

# Busca e Ordenação

## Algoritmos de Busca

---

Prof. Edson Alves - UnB/FGA

2018

1. Busca Sequencial
2. Busca Binária
3. Busca Ternária

# Busca Sequencial

---

# Algoritmos de busca

- Um algoritmo de busca consiste em uma função que identifica se um elemento  $x$  pertence ou não a um conjunto de elementos  $S$
- A função pode retornar um valor booleano (verdadeiro ou falso), caso o elemento pertença ou não ao conjunto  $S$
- Outra alternativa é retornar a posição (índice) do elemento no conjunto, caso este faça parte do mesmo, ou um valor sentinela, indicando que o elemento não pertence ao conjunto
- Caso o conjunto  $S$  seja um vetor, o algoritmo de busca mais simples é a busca sequencial, onde todos os elementos do vetor são comparados com o elemento que se deseja encontrar
- A ordem de complexidade do algoritmo é  $O(N)$ , onde  $N$  é número de elementos do vetor
- Embora existam algoritmos mais eficientes, este algoritmo funciona independentemente da ordenação dos elementos do vetor

# Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 0

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 1

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 2

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 3

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----



## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 4

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 5

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

# Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 6

Valor do retorno: -1

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de busca sequencial

Elemento a ser localizado: 34

Índice do elemento a ser comparado: 7

Valor do retorno: 7

20	35	14	95	68	71	9	34	46
----	----	----	----	----	----	---	----	----

## Exemplo de uso de busca sequencial

```
1 #include <stdio.h>
2
3 int i;
4
5 void imprimir_edicoes(const int *edicoes, int N) {
6     const char msg[][16] = { " ", " ", "\n" };
7
8     for (i = 0; i < N; i++)
9         printf("%d%s", edicoes[i], msg[i + 1 == N]);
10 }
11
12 int buscar_edicao(int edicao, const int *edicoes, int N) {
13     for (i = 0; i < N; i++)
14         if (edicoes[i] == edicao)
15             return 1;
16
17     return 0;
18 }
19
20 int main() {
21     int edicoes[] = {23, 44, 35, 10, 23, 99, 10, 23}, N = 8, edicao;
```

## Exemplo de uso de busca sequencial

```
23  do {
24      printf("\nColecao disponivel: ");
25      imprimir_edicoes(edicoes, N);
26
27      printf("Insira o numero da edicao a ser localizada: ");
28
29      if (scanf("%d", &edicao) != 1 || edicao < 0)
30          break;
31
32      if (buscar_edicao(edicao, edicoes, N))
33          printf("Edicao %d encontrada\n", edicao);
34      else
35          printf("A edicao %d nao faz parte da colecao\n", edicao);
36  } while (1);
37
38  printf("Finalizando o programa...\n");
39
40  return 0;
41 }
```

# Busca sequencial em C++

- A biblioteca `algorithm` do C++ contém uma implementação da busca sequencial
- A função `find()` recebe dois iteradores  $a$  e  $b$ , e um valor  $x$ , a ser procurado
- Caso  $x$  se encontre dentre os elementos que estão no intervalo  $[a, b)$ , é retornado um iterador para a primeira ocorrência de  $x$
- Caso  $x$  não esteja no intervalo, é retornado o valor  $b$
- A sintaxe da função `find()` é  
`InputIterator find(InputIterator first, InputIterator last, const T& val);`
- Esta função pode ser usada em qualquer contêiner que tenha iteradores que suportem a operação de incremento e que armazenem qualquer tipo que suporte o operador de comparação `==`

## Exemplo de uso da função find()

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
5 using namespace std;
6
7 int main()
8 {
9     const vector<int> ps { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
10    int p = 21;
11
12    auto it = find(ps.begin(), ps.end(), p);
13
14    if (it != ps.end())
15        cout << p << " encontrado na posicao " << it - ps.begin() << "\n";
16    else
17        cout << p << " nao encontrado\n";
18
19    return 0;
20 }
```



# Travessias e Filtros

- O processo de se visitar cada um dos elementos contidos em um contêiner é denominado travessia
- A busca sequencial usa uma travessia para confrontar cada um dos elementos do contêiner contra o valor que se deseja localizar
- Um padrão comum associado à travessia é o de se escolher um ou mais elementos do contêiner, de acordo com um predicato  $P$
- Um predicato  $P$  é uma função que recebe, dentre seus parâmetros, um elemento  $e$  do tipo  $T$  e retorna ou verdadeiro ou falso
- Este padrão recebe o nome de filtro
- Uma busca sequencial é um filtro que seleciona um (ou mais) elemento do contêiner com o predicado  $P$  definido como

```
bool P(const T& e, const T& x) { return e == x; }
```

# Filtros em C++

- A biblioteca `algorithm` do C++ contém uma implementação genérica de filtros
- A função `copy_if()` recebe um par de iteradores  $a$  e  $b$ , que definem um intervalo  $[a, b)$ ; um iterador de saída  $s$ , onde serão escritos os elementos que atendem o filtro; e um predicato  $P$  unário (que aceita um único parâmetro do tipo  $T$ )
- Todos os elementos  $e$  tais que  $P(e)$  é verdadeiro serão copiados no iterador de saída  $s$
- A sintaxe da função `copy_if()` é  

```
OutputIterator copy_if(InputIterator first, InputIterator last,  
                      OutputIterator result, UnaryPredicate pred);
```
- A função `back_inserter()`, da biblioteca `iterator`, gera um iterador de saída para o contêiner passado como parâmetro
- O contêiner em questão deve ter suporte para a função `push_back()`

# Exemplo de uso de filtro

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 const string vowels { "aeiou" };
7
8 int main()
9 {
10     string message { "Exemplo de busca sequencial" }, res;
11
12     copy_if(message.begin(), message.end(), back_inserter(res),
13             [](char c) { return vowels.find(c) != string::npos; });
14
15     if (res.empty())
16         cout << "Nenhuma vogal encontrada\n";
17     else
18         cout << res.size() << " vogais encontradas: " << res << "\n";
19
20     return 0;
21 }
```

- Outro padrão associado à travessia é a transformação
- Uma transformação visita cada um dos elementos  $x$  de  $S$ , e o substitui pelo resultado da transformação  $T(x)$
- A biblioteca `algorithm` do C++ implementa transformações através da função `transform()`
- A sintaxe da função `transform()` é  

```
OutputIterator transform(InputIterator first, InputIterator last,  
                        OutputIterator result, UnaryOperation op);
```
- Há também uma versão da função `transform()` que aceita uma operação binária

## Exemplo de uso de transformações

```
1 #include <iostream>
2 #include <algorithm>
3
4 using namespace std;
5
6 int main()
7 {
8     int xs[] { -1, 0, 4 }, ys[] { 2, 3, -3 }, zs[3];
9
10    transform(xs, xs + 3, zs, [](int x) { return abs(x); });
11
12    for (int i = 0; i < 3; ++i)
13        printf("%d%c", zs[i], " \n"[i + 1 == 3]);
14
15    transform(xs, xs + 3, ys, zs, [](int x, int y) { return x * y; });
16
17    for (int i = 0; i < 3; ++i)
18        printf("%d%c", zs[i], " \n"[i + 1 == 3]);
19
20    return 0;
21 }
```

# Busca Binária

---

# Busca binária

- A busca binária se vale da ordenação de um vetor de  $N$  elementos para acelerar o processo de busca
- A ordem de complexidade da busca binária é  $O(\log N)$
- O vetor deve estar em ordem crescente
- A busca binária identifica, primeiramente, o elemento  $m$  que está na posição central do intervalo  $[a, b]$  ( $m = (a + b)/2$ ) e o elemento  $x$  a ser localizado
- Se  $x = m$ , a busca retorna verdadeiro; caso contrário, ela compara os valores de  $x$  e  $m$
- Se  $x < m$ , a busca reinicia no intervalo à esquerda de  $m$  ( $[a, m - 1]$ ); se  $x > m$ , a busca continua no subvetor à direita da  $m$  ( $[m + 1, b]$ )
- Se  $b < a$ , a busca retorna falso

# Visualização da busca binária

Elemento a ser encontrado: 34

Intervalo considerado:  $[0, 8]$

Elemento central: 4

12	28	34	40	51	67	77	80	95
----	----	----	----	----	----	----	----	----



# Visualização da busca binária

Elemento a ser encontrado: 34

Intervalo considerado:  $[0, 3]$

Elemento central: 1

12	28	34	40	51	67	77	80	95
----	----	----	----	----	----	----	----	----

# Visualização da busca binária

Elemento a ser encontrado: 34

Intervalo considerado: [2,3]

Elemento central: 2

12	28	34	40	51	67	77	80	95
----	----	----	----	----	----	----	----	----

# Exemplo de uso de busca binária

```
13 int buscar_edicao(int edicao, const int *edicoes, int N)
14 {
15     int a = 0, b = N - 1, m;
16
17     while (a <= b) {
18         m = a + (b - a)/2;
19
20         if (edicoes[m] == edicao)
21             return 1;
22         else if (edicoes[m] > edicao)
23             b = m - 1;
24         else
25             a = m + 1;
26     }
27
28     return 0;
29 }
```

# Busca binária em C/C++

- A função `bsearch()` da biblioteca `stdlib.h` do C implementa a busca binária
- A sintaxe da função `bsearch()` é

```
void * bsearch(const void *key, const void *base, size_t nmemb,
               size_t size, int (*compar)(const void *, const void *));
```
- A biblioteca `algorithm` do C++ traz três funções associadas à busca binária
- A função `binary_search()` retorna verdadeiro se o elemento a ser encontrado está no intervalo indicado

```
bool binary_search(ForwardIterator first, ForwardIterator last, const T& val);
```
- As funções `lower_bound()` e `upper_bound()` retorna um iterador para o primeiro elemento maior ou igual a  $x$ , ou estritamente maior do que  $x$ , respectivamente:

```
ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last, const T& val);
ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last, const T& val);
```

# Exemplo de uso de busca binária em C e C++

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <algorithm>
4
5 using namespace std;
6
7 int compare(const void *a, const void *b)
8 {
9     const int *x = (const int *) a, *y = (const int *) b;
10    return *x == *y ? 0 : (*x < *y ? -1 : 1);
11 }
12
13 int main()
14 {
15     int ns[] { 2, 18, 45, 67, 99, 99, 99, 112, 205 }, N = 9, n = 99;
16     auto p = (int *) bsearch(&n, ns, N, sizeof(int), compare);
17
18     if (p == NULL)
19         cout << "Elemento " << n << " não encontrado\n";
20     else
21         cout << n << " encontrado na posição: " << p - ns << "\n";
```

## Exemplo de uso de busca binária em C e C++

```
23     n = 100;
24
25     cout << "Elemento " << n << (binary_search(ns, ns + N, n) ?
26         " " : " não ") << "encontrado\n";
27
28     n = 99;
29
30     auto it = lower_bound(ns, ns + N, n);
31     cout << "Cota inferior de " << n << ": " << it - ns << endl;
32
33     auto jt = upper_bound(ns, ns + N, n);
34     cout << "Cota superior de " << n << ": " << jt - ns << endl;
35
36     cout << "Número de aparições de " << n << ": " << jt - it << endl;
37
38     return 0;
39 }
```

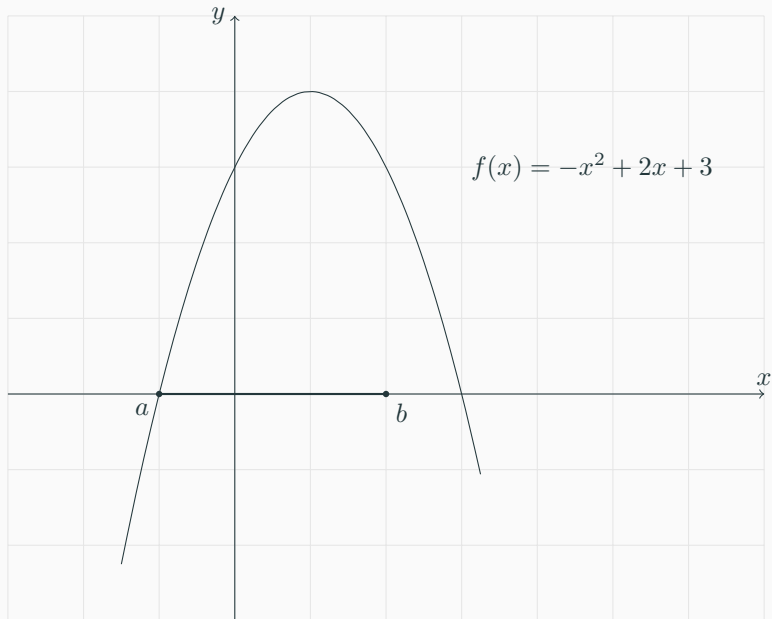
## **Busca Ternária**

---

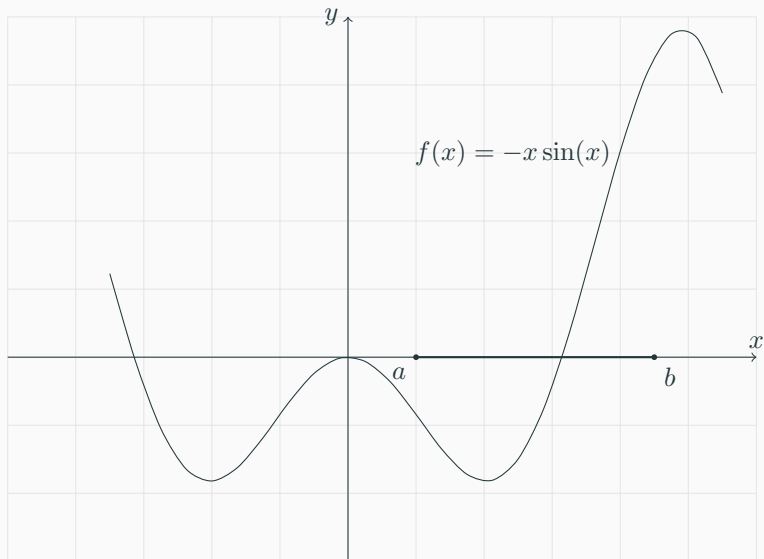
- Assim como a busca linear e a busca binária, e busca ternária também pode ser utilizada para localizar um elemento específico em um vetor ordenado
- Entretanto, ela pode ser utilizada também para localizar o valor máximo ou mínimo de uma função unimodal em um intervalo  $[a, b]$
- Uma função  $f(x)$  é unimodal no intervalo  $I = [a, b]$  se ela existe um ponto  $c \in I$  tal que
  1.  $f'(x) > 0$  se  $x \in [a, c)$ ,  $f'(c) = 0$  e  $f'(x) < 0$  se  $x \in (c, b]$ ; ou
  2.  $f'(x) < 0$  se  $x \in [a, c)$ ,  $f'(c) = 0$  e  $f'(x) > 0$  se  $x \in (c, b]$
- Observe que a busca binária não é capaz de localizar tal máximo diretamente neste cenário



# Exemplos de funções unimodais



## Exemplos de funções unimodais



# Algoritmo

- Seja  $f(x)$  uma função unimodal no intervalo  $I = [a, b]$  e  $m_1, m_2 \in I$  tais que  $a < m_1 < m_2 < b$ , com um valor máximo no ponto  $c \in I$
- Os valores  $f(m_1)$  e  $f(m_2)$  se relacionam de uma das três maneiras seguintes:
  1.  $f(m_1) < f(m_2)$
  2.  $f(m_1) > f(m_2)$
  3.  $f(m_1) = f(m_2)$
- No primeiro caso, o máximo não pode estar no intervalo  $[a, m_1]$ , pois ambos pontos  $m_1, m_2$  estão na área de crescimento da função
- Assim  $c > m_1$  e a busca deve prosseguir no intervalo  $[m_1, b]$
- O segundo caso é simétrico ao primeiro: ambos pontos estão na região de decrescimento, logo  $c$  deve estar no intervalo  $[a, m_2]$

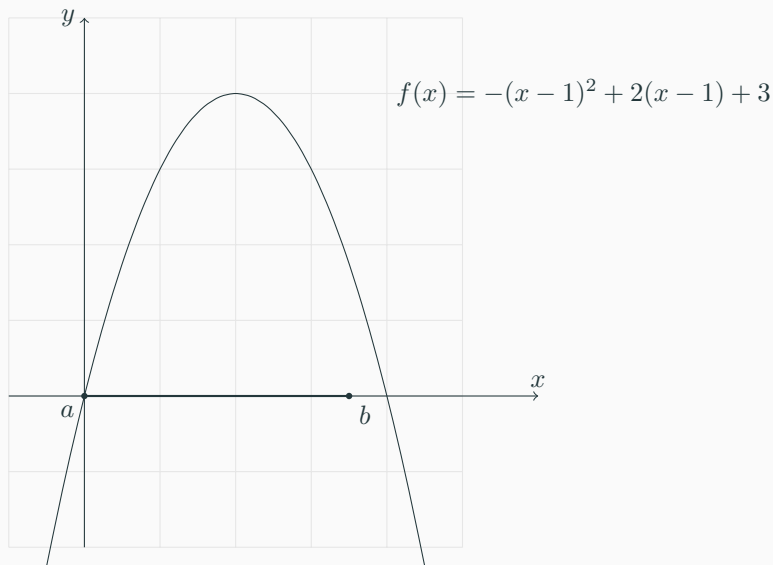
# Algoritmo

- No terceiro caso ocorre ou quando  $m_1 = m_2$  ou se  $m_1$  está na área de crescimento e  $m_2$  na área de crescimento
- Assim,  $c \in [m_1, m_2]$
- Para simplificar o algoritmo, o terceiro caso pode ser reduzido a um dos dois primeiros
- Se  $m_1$  e  $m_2$  dividirem  $[a, b]$  em três regiões iguais, a cada etapa da busca o intervalo é reduzido em um terço de seu tamanho
- Para esta divisão os valores a serem escolhidos são

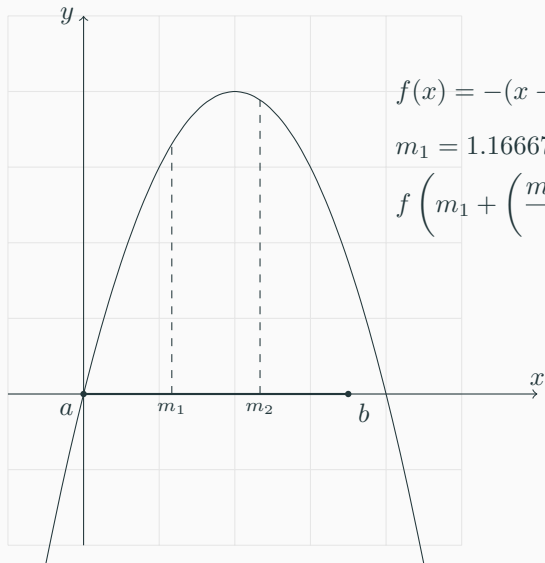
$$m_1 = a + \left( \frac{b - a}{3} \right)$$

$$m_2 = b - \left( \frac{b - a}{3} \right)$$

## Exemplos de funções unimodais



# Exemplos de funções unimodais

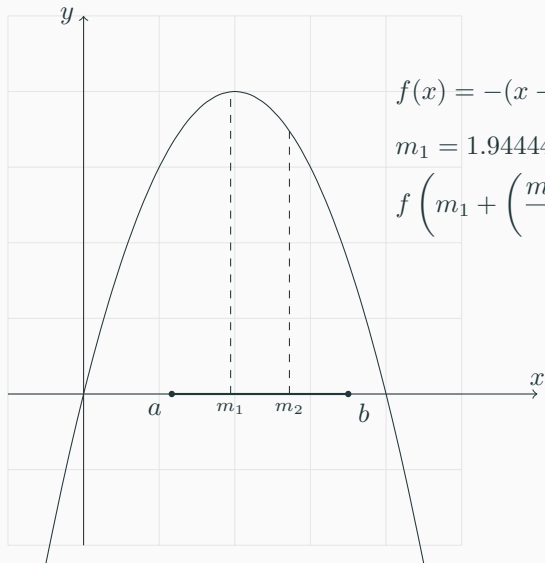


$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.16667, m_2 = 2.33333$$

$$f\left(m_1 + \left(\frac{m_2 - m_1}{2}\right)\right) = 3.88889$$

## Exemplos de funções unimodais

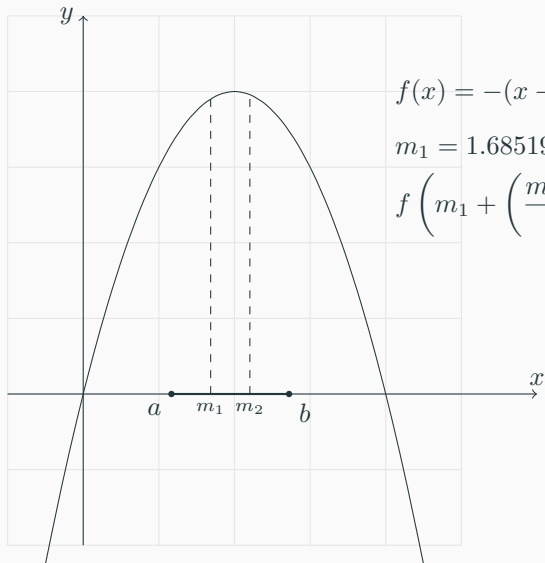


$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.94444, m_2 = 2.72222$$

$$f\left(m_1 + \left(\frac{m_2 - m_1}{2}\right)\right) = 3.99691$$

## Exemplos de funções unimodais





# Implementação iterativa da busca ternária

```
1 #include <bits/stdc++.h>
2
3 double f(double x)
4 {
5     return -(x - 1)*(x - 1) + 2*(x - 1) + 3;
6 }
7
8 double ternary_search(double a, double b, int runs = 50)
9 {
10     while (runs--)
11     {
12         auto m1 = a + (b - a)/3.0;
13         auto m2 = b - (b - a)/3.0;
14
15         f(m1) < f(m2) ? a = m1 : b = m2;
16     }
17
18     return f(a + (b - a)/2.0);
19 }
```

# Implementação recursiva da busca ternária

```
1 #include <bits/stdc++.h>
2
3 double f(double x)
4 {
5     return -(x - 1)*(x - 1) + 2*(x - 1) + 3;
6 }
7
8 double ternary_search(double a, double b, double eps = 1e-6)
9 {
10     if (fabs(b - a) < eps)
11         return f(a + (b - a)/2.0);
12
13     auto m1 = a + (b - a)/3.0;
14     auto m2 = b - (b - a)/3.0;
15
16     if (f(m1) < f(m2))
17         return ternary_search(m1, b, eps);
18     else
19         return ternary_search(a, m2, eps);
20 }
21
```

1. C Man Pages<sup>1</sup>.
2. CP Algorithms. [Ternary Search](#), acesso em 31/05/2019.
3. C++ Reference<sup>2</sup>.
4. Hacker Earth. [Ternary Search](#), acesso em 31/05/2019.
5. Wikipédia. [Ternary Search](#), acesso em 31/05/2019.

---

<sup>1</sup>Comando man no Linux.

<sup>2</sup><https://en.cppreference.com/w/>