

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	3
1.1	Template	3
1.2	Compilação	3
2	Vetores	4
2.1	Inicializacao	4
2.1.1	1D	4
2.1.2	2D	4
2.2	Difference Array	4
2.2.1	Difference Array 2D	4
2.3	Segment tree	5
2.3.1	SegTree Aditiva	5
2.3.2	SegTree Multiplicativa	6
2.3.3	SegTree Aditiva com Lazy Propagation	7
2.4	Fenwick Tree (BIT)	9
2.4.1	BIT aditiva	9
2.5	Bitset	10
3	Ordenação e Busca	11
3.1	Insertion Sort	11
3.2	Merge Sort	11
3.3	Binary Search	12
4	Strings	13
4.1	Comandos básicos	13
4.2	Substrings	13
4.3	char to int	13
4.4	int to string	13
5	Teoria dos Números	15
5.1	Exponenciação binária	15
5.1.1	Exponenciação binária modular	15
5.2	Aritmética modular	16
5.3	Recorrências lineares	16
5.3.1	Multiplicação de matrizes	17
5.3.2	Fibonnaci por exponenciação binária de matrizes	17
5.4	Fatoração em números primos	18
5.5	GCD e LCM	18

6	Grafos	20
6.1	Map adjacência	20
7	Teoria dos Jogos	22
7.1	Jogo do Nim	22
8	Programação Dinâmica	24
8.1	Fibonacci	24
8.2	Problema da mochila	24
8.3	Longest Common Subsequence	25
9	Outros	26
9.1	MUF - Make Union Find	26

1 Template e Compilação

1.1 Template

1. Diretório C++ que contém quase todos os comandos necessários

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

2. Comando para deixar operações do C++ como cin/cout mais rápidas.

```
cin.tie(0);
ios_base::sync_with_stdio(0);
```

3. Template:

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

int main(){
    cin.tie(0);
    ios_base::sync_with_stdio(0);

    return 0;
}
```

1.2 Compilação

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

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

3. Se Linux: executar o comando "./a.out < tmp.in", onde tmp.in é o arquivo com as entradas

2 Vetores

2.1 Inicializacao

2.1.1 1D

- `memset(vetor, valor, n)`

Coloca valor nas primeiras n posições de v .

2.1.2 2D

- Tamanhos dados por input:

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

2.2 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;
}
```

2.2.1 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;
    }
}
```

2.3 Segment tree

Extensivamente utilizada para fazer operações em intervalos de vetores, tanto atualização como consultas são 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)`

2.3.1 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);
}

```

2.3.2 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);
}
```

2.3.3 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);
    }
```

2.4 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.

2.4.1 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;
}
```

2.5 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.*/
```

3 Ordenação e Busca

3.1 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;
    }
}
```

3.2 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);
}
```

3.3 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.

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);
```

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

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

4 Strings

4.1 Comandos básicos

4.2 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);
```

4.3 char to int

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

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

4.4 int to string

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

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

s

5 Teoria dos Números

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

5.1 Exponenciação binária

Queremos calcular um número a^x , podemos fazer isso em complexidade $O(\log x)$, 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;
}
```

5.1.1 Exponenciação binária modular

Podemos querer realizar operações 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;  
}
```

5.2 Aritmética modular

Propriedades:

- $(a + b + c) \bmod(n) = ((a + b) \bmod(n) + c) \bmod(n)$
- $(a - b) \bmod(n)$

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

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

b^{-1} pode ser lido como o inverso multiplicativo de b , ou seja, temos que $b \cdot \frac{1}{b} = 1 \bmod(n)$. 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. $a^p \equiv a \bmod(p)$
2. $a^{p-1} \equiv 1 \bmod(p)$
3. Logo: $a \cdot a^{p-2} \equiv 1 \bmod(p)$, portanto: $a^{p-2} \equiv a^{-1}$
4. Para calcular a^{p-2} , utilizar exponenciação binária modular.

5.3 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: Fibonacci:

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

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

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

bastando tomar

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

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

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

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

5.3.1 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;
}
```

5.3.2 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];
}
```

5.4 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;
}
```

5.5 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);  
}
```

6 Grafos

6.1 Map adjacência

Implementação de uma lista de adjacência (sem classe/struct)

```
map<int, vector<int>> adj_map;

void addEdge (map<int, vector<int>> adj_map, int u, int v){
    adj_map[u].push_back(v);
}
```

2- DFS: busca em profundidade de grafos (depth-first search)

```
void visit (int u, map<int, vector<int>> &adj_map, vector<bool> &visited){
    visited[u] = true;

    vector<int>::iterator i;
    for (i = adj_map[u].begin(); i != adj_map[u].end(); ++i)
        if (!visited[*i])
            visit(*i, adj_map, visited);
}

void DFS(map<int, vector<int>> &adj_map, int V){
    vector<bool> visited(V, false);
    for (int u=0; u<V; u++)
        if (visited[u]==false)
            visit(u, adj_map, visited);
}
```

3- BFS: busca em largura de grafos (breadth-first search)

4- Ordenação Topológica: atribui rótulos aos vértices de um grafo DAG de modo que para toda aresta (u, v) , $u.\text{rótulo} < v.\text{rótulo}$.

```
void BFS (int u, map<int, vector<int>> &adj_map, vector<bool> &visited, vector<int>
    visited[u] = true;
    vector<int>::iterator i;
    for (i=adj_map[u].begin(); i!=adj_map[u].end(); ++i)
```

```
        if (!visited[*i])
            BFS(*i, adj_map, visited, rot_default);
    rot_default[u] = rotulo++;
}

void ord_topologica (map<int, vector<int>> &adj_map, int V){
    vector<int> rot_default(V, 0);
    vector<bool> visited (V, false);
    for (int u=0; u<V; u++)
        if (visited[u] == false)
            BFS(u, adj_map, visited, rot_default);
}
```

5- Componentes Fortemente Conexas: irá calcular a quantidade de componentes conexas em um grafo DAG.

4- 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];
}
```

7 Teoria dos Jogos

7.1 Jogo do Nim

1- Jogo do Nim: Dadas k pilhas (g_1, g_2, \dots, g_k), 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, $g_1 \oplus g_2 \oplus \dots \oplus g_k \neq 0$

2- Grundy Numbers & mex (mínimo excluído): o número de Grundy é utilizado para analisar o estado do jogo. Todo jogo imparcial possui um número 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.

$$S = g_1 \oplus g_2 \oplus \dots \oplus g_n$$

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

S

8 Programação Dinâmica

8.1 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);
```

8.2 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]);
}
```

8.3 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]);  
}
```

9 Outros

9.1 MUF - Make Union Find

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

```
#define MAX 11234567
int n, _p[MAX], _rank[MAX];

// MUF - Make Union Find

int _find(int u) { return _p[u] == u ? u : (_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]++;
    }
}
```