Busca e Ordenação

Algoritmos de Busca

Prof. Edson Alves - UnB/FGA

2020

Sumário

1. Busca Sequencial

2. Busca Binária

3. Busca Ternária

Busca Sequencial

Busca sequencial

- 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

Implementação da busca sequencial

```
1 int search(int x, const vector<int>& xs)
2 {
3     for (size_t i = 0; i < xs.size(); ++i)
4         if (xs[i] == x)
5             return i;
6
7     return -1;
8 }
```

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 assinatura da função `find()` é

```
InputIterator find(InputIterador 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 um tipo que suporte o operador de comparação ==

Exemplo de uso da função find()

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

using namespace std;

int main()
{
    const vector<int> ps { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
    int p = 21;

    auto it = find(ps.begin(), ps.end(), p);

    if (it != ps.end())
        cout << p << " encontrado na posicao " << it - ps.begin() << "\n";
    else
        cout << p << " nao encontrado\n";

    return 0;
}
```

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 predicado P
- Um predicado 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
- A busca sequencial pode ser interpretada como um filtro que seleciona um (ou mais) elemento do contêiner a partir do predicado

```
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 predicado 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 assinatura da função `copy_if()` é

```
OutputIterator copy_if(InputIterator first, InputIterador 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 implementar a função `push_back()`

Exemplo de uso de filtro

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

using namespace std;

int main()
{
    const string vowels { "aeiou" };
    string message { "Exemplo de busca sequencial" }, res;

    auto P = [&vowels](char c) { return vowels.find(c) != string::npos; };

    copy_if(message.begin(), message.end(), back_inserter(res), P);

    if (res.empty())
        cout << "Nenhuma vogal encontrada\n";
    else
        cout << res.size() << " vogais encontradas: " << res << "\n";

    return 0;
}
```

Transformações

- 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 assinatura da função `transform()` é

```
OutputIterator transform(InputIterator first, InputIterator last,
    OutputIterador result, UnaryOperation op);
```

- Há também uma versão da função `transform()` que aceita uma operação binária, cuja assinatura é

```
OutputIterator transform(InputIterator first1, InputIterator last1,
    InputIterator first2, OutputIterador result, BinaryOperation op);
```

Exemplo de uso de transformações

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

using namespace std;

int main()
{
    int xs[] { -1, 0, 4 }, ys[] { 2, 3, -3 }, zs[3];

    transform(xs, xs + 3, zs, [](int x) { return abs(x); });

    for (int i = 0; i < 3; ++i)
        cout << z[i] << (i + 1 == 3 ? '\n' : ' ');

    transform(xs, xs + 3, ys, zs, [](int x, int y) { return x * y; });

    for (int i = 0; i < 3; ++i)
        cout << z[i] << (i + 1 == 3 ? '\n' : ' ');

    return 0;
}
```

Busca Binária

Busca binária

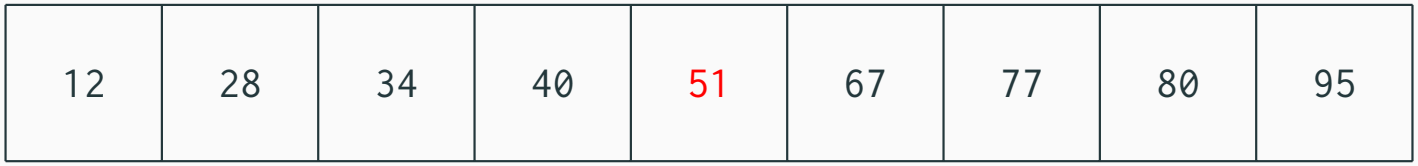
- A busca binária se vale da ordenação de um vetor de N elementos para acelerar o processo de busca
- A ordem de complexidade da busca binária é $O(\log N)$
- O vetor deve estar em ordem crescente
- A busca binária identifica, primeiramente, o elemento m que está na posição central do intervalo $[a, b]$ ($m = (a + b)/2$) 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

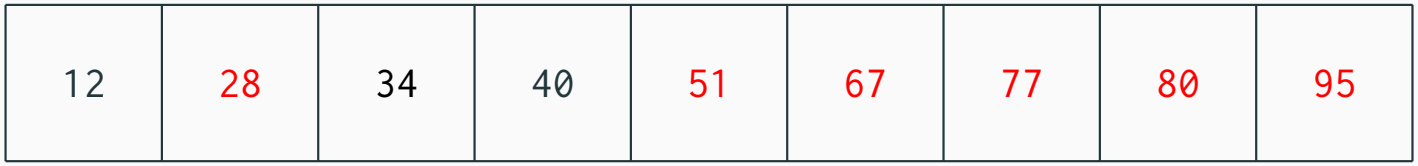


Visualização da busca binária

Elemento a ser encontrado: 34

Intervalo considerado: [0,3]

Elemento central: 1

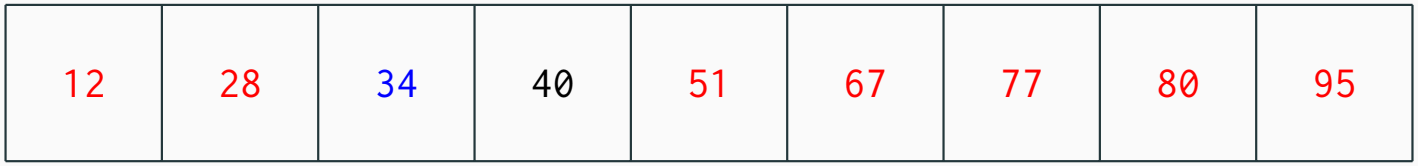


Visualização da busca binária

Elemento a ser encontrado: 34

Intervalo considerado: [2,3]

Elemento central: 2



Exemplo de implementação da busca binária

```
1  int binary_search(int x, const vector<int>& xs)
2  {
3      int a = 0, b = xs.size() - 1;
4
5      while (a <= b)
6      {
7          auto m = a + (b - a)/2;
8
9          if (xs[m] == x)
10             return m;
11         else if (xs[m] > x)
12             b = m - 1;
13         else
14             a = m + 1;
15     }
16
17     return -1;
18 }
```

Busca binária em C

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

```
void * bsearch(const void *key, const void *base, size_t nmemb,
    size_t size, int (*compar)(const void *, const void *));
```

- O parâmetro `key` é um ponteiro para o valor a ser localizado no vetor `base`
- O número de elementos do vetor `base` é igual a `nmemb`, e cada um deste elementos ocupa `size` *bytes* em memória
- O parâmetro `compar` é um ponteiro para uma função que recebe dois ponteiros e retorna negativo, zero ou positivo se o primeiro ponteiro aponta para um valor menor, igual ou maior do que o valor apontado pelo segundo ponteiro, respectivamente

Busca binária em C++

- 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()` retornam 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++

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

using namespace std;

int compare(const void *a, const void *b)
{
    const int *x = (const int *) a, *y = (const int *) b;
    return *x == *y ? 0 : (*x < *y ? -1 : 1);
}

int main()
{
    int ns[] { 2, 18, 45, 67, 99, 99, 99, 112, 205 }, N = 9, n = 99;
    auto p = (int *) bsearch(&n, ns, N, sizeof(int), compare);

    if (p == NULL)
        cout << "Elemento " << n << " não encontrado\n";
    else
        cout << n << " encontrado na posição: " << p - ns << "\n";

    n = 100;
```

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

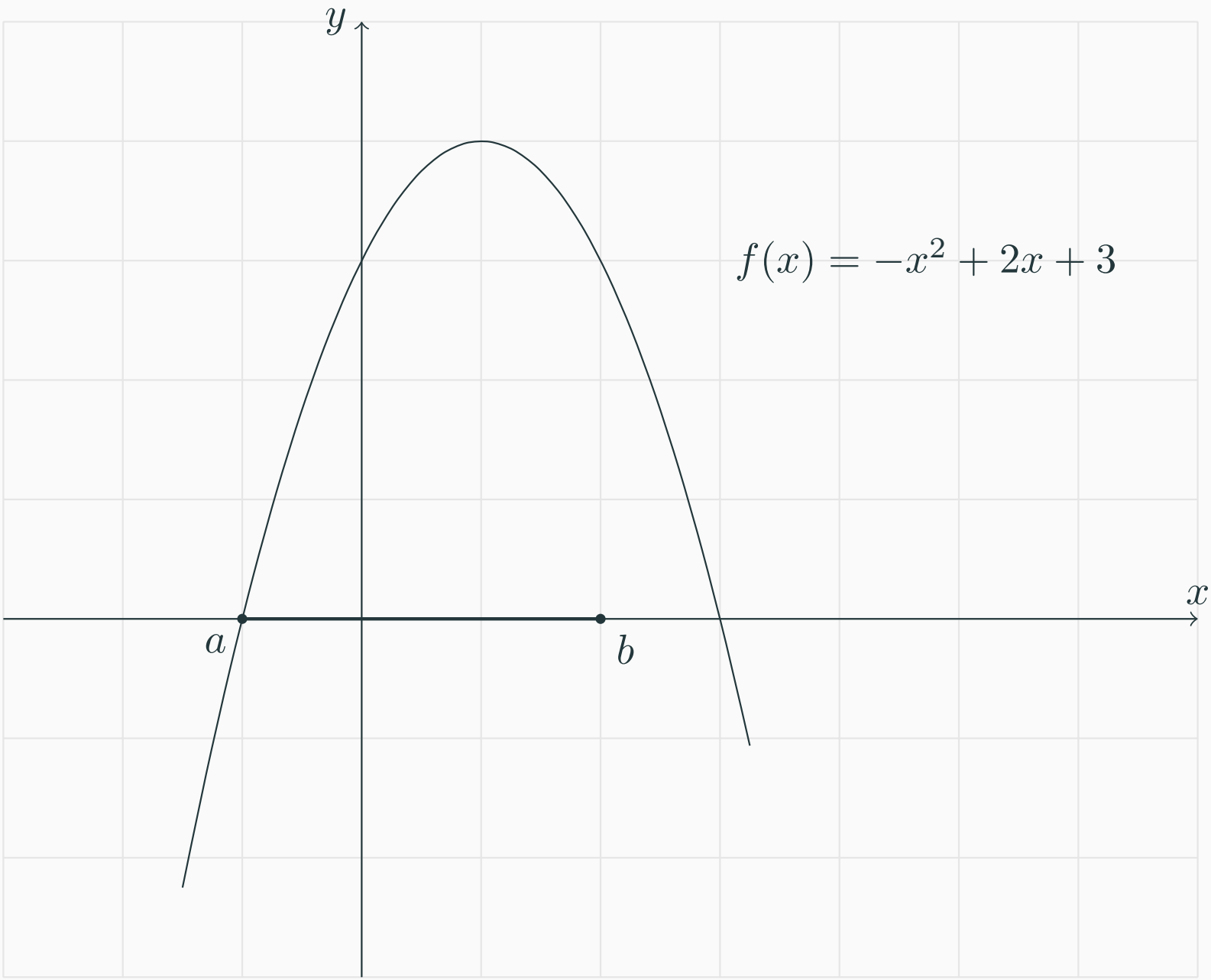
```
23      cout << "Elemento " << n << (binary_search(ns, ns + N, n) ?
24          " " : " não ") << "encontrado\n";
25
26      n = 99;
27
28      auto it = lower_bound(ns, ns + N, n);
29      cout << "Cota inferior de " << n << ": " << it - ns << endl;
30
31      auto jt = upper_bound(ns, ns + N, n);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

Busca Ternária

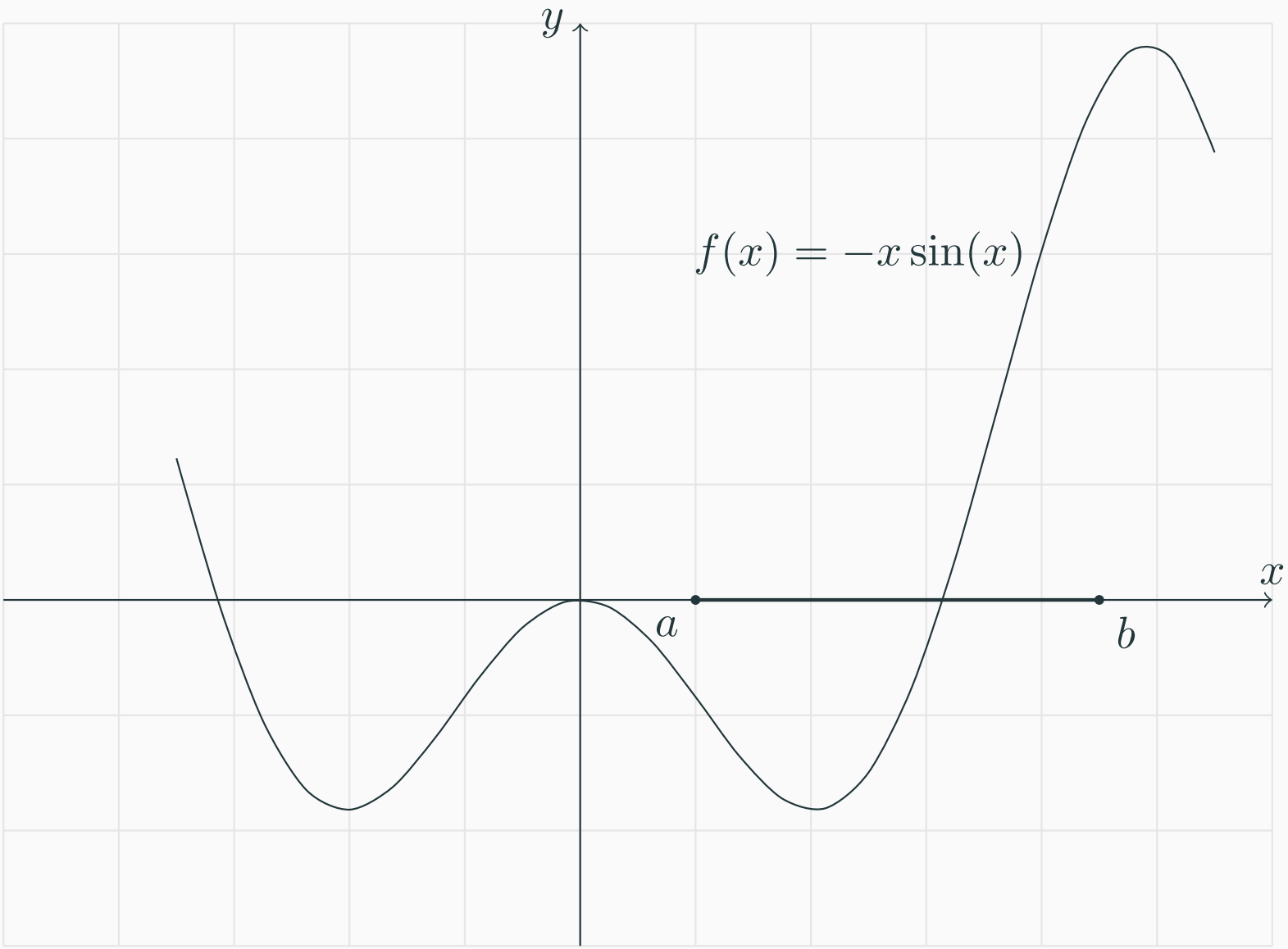
Motivação

- Assim como a busca linear e a busca binária, a 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 área de crescimento da função está à direita de m_1
- Assim $c > m_1$ e a busca deve prosseguir no intervalo $[m_1, b]$
- O segundo caso é simétrico ao primeiro: a região de decrescimento está à esquerda de m_2, logo c está no intervalo $[a, m_2]$

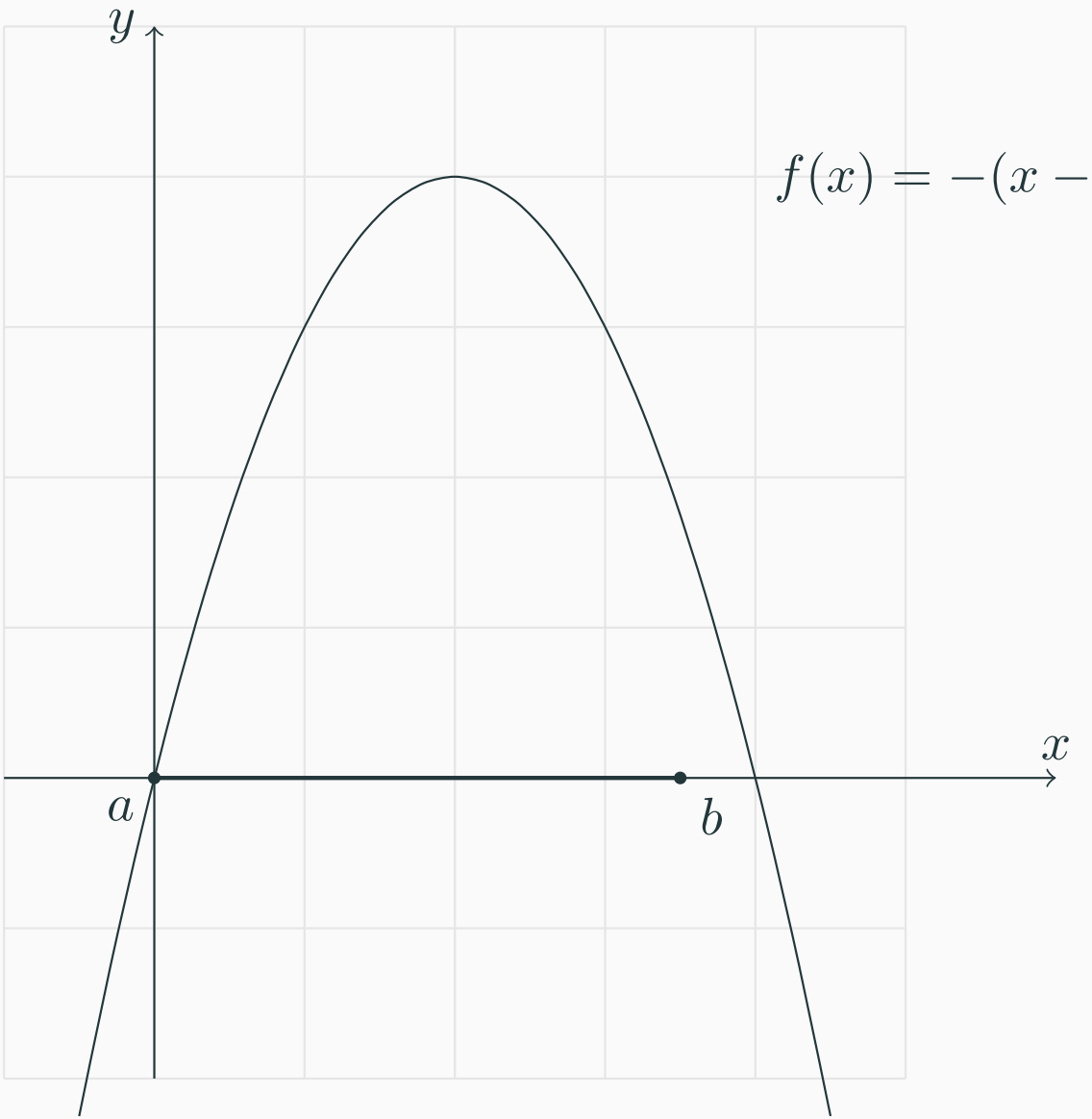
Algoritmo

- 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, ou vice-versa
- 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 o intervalo de busca é reduzido em um terço de seu tamanho
- Para esta divisão os valores a serem escolhidos são

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

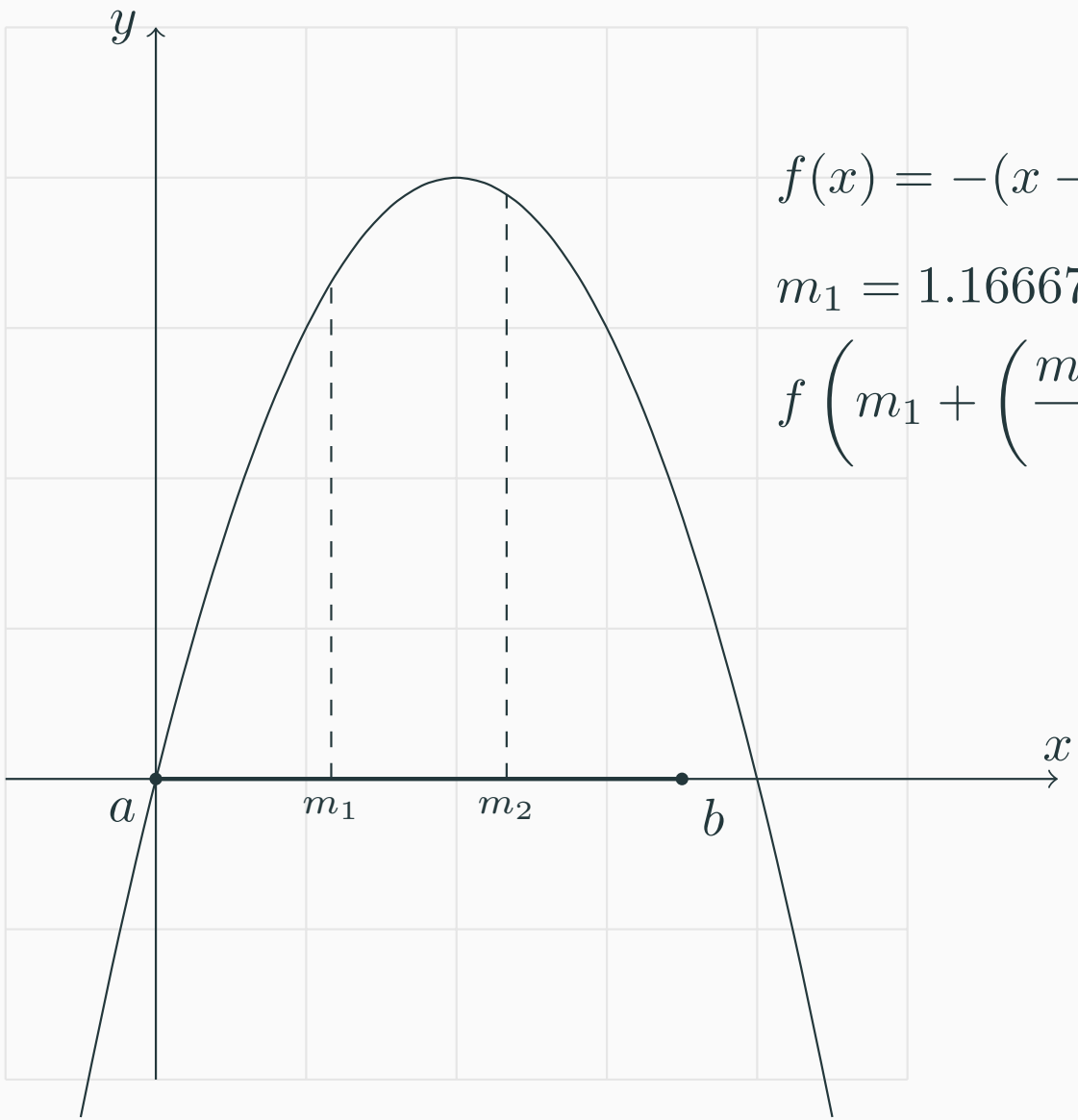
$$m_2 = b - \left(\frac{b-a}{3}\right)$$

Exemplos de busca ternária em função unimodal



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

Exemplos de busca ternária em funções unimodais

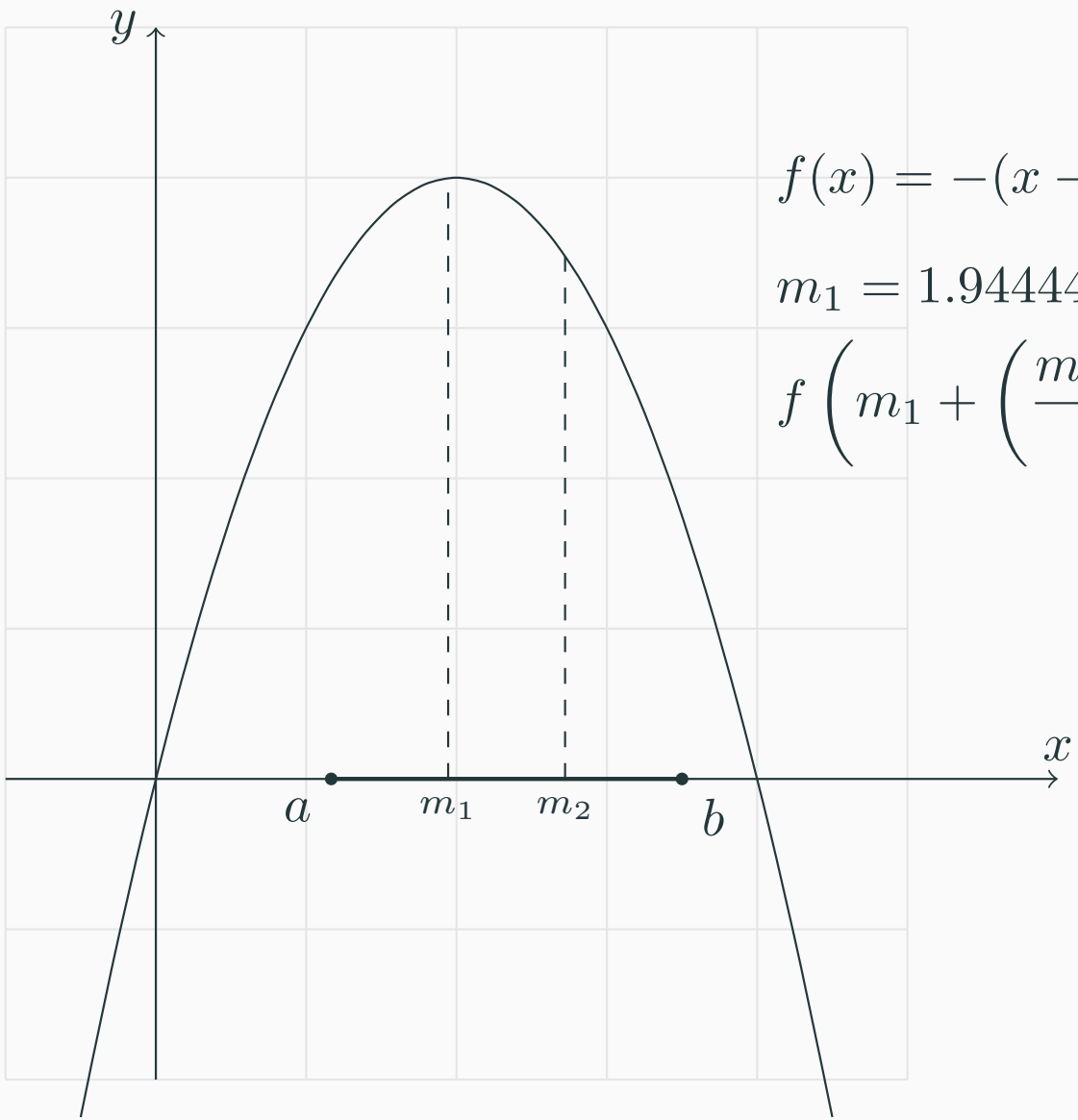


$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.16667, m_2 = 2.33333$$

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

Exemplos de busca ternária em funções unimodais

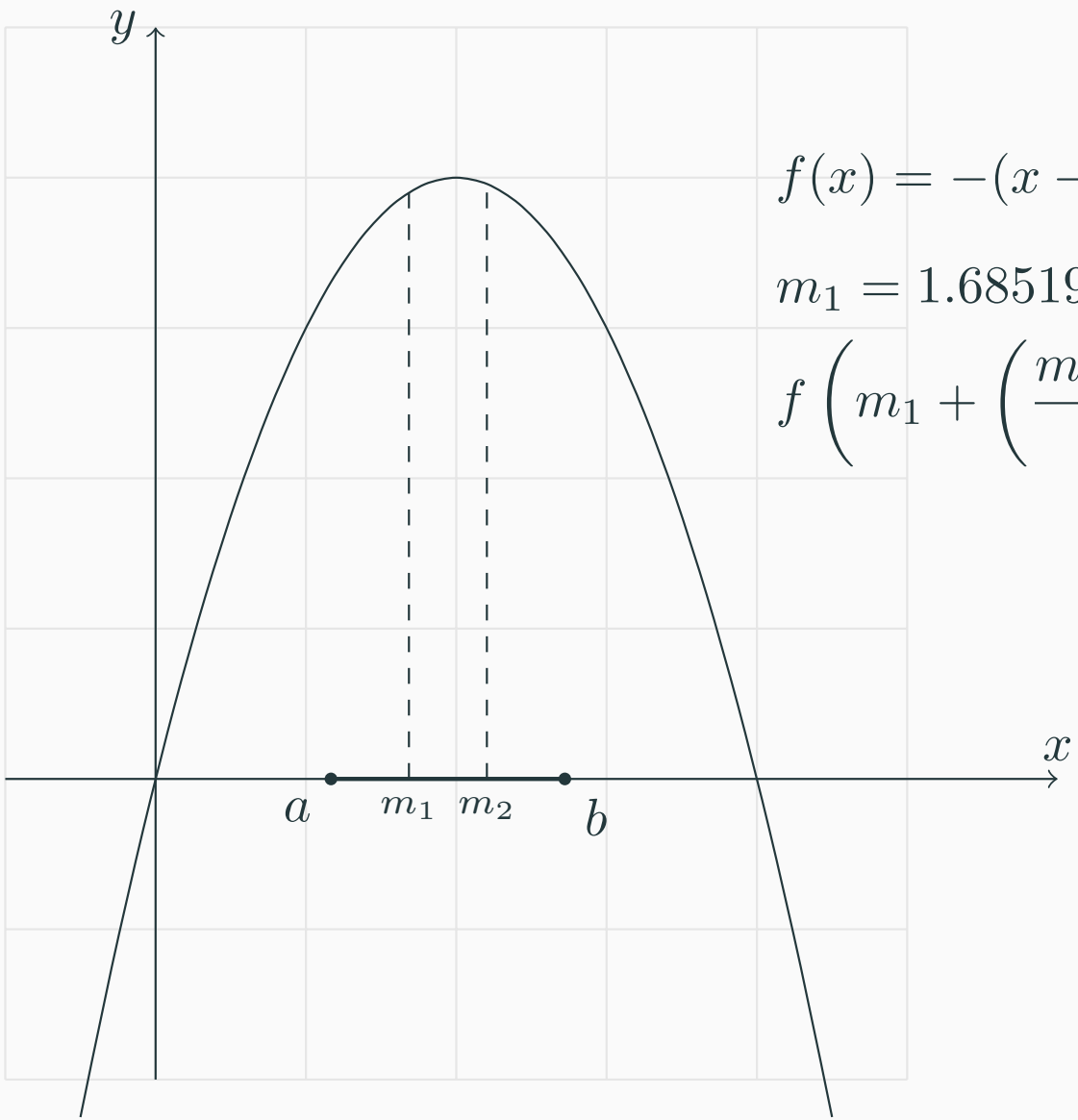


$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.94444, m_2 = 2.72222$$

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

Exemplos de busca ternária em funções unimodais



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

$$m_1 = 1.68519, m_2 = 2.20370$$

$$f\left(m_1 + \left(\frac{m_2 - m_1}{2}\right)\right) = 3.9585$$

Implementação iterativa da busca ternária

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

double f(double x)
{
    return -(x - 1)*(x - 1) + 2*(x - 1) + 3;
}

double ternary_search(double a, double b, int runs = 50)
{
    while (runs--)
    {
        auto m1 = a + (b - a)/3.0;
        auto m2 = b - (b - a)/3.0;

        f(m1) < f(m2) ?  a = m1 : b = m2;
    }

    return f(a + (b - a)/2.0);
}
```

Implementação recursiva da busca ternária

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

double f(double x)
{
    return -(x - 1)*(x - 1) + 2*(x - 1) + 3;
}

double ternary_search(double a, double b, double eps = 1e-6)
{
    if (fabs(b - a) < eps)
        return f(a + (b - a)/2.0);

    auto m1 = a + (b - a)/3.0;
    auto m2 = b - (b - a)/3.0;

    if (f(m1) < f(m2)
        return ternary_search(m1, b, eps);
    else
        return ternary_search(a, m2, eps);
}

```

Referências

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

[1] Comando `man` no Linux.
[2] https://en.cppreference.com/w/