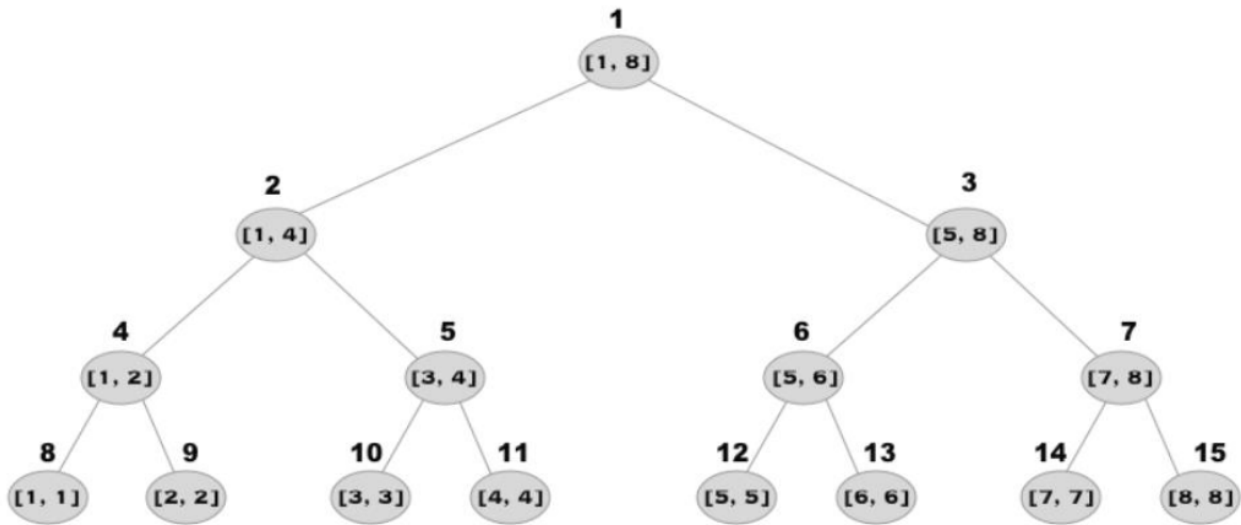
```

## 4.23 Árvore de Segmentos

Encontra o valor mínimo em um interval em  $O(\log n)$ .  $acao[i]$  representa o preço da ação de índice  $i$   $arvore[i]$  representa o valor contido no nó  $i$  da árvore.

Ou seja,  $arvore[i]$  contém o índice da ação mais barata no intervalo representado pelo nó  $i$  (no)

representa o nó que estamos na função recursiva o nó que estamos representa o segmento  $[i, j]$   
A função coloca altera o valor da ação de índice (posicao) para (novo\_valor) e altera a árvore de acordo com o necessário



```

1 void atualiza(int no, int i, int j, int posicao, int novo_valor){
2     // se tivermos i = j, temos i = posicao = j. Logo, estamos no nó mais baixo da árvore
3     if(i == j){
4         arvore[no] = i;
5         acao[posicao] = novo_valor;
6     }
7     else{
8         int esquerda = 2*no; // índice do filho da esquerda
9         int direita = 2*no + 1; // índice do filho da direita
10        int meio = (i + j)/2;
11        if(posicao <= meio) atualiza(esquerda, i, meio, posicao, novo_valor);
12        else atualiza(direita, meio+1, j, posicao, novo_valor);
13        if( acao[ arvore[esquerda] ] < acao[ arvore[direita] ] ) arvore[no] = arvore[esquerda];
14        else arvore[no] = arvore[direita];
15    }
16 }
17 int consulta(int no, int i, int j, int A, int B){
18     if(A <= i && j <= B){
19         return arvore[no];
20     }
21     if(i > B || A > j){
22         return -1;
23     }
24     int esquerda = 2*no;
25     int direita = 2*no + 1;
26     int meio = (i + j)/2;
27     int resposta_esquerda = consulta(esquerda, i, meio, A, B);
28     int resposta_direita = consulta(direita, meio+1, j, A, B);
29     if(resposta_esquerda == -1) return resposta_direita;
30     if(resposta_direita == -1) return resposta_esquerda;
31     if(acao[resposta_esquerda] < acao[resposta_direita]) return resposta_esquerda;
32     else return resposta_direita;
33 }

```

## 4.24 Árvore de Indexação Binária (BIT)

Dado um intervalo 1 a N. Permite adicionar valores aos elementos do intervalo e, efetuar o somatório de um intervalo intermediário entre 1 e N em  $O(\log n)$ .

```

1 // Consulta O(log n)
2 // Atualização O(n)
3 // n = tamanho do vetor (Colocar tamanho máximo)
4 int n = 100000;
5 int v[100000];
6 int arvore[100000+1];
7
8 int soma(int posicao){
9     int s = 0;

```

```

10 // vamos reduzindo x até acabarmos (quando chegamos a zero)
11 while(posicao > 0){
12     s += arvore[posicao]; // adicionamos o pedaço de árvore atual à soma
13     posicao -= (posicao & -posicao); // removemos o bit menos significativo
14 }
15 return s;
16 }
17 void atualiza(int valor, int posicao){ // v[x] = v
18     while(posicao <= n){ // nosso teto, que é quando vamos parar de rodar o algoritmo
19         arvore[posicao] += valor; // adicionamos v frutas a arvore[x], como devemos
20         posicao += (posicao & -posicao); // atualizamos o valor de x adicionado ele ao seu LSB
21     }
22 }
23
24 // Chamar antes de realizar consulta
25 void criaArvore(){
26     memset(arvore, 0, sizeof arvore);
27
28     for(int i = 0; i < n; i++){
29         atualiza(v[i], i+1);
30     }
31 }
32
33 // [l, r]
34 int consulta(int l, int r){
35     // Índices na árvore começam a partir de 1
36     l++, r++;
37     // Para efetuar consulta
38     l -= 1;
39
40     if(l < 0){
41         return soma(r);
42     } else {
43         return soma(r) - soma(l);
44     }
45 }
46
47 /*
48     criaArvore();
49     printf("%d", consulta(0,2));
50
51     atualiza(-v[pos-1], pos); // Zera o valor do vetor na "pos" e atualiza a árvore
52 */

```

## 4.25 Lazy Propagation

Você tem caixas N de frutas, numeradas de 1 a N, e duas possíveis operações.

Operação 1: adicionar v frutas a cada uma das caixas de índice entre a e b(inclusive).

Operação 2: responder quantas frutas existem nas caixas de índice entre a e b(inclusive).

Com Lazy Propagation, uma adaptação que se faz na Árvore de Segmentos que permite fazer ambas as operações em  $O(\log n)$  arvore[i] representa o valor contido no nó i da árvore.

Ou seja, se o nó i representa o intervalo [X, Y], arvore[i] representa a soma das caixas de X a Y lazy[i] representa a soma de todas as operações atrasadas que devemos fazer ao nó i (no) representa o nó que estamos na função recursiva o nó que estamos representa o segmento [i, j] vamos somar (valor) a cada um dos índices no intervalo [a, b].

```

1 void atualiza(int no, int i, int j, int a, int b, int valor){
2     int esquerda = 2*no; // índice do filho da esquerda
3     int direita = 2*no + 1; // índice do filho da direita
4     int meio = (i + j)/2;
5     if(lazy[no]){
6         arvore[no] += lazy[no]*(j - i + 1);
7         if(i != j){
8             lazy[direita] += lazy[no];
9             lazy[esquerda] += lazy[no];
10        }
11        lazy[no] = 0;
12    }
13    if(i > j || i > b || a > j) return;
14    if(a <= i && j <= b){
15        arvore[no] += valor*(j-i+1);
16        if(i != j){
17            lazy[direita] += valor;
18            lazy[esquerda] += valor;
19        }
20    }
21    else{

```

```

22 // atualizamos o filho da esquerda
23 atualiza(esquerda, i, meio, a, b, valor);
24 // atualizamos o filho da direita
25 atualiza(direita, meio+1, j, a, b, valor);
26 // atualizamos o nó que estamos
27 arvore[no] = arvore[esquerda] + arvore[direita];
28 }
29 }
30
31 // queremos saber a soma de todos os valores de índice no intervalo [A, B]
32 int consulta(int no, int i, int j, int a, int b){
33     int esquerda = 2*no; // índice do filho da esquerda
34     int direita = 2*no + 1; // índice do filho da direita
35     int meio = (i + j)/2;
36     if(lazy[no]){
37         arvore[no] += lazy[no]*(j - i + 1);
38         if(i != j){
39             lazy[direita] += lazy[no];
40             lazy[esquerda] += lazy[no];
41         }
42         lazy[no] = 0;
43     }
44     if(i > j || i > b || a > j) return 0;
45     if(a <= i && j <= b)
46         return arvore[no];
47     else{
48         int soma_esquerda = consulta(esquerda, i, meio, a, b);
49         int soma_direita = consulta(direita, meio+1, j, a, b);
50         return soma_esquerda + soma_direita;
51     }
52 }

```

## 4.26 Sort em structs

```

1 // Criar struct
2 typedef struct {
3     int moradores;
4     int gastoss;
5     int media;
6 } Imovel;
7 // Definir comparador para a struct
8 bool cmp(Imovel const & x, Imovel const & y){
9     if(x.media < y.media) {
10         return true;
11     } else {
12         return false;
13     }
14 }
15 // Efetuar o sort no main
16 Imovel imoveis[10];
17 sort(&imoveis[0], &imoveis[10], cmp);

```

## 5 Grafos

### 5.1 Representações de um Grafo

#### 5.1.1 Matriz de Adjacência

A matriz de adjacência consiste em saber, para cada possível par de vértices  $(u,v)$ , se existe ou não a aresta  $(u,v)$ .

Vamos guardar na posição  $(i,j)$  da matriz a informação sobre a aresta  $(i,j)$ . Podemos definir 0 para caso ela não existe e 1 para caso ela existe. No caso de termos um grafo com peso, podemos colocar o valor  $w$  na posição  $(i,j)$ , onde  $w$  é o peso da aresta  $(i,j)$ .

Essa representação é fácil de implementar, mas sua complexidade de espaço é muito grande, equivalendo a  $O(N^2)$ , onde  $N$  é o número de vértices.

Com relação ao tempo, podemos inserir e deletar uma aresta em  $O(1)$ , mas saber quais são os vizinhos de um vértice custa  $O(N)$ .

A representação do grafo fica da seguinte maneira:

|       | $V_1$ | $V_2$ | $V_3$ | $V_4$ | $V_5$ | $V_6$ |
|-------|-------|-------|-------|-------|-------|-------|
| $V_1$ | 0     | 1     | 1     | 0     | 0     | 1     |
| $V_2$ | 1     | 0     | 1     | 0     | 0     | 0     |
| $V_3$ | 1     | 1     | 0     | 1     | 0     | 0     |
| $V_4$ | 0     | 0     | 0     | 0     | 0     | 0     |
| $V_5$ | 0     | 0     | 0     | 0     | 0     | 0     |
| $V_6$ | 1     | 0     | 0     | 0     | 0     | 0     |

Implementação em C++:

```
1 memset(grafo, 0, sizeof(grafo[0][0]) * 10 * 10)
2 int grafo[10][10];
3
4 grafo[1][2] = grafo[2][1] = 1;
5 grafo[1][3] = grafo[3][1] = 1;
6 grafo[1][6] = grafo[6][1] = 1;
7 grafo[2][3] = grafo[3][2] = 1;
8 grafo[3][4] = grafo[4][3] = 1;
```

#### 5.1.2 Lista de Adjacência

A lista de adjacência se baseia em guardar, para cada vértice, quais são os seus vizinhos, ou, de uma maneira geral, guardar as arestas que partem desse vértice.

O uso da lista de adjacência pode ser complicado de implementar e debugar para pessoas iniciantes, mas acaba sendo a representação mais usada por pessoas experientes. Isso se deve ao fato de a lista de adjacência possuir complexidade  $O(1)$  para inserir novas arestas e um tempo otimizado para consultas num único vértice.

A representação do grafo por lista de adjacência fica da seguinte maneira:

| Vértice  | Vizinhos  |
|----------|-----------|
| <i>1</i> | {2, 3, 6} |
| <i>2</i> | {1, 3}    |
| <i>3</i> | {1, 2}    |
| <i>4</i> | {3}       |
| <i>5</i> | {}        |
| <i>6</i> | {1}       |

Implementação em C++:

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4 // 1 – Grafo sem pesos
5 map<int, vector<int>> > adj_map;
6 // 2 – Grafo sem pesos (Usar de preferencialmente essa)
7 vector<int> adj[10];
8 // 3 – Grafo com pesos
9 vector<pair<int, int>> adj_peso[10];
10
11 // Adicionar aresta usando representao 1
12 // Supondo que grafo seja no direcionado
13 void addEdge(map<int, vector<int>> > adj_map, int u, int v){
14     adj_map[u].push_back(v);
15     adj_map[v].push_back(u);
16 }
17 // Adicionar aresta usando representao 2
18 // Supondo que grafo seja no direcionado
19 void addEdge(vector<int> adj[], int u, int v)
20 {
21     adj[u].push_back(v);
22     adj[v].push_back(u);
23 }
24
25 // Adicionar aresta usando representao 3
26 // Supondo que grafo seja no direcionado
27 void addEdge(vector<pair<int, int>> adj[], int u, int v, int w)
28 {
29     adj[u].push_back(make_pair(v, w));
30     adj[v].push_back(make_pair(v, w));
31 }
32
33
34 int main(){
35     // 1
36     addEdge(adj_map, 0, 1);
37     addEdge(adj_map, 0, 2);
38     addEdge(adj_map, 2, 3);
39
40     // 2
41     addEdge(adj, 0, 1);
42     addEdge(adj, 0, 2);
43     addEdge(adj, 2, 3);
44
45     // 3
46     addEdge(adj_peso, 0, 1, 10);
47     addEdge(adj_peso, 0, 1, 5);
48     addEdge(adj_peso, 0, 1, 15);
49
50     return 0;
51 }

```

## 5.2 Lista de Arestas

```
1 struct t_aresta{
2     int dis;
3     int x, y;
4 };
5
6 t_aresta aresta[MAXM];
```

## 5.3 Algoritmos

### 5.3.1 DFS(Busca em profundidade)

Como o próprio nome já sugere, o algoritmo começa em um nó raiz e explora tanto quanto possível cada um dos seus ramos, antes de retroceder.

```
1 void addEdge(vector<int> adj[], int u, int v, int w)
2 {
3     adj[u].pb(mp(v, w));
4     adj[v].pb(mp(v, w));
5 }
6
7 // DFS não leva em consideração pesos dos grafos, por conta disso,
8 // procurar usar representação do grafo que não precisa de peso.
9
10 // Função para realizar DFS recursivamente
11 // no grafo a partir do vértice u.
12 void DFSUtil(int u, vector<int> adj[],
13              vector<bool> &visited)
14 {
15     visited[u] = true;
16     cout << u << " ";
17     for (int i=0; i<adj[u].size(); i++)
18         if (visited[adj[u][i]] == false)
19             DFSUtil(adj[u][i], adj, visited);
20 }
21
22 // Realiza DFSUtil() em todos
23 // os vértices não visitados.
24 void DFS(vector<int> adj[], int V)
25 {
26     vector<bool> visited(V, false);
27     for (int u=0; u<V; u++)
28         if (visited[u] == false)
29             DFSUtil(u, adj, visited);
30 }
31
32
33 int main(){
34     int V = 5;
35     // lista de adjacência
36     vector<int> adj[V];
37
38     addEdge(adj, 0, 1);
39     addEdge(adj, 0, 4);
40     addEdge(adj, 1, 2);
41     addEdge(adj, 1, 3);
42     addEdge(adj, 1, 4);
43     addEdge(adj, 2, 3);
44     addEdge(adj, 3, 4);
45     DFS(adj, V);
46     return 0;
47 }
```

### 5.3.2 BFS(Busca em largura)

```
1 // Função utilitária para adicionar arestas
2 // em um grafo não direcionado.
3 // u = origem
4 // v = destino
5 void addEdge(vector<int> adj[], int u, int v)
6 {
7     adj[u].pb(v);
8     adj[v].pb(u);
9 }
10
```



```

11 // BFS não leva em consideração pesos dos grafos, por conta disso,
12 // usar representação do grafo que não precisa de peso.
13 void BFS(vector<int> adj[], int s, int vertices) {
14     // Marca todos os vértices como não visitado
15     vector<bool> visited(vertices, false);
16     for(int i = 0; i < vertices; i++)
17         visited[i] = false;
18
19     // Cria uma fila para BFS
20     list<int> queue;
21
22     // Marca o vértice atual como visitado, e adiciona à fila
23     visited[s] = true;
24     queue.push_back(s);
25
26     // 'i' será usado para obter todos os
27     // vértices adjacentes de um vértice
28     vector<int>::iterator i;
29
30     while(!queue.empty())
31     {
32         // tira um vértice da lista e printa ele
33         s = queue.front();
34         cout << s << " ";
35         queue.pop_front();
36
37         // pega todos os vértices adjancetes do vértice
38         // que saiu da fila. Se o adjacente não foi visitado,
39         // então marca como visitado, e enfilera.
40         for (i = adj[s].begin(); i != adj[s].end(); ++i)
41         {
42             if (!visited[*i])
43             {
44                 visited[*i] = true;
45                 queue.push_back(*i);
46             }
47         }
48     }
49 }
50
51 int main(){
52     int vertices = 5;
53     vector<int> adj[vertices];
54
55     addEdge(adj, 0, 1);
56     addEdge(adj, 0, 4);
57     addEdge(adj, 1, 2);
58     addEdge(adj, 1, 3);
59     addEdge(adj, 1, 4);
60     addEdge(adj, 2, 3);
61     addEdge(adj, 3, 4);
62     BFS(adj, 2, vertices);
63
64     return 0;
65 }

```

### 5.3.3 Dijkstra - Caminho Mínimo entre dois pontos

Algoritmo 1: "Lista de Adjacência"

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 #define INFINITO 10000000
6
7 void addEdge(vector<pair<int, int> > adj[], int v1, int v2, int custo)
8 {
9     adj[v1].push_back(make_pair(v2, custo));
10 }
11
12 int dijkstra(vector<pair<int, int> > adj[], int orig, int dest, int vertices)
13 {
14     // vetor de distâncias
15     int dist[vertices];
16
17     /*
18     vetor de visitados serve para caso o vértice já tenha sido
19     expandido (visitado), não expandir mais
20     */
21     int visitados[vertices];
22
23     // fila de prioridades de pair (distancia, vértice)

```

```

24     priority_queue < pair<int, int>,
25                   vector<pair<int, int> >, greater<pair<int, int> > > pq;
26
27     // inicia o vetor de distâncias e visitados
28     for(int i = 0; i < vertices; i++)
29     {
30         dist[i] = INFINITO;
31         visitados[i] = false;
32     }
33
34     // a distância de orig para orig é 0
35     dist[orig] = 0;
36
37     // insere na fila
38     pq.push(make_pair(dist[orig], orig));
39
40     // loop do algoritmo
41     while(!pq.empty())
42     {
43         pair<int, int> p = pq.top(); // extrai o par do topo
44         int u = p.second; // obtém o vértice do par
45         pq.pop(); // remove da fila
46
47         // verifica se o vértice não foi expandido
48         if(visitados[u] == false)
49         {
50             // marca como visitado
51             visitados[u] = true;
52
53             vector<pair<int, int> >::iterator it;
54
55             // percorre os vértices "v" adjacentes de "u"
56             for(it = adj[u].begin(); it != adj[u].end(); it++)
57             {
58                 // obtém o vértice adjacente e o custo da aresta
59                 int v = it->first;
60                 int custo_aresta = it->second;
61
62                 // relaxamento (u, v)
63                 if(dist[v] > (dist[u] + custo_aresta))
64                 {
65                     // atualiza a distância de "v" e insere na fila
66                     dist[v] = dist[u] + custo_aresta;
67                     pq.push(make_pair(dist[v], v));
68                 }
69             }
70         }
71     }
72
73     // retorna a distância mínima até o destino
74     return dist[dest];
75 }
76
77 int main()
78 {
79     int vertices = 10;
80     vector<pair<int, int> > adj[vertices];
81
82     addEdge(adj, 0, 1, 4);
83     addEdge(adj, 0, 2, 2);
84     addEdge(adj, 0, 3, 5);
85     addEdge(adj, 1, 4, 1);
86     addEdge(adj, 2, 1, 1);
87     addEdge(adj, 2, 3, 2);
88     addEdge(adj, 2, 4, 1);
89     addEdge(adj, 3, 4, 1);
90
91     cout << dijkstra(adj, 1, 3, vertices) << endl;
92
93     return 0;
94 }

```

## Algoritmo 2: "Matriz de Adjacência"

```

1 #define MAX 999999999
2 #define max 501
3
4 using namespace std;
5
6 int g[max][max], vertices;
7
8 int dijkstra(int origem, int destino) {
9     int minimo, atual;
10    int passou[max], pred[max], custo[max];
11    for (int i=1; i<=vertices; i++) {
12        pred[i]=-1;

```

```

13     custo[i]=MAX;
14     passou[i]=0;
15 }
16 custo[origem] = 0;
17 atual = origem;
18 while (atual != destino) {
19     for (int i=1; i<=vertices; i++) {
20         if (custo[i] > custo[atual] + g[atual][i]) {
21             custo[i] = custo[atual] + g[atual][i];
22         }
23     }
24     minimo = MAX;
25     passou[atual]=1;
26     for (int i=1; i<=vertices; i++) {
27         if ((custo[i]<minimo) && (!passou[i])) {
28             minimo = custo[i];
29             atual = i;
30         }
31     }
32     // caso nao consiga ir a lugar algum saindo da origem
33     if (minimo == MAX) {
34         return MAX;
35     }
36 }
37 return custo[destino];
38 }

```

### 5.3.4 Kruskal - Árvore Geradora Mínima

Conjunto de arestas com peso mínimo que ligam todo o grafo. Para este algoritmo representar grafo utilizando lista de arestas.

```

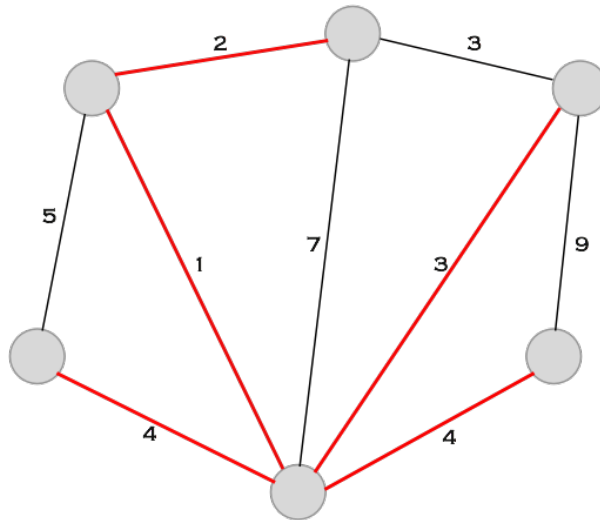
1 struct t_aresta{
2     int dis;
3     int x, y;
4 };
5 bool comp(t_aresta a, t_aresta b){ return a.dis < b.dis; }
6 //
7 #define MAXN 50500
8 #define MAXM 200200
9
10 int n, m; // número de vértices e arestas
11 t_aresta aresta[MAXM];
12 // para o union find
13 int pai[MAXN];
14 int peso[MAXN];
15 // a árvore
16 t_aresta mst[MAXM];
17 //
18 // funções do union find
19 int find(int x){
20     if(pai[x] == x) return x;
21     return pai[x] = find(pai[x]);
22 }
23
24 void join(int a, int b){
25     a = find(a);
26     b = find(b);
27     if(peso[a] < peso[b]) pai[a] = b;
28     else if(peso[b] < peso[a]) pai[b] = a;
29     else{
30         pai[a] = b;
31         peso[b]++;
32     }
33 }
34
35 int main(){
36     // ler a entrada
37     cin >> n >> m;
38     for(int i = 1; i <= m; i++){
39         cin >> aresta[i].x >> aresta[i].y >> aresta[i].dis;
40     }
41     // inicializar os pais para o union-find
42     for(int i = 1; i <= n; i++) pai[i] = i;
43     // ordenar as arestas
44     sort(aresta+1, aresta+m+1, comp);
45
46     int size = 0;
47     for(int i = 1; i <= m; i++){
48         if( find(aresta[i].x) != find(aresta[i].y) ){ // se estiverem em componentes distintas
49             join(aresta[i].x, aresta[i].y);
50             mst[++size] = aresta[i];

```

```

51 }
52 }
53
54 // imprimir a MST
55 for(int i = 1; i < n; i++)
56     cout << mst[i].x << " " << mst[i].y << " " << mst[i].dis << "\n";
57 return 0;
58 }

```



### 5.3.5 Prim - Árvore Geradora Mínima

Algoritmo parecido com Dijkstra para encontrar árvore geradora mínima.

Implementação em C++:

```

1 typedef pair<int, int> pii;
2 #define MAXN 10100
3 #define INFINITO 999999999
4 int n, m; // número de vértices e arestas
5 int distancia[MAXN]; // o array de distâncias à fonte
6 int processado[MAXN]; // o array que guarda se um vértice foi processado
7 vector<pii> vizinhos[MAXN]; // nossas listas de adjacência. O primeiro elemento do par representa a
8 // distância e o segundo representa o vértice
9 int Prim(){
10     for(int i = 2; i <= n; i++)
11         distancia[i] = INFINITO;
12     distancia[1] = 0;
13     priority_queue<pii, vector<pii>, greater<pii>> fila;
14     fila.push( pii(distancia[1], 1) );
15     while(true){
16         int davez = -1;
17         while(!fila.empty()){
18             int atual = fila.top().second;
19             fila.pop();
20             if(!processado[atual]){
21                 davez = atual;
22                 break;
23             }
24         }
25         if(davez == -1) break;
26         processado[davez] = true;
27         for(int i = 0; i < (int)vizinhos[davez].size(); i++){
28             int dist = vizinhos[davez][i].first;
29             int atual = vizinhos[davez][i].second;
30             if( distancia[atual] > dist && !processado[atual]){
31                 distancia[atual] = dist;
32                 fila.push( pii(distancia[atual], atual) );
33             }
34         }
35     }
36     int custo_arvore = 0;
37     for(int i = 1; i <= n; i++)
38         custo_arvore += distancia[i];
39     return custo_arvore;
40 }
41 int main(){

```

```

42 cin >> n >> m;
43 for(int i = 1; i <= m; i++){
44     int x, y, tempo;
45     cin >> x >> y >> tempo;
46     vizinhos[x].push_back( pii(tempo, y) );
47     vizinhos[y].push_back( pii(tempo, x) );
48 }
49 cout << Prim() << endl;
50 return 0;
51 }

```

### 5.3.6 Ordenação Topológica

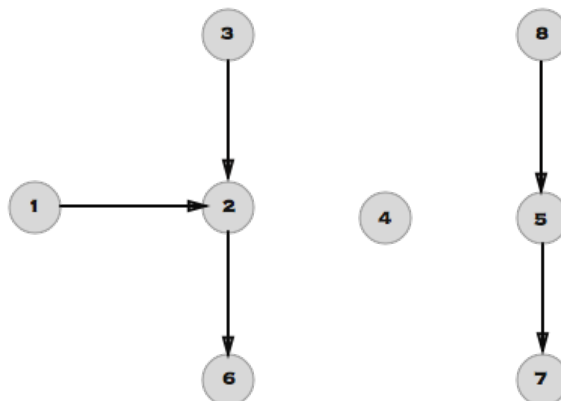
"Malter Warinho está ensinando seu filho a se vestir. Para isso, está dando instruções simples sobre a ordem em que seu filho deve se vestir para não colocar a roupa em ordem contrária (como o Superman). As instruções são do seguinte formato: as meias devem ser colocadas antes dos sapatos; as calças devem ser vestidas antes do cinto; a camisa deve ser vestida antes do casaco; e por aí vai. Em alguns casos, não interessa a ordem em que deve ser colocada a roupa. Por exemplo, o filho pode colocar as calças antes do chapéu e vice-versa. Dada a lista de instruções e o número de peças de roupas, ajude o filho de Malter a se vestir."

Bem, para formalizar um pouco o problema, vamos montar um grafo direcionado onde:

- cada vértice é uma peça de roupa.
- cada aresta partindo de um vértice X para um vértice Y significa que X tem que vir antes de Y.

Assim, pode-se notar uma relação de transição: se X tem que vir antes de Y e Y tem que vir antes de Z, X tem que vir antes de Z.

Teremos então um grafo semelhante a este:



Tendo uma noção do grafo, é fácil perceber alguns fatos simples:

- se o grafo possui um ciclo, não há ordem em que se possa resolver o problema.
- podemos executar um vértice (vestir uma roupa) se, e somente se, todos os vértices (roupas) que possuem algum caminho até ele já foram executados.

Com apenas isso, já se pode pensar em um algoritmo bem simples para resolver o problema:

- Pegar um vértice de grau de entrada zero (nenhuma aresta chega a ele) e acrescentar o vértice à ordem de execução.

- Remover todas as arestas que partem desse vértice e atualizar os graus dos vértices ligados a essas arestas.
- Repetir o processo até não haver mais vértices de grau de entrada zero (ou acabarem todos os vértices).

Se, ao final do processo, ainda sobraem vértices, há um ciclo e não há ordem para resolver o problema. Caso contrário, o problema está resolvido.

Implementação em C++:

```

1 #include <vector>
2 #include <iostream>
3 using namespace std;
4 //-----
5 #define MAXN 100100
6 int n; // número de vértices
7 int m; // número de arestas
8 vector<int> grafo[MAXN];
9 int grau[MAXN];
10 vector<int> lista; // dos vértices de grau zero
11 //-----
12 int main(){
13
14     cin >> n >> m;
15
16     for(int i = 1; i <= m; i++){
17         int x, y;
18         cin >> x >> y;
19
20         // tarefa X tem que ser executada antes da tarefa Y
21         grau[y]++;
22         grafo[x].push_back(y);
23     }
24
25     for(int i = 1; i <= n; i++) if(grau[i] == 0) lista.push_back(i);
26
27     // o procedimento a ser feito é semelhante a uma BFS
28     int ini = 0;
29     while(ini < (int)lista.size()){
30
31         int atual = lista[ini];
32         ini++;
33
34         for(int i = 0; i < (int)grafo[atual].size(); i++){
35             int v = grafo[atual][i];
36             grau[v]--;
37             if(grau[v] == 0) lista.push_back(v); // se o grau se tornar zero, acrescenta-se a lista
38         }
39     }
40
41     // agora, se na lista não houver N vértices,
42     // sabemos que é impossível realizar o procedimento
43
44     if((int)lista.size() < n) cout << "impossivel\n";
45     else{
46         for(int i = 0; i < (int)lista.size(); i++) cout << lista[i] << " ";
47         cout << endl;
48     }
49
50     return 0;
51 }

```

### 5.3.7 Floyd-Warshall - Menor Caminho

Menor distância de qualquer vértice para qualquer outro.

```

1 for(int k = 1; k <= n; k++)
2     for(int i = 1; i <= n; i++)
3         for(int j = 1; j <= n; j++)
4             distancia[i][j] = min(distancia[i][j], distancia[i][k] + distancia[k][j]);
5
6 #include <stdio.h>
7
8 // Number of vertices in the graph
9 #define V 4
10
11 /* Define Infinite as a large enough value. This value will be used
12    for vertices not connected to each other */

```

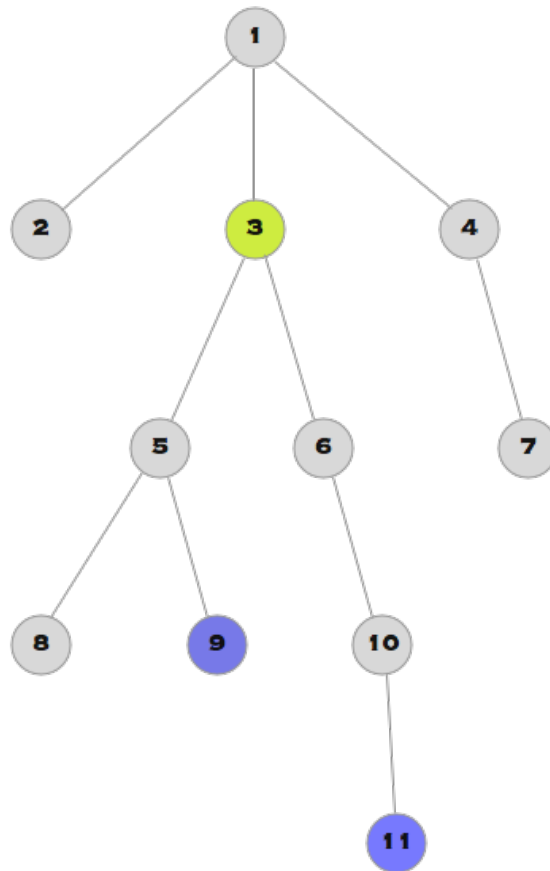
```

13 #define INF 99999
14
15 // A function to print the solution matrix
16 void printSolution(int dist[][V]);
17
18 // Solves the all-pairs shortest path problem using Floyd Warshall algorithm
19 void floydWarshall (int graph[][V])
20 {
21     /* dist[][] will be the output matrix that will finally have the shortest
22        distances between every pair of vertices */
23     int dist[V][V], i, j, k;
24
25     /* Initialize the solution matrix same as input graph matrix. Or
26        we can say the initial values of shortest distances are based
27        on shortest paths considering no intermediate vertex. */
28     for (i = 0; i < V; i++)
29         for (j = 0; j < V; j++)
30             dist[i][j] = graph[i][j];
31
32     /* Add all vertices one by one to the set of intermediate vertices.
33        —> Before start of an iteration, we have shortest distances between all
34        pairs of vertices such that the shortest distances consider only the
35        vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
36        —> After the end of an iteration, vertex no. k is added to the set of
37        intermediate vertices and the set becomes {0, 1, 2, .. k} */
38     for (k = 0; k < V; k++)
39     {
40         // Pick all vertices as source one by one
41         for (i = 0; i < V; i++)
42         {
43             // Pick all vertices as destination for the
44             // above picked source
45             for (j = 0; j < V; j++)
46             {
47                 // If vertex k is on the shortest path from
48                 // i to j, then update the value of dist[i][j]
49                 if (dist[i][k] + dist[k][j] < dist[i][j])
50                     dist[i][j] = dist[i][k] + dist[k][j];
51             }
52         }
53     }
54
55     // Print the shortest distance matrix
56     printSolution(dist);
57 }
58
59 /* A utility function to print solution */
60 void printSolution(int dist[][V])
61 {
62     printf ("The following matrix shows the shortest distances"
63            " between every pair of vertices \n");
64     for (int i = 0; i < V; i++)
65     {
66         for (int j = 0; j < V; j++)
67         {
68             if (dist[i][j] == INF)
69                 printf ("%7s", "INF");
70             else
71                 printf ("%7d", dist[i][j]);
72         }
73         printf ("\n");
74     }
75 }
76
77 // driver program to test above function
78 int main()
79 {
80     /* Let us create the following weighted graph
81
82         (0)----->(3)
83         |           /|\
84         5 |         / 1
85         | |       /
86         \|\----->(2)
87         (1)         3
88
89     */
90     int graph[V][V] = { {0, 5, INF, 10},
91                         {INF, 0, 3, INF},
92                         {INF, INF, 0, 1},
93                         {INF, INF, INF, 0}
94     };
95
96     // Print the solution
97     floydWarshall(graph);
98     return 0;
99 }

```

### 5.3.8 LCA - Menor Ancestral Comum

Ancestral comum mais próximo à dois vértices.



```
1 int LCA(int u, int v){
2
3   if(nivel[u] < nivel[v]) swap(u, v); // isto é para definir u como estando mais abaixo
4
5   // vamos agora fazer nivel[u] ser
6   // igual nivel[v], subindo pelos
7   // ancestrais de u
8
9   for(int i = MAXL-1; i >= 0; i--)
10    if(nivel[u] - (1<<i) >= nivel[v])
11      u = ancestral[u][i];
12
13   // agora, u e v estão no mesmo nível
14   if(u == v) return u; // se eles forem o mesmo nó já achamos nossa resposta
15
16   // subimos o máximo possível de forma
17   // que os dois NÃO passem a ser iguais
18
19   for(int i = MAXL-1; i >= 0; i--)
20     if(ancestral[u][i] != -1 && ancestral[u][i] != ancestral[v][i]){
21       u = ancestral[u][i];
22       v = ancestral[v][i];
23     }
24
25   // como subimos o máximo possível
26   // sabemos que u != v e que pai[u] == pai[v]
27   // logo, LCA(u, v) == pai[u] == pai[v]
28
29   return ancestral[u][0];
30 }
```



### 5.3.9 Caminho Euleriano

Um Caminho Euleriano de um grafo é um trajeto que passa por todas as arestas do grafo sem repetição.

**Existência:** Porém, antes de procurarmos um Caminho Euleriano para um grafo, precisamos saber se ele existe. Checar a existência de um Caminho Euleriano é, na verdade, bem fácil. Para um grafo ter um Caminho Euleriano, é suficiente e necessário satisfazer uma de duas condições:

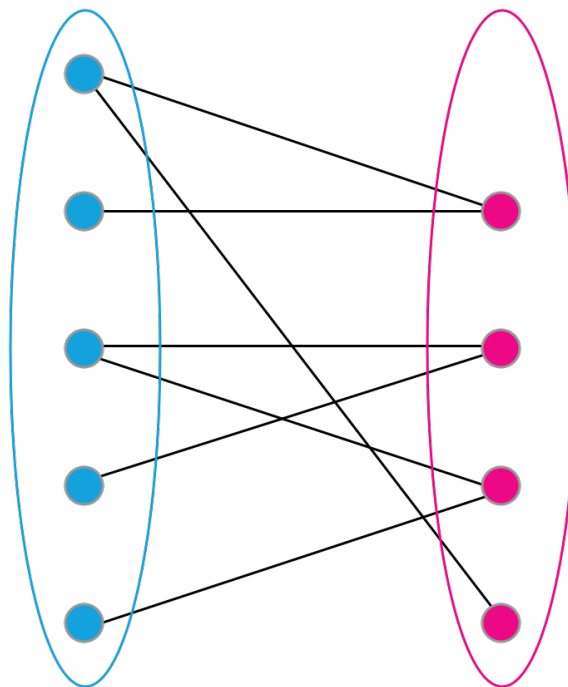
- Todos os vértices do grafo tem que ter grau par.
- Todos os vértices (ignorando-se os de grau zero) tem grau par, exceto dois vértices que possuem grau ímpar. Nesse caso, os dois vértices de grau ímpar são o início e o fim do Caminho.

Implementação em C++:

```
1 vector<int> caminho; // guardará nosso Caminho Euleriano (invertido)
2 vector<int> vizinhos[MAXN]; // nossa lista de adjacência
3 map< pair<int, int>, bool > deletada; // mapa que checa se a aresta já foi deletada
4 void acha_caminho(int v){
5     for(int i = 0; i < (int)vizinhos[v].size(); i++){
6         int viz = vizinhos[v][i];
7         if( deletada[make_pair(v, viz)] == true )
8             continue;
9         deletada[make_pair(v, viz)] = true;
10        deletada[make_pair(viz, v)] = true;
11        acha_caminho(viz);
12    }
13    caminho.push_back(v);
14 }
```

### 5.3.10 Grafos bipartidos

Um grafo é dito bipartido quando seus vértices podem ser divididos em dois conjuntos disjuntos tais que cada aresta ligue apenas vértices de grupos diferente.



Implementação em C++:

```

1 int n; // número de vértices
2 vector<int> vizinhos[MAXN]; // a lista de adjacência de cada vértice
3 int cor[MAXN];
4 // a cor de cada vértice. Inicialmente, cor[i] = -1 para todos os vértices.
5 // definimos cor[i] = 0 como sendo azul e cor[i] = 1 como sendo rosa.
6 void colore(int x){
7     cor[x] = 0;
8     vector<int> fila;
9     fila.push_back(x);
10    int pos = 0;
11    while(pos < (int)fila.size()){ // BFS
12        int atual = fila[pos];
13        pos++;
14        for(int i = 0; i < (int)vizinhos[atual].size(); i++){
15            int v = vizinhos[atual][i];
16            if(cor[v] == -1){
17                cor[v] = 1 - cor[atual];
18                fila.push_back(v); // adicionamos v a fila da BFS
19            }
20        }
21    }
22 }
23 bool checa_bipartido(){
24     for(int i = 1; i <= n; i++){
25         if(cor[i] == -1){
26             colore(i);
27         }
28     }
29     for(int i = 1; i <= n; i++){
30         for(int j = 0; j < (int)vizinhos[i].size(); j++){
31             int v = vizinhos[i][j];
32             if(cor[i] == cor[v])
33                 return false;
34         }
35     }
36     return true;
37 }

```

## 6 Programação dinâmica

Aplicar a programação dinâmica se:

- É possível dividir o problema em problemas menores.
- É possível encontrar as soluções ótimas para os subproblemas.
- Há sobreposição de subproblemas, isto é, há subproblemas que compartilham as mesmas respostas ótimas.

### 6.1 Problema da mochila

O problema da mochila consiste de uma mochila de capacidade total  $C$  e de  $N$  itens que podem ser colocados dentro da mochila. Cada item possui um peso  $p_i$  e um valor  $v_i$ . Objetiva-se colocar o maior número de itens dentro da mochila a fim de maximizar o valor total dos itens colocados. Matematicamente:

$$\max \sum_{i=1}^I v_i, \quad (3)$$

onde  $I$  é o conjunto de itens dentro da mochila. Sujeito as restrições:

$$\sum_{i=1}^I p_i \leq C \quad (4)$$

```
1 #include <stdio.h>
2
3 // Retorna o máximo de dois números.
4 int max(int a, int b) {
5     return a>b?a:b;
6 }
7
8 int main() {
9
10     // Número de itens
11     int n_itens = 3;
12
13     // Vetor para armazenar o valor de cada item.
14     int valor[] = {3, 4, 5};
15
16     // Vetor para armazenar o peso de cada item.
17     int peso[] = {2, 3, 2};
18
19     // Capacidade da mochila.
20     int capacidade = 6;
21
22     // Matriz de memorização.
23     int memo[capacidade+1][n_itens+1];
24
25     int w, j;
26     for(w = 0; w <= capacidade; ++w) {
27         for(j = 0; j <= n_itens; ++j) {
28             if(j == 0 || w == 0) {
29                 memo[w][j] = 0;
30             } else {
31                 memo[w][j] = memo[w][j-1];
32                 if(peso[j-1] <= w) {
33                     // Caso em que colocamos apenas uma vez um item
34                     memo[w][j] = max(memo[w][j-1],
35                                     memo[w-peso[j-1]][j-1]+valor[j-1]);
36                     // Caso em que um item pode ser colocado mais de uma vez
37                     memo[w][j] = max(memo[w][j],
38                                     memo[w-peso[j-1]][j]+valor[j-1]);
39                 }
40             }
41         }
42     }
43
44 }
```

```

45     }
46     }
47 }
48
49 // Solução ótima está no último elemento da matriz de memorização.
50 printf("Valor ótimo = %d", memo[capacidade][n_itens]);
51
52 return 0;
53 }

```

## 6.2 Problema do troco

### 6.2.1 Problema do corte de hastes

Dada uma haste de tamanho  $n$  e o preço  $p_i$  do corte de tamanho  $i$ . Qual é a melhor maneira de cortar a haste para maximizar o preço?

```

1 #include <stdio.h>
2
3 int main() {
4
5     // tamanho da haste.
6     int n = 5;
7
8     // Preço para cada tamanho de corte.
9     // ex. para o tamanho 1 o preço é 2.
10    int precos_corte[] = {2,4,3,1,5};
11
12    int i, j;
13
14    // Vetor de memorização.
15    int memo[n+1];
16    memo[0] = 0; // Solução trivial - corte de tamanho zero com preço zero.
17
18    for(i = 1; i <= n; ++i) {
19        int q = -1;
20        for(j = 1; j <= i; ++j) {
21            if(q < (precos_corte[j-1] + memo[i-j])) {
22                q = precos_corte[j-1] + memo[i-j];
23            }
24        }
25        memo[i] = q;
26    }
27
28    // Solução está na última posição do vetor de memorização:
29    printf("Valor de venda = %d", memo[n]);
30
31    return 0;
32 }

```

### 6.2.2 Dado valor e as moedas existe troco possível?

```

1 // Lembre-se de inicializar todos os valores da DP como -1 (não calculado) e, para cada estado, ela
2 // retornará 1, se for true, ou 0, caso seja false
3 int dp[MAX];
4 // F função retorna true assim que encontramos uma maneira de formar o valor x. Se testarmos todas
5 // as moedas e nenhuma funcionar, ela retorna false
6 int solve(int x, vector<int> &c){
7     if(x==0) return 1;
8     if(x<0) return 0;
9     if(dp[x]>=0) return dp[x];
10    for(int i=0;i<c.size();i++)
11        if(solve(x-c[i])) return dp[x-c[i]]=1;
12
13    return dp[x]=0;
14 }

```

## 6.3 Calculadora Quebrada

```
1 package calculadora_quebrada;
2
3 import java.io.IOException;
4 import java.util.ArrayList;
5 import java.util.LinkedList;
6 import java.util.Queue;
7 import java.util.Scanner;
8
9 class Main {
10     ArrayList<Estado> estados;
11     static int[] estadosPositivosJaGerados = new int[100000];
12     static int[] estadosNegativosJaGerados = new int[100000];
13     static int valorAlvo;
14     static int numeroTeclasFuncionando;
15     static int numeroOperacoesFuncionando;
16     static int[] teclasFuncionando = new int[10];
17     // 0 = +, 1 = -, 2 = /, 3 = *
18     static int[] operacoesFuncionando = new int[4];
19
20     public static void main(String[] args) throws IOException {
21         Scanner s = new Scanner(System.in);
22
23         valorAlvo = s.nextInt();
24         numeroTeclasFuncionando = s.nextInt();
25         for(int i = 0; i < numeroTeclasFuncionando; i++) {
26             teclasFuncionando[s.nextInt()] = 1;
27         }
28
29         numeroOperacoesFuncionando = s.nextInt();
30         s.nextLine();
31         String operacoes = s.nextLine();
32         String[] operacoesVetor = operacoes.split("\\s+");
33         for(int i = 0; i < operacoesVetor.length; i++) {
34             char op = operacoesVetor[i].charAt(0);
35             if(op == '+') {
36                 operacoesFuncionando[0] = 1;
37             } else if(op == '-') {
38                 operacoesFuncionando[1] = 1;
39             } else if(op == '/') {
40                 operacoesFuncionando[2] = 1;
41             } else if(op == '*') {
42                 operacoesFuncionando[3] = 1;
43             }
44         }
45
46         /* Definindo valores manualmente
47         valorAlvo = 13;
48         numeroTeclasFuncionando = 4;
49
50         teclasFuncionando[0] = 1;
51         teclasFuncionando[1] = 1;
52         teclasFuncionando[2] = 1;
53         teclasFuncionando[7] = 1;
54
55         numeroOperacoesFuncionando = 3;
56         operacoesFuncionando[1] = 1; // -
57         operacoesFuncionando[3] = 1; // *
58         operacoesFuncionando[0] = 1; // +
59         */
60
61         Estado raiz = new Estado(0, null);
62         estadosPositivosJaGerados[raiz.valor] = 1;
63         estadosNegativosJaGerados[raiz.valor] = 1;
64
65         Estado solucao = bfs(raiz);
66         int acoes = getAcoesAPartirDaSolucao(solucao, raiz);
67         acoes--;
68
69         System.out.println(String.valueOf(acoes));
70     }
71
72     private static int getAcoesAPartirDaSolucao(Estado estadoSolucao, Estado raiz){
73         int acoes = 0;
74         Estado estadoAtual = estadoSolucao;
75         while(estadoAtual != raiz) {
76             acoes++;
77             estadoAtual = estadoAtual.pai;
78         }
79
80         return acoes;
81     }
82
83     private static Estado bfs(Estado raiz) {
84         Estado paiAtual;
85     }
```

```

86     ArrayList<Estado> filhosDoPaiAtual;
87
88     Queue<Estado> fila = new LinkedList<Estado>();
89
90     fila.add(raiz);
91
92     while(!fila.isEmpty()) {
93         paiAtual = fila.poll();
94         filhosDoPaiAtual = geraFilhos(paiAtual);
95
96         for(int i = 0; i < filhosDoPaiAtual.size(); i++) {
97             if(filhosDoPaiAtual.get(i).valor == valorAlvo) {
98                 return filhosDoPaiAtual.get(i);
99             } else {
100                 fila.add(filhosDoPaiAtual.get(i));
101             }
102         }
103     }
104     // Se nao encontrar solucao retorna a propria raiz
105     return raiz;
106 }
107
108 private static ArrayList<Estado> geraFilhos(Estado pai) {
109     ArrayList<Estado> filhos = new ArrayList<>();
110
111     int valorFilho;
112     boolean divisaoZero;
113
114     //if(pai.valor == 0) {
115
116     //}else {
117     // A partir da segunda acao
118     for(int i = 0; i < 4; i++) {
119         // Verifica se a operacao esta funcionando
120         if(operacoesFuncionando[i] != 0) {
121             for(int j = 0; j < 10; j++) {
122                 // Verifica se tecla esta funcionando
123                 if(teclasFuncionando[j] != 0) {
124                     divisaoZero = false;
125                     valorFilho = 0;
126                     switch(i) {
127                         case 0:
128                             valorFilho = pai.valor+j;
129                             break;
130                         case 1:
131                             valorFilho = pai.valor-j;
132                             break;
133                         case 2:
134                             if(j != 0) {
135                                 valorFilho = pai.valor/j;
136                             } else {
137                                 divisaoZero = true;
138                             }
139                             break;
140                         case 3:
141                             valorFilho = pai.valor*j;
142                             break;
143                     }
144                     if(!divisaoZero) {
145                         // Verifica se numero e positivo ou negativo
146                         if(valorFilho >= 0) {
147                             if(estadosPositivosJaGerados[valorFilho] != 1) {
148                                 filhos.add(new Estado(valorFilho, pai));
149                                 estadosPositivosJaGerados[valorFilho] = 1;
150                             }
151                         } else {
152                             if(estadosNegativosJaGerados[Math.abs(valorFilho)] != 1) {
153                                 filhos.add(new Estado(valorFilho, pai));
154                                 estadosNegativosJaGerados[Math.abs(valorFilho)] = 1;
155                             }
156                         }
157                     }
158                 }
159             }
160         }
161     }
162     //}
163
164     return filhos;
165 }
166
167 private static void printaFilhos(Estado pai, ArrayList<Estado> filhos) {
168     System.out.printf("%s: %d filhos => ", pai, filhos.size());
169     for(int i = 0; i < filhos.size(); i++) {
170         System.out.print(filhos.get(i) + " ");
171     }
172     System.out.println();
173 }

```

```

174     }
175
176 }
177
178 class Estado{
179     int valor;
180     Estado pai;
181
182     public Estado(int valor, Estado pai) {
183         this.valor = valor;
184         this.pai = pai;
185     }
186
187     public String toString() {
188         return String.valueOf(valor);
189     }
190 }

```

### 6.3.1 Mínimo de moedas para troco

```

1 // função que recebe o valor de troco N, o número de moedas disponíveis M,
2 // e um vetor com as moedas disponíveis m
3 // essa função deve retornar o número mínimo de moedas,
4 // de acordo com a solução com Programação Dinâmica.
5 int num_moedas(int N, int M, int * m) {
6     int dp[N+1];
7
8     // caso base
9     dp[0] = 0;
10
11    // sub-problemas
12    for(int i=1; i<=N; i++) {
13        // é comum atribuir um valor alto, que concerteza
14        // é maior que qualquer uma das próximas possibilidades,
15        // sendo assim substituído
16        dp[i] = 1000000;
17
18        for(int j=0; j<M; j++) {
19            if(i-m[j] >= 0) {
20                dp[i] = min(dp[i], dp[ i-m[j] ]+1);
21            }
22        }
23    }
24
25    // solução
26    return dp[N];
27 }

```

## 6.4 Contagem de inversões

Um dos problemas mais clássicos de programação é a contagem de inversões em uma sequência. De maneira simples, seja  $S = a_1, a_2, \dots, a_n$ . Uma inversão em  $S$  é um par  $(i, j)$ , com  $i < j$ , tal que  $a_i > a_j$ . Sabendo disso, faça um programa que calcula o número de inversões em uma sequência  $S$ .

```

1 #define INF 1000000000
2
3 // função merge_sort que ordena um vector v
4 int merge_sort(vector<int> &v){
5
6     // declaro inv, o total de inversões
7     int inv=0;
8
9     // se o tamanho de v for 1, não há inversões
10    if(v.size()==1) return 0;
11
12    // se não
13
14    // declaro os vetores u1 e u2
15    vector<int> u1, u2;
16
17    // e faço cada um receber uma metade de v
18    for(int i=0; i<v.size()/2; i++){
19        u1.push_back(v[i]);
20    }
21    for(int i=v.size()/2; i<v.size(); i++){

```

```

22     u2.push_back(v[i]);
23 }
24 // ordeno u1 e u2
25 // e adiciono a inv as inversões de cada metade do vetor
26 inv+=merge_sort(u1);
27 inv+=merge_sort(u2);
28
29 // e adiciono INF ao final de cada um deles
30 u1.push_back(INF);
31 u2.push_back(INF);
32
33 // declaro ini1 e ini2 com valor inicial zero
34 int ini1=0, ini2=0;
35
36 // percorro cada posição de v
37 for(int i=0;i<v.size();i++){
38
39     // se o menor não usado de u1 for menor o mesmo em u2
40     if(u1[ini1]<=u2[ini2]){
41
42         // então o coloco em v
43         v[i]=u1[ini1];
44
45         // e incremento o valor de ini1
46         ini1++;
47     }
48
49     // caso contrário, faço o análogo com u2 e ini2
50     else{
51
52         v[i]=u2[ini2];
53         ini2++;
54
55         // não se esquecendo de adicionar o número de elementos em u1
56         // ao total de inversões em v
57         inv+=u1.size()-ini1-1;
58     }
59 }
60
61 // por fim, retorno a quantidade de inversões
62 return inv;
63 }

```

## 6.5 Maior Subsequência Comum

Dadas duas sequências  $s_1$  e  $s_2$ , uma de tamanho  $n$  e outra de tamanho  $m$ , qual a maior subsequência comum às duas? Lembre-se que uma subsequência de  $s_1$ , por exemplo, é simplesmente um subconjunto dos elementos de  $s_1$  na mesma ordem em que apareciam antes. Isto significa que 1, 3, 5 é uma subsequência de 1, 2, 3, 4, 5, mesmo 1 não estando do lado do 3 na sequência original

```

1 using namespace std;
2
3 // defino MAXN como 1010
4 #define MAXN 1010
5
6 // declaro as variáveis que vou usar
7 int s1[MAXN], s2[MAXN], tab[MAXN][MAXN];
8
9 int lcs(int a, int b){ // declaro a função da DP, de nome lcs
10
11     // se já calculamos esse estado da dp antes
12     if(tab[a][b]>=0) return tab[a][b]; // retornamos o valor salvo para ele
13
14     // se uma das sequências for vazia, retornamos zero
15     if(a==0 || b==0) return tab[a][b]=0;
16
17     // se s1[a] for igual a s2[b], os retiramos das sequências
18     if(s1[a]==s2[b]) return 1+lcs(a-1, b-1); // e adicionamos ele à lcs das subsequências restantes
19
20     // se forem diferentes, retorno o máximo entre retirar s1[a] ou s1[b]
21     return tab[a][b]=max(lcs(a-1, b), lcs(a, b-1));
22 }

```

```

1 #include <bits/stdc++.h>
2
3 int max(int a, int b);
4
5 /* Returns length of LCS for X[0..m-1], Y[0..n-1] */
6 int lcs( char *X, char *Y, int m, int n )

```



```

7 {
8     if (m == 0 || n == 0)
9         return 0;
10    if (X[m-1] == Y[n-1])
11        return 1 + lcs(X, Y, m-1, n-1);
12    else
13        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
14 }
15
16 /* Utility function to get max of 2 integers */
17 int max(int a, int b)
18 {
19     return (a > b)? a : b;
20 }
21
22 /* Driver program to test above function */
23 int main()
24 {
25     char X[] = "AGGTAB";
26     char Y[] = "GXTXAYB";
27
28     int m = strlen(X);
29     int n = strlen(Y);
30
31     printf("Length of LCS is %d", lcs( X, Y, m, n ) );
32
33     return 0;
34 }

```

## 6.6 Maior Subsequência crescente

Dada uma sequência  $s$  qualquer, descobrir o tamanho da maior subsequência crescente de  $s$ . Vale lembrar que uma subsequência de  $s$  é qualquer subconjunto de elementos de  $s$ . Veja, por exemplo:  $s = 3, 4, 3, 5, 2, 7$  A maior subsequência crescente de  $s$ , neste caso é:  $s' = 3, 4, 5, 7$  e tem tamanho 4.

```

1 #define PB push_back // por simplicidade
2 #define MAXN 100100 // defino o valor de MAXN
3
4 vector<int> lis(vector<int> &v){
5
6     // declaro s variáveis que vou usar
7     vector<int> pilha, resp;
8     int pos[MAXN], pai[MAXN];
9
10    // para cada elemento
11    for(int i=0; i<v.size(); i++){
12
13        // declaro um iterador que guardará o elemento mais à esquerda de pilha
14        // que não é menor que v[i]
15        vector<int>::iterator it = lower_bound(pilha.begin(), pilha.end(), v[i]);
16
17        // guardo a posição da pilha em que adicionarei o elemento
18        int p = it-pilha.begin();
19
20        // se it for o fim do vector, então não há elemento que não seja menor que v[i]
21        // ou seja, todos os topos de pilha são menores ou iguais a v[i]
22
23        // logo, criamos uma nova pilha e colocamos x no seu topo
24        if(it==pilha.end()) pilha.PB(v[i]);
25
26        // porém, se it apontar para alguma posição válida do vector
27        // colocamos v[i] no topo desta pilha, substituindo o valor que it aponta por v[i]
28        else *it = v[i];
29
30        // a posição original na sequência do número no topo da pilha p agora é i
31        pos[p]=i;
32
33        // se o elemento foi inserido na primeira pilha
34        if(p==0) pai[i]=-1; // seu pai será -1
35
36        // caso contrário, seu pai será a posição do elemento no topo da pilha anterior a ele
37        else pai[i]=pos[p-1];
38    }
39
40    // p será a posição do elemento no topo da última pilha
41    int p = pos[pilha.size()-1];
42
43    // enquanto p não for -1
44    while(p>=0){
45

```

```

46 // adiciono o elemento na posição p à resposta
47 resp.PB(v[p]);
48
49 // e vou para o pai de p
50 p=pai[p];
51 }
52
53 // inverte a ordem da resposta
54 reverse(resp.begin(), resp.end());
55
56 // por fim, retorno o vetor resp
57 return resp;
58 }

```

## 6.7 Soma máxima em um intervalo

Dada uma sequência qualquer  $S=(s_1,s_2,s_3,\dots,s_n)$  qual a maior soma que podemos obter escolhendo um subconjunto de termos adjacentes de  $S$ ? Se a sequência for, por exemplo, (1,-3,5,-2,1,-1), a soma máxima é 4, com os termos (5,-2,1).

```

1 int max_sum(vector<int> s){
2
3     int resp=0, maior=0;
4
5     for(int i=0;i<s.size();i++){
6
7         maior=max(0,maior+s[i]);
8
9         resp=max(resp, maior);
10    }
11
12    return resp;
13 }

```

## 6.8 Vertex Cover

Um reino possui  $N$  cidades conectadas entre si por  $N-1$  rotas bidirecionais, onde se é possível viajar de qualquer cidade a qualquer outra. Preocupada com a segurança das rotas, a rainha decidiu instalar postos de segurança em algumas cidades. O objetivo é que, para toda rota, exista um posto de segurança em ao menos uma das cidades conectadas por essa rota. Também preocupada com as finanças do reino, a rainha lhe contratou para selecionar achar o número mínimo de cidades em que é preciso construir um posto de segurança de forma a satisfazer as condições necessárias.

```

1 int VertexCover(X, pai_colorido){
2
3     if(PD[X][pai_colorido] != -1)
4         return PD[X][pai_colorido]; // se já calculamos esse caso, retornamos o valor para evitar
           recálculo
5                                     // não se esqueça de, não função main, inicializar todos os valores
           de PD como -1.
6
7     int caso1 = 1, caso2 = 0;
8
9     for(int i = 0; i < (int)vizinhos[X].size(); i++){ // percorremos todos os vizinhos de X
10
11         int V = vizinhos[X][i];
12
13         if(V == pai[X]) continue; // checamos se V é o pai de X
14
15         // agora, sabemos que V é um filho de X
16         pai[V] = X; // definimos o pai de V como sendo X
17         caso1 += VertexCover(V, true); // caso escolhamos colorir X
18         caso2 += VertexCover(V, false); // caso escolhamos não-colorir X
19     }
20
21     if(pai_colorido == true) PD[X][pai_colorido] = min(caso1, caso2); // caso o pai de X esteja
           colorido, escolhemos o melhor caso
22     if(pai_colorido == false) PD[X][pai_colorido] = caso1; // caso o pai de X não seja
           colorido, escolhemos o caso1
23
24     return PD[X][pai_colorido]; // retornamos o valor da resposta

```



## 7 Outros

### 7.1 Compilar programa em Java no CMD

```
javac NomeDoArquivo.java  
java NomeDaClasse
```

### 7.2 Compilar programa em C++ no CMD

```
g++ arquivo.cpp  
a.exe < input.txt (No Windows)  
./a < input.txt (No Linux)
```

### 7.3 Tabela ASCII

## ASCII Table

| Dec | Hex | Oct | Char | Dec | Hex | Oct | Char    | Dec | Hex | Oct | Char | Dec | Hex | Oct | Char |
|-----|-----|-----|------|-----|-----|-----|---------|-----|-----|-----|------|-----|-----|-----|------|
| 0   | 0   | 0   |      | 32  | 20  | 40  | [space] | 64  | 40  | 100 | @    | 96  | 60  | 140 | `    |
| 1   | 1   | 1   |      | 33  | 21  | 41  | !       | 65  | 41  | 101 | A    | 97  | 61  | 141 | a    |
| 2   | 2   | 2   |      | 34  | 22  | 42  | "       | 66  | 42  | 102 | B    | 98  | 62  | 142 | b    |
| 3   | 3   | 3   |      | 35  | 23  | 43  | #       | 67  | 43  | 103 | C    | 99  | 63  | 143 | c    |
| 4   | 4   | 4   |      | 36  | 24  | 44  | \$      | 68  | 44  | 104 | D    | 100 | 64  | 144 | d    |
| 5   | 5   | 5   |      | 37  | 25  | 45  | %       | 69  | 45  | 105 | E    | 101 | 65  | 145 | e    |
| 6   | 6   | 6   |      | 38  | 26  | 46  | &       | 70  | 46  | 106 | F    | 102 | 66  | 146 | f    |
| 7   | 7   | 7   |      | 39  | 27  | 47  | '       | 71  | 47  | 107 | G    | 103 | 67  | 147 | g    |
| 8   | 8   | 10  |      | 40  | 28  | 50  | (       | 72  | 48  | 110 | H    | 104 | 68  | 150 | h    |
| 9   | 9   | 11  |      | 41  | 29  | 51  | )       | 73  | 49  | 111 | I    | 105 | 69  | 151 | i    |
| 10  | A   | 12  |      | 42  | 2A  | 52  | *       | 74  | 4A  | 112 | J    | 106 | 6A  | 152 | j    |
| 11  | B   | 13  |      | 43  | 2B  | 53  | +       | 75  | 4B  | 113 | K    | 107 | 6B  | 153 | k    |
| 12  | C   | 14  |      | 44  | 2C  | 54  | ,       | 76  | 4C  | 114 | L    | 108 | 6C  | 154 | l    |
| 13  | D   | 15  |      | 45  | 2D  | 55  | -       | 77  | 4D  | 115 | M    | 109 | 6D  | 155 | m    |
| 14  | E   | 16  |      | 46  | 2E  | 56  | .       | 78  | 4E  | 116 | N    | 110 | 6E  | 156 | n    |
| 15  | F   | 17  |      | 47  | 2F  | 57  | /       | 79  | 4F  | 117 | O    | 111 | 6F  | 157 | o    |
| 16  | 10  | 20  |      | 48  | 30  | 60  | 0       | 80  | 50  | 120 | P    | 112 | 70  | 160 | p    |
| 17  | 11  | 21  |      | 49  | 31  | 61  | 1       | 81  | 51  | 121 | Q    | 113 | 71  | 161 | q    |
| 18  | 12  | 22  |      | 50  | 32  | 62  | 2       | 82  | 52  | 122 | R    | 114 | 72  | 162 | r    |
| 19  | 13  | 23  |      | 51  | 33  | 63  | 3       | 83  | 53  | 123 | S    | 115 | 73  | 163 | s    |
| 20  | 14  | 24  |      | 52  | 34  | 64  | 4       | 84  | 54  | 124 | T    | 116 | 74  | 164 | t    |
| 21  | 15  | 25  |      | 53  | 35  | 65  | 5       | 85  | 55  | 125 | U    | 117 | 75  | 165 | u    |
| 22  | 16  | 26  |      | 54  | 36  | 66  | 6       | 86  | 56  | 126 | V    | 118 | 76  | 166 | v    |
| 23  | 17  | 27  |      | 55  | 37  | 67  | 7       | 87  | 57  | 127 | W    | 119 | 77  | 167 | w    |
| 24  | 18  | 30  |      | 56  | 38  | 70  | 8       | 88  | 58  | 130 | X    | 120 | 78  | 170 | x    |
| 25  | 19  | 31  |      | 57  | 39  | 71  | 9       | 89  | 59  | 131 | Y    | 121 | 79  | 171 | y    |
| 26  | 1A  | 32  |      | 58  | 3A  | 72  | :       | 90  | 5A  | 132 | Z    | 122 | 7A  | 172 | z    |
| 27  | 1B  | 33  |      | 59  | 3B  | 73  | ;       | 91  | 5B  | 133 | [    | 123 | 7B  | 173 | {    |
| 28  | 1C  | 34  |      | 60  | 3C  | 74  | <       | 92  | 5C  | 134 | \    | 124 | 7C  | 174 |      |
| 29  | 1D  | 35  |      | 61  | 3D  | 75  | =       | 93  | 5D  | 135 | ]    | 125 | 7D  | 175 | }    |
| 30  | 1E  | 36  |      | 62  | 3E  | 76  | >       | 94  | 5E  | 136 | ^    | 126 | 7E  | 176 | ~    |
| 31  | 1F  | 37  |      | 63  | 3F  | 77  | ?       | 95  | 5F  | 137 | _    | 127 | 7F  | 177 |      |

## 7.4 C++ Limits

bool: a boolean (true/false)

char: an 8-bit signed integer (often used to represent characters with ASCII)

short: a 16-bit signed integer

int: a 32-bit signed integer

long long: a 64-bit signed integer

float: a 32-bit floating-point number

double: a 64-bit floating-point number

long double: a 128-bit floating-point number

string: a string of characters

| Type      | Bytes | Min value            | Max value           |
|-----------|-------|----------------------|---------------------|
| bool      | 1     |                      |                     |
| char      | 1     | -128                 | 127                 |
| short     | 2     | -32768               | 32767               |
| int       | 4     | -2148364748          | 2147483647          |
| long long | 8     | -9223372036854775808 | 9223372036854775807 |
|           | $n$   | $-2^{8n-1}$          | $2^{8n-1} - 1$      |

| Type               | Bytes | Min value | Max value            |
|--------------------|-------|-----------|----------------------|
| unsigned char      | 1     | 0         | 255                  |
| unsigned short     | 2     | 0         | 65535                |
| unsigned int       | 4     | 0         | 4294967295           |
| unsigned long long | 8     | 0         | 18446744073709551615 |
|                    | $n$   | 0         | $2^{8n} - 1$         |

| Type        | Bytes | Min value                       | Max value                      | Precision           |
|-------------|-------|---------------------------------|--------------------------------|---------------------|
| float       | 4     | $\approx -3.4 \times 10^{38}$   | $\approx 3.4 \times 10^{38}$   | $\approx 7$ digits  |
| double      | 8     | $\approx -1.7 \times 10^{308}$  | $\approx 1.7 \times 10^{308}$  | $\approx 14$ digits |
| long double | 16    | $\approx -1.1 \times 10^{4932}$ | $\approx 1.1 \times 10^{4932}$ | $\approx 18$ digits |

## 7.5 Estruturas de dados C++

```
Static arrays - int arr[10]  
Dynamic arrays - vector<int>  
Linked lists - list<int>  
Stacks - stack<int>  
Queues - queue<int>  
Priority queues - priority_queue<int>  
Sets - set<int>  
Maps - map<int, int>
```

## 7.6 Conversão para números romanos

Fazer teste com 444 IX IV CM CD XC XL. Lembre-se que I representa 1, V é 5, X é 10, L é 50, C é 100, D é 500 e M 1000

## 7.7 Antes e depois de cristo

Não existe ano 0, existe 1 A.C. e 1 D.C.

## 7.8 Problemas que envolvem horário

Procurar sempre usar minutos/segundos

## 7.9 Ano bissexto

Considerar 366 dias

## 7.10 Ano normal

Considerar 365 dias

## 7.11 Dias de cada mês

Janeiro(1) 31  
Fevereiro(2) 28(29 bissexto)  
Março(3) 31  
Abril(4) 30  
Maio(5) 31  
Junho(6) 30  
Julho(7) 31  
Agosto(8) 31  
Setembro(9) 30  
Outubro(10) 31  
Novembro(11) 30  
Dezembro(12) 31  
30: 4 | 31: 7 | \*28: 1 | \*29: 1

## 7.12 Número de letras no alfabeto

26, contando K, W e Y

## 7.13 Forma alternativa para escrita de nome para tipos de dados

```
1 long int == long
2 long long int == long long
3 unsigned int == unsigned
4 unsigned long long int == unsigned long long
```

## 7.14 Inicializar vetor com valor predefinido

```
1 // for 1d array, use STL fill_n or fill to initialize array
2 fill(a, a+size_of_a, value)
3 fill_n(a, size_of_a, value)
4 // for 2d array, if want to fill in 0 or -1
5 memset(a, 0, sizeof(a));
6 // otherwise, use a loop of fill or fill_n through every a[i]
7 fill(a[i], a[i]+size_of_ai, value) // from 0 to number of row.
```

## 7.15 Operações para modificar sequências

```
1 void copy(first, last, result);
2 void swap(a,b);
3 void swap(first1, last1, first2); // swap range
4 void replace(first, last, old_value, new_value); // replace in range
5 void replace_if(first, last, pred, new_value); // replace in conditions
6 // pred can be represented in function
7 // e.x. bool IsOdd (int i) { return ((i%2)==1); }
8 void reverse(first, last); // reverse a range of elements
9 void reverse_copy(first, last, result); // copy a reverse of range of elements
10 void random_shuffle(first, last); // using built-in random generator to shuffle array
```

## 7.16 Permutações

```
1 bool next_permutation(iterator first, iterator last);
2 bool next_permutation(iterator first, iterator last, LessThanOrEqualFunction comp);
3 bool prev_permutation(iterator first, iterator last);
4 bool prev_permutation(iterator first, iterator last, LessThanOrEqualFunction comp);
```

## 7.17 Gerar números aleatórios

```
1 srand(time(NULL));
2 // generate random numbers between [a,b)
3 rand() % (b - a) + a;
4 // generate random numbers between [0,b)
5 rand() % b;
6 // generate random permutations
7 random_permutation(anArray, anArray + 10);
8 random_permutation(aVector, aVector + 10);
```

## 7.18 Pesquisa binária

```
1 // Necessário vetor estar ordenado
2 int binarySearch(int arr[], int l, int r, int x)
3 {
4     if (r >= l)
```

```
5 {  
6   int mid = l + (r - l) / 2;  
7   if (arr[mid] == x)  
8     return mid;  
9   if (arr[mid] > x)  
10    return binarySearch(arr, l, mid-1, x);  
11   return binarySearch(arr, mid+1, r, x);  
12 }  
13 return -1;  
14 }
```

---