Universidade Tecnológica Federal do Paraná - UTFPR

Clube de Programação

Discentes:
Gabriel Moro Conke,
Henrique Alberto Padilha Mendes,
Pedro Henrique de Freitas Neves.

Conteúdo

1 Template e Compilação

1.1 Template

- Template:

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

#define fast_io ios_base::sync_with_stdio(false);
cin.tie(NULL); cout.tie(NULL)
#define endl '\n'
#define ll long long
#define ii pair<int,int>

const int MAX = 212345;

int main(){
    fast_io;

    return 0;
}
```

- Uma vez criado o template, para criar os arquivos da questão realizar o comando:

```
cp template.cpp X.cpp
```

1.2 Compilação

1. Entrar no diretório que está o arquivo .cpp
2. Compilar (substituir "A.cpp"pelo arquivo .cpp em questão):

```
g++ A.cpp -Wall -O2
```

3. Executar o comando "./a.out < tmp.in", onde tmp.in é o arquivo com as entradas

2 STL Stuffs

As estruturas Vector, List e Deque armazenam os elementos em ordem linear, seguindo a estrutura na qual foram inseridos.

2.1 Pair

Combinação de dois valores que podem (ou não) possuir tipos diferentes.

```
pair <type1, type2> Pair_name
//Criação de um pair.

pair <type1, type2> Pair_name (value1, value2)
//Criação do um pair com valores já escritos.

pair pair_name (1, 'a')
//Criação de um pair sem pré-determinar os tipos.

g2 = make_pair(1, 'a')
g2 = {1, 'a'}
//Atribuição de um pair (ideal para uso em vectors).
```

2.2 Vector

Vetor de alocação dinâmica, que possui os seguintes comandos principais:

```
begin()
//Returns an iterator pointing to the first element.

end()
//Returns an iterator pointing to the last element.

size()
//Returns the number of elements in the vector.

empty()
//Returns whether the container is empty.

front()
//Returns a reference to the first element in the vector.

nome_vec[i]
```

```
//Returns a reference to the element at position ‘i’ in the vector.

push_back(element)
//Push the elements into a vector from the back. (copia o elemento)

pop_back()
//Remove elements from a vector from the back.

insert(index, val)
//Inserts new elements before the element at the specified position
//No casos envolvendo inserção de um caracter em uma string, deve-se usar
str.insert(index, 1, ’char’);

clear()
//Destroy all the elements of the vector.

erase(posIn, posFinal)
//Remove elements from a container from the specified range (used with iterator
- Feito em tempo O(n). Set faz esse processo em tempo O(1).

emplace(position /*Iterator*/, element);
//Insert a element at the position (constrói o elemento).

emplace_back(value);
//Insert a element at the end of the vector.
```

2.3 List

2.4 Deque

Lista que possui dois fins (um no começo e outro no final). - Parecido com a fila, mas faz a inserção/remoção do começo/fim em O(1).

```
deque_name.insert (iterator, value);
//Insere o elemento na posição do itarator.

deque_name.push_front(value);
deque_name.push_back(value);
//Adiciona um elemento no começo/fim do deque.

deque_name.pop_front();
```

```
deque_name.pop_back();
//Retira um elemento do começo/fim do deque. (possui tipo void).

deque_name.clear();
//Exclui todos os elementos do deque.

deque_name.erase(iterator1, iterator2);
//Exclui os elementos do intervalo delimitado.

deque_name.empty();
//Verifica se o deque está vazio.

deque_name.front();
deque_name.back();
//Acessa os elementos do começo/fim do deque.

deque_name.size();
//Verifica o tamanho do deque.
```

2.5 Stack

Feito de acordo com os comandos padrões do vector.

2.6 Queue

2.7 Priority Queue

2.8 Map

2.9 Set

erase(posIn, posFinal) //Remove elements from a container from the specified range (used with iterators). - Feito em tempo O(1).

2.10 MultiSet

3 Vetores

3.1 Inicializacao

3.1.1 1D

- vector<int> v(size, val);

3.1.2 2D

- vector<vector<int> > a(rows, vector<int>(columns, 0));

3.2 Difference Array

Utilizado para somar ou subtrair valores de um vetor dado um range e um comportamento específico.

- Estrutura inicial: array a[]
- Operações de range O(1) ($\pm d$)
- Restaurar os valores do array: O(n)

```
delta = 0;
for (int i = 0; i < MAX; i++){
    delta += diff[i]; // array de diferenças
    a[i] += delta;
}
```

3.2.1 Difference Array 2D

- dcol é um vetor de acumulado para as colunas, responsável por garantir o comportamento do range de somas.

```
for (int i = 0; i < MAX; i++){
    int delta = 0;
    for (int j = 0; j < MAX; j++){
        dcol[j] += diff[i][j];
        delta += dcol[j];
        a[i][j] += delta;
    }
}
```

3.3 Segment tree

Extensivamente utilizada para fazer operações em intervalos de vetores, tanto atualização como consultas sao feitas em $O(\log n)$.
Um teste com n operações de consulta implementadas linearmente criariam um algoritmo $O(n^2)$, enquanto aplicando SegTree, n operações geram um algoritmo $O(n \log n)$.

OBS: a árvore é indexada com o pai em 1, ou seja, chamar _build(1, 0, n−1), _query(1, 0, n−1, i, j), _update(1, 0, n−1, i, val)

3.3.1 SegTree Aditiva

```
int a[MAX], tree[4*MAX];

void build(int node, int start, int end){
    // Leaf node
    if(start == end) tree[node] = a[start];
    else{
        int mid = (start + end) >> 1;

        build(2*node, start, mid);
        build(2*node+1, mid+1, end);

        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

void update(int node, int start, int end, int idx, int val){
    // Leaf node
    if(start == end){
        A[idx] += val;
        tree[node] += val;
        return;
    }

    int mid = (start + end) >> 1;

    if(start <= idx and idx <= mid)
        update(2*node, start, mid, idx, val); // left child
    else
```

```
        update(2*node+1, mid+1, end, idx, val); // right child

    tree[node] = tree[2*node] + tree[2*node+1];

}

int query(int node, int start, int end, int l, int r){
    // range completely outside
    if(r < start or end < l) return 0; // sum neutral

    // range completely inside
    if(l <= start and end <= r) return tree[node];

    // range partially inside and partially outside
    int mid = (start + end) >> 1;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```

3.3.2 SegTree Multiplicativa

```
int a[MAX], tree[4*MAX];

void build(int node, int start, int end){
    // Leaf node
    if(start == end) tree[node] = a[start];
    else{
        int mid = (start + end) >> 1;

        build(2*node, start, mid);
        build(2*node+1, mid+1, end);

        tree[node] = tree[2*node] * tree[2*node+1];
    }
}

void update(int node, int start, int end, int idx, int val){
    // Leaf node
    if(start == end){
```

```
            A[idx] += val;
            tree[node] += val;
            return;
        }

        int mid = (start + end) >> 1;

        if(start <= idx and idx <= mid)
            update(2*node, start, mid, idx, val); // left child
        else
            update(2*node+1, mid+1, end, idx, val); // right child

        tree[node] = tree[2*node] * tree[2*node+1];

    }

    int query(int node, int start, int end, int l, int r){
        // range completely outside
        if(r < start or end < l) return 1; // multiplication neutral

        // range completely inside
        if(l <= start and end <= r) return tree[node];

        // range partially inside and partially outside
        int mid = (start + end) >> 1;
        int p1 = query(2*node, start, mid, l, r);
        int p2 = query(2*node+1, mid+1, end, l, r);
        return (p1 * p2);
    }
```

3.3.3 SegTree Aditiva com Lazy Propagation

Lazy propagation serve para atualizações em intervalos.

Código adaptado para range queries e range updates (lazy propagation)

```
    vector<ll> a(MAX), tree(4*MAX), lazy(4*MAX);

    // call build(1, 0, n-1);
    void build(int node, int start, int end){
        if (start == end) tree[node] = a[start];
```

```
    else{
        int mid = (start+end)>>1;

        build(2*node,start,mid);
        build(2*node+1,mid+1,end);

        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

// call updateRange(1, 0, n-1, l, r, val);
void updateRange(int node, int start, int end, int l, int r, int val){
    if (lazy[node]){
        tree[node] += (end-start+1) * lazy[node];
        if (start != end){
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }
        lazy[node] = 0;
    }

    if (start>end || start>r || end<l) return;

    if (start>=l && end<=r){
        tree[node] += (end-start+1) * val;
        if (start != end){
            lazy[2*node] += val;
            lazy[2*node+1] += val;
        }
        return;
    }

    int mid = (start+end) >> 1;
    updateRange(2*node, start, mid, l, r, val);
    updateRange(2*node+1, mid+1, end, l, r, val);
    tree[node] = tree[2*node] + tree[2*node+1];
}

// call queryRange(1, 0, n-1, l, r);
ll queryRange(int node, int start, int end, int l, int r){
```

```
    if (start>end || start>r || end<l) return 0;

    if (lazy[node]){
        tree[node] += (end-start+1) * lazy[node];
        if (start != end){
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }
        lazy[node] = 0;
    }

    if (start>=l && end<=r) return tree[node];

    int mid = (start+end)>>1;
    ll p1 = queryRange(2*node, start, mid, l, r);
    ll p2 = queryRange(2*node+1, mid+1, end, l, r);
    return (p1+p2);
}
```

3.4 Fenwick Tree (BIT)

Uma estrutura que, de modo semelhante à SegTree, realiza operações de consulta e de atualização ambas em $O(\log n)$. OBS: a árvore bit é indexada a partir de 1.

3.4.1 BIT aditiva

```
int n, bit[MAX];

void setbit(int i, int delta){
    while (i <= n){
        bit[i] += delta;
        i += i&-i; // i&-i retorna a maior potencia de 2 que divide i
    }
}

int getbit(int i){
    int ans = 0;
    while(i){
        ans += bit[i];
        i -= i&-i; // subtrai maior potencia de 2 que divide i
    }
```

```
        return ans;
    }
```

3.5 Bitset

Maneira de armazenar uma string binária de forma eficiente, com complexidade O(n/32) e O(n/64). A utilização pode ser feita da mesma forma que vetores, com a condição da estrutura favorecer operações bit a bit.

```
bitset<n> nome(k); /*criar bitset de tamanho n (deve ser
conhecido em tempo de compilação) que armazena o número k.*/
```

4 Ordenação e Busca

4.1 Insertion Sort

Complexidade O(n^2);

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

4.2 Merge Sort

Complexidade O(n log(n))

```
void merge(vector<int> &array, int left, int mid, int right){
    vector<int> firstHalf (mid-left+1);
    vector<int> secondHalf (right-mid);

    for (int i=0; i < mid-left+1; i++)
        firstHalf[i] = array[left+i];
    for (int i=0; i < right-mid; i++)
        secondHalf[i] = array[mid + i + 1];

    int j=0, k=0, i=left;

    while (j<mid-left+1 && k < right - mid){
        if (firstHalf[j] <= secondHalf[k])
            array[i++] = firstHalf[j++];
        else
            array[i++] = secondHalf[k++];
    }

    while (j < mid-left+1)
```

```
        array[i++] = firstHalf[j++];
    while (k < right-mid)
        array[i++] = secondHalf[k++];
}


void mergeSort(vector<int> &array, int begin, int end){
    if (begin >= end)
        return;

    int mid = (begin + end)/2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}
```

4.3 STL Sort

Ordena os elementos de um vector em complexidade O(n*log(n))

```
    sort (Iterator_begin, Iterator_first);
    //Geralmente utilizado da seguinte forma,
    sort (vector_name.begin(), vector_name.end());
```

4.4 Binary Search

Para a realização da busca binária, é necessário que o vetor esteja inicialmente ordenado. Existem comandos da STL que facilitam a realização do processo.

– UTILIZANDO VECTOR – 1. lower_bound: retorna um iterator da primeira posição do vetor com um valor não menor que o desejado.

```
auto it = lower_bound(vec.begin(), vec.end(), wanted);
```

2. upper_bound: retorna um iterator apontando para o primeiro elemento maior que o valor desejado.

```
auto it = upper_bound(vec.begin(), vec.end(), wanted);
```

- Nos casos em que todos os valores sejam menores que "wanted"a função retorna um iterador apontando para o último elemento do vector + 1.

– UTILIZANDO SET – (utilizar as funções abaixo, caso contrário existem casos O(n))

1. lower_bound: retorna um iterator da primeira posição do vetor com um valor não menor que o desejado.

```
auto it = set_name.lower_bound(wanted);
```

2. upper_bound: retorna um iterator apontando para o primeiro elemento maior que o valor desejado.

```
auto it = set_name.upper_bound(wanted);
```

- Nos casos em que todos os valores sejam menores que "wanted"a função retorna um iterador apontando para o último elemento do vector + 1.

5 Combinatória

5.1 Permutação

Dados um conjunto com n caixas iguais, queremos saber as diferentes formas de organizar n bolas.

É descobrir todas as permutações por meio de uma função da STL de complexidade $O(n * n!)$

```
bool next_permutation (Iterator 1, Iterator 2);
//Rearranja os elementos na próxima permutação, sendo essa a menor
possível maior que a atual (no quesito lexicográfico); retornando
se a operação foi bem sucedida ou não.
```

5.1.1 Permutação sem repetição

$$P(n) = n! \tag{1}$$

5.1.2 Permutação com repetição

$$P(n, k, l, ...) = n!/k!l!... \tag{2}$$

k, l,... representam a quantidade de vezes que a bola (k, l, ...) se repete

5.2 Combinação

Dados um conjunto com n caixas iguais, queremos saber as possibilidades para se colocar k bolas iguais dentro delas

$$C(n, k) = \frac{n!}{k!(n-k)!} \tag{3}$$

6 Strings

6.1 Comandos básicos

6.2 Substrings

A função substr de string retorna uma substring interna da string inicial, os parâmetros são:

- Posição inicial
- Tamanho da substring

Ex: substring que remove o último char da string str.

```
str.substr(0, str.length()-1);
```

6.3 char to int

Para converter um char (i-ésimo char da string) em int, é possível fazer:

```
int x = str[i] - '0';
```

6.4 int to string

Para converter um int x em uma string com seus algarismos, fazemos:

```
string str = to_string(x);
```

7 Teoria dos Números

1- Número de subconjuntos de um conjunto é 2^n, sendo n o número de elementos do conjunto.

7.1 Exponenciação binária

Queremos calcular um número a^x, podemos fazer isso em complexidade $O(logx)$, utilizando o seguinte algoritmo:

```
long long power(long long base, long long exp) {
    long long result = 1;

    while (exp > 0) {
        if (exp % 2 == 1) {
            result *= base;
        }
        base *= base;
        exp /= 2;
    }

    return result;
}
```

7.1.1 Exponenciação binária modular

Podemos querer realizar operaçõs de exponenciais grandes, mod algum primo grande. Para isso (REVISAR ESTE CODIGO):

```
// P é o primo grande em questão (i.e. whatsapp 998244353)
long long power(long long base, long long exp) {
    long long result = 1;

    while (exp > 0) {
        if (exp % 2 == 1) {
            result *= base;
            result %= P;
        }
        base *= base;
        base %= P;
        exp /= 2;
```

```
        }

        return result;
    }
```

7.2 Aritmética modular

Propriedades:

- $(a + b + c)mod(n) = ((a + b)mod(n) + c)mod(n)$
- $(a - b)mod(n)$

  Esta operação tem um problema, pois seu valor pode ser negativo, para contornar isso, fazemos: $(a+n-b)mod(n)$. ATENÇÃO: os valores de a e b devem ser válidos, isto é, $a, b \in \{0, 1, \cdots, n-1\}$.

- $(a \cdot b \cdot c)\ mod(n) = ((a \cdot b)\ mod(n)) \cdot c\ mod(n)\cdot$
- $(a \cdot b)\ mod(n) = ((a\ mod(n) \cdot b\ mod(n))\ mod(n)$
- $(a \cdot b^{-1})mod(n)$

  b^{-1} pode ser lido como o inverso multiplicativo de b, ou seja, temos que $b \cdot \frac{1}{b} = 1\ mod(n)$. Se n for primo, todo a, b terá inverso multiplicativo (com exceção de 0).

  Pequeno Teorema de Fermat ($a \neq 0$ e p é primo):

  1. $a^p \equiv a\ mod(p)$
  2. $a^{p-1} \equiv 1 mod(p)$
  3. Logo: $a \cdot a^{p-2} \equiv 1\ mod(p)$, portanto: $a^{p-2} \equiv a^{-1}$
  4. Para calcular a^{p-2}, utilizar exponenciação binária modular.

7.3 Recorrências lineares

Em recorrências lineares, é possível encontrar fórmulas fechadas com complexidade $O(\log n)$. Para isto, é utilizada a exponenciação binária de matrizes.

Exemplo: Fibonnacci:

$$f_n = f_{n-1} + f_{n-2},\ \text{para } n \geq 2.$$

Ainda, é possível obter uma transformação linear A de $\mathbb{C}^2$ em $\mathbb{C}^2$ tal que, para todo $n \geq 2$,

$$\mathsf{A}\begin{pmatrix} f(n-2) \\ f(n-1) \end{pmatrix} = \begin{pmatrix} f(n-1) \\ f(n) \end{pmatrix},$$

bastando tomar

$$\mathsf{A} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Assim, indutivamente, temos, para todo $n \geq 2$, que

$$\mathsf{A}^{n-1}\begin{pmatrix} f(0) \\ f(1) \end{pmatrix} = \begin{pmatrix} f(n-1) \\ f(n) \end{pmatrix}.$$

Segue o algoritmo abaixo para o cálculo do n-ésimo número de Fibonacci.

7.3.1 Multiplicação de matrizes

```
vector<vector<int>> matMult(vector<vector<int>>& A, vector<vector<int>>& B) {
int m = A.size();    // Número de linhas em matrizA
int n = A[0].size(); // Número de colunas em matrizA
int p = B[0].size(); // Número de colunas em matrizB

// Inicialize a matriz de resultado com zeros
vector<vector<int>> ans(m, vector<int>(p, 0));

// Realize a multiplicação das matrizes
for (int i = 0; i < m; i++) {
    for (int j = 0; j < p; j++) {
        for (int k = 0; k < n; k++) {
            ans[i][j] += A[i][k] * B[k][j];
        }
    }
}

return ans;
}
```

7.3.2 Fibonnaci por exponenciação binária de matrizes

```
vector<vector<int>> I = {{1,0},{0, 1}}; // matriz identidade
```

```
    // exponenciação binária de matrizes
    vector<vector<int>> matrixExp(vector<vector<int>> A, int exp){
        if (exp == 0) return I;
        if (exp%2 == 0){
            vector<vector<int>> B = matrixEXp(A, exp/2);
            return matMult(B, B);
        }
        return matMult(A, matrixExp(A, exp-1));
    }

    int fibonnaci(n){
        if (n <= 1) return n;
        vector<vector<int>> B = matrixExp(A, n-1)
        return B[1][1];
    }
```

7.4 Fatoração em números primos

A fatoração em números primos tranforma um número grande em um produto de primos. Por exemplo, 5733, a fatoração pode transformá-lo em 3.3.7.7.13; sendo que cada fator ocupa um índice no vector.

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

7.5 GCD e LCM

Algoritmo de Euclides (atenção para os limites de INT e LL):

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

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

8 Grafos

8.1 Lista de adjacências

A estrutura utilizada é tal que as adjacências são da forma g[u] = $\{w, v\}$. O peso está em primeiro para facilitar um eventual sort.

```
vector<vector<ii>> g(MAX);

// para adicionar aresta que conecta $u$ e $v$ com custo $w$ fazer:
g[u].push_back({w,v});
g[v].push_back({w,u});
```

8.2 DFS:

Chamar uma dfs para cada vértice, da forma:

```
for (int u = 0; u < n; u++)
    if (!vis[u]) dfs(u, u+1);
```

Complexidade : $\mathcal{O}(V + E)$

```
void dfs(int u, int val){
    vis[u] = val;
    for (auto &v : g[u])
        if (!vis[v.second]) dfs(v.second,val);

}
```

8.3 BFS

Salva as distâncias a partir do vértice de início s.
Complexidade : $\mathcal{O}(V + E)$

```
void bfs(int s){
    d[s] = 0;
    queue<int> q; q.push(s);
    while(!q.empty()){
        int u = q.front(); q.pop();
        for (auto &v : g[u])
            if (d[v.second] == INF){
```

```
                d[v.second] = d[u]+1;
                q.push(v.second);
            }
        }
    }
```

8.4 Ordenação Topológica

Atribui rótulos aos vértices de um grafo DAG de modo que para toda aresta (u, v), u.rótulo < v.rótulo. Utiliza uma DFS com pequena variação:

ATENÇÃO: esse código salva a ordem topológia reversa.

```
vector<int> ts; // topological sort

void dfs(int u, int val){
    if (vis[u]) return;
    vis[u] = val;
    for (auto &v : g[u])
        if (!vis[v.second]) dfs(v.second,val);

    ts.push_back(u);
}
```

8.5 Componentes Conexas

Código que conta a quantidade de componentes conexas e marca cada vértice com o valor de sua componente.

```
int ans = 0;
for (int u = 0; u < n; u++)
    if (!vis[i]) dfs(u, ++ans);
```

8.6 Checagem de grafo bipartido

Para checar se um grafo é bipartido, "colorimos"ele com uma BFS modificada.

```
bool bfs(){
    color[s] = 0;
    bool isBipartite = true;
```

```
        queue<int> q; q.push(s);
        while(!q.empty() && isBipartite){
            int u = q.front(); q.pop();
            for (auto &v : g[u]){
                if (color[v.second] == INF){
                    color[v.second] = 1-color[u];
                    q.push(v.second);
                }
                else if (color[v.second] == color[u]{
                    isBipartite = false; break;
                }
            }
        }
        return isBipartite;
    }
```

8.7 Árvores Geradoras Mínimas - MST

8.7.1 Kruskal

Utiliza MUF para unir as arestas e seus vértices. Complexidade: $\mathcal{O}(E \log V)$

```
    int kruskal(){
        int cost = 0, u, v, w;
        for (int i = 0; i < n; i++) {_p[i]=i; _rank[i]=0;}
        sort(edges.begin(), edges.end());
        for (auto &e : edges){
            u = e.second.first;
            v = e.second.second;
            w = e.first;
            if (_find(u) != _find(v)){
                cost += w;
                _union(u, v);
            }
        }
        return cost;
    }
```

8.7.2 Prim

Utiliza fila de prioridade e lista de adjacência para escolher novo vértice com aresta de menor peso. Complexidade: $\mathcal{O}(E \log V)$

```
vi taken(MAX);

int prim(){
    priority_queue<ii> pq;
    taken[0] = 1;
    for (auto &v : g[0])
        if (!taken[v.second]) pq.push({-v.first, -v.second});

    int cost = 0;
    while(!pq.empty()){
        ii front = pq.top(); pq.pop();
        int w = -front.first;
        int u = -front.second;
        if (!taken[u]) {
            cost += w;
            taken[u] = 1;
            for (auto &v : g[u])
                if (!taken[v.second]) pq.push({-v.first, -v.second});
        }
    }
    return cost;
}
```

8.8 Caminhos mínimos

8.8.1 Dijkstra

Complexidade: $\mathcal{O}(E \log V)$

```
void dijkstra(int s){
    int d, u, v;
    for (int i = 0; i < n; i++) dist[i] = INF;
    priority_queue<ii, vector<ii>, greater<ii>> pq;
    pq.push({0,s});
    while(!pq.empty()){
```

```
            d = pq.top().first;
            u = pq.top().second;
            pq.pop();
            if (d > dist[u]) continue;

            for (auto &x : adj[u]){
                d = x.first;
                v = x.second;
                if (dist[v] > dist[u] + d){
                    dist[v] = dist[u] + d;
                    pq.push({dist[v], v});
                }
            }
        }
    }
```

8.8.2 Belmann-Ford

Pode ser utilizado para detectar ciclos negativos. Complexidade: $\mathcal{O}(EV)$

```
    void belmann_ford(int s){
        for (int i = 0; i < V; i++) dist[i] = INF;
        dist[s] = 0;
        for (int i = 0; i < V; i++)
            for (int u = 0; u < V; u++)
                for (auto &v : adj[u])
                    dist[v.second] = min(dist[v.second], dist[u]+v.first);
    }
```

Para detectar ciclos negativos, executar mais uma varredura, de forma que o código fique da seguinte maneira:

```
    void belmann_ford(int s){
        for (int i = 0; i < V; i++) dist[i] = INF;
        dist[s] = 0;
        for (int i = 0; i < V; i++)
            for (int u = 0; u < V; u++)
                for (auto &v : adj[u])
                    dist[v.second] = min(dist[v.second], dist[u]+v.first);
    }
```

```
    int main(){
        ...

        bellmann_ford(s);

        bool hasNegativeCycle = false;
        for (int u = 0; u < V; u++)
            for (auto &v : adj[u])
                if (dist[v].second > dist[u]+v.first)
                    hasNegativeCycle = true;

        if (hasNegativeCycle) cout << "CYCLE" << endl;
        else cout << "NO CYCLE" << endl;
        ...
    }
```

8.8.3 Floyd Warshall

calcula as menores distâncias entre todos os vértices de um grafo.

```
void floyd_warshall(int graph[][N], int N) {
    int dist[N][N], i, j, k;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < N; k++)
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
}
```

9 Teoria dos Jogos

9.1 Jogo do Nim

1- Jogo do Nim: Dadas k pilhas (g1, g2, ..., gk), deve-se retirar uma quantidade de pedras de alguma pilha. O vencedor será sempre o jogador que possui a soma de Nim não nula em sua jogada, isto é, $g1 \oplus g2 \oplus ...gk! = 0$

2- Grundy Numbers & mex (mínimo excluído): o número de Grundy é utilizado para analisar o estado do jogo. Todo jogo imparcial possui um numero de Grundy. O número de Grundy é o MEX de todos os estados possíveis que podemos atingir a partir do estado atual, considerando as jogadas válidas.

3- Grundy numbers em jogos de Nim compostos (mais de uma pilha): o número de Grundy do jogo completo é dado pelo XOR bit a bit de todas as n pilhas.

$$S = g_1 \bigoplus g_2 \bigoplus \cdots \bigoplus g_n$$

4- Teorema de Sprague Grundy: se o Grundy number resultante for diferente de 0, o jogador da vez ganha; do contrário, o adversário ganha.

10 Programação Dinâmica

10.1 Fibonacci

Exemplo clássico de programação dinâmica, reduzindo a complexidade do Fibonacci para O(n)

```
int Fibonacci(vector<int> &MEMO, int n){
    if (MEMO[n] != -1)
        return MEMO[n];
    if (n == 1 || n==2){
        MEMO [n] = 1;
        return MEMO[n];
    }
    MEMO[n] = Fibonacci(MEMO, n-1) + Fibonacci(MEMO, n-2);
    return MEMO[n];
}

vector<int> MEMO(n+1, -1);
int result = Fibonacci(MEMO, n);
```

10.2 Problema da mochila

Selecionar um conjunto de itens (itens que possuem pesos e valores) de modo que o somatório dos valores seja máximo, respeitando a capacidade da mochila.

```
int mochila01 (int W, int N, vector<int>& peso, vector<int>& valor){
    int MEMO [N+1][W+1];
    for (int i=0; i<=N; i++)
        for (int j=0; j<=W; j++)
            MEMO[i][j] = 0;

    for (int i=1; i<=N; i++){
        for (int j=1; j<=W; j++){
            if (peso[i] > j)
                MEMO[i][j]=MEMO[i-1][j];
            else
                MEMO[i][j] = max(MEMO[i-1][j], MEMO[i-1][j-peso[i]]+valor[i]);
        }
    }
    return (MEMO[N][W]);
}
```

10.3 Longest Common Subsequence

Dada duas strings X e Y, calcula a maior sequência comum entre ambas.

```
//Essa função pode ser alterada para calcular a sequência de qualquer tipo.
//Para se descobrir essa subsequência, deve-se criar uma matriz auxiliar de setas.
int lcs (string X, string Y, int m, int n){
    vector<vector<int>> MEMO (m+1, vector<int>(n+1, 0));

    for (int i=0; i<=m; i++){
        for (int j=0; j<=n; j++){
            if (i == 0 || j==0)
                MEMO[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                MEMO[i][j] = MEMO[i-1][j-1] + 1;
            else
                MEMO[i][j] = max(MEMO[i-1][j], MEMO[i][j-1]);
        }
    }

    return (MEMO[m][n]);
}
```

11 Outros

11.1 MUF - Make Union Find

Algoritmo relacionado a conectividade em grafos e algoritmos de agrupamento. Ex: uva793.cpp

Utilizado no algoritmo de Kruskal para Árvores Geradoras Mínimas.

```
vi _p(MAX);
vi _rank(MAX);

int _find(int u){
    if (_p[u] == u) return u;
    return _p[u] = _find(_p[u]);
}

void _union(int u, int v){
    u = _find(u);
    v = _find(v);
    if (_rank[u] < _rank[v]) _p[u] = v;
    else{
        _p[v] = u;
        if (_rank[u] == _rank[v]) _rank[u]++;
    }
}
```

Antes de realizar as operações, utilizar as seguintes definições:

```
for (int i = 0; i < n; i++) {_p[i]=i; _rank[i]=0;}
```