Sumário

Estratégias para a competição

Início da prova

1. Um procura no inicio, outro no meio e um no final
2. Quem achar a questão mais fácil começa no computador
3. É preferível demorar alguns minutos a tomar uma penalidade

Durante a prova

1. Ao ler um problema, **destaque** as partes mais importantes
2. Fique de olho nos *clarifications*
3. Determinar um tempo máx. que uma pessoa pode usar o PC
4. Caso o computador esteja ocupado, escreva no papel

Discussão de problemas

1. Apresentar uma solução para quem não pensou uma solução mata a criatividade

2. Discutir uma solução com alguém que pensou em outra solução faz com que cada um aponte defeitos na solução do colega, de modo que se cheguem em uma solução ótima
3. Pensem no menor caso de teste/menor resposta possível

Testando

1. Gaste algum tempo testando o seu programa
2. Testar os *casos de borda*
3. Definir um limite de tempo para os testes
4. Procure exceções (números negativos, impares, pares, 0, ...)

Submissão

1. **Sempre** faça a impressão do código logo após a submissão, *submit and print*

Wrong Answer

1. **NÃO ENTRE EM PÂNICO**
2. Analise o código impresso no papel
3. Leia o enunciado novamente, preste nos detalhes, limites e *overflows*
4. Utilize o python para gerar entradas aleatórias para o problema
5. Veja na tabela quem já leu o problema e descreva o algoritmo para a pessoa (pato)
    - Obs.: **NÃO** faça isso sem que a pessoa pense em uma solução por si só
6. Use o teste de mesa, ele funciona 😉

Template

Template para as questões que fazemos em C++

```
/**
 * [Link]
 * [Assuntos]
 */

#include <bits/stdc++.h>

using namespace std;

#define SPEED cin.tie(0)->sync_with_stdio(0);
#define DEBUG true
#define db(x) \
    if (DEBUG) cout << #x << ": " << x << endl
#define dbpair(x) \
    if (DEBUG) cout << #x << ": " << x.f << ", " << x.s << endl
#define dbvector(vector)                              \
    {                                                 \
        cout << #vector << " = ";                     \
        for (auto& it : vector) cout << it << " "; \
        cout << endl;                                 \
    }
#define dbmap(map)                           \
    for (auto e : map)                       \
        cout << e.first << " " << e.second; \
    cout << endl
#define all(x) begin(x), end(x)
#define pb push_back
#define pf push_front
#define endl "\n"
#define f first
#define s second
#define MOD 1e9 + 7
#define mp make_pair

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef long double ld;
typedef priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> min_pq;

const int INF = 0x3f3f3f3f;
const ll LINF = 0x3f3f3f3f3f3f3f3fll;
```

```
void solve() {
}

int main(int argc, char** argv) {
    SPEED;

    /**
     * Não esqueça de adicionar o link da questão e o assunto S2
     */

    int t;
    cin >> t;

    while (t--) solve();

    return 0;
}
```

Limites

Big O

N <=	O(máx)
11	O(n!)
22	$O(2^n * n)$
100	$O(n^4)$
400	$O(n^3)$
2000	$O(n^2 * \log_2(n))$
10^4	$O(n^2)$
10^5	$O(n * \log_2^2(n))$
10^6	$O(n * \log_2(n))$
10^8	O(n)
10^{18}	$O(\log_2(n))$, O(1)

Tipos de dados

Table 2-6 Integer Data Types

Data Type	Typical Size	Typical Range
`short int`	2 bytes	−32,768 to +32,767
`unsigned short int`	2 bytes	0 to +65,535
`int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long int`	4 bytes	−2,147,483,648 to +2,147,483,647
`unsigned long int`	4 bytes	0 to 4,294,967,295
`long long int`	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`	8 bytes	0 to 18,446,744,073,709,551,615

Tipo	Formato	Bits	Mínimo	Máximo	Precisão Decimal
char	%c	8	0	255	2
signed char	%hhd	8	-128	127	2
unsigned char	%hhu	8	0	255	2
short	%hd	16	-32,768	32,767	4
unsigned short	%hu	16	0	65,535	4
int	%d	32	-2 × 10^9	2 × 10^9	9
unsigned int	%u	32	0	4 × 10^9	9
long long	%lld	64	-9 × 10^18	9 × 10^18	18
unsigned long long	%llu	64	0	18 × 10^18	19

Tipo	Formato	Bits	Expoente	Precisão Decimal
float	%f	32	38	6
double	%lf	64	308	15
long double	%Lf	80	19.728	18

STL

Sorting

Selection Sort:

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}
```

Insertion Sort:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Merge Sort:

```
void merge(int arr[], int left, int middle, int right) {
    // Merge two sub-arrays into a temporary array and then copy back
    // to the original array
    // ...
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;
        mergeSort(arr, left, middle);
```

```
        mergeSort(arr, middle + 1, right);
        merge(arr, left, middle, right);
    }
}
```

Quick Sort:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; ++j) {
        if (arr[j] < pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Counting Sort:

```
void countingSort(int arr[], int n, int range) {
    std::vector<int> count(range + 1, 0); // Create a count array with size 'range
+ 1'

    // Count the occurrences of each element
    for (int i = 0; i < n; ++i) {
        ++count[arr[i]];
    }

    // Update the count array to store the cumulative count
    for (int i = 1; i <= range; ++i) {
        count[i] += count[i - 1];
    }

    // Build the sorted output array using the count array
    std::vector<int> output(n);
    for (int i = n - 1; i >= 0; --i) {
        output[count[arr[i]] - 1] = arr[i];
        --count[arr[i]];
```

```
    }

    // Copy the sorted array back to the input array
    for (int i = 0; i < n; ++i) {
        arr[i] = output[i];
    }
}
```

Stack

Uma pilha é uma estrutura de dados que oferece duas operações em tempo O(1): adicionar um elemento ao topo e remover um elemento do topo. É possível acessar apenas o elemento do topo de uma pilha. O código a seguir mostra como uma pilha pode ser utilizada:

```
stack<int> s;
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

The Polish Notation

```
int evaluateRPN(const string& expression) {
    stack<int> operands;

    istringstream iss(expression);
    string token;

    while (iss >> token) {
        if (isdigit(token[0]) || (token.size() > 1 && isdigit(token[1]))) {
            // Token is an operand (number)
            operands.push(stoi(token));
        } else {
            // Token is an operator
            int operand2 = operands.top();
            operands.pop();
            int operand1 = operands.top();
            operands.pop();

            switch (token[0]) {
                case '+':
                    operands.push(operand1 + operand2);
                    break;
                case '-':
                    operands.push(operand1 - operand2);
```

```
                    break;
                case '*':
                    operands.push(operand1 * operand2);
                    break;
                case '/':
                    operands.push(operand1 / operand2);
                    break;
                default:
                    cerr << "Invalid operator: " << token << endl;
                    return -1;
            }
        }
    }

    return operands.top();
}
```

Balanced Brackets

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

using namespace std;

int main(){
    int n, i;
    string s;
    cin >> n;
    while(n--){
        cin >> s;
        stack<char> p;
        for(i = 0; i < s.size(); i++)
        {
            if ((s[i] == '(') || (s[i] == '[') || (s[i] == '{'))
                p.push(s[i]);
            else if (p.empty())
                break;
            else if ((s[i] == ')' && p.top() == '(') ||
                     (s[i] == ']' && p.top() == '[') ||
                     (s[i] == '}' && p.top() == '{'))
                p.pop();
            else
                break;
        }
        if ((i == s.size()) && p.empty())
            cout << "YES" << endl;
        else
            cout << "NO" << endl;
    }
}
```

Queue

Uma fila também oferece duas operações em tempo O(1): adicionar um elemento ao final da fila e remover o primeiro elemento da fila. É possível acessar apenas o primeiro e o último elemento de uma fila. O código a seguir mostra como uma fila pode ser utilizada.

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Street Parede

```
/*
Problema: Street Parade
Categorias:
    data structure > queue
    data structure > stack
Dificuldade: facil
Descricao:
Dica:
Autor: Paiola
*/

#include<bits/stdc++.h>

using namespace std;

int main()
{
    int n, i, x;
    while(1)
    {
        cin >> n;
        if (n == 0)
            break;

        queue<int> fila;
        stack<int> pilhaMenores, pilhaMaiores;
        for(i = 0; i < n; i++)
        {
            cin >> x;
            fila.push(x);
        }
```

```
        pilhaMenores.push(0);
        while(!fila.empty())
        {
            if (fila.front() == (pilhaMenores.top() + 1))
            {
                pilhaMenores.push(fila.front());
                fila.pop();
            }
            else
            {
                if (pilhaMaiores.empty())
                {
                    pilhaMaiores.push(fila.front());
                    fila.pop();
                }
                else
                {
                    while(!pilhaMaiores.empty() && (pilhaMaiores.top() ==
(pilhaMenores.top() + 1)))
                    {
                        pilhaMenores.push(pilhaMaiores.top());
                        pilhaMaiores.pop();
                    }

                    if (fila.front() == (pilhaMenores.top() + 1))
                    {
                        pilhaMenores.push(fila.front());
                        fila.pop();
                    }
                    else
                    {
                        pilhaMaiores.push(fila.front());
                        fila.pop();
                    }
                }
            }
        }
        while(!pilhaMaiores.empty() && (pilhaMaiores.top() == (pilhaMenores.top()
+ 1)))
        {
            pilhaMenores.push(pilhaMaiores.top());
            pilhaMaiores.pop();
        }
        cout << ((fila.empty() && pilhaMaiores.empty()) ? "yes" : "no") << endl;
    }
}
```

Priority Queue

Uma fila de prioridade mantém um conjunto de elementos. As operações suportadas são a inserção e, dependendo do tipo de fila, a recuperação e a remoção do elemento mínimo ou máximo. A inserção e a

remoção levam O(logn) tempo, e a recuperação leva O(1) tempo. Embora um conjunto ordenado suporte eficientemente todas as operações de uma fila de prioridade, o benefício de usar uma fila de prioridade é que ela tem fatores constantes menores. Uma fila de prioridade geralmente é implementada usando uma estrutura de heap que é muito mais simples do que uma árvore binária balanceada usada em um conjunto ordenado. Por padrão, os elementos em uma fila de prioridade C++ são ordenados em ordem decrescente, e é possível encontrar e remover o maior elemento na fila. O código a seguir ilustra isso:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Se quisermos criar uma fila de prioridade que suporte encontrar e remover o elemento menor, podemos fazer isso da seguinte maneira:

```
priority_queue<int,vector<int>,greater<int>> q;
```

Telemarketing

O problema do telemarketing, envolve a atribuição eficiente de tarefas a um grupo de trabalhadores, onde cada tarefa tem um horário de início e fim, e cada trabalhador está disponível apenas durante um intervalo específico de tempo. O objetivo é maximizar o número de tarefas que são atribuídas sem sobrepor os intervalos de disponibilidade dos trabalhadores. Isso requer uma abordagem que combine tarefas e trabalhadores de forma a otimizar a utilização do tempo e realizar o maior número de tarefas possível.

```
#include <cstdio>
#include <queue>

#define MAXN 1010
#define MAXL 1000100

using namespace std;

int n, l, t, qtd[MAXN];

struct atendente{ // declaro a struct atendente

    int id, livre; // ela terá dois inteiros como membros
```

```
    // terá um construtor para atribuir valores aos membros
    atendente(int id_, int livre_=0){ id=id_; livre=livre_; }

    // terá também um operador >, para compará-la com outro atendente

    // note que a priority_queue ordena do maior para o menor
    // logo este operador deve retornar true se a struct vem antes na ordenação
    bool operator >(const atendente x) const{

        // se eles ficarem livres em momentos diferentes
        if(livre!=x.livre) return livre<x.livre; // ele retorna o que fica livre
antes
        return id<x.id; // caso contrário, retorna o de menor id
    }

    // crio também o operador < de maneira análoga ao >
    bool operator <(const atendente x) const{

        if(livre!=x.livre) return livre>x.livre;
        return id>x.id;
    }
};

priority_queue<atendente> heap; // crio a priority_queue de atendente de nome
"heap"

int main(){

    scanf("%d %d", &n, &l); // leio os valores de n e l

    for(int i=1; i<=n; i++) heap.push(atendente(i)); // insiro os atendentes na
fila de prioridade

    int tempo=0; // inicializo tempo com o valor 0

    for(int i=1; i<=l; i++){ // para cada ligação

        scanf("%d", &t); // leio sua duração

        atendente davez=heap.top(); // e vejo qual o atendente que irá atendê-la
(o primeiro de heap)

        heap.pop(); // tiro ele do topo
        qtd[davez.id]++; // e aumento a quantidade de ligações por ele atendidas

        if(davez.livre>tempo) tempo=davez.livre; // se a ligação teve que esperar,
atualizo o tempo

        davez.livre+=t; // salvo que o atendente davez só ficará livre novamente
em t minutos
        heap.push(davez); // e o reinsiro na fila de prioridade
    }
```

```
    // depois, para cada um dos atendentes, imprimo sua identificação e a
quantidade de ligações atendidas
    for(int i=1; i<=n; i++) printf("%d %d\n", i, qtd[i]);

    return 0;
}
```

Set

Um conjunto é uma estrutura de dados que mantém uma coleção de elementos. As operações básicas dos conjuntos são inserção de elementos, busca e remoção. A biblioteca padrão de C++ contém duas implementações de conjuntos: A estrutura `set` é baseada em uma árvore binária balanceada e suas operações funcionam em tempo O(logn). A estrutura `unordered_set` usa hashing, e suas operações funcionam em tempo O(1) em média. A escolha de qual implementação de conjunto usar frequentemente é uma questão de preferência. A vantagem da estrutura `set` é que ela mantém a ordem dos elementos e fornece funções que não estão disponíveis no `unordered_set`. Por outro lado, o `unordered_set` pode ser mais eficiente. O código a seguir cria um conjunto que contém inteiros e demonstra algumas das operações. A função `insert` adiciona um elemento ao conjunto, a função `count` retorna o número de ocorrências de um elemento no conjunto e a função `erase` remove um elemento do conjunto.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

Um conjunto pode ser usado principalmente como um vetor, mas não é possível acessar os elementos usando a notação `[]`. O código a seguir cria um conjunto, imprime o número de elementos nele e, em seguida, itera por todos os elementos:

```
set<int> s = {2, 5, 6, 8};
cout << s.size() << "\\n"; // 4
for (auto x : s) {
    cout << x << "\\n";
}
```

Uma propriedade importante dos conjuntos é que todos os seus elementos são distintos. Portanto, a função `count` sempre retorna 0 (o elemento não está no conjunto) ou 1 (o elemento está no conjunto), e a função `insert` nunca adiciona um elemento ao conjunto se ele já estiver lá. O código a seguir ilustra isso:

```
set<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\\n"; // 1
```

Nesse exemplo, mesmo que o valor 5 seja inserido três vezes, o conjunto mantém apenas uma instância do valor, pois todos os elementos em um conjunto são únicos. A função `count` retorna 1 para o valor 5, indicando que ele está presente no conjunto.

Multi Set

C++ também contém as estruturas `multiset` e `unordered_multiset` que funcionam de forma semelhante a `set` e `unordered_set`, mas podem conter várias instâncias de um elemento. Por exemplo, no código a seguir, todas as três instâncias do número 5 são adicionadas a um `multiset`:

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3
```

A função `erase` remove todas as instâncias de um elemento de um `multiset`:

```
s.erase(5);
cout << s.count(5) << "\\n"; // 0
```

Frequentemente, apenas uma instância deve ser removida, o que pode ser feito da seguinte maneira:

```
s.erase(s.find(5));
cout << s.count(5) << "\\n"; // 2
```

Nesses exemplos, a função `erase` remove todas as instâncias do elemento 5 do `multiset`. No entanto, se você quiser remover apenas uma instância, pode usar a função `find` para localizar a primeira ocorrência do elemento e, em seguida, passar o resultado para a função `erase`. Isso resultará em duas instâncias restantes do elemento 5 no `multiset`.

Iterator

Iteradores são frequentemente usados para acessar elementos de um conjunto. O código a seguir cria um iterador `it` que aponta para o menor elemento em um conjunto:

```
set<int>::iterator it = s.begin();
```

Uma maneira mais curta de escrever o código é a seguinte:

```
auto it = s.begin();
```

O elemento para o qual um iterador aponta pode ser acessado usando o símbolo `*`. Por exemplo, o código a seguir imprime o primeiro elemento no conjunto:

```
auto it = s.begin();
cout << *it << "\\n";
```

Os iteradores podem ser movidos usando os operadores ++ (avanço) e -- (retrocesso), o que significa que o iterador avança para o próximo ou anterior elemento no conjunto. O código a seguir imprime todos os elementos em ordem crescente:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\\n";
}
```

O seguinte código imprime o maior elemento no conjunto:

```
auto it = s.end();
it--;
cout << *it << "\\n";
```

A função `find(x)` retorna um iterador que aponta para um elemento cujo valor é x. No entanto, se o conjunto não contiver x, o iterador será `end`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x não foi encontrado
```

```
}
```

A função `lower_bound(x)` retorna um iterador para o menor elemento no conjunto cujo valor é pelo menos x, e a função `upper_bound(x)` retorna um iterador para o menor elemento no conjunto cujo valor é maior que x. Em ambas as funções, se tal elemento não existir, o valor de retorno é `end`. Essas funções não são suportadas pela estrutura `unordered_set`, que não mantém a ordem dos elementos.

Por exemplo, o seguinte código encontra o elemento mais próximo de x:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\\n";
} else {
    int a = *it;
    it--;
    int b = *it;
    if (x - b < a - x) cout << b << "\\n";
    else cout << a << "\\n";
}
```

O código pressupõe que o conjunto não está vazio e passa por todos os casos possíveis usando um iterador `it`. Primeiro, o iterador aponta para o menor elemento cujo valor é pelo menos x. Se ele for igual a `begin`, o elemento correspondente está mais próximo de x. Se for igual a `end`, o maior elemento no conjunto está mais próximo de x. Se nenhum dos casos anteriores for verdadeiro, o elemento mais próximo de x é o elemento que corresponde a ele ou o elemento anterior.

Funções úteis do C++

GCD (Greatest common divisor):

Maior divisor comum

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// OR

__gcd(a, b)
```

LCM (Least Common Multiple):

MMC, menor múltiplo comum

```
// Recursive function to return gcd of a and b
long long gcd(long long int a, long long int b)
{
  if (b == 0)
    return a;
  return gcd(b, a % b);
}

// Function to return LCM of two numbers
long long lcm(int a, int b)
{
    return (a / gcd(a, b)) * b;
}
```

Conversão de tipos

1. stoi: **string** to **int**
2. stol: **string** to **long**
3. stoll: **string** to **long long**
4. stod: **string** to **double**
5. to_string: **number** to **string**

Produto dos i-th fatoriais

```
// To compute (a * b) % MOD
long long int mulmod(long long int a, long long int b,
                                    long long int mod)
```

```
{
    long long int res = 0; // Initialize result
    a = a % mod;
    while (b > 0) {

        // If b is odd, add 'a' to result
        if (b % 2 == 1)
            res = (res + a) % mod;

        // Multiply 'a' with 2
        a = (a * 2) % mod;

        // Divide b by 2
        b /= 2;
    }

    // Return result
    return res % mod;
}

// This function computes factorials and
// product by using above function i.e.
// modular multiplication
long long int findProduct(long long int N)
{
    // Initialize product and fact with 1
    long long int product = 1, fact = 1;
    long long int MOD = 1e9 + 7;
    for (int i = 1; i <= N; i++) {

        // ith factorial
        fact = mulmod(fact, i, MOD);

        // product of first i factorials
        product = mulmod(product, fact, MOD);

        // If at any iteration, product becomes
        // divisible by MOD, simply return 0;
        if (product == 0)
            return 0;
    }
    return product;
}

N = 5;
cout << findProduct(N) << endl; // 34560
```

Josephus

There are N people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped

and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom.

Given the total number of persons N and a number k which indicates that k-1 persons are skipped and the kth person is killed in a circle. The task is to choose the person in the initial circle that survives.

```
int Josephus(int N, int k)
{

    // Initialize variables i and ans with 1 and 0
    // respectively.

    int i = 1, ans = 0;

    // Run a while loop till i <= N

    while (i <= N) {

        // Update the Value of ans and Increment i by 1
        ans = (ans + k) % i;
        i++;
    }

    // Return required answer
    return ans + 1;
}

int N = 14, k = 2;
cout << Josephus(N, k) << endl; // 14
```

Por recursão:

```
// Recursive function to implement the Josephus problem
int josephus(int n, int k)
{
    if (n == 1)
        return 1;
    else
        // The position returned by josephus(n - 1, k)
        // is adjusted because the recursive call
        // josephus(n - 1, k) considers the
        // original position k % n + 1 as position 1
        return (josephus(n - 1, k) + k - 1) % n + 1;
}
```

Time Complexity: O(N)

Números Primos

Verificar se N é primo

Time complexity: O(sqrt(N))

```
bool is_prime(int n) {
    // Assumes that n is a positive natural number
    // We know 1 is not a prime number
    if (n == 1) {
        return false;
    }

    int i = 2;
    // This will loop from 2 to int(sqrt(x))
    while (i*i <= n) {
        // Check if i divides x without leaving a remainder
        if (n % i == 0) {
            // This means that n has a factor in between 2 and sqrt(n)
            // So it is not a prime number
            return false;
        }
        i += 1;
    }
    // If we did not find any factor in the above loop,
    // then n is a prime number
    return true;
}
```

Sieve of Eratosthenes

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

```
#include <bitset>
#include <iostream>
using namespace std;
bitset<500001> Primes;
void SieveOfEratosthenes(int n)
{
    Primes[0] = 1;
    for (int i = 3; i*i <= n; i += 2) {
        if (Primes[i / 2] == 0) {
            for (int j = 3 * i; j <= n; j += 2 * i)
                Primes[j / 2] = 1;
        }
    }
}
int main()
{
    int n = 100;
    SieveOfEratosthenes(n);
    for (int i = 1; i <= n; i++) {
```

```
        if (i == 2)
            cout << i << ' ';
        else if (i % 2 == 1 && Primes[i / 2] == 0)
            cout << i << ' ';
    }
    // 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
    return 0;
}
```

Binomial Coefficient

A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (or k-combinations) of a n-element set.

O coeficiente binomial, também chamado de número binomial, de um número n, na classe k, consiste no número de combinações de n termos, k a k.

```
int binomialCoeff(int n, int r)
{

    if (r > n)
        return 0;
    long long int m = 1000000007;
    long long int inv[r + 1] = { 0 };
    inv[0] = 1;
    if(r+1>=2)
    inv[1] = 1;

    // Getting the modular inversion
    // for all the numbers
    // from 2 to r with respect to m
    // here m = 1000000007
    for (int i = 2; i <= r; i++) {
        inv[i] = m - (m / i) * inv[m % i] % m;
    }

    int ans = 1;

    // for 1/(r!) part
    for (int i = 2; i <= r; i++) {
        ans = ((ans % m) * (inv[i] % m)) % m;
    }

    // for (n)*(n-1)*(n-2)*...*(n-r+1) part
    for (int i = n; i >= (n - r + 1); i--) {
        ans = ((ans % m) * (i % m)) % m;
    }
    return ans;
}
int n = 5, r = 2;
```

```
cout << "Value of C(" << n << ", " << r << ") is "
<< binomialCoeff(n, r) << endl; // Value of C(5, 2) is 10
```

Conversão de bases numéricas

Qualquer base -> decimal

```
string base2 = "1100";
string base8 = "21";
string base10 = "25";
string base16 = "1E";

cout << (stoi(base2, nullptr, 2)) << endl; // 12
cout << (stoi(base8, nullptr, 8)) << endl; // 17
cout << (stoi(base10, nullptr, 10)) << endl; // 25
cout << (stoi(base16, nullptr, 16)) << endl; // 30
```

Decimal -> Qualquer base

```
// To return char for a value. For example '2'
// is returned for 2. 'A' is returned for 10. 'B'
// for 11
char reVal(int num)
{
    if (num >= 0 && num <= 9)
        return (char)(num + '0');
    else
        return (char)(num - 10 + 'A');
}

// Function to convert a given decimal number
// to a base 'base' and
string fromDeci(string& res, int base, int inputNum)
{
    int index = 0; // Initialize index of result

    // Convert input number is given base by repeatedly
    // dividing it by base and taking remainder
    while (inputNum > 0) {
        res.push_back(reVal(inputNum % base));
        index++;
        inputNum /= base;
    }

    // Reverse the result
    reverse(res.begin(), res.end());
```

```
    return res;
}
int inputNum = 282, base = 16; string res;
cout << "Equivalent of " << inputNum << " in base "
    << base << " is " << fromDeci(res, base, inputNum)
    << endl; //Equivalent of 282 in base 16 is 11A
```

Partição de um número

Given a positive integer n, generate all possible unique ways to represent n as sum of positive integers.

Time Complexity: O(2^n)

```
class Solution {
public:
    vector<int> temp;
    void solve(vector<int> a, vector<vector<int> >& v,
               int idx, int sum, int n)
    {
        // first base case if sum=n we can store vector in a
        // vector
        if (sum == n) {
            v.push_back(temp);
            return;
        }
        // if idx < 0 return
        if (idx < 0) {
            return;
        }
        // not take condition
        solve(a, v, idx - 1, sum, n);
        if (sum < n) {
            temp.push_back(a[idx]);
            // this is main condition where we can take one
            // element many times
            solve(a, v, idx, sum + a[idx], n);
            temp.pop_back();
        }
    }
    vector<vector<int> > UniquePartitions(int n)
    {
        vector<int> a;
        // vector to store elements from 1 to n
        for (int i = 1; i <= n; i++) {
            a.push_back(i);
        }
        vector<vector<int> > v;
        // call solve to get answer
        solve(a, v, n - 1, 0, n);
        reverse(v.begin(), v.end());
        return v;
```

```
    }
};
// using
vector<vector<int> > ans = ob.UniquePartitions(4);
cout << "for 4\n";
for (auto i : ans) {
    for (auto j : i) {
        cout << j << " ";
    }
    cout << "\n";
}
```

1.5 Dicas Sujas

- **Método Steve Halim**: As possíveis saídas do problema cabem no código do problema? Deixe um algoritmo *naive* brutando o problema na máquina por alguns minutos e escreva as respostas direto no código para submeter. Exemplo: problema cuja entrada é um único número da ordem de 10^5. Verificar o tamanho máximo de caracteres de uma submissão.
- **Fatoriais até 10^9**: Deixe um programa na sua máquina brutando os fatoriais até 10^9. A cada 10^3 ou 10^6, imprima. Cole a saída no código e use os valores pré-calculados pra calcular um fatorial com 10^3 ou 10^6 operações.
- **Problemas com constantes**: Se algum valor útil de algum problema for constante (independe da entrada), mas você não sabe, *brute* ele na sua máquina e cole no código.
- **Debug com *assert***: Pode colocar *assert* em código para submeter. Tente usar isso pra transformar um WA em um RTE. É uma forma válida de debug. Usar isso somente no desespero (fica gastando submissões)

2 Força Bruta e Backtracking

Tentativa e erro: decompor o processo em um número finito de subtarefas parciais que devem ser exploradas exaustivamente.

- O processo de tentativa gradualmente constrói e percorre uma árvore de subtarefas.
- Algoritmos tentativa e erro não seguem uma regra fixa de computação:
  - Passos em direção à solução final são tentados e registrados.
  - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.

2.1 Labirinto

Dado um labirinto 5x5 definir se é possível chegar até o final do labirinto começando da posição 0,0:

```
int T;
int maze[5][5];
bool vis[5][5];

map<char, pii> movimento = {
  {'D', {-1, 0}},
  {'E', {1, 0}},
  {'B', {0, 1}},
  {'C', {0, -1}},
};
vector<char> movimentos_possiveis = {'B', 'C', 'D', 'E'};
bool ganhou = false;

bool deslocamento_possivel(int x, int y, char caminho) {
  x += movimento[caminho].f;
  y += movimento[caminho].s;
```

```
  if(x >= 0 && x < 5 && y >= 0 && y < 5 && maze[x][y] != 1 && vis[x][y] == 0)
return true;
  return false;
}

void backtracking(int x, int y) {
  if(ganhou) return;

  vis[x][y] = true;
  if(x == 4 && y == 4) {
    ganhou = true;
    return;
  }

  for(auto c : movimentos_possiveis) {
    if(!deslocamento_possivel(x, y, c)) continue;
    backtracking(x + movimento[c].f, y + movimento[c].s);
  }
}

// ...

backtracking(0, 0);
```

2.2 Cavalo

Dado um tabuleiro de xadrez $n \times n$ e uma posição (x, y) do tabuleiro, queremos encontrar um passeio de um cavalo que visite cada casa exatamente uma vez.

- Movimento do cavalo – formato de L:
  - dois quadrados horizontalmente e um verticalmente, ou
  - dois quadrados verticalmente e um horizontalmente.

```
int m[MAX][MAX], n;
vector<pii> movimentos = {
    {2, -1}, {2, 1}, {-2, 1}, {-2, -1},
    {1, 2}, {-1, 2}, {-1, -2}, {1, -2}
};

bool posicaoValida(int x, int y){
  return (x >= 0) && (x < n) && (y >= 0) && (y < n) && !m[x][y];
}

int passeioCavalo(int x, int y)
{
  if (m[x][y] == n * n)
    return 1;
  for (auto mov : movimentos)
  {
    int x2 = x + mov.first;
```

```
      int y2 = y + mov.second;
      if (posicaoValida(x2, y2))
      {
        m[x2][y2] = m[x][y] + 1;
        if (passeioCavalo(x2, y2))
          return 1;
        m[x2][y2] = 0;
      }
    }
    return 0;
}
```

2.3 O Problema das N Rainhas

A Rainha N é o problema de colocar N rainhas de xadrez em um tabuleiro de xadrez N×N de modo que duas rainhas não se ataquem.

```
#include <bits/stdc++.h>
using namespace std;
#define N 4

// ld is an array where its indices indicate row-col+N-1
// (N-1) is for shifting the difference to store negative
// indices
int ld[30] = { 0 };

// rd is an array where its indices indicate row+col
// and used to check whether a queen can be placed on
// right diagonal or not
int rd[30] = { 0 };

// Column array where its indices indicates column and
// used to check whether a queen can be placed in that
// row or not*/
int cl[30] = { 0 };

// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout << " " << board[i][j] << " ";
        cout << endl;
    }
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
```

```
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]

        // To check if a queen can be placed on
        // board[row][col].We just need to check
        // ld[row-col+n-1] and rd[row+coln] where
        // ld and rd are for left and right
        // diagonal respectively
        if ((ld[i - col + N - 1] != 1 && rd[i + col] != 1)
            && cl[i] != 1) {

            // Place this queen in board[i][col]
            board[i][col] = 1;
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
            ld[i - col + N - 1] = rd[i + col] = cl[i] = 0;
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}

// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one  of the
// feasible solutions.
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
```

```
                          { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }

    printSolution(board);
    return true;
}

// Driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

3. Busca Binária

Para aplicar um algoritmo de busca binária preciso de:

- Uma estrutura de dados ordenada.
- Acessar qualquer elemento dessa estrutura com a complexidade contante.

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

// An iterative binary search function.
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}
```

3.1 Funções

- **binary_search(first, last, val)**
  - Retorna um booleano indicando se existe o elemento
- **lower_bound(first, last, val)**
  - Retorna iterator para o **primeiro** valor **não-inferior** a **val**
  - Ou retorna last, caso não encontre **val**
- **upper_bound(first, last, val)**
  - Retorna iterator para o **primeiro** valor **superior** a **val**
  - Ou retorna last, caso não encontre **val**

3.2 Método da bissetriz

Exemplo de cálculo de raiz quadrada:

```
double raiz(double x, double eps = 1e-3)
{
  double l = 0, r = x;
  while (r - l > eps) {
    double m = (l + r) / 2.0;
    cout << m << endl;

    if (m * m < x)
      l = m;
    else
      r = m;
  }

  return (l + r) / 2.0;
}
```

> sqrt(2) = 1.41421 1 1.5 1.25 1.375 1.4375 1.40625 1.42188 1.41406 1.41797 1.41602 1.41504 1.41455
> raiz(2) = 1.41455

```
ll n;
ll k;
vector<ld> acumuladores(1e4+1);

bool eh_possivel(ld max_energia) {
  ld doadores = 0.0, receptores = 0.0;
  for(ll i = 0; i < n; i++) {
    if(acumuladores[i] > max_energia) doadores += abs(acumuladores[i] -
max_energia);

    if(acumuladores[i] < max_energia) receptores += abs(max_energia -
acumuladores[i]);
  }

  doadores -= (doadores * k) / 100.0;
  return doadores >= receptores;
}

int main()
{
  speed;
  cin >> n >> k;
  for(ll i = 0; i < n; i++) cin >> acumuladores[i];

  ld l = 0.0, r = 1e12, eps = 1e-6, ans;
  while((r - l) > eps) {
    ld mid = (l + r) / 2.0;

    if(eh_possivel(mid)) {
      ans = mid;
      l = mid;
```

```
    } else r = mid;
  }

  cout << fixed << setprecision(9) << (l + r) / 2.0 << endl;

  return 0;
}
```

```
ll N, A;
vll tiras(1e5+1);

ld area_corte(ld altura) {
  ld total = 0.0;

  for(ll i = 0; i < N; i++) {
    if(tiras[i] <= altura) continue;

    total += ((ld)tiras[i] - altura);
  }

  return total;
}

int main()
{
  speed;
  while(cin >> N >> A) {
    if(N == 0 && A == 0) break;

    for(ll i = 0; i < N; i++) cin >> tiras[i];

    ll max_area = 0;
    for(ll i = 0; i < N; i++) {
      max_area += tiras[i];
    }

    if(max_area < A) {
      cout << "-.-" << endl;
      continue;
    }
    if(max_area == A) {
      cout << ":D" << endl;
      continue;
    }

    ld l = 0, r = (ld)max_area;
    while((r - l) > 1e-6) {
      ld mid = (l + r) / 2.0;

      if(area_corte(mid) > A) {
        l = mid;
```

```
      } else {
        r = mid;
      }
    }

    cout << fixed << setprecision(4) << ((l + r) / 2.0) << endl;
  }



  return 0;
}
```

3.3 Busca binária na resposta

```
vll pipocas;
ll competidores, tempo, qtd;

bool eh_possivel(ll chute) {
  ll competidor_atual = 1, resta = chute * tempo;
  for(ll i = 0; i < sz(pipocas); i++) {
    if(resta >= pipocas[i]) resta -= pipocas[i];
    else {
      competidor_atual++;
      resta = chute * tempo;
      i--;
    }
    if(competidor_atual > competidores) return false;
  }

  return true;
}

int main()
{
  speed;
  cin >> qtd >> competidores >> tempo;
  pipocas.assign(qtd, 0);

  for(ll i = 0; i < qtd; i++) cin >> pipocas[i];

  ll l = 0, r = 1e9+1;
  while(l < r) {
    ll m = (l + r) / 2;

    if(!eh_possivel(m)) l = m + 1;
    else r = m;
  }

  cout << l << endl;
```

```
    return 0;
}
```

4. Guloso

- Aplicado a problemas de **otimização**.
- Independente do que possa acontecer mais tarde, nunca **reconsidera** a decisão.
- Não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões tomadas previamente.

4.1 Dicas

- Quando funciona corretamente, a primeira solução encontrada é sempre ótima.
- A função de seleção é geralmente relacionada com a função objetivo.
- Se o objetivo é:
    - Maximizar ⇒ provavelmente escolherá o candidato restante que proporcione o maior ganho individual.
    - Minimizar ⇒ então será escolhido o candidato restante de menor custo.
- O algoritmo **nunca** muda de ideia:
    - Um candidato escolhido e adicionado à solução passa a fazer parte dessa solução **permanentemente**.
    - Um candidato excluído do conjunto solução, não é mais **reconsiderado**

4.2 Propriedade

Solução global ótima pode ser obtida a partir de escolhas **locais ótimas**.

- Estratégia diferente de programação dinâmica (PD).
- Uma vez feita a escolha, resolve o problema a partir do "estado" em que se encontra.
- Escolha na técnica gulosa depende só do que foi feito e não do que será feito no futuro.
- Progride na forma *top-down*:
    - Através de iterações vai "transformando" uma instância do problema em uma outra menor.
- Estratégia da prova que a escolha gulosa leva a uma **solução global ótima**:
    - Examine a solução global ótima.
    - Mostre que a solução pode ser modificada de tal forma que uma escolha gulosa pode ser aplicada como primeiro passo.
    - Mostre que essa escolha reduz o problema a um similar mas menor.
    - Aplique indução para mostrar que uma escolha gulosa pode ser aplicada a cada passo.

4.3 Ordenação

Existe uma série de problemas gulosos que são baseados em ordenar os elementos, dependendo de como eles forem representados:

- Vetor de números: ordenar de forma crescente/decrescente e, sempre que possível, adicionando os valores na resposta.
- Vetor de palavras: ordenar lexicograficamente
- Vetor de pares: ordenar pela diferença (*second-first*), ordenar pelo primeiro/segundo elemento, ordenar por alguma fórmula $f(x, y)$.
- Aquilo que importa é deixar o seu vetor propício às suas escolhas gulosas funcionarem.

Problema dos Pedidos Compatíveis

Este problema deve escolher o maior número de pedidos com tempo de inicio e final, afim de ter o maior número de pedidos.

```cpp
bool cmp(pair<int, int> a, pair<int, int> b)
{
  return a.second < b.second;
}

int main()
{
  int qtd;
  cin >> qtd;

  vector<pair<int, int>> orders(qtd, {0, 0});

  for (int i = 0; i < qtd; i++)
  {
    pair<int, int> order;
    cin >> order.first;
    cin >> order.second;
    orders[i] = order;
  }

  sort(orders.begin(), orders.end(), cmp);

  int ans = 0;
  int fim = -1;
  for (int i = 0; i < qtd; i++)
  {
    if (orders[i].first > fim)
    {
      fim = orders[i].second;
      ans++;
    }
  }

  cout << ans << endl;

  return 0;
}
```

5. Strings

5.1 KMP

Retorna os índices das ocorrências de ***S*** em ***T.***

```
template<typename T> vector<int> pi(T s) {
    vector<int> p(s.size());
    for (int i = 1, j = 0; i < s.size(); i++) {
        while (j and s[j] != s[i]) j = p[j-1];
        if (s[j] == s[i]) j++;
        p[i] = j;
    }
    return p;
}

template<typename T> vector<int> matching(T& s, T& t) {
    vector<int> p = pi(s), match;
    for (int i = 0, j = 0; i < t.size(); i++) {
        while (j and s[j] != t[i]) j = p[j-1];
        if (s[j] == t[i]) j++;
        if (j == s.size()) match.push_back(i-j+1), j = p[j-1];
    }
    return match;
}

struct KMPaut : vector<vector<int>> {
    KMPaut(){}
    KMPaut (string& s) : vector<vector<int>>(26, vector<int>(s.size()+1)) {
        vector<int> p = pi(s);
        auto& aut = *this;
        aut[s[0]-'a'][0] = 1;
        for (char c = 0; c < 26; c++)
            for (int i = 1; i <= s.size(); i++)
                aut[c][i] = s[i]-'a' == c ? i+1 : aut[c][p[i-1]];
    }
};
```

Complexidades:

> pi - *O(n)* match - *O(n + m)* construir o autômato - O(|sigma|*n) n = |padrão| e m = |texto|

5.2 Palíndromo

Palíndromo é uma sequência de caracteres que ao ser invertida mantém-se idêntica.

```
bool isPalindrome(string S)
{
```

```
    // Iterate over the range [0, N/2]
    for (int i = 0; i < S.length() / 2; i++)
    {

        // If S[i] is not equal to
        // the S[N-i-1]
        if (S[i] != S[S.length() - i - 1])
        {
            return false;
        }

    return true;
}
```

Complexidade:

> *O(n)*

5.3 Algoritmo de Manacher

Determina qual a maior substring palindrômica e também quantas substrings são palíndromos.

```
// manacher recebe um vetor de T e retorna o vetor com tamanho dos palindromos
template<typename T> vector<int> manacher(const T& s) {
    int l = 0, r = -1, n = s.size();
    vector<int> d1(n), d2(n);
    for (int i = 0; i < n; i++) {
        int k = i > r ? 1 : min(d1[l+r-i], r-i);
        while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) k++;
        d1[i] = k--;
        if (i+k > r) l = i-k, r = i+k;
    }
    l = 0, r = -1;
    for (int i = 0; i < n; i++) {
        int k = i > r ? 0 : min(d2[l+r-i+1], r-i+1); k++;
        while (i+k <= n && i-k >= 0 && s[i+k-1] == s[i-k]) k++;
        d2[i] = --k;
        if (i+k-1 > r) l = i-k, r = i+k-1;
    }
    vector<int> ret(2*n-1);
    for (int i = 0; i < n; i++) ret[2*i] = 2*d1[i]-1;
    for (int i = 0; i < n-1; i++) ret[2*i+1] = 2*d2[i+1];
    return ret;
}

// verifica se a string s[i..j] eh palindromo
template<typename T> struct palindrome {
    vector<int> man;

    palindrome(const T& s) : man(manacher(s)) {}
    bool query(int i, int j) {
```

```
        return man[i+j] >= j-i+1;
    }
};

// tamanho do maior palindromo que termina em cada posicao
template<typename T> vector<int> pal_end(const T& s) {
    vector<int> ret(s.size());
    palindrome<T> p(s);
    ret[0] = 1;
    for (int i = 1; i < s.size(); i++) {
        ret[i] = min(ret[i-1]+2, i+1);
        while (!p.query(i-ret[i]+1, i)) ret[i]--;
    }
    return ret;
}
```

Complexidades:

> manacher - *O(n)* palindrome - *<O(n), O(1)>* pal*end* - *_O(n)*

5.4 Trie

Um trie (derivado de recuperação) é uma estrutura de dados em árvore multidirecional usada para armazenar strings em um alfabeto. É usado para armazenar uma grande quantidade de strings. A correspondência de padrões pode ser feita de forma eficiente usando tentativas.

```
struct trie {
    vector<vector<int>> to;
    vector<int> end, pref;
    int sigma; char norm;
    trie(int sigma_=26, char norm_='a') : sigma(sigma_), norm(norm_) {
        to = {vector<int>(sigma)};
        end = {0}, pref = {0};
    }
    void insert(string s) {
        int x = 0;
        for(auto c : s) {
            int &nxt = to[x][c-norm];
            if(!nxt) {
                nxt = to.size();
                to.push_back(vector<int>(sigma));
                end.push_back(0), pref.push_back(0);
            }
            x = nxt, pref[x]++;
        }
        end[x]++;
    }
    void erase(string s) {
        int x = 0;
        for(char c : s) {
            int &nxt = to[x][c-norm];
```

```
            x = nxt, pref[x]--;
            if(!pref[x]) nxt = 0;
        }
        end[x]--;
    }

    // retorna a posicao, 0 se nao achar
    int find(string s) {
        int x = 0;
        for(auto c : s) {
            x = to[x][c-norm];
            if(!x) return 0;
        }
        return x;
    }
    // numero de strings que possuem s como prefixo
    int count_pref(string s) {
        return pref[find(s)];
    }
};

int main()
{
  trie t(26);
  t.insert("opa");
  t.insert("opa2");
  t.insert("opa3s");

  cout << t.count_pref("opa") << endl; // 3

  return 0;
}
```

Complexidades:

> T.insert(s) - O(|s|*Σ) T.erase(s) - O(|s|) T.find(s) - O(|s|) T.count_pref(s) - O(|s|)

5.5 Aho Corasick

O numero de ocorrências de alguma *string* do conjunto como ***substring*** de ***s*** e em tempo linear.

```
namespace aho
{
  int go(int v, char ch);
  const int K = 26; // tamanho do alfabeto
  struct trie
  {
    char me;            // char correspondente ao no atual
    int go[K];          // proximo vertice que eu devo ir estando em um estado (v,
c)
    int down[K];        // proximo vertice da trie
```

```
    int is_leaf = 0;    // se o vertice atual da trie eh uma folha (fim de uma ou
mais strings)
    int parent = -1;    // no ancestral do no atual
    int link = -1;      // link de sufixo do no atual (outro no com o maior
matching de sufixo)
    int exit_link = -1; // folha mais proxima que pode ser alcancada a partir de v
usando links de sufixo
    trie(int p = -1, char ch = '$') : parent(p), me(ch)
    {
      fill(begin(go), end(go), -1);
      fill(begin(down), end(down), -1);
    }
  };
  vector<trie> ac;
  void init() // criar a raiz da trie
  {
    ac.resize(1);
  }
  void add_string(string s) // adicionar string na trie
  {
    int v = 0;
    for (auto const &ch : s)
    {
      int c = ch - 'a';
      if (ac[v].down[c] == -1)
      {
        ac[v].down[c] = ac.size();
        ac.emplace_back(v, ch);
      }
      v = ac[v].down[c];
    }
    ac[v].is_leaf++;
  }
  int get_link(int v) // pegar o suffix link saindo de v
  {
    if (ac[v].link == -1)
      ac[v].link = (!v || !ac[v].parent) ? 0 : go(get_link(ac[v].parent),
ac[v].me);
    return ac[v].link;
  }
  int go(int v, char ch) // proximo estado saindo do estado(v, ch)
  {
    int c = ch - 'a';
    if (ac[v].go[c] == -1)
    {
      if (ac[v].down[c] != -1)
        ac[v].go[c] = ac[v].down[c];
      else
        ac[v].go[c] = (!v) ? 0 : go(get_link(v), ch);
    }
    return ac[v].go[c];
  }
  int get_exit_link(int v) // suffix link mais proximo de v que seja uma folha
  {
```

```
    if (ac[v].exit_link == -1)
    {
      int curr = get_link(v);
      if (!v || !curr)
        ac[v].exit_link = 0;
      else if (ac[curr].is_leaf)
        ac[v].exit_link = curr;
      else
        ac[v].exit_link = get_exit_link(curr);
    }
    return ac[v].exit_link;
  }
  int query(string s) // query O(n + ans)
  {
    int ans = 0, curr = 0, at;
    for (auto const &i : s)
    {
      curr = go(curr, i);
      ans += ac[curr].is_leaf;
      at = get_exit_link(curr);
      while (at)
      {
        ans += ac[at].is_leaf;
        at = get_exit_link(at);
      }
    }
    return ans;
  }
}
```

Utilização:

```
int main()
{
  int n, q;
  cin >> n >> q;
  aho::init();
  for (int i = 0; i < n; i++)
  {
    string s;
    cin >> s;
    aho::add_string(s);
  }
  while (q--)
  {
    string t;
    cin >> t;
    cout << aho::query(t) << endl;
  }
  return 0;
}
```

Matemática

Formulas matemáticas, volume, área, perímetro e etc.

Formulas Gerais

Progressão Aritmética

Fórmula do Termo Geral: an = a1 + (n − 1) × r

Soma dos termos da PA: Sn = n × (a1 + an) / 2

Progressão Geométrica

Fórmula do Termo Geral: an = a1 × q^(n−1)

Soma dos termos da PG: Sn = a1 × (q^n - 1) / (q - 1)

Número de áreas em um plano divididas por retas e suas intersecções

Fórmula: A = N + I + 1

Onde N é o número de retas e I é o número de intersecções. Cada reta horizontal tem uma intersecção com uma reta vertical, então sempre existem pelo menos v × h intersecções, onde v é o número de retas verticais e h horizontais.

Números Triangulares

Um número triangular é um número natural representado na forma de um triângulo equilátero. O n-ésimo número triangular pode ser visto como o número de pontos de uma forma triangular com lado formado por n pontos, o que equivale à soma dos primeiros n números naturais.

Em geral, o n-ésimo número triangular é dado por: Tn = Σ[k=1 to n] k = 1 + 2 + 3 + ... + (n − 2) + (n − 1) + n = n(n + 1) / 2

A soma dos primeiros n números triangulares é o n-ésimo número tetraédrico, que tem a fórmula: n(n + 1)(n + 2) / 6

Raízes triangulares e teste de identificação (número de linhas triangulares que podem ser formadas com n elementos): n = √(8x + 1 − 1) / 2

Múltiplos positivos de k num intervalo

O número de múltiplos positivos m(k) de k no intervalo [1,N] é igual a m(k) = N / k.

Número par ou ímpar de divisores

Números que são quadrados perfeitos têm um número ímpar de divisores, enquanto os outros têm um número par.

Número de quadrados perfeitos de A a B

N = floor(sqrt(B)) - ceil(sqrt(A)) + 1

Quadrados e retângulos em um Grid de N lados com K dimensões

Quadrados: NK + (N − 1)K + (N − 2)K até 1

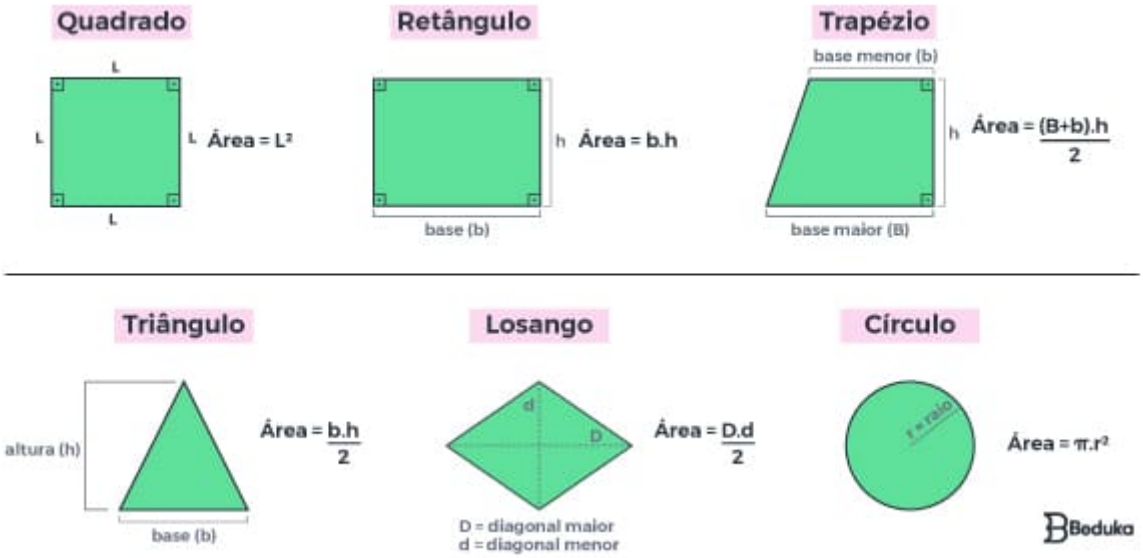
Retângulos: (NK(N+1)K) / 2 - Quadrados*

Número de pares que podem ser formados combinando N elementos

P = (n × (n − 1)) / 2

Geometria 2D

Formulas matemáticas de figuras em 2D.



- **Retângulo**:

```
A (retângulo) = b . h
```

- **Quadrado:**

```
A (quadrado) = l²
```

- **Trapézios:** podem ser **divididos em triângulos e retângulos**, então basta guardar essas duas fórmulas, calcular e somar. **Porém, existe uma fórmula própria** dos trapézios que envolve a **base maior (B**) e **base menor (b)**:

```
A (trapézio) = (B + b) h / 2
```

- **Losango:** também **pode ser dividido em triângulos**, então basta calcular eles e somar. **Porém, existe uma fórmula própria** para losangos com base em sua **diagonal maior (D**) e **diagonal menor (d)**:

```
A = D . d / 2
```

- **Triângulos:** também é dada pela multiplicação de área por altura, mas o valor é dividido na metade **porque o triângulo vai "afunilando"**:

```
A (triângulo) = b . h / 2** ou **A (t. equilátero) = √3 . l² / 4)
```

- **Circunferência**:

```
A (círculo) = π.r²
```

Figuras

Quadrado

Área de um Quadrado (A): A área de um quadrado pode ser calculada multiplicando o comprimento de um dos lados pelo próprio lado:

```
A = Lado × Lado = Lado²
```

Perímetro de um Quadrado (P): O perímetro de um quadrado é a soma dos comprimentos dos quatro lados:

```
P = 4 × Lado
```

Comprimento da Diagonal (d): A diagonal de um quadrado divide o quadrado em dois triângulos retângulos congruentes. O comprimento da diagonal pode ser calculado usando o teorema de Pitágoras, onde "Lado" é o comprimento dos lados do quadrado:

```
d = Lado × √2
```

Raio da Circunferência Inscrita (r): A circunferência inscrita é uma circunferência que toca os quatro lados do quadrado. O raio dessa circunferência pode ser calculado como metade do lado do quadrado:

```
r = Lado / 2
```

Raio da Circunferência Circunscrita (R): A circunferência circunscrita é uma circunferência que passa pelos quatro vértices do quadrado. O raio dessa circunferência é igual à metade da diagonal do quadrado:

```
R = (Lado × √2) / 2
```

Área do Quadrado em Função da Diagonal (d): A área do quadrado também pode ser expressa em termos do comprimento da diagonal:

```
A = (d²) / 2
```

Triângulo

Claro! Aqui estão as fórmulas relacionadas ao triângulo formatadas em Markdown para você copiar e colar:

Área de um Triângulo (A) usando a base e a altura:

```
A = (Base × Altura) / 2
```

Área de um Triângulo (A) usando os lados (Fórmula de Heron): Onde "s" é o semiperímetro do triângulo.

```
Perímetro (s) = (a + b + c) / 2
A = √(s × (s - a) × (s - b) × (s - c))
```

Teorema de Pitágoras para Triângulos Retângulos:

```
a² + b² = c²
```

Lei dos Senos:

```
a / sen(A) = b / sen(B) = c / sen(C)
```

Lei dos Cossenos:

```
c² = a² + b² - 2ab × cos(C)
```

Altura de um Triângulo:

```
Altura = (2 × Área) / Base
```

Mediana de um Triângulo: A mediana de um triângulo é o segmento que liga um vértice ao ponto médio do lado oposto. As medianas de um triângulo se encontram em um ponto chamado centroide.

Círculo

Claro! Aqui estão algumas fórmulas relacionadas ao círculo formatadas em Markdown para você copiar e colar:

Circunferência de um Círculo (C):

```
C = 2πr
```

Área de um Círculo (A):

```
A = πr²
```

Diâmetro de um Círculo (d):

```
d = 2r
```

Relação entre o Diâmetro e a Circunferência:

```
C = πd
```

Comprimento da Circunferência de um Setor Circular: Se o ângulo central do setor circular é θ (em radianos) e o raio é r:

```
Comprimento = θr
```

Área de um Setor Circular: Se o ângulo central do setor circular é θ (em radianos) e o raio é r:

```
Área = (θ/2) × r²
```

Comprimento do Arco de um Círculo: Se o ângulo central do arco é θ (em radianos) e o raio é r:

```
Comprimento do Arco = θr
```

Fórmula da Área do Círculo em Função do Diâmetro:

```
A = (π/4) × d²
```

Relação entre a Área do Círculo e o Comprimento da Circunferência:

```
A = (C²) / (4π)
```

Comprimento da Corda de um Círculo: Se o ângulo central do setor circular é θ (em radianos) e o raio é r, e a corda é igual ao raio, a fórmula para o comprimento da corda é:

```
Comprimento da Corda = 2r × sen(θ/2)
```

Essas são algumas das fórmulas matemáticas básicas relacionadas ao círculo, cada uma descrevendo diferentes propriedades e relações geométricas do círculo.

Inscrito e circunscrito

Círculo Circunscrito: Um círculo circunscrito é aquele que passa por todos os vértices de uma figura geométrica, geralmente um polígono. No caso de triângulos, por exemplo, um círculo circunscrito passa pelos três vértices do triângulo, tocando cada vértice. A posição do centro do círculo circunscrito é tal que os raios a partir do centro até os vértices do polígono têm o mesmo comprimento, que é o raio da circunferência.

Círculo Inscrito: Um círculo inscrito é aquele que está inteiramente contido dentro de uma figura geométrica, geralmente um polígono. No caso de triângulos, um círculo inscrito está inscrito no interior do triângulo, tangenciando os lados do triângulo em pontos específicos. A posição do centro do círculo inscrito é tal que as linhas que ligam o centro aos pontos de tangência nos lados do polígono são perpendiculares aos lados.

Fórmulas

Triângulo:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

  ```
  R = (a * b * c) / (4 * Área)
  ```

- Verificação: Se $a^2 + b^2 = c^2$, o triângulo é retângulo e está circunscrito a uma circunferência.

Círculo Inscrito:

- Raio r do círculo inscrito:

  ```
  r = Área / s
  ```

- Verificação: Se a + b > c, a + c > b e b + c > a, o triângulo tem uma circunferência inscrita.

Quadrado:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

```
R = lado / 2
```

Círculo Inscrito:

- Raio r do círculo inscrito:

```
r = lado / 2
```

Hexágono Regular:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

```
R = lado
```

Círculo Inscrito:

- Raio r do círculo inscrito:

```
r = (lado * √3) / 2
```

Pentágono Regular:

Círculo Circunscrito:

- Raio R do círculo circunscrito:

```
R = (lado / 2) * √(5 + 2√5)
```

Círculo Inscrito:

- Raio r do círculo inscrito:

```
r = (lado / 4) * √(5 - 2√5)
```

Lembrando que nas fórmulas acima, 'lado' representa o comprimento do lado do polígono, 'Área' é a área do polígono e 's' é o semiperímetro do triângulo. Além disso, as verificações mencionadas são critérios para a existência de círculos inscritos ou circunscritos com base nas propriedades dos polígonos.

Geometria 3D

Cubo

Área da superfície do cubo: A área total da superfície de um cubo é dada por:

```
Área = 6 * (lado)^2
```

Volume do cubo: O volume de um cubo é calculado através da fórmula:

```
Volume = (lado)^3
```

Diagonal do cubo: A diagonal de um cubo pode ser encontrada usando o teorema de Pitágoras em três dimensões:

```
Diagonal = √(3) * lado
```

Cilindro

Área da superfície do cilindro: A área total da superfície de um cilindro é a soma da área lateral e das áreas das bases:

```
Área = 2 * π * raio * altura + 2 * π * (raio)^2
```

Volume do cilindro: O volume de um cilindro é dado por:

```
Volume = π * (raio)^2 * altura
```

Diagonal do cilindro: A diagonal de um cilindro retangular pode ser calculada usando o teorema de Pitágoras:

```
Diagonal = √(altura^2 + (2 * raio)^2)
```

Prisma

Área da superfície do prisma: A área total da superfície de um prisma é a soma da área lateral e das áreas das bases:

```
Área = 2 * (área da base) + (perímetro da base) * altura
```

Volume do prisma: O volume de um prisma é dado por:

```
Volume = (área da base) * altura
```

Diagonal do prisma: A diagonal de um prisma retangular pode ser calculada usando o teorema de Pitágoras:

```
Diagonal = √(altura^2 + diagonal da base^2)
```

Pirâmide

Área da superfície da pirâmide: A área total da superfície de uma pirâmide é a soma da área da base e da área lateral:

```
Área = (área da base) + (1/2) * (perímetro da base) * apótema + (área lateral)
```

Volume da pirâmide: O volume de uma pirâmide é dado por:

```
Volume = (1/3) * (área da base) * altura
```

Relação entre altura da pirâmide e altura da pirâmide truncada: Se uma pirâmide é truncada paralelamente à base para formar outra pirâmide, a relação entre as alturas é proporcional à relação das áreas das bases:

```
Altura_truncada = (área da base truncada) / (área da base original) *
Altura_original
```

A "apótema" de uma figura geométrica é a distância entre o centro da figura e o ponto médio de um dos lados. Em muitos casos, é usada para representar a distância do centro até o ponto médio de um lado de um polígono regular, como um triângulo, quadrado, pentágono, hexágono, etc.

Cone

Área da superfície do cone: A área total da superfície de um cone é a soma da área lateral e da área da base:

```
Área = π * raio * geratriz + π * (raio)^2
```

onde a "geratriz" é o comprimento da linha reta que liga o vértice do cone até um ponto qualquer na circunferência da base.

Volume do cone: O volume de um cone é dado por:

```
Volume = (1/3) * π * (raio)^2 * altura
```

Relação entre cones semelhantes: Se você tem dois cones com alturas proporcionais, a razão dos volumes é igual ao cubo da razão dos raios:

```
Volume_cone1 / Volume_cone2 = (raio1 / raio2)^3
```

Paralelepípedo

Área da superfície do paralelepípedo: A área total da superfície de um paralelepípedo é a soma das áreas de suas faces:

```
Área = 2 * (comprimento * largura + comprimento * altura + largura * altura)
```

Volume do paralelepípedo: O volume de um paralelepípedo é dado por:

```
Volume = comprimento * largura * altura
```

Diagonais do paralelepípedo: As diagonais de um paralelepípedo podem ser calculadas usando o teorema de Pitágoras:

```
Diagonal1 = √(comprimento^2 + largura^2 + altura^2)
Diagonal2 = √(comprimento^2 + largura^2 + altura^2)
Diagonal3 = √(comprimento^2 + largura^2 + altura^2)
```

Esfera

Área da superfície da esfera: A área total da superfície de uma esfera é dada por:

```
Área = 4 * π * (raio)^2
```

Volume da esfera: O volume de uma esfera é calculado através da fórmula:

```
Volume = (4/3) * π * (raio)^3
```

Diâmetro da esfera: O diâmetro de uma esfera é duas vezes o raio:

```
Diâmetro = 2 * raio
```

Programação dinâmica

Problema do Troco

☐ of size N representing different types of denominations and an integer sum, the task is to find the number of ways to make sum by using different denominations.

Bottom-Up

Time complexity : O(N*sum)

Auxiliary Space : O(sum)

```
int count(int coins[], int n, int sum)
{
    // table[i] will be storing the number of solutions for
    // value i. We need sum+1 rows as the dp is
    // constructed in bottom up manner using the base case
    // (sum = 0)
    int dp[sum + 1];

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

    // Base case (If given value is 0)
    dp[0] = 1;

    // Pick all coins one by one and update the table[]
    // values after the index greater than or equal to the
    // value of the picked coin
    for (int i = 0; i < n; i++)
        for (int j = coins[i]; j <= sum; j++)
            dp[j] += dp[j - coins[i]];
    return dp[sum];
}


int coins[] = { 1, 2, 3 };
int n = sizeof(coins) / sizeof(coins[0]);
int sum = 5;
cout << count(coins, n, sum); // 5
```

Top-Down

```
int count(vector<int>& coins, int n, int sum,
          vector<vector<int> >& dp)
{
    // Base Case
```

```
    if (sum == 0)
        return dp[n][sum] = 1;

    // If number of coins is 0 or sum is less than 0 then
    // there is no way to make the sum.
    if (n == 0 || sum < 0)
        return 0;

    // If the subproblem is previously calculated then
    // simply return the result
    if (dp[n][sum] != -1)
        return dp[n][sum];

    // Two options for the current coin
    return dp[n][sum]
           = count(coins, n, sum - coins[n - 1], dp)
             + count(coins, n - 1, sum, dp);
}

int n, sum;
n = 3, sum = 5;
vector<int> coins = { 1, 2, 3 };
// 2d dp array to store previously calculated
// results
vector<vector<int> > dp(n + 1,
                        vector<int>(sum + 1, -1));
int res = count(coins, n, sum, dp);
cout << res << endl; // 5
```

Cortando canos

Dada uma relação de comprimentos de cano e seus respectivos valores de venda, determine o maior valor total que possa ser obtido com o corte de um cano de comprimento inicial determinado.

```
ll lucro_maximo(vector<pll>& canos, ll sizeCano) {
    vll memo(1e4 + 10, 0);
    rep(i, sizeCano + 1) {
        foreach (cano, canos) {
            if (i < cano.f) continue;
            memo[i] = max(memo[i], cano.s + memo[i - cano.f]);
        }
    }
    return memo[sizeCano];
}
```

Problema da Mochila

Suponha dado um conjunto de objetos, cada um com um certo peso e um certo valor. Quais dos objetos devo colocar na minha mochila para que o valor total seja o maior possível? Minha mochila tem capacidade para 15 kg apenas.

Bottom-Up

```
// Function to find the maximum profit
int knapSack(int W, int wt[], int val[], int n)
{
    // Making and initializing dp array
    int dp[W + 1];
    memset(dp, 0, sizeof(dp));

    for (int i = 1; i < n + 1; i++) {
        for (int w = W; w >= 0; w--) {

            if (wt[i - 1] <= w)

                // Finding the maximum value
                dp[w] = max(dp[w],
                            dp[w - wt[i - 1]] + val[i - 1]);
        }
    }
    // Returning the maximum value of knapsack
    return dp[W];
}
int profit[] = { 60, 100, 120 };
int weight[] = { 10, 20, 30 };
int W = 50;
int n = sizeof(profit) / sizeof(profit[0]);
cout << knapSack(W, weight, profit, n);
```

Top-Down

```
class Item {
   public:
    int peso;
    ll valor;
};

vector<vector<ll>> memo(100, vector<ll>(100005, -1));

ll valor_maximo(int item_atual, int capacidade_disponivel, vector<Item>& itens) {
    if (capacidade_disponivel < 0) return -LONG_MAX / 2;
    if (capacidade_disponivel == 0 || item_atual == itens.size()) return 0;

    if (memo[item_atual][capacidade_disponivel] != -1) return memo[item_atual]
[capacidade_disponivel];

    return memo[item_atual][capacidade_disponivel] = max(
               // Caso a capacidade se torne negativa o retorno será -LONG_MAX,
dessa forma assumimos que o valor
               // retornado é tão pequeno que será desconsiderado pelo MAX()
               itens[item_atual].valor + valor_maximo(item_atual + 1,
```

```
capacidade_disponivel - itens[item_atual].peso, itens),
               valor_maximo(item_atual + 1, capacidade_disponivel, itens));
}
cout << valor_maximo(0, capacidade, itens) << "\n";
```

Com tracking de itens

```
void printknapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] +
                    K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    // stores the result of Knapsack
    int res = K[n][W];
    cout<< res << endl;

    w = W;
    for (i = n; i > 0 && res > 0; i--) {

        // either the result comes from the top
        // (K[i-1][w]) or from (val[i-1] + K[i-1]
        // [w-wt[i-1]]) as in Knapsack table. If
        // it comes from the latter one/ it means
        // the item is included.
        if (res == K[i - 1][w])
            continue;
        else {

            // This item is included.
            cout<<" "<<wt[i - 1] ;

            // Since this weight is included its
            // value is deducted
            res = res - val[i - 1];
            w = w - wt[i - 1];
        }
    }
}
```

```
int val[] = { 60, 100, 120 };
int wt[] = { 10, 20, 30 };
int W = 50;
int n = sizeof(val) / sizeof(val[0]);

printknapSack(W, wt, val, n);
```

Com repetição de itens

```
// Returns the maximum value with knapsack of
// W capacity
int unboundedKnapsack(int W, int n,
                      int val[], int wt[])
{
    // dp[i] is going to store maximum value
    // with knapsack capacity i.
    int dp[W+1];
    memset(dp, 0, sizeof(dp));

    // Fill dp[] using above recursive formula
    for (int i=0; i<=W; i++)
      for (int j=0; j<n; j++)
         if (wt[j] <= i)
            dp[i] = max(dp[i], dp[i-wt[j]] + val[j]);

    return dp[W];
}
int W = 100;
int val[] = {10, 30, 20};
int wt[] = {5, 10, 15};
int n = sizeof(val)/sizeof(val[0]);

cout << unboundedKnapsack(W, n, val, wt); // 300
```

Com repetição e tracking dos itens

```
class UnboundedKnapsack {
   public:
    vector<ll> knapsack;
    vector<vector<ll>> selectedElements;
    ll maximumCapacity;

    vector<ll> knapsack_unbounded(vector<ll>& pesos, vector<ll>& valores, ll
num_itens, ll capacidade) {
        // Stores the maximum value which can be reached with a certain capacity
        knapsack.clear();
        knapsack.resize(capacidade + 1);
```

```
        maximumCapacity = capacidade + 1;

        // Stores selected elements with a certain capacity
        selectedElements.resize(capacidade + 1);

        // Initializes maximum value vector with zero
        for (ll i = 0; i < capacidade + 1; i++) {
            knapsack[i] = 0;
        }

        // Computes the maximum value that can be reached for each capacity
        for (ll capacity = 0; capacity < capacidade + 1; capacity++) {
            // Goes through all the elements
            for (ll n = 0; n < num_itens; n++) {
                if (pesos[n] <= capacity) {
                    if (knapsack[capacity] <= knapsack[capacity - pesos[n]] +
valores[n]) {
                        knapsack[capacity] = knapsack[capacity - pesos[n]] +
valores[n];

                        // Stores selected elements
                        selectedElements[capacity].clear();
                        selectedElements[capacity].push_back(n + 1);

                        for (ll elem : selectedElements[capacity - pesos[n]]) {
                            selectedElements[capacity].push_back(elem);
                        }
                    }
                }
            }
        }

        return this->selectedElements[capacidade];
    }
};

UnboundedKnapsack mochila;
vll index_itens_escolhidos = mochila.knapsack_unbounded(pesos, valores, num_itens,
capacidade);

ll sum = 0;
foreach (i, index_itens_escolhidos) {
    sum += valores[i - 1];
}

log(sum);
```

Problema da mochila fracionado

Given the weights and profits of N items, in the form of {profit, weight} put these items in a knapsack of capacity W to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

```
struct Item {
    int profit, weight;

    // Constructor
    Item(int profit, int weight)
    {
        this->profit = profit;
        this->weight = weight;
    }
};

// Comparison function to sort Item
// according to profit/weight ratio
static bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.profit / (double)a.weight;
    double r2 = (double)b.profit / (double)b.weight;
    return r1 > r2;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N)
{
    // Sorting Item on basis of ratio
    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items
    for (int i = 0; i < N; i++) {

        // If adding Item won't overflow,
        // add it completely
        if (arr[i].weight <= W) {
            W -= arr[i].weight;
            finalvalue += arr[i].profit;
        }

        // If we can't add current Item,
        // add fractional part of it
        else {
            finalvalue
                += arr[i].profit
                   * ((double)W / (double)arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}
```

```
int W = 50;
Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
int N = sizeof(arr) / sizeof(arr[0]);

cout << fractionalKnapsack(W, arr, N); // 240
```

LCS

Given two strings, S1 and S2, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.

A longest common subsequence (LCS) is defined as the longest subsequence which is common in all given input sequences.

Bottom-Up

Time Complexity: O(m * n) which remains the same.

Auxiliary Space: O(m) because the algorithm uses two arrays of size m.

```
int longestCommonSubsequence(string& text1, string& text2)
{
    int n = text1.size();
    int m = text2.size();

    // initializing 2 vectors of size m
    vector<int> prev(m + 1, 0), cur(m + 1, 0);

    for (int idx2 = 0; idx2 < m + 1; idx2++)
        cur[idx2] = 0;

    for (int idx1 = 1; idx1 < n + 1; idx1++) {
        for (int idx2 = 1; idx2 < m + 1; idx2++) {
            // if matching
            if (text1[idx1 - 1] == text2[idx2 - 1])
                cur[idx2] = 1 + prev[idx2 - 1];

            // not matching
            else
                cur[idx2]
                    = 0 + max(cur[idx2 - 1], prev[idx2]);
        }
        prev = cur;
    }

    return cur[m];
}
longestCommonSubsequence(S1, S2);
```

Top-Dowm

Time Complexity: O(m * n) where m and n are the string lengths.

Auxiliary Space: O(m * n) Here the recursive stack space is ignored.

```
int lcs(string& X, string& Y, int m, int n, vector<vector<int>>& dp) {
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {
        return dp[m][n];
    }
    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                          lcs(X, Y, m - 1, n, dp));
}

vector<vector<int>> dp(s1.size() + 1, vector<int>(s2.size() + 1, -1));
ll resp = lcs(s1, s2, s1.size(), s2.size(), dp);
```

LIS

Longest Increasing Subsequence

```
int lis(int arr[], int n)
{
    int lis[n];

    lis[0] = 1;

    // Compute optimized LIS values in
    // bottom up manner
    for (int i = 1; i < n; i++) {
        lis[i] = 1;
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;
    }

    // Return maximum value in lis[]
    return *max_element(lis, lis + n);
}

int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
int n = sizeof(arr) / sizeof(arr[0]);
lis(arr, n); // 5
```

Graph

BFS with path

```
class Graph {
  public:
    int num_nodes;
    vector<vi> edges;

    Graph(int qtd_nodes){
        num_nodes = qtd_nodes + 1;

        edges.resize(num_nodes);
    }

    void add_edge(int u, int v) {
        edges[u].pb(v);
        edges[v].pb(u);
    }

    vi bfs(int from, int to) {
        vector<bool> visited(num_nodes);
        queue<int> q;
        vi parent(num_nodes);

        q.push({ from });
        parent[1] = 0;
        visited[1] = true;

        while(!q.empty()) {
            int current = q.front();
            q.pop();

            for(auto neighbor: edges[current]) {
                if(visited[neighbor]) continue;

                parent[neighbor] = current;
                visited[neighbor] = true;

                q.push(neighbor);
            }
        }

        if(!visited[n]) return vi();

        vi path;
        for(int at = n; at != 0; at = parent[at]) {
            path.pb(at);
        }
```

```
        return path;
    }
};

void solve() {
    cin >> n >> m;

    Graph graph(n);
    for(int i=0;i<m;i++) {
        int u, v;
        cin >> u >> v;
        graph.add_edge(u, v);
    }

    vi path = graph.bfs(1, n);

    if(path.size() != 0) {
        cout << path.size() << endl;
        for(int i=path.size() - 1; i>=0;i--) cout << path[i] << " ";
        cout << endl;
    } else
        cout << "IMPOSSIBLE" << endl;
}
```

Finding Connected Components (Undirected Graph) - O(V+E)

```
class Grafo {
   public:
    int num_vertices;
    map<string, vector<string>> graph;
    set<string> visited;

    Grafo(int n) {
        num_vertices = n;
    }

    void add_edge(string u, string v) {
        graph[u].pb(v);
        graph[v].pb(u);
    }

    void dfs(string u) {
        stack<string> pilha;
        pilha.push(u);
        visited.insert(u);

        while (!pilha.empty()) {
            string atual = pilha.top();
            pilha.pop();

            for (auto vizinho : graph[atual]) {
```

```
                if (visited.find(vizinho) != visited.end()) continue;

                pilha.push(vizinho);
                visited.insert(vizinho);
            }
        }
    }

    int components() {
        int resp = 0;
        for (auto vertice : graph) {
            if (visited.find(vertice.f) != visited.end()) continue;

            dfs(vertice.f);
            resp++;
        }

        return resp;
    }
};

void solve() {
    int vertices, arestas;
    cin >> vertices >> arestas;

    Grafo grafo(vertices);

    for (int i = 0; i < arestas; i++) {
        string u, w, v;
        cin >> u >> w >> v;
        grafo.add_edge(u, v);
    }

    cout << grafo.components() << endl;
}
```

Topological Sort (Directed Acyclic Graph) - O(V+E)

One application of topological sorting is to find a possible sequence of modules that aUniversity student has to take to fulfill graduation requirement. Each module has certainpre-requisites to be met. These pre-requisites are never cyclic, so they can be modeled as aDAG. Topological sorting this module pre-requisites DAG gives the student a linear list ofmodules to be taken one after another without violating the pre-requisites constraints.

Simple DFS Variant

```
#include <iostream>
#include <vector>
#include <stack>
```

```
using namespace std;

// Function to perform DFS and store the result in the stack
void topologicalSortDFS(int v, vector<bool> &visited, stack<int> &Stack, const
vector<vector<int>> &adj) {
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    for (int i : adj[v]) {
        if (!visited[i]) {
            topologicalSortDFS(i, visited, Stack, adj);
        }
    }

    // Push the current vertex to stack which stores the result
    Stack.push(v);
}

// Function to perform Topological Sort on the graph
void topologicalSort(int V, const vector<vector<int>> &adj) {
    stack<int> Stack;
    vector<bool> visited(V, false);

    // Call the recursive helper function to store Topological Sort starting from
all vertices one by one
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            topologicalSortDFS(i, visited, Stack, adj);
        }
    }

    // Print contents of the stack
    while (!Stack.empty()) {
        cout << Stack.top() << " ";
        Stack.pop();
    }
    cout << endl;
}

int main() {
    // Number of vertices
    int V = 6;

    // Adjacency list representation of the graph
    vector<vector<int>> adj(V);

    // Adding edges (Graph is Directed)
    adj[5].push_back(2);
    adj[5].push_back(0);
    adj[4].push_back(0);
    adj[4].push_back(1);
    adj[2].push_back(3);
    adj[3].push_back(1);
```

```
    // Perform Topological Sort
    topologicalSort(V, adj);

    return 0;
}
```

Bipartite Graph Check (Undirected Graph) - O(V*V)

```
class Grafo {
   public:
    int num_vertices;
    vector<vi> edges;
    vector<int> color;

    Grafo(int n) {
        num_vertices = n;

        edges.resize(num_vertices);
        color.resize(num_vertices);
    }

    void add_edge(int u, int v) {
        edges[u].pb(v);
        edges[v].pb(u);
    }

    // Devolve true quando uma componente pode ser bipartida
    bool dfs(int u, int c) {
        if (color[u] != 0)
            return color[u] == c;

        color[u] = c;

        bool is_bipartite = true;
        for (auto neighbor : edges[u]) {
            is_bipartite &= dfs(neighbor, c == 1 ? 2 : 1);
        }

        return is_bipartite;
    }

    // Devolve true quando todas as componentes são bipartidas
    bool is_bipartite() { return dfs(0, 1); }
};

void solve() {
    Grafo grafo(N);

    for (int i = 0; i < M; i++) {
        int u, v;
```

```
        cin >> u >> v;

        grafo.add_edge(u, v);
    }

    cout << (grafo.is_bipartite() ? "BICOLORABLE." : "NOT BICOLORABLE.") << endl;
}
```

Cycle Check (Directed Graph) - O(V+E)

```
#include <iostream>
#include <vector>

using namespace std;

// Function to perform DFS and check for a cycle
bool isCyclicDFS(int v, vector<bool> &visited, vector<bool> &recStack, const
vector<vector<int>> &adj) {
    // Mark the current node as visited and part of the recursion stack
    visited[v] = true;
    recStack[v] = true;

    // Recur for all vertices adjacent to this vertex
    for (int i : adj[v]) {
        // If the adjacent vertex is not visited, then recurse on it
        if (!visited[i] && isCyclicDFS(i, visited, recStack, adj)) {
            return true;
        }
        // If the adjacent vertex is already in the recursion stack, there's a
cycle
        else if (recStack[i]) {
            return true;
        }
    }

    // Remove the vertex from the recursion stack
    recStack[v] = false;
    return false;
}

// Function to check if the graph contains a cycle
bool isCyclic(int V, const vector<vector<int>> &adj) {
    vector<bool> visited(V, false);
    vector<bool> recStack(V, false);

    // Call the recursive helper function to detect cycle in different DFS trees
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            if (isCyclicDFS(i, visited, recStack, adj)) {
                return true;
            }
```

```
            }
        }
        return false;
    }

    int main() {
        // Number of vertices
        int V = 4;

        // Adjacency list representation of the graph
        vector<vector<int>> adj(V);

        // Adding edges (Graph is Directed)
        adj[0].push_back(1);
        adj[1].push_back(2);
        adj[2].push_back(3);
        adj[3].push_back(1);  // This edge introduces a cycle

        // Check if the graph contains a cycle
        if (isCyclic(V, adj)) {
            cout << "Graph contains a cycle" << endl;
        } else {
            cout << "Graph doesn't contain a cycle" << endl;
        }

        return 0;
    }
```

Finding Strongly Connected Components (Directed Graph) - O(V+E)

```
#include <iostream>
#include <list>
#include <stack>
#include <vector>
using namespace std;

class Graph {
    int V; // Number of vertices
    list<int> *adj; // Adjacency list

    // A function to perform DFS and fill the stack
    void fillOrder(int v, bool visited[], stack<int> &Stack);

    // A function to perform DFS on the transposed graph
    void DFS(int v, bool visited[]);

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // Function to add an edge to the graph
    void printSCC(); // Function to print all strongly connected components
    Graph transpose(); // Function to get the transposed graph
```

```
};

// Constructor
Graph::Graph(int V) {
    this->V = V;
    adj = new list<int>[V]; // Create an adjacency list
}

// Function to add an edge to the graph
void Graph::addEdge(int v, int w) {
    adj[v].push_back(w); // Add w to v's list
}

// A function to perform DFS and fill the stack
void Graph::fillOrder(int v, bool visited[], stack<int> &Stack) {
    visited[v] = true; // Mark the current node as visited
    for (int i : adj[v]) {
        if (!visited[i]) {
            fillOrder(i, visited, Stack); // Recur for all the vertices adjacent
to this vertex
        }
    }
    Stack.push(v); // Push current vertex to stack
}

// A function to perform DFS on the transposed graph
void Graph::DFS(int v, bool visited[]) {
    visited[v] = true; // Mark the current node as visited
    cout << v << " "; // Print the current node

    // Recur for all the vertices adjacent to this vertex
    for (int i : adj[v]) {
        if (!visited[i]) {
            DFS(i, visited);
        }
    }
}

// Function to get the transposed graph
Graph Graph::transpose() {
    Graph g(V); // Create a new graph
    for (int v = 0; v < V; v++) {
        for (int i : adj[v]) {
            g.adj[i].push_back(v); // Reverse the direction of edges
        }
    }
    return g;
}

// Function to print all strongly connected components
void Graph::printSCC() {
    stack<int> Stack; // Stack to hold the vertices
    bool *visited = new bool[V]; // Mark all the vertices as not visited
```

```
    // Fill vertices in stack according to their finishing times
    for (int i = 0; i < V; i++) {
        visited[i] = false; // Initialize visited array
    }
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            fillOrder(i, visited, Stack); // Fill the stack
        }
    }

    // Create a transposed graph
    Graph transposedGraph = transpose();

    // Mark all the vertices as not visited for the second DFS
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }

    // Now process all vertices in order defined by the stack
    while (!Stack.empty()) {
        int v = Stack.top();
        Stack.pop();
        if (!visited[v]) {
            transposedGraph.DFS(v, visited); // Perform DFS on the transposed
graph
            cout << endl; // Print a new line after each component
        }
    }
}

// Driver code
int main() {
    Graph g(8); // Create a graph with 8 vertices
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 4);
    g.addEdge(4, 5);
    g.addEdge(5, 6);
    g.addEdge(6, 4);
    g.addEdge(7, 6);

    cout << "Strongly Connected Components in the graph:\n";
    g.printSCC(); // Print all strongly connected components
    return 0;
}
```

Kruskal’s Algorithm - O(E log E)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E; // Number of vertices and edges
    vector<Edge> edges;
};

// Structure to represent a subset for union-find
struct Subset {
    int parent;
    int rank;
};

// Function to find the subset of an element `i` (uses path compression)
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i) {
        subsets[i].parent = find(subsets, subsets[i].parent);
    }
    return subsets[i].parent;
}

// Function to unite two subsets `x` and `y` (uses union by rank)
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Comparator function to sort edges by their weight in non-decreasing order
bool compare(Edge e1, Edge e2) {
    return e1.weight < e2.weight;
}

// Function to implement Kruskal's algorithm
void KruskalMST(Graph &graph) {
```

```
    int V = graph.V;
    vector<Edge> result;  // To store the resultant MST
    int e = 0;  // An index variable for result[]

    // Step 1: Sort all the edges in non-decreasing order of their weight
    sort(graph.edges.begin(), graph.edges.end(), compare);

    // Allocate memory for creating V subsets
    Subset *subsets = new Subset[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    int i = 0;  // Index used to pick the smallest edge
    while (e < V - 1 && i < graph.edges.size()) {
        // Step 2: Pick the smallest edge. Check if it forms a cycle with the
spanning tree
        Edge next_edge = graph.edges[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does not cause a cycle, include it in the result
        // and move to the next edge.
        if (x != y) {
            result.push_back(next_edge);
            Union(subsets, x, y);
            e++;
        }
    }

    // Print the contents of the resultant MST
    cout << "Following are the edges in the constructed MST:" << endl;
    for (i = 0; i < result.size(); ++i) {
        cout << result[i].src << " -- " << result[i].dest << " == " <<
result[i].weight << endl;
    }

    // Free the allocated memory
    delete[] subsets;
}

int main() {
    // Create a graph with 4 vertices and 5 edges
    int V = 4, E = 5;
    Graph graph = {V, E};

    // Add edges
    graph.edges.push_back({0, 1, 10});
    graph.edges.push_back({0, 2, 6});
```

```
    graph.edges.push_back({0, 3, 5});
    graph.edges.push_back({1, 3, 15});
    graph.edges.push_back({2, 3, 4});

    // Run Kruskal's algorithm
    KruskalMST(graph);

    return 0;
}
```

Dijkstra's Algorithm - O(V + E * log V)

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

using namespace std;

#define SPEED cin.tie(0)->sync_with_stdio(0);
#define DEBUG false
#define db(x) \
    if (DEBUG) cout << #x << ": " << x << endl
#define dbpair(x) \
    if (DEBUG) cout << #x << ": " << x.f << ", " << x.s << endl
#define dbvector(vector)                                \
    {                                                   \
        cout << #vector << " = ";                       \
        for (auto& it : vector) cout << it << " "; \
        cout << endl;                                   \
    }
#define dbmap(map)                              \
    for (auto e : map)                          \
        cout << e.first << " " << e.second; \
    cout << endl
#define all(x) begin(x), end(x)
#define pb push_back
#define pf push_front
#define endl "\n"
#define f first
#define s second
#define MOD 1e9 + 7
#define mp make_pair

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef long double ld;
typedef priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> min_pq;

const int INF = 0x3f3f3f3f;
```

```
const ll LINF = 0x3f3f3f3f3f3f3f3fll;

int n, m, k;

class Graph {
   private:
    int num_vertex;
    vector<vector<pii>> edges;

   public:
    Graph(int n) {
        num_vertex = n + 1;
        edges.resize(num_vertex);
    }

    void add_edge(int u, int v, int weight) {
        edges[u].pb({v, weight});
        edges[v].pb({u, weight});
    }

    int dijkstra(int origin, int target) {
        vi distance(num_vertex);
        vi visited(num_vertex);

        min_pq pq;

        for (int i = 0; i < num_vertex; i++) {
            distance[i] = INF;
            visited[i] = false;
        }

        distance[origin] = 0;

        pq.push({distance[origin], origin});

        while (!pq.empty()) {
            int current = pq.top().s;

            pq.pop();

            if (visited[current]) continue;

            visited[current] = true;

            for (auto neighborEdge : edges[current]) {
                int neighbor = neighborEdge.f;
                int weight = neighborEdge.s;

                if (distance[current] + weight < distance[neighbor]) {
                    distance[neighbor] = distance[current] + weight;

                    pq.push({distance[neighbor], neighbor});
                }
            }
```

```
        }

        return distance[target];
    }
};

void solve() {
    cin >> n >> m;

    Graph graph(n);
    for (int i = 0; i < m; i++) {
        int u, v, weight;
        cin >> u >> v >> weight;

        graph.add_edge(u, v, weight);
    }

    cout << graph.dijkstra(1, n) << endl;
}

int main(int argc, char** argv) {
    SPEED;

    solve();

    return 0;
}
```

Bellman-Ford Algorithm - O(V*E)

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Function to perform the Bellman-Ford algorithm
void BellmanFord(int V, int E, const vector<Edge>& edges, int src) {
    // Initialize the distance to all vertices as infinite and source distance as
0
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    // Relax all edges V-1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
```

```
            int u = edges[j].src;
            int v = edges[j].dest;
            int weight = edges[j].weight;

            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    // Check for negative-weight cycles
    for (int j = 0; j < E; j++) {
        int u = edges[j].src;
        int v = edges[j].dest;
        int weight = edges[j].weight;

        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graph contains a negative-weight cycle" << endl;
            return;
        }
    }

    // Print the calculated shortest distances
    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < V; i++) {
        cout << i << "\t\t" << dist[i] << endl;
    }
}

int main() {
    // Number of vertices and edges
    int V = 5; // Number of vertices
    int E = 8; // Number of edges

    // Create a graph
    vector<Edge> edges(E);

    // Define the edges of the graph
    edges[0] = {0, 1, -1};
    edges[1] = {0, 2, 4};
    edges[2] = {1, 2, 3};
    edges[3] = {1, 3, 2};
    edges[4] = {1, 4, 2};
    edges[5] = {3, 2, 5};
    edges[6] = {3, 1, 1};
    edges[7] = {4, 3, -3};

    // Run Bellman-Ford algorithm from source vertex 0
    BellmanFord(V, E, edges, 0);

    return 0;
}
```

Floyd-Warshall Algorithm - $O(V^3)$

```
#include <iostream>
#include <vector>
#include <iomanip>

using namespace std;

const int INF = 1e9; // A large value to represent infinity

// Function to print the distance matrix
void printSolution(const vector<vector<int>> &dist, int V) {
    cout << "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)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }
}

// Function to implement Floyd-Warshall algorithm
void floydWarshall(vector<vector<int>> &graph, int V) {
    // Initialize distance matrix
    vector<vector<int>> dist = graph;

    // Adding vertices individually
    for (int k = 0; k < V; ++k) {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; ++i) {
            // Pick all vertices as destination for the above picked source
            for (int 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] != INF && dist[k][j] != INF)
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }

    // Detect negative cycles
    for (int i = 0; i < V; ++i) {
        if (dist[i][i] < 0) {
            cout << "Graph contains a negative-weight cycle.\n";
            return;
        }
    }

    // Print the shortest distance matrix
```

```
    printSolution(dist, V);
}

int main() {
    // Number of vertices
    int V = 4;

    // Initialize the graph with INF
    vector<vector<int>> graph(V, vector<int>(V, INF));

    // Distance from a vertex to itself is always 0
    for (int i = 0; i < V; ++i)
        graph[i][i] = 0;

    /*
        Example Graph:
        Edge list with weights:
        0 -> 1 (5)
        0 -> 3 (10)
        1 -> 2 (3)
        2 -> 3 (1)
        3 -> 1 (-2)
    */

    // Adding edges
    graph[0][1] = 5;
    graph[0][3] = 10;
    graph[1][2] = 3;
    graph[2][3] = 1;
    graph[3][1] = -2;

    // Display the initial graph
    cout << "Initial graph adjacency matrix:\n";
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (graph[i][j] == INF)
                cout << "INF ";
            else
                cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;

    // Run Floyd-Warshall algorithm
    floydWarshall(graph, V);

    return 0;
}
```