



# Capivalivro

Lucas Joviniano

07/10/2022

# Table of Contents

Introdução .....	1
Troubleshoot .....	1
C++ e STL .....	3
template.cpp .....	3
Complex .....	3
Pair .....	4
List .....	4
Vector .....	4
Deque .....	5
Queue .....	5
Stack .....	6
Map .....	6
Set .....	6
Bitset .....	7
String .....	7
Algorithm .....	7
Estruturas de Dados .....	10
Fenwick Tree .....	10
Segment Tree .....	10
Processamento de Strings .....	12
Knuth-Morris-Pratt .....	12
Distância de Levenshtein .....	13
Combinatória .....	14
Regra da Soma .....	14
Regra do Produto .....	14
Inclusão-Exclusão .....	14
Combinação .....	14
Arranjo .....	14
Multiset .....	14
Coeficiente Binomial .....	14
Fibonacci .....	15
Catalan .....	15
Teoria dos Números .....	16
Teste de Primalidade .....	16
Fatoração .....	17
Primos Relativos .....	17
MDC e MMC .....	17
Busca Ciclos (lebre e tartaruga) .....	18

Geometria .....	19
Vetores .....	19
Misc .....	20
Busca Binária .....	20
Busca Ternária .....	21

# Introdução

## Troubleshoot

### *Antes de Enviar*

- Escreva testes simples se os fornecidos não são suficientes
- O tempo limite é muito curto? Se sim, escreva um caso de teste de tamanho máximo.
- O uso de memória está no limite?
- Algo pode dar overflow?
- Tenha certeza de que está enviando o arquivo certo

### *Wrong Answer*

- Imprima sua solução! Imprima também os resultados intermediários.
- Limpou todas as estruturas de dados entre os casos de teste?
- Seu algoritmo trata toda a entrada?
- Leia o problema inteiro de novo.
- Todos os casos especiais são tratados?
- Você entendeu o problema corretamente?
- Alguma variável não inicializada?
- Algum overflow?
- Não está confundindo M e N, i e j, etc.?
- Tem certeza que seu algoritmo funciona?
- Quais os casos especiais que você pode não ter pensado?
- Tem certeza que as funções da STL funcionam como você espera?
- Adicione alguns asserts, talvez tente enviar novamente.
- Crie alguns casos de teste para passar pelo algoritmo.
- Explique seu algoritmo para um colega de time.
- Peça para o colega de time olhar seu código.
- Vai dar uma volta
- Está imprimindo corretamente?
- Reescreva toda sua solução ou peça para alguém fazer isso.

### *Runtime Error*

- Testou todos os casos especiais localmente?
- Alguma variável não inicializada?
- Escrevendo ou lendo fora do tamanho de algum vetor?
- Algum assert que possa falhar?

- Alguma possível divisão por 0? (Ou mod 0)
- Alguma possível recursão infinita
- Ponteiros ou iteradores invalidados?
- Usando muita memória?
- Debug enviando novamente

#### *Time Limit Exceeded*

- Algum possível loop infinito?
- Qual a complexidade do seu algoritmo?
- Copiando muitos dados desnecessariamente? (Use referências)
- A entrada ou a saída não são muito grandes? (Considere usar scanf)
- Evite vector e map
- O que seus colegas acham do algoritmo?

#### *Memory Limit Exceeded*

- Qual o máximo de memória que seu algoritmo deveria precisar?
- Limpou todas as estruturas de dados entre casos de teste?

# C++ e STL

## template.cpp

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

using namespace std;
#define MOD 1000000007
#define EPS 1e-9
#define pb push_back
#define pf push_front
#define fi first
#define se second
#define mp make_pair
#define all(x) x.begin(), x.end()

template<typename A, typename B>
ostream &operator<<(ostream &s, const pair <A, B> &p) {
    return s << "(" << p.fi << ", " << p.se << ")";
}

template<typename T>
ostream &operator<<(ostream &s, const vector <T> &v) {
    s << "[";
    for (auto it: v) s << it << " ";
    s << "]";
    return s;
}

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin.exceptions(cin.failbit);
}
```

## Complex

```
#include <complex>

complex<double> point(10.0, 1.0);

std::real(point);    // 10
std::imag(point);    // 1
std::abs(point);     // 10.0499
std::arg(point);     // 0.0996687
std::norm(point);    // 101
std::conj(point);    // (10, -1)
```

```
std::polar(2.0, 0.5); // (1.75517,0.958851)
```

## Pair

```
#include <utility>

pair<T, U> p;
auto m = make_pair(13, "Lula");
```

## List

```
#include <list>

list<int> first;           // Lista vazia
list<int> second(4, 100); // Lista com 4 elementos iguais a 100
list<int> third(second.begin(), second.end()); // Itera por second
list<int> fourth(third); // Copia third

vector<int> v{16, 2, 77, 29, 9};
list<int> fifth(v.begin(), v.begin() + 3); // [16, 2, 77, 29]

fifth.begin(); // 16
fifth.rbegin(); // 29
fifth.size(); // 4 - O(n)
fifth.empty(); // false
fifth.clear(); // []
fifth.front(); // 16
fifth.back(); // 29
fifth.push_back(9); // [16, 2, 77, 29, 9] - O(1)
fifth.push_front(1); // [1, 16, 2, 77, 29, 9] - O(n)
fifth.pop_back(); // [1, 16, 2, 77, 29] - O(1)
fifth.pop_front(); // [16, 2, 77, 29] - O(n)
fifth.insert(next(fifth.begin(), 3), 3); // [16, 3, 2, 77, 29]
fifth.erase(fifth.begin()); // [2, 77, 29]
fifth.erase(fifth.begin(), fifth.end()); // []
```

## Vector

```
#include <vector>

vector<int> first;           // Vetor vazio
vector<int> second(4, 100); // Vetor com 4 elementos iguais a 100
vector<int> third(second.begin(), second.end()); // Itera por second
vector<int> fourth(third); // Copia third
```

```

int v[] = {16, 2, 77, 29, 9};
vector<int> fifth(v, v + sizeof(v) / sizeof(int)); // [16, 2, 77, 29]

fifth.begin(); // 16
fifth.rbegin(); // 29
fifth.size(); // 4
fifth.empty(); // false
fifth.clear(); []
fifth.reserve(10); // Aloca uma capacidade mínima para o vetor
fifth.front(); // 16
fifth.back(); // 29
fifth.push_back(9); // [16, 2, 77, 29, 9]
fifth.pop_back(); // [16, 2, 77, 29]
fifth.erase(fifth.begin()); // [2, 77, 29, 9]
fifth.erase(fifth.begin(), fifth.end()); // []

```

## Deque

```

#include <deque>

deque<int> first; // Deque vazio
deque<int> second(4, 100); // Lista com 4 elementos iguais a 100
deque<int> third(second.begin(), second.end()); // Itera por second
deque<int> fourth(third); // Copia third

vector<int> v{16, 2, 77, 29, 9};
deque<int> fifth(v.begin(), v.begin() + 3); // [16, 2, 77, 29]

fifth.begin(); // 16
fifth.rbegin(); // 29
fifth.size(); // 4
fifth.empty(); // false
fifth.clear(); // []
fifth.front(); // 16
fifth.back(); // 29
fifth.push_back(9); // [16, 2, 77, 29, 9]
fifth.push_front(1); // [1, 16, 2, 77, 29, 9]
fifth.pop_back(); // [1, 16, 2, 77, 29]
fifth.pop_front(); // [16, 2, 77, 29]
fifth.insert(next(fifth.begin(), 3), 3); // [16, 3, 2, 77, 29]
fifth.erase(fifth.begin()); // [2, 77, 29]
fifth.erase(fifth.begin(), fifth.end()); // []

```

## Queue

```

#include <queue>

```



```

deque<int> d{10, 12, 13};
queue<int> q(d);

q.back(); // 13
q.empty(); // false
q.front(); // 10
q.pop(); // [12, 13]
q.push(30); // [12, 13, 30]
q.size(); // 3

```

## Stack

```

#include <stack>

deque<int> d{10, 12, 13};
stack<int> s(d);

q.empty(); // false
q.top(); // 13
q.pop(); // [10, 12]
q.push(30); // [10, 12, 30]
q.size(); // 3

```

## Map

Há também a variante `unordered_map`, que usa Hash Table (é mais rápido)

```

#include <map>
#include <string>

map<string, int> m;
m['a'] = 10;
m['b'] = 30;
m['c'] = 50;
m['d'] = 70;

m.begin(); // (a, 10)
m.empty(); // false
m.size(); // 4;
m.count('f'); // 0
m.find('c'); // iterador para c

```

## Set

Há também a variante `unordered_set`, que usa Hash Table (é mais rápido)

```
#include <set>

set<int> s{10, 12, 13};

s.begin(); // 10
m.empty(); // false
m.size(); // 3;
m.insert(30); // [10, 12, 13, 30]
m.find(13); // iterador para 13
```

## Bitset

Mesma coisa que um vetor de booleanos, porém com tamanho fixo.

```
#include <bitset>

bitset<4> bs; // 0000
bs.set(); // 1111
bs.to_string(); // "1111"
bs.to_ulong(); // 15
bs.to_ullong(); // 15
bs.reset(); // 0000
bs.flip(2); // 0100
bs.flip(); // 1011
```

## String

```
#include <string>

strings s = "Capivalivro";
s.push_back('m'); // "Capivalivrom"
s.erase(4, s.size() - 5); // "Capim"
s.find("pi"); // 2
s.substr(1, 4); // "api"
```

## Algorithm

**any\_of(beg, end, eval):** Retorna **true** se algum dos elementos são avaliados como **true** pela função eval.

**all\_of(beg, end, eval):** Retorna **true** se todos os elementos são avaliados como **true** pela função eval.

**none\_of(beg, end, eval):** Retorna **true** se nenhum elemento é avaliado como **true** pela função eval.

**for\_each(beg, end, proc):** Executa a função `void proc(T a)` para cada elemento.

**count(beg, end, v):** Conta quantos elementos são iguais a `v`;

**count\_if(beg, end, eval):** Conta quantos elementos são avaliados como `true` pela função `eval`.

**fill(beg, end, v):** Atribui `v` a todos os elementos

**generate(beg, end, acc):** Atribui a cada posição um valor retornado por `T acc()`.

**remove(beg, end, v):** Remove todos os elementos iguais a `v`. Retorna um ponteiro para o novo final.

**remove\_if(beg, end, eval):** Remove todos elementos em que `eval` retorna `true`.

**replace(beg, end, u, v):** Substitui por `v` todos os elementos iguais a `u`

**replace\_if(beg, end, eval, v):** Substitui por `v` todos os elementos onde `eval` retorna `true`

**swap(a, b):** Troca duas coleções.

**reverse(beg, end):** Inverte a ordem

**rotate(beg, beg + i, end):** Rotaciona a coleção de forma em que o `i`-ésimo elemento fique em primeiro.

**random\_shuffle(beg, end):** Aplica permutação aleatória

**unique(beg, end):** Remove todos os elementos duplicados consecutivos. Retorna um ponteiro para o novo fim.

**partition(beg, end, eval):** Reordena de forma que todos os elementos avaliados como `true` fiquem antes dos avaliados como `false`.

**stable\_partition(beg, end, eval):** O mesmo de `partition`, mas mantém as ordens relativas.

**is\_sorted(beg, end):** Verifica se o intervalo está ordenado.

**sort(beg, end):** Ordena o intervalo.

**stable\_sort(beg, end):** Ordena o intervalo mantendo as ordens relativas.

**nth\_element(beg, beg+n, beg):** Realiza a partição do intervalo de modo que o `n`-ésimo fique no lugar, com os menores antes e os maiores depois.

**lower\_bound(beg, end, v):** Retorna um iterador ao primeiro elemento maior ou igual a `v`.

**upper\_bound(beg, end, v):** Retorna um iterados ao primeiro elemento maior que `v`.

**binary\_search(beg, end, v):** Retorna se `v` está no intervalo.

**make\_heap(beg, end):** Transforma o intervalo em um heap de máximo.

**push\_heap(beg, end, v):** Adiciona ao heap.

**pop\_heap(beg, end):** Remove o maior elemento do heap.

**sort\_heap(beg, end):** Ordena o heap de forma crescente.

**max(a, b):** Retorna o maior entre a e b.

**min(a, b):** Retorna o menor entre a e b.

**max\_element(beg, end):** Retorna um ponteiro para o maior elemento no intervalo.

**min\_element(beg, end):** Retorna um ponteiro para o menor elemento no intervalo.

**next\_permutation(beg, end):** Reordena para a próxima permutação lexicográfica.

**prev\_permutation(beg, end):** Reordena para a permutação lexicográfica anterior.

**accumulate(beg, end, acc):** Soma todos os elementos a partir do valor inicial **acc**.

**partial\_sum(beg, end):** Transforma o intervalo em um array de somas parciais.

# Estruturas de Dados

## Fenwick Tree

**Descrição:** Calcula soma de prefixos em um array.

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

// ATENÇÃO: Indexa do 1
class FenwickTree {
private:
    vector<int> ft;
public:
    FenwickTree(int n) {
        ft.assign(n + 1, 0);
    }

    int rsq(int i) { // Retorna RSQ(1, i)
        int sum = neutral;
        for (; i; i -= (i & -i)) sum = comp(sum, ft[i]);
        return sum;
    }

    int rsq(int i, int j) {
        return rsq(j) - rsq(i - 1);
    }

    void update(int i, int v) {
        for (; i < (int)ft.size(); i += (i & -i))
            ft[i] = comp(v, ft[i]);
    }
};
```

## Segment Tree

**Descrição:** Permite responder queries em intervalos de um array de forma eficiente, incluindo:

- Achar a soma de elementos consecutivos
- Achar o menor elemento em um range

```
const int neutral = 0;
#define comp(a, b) ((a)+(b))

class SegmentTree {
    vector<int> a;
    int n;
```

```

public:
    SegmentTree(int *st, int *en) {
        int sz = int(en - st);

        for (n = 1; n < sz; n <= 1);
        a.assign(n < 1, neutral);
        for (int i = 0; i < sz; i++) a[i + n] = st[i];
        for (int i = n + sz - 1; i > 1; i--) {
            a[i >> 1] = comp(a[i >> 1], a[i]);
        }
    }

    void update(int i, int x) {
        a[i += n] = x;
        for (i >>= 1; i; i >>= 1) {
            a[i] = comp(a[i << 1], a[1 + (i << 1)]);
        }
    }

    int query(int l, int r) {
        int ans = neutral;
        for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1) ans = comp(ans, a[l++]);
            if (r & 1) ans = comp(ans, a[--r]);
        }
        return ans;
    }
};

```

# Processamento de Strings

## Knuth-Morris-Pratt

Inicializar a classe com a string a ser procurada.

### Aplicações

- Encontrar uma substring em uma string
- Contar a quantidade de ocorrências de cada prefixo
- Número de substrings diferentes em uma string
- Comprimir uma string

```
class KMP {
    string p;
    vector<int> v;
    int m;

public:
    KMP(const char *_p) : p(_p) {
        m = p.size();
        v.assign(m + 1, -1);
        for (int i = 0, j = -1; i < m; i++) {
            while (j >= 0 && p[i] != p[j])
                j = v[j];
            v[++i] = ++j;
        }
    }

    vector<int> match(const char* t) {
        vector<int> ans;

        for (int i = 0, j = 0, n = strlen(t); i < n; i++) {
            while (j >= 0 && t[i] != p[j])
                j = v[j];
            i++;
            j++;
            if (j == m) {
                ans.push_back(i - j);
                j = v[j];
            }
        }

        return ans;
    }
};
```

# Distância de Levenshtein

Dadas duas strings, descobrir quantas operações são necessárias para que elas se tornem iguais. As operações podem ser: remover caractere, adicionar caractere e trocar caractere.

```
int min(int x, int y, int z) { return min(min(x, y), z); }

int editDist(string str1, string str2, int m, int n)
{
    if (m == 0)
        return n;

    if (n == 0)
        return m;

    if (str1[m - 1] == str2[n - 1])
        return editDist(str1, str2, m - 1, n - 1);

    return 1
        + min(editDist(str1, str2, m, n - 1), // Insert
              editDist(str1, str2, m - 1, n), // Remove
              editDist(str1, str2, m - 1,
                      n - 1) // Replace
        );
}
```



# Combinatória

## Regra da Soma

Se A tem  $|A|$  elementos e B tem  $|B|$  elementos, então existem  $|A| + |B|$  maneiras de escolher elementos de A ou de B. Assumindo que os elementos são distintos.

## Regra do Produto

Se A tem  $|A|$  elementos e B tem  $|B|$  elementos, então existem  $|A| \times |B|$  maneiras de escolher um elemento de A e um de B. Assumindo que os elementos são independentes.

## Inclusão-Exclusão

Para 2:  $|A \cup B| = |A| + |B| - |A \cap B|$

Para 3:  $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

## Combinação

Existem  $\frac{n!}{k!(n-k)!}$  maneiras de escolher  $k$  pessoas de um grupo de  $n$

## Arranjo

Existem  $\frac{n!}{(n-k)!}$  maneiras de escolher  $k$  pessoas de um grupo de  $n$ , se importa a ordem de escolha.

## Multiset

Existem  $\binom{n+k-1}{k}$  maneiras de escolher  $k$  pessoas de um grupo de  $n$ , se não importa a ordem e pode repetir.

## Coeficiente Binomial

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ : Número de combinações de  $k$  elementos de um conjunto  $n$ .

```
long binomial_coefficient(int n, int k) {
    int C[k + 1];
    memset(C, 0, sizeof(C));

    C[0] = 1;

    for (int i = 1; i <= n; i++) {
        for (int j = min(i, k); j > 0; j--) {
            C[j] = C[j] + C[j - 1];
        }
    }
}
```

```
}  
  
return C[k];  
}
```

## Fibonacci

Definida pela recorrência  $F_n = F_{n-1} + F_{n-2}$ , com  $F_0 = 0$  e  $F_1 = 1$

Fórmula Fechada:

$$F_n = \frac{1}{\sqrt{5}} \left( \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right)$$

## Catalan

Definida pela recorrência  $C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$

Fórmula fechada:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

# Teoria dos Números

## Teste de Primalidade

- Pré-processamento: Gerar primos  $\leq 10.000.000$  pelo Crivo de Eratóstenes
- Verificação: Se  $n \leq 10.000.000$  usar marcação do crivo.
- Senão testar divisibilidade com os primos do crivo



Só funciona se  $n \leq 100.000.000.000.000$

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

long long _tam_crivo;
bitset<10000010> bs;
vector<int> primos;

void crivo(long long limite) {
    _tam_crivo = limite + 1;
    bs.reset();
    bs.flip();
    bs.set(0, false);
    bs.set(1, false);
    for (long long i = 2; i <= _tam_crivo; i++) {
        if (bs.test((size_t)i)) {
            for (long long j = i * i; j <= _tam_crivo; j += i)
                bs.set((size_t)j, false);
            primos.push_back((int)i);
        }
    }
}

// Chamar na função main

bool is_prime(long long n) {
    if (n < _tam_crivo) {
        return bs.test(n);
    }

    for (int i = 0; i < primos.size(); i++) {
        if (n % primos[i] == 0) {
            return false;
        }
    }
    // Só funciona se n <= que o último primo do vetor primos

    return true;
}
```

# Fatoração

```
vector<int> primeFactors(int n) {
    vector<int> factors;

    int pf_idx = 0, pf = primos[pf_idx]; // Primos gerados pelo crivo

    while (n != 1 && (pf * pf <= n)) {
        while (n % pf == 0) {
            n /= pf;
            factors.push_back(pf);
        }
        pf = primos[++pf_idx];
    }

    if (n != 1)
        factors.push_back(n);
    return factors;
}
```

## Número de Divisores

Se  $N = a^i \times b^j \times \dots \times c^k$  é a fatora  o, ent  o  $N$  tem  $(i + 1) \times (j + 1) \times \dots \times (k + 1)$  divisores

## Primos Relativos

Quantos n  meros Unknown charactern s  o relativamente primos com  $n$

```
int euler_phi(int n) {
    vector<int> factors = primeFactors(n);
    auto new_end = unique(factors.begin(), factors.end());

    int result = n;

    for (auto it = factors.begin(); it != new_end; it++) {
        result = result - result / *it;
    }

    return result;
}
```

## MDC e MMC

```
long long mdc(long long a, long long b) {
    return (b == 0 ? a : mdc(b, a % b));
}
```

```
long long mmc(long long a, long long b) {  
    return (a * (b / mdc(a, b)));  
}
```

## Busca Ciclos (lebre e tartaruga)

```
pair<int, int> floyd_cycle_finding(function<int(int)> f, int x0) {  
    int tart = f(x0), lebr = f(f(x0));  
    while (lebr != tart) {  
        tart = f(tart);  
        lebr = f(f(lebr));  
    }  
  
    int mu = 0;  
    lebr = tart;  
    tart = x0;  
  
    while (lebr != tart) {  
        tart = f(tart);  
        lebr = f(lebr);  
        mu++;  
    }  
  
    int lamb = 1;  
    lebr = f(tart);  
  
    while (lebr != tart) {  
        lebr = f(lebr);  
        lamb++;  
    }  
  
    return make_pair(mu, lamb);  
}
```

# Geometria

## Vetores

- Dados dois pontos A e B, o vetor  $\vec{V} = \vec{AB}$  é dado por  $B - A$ .

Sejam  $\vec{U}$ ,  $\vec{V}$  e  $\vec{W}$  três vetores e  $\alpha$ ,  $\beta$  escalares.

- $\vec{U} + \vec{V} = \vec{V} + \vec{U}$
- $(\vec{U} + \vec{V}) + \vec{W} = \vec{U} + (\vec{V} + \vec{W})$
- $\alpha(\beta\vec{U}) = (\alpha\beta)\vec{U}$
- $\alpha(\vec{U} + \vec{V}) = \alpha\vec{U} + \alpha\vec{V}$
- $(\alpha + \beta)\vec{U} = \alpha\vec{U} + \beta\vec{U}$

# Misc

## Busca Binária

```
// predicado a ser usado...
bool p(int x) {
    return x >=
        4; // exemplo para o predicado [false, false, false, false, true, true]
}

// considera o intervalo fechado [lo,hi]
// false false true true --> retorna 2
int buscaBinariaMenorTrue(int lo, int hi) {
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2; // cuidado aqui!

        // de mid para a frente é true --> podemos descartar [mid+1, hi]
        if (p(mid))
            hi = mid;
        else
            lo = mid + 1; // até mid é false --> podemos descartar [lo,mid]
    }
    if (!p(lo)) {
        return -1; // erro...é falso para tudo!
    }
    return lo;
}

// considera o intervalo fechado [lo,hi]
// false false true true --> retorna 1
int buscaBinariaMaiorFalse(int lo, int hi) {
    while (lo < hi) {
        // lo + (hi-lo)/2 falharia para o caso de false true (loop infinito) !!!!!
        // ATENÇÃO! sempre teste para o caso de intervalos com dois elementos!
        int mid = lo + (hi - lo + 1) / 2; // cuidado aqui!

        // de mid para a frente é true --> podemos descartar [mid+1, hi]
        if (p(mid))
            hi = mid - 1;
        else
            lo = mid; // até mid é false --> podemos descartar [lo,mid]
    }
    if (p(lo)) {
        return -1; // erro...é true para tudo!
    }
    return lo;
}

// Em um intervalo contínuo, temos varios false, false, true, true
```

```
// retorna um double proximo da região que divide os false dos true.
double buscaBinariaContinua(double lo, double hi) {
    // mudar! podemos executar X iteracoes, parar quando distancia
    // entre hi e lo for pequena, etc...
    for (int i = 0; i < 100; i++) {
        double mid = lo + (hi - lo) / 2;
        if (p(mid))
            hi = mid;
        else
            lo = mid;
    }

    return lo;
}
```

## Busca Ternária

```
double ternary_search(double l, double r) {
    double eps = 1e-9; // limite para comparação de floats
    // podemos, ao invés de comparar por diferença, usar uma quantidade fixa
    // 200 a 300 deve ser o suficiente
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1);
        double f2 = f(m2);
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); // máximo de f(x) em [l, r]
}
```