# Paradigmas de Solução de Problemas

## Divisão e Conquista: Busca Binária e Busca Ternária

Prof. Edson Alves – UnB/FGA

# Sumário

# Busca Binária

# Busca binária

- A busca binária utiliza o paradigma de divisão e conquista para, a cada etapa de uma busca, descartar uma parte significativa das possíveis localizações do elemento a ser identificado
- Para tal, é necessário que os elementos estejam dispostos em uma determinada ordenação
- A etapa de divisão escolhe o elemento $m$ que esteja na posição central, ou próximo a ela, quando os elementos estão dispostos de acordo com a ordenação
- O conjunto de elemento então é dividido em 3 conjuntos disjuntos: os elementos que ficaram à esquerda de $m$ ($L$), um conjunto unitário com o próprio $m$ e os elementos que estão à direita de $m$ ($R$)

## Busca binária

- A etapa de conquista acontece neste conjunto unitário
- Caso o elemento desejado seja $m$, o algoritmo termina
- Caso não seja, a busca continua em apenas um dos conjuntos $L$ ou $R$, a depender da relação do elemento procurado com $m$
- Neste algoritmo, não há uma etapa de fusão
- A ordem de complexidade da busca binária é $O(\log N)$, desde que o acesso aleatório seja feito em $O(1)$

# Busca binária em vetores

- A busca binária se vale da ordenação de um vetor de $N$ elementos para acelerar o processo de busca
- Assuma que o vetor esteja 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]`

Índice do 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]

Índice do 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]`

Índice do elemento central: `2`

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

# Exemplo de implementação da busca binária

```
int binary_search(int x, const vector<int>& xs)
{
    auto a = 0ul, b = xs.size() - 1;

    while (a <= b)
    {
        auto m = a + (b - a)/2;

        if (xs[m] == x)
            return m;
        else if (xs[m] > x)
            b = m - 1;
        else
            a = m + 1;
    }

    return -1;
}
```

# Busca binária em C

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

```c
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";
```

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

```
    n = 100;

    cout << "Elemento " << n << (binary_search(ns, ns + N, n) ?
        " " : " não ") << "encontrado\n";

    n = 99;

    auto it = lower_bound(ns, ns + N, n);
    cout << "Cota inferior de " << n << ": " << it - ns << endl;

    auto jt = upper_bound(ns, ns + N, n);
    cout << "Cota superior de " << n << ": " << jt - ns << endl;

    cout << "Número de aparições de " << n << ": " << jt - it << endl;

    return 0;
}
```

# Método da bisecção

- O método da bisecção utiliza a busca binária para identificar uma raiz de uma função $f(x)$ em um intervalo $(a, b)$
- Este método pode ser aplicado se $f(x)$ for contínua em $(a, b)$ e se $f(a)f(b) < 0$
- Isto significa que os valores de $f(x)$ nos extremos do intervalo tem sinais opostos
- Como $f(x)$ é continua no intervalo, partindo de $a$, ela tem que atravessar, ao menos uma vez, o eixo-$x$ para atingir o ponto $b$
- Seja $c$ um ponto tal que $f(c) = 0$
- O método da bisecção tenta localizar tal $c$, buscando, inicialmente, o elemento central do intervalo

# Método da bisecção

- Caso este elemento não seja igual a $c$, isto significa que $f(m) \neq 0$
- A partir das relações entre $f(a), f(m)$ e $f(b)$, a busca continua ou no intervalo $(a, m)$ ou $(b, m)$
- O algoritmo deve ser interrompido quando $f(m) = 0$
- Porém, devido à aritmética de ponto flutuante, pode ser que ista condição jamais seja satisfeita
- Assim, pode-se adotar como critério de parada
  1. um limiar $\varepsilon > 0$ e parar o algoritmo quando $f(m) < \varepsilon$, ou
  2. um número fixo $N$ de interações do algoritmo
- Em ambos casos, a complexidade do algoritmo é $O(\log(b - a))$

# Implementação da biseção com limiar

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

using namespace std;
const double EPS { 1e-7 }, PI = acos(-1.0);

double bisection(const function<double(double)>& f, double a, double b)
{
    auto m = (a + b)/2.0, y = f(m);

    return fabs(y) < EPS ? m : (y*f(a) < 0 ? bisection(f, a, m) : bisection(f, m, b));
}

int main()
{
    auto f = [](double x) { return sin(x) - 0.8; };

    cout << setprecision(8) << bisection(f, 0, PI/2) << '\n';

    return 0;
}
```

# Busca binária na resposta

- Seja função $f(x)$ é monótona em um intervalo $[a, b]$
- Isto significa que $f(x)$ é não-decrescente ($f(x) \leq f(y)$, se $x \leq y$) ou não-crescente ($f(x) \geq f(y)$, se $x \leq y$) em $[a, b]$
- Assim, para uma sequência de valores $a \leq x_1 < x_2 < \ldots < x_N \leq b$, as imagens $y_i = f(x_i)$ formarão uma sequência também monótona
- Deste modo, interpretanto os valores $x_i$ como os índices de um vetor cujos valores são $y_i$, é possível, por meio da busca binária, identificar um $x_0$ tal que $f(x_0) = y_0$ para um $y_0$ escolhido
- Esta técnica, denominada busca binária na resposta, é útil quando $f(x)$ é uma função monótona que é difícil de computar ou que representa um processo elaborado, e se deseja encontrar um valor $x_0$ que atenda uma série de pré-requisitos ou condições

# Exemplo de busca binária na resposta: Triângulo de Pascal

- Considere o seguinte problema: dado $M \leq 10^{18}$, determine o menor $N$ tal que a $N$-ésima linha do Triângulo de Pascal contenha ao menos um coeficiente maior ou igual a $M$
- O Triângulo de Pascal é formado pela linha 0, que contém apenas o número 1, a linha 1, com dois números 1, e as demais linhas começam e terminam com 1, e os elementos intermediários são formados pela soma dos dois elementos imediatamente acima

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

## Exemplo de busca binária na resposta: Triângulo de Pascal

- De fato, o $i$-ésimo coeficienta da linha $n$ é dado por

$$C(n,i) = \binom{n}{i} = \frac{n!}{(n-i)!i!}$$

- Observe que o maior coeficiente da linha $n$ ocupa a posição central
- Assim, se $c_k$ é o coeficiente que ocupa a posição central da $k$-ésima linha, a sequência $\{c_1, c_2, \ldots, c_N\}$ é monótona, de modo que o problema pode ser resolvido por meio de busca binária na resposta
- Por inspeção, $c_0 = 1$ e $c_{64} = 1832624140942590534 > 10^{18}$
- Assim, basta realizar a busca no intervalo $[0, 64]$
- Como cada coeficiente pode ser computado em $O(N)$, a complexidade da solução será $O(N \log I)$, onde $I$ é o tamanho do intervalo de busca

# Implementação do exemplo em Python

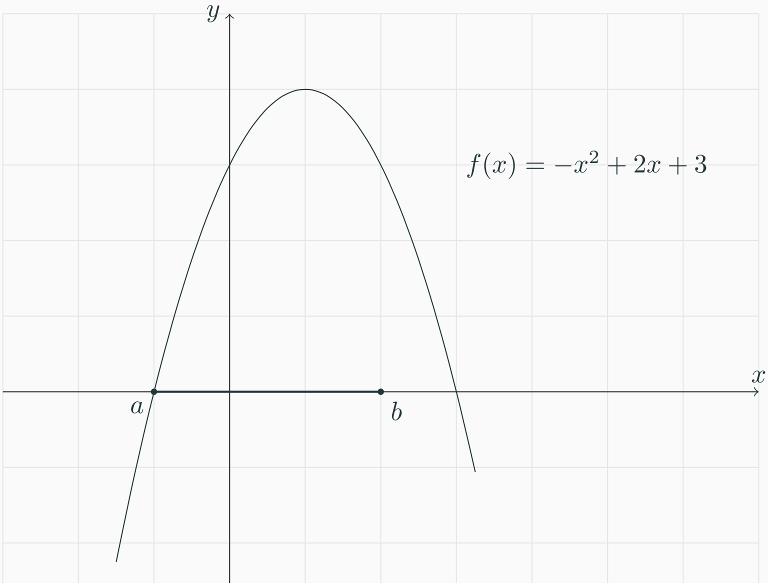
```python
from math import factorial as f

def binom(n, m):
    return f(n) // (f(n - m) * f(m))

def min_row(M):
    a = 0
    b = 64
    N = 64

    while a <= b:
        m = (a + b) // 2

        if binom(m, m // 2) >= M:
            N = m
            b = m - 1
        else:
            a = m + 1

    return N
```

# Busca Ternária

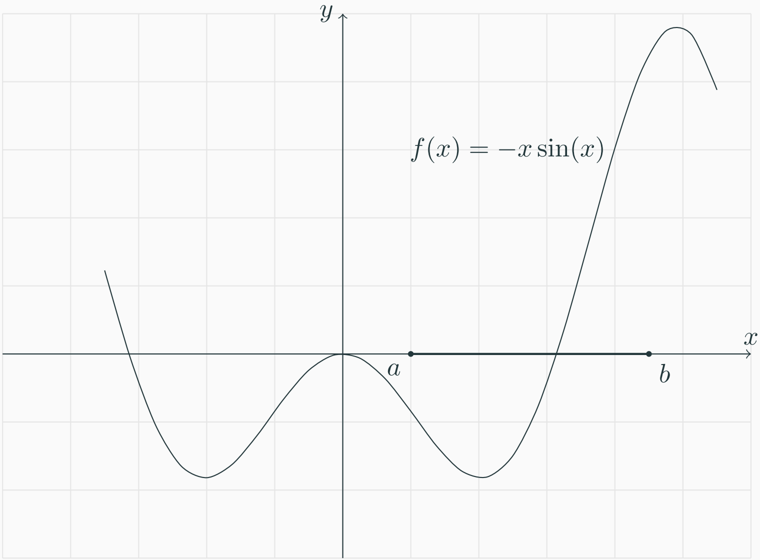
# Motivação

- A busca ternária também utiliza a divisão e conquista para reduzir significativamente o espaço de busca a cada iteração do algoritmo
- Ela pode ser utilizada 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 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

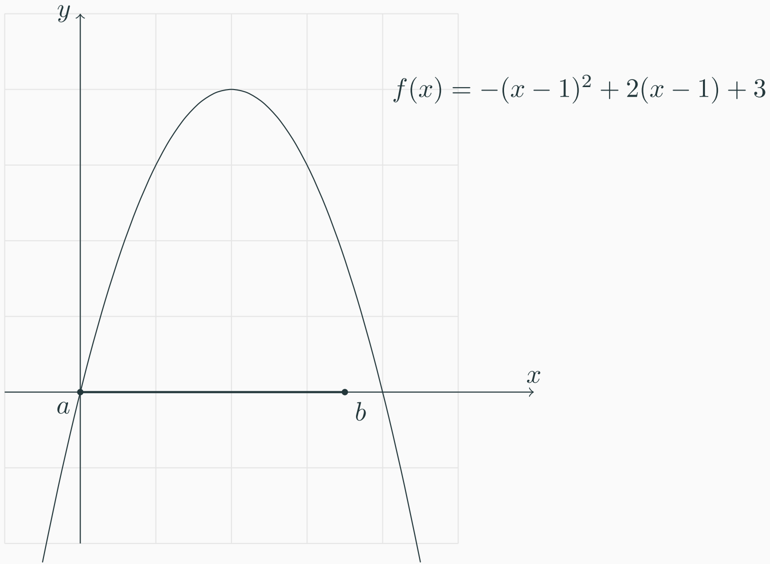
- Seja $f(x)$ uma função unimodal no intervalo $I = [a, b]$ e $m_1, m_2 \in I$ tais que $a < m_1 < m_2 < b$, com um valor máximo no ponto $c \in I$
- Os valores $f(m_1)$ e $f(m_2)$ se relacionam de uma das três maneiras seguintes:
  1. $f(m_1) < f(m_2)$
  2. $f(m_1) > f(m_2)$
  3. $f(m_1) = f(m_2)$
- 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 direita de $m_2$, logo $c$ está no intervalo $[a, m_2]$

# Algoritmo

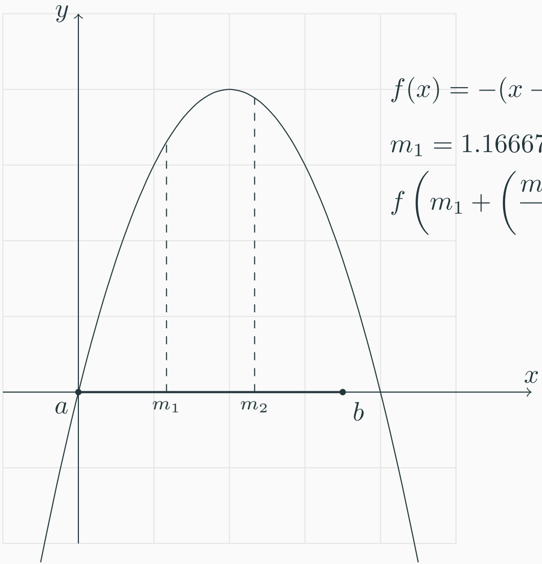
- No terceiro caso ocorre ou quando $m_1 = m_2$ ou se $m_1$ está na área de crescimento e $m_2$ na área de decrescimento, ou vice-versa
- Assim, $c \in [m_1, m_2]$
- Para simplificar o algoritmo, o terceiro caso pode ser reduzido a um dos dois primeiros
- Se $m_1$ e $m_2$ dividirem $[a, b]$ em três regiões 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



# Exemplos de busca ternária em funções unimodais

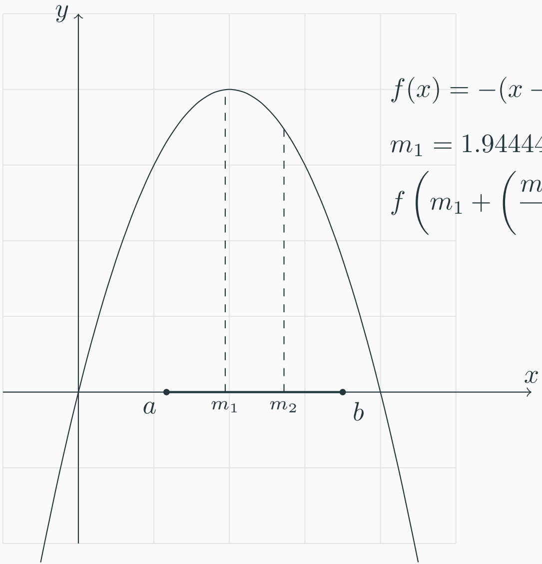



$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.16667, m_2 = 2.33333$$

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

# Exemplos de busca ternária em funções unimodais

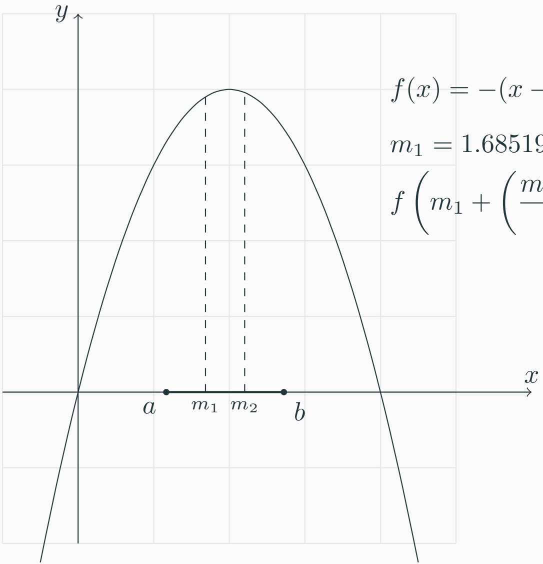



$$f(x) = -(x-1)^2 + 2(x-1) + 3$$

$$m_1 = 1.94444, m_2 = 2.72222$$

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

# 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. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
6. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
7. Wikipédia. Ternary Search, acesso em 31/05/2019.


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