INSTITUTO FEDERAL
Sul de Minas Gerais
Campus Poços de Caldas

Notebook para as Maratonas de Programação

Equipe SK Teletom

Versão 1.2 - Novembro 2018

Sumário

1 Template

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

// Nome de Tipos
typedef long ll;
typedef unsigned long long ull;
typedef long double ld;

// Atalhos
#define f first
#define s second
#define pb push_back
#define mp make_pair
#define min(a,b) ((a<b)?a:b)
#define max(a,b) ((a<b)?b:a)
#define l length()
#define forn(i, n) for ( int i = 0; i < (n); ++i )
#define fornx(i, x, n) for ( int i = (x); i < (n); ++i )

using namespace std;

int main() {

  return 0;
}
```

2 Matemática Computacional

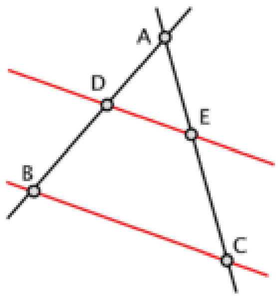
2.1 Geometria Básica

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 180º

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 (N2) 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).

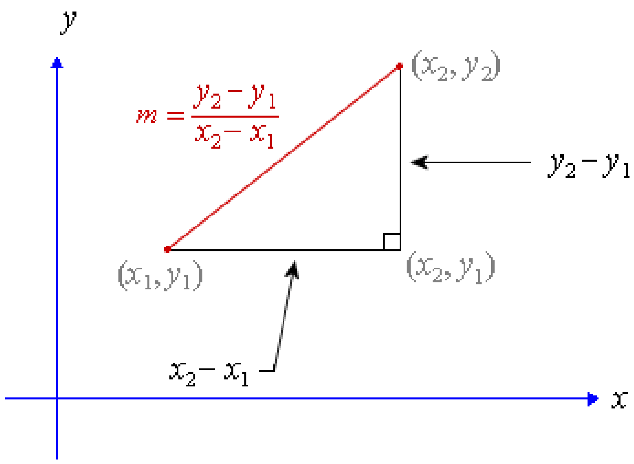
Na geometria analítica, nós consideramos que nossas figuras estão em um plano com dois eixos ortogonais que se cruzam na origem, esses eixos nos permitem definir coordenadas para os pontos dos planos e transformar um problema de geometria em um problema de álgebra, que computadores conseguem resolver.

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 x1 e y1) e o segundo P2 (de coordenadas x2 e y2), então vemos que se construirmos um ponto P3 de coordenadas x2 e y1, 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 = infinito.

Círculo

O circulo 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 xc e yc, então todos os pontos obedecem a seguinte equação:

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

2.2 Geometria Computacional

Essa clase 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:

```cpp
#include <bits/stdc++.h>
#define x first
#define y second

using namespace std;

typedef pair <double, double> point;

point sum(point a, point b){
  point ret;
  ret.x = a.x + b.x;
  ret.y = a.y + b.y;
  return ret;
}

point neg(point a){
  point ret;
  ret.x = -a.x;
  ret.y = -a.y;
  return ret;
}
```

Objeto:

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

using namespace std;

struct point{
    double x, double y;
    point(){}
    point(double _x, double _y){
      x = _x;
      y = _y;
    }
    point operator+(const point &oth){
      return point(x + oth.x, y + oth.y);
    }
    point operator-(const point &oth){
      return point(x - oth.x, y - oth.y);
    }
};
```

Complex:

```cpp
#include <bits/stdc++.h>
#define x real()
#define y imag()

using namespace std;
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 pair 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:

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

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

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

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

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 importaremos 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:

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

Ou ainda, se você estiver usando complex:

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

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 nor 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:

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

Usando a complex podemos escrever:

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

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:

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

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:

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

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

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

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:

```
  if(a*d == b*c){
    \\seu código aqui
  }
```

2.3 Primos

2.3.1 Primo rápido – O($\sqrt{n}$)

```
bool e_primo(int x) {
  if (x == 1)  return 0;
  //note que se o nmero for 2 ele no entra no loop, comportamento desejado
  for (int i = 2; i*i <= x; ++i) {
    if (x % i == 0) { //se o resto de x por i for 0, ento i divide x
      return 0;
    }
  }
  return 1;
}
```

2.3.2 Crivo de Erastótenes

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

```
// 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  primo

bool e_composto[1000010];

void crivo(int n) {
  // 1 no  composto, mas o vetor na verdade guarda os nmeros que no so primos
  e_composto[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!e_composto[i]) {
      for (int j = 2; j*i <= n; ++j) {
        e_composto[i*j] = 1;
      }
    }
  }
  return;
}

int main() {
  int N, Q, a;
  cin >> N >> Q;
  crivo(N); // Complexidade O(n*log(log(n)))
  for (int i = 0; i < Q; ++i) { // Complexidade O(Q)
    cin >> a;
    // Se composto  falso, ento  primo, caso contrrio  composto.
    cout << !e_composto[a] << "\n";
  }

  return 0;
}
```

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 3x3x7x7x13.

```
vector<int> factors(int n) {
vector<int> f;
for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
if (n > 1) f.push_back(n);
return f;
}
```

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

```
int gcd(int a, int b)
{
    if (b==0) return a;
    else return gcd(b, a%b);
}
```

2.4.2 Menor divisor comum – LCM

```
int lcm(int a, int b)
{
    return a*b/gcd(a,b);
}
```

2.4.3 Maior múltiplo comum – MMC

```
int mdc(int a, int b){
  return (b == 0 ? a : mdc(b, a%b)); //b == 0 ? Caso sim, retorne a, caso não, retorne mdc(b, a%b)
}

int mmc = (a*b)/mdc(a,b);
```

2.5 Operadores binários

2.5.1 OR nos Bits (|)

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

2.5.2 AND nos bits (&)

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

2.5.3 XOR nos bits (^)

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

2.5.4 Shift Esquerdo («)

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

2.5.5 Shift Direito (»)

```
// 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

```
// Note que uma potncia de 2 tem sempre apenas um bit igual a 1, dessa forma se queremos saber se o
    bit i est igual a 1 , precisamos apenas checar se o and dele e de 2^i  diferente de 0.

bool is_set(int x, int i) {
  bool ret = ((x&(1 << i)) != 0);
  return ret;
}
```

2.8 Extrair o bit menos significante

```
// O bit menos significativo de um nmero  menor bit de um nmero igual a 1, chamamos ele de lsb.
    Exemplo:
// lsb(20) = lsb(10100) = 100 = 4

int lsb(int x) {
  return x & -x;
}
```

2.9 Contar o número de bits iguais a 1

```
int count_bits(int x) {
  int ret = 0;
  while (x != 0) {
    ++ret;
    x -= x & -x;
  }
  return ret;
}
```

2.10 Checar se um número é potência de 2

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

2.11 Ligar um bit em um número

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

2.12 Desligar o bit

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

2.13 Divisão de números inteiros com resto negativo

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

```cpp
int a, b, c;
int q, r;

cin >> a >> b;

q = a / b;
r = a % b;

if (r < 0) {
  int c, d;
  c = (a < 0) ? a * -1 : a;
  d = (b < 0) ? b * -1 : b;

  q = (c + d) / d;
  r = (c - (q * d))*-1;

  q = (a*b > 0) ? q : q * -1;
}
```

2.14 Condição existência e classificação de triângulo

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

```cpp
int max(int a, int b) {
  return (a>b) ? a : b;
}
int min(int a, int b) {
  return (a<b) ? a : b;
}

cin >> a >> b >> c;

// x > y > z
int x, y, z;

x = max(a, max(b, c));
z = min(a, min(b, c));
// a+b+c = soma total | -x - z = total-(maior+menor)
y = a + b + c - x - z;

if (x < (y + z)) {
  if (x == y && y == z) {
    cout << "Valido-Equilatero" << endl;
  }
  else if (x != y && x != z && y != z) {
    cout << "Valido-Escaleno" << endl;
  }
  else {
    cout << "Valido-Isoceles" << endl;
  }
  // pitagoras x=y+z
  if ((x*x) == ((y*y) + (z*z))) {
    cout << "Retangulo: S" << endl;
  }
  else {
    cout << "Retangulo: N" << endl;
  }
}
else {
  cout << "Invalido" << endl;
}
```

2.15 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.

```cpp
bool comparaDouble(double val1, double val2, string cmp) {
  if (cmp == "==") {
    return fabs(val1 - val2) < EPSILON;
  }
  else if (cmp == "<=") {
    if (fabs(val1 - val2) < EPSILON) {
      return true;
    }
    else {
      return val1 <= val2;
    }
  }
  else if (cmp == ">=") {
    if (fabs(val1 - val2) < EPSILON) {
      return true;
    }
    else {
      return val1 >= val2;
    }
  }
}
```

2.16 Arredondamento para cima

```cpp
ceil(numero)
```

2.17 Número de casas decimais de um número

```cpp
ceil(log10(numero+1))
```

2.18 Zerar conteúdo de um array 2d

```cpp
memset(array, 0, sizeof(array[0][0]) * n * n)
```

2.19 Zerar conteúdo de um array 1d

```cpp
memset(array, 0, sizeof(array))
```

2.20 Cuidado para divisão de dois floats ou double

```cpp
// 1/6=0
// 1.0/6.0=0,1666667
```

2.21 Conversão inteiro para hexadecimal

```cpp
cout << hex << v
// Considerando v um inteiro
```

2.22 Adicionar notação cietífica

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

2.23 Adicionar casas decimais fixas

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

2.24 Volume do cilindro

$pi * r2 * h$

2.25 Área Total

$A = Ab + Al = 2 * \pi * r * (2 + h)$
$Ab = 2 * \pi * r2$
$Al = 2 * \pi * r * h$

2.26 Somatório de Feynman

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

2.27 Somatório de um intervalo [a,b] inclusivo

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

2.28 Distância entre 2 pontos

$sqrt(pow((xf - xi), 2) + pow((yf - yi), 2))$

2.29 Conversão cartesiano para polar

$r = \sqrt{}(a2 + b2)$
$\Phi = tg - 1b/a$

2.30 Conversão polar para cartesiano

$a = rcos$
$b = rsem$

2.31 Número de permutações de um conjunto

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

2.32 Número de combinações de um conjunto

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

2.33 Tricks do cmath

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

2.34 Máximo entre dois números

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

2.35 Mínimo entre dois números

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

2.36 A^b mod p

```
long powmod(long base, long exp, long modulus) {
    base %= modulus;
    long result = 1;

    while (exp > 0) {
        if (exp & 1) result = (result * base) % modulus;
            base = (base * base) % modulus;
        exp >>= 1;
    }

    return result;
}
```

2.37 n! mod p

```
int factmod (int n, int p) {
    long long res = 1;
    while (n > 1) {
        res = (res * powmod (p-1, n/p, p)) % p;
        for (int i=2; i<=n%p; ++i)
            res=(res*i) %p;
        n /= p;
    }
    return int (res % p);
}
```

3 Strings

3.1 Modificações

3.1.1 Dividir uma string de acordo com um token

```
std::string s = "scott>=tiger>=mushroom";
std::string delimiter = ">=";
size_t pos = 0;
std::string token;
while ((pos = s.find(delimiter)) != std::string::npos) {
  token = s.substr(0, pos);
  std::cout << token << std::endl;
  s.erase(0, pos + delimiter.length());
}
std::cout << s << std::endl;
```

3.1.2 Apagar um intervalo de uma string

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

3.1.3 Remover um caracter de toda a string

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

3.1.4 Inverter String

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

3.1.5 Substring

```
string str1 = line.substr(0,meio);
```

3.2 Verificações

3.2.1 Verificar se uma string está vazia

```
n.empty() // Retona true ou false
```

3.2.2 Verificar se caracter está entre [A-z]

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

3.3 Conversões

3.3.1 String para int

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

3.3.2 String para long long

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

3.3.3 String para unsigned int

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

3.3.4 String para unsigned long long

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

3.3.5 Char para int

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

3.3.6 Int para String

```
int a = 10;
stringstream ss;
ss << a;
string str = ss.str();
```

3.3.7 Caracteres minúsculos

```
tolower(char)
```

3.3.8 Caracteres maiúsculos

```
toupper(char)
```

3.4 Apagar um intervalo de uma string

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

3.5 Remover um caracter de toda a string

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

3.6 Verificar se uma string está vazia

```
n.empty() // Retona true ou false
```

3.7 Inverter String

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

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

```
string str1 = line.substr(0,meio);
```

3.9 Verificar se caracter está entre [A-z]

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

3.10 Busca [A-z]

```
unsigned int find(const string &s2, unsigned int pos1 = 0);
unsigned int rfind(const string &s2, unsigned int pos1 = end);
unsigned int find_first_of(const string &s2, unsigned int pos1 = 0);
unsigned int find_last_of(const string &s2, unsigned int pos1 = end);
unsigned int find_first_not_of(const string &s2, unsigned int pos1 = 0);
unsigned int find_last_not_of(const string &s2, unsigned int pos1 = end);
```

3.11 Insert, Erase, Replace

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

3.12 String Streams

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

4 Estruturas

4.1 Verificar se elemento existe em um vetor

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

4.2 Apagar elementos duplicados em um vetor

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

4.3 Ordenar vector forma crescente

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

4.4 Ordenar vector forma crescente

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

4.5 Excluir primeiro elemento de um vetor

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

4.6 Excluir último elemento de um vetor

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

4.7 Alterar tamanho de um vector

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

4.8 Busca em um vetor

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

4.9 Busca em um vetor

```
iterator find(iterator first, iterator last, const T &value);
iterator find_if(iterator first, iterator last, const T &value, TestFunction test);
bool binary_search(iterator first, iterator last, const T &value);
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

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

4.12 Leitura de um pair

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

4.13 Utilizando pair de pair

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

4.14 Criando pair com dois valores

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

4.15 Fila

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

4.16 Pilha

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

4.17 SET

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

4.18 Map

```
// 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.
map<string, int> M; // Declaração
M.insert(make_pair("Alana", 10)); //Inserimos uma variável do tipo pair diretamente no map
M["Alana"] = 10; // Relacionando o valor 10 à chave "Alana"
if(M.find("Alana") != M.end()) //Se a chave "Alana" foi inserida no map
cout<<M["Alana"]<<"\n"; //Imprime o valor da chave "Alana", no caso, o valor 10.
M.erase("Alana"); //Apaga o elemento que possui a chave "Alana" do map
// clear(): Apaga todos os elementos.
// size(): Retorna a quantidade de elementos.
// begin(): Retorna um ponteiro para o inicio do map
// end(): Retorna um ponteiro para o final do map
```

4.19 For em Map

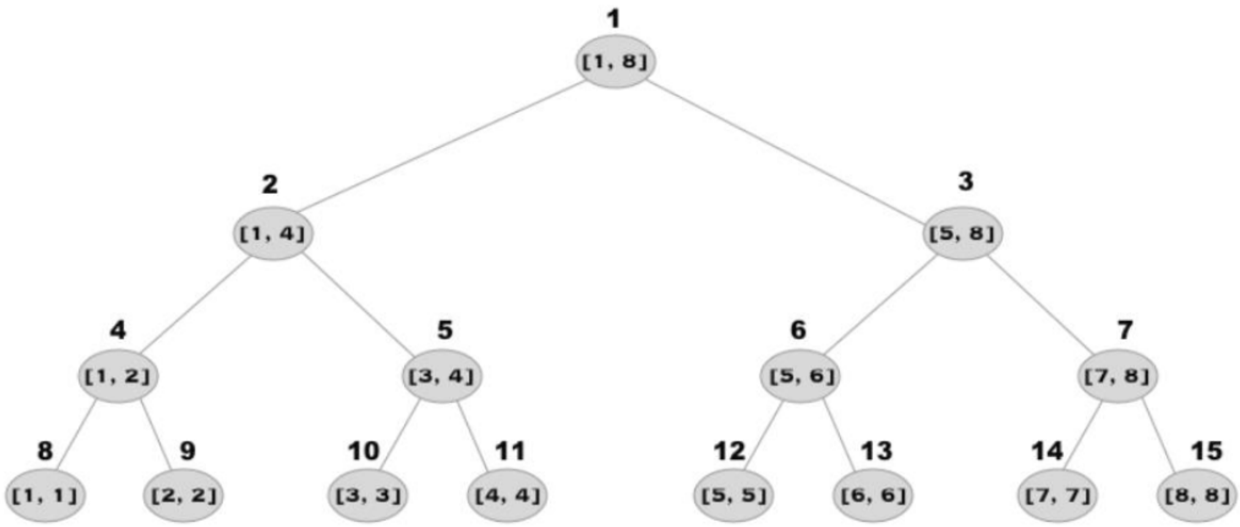
```
for (map<string,int>::iterator it=M.begin(); it!=M.end(); ++it){
  cout << "(" << it->first << ", " << it->second << ") ";
}
```

4.20 Fila de prioridades

```
priority_queue< pair<int, string> > pokemon;
pokemon.push(make_pair(poder, nome));
pokemon.top();
pokemon.pop();
```

4.21 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




```
void atualiza(int no, int i, int j, int posicao, int novo_valor){
  // se tivermos i = j, temos i = posicao = j. Logo, estamos no nó mais baixo da ávore
  if(i == j){
    arvore[no] = i;
    acao[posicao] = novo_valor;
  }
  else{
    int esquerda = 2*no; // índice do filho da esquerda
    int direita = 2*no + 1; // índice do filho da direita
    int meio = (i + j)/2;
    if(posicao <= meio) atualiza(esquerda, i, meio, posicao, novo_valor);
    else atualiza(direita, meio+1, j, posicao, novo_valor);
    if( acao[ arvore[esquerda] ] < acao[ arvore[direita] ] ) arvore[no] = arvore[esquerda];
    else arvore[no] = arvore[direita];
  }
}
int consulta(int no, int i, int j, int A, int B){
  if(A <= i && j <= B){
    return arvore[no];
  }
  if(i > B || A > j){
    return -1;
  }
  int esquerda = 2*no;
  int direita = 2*no + 1;
  int meio = (i + j)/2;
  int resposta_esquerda = consulta(esquerda, i, meio, A, B);
  int resposta_direita = consulta(direita, meio+1, j, A, B);
  if(resposta_esquerda == -1) return resposta_direita;
  if(resposta_direita == -1) return resposta_esquerda;
  if(acao[resposta_esquerda] < acao[resposta_direita]) return resposta_esquerda;
  else return resposta_direita;
}
```

4.22 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).

```
int soma(int x){
  int s = 0;
  // vamos reduzindo x até acabarmos (quando chegamos a zero)
  while(x > 0){
    s += arvore[x]; // adicionamos o pedaço de árvore atual à soma
    x -= (x & -x); // removemos o bit menos significante
  }
}
void atualiza(int x, int v){ // adicionar v frutas a caixa x
  while(x <= N){ // nosso teto, que é quando vamos parar de rodar o algoritmo
    arvore[x] += v; // adicionamos v frutas a arvore[x], como devemos
    x += (x & -x); // atualizamos o valor de x adicionado ele ao seu LSB
  }
}
```

4.23 Lazy Propagation

Você tem caixas N de frutas, númeradas 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].

```
void atualiza(int no, int i, int j, int a, int b, int valor){
  int esquerda = 2*no; // índice do filho da esquerda
  int direita = 2*no + 1; // índice do filho da direita
  int meio = (i + j)/2;
  if(lazy[no]){
    arvore[no] += lazy[no]*(j - i + 1);
    if(i != j){
      lazy[direita] += lazy[no];
      lazy[esquerda] += lazy[no];
    }
    lazy[no] = 0;
  }
  if(i > j || i > b || a > j) return;
  if(a <= i && j <= b){
    arvore[no] += valor*(j-i+1);
    if(i != j){
      lazy[direita] += valor;
      lazy[esquerda] += valor;
    }
  }
  else{
    // atualizamos o filho da esquerda
    atualiza(esquerda, i, meio, a, b, valor);
    // atualizamos o filho da direita
    atualiza( direita, meio+1, j, a, b, valor);
    // atualizamos o nó que estamos
    arvore[no] = arvore[esquerda] + arvore[direita];
  }
}

// queremos saber a soma de todos os valores de índice no intervalo [A, B]
int consulta(int no, int i, int j, int a, int b){
  int esquerda = 2*no; // índice do filho da esquerda
  int direita = 2*no + 1; // índice do filho da direita
  int meio = (i + j)/2;
  if(lazy[no]){
    arvore[no] += lazy[no]*(j - i + 1);
    if(i != j){
      lazy[direita] += lazy[no];
      lazy[esquerda] += lazy[no];
    }
    lazy[no] = 0;
  }
  if(i > j || i > b || a > j) return 0;
  if(a <= i && j <= b)
    return arvore[no];
  else{
    int soma_esquerda = consulta(esquerda, i, meio, a, b);
    int soma_direita = consulta( direita, meio+1, j, a, b);
    return soma_esquerda + soma_direita;
  }
}
```

4.24 Sort em structs

```
// Criar struct
typedef struct {
  int moradores;
  int gastoss;
  int media;
} Imovel;
// Definir comparator para a struct
bool cmp(Imovel const & x,Imovel const & y){
```

```
  if(x.media < y.media) {
    return true;
  }else {
    return false;
  }
}
// Efetuar o sort no main
Imovel imoveis[10];
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(N2), 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++:

```cpp
memset(grafo, 0, sizeof(grafo[0][0]) * 10 * 10)
int grafo[10][10];

grafo[1][2] = grafo[2][1] = 1;
grafo[1][3] = grafo[3][1] = 1;
grafo[1][6] = grafo[6][1] = 1;
grafo[2][3] = grafo[3][2] = 1;
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
1	{2, 3, 6}
2	{1, 3}
3	{1, 2}
4	{3}
5	{}
6	{1}

Implementação em C++:

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

using namespace std;
// 1 - Grafo sem pesos
map<int, vector<int> > adj_map;
// 2 - Grafo sem pesos (Usar de preferncialmente essa)
vector<int> adj[10];
// 3 - Grafo com pesos
vector<pair<int, int> > adj_peso[10];

// Adicionar aresta usando representao 1
// Supondo que grafo seja no direcionado
void addEdge(map<int, vector<int> > adj_map, int u, int v){
  adj_map[u].push_back(v);
  adj_map[v].push_back(u);
}
// Adicionar aresta usando representao 2
// Supondo que grafo seja no direcionado
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}

// Adicionar aresta usando representao 3
// Supondo que grafo seja no direcionado
void addEdge(vector<pair<int, int> > adj[], int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(v, w));
}


int main(){
  // 1
  addEdge(adj_map, 0, 1);
  addEdge(adj_map, 0, 2);
  addEdge(adj_map, 2, 3);

  // 2
  addEdge(adj, 0, 1);
  addEdge(adj, 0, 2);
  addEdge(adj, 2, 3);

  // 3
  addEdge(adj_peso, 0, 1, 10);
  addEdge(adj_peso, 0, 1, 5);
  addEdge(adj_peso, 0, 1, 15);

  return 0;
}
```

5.2 Lista de Arestas

```
struct t_aresta{
  int dis;
  int x, y;
};

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.

```
void addEdge(vector<int> adj[], int u, int v, int w)
{
    adj[u].pb(mp(v, w));
    adj[v].pb(mp(v, w));
}

// DFS não leva em consideração pesos dos grafos, por conta disso,
// procurar usar representação do grafo que não precisa de peso.

// Função para realizar DFS recursivamente
// no grafo a partir do vértice u.
void DFSUtil(int u, vector<int> adj[],
                    vector<bool> &visited)
{
    visited[u] = true;
    cout << u << " ";
    for (int i=0; i<adj[u].size(); i++)
        if (visited[adj[u][i]] == false)
            DFSUtil(adj[u][i], adj, visited);
}

// Realiza DFSUtil() em todos
// os vértices não visitados.
void DFS(vector<int> adj[], int V)
{
    vector<bool> visited(V, false);
    for (int u=0; u<V; u++)
        if (visited[u] == false)
            DFSUtil(u, adj, visited);
}


int main(){
    int V = 5;
    // lista de adjacência
    vector<int> adj[V];

    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    DFS(adj, V);
    return 0;
}
```

5.3.2 BFS(Busca em largura)

```
// Função utilitária para adicionar arestas
// em um grafo não direcionado.
// u = origem
// v = destino
void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].pb(v);
    adj[v].pb(u);
}

```

```cpp
// BFS não leva em consideração pesos dos grafos, por conta disso,
// usar representação do grafo que não precisa de peso.
void BFS(vector<int> adj[], int s, int vertices) {
    // Marca todos os vértices como não visitado
    vector<bool> visited(vertices, false);
    for(int i = 0; i < vertices; i++)
        visited[i] = false;

    // Cria uma fila para BFS
    list<int> queue;

    // Marca o vértice atual como visitado, e adiciona à fila
    visited[s] = true;
    queue.push_back(s);

    // 'i' será usado para obter todos os
    // vértices adjacentes de um vértice
    vector<int>::iterator i;

    while(!queue.empty())
    {
        // tira um vértice da lista e printa ele
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // pega todos os vértices adjancetes do vértice
        // que saiu da fila. Se o adjacente não foi visitado,
        // então marca como visitado, e enfilera.
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

int main(){
    int vertices = 5;
    vector<int> adj[vertices];

    addEdge(adj, 0, 1);
    addEdge(adj, 0, 4);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 3);
    addEdge(adj, 3, 4);
    BFS(adj, 2, vertices);

    return 0;
}
```

5.3.3 Dijkstra - Caminho Mínimo entre dois pontos

Algoritmo 1: "Lista de Adjacência"

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

using namespace std;

#define INFINITO 10000000

void addEdge(vector<pair<int, int> > adj[], int v1, int v2, int custo)
{
    adj[v1].push_back(make_pair(v2, custo));
}

int dijkstra(vector<pair<int, int> > adj[], int orig, int dest, int vertices)
{
    // vetor de distâncias
    int dist[vertices];

    /*
        vetor de visitados serve para caso o vértice já tenha sido
        expandido (visitado), não expandir mais
    */
    int visitados[vertices];

    // fila de prioridades de pair (distancia, vértice)
```

```
    priority_queue < pair<int, int>,
                vector<pair<int, int> >, greater<pair<int, int> > > pq;

    // inicia o vetor de distâncias e visitados
    for(int i = 0; i < vertices; i++)
    {
        dist[i] = INFINITO;
        visitados[i] = false;
    }

    // a distância de orig para orig é 0
    dist[orig] = 0;

    // insere na fila
    pq.push(make_pair(dist[orig], orig));

    // loop do algoritmo
    while(!pq.empty())
    {
        pair<int, int> p = pq.top(); // extrai o pair do topo
        int u = p.second; // obtém o vértice do pair
        pq.pop(); // remove da fila

        // verifica se o vértice não foi expandido
        if(visitados[u] == false)
        {
            // marca como visitado
            visitados[u] = true;

            vector<pair<int, int> >::iterator it;

            // percorre os vértices "v" adjacentes de "u"
            for(it = adj[u].begin(); it != adj[u].end(); it++)
            {
                // obtém o vértice adjacente e o custo da aresta
                int v = it->first;
                int custo_aresta = it->second;

                // relaxamento (u, v)
                if(dist[v] > (dist[u] + custo_aresta))
                {
                    // atualiza a distância de "v" e insere na fila
                    dist[v] = dist[u] + custo_aresta;
                    pq.push(make_pair(dist[v], v));
                }
            }
        }
    }

    // retorna a distância mínima até o destino
    return dist[dest];
}

int main()
{
    int vertices = 10;
  vector<pair<int, int> > adj[vertices];

  addEdge(adj, 0, 1, 4);
  addEdge(adj, 0, 2, 2);
  addEdge(adj, 0, 3, 5);
  addEdge(adj, 1, 4, 1);
  addEdge(adj, 2, 1, 1);
  addEdge(adj, 2, 3, 2);
  addEdge(adj, 2, 4, 1);
  addEdge(adj, 3, 4, 1);

  cout << dijkstra(adj, 1, 3, vertices) << endl;

  return 0;
}
```

Algoritmo 2: "Matriz de Adjacência"

```
#define MAX 999999999
#define max 501

using namespace std;

int g[max][max], vertices;

int dijkstra(int origem, int destino) {
    int minimo, atual;
    int passou[max], pred[max],custo[max];
    for (int i=1; i<=vertices;i++) {
        pred[i]=-1;
```

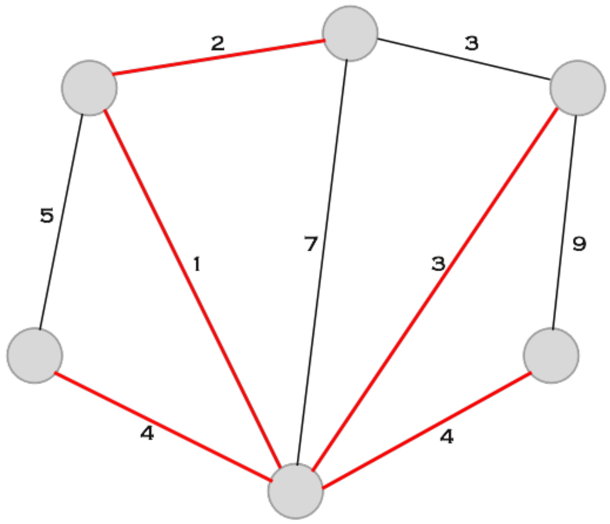
```
        custo[i]=MAX;
        passou[i]=0;
    }
    custo[origem] = 0;
    atual = origem;
    while (atual != destino) {
        for (int i=1; i<=vertices;  i++) {
            if (custo[i] > custo[atual] + g[atual][i]) {
                custo[i] = custo[atual] + g[atual][i];
            }
        }
        minimo = MAX;
        passou[atual]=1;
        for (int i=1; i<=vertices; i++) {
            if ((custo[i]<minimo) && (!passou[i])) {
                minimo = custo[i];
                atual = i;
            }
        }
        // caso nao consiga ir a lugar algum saindo da origem
        if (minimo == MAX) {
            return MAX;
        }
    }
    return custo[destino];
}
```

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.**

```
struct t_aresta{
  int dis;
  int x, y;
};
bool comp(t_aresta a, t_aresta b){ return a.dis < b.dis; }
//--------------------
#define MAXN 50500
#define MAXM 200200

int n, m; // número de vértices e arestas
t_aresta aresta[MAXM];
// para o union find
int pai[MAXN];
int peso[MAXN];
// a árvore
t_aresta mst[MAXM];
//--------------------
// funções do union find
int find(int x){
  if(pai[x] == x) return x;
  return pai[x] = find(pai[x]);
}

void join(int a, int b){
  a = find(a);
  b = find(b);
  if(peso[a] < peso[b]) pai[a] = b;
  else if(peso[b] < peso[a]) pai[b] = a;
  else{
    pai[a] = b;
    peso[b]++;
  }
}

int main(){
  // ler a entrada
  cin >> n >> m;
  for(int i = 1;i <= m;i++)
    cin >> aresta[i].x >> aresta[i].y >> aresta[i].dis);

  // inicializar os pais para o union-find
  for(int i = 1;i <= n;i++) pai[i] = i;
    // ordenar as arestas
    sort(aresta+1, aresta+m+1, comp);

  int size = 0;
  for(int i = 1;i <= m;i++){
    if( find(aresta[i].x) != find(aresta[i].y) ){ // se estiverem em componentes distintas
      join(aresta[i].x, aresta[i].y);
      mst[++size] = aresta[i];
```

```
    }
  }

  // imprimir a MST
  for(int i = 1;i < n;i++)
    cout << mst[i].x << " " << mst[i].y << " " << mst[i].dis) << "\n";
  return 0;
}
```



5.3.5 Prim - Árvore Geradora Mínima

Algoritmo parecido com Dijkstra para encontrar árvore geradora mínima.
Implementação em C++:

```
typedef pair<int, int> pii;
#define MAXN 10100
#define INFINITO 999999999
int n, m; // número de vértices e arestas
int distancia[MAXN]; // o array de distâncias à fonte
int processado[MAXN]; // o array que guarda se um vértice foi processado
vector<pii> vizinhos[MAXN]; // nossas listas de adjacência. O primeiro elemento do par representa a
    distância e o segundo representa o vértice
int Prim(){
  for(int i = 2;i <= n;i++)
    distancia[i] = INFINITO;
  distancia[1] = 0;
  priority_queue< pii, vector<pii>, greater<pii> > fila;
  fila.push( pii(distancia[1], 1) );
  while(true){
    int davez = -1;
    while(!fila.empty()){
      int atual = fila.top().second;
      fila.pop();
      if(!processado[atual]){
        davez = atual;
        break;
      }
    }
    if(davez == -1) break;
    processado[davez] = true;
    for(int i = 0;i < (int)vizinhos[davez].size();i++){
      int dist = vizinhos[davez][i].first;
      int atual = vizinhos[davez][i].second;
      if( distancia[atual] > dist && !processado[atual]){
        distancia[atual] = dist;
        fila.push( pii(distancia[atual], atual) );
      }
    }
  }
  int custo_arvore = 0;
  for(int i = 1;i <= n;i++)
    custo_arvore += distancia[i];
  return custo_arvore;
}

int main(){
```

```
  cin >> n >> m;
  for(int i = 1;i <= m;i++){
    int x, y, tempo;
    cin >> x >> y >> tempo;
    vizinhos[x].push_back( pii(tempo, y) );
    vizinhos[y].push_back( pii(tempo, x) );
  }
  cout << Prim() << endl;
  return 0;
}
```

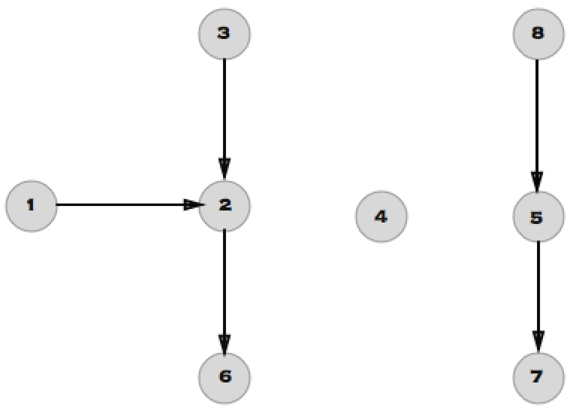
5.3.6 Ordenação Topológica

"*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 do 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 a 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 sobrarem vértices, há um ciclo e não há ordem para resolver o problema. Caso contrário, o problema está resolvido.
Implementação em C++:

```cpp
#include <vector>
#include <iostream>
using namespace std;
//----------------------------------
#define MAXN 100100
int n; // número de vértices
int m; // número de arestas
vector<int> grafo[MAXN];
int grau[MAXN];
vector<int> lista; // dos vértices de grau zero
//----------------------------------
int main(){

  cin >> n >> m;

  for(int i = 1;i <= m;i++){
    int x, y;
    cin >> x >> y;

    // tarefa X tem que ser executada antes da tarefa Y
    grau[y]++;
    grafo[x].push_back(y);
  }

  for(int i = 1;i <= n;i++) if(grau[i] == 0) lista.push_back(i);

  // o procedimento a ser feito é semelhante a uma BFS
  int ini = 0;
  while(ini < (int)lista.size()){

    int atual = lista[ini];
    ini++;

    for(int i = 0;i < (int)grafo[atual].size();i++){
      int v = grafo[atual][i];
      grau[v]--;
      if(grau[v] == 0) lista.push_back(v); // se o grau se tornar zero, acrescenta-se a lista
    }
  }

  // agora, se na lista não houver N vértices,
  // sabemos que é impossível realizar o procedimento

  if((int)lista.size() < n) cout << "impossivel\n";
  else{
    for(int i = 0;i < (int)lista.size();i++) cout << lista[i] << " ";
    cout << endl;
  }

  return 0;
}
```

5.3.7 Floyd-Warshall - Menor Caminho

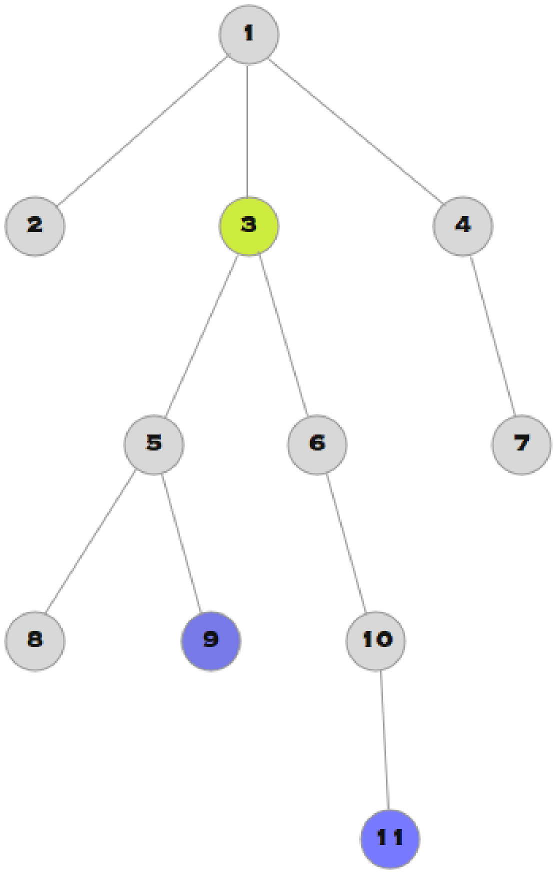
Menor distância de qualquer vértice para qualquer outro.

```cpp
for(int k = 1;k <= n;k++)
    for(int i = 1;i <= n;i++)
        for(int j = 1;j <= n;j++)
          distancia[i][j] = min(distancia[i][j], distancia[i][k] + distancia[k][j]);

#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
  for vertices not connected to each other */
```

```c
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[][] will be the output matrix that will finally have the shortest
      distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
      ---> Before start of an iteration, we have shortest distances between all
      pairs of vertices such that the shortest distances consider only the
      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of an iteration, vertex no. k is added to the set of
      intermediate vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on the shortest path from
                // i to j, then update the value of dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("The following matrix shows the shortest distances"
            " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { {0,   5,  INF, 10},
                        {INF, 0,   3, INF},
                        {INF, INF, 0,   1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}
```

5.3.8 LCA - Menor Ancestral Comum

Ancestral comum mais próximo à dois vértices.




```
int LCA(int u, int v){

  if(nivel[u] < nivel[v]) swap(u, v); // isto é para definir u como estando mais abaixo

  // vamos agora fazer nivel[u] ser
  // igual nivel[v], subindo pelos
  // ancestrais de u

  for(int i = MAXL-1;i >= 0;i--)
    if(nivel[u] - (1<<i) >= nivel[v])
      u = ancestral[u][i];

  // agora, u e v estão no mesmo nível
  if(u == v) return u; // se eles forem o mesmo nó já achamos nossa resposta

  // subimos o máximo possível de forma
  // que os dois NÃO passem a ser iguais

  for(int i = MAXL-1;i >= 0;i--)
    if(ancestral[u][i] != -1 && ancestral[u][i] != ancestral[v][i]){
      u = ancestral[u][i];
      v = ancestral[v][i]
    }

  // como subimos o máximo possível
  // sabemos que u != v e que pai[u] == pai[v]
  // logo, LCA(u, v) == pai[u] == pai[v]

  return ancestral[u][0];
}
```

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:

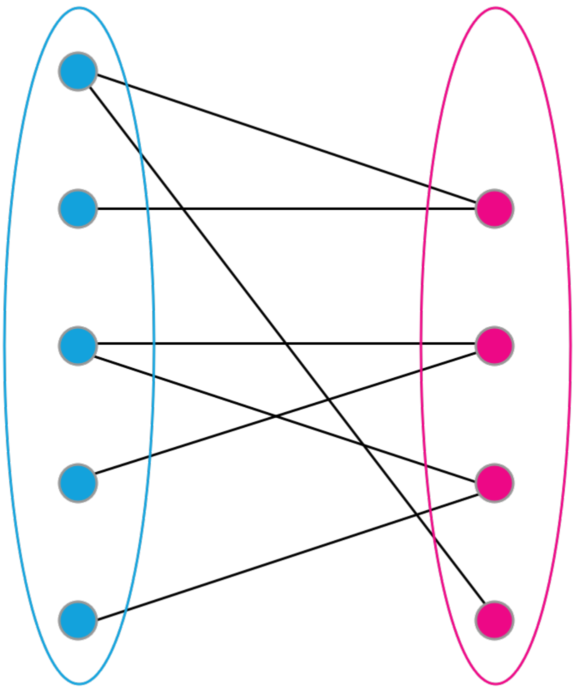
- Todos os vértices do grafo tem que ter grau par.
- Todos os vértices (ignorando-se os de grau zero) tem grau par, exceto dois vértices que possuem grau ímpar. Nesse caso, os dois vértices de grau ímpar são o início e o fim do Caminho.

Implementação em C++:

```
vector<int> caminho; // guardará nosso Caminho Euleriano (invertido)
vector<int> vizinhos[MAXN]; // nossa lista de adjacência
map< pair<int, int>, bool > deletada; // mapa que checa se a aresta já foi deletada
void acha_caminho(int v){
  for(int i = 0;i < (int)vizinhos[v].size();i++){
    int viz = vizinhos[v][i];
    if( deletada[make_pair(v, viz)] == true )
      continue;
    deletada[make_pair(v, viz)] = true;
    deletada[make_pair(viz, v)] = true;
    acha_caminho(viz);
  }
  caminho.push_back(v);
}
```

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++:

```
int n; // número de vértices
vector<int> vizinhos[MAXN]; // a lista de adjacência de cada vértice
int cor[MAXN];
// a cor de cada vértice. Inicialmente, cor[i] = -1 para todos os vértices.
// definimos cor[i] = 0 como sendo azul e cor[i] = 1 como sendo rosa.
void colore(int x){
  cor[x] = 0;
  vector<int> fila;
  fila.push_back(x);
  int pos = 0;
  while(pos < (int)fila.size()){ // BFS
    int atual = fila[pos];
    pos++;
    for(int i = 0;i < (int)vizinhos[atual].size();i++){
      int v = vizinhos[atual][i];
      if(cor[v] == -1){
        cor[v] = 1 - cor[atual];
        fila.push_back(v); // adicionamos v a fila da BFS
      }
    }
  }
}
bool checa_bipartido(){
  for(int i = 1;i <= n;i++){
    if(cor[i] == -1){
      colore(i);
    }
  }
  for(int i = 1;i <= n;i++){
    for(int j = 0;j < (int)vizinhos[i].size();j++){
      int v = vizinhos[i][j];
      if(cor[i] == cor[v])
        return false;
    }
  }
  return true;
}
```

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:

$$\max \sum_{i=1}^{I} v_i, \tag{1}$$

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

$$\sum_{i=1}^{I} p_i \leq C \tag{2}$$

```c
#include <stdio.h>

// Retorna o máximo de dois números.
int max(int a, int b) {
    return a>b?a:b;
}

int main() {

    // Número de itens
    int n_itens = 3;

    // Vetor para armazenar o valor de cada item.
    int valor[] = {3, 4, 5};

    // Vetor para armazenar o peso de cada item.
    int peso[] = {2, 3, 2};

    // Capacidade da mochila.
    int capacidade = 6;

    // Matriz de memorização.
    int memo[capacidade+1][n_itens+1];

    int w,j;
    for(w = 0; w <= capacidade; ++w) {
        for(j = 0; j <= n_itens; ++j) {
            if(j == 0 || w == 0) {
                memo[w][j] = 0;
            } else {
                memo[w][j] = memo[w][j-1];
                if(peso[j-1] <= w) {
                    // -----------------------------------------
                    // Caso em que colocamos apenas uma vez um item
                    memo[w][j] = max(memo[w][j-1],
                        memo[w-peso[j-1]][j-1]+valor[j-1]);
                    // -----------------------------------------

                    // -------------------------------------------------
                    // Caso em que um item pode ser colocado mais de uma vez
                    //memo[w][j] = max(memo[w][j],
                    //    memo[w-peso[j-1]][j]+valor[j-1]);
                    // -------------------------------------------------
                }
```

```
            }
        }
    }

    // Solução ótima está no último elemento da matriz de memorização.
    printf("Valor ótimo = %d",memo[capacidade][n_itens]);

    return 0;
}
```

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?

```
#include <stdio.h>

int main() {

    // tamanho da haste.
    int n = 5;

    // Preço para cada tamanho de corte.
    // ex. para o tamanho 1 o preço é 2.
    int precos_corte[] = {2,4,3,1,5};

    int i, j;

    // Vetor de memorização.
    int memo[n+1];
    memo[0] = 0;    // Solução trivial - corte de tamanho zero com preço zero.

    for(i = 1; i <= n; ++i) {
        int q = -1;
        for(j = 1; j <= i; ++j) {
            if(q < (precos_corte[j-1] + memo[i-j])) {
                q = precos_corte[j-1]+memo[i-j];
            }
        }
        memo[i] = q;
    }

    // Solução está na última posição do vetor de memorização:
    printf("Valor de venda = %d",memo[n]);

    return 0;
}
```

6.2.2 Dado valor e as moedas existe troco possível?

```
// Lembre-se de inicializar todos os valores da DP como -1 (não calculado) e, para cada estado, ela
    retornará 1, se for true, ou 0, caso seja false
int dp[MAX];

// F função retorna true assim que encontramos uma maneira de formar o valor x. Se testarmos todas
    as moedas e nenhuma funcionar, ela retorna false
int solve(int x, vector<int> &c){

  if(x==0) return 1;

  if(x<0) return 0;

  if(dp[x]>=0) return dp[x];

  for(int i=0;i<c.size();i++)
    if(solve(x-c[i])) return dp[x-c[i]]=1;

  return dp[x]=0;
}
```

6.2.3 Mínimo de moedas para troco

```
// função que recebe o valor de troco N, o número de moedas disponíveis M,
// e um vetor com as moedas disponíveis m
// essa função deve retornar o número mínimo de moedas,
// de acordo com a solução com Programação Dinamica.
int num_moedas(int N, int M, int * m) {
    int dp[N+1];

    // caso base
    dp[0] = 0;

    // sub-problemas
    for(int i=1; i<=N; i++) {
        // é comum atribuir um valor alto, que concerteza
        // é maior que qualquer uma das próximas possibilidades,
        // sendo assim substituido
        dp[i] = 1000000;

        for(int j=0; j<M; j++) {
            if(i-m[j] >= 0) {
                dp[i] = min(dp[i], dp[ i-m[j] ]+1);
            }
        }
    }

    // solução
    return dp[N];
}
```

6.3 Contagem de inversões

Um dos problemas mais clássicos de programação é a contagem de inversões em uma sequência. De maneira simples, seja S = a1,a2,...,an. Uma inversão em S é um par (i,j), com i<j, tal que ai>aj. Sabendo disso, faça um programa que calcula o número de inversões em uma sequência S.

```
#define INF 1000000000

// função merge_sort que ordena um vector v
int merge_sort(vector<int> &v){

  // declaro inv, o total de inversões
  int inv=0;

  // se o tamanho de v for 1, não há inversões
  if(v.size()==1) return 0;

  // se não

  // declaro os vetore u1 e u2
  vector<int> u1, u2;

  // e faço cada um receber uma metade de v
  for(int i=0;i<v.size()/2;i++){
    u1.push_back(v[i]);
  }
  for(int i=v.size()/2;i<v.size();i++){
    u2.push_back(v[i]);
  }
  // ordeno u1 e u2
  // e adiciono a inv as inversões de cada metade do vetor
  inv+=merge_sort(u1);
  inv+=merge_sort(u2);

  // e adiciono INF ao final de cada um deles
  u1.push_back(INF);
  u2.push_back(INF);

  // declaro ini1 e ini2 com valore inicial zero
  int ini1=0, ini2=0;

  // percorro cada posição de v
  for(int i=0;i<v.size();i++){

    // se o menor não usado de u1 for menor o mesmo em u2
    if(u1[ini1]<=u2[ini2]){

      // então o coloco em v
      v[i]=u1[ini1];

      // e incremento o valor de ini1
```

```
      ini1++;
    }

    // caso contrário, faço o análogo com u2 e ini2
    else{

      v[i]=u2[ini2];
      ini2++;

      // não se esquecendo de adicionar o número de elementos em u1
      // ao total deinversões em v
      inv+=u1.size()-ini1-1;
    }
  }

  // por fim, retorno a quantidade de inversões
  return inv;
}
```

6.4 Maior Subsequência Comum

Dadas duas sequências s1 e s2, uma de tamanho n e outra de tamanho m, qual a maior subsequência comum às duas? Lembre-se que uma subsequência de s1, por exemplo, é simplesmente um subconjunto dos elementos de s1 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

```
using namespace std;

// defino MAXN como 1010
#define MAXN 1010

// declaro as variáveis que vou usar
int s1[MAXN], s2[MAXN], tab[MAXN][MAXN];

int lcs(int a, int b){ // declaro a função da DP, de nome lcs

  // se já calculamos esse estado da dp antes
  if(tab[a][b]>=0) return tab[a][b]; // retornamos o valor salvo para ele

  // se uma das sequências for vazia, retornamos zero
  if(a==0 or b==0) return tab[a][b]=0;

  // se s1[a] for igual a s2[b], os retiramos das sequências
  if(s1[a]==s2[b]) return 1+lcs(a-1, b-1); // e adicionamos ele à lcs das subsequâncias restantes

  // se forem diferentes, retorno o máximo entre retirar s1[a] ou s1[b]
  return tab[a][b]=max(lcs(a-1, b), lcs(a, b-1));
}
```

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

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if (X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";

  int m = strlen(X);
  int n = strlen(Y);

```

```
  printf("Length of LCS is %d", lcs( X, Y, m, n ) );

  return 0;
}
```

6.5 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.

```
#define PB push_back // por simplicidade
#define MAXN 100100 // defino o valor de MAXN

vector<int> lis(vector<int> &v){

  // declaro s variáveis que vou usar
  vector<int> pilha, resp;
  int pos[MAXN], pai[MAXN];

  // para cada elemento
  for(int i=0; i<v.size(); i++){

    // declaro um iterador que guardará o elemento mais à esquerda de pilha
    // que não é menor que v[i]
    vector<int>::iterator it = lower_bound(pilha.begin(), pilha.end(), v[i]);

    // guardo a posição da pilha em que adicionarei o elemento
    int p = it-pilha.begin();

    // se it for o fim do vector, então não há elemento que não seja menor que v[i]
    // ou seja, todos os topos de pilha são menores ou iguais a v[i]

    // logo, criamos uma nova pilha e colocamos x no seu topo
    if(it==pilha.end()) pilha.PB(v[i]);

    // porém, se it apontar para alguma posição válida do vector
    // colocamos v[i] no topo desta pilha, substintuindo o valor que it aponta por v[i]
    else *it = x;

    // a posição original na sequência do número no topo da pilha p agora é i
    pos[p]=i;

    // se o elemento foi inserido na primeira pilha
    if(p==0) pai[i]=-1; // seu pai será -1

    // caso contrário, seu pai será a posição do elemento no topo da pilha anterior a ele
    else pai[i]=pos[p-1];
  }

  // p será a posição do elemento no topo da última pilha
  int p = pos[pilha.size()-1];

  // enquanto p não for -1
  while(p>=0){

    // adiciono o elemento na posição p à resposta
    resp.PB(v[p]);

    // e vou para o pai de p
    p=pai[p];
  }

  // inverto a ordem da resposta
  reverse(resp.begin(), resp.end());

  // por fim, retorno o vetor resp
  return resp;
}
```

6.6 Soma máxima em um intervalo

Dada uma sequência qualquer S=(s1,s2,s3,...,sn) 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).

```
int max_sum(vector<int> s){

  int resp=0, maior=0;

  for(int i=0;i<s.size();i++){

    maior=max(0,maior+s[i]);

    resp=max(resp,maior);
  }

  return resp;
}
```

6.7 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 seguranças 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.

```
int VertexCover(X, pai_colorido){

  if(PD[X][pai_colorido] != -1)
    return PD[X][pai_colorido]; // se já calculamos esse caso, retornamos o valor para evitar recálculo
                                // não se esqueça de, não função main, inicializar todos os valores de PD como -1.

  int caso1 = 1, caso2 = 0;

  for(int i = 0;i < (int)vizinhos[X].size();i++){ // percorremos todos os vizinhos de X

    int V = vizinhos[X][i];

    if(V == pai[X]) continue; // checamos se V é o pai de X

    // agora, sabemos que V é um filho de X
    pai[V] = X; // definimos o pai de V como sendo X
    caso1 += VertexCover(V, true);  // caso escolhamos colorir X
    caso2 += VertexCover(V, false); // caso escolhamos não-colorir X
  }

  if(pai_colorido == true) PD[X][pai_colorido] = min(caso1, caso2);// caso o pai de X esteja colorido, escolhemos o melhor caso
  if(pai_colorido == false) PD[X][pai_colorido] = caso1;          // caso o pai de X não seja colorido, escolhemos o caso1

  return PD[X][pai_colorido]; // retornamos o valor da resposta
}
```

7 Outros

7.1 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	

7.2 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	$\approx -1.1 \times \times 10^{4932}$	$\approx 1.1 \times 10^{4932}$	$\approx$ 18 digits

7.3 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>

7.4 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

7.5 Antes e depois de cristo

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

7.6 Problemas que envolvem horário

Procurar sempre usar minutos/segundos

7.7 Ano bissexto

Considerar 366 dias

7.8 Ano normal

Considerar 365 dias

7.9 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

7.10 Número de letras no alfabeto

26, contando K, W e Y

7.11 Forma alternativa para escrita de nome para tipos de dados

```
long int == long
long long int == long long
unsigned int == unsigned
unsigned long long int == unsigned long long
```

7.12 Inicializar vetor com valor predefinido

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

7.13 Operações para modificar sequências

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

7.14 Permutações

```
bool next_permutation(iterator first, iterator last);
bool next_permutation(iterator first, iterator last, LessThanOrEqualFunction comp);
bool prev_permutation(iterator first, iterator last);
bool prev_permutation(iterator first, iterator last, LessThanOrEqualFunction comp);
```

7.15 Gerar números aleatórios

```
srand(time(NULL));
// generate random numbers between [a,b)
rand() % (b - a) + a;
// generate random numbers between [0,b)
rand() % b;
// generate random permutations
random_permutation(anArray, anArray + 10);
random_permutation(aVector, aVector + 10);
```

7.16 Pesquisa binária

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

```
  {
    int mid = l + (r - l)/2;
    if (arr[mid] == x)
    return mid;
    if (arr[mid] > x)
    return binarySearch(arr, l, mid-1, x);
    return binarySearch(arr, mid+1, r, x);
  }
  return -1;
}
```