Métodos de Busca

Pesquisa, Ordenação e Técnicas de Armazenamento

Prof. Msc. Bruno de A. Iizuka Moritani
bruno.moritani@anhembi.br

Agenda

- O problema da busca
- Busca sequencial
  - Definição
  - Método tradicional
  - Complexidade
  - Otimização com sentinela
  - Otimização para vetor ordenado
- Busca binária
  - Definição
  - Complexidade
  - Método tradicional
- Busca por interpolação
  - Definição
  - Complexidade
  - Método tradicional
  - Implementação

Análise de Algoritmos

- O que é analisar um algoritmo?
  - Prever os recursos de que o algoritmo necessitará;
  - Prever desempenho, comparar algoritmos e ajustar parâmetros;
  - Além de memória, largura de banda de comunicação e hardware, algo extremamente importante medir é o tempo de computação, identificando assim qual algoritmo é mais eficiente na resolução de um problema.

Análise de Algoritmos

- Modelo de tecnologia de implementação:
  - Um único processador;
  - Utilização da memória RAM;
  - Instruções executadas uma após a outra;
  - Sem operações concorrentes.

Análise de Algoritmos

- Modelo de tecnologia de implementação:
  - Um único processador;
  - Utilização da memória RAM;
  - Instruções executadas uma após a outra;
  - Sem operações concorrentes.

Operações Primitivas em Algoritmos

- Atribuição de valores a variáveis;
  - Exemplo:
    - `variavel = “Bruno”;`
- Chamadas de métodos;
  - Exemplo:
    - `MetodoCalculaSoma(10,20);`
- Operações aritméticas;
  - Exemplo:
    - `total = x + y;`
- Comparação entre dois números;
  - Exemplo:
    - `total == 10;`
- Acesso a um arranjo;
  - Exemplo:
    - `Vetor[0] = 10;`
- Seguimento de uma referência para um objeto;
  - Exemplo:
    - `metodoVerificaNome (aluno);`
- Retorno de um método.
  - Exemplo:
    - `return total;`

Contagem das Operações Primitivas

- Contagem de quantas operações primitivas são executadas;
- Usa este número *t* como uma estimativa do tempo de execução do algoritmo;
- Operações Primitivas equivalem a 1.

Notação Assintótica

- Casos a considerar são:
  - Melhor caso
  - Pior caso
  - Médio caso

Notação Assintótica

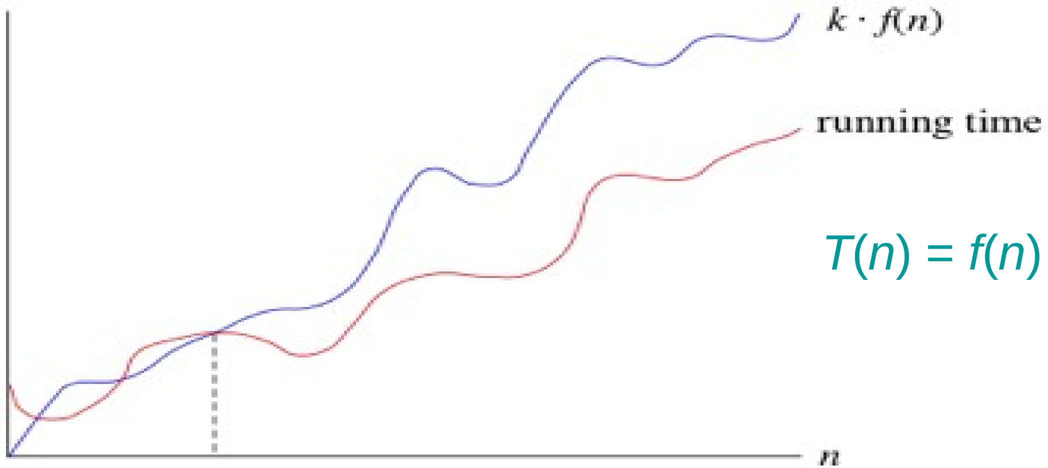
- O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.
- Para uma dominação assintótica existem notações:
  - Notação O (ômicron) – conhecido como “Big O”
  - Notação Ω (ômega)
  - Notação Θ (theta)

Notação Assintótica O

- Dizemos que $T(n)$ é $O(g(n))$ se existem constantes positivas c e n_0 tal que

$$T(n) <= c \, . \, g(n) \text{ para todo } n >= n_0$$

- Para n grande, $T(n)$ não cresce mais rápido que $g(n)$.
- $T(n)$ é $O(g(n))$ deve ser entendido como $T(n) \in O(g(n))$.

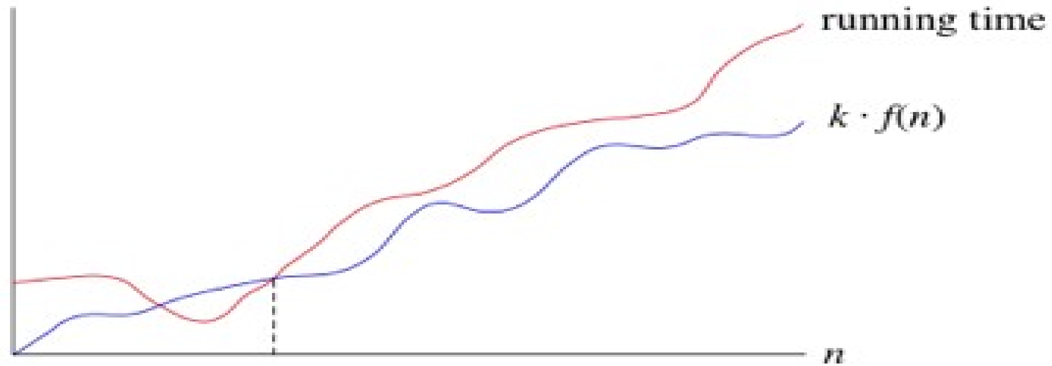


$T(n) = f(n)$

Notação Assintótica Ω

- Dizemos que $T(n)$ é $\Omega(g(n))$ se existem constantes positivas c e n_0 tal que

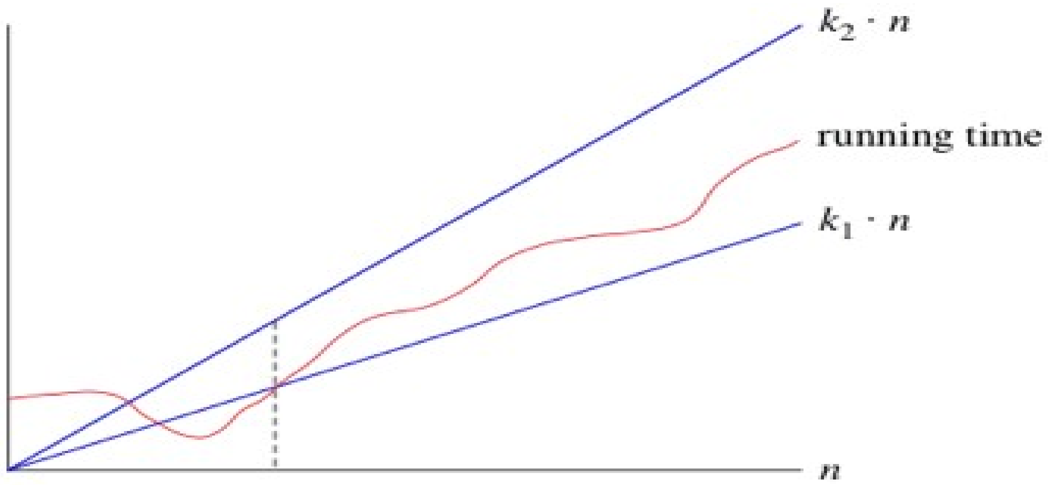
$$T(n) >= c \,.\, g(n) \text{ para todo } n >= n_0 \,.$$



$T(n) = f(n)$

Notação Assintótica Θ

- Θ define uma relação de equivalência.
- $T(n)$ é $\Theta(f(n))$ se $T(n) \in O(f(n))$ e $T(n) \in \Omega(f(n))$
- Limite assintótico justo.



$T(n) = f(n)$

Comparação de Execução

- Seja a função fibonacci(*n*) que calcula o *n*-ésimo elemento da seqüência de Fibonacci.
- Input: Valor de *n*
- Output: O *n*-ésimo elemento da sequência de Fibonacci

```
private static int fibonacci(int n) {

    if (n <= 1) {
        vetor[n] = n;
        return n;
    } else {
        vetor[n] = fibonacci(n - 1) + fibonacci(n - 2);

        return vetor[n];
    }

}
```

Comparação de Execução

```java
private static void fibonacci(int numero) {
    int total = 0;
    int anterior = 0;
    int preAnterior = 0;
    int i = 0;

    while (i < numero) {
        if (i == 0) {
            total = 0;
            anterior = 0;
            preAnterior = 0;
        } else if (i == 1){
            total = 1;
            anterior = total;
            preAnterior = 0;
        } else {
            total = anterior + preAnterior;
            preAnterior = anterior;
            anterior = total;

        }
        i++;
        if (i == numero){
            System.out.println (total);
        } else {
            System.out.print(total + ", ");
        }
    }
}
```

- A complexidade agora passou de $O(2^n)$ para $O(n)$

Exemplo de Análise de Algoritmo

- Vamos considerar o seguinte algoritmo de busca. Como parâmetro, passaremos um valor de *k* que não se encontra no vetor (pior caso):

```
int busca(int[] vetor, int k) {
    int i = 1;
    while (i <= n) {

        if (vetor[i] == k) {

            return i;
        }
        i++;
    }
    return -1;
}
```

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

Neste caso, a atribuição da linha 21 será executada 1 vez.

- $T(n) = 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20    int busca(int[] vetor, int k) {
21        int i = 1;
22        while (i <= n) {
23
24            if (vetor[i] == k) {
25
26                return i;
27            }
28            i++;
29        }
30        return -1;
31    }
```

A comparação da linha 22 será executada para cada valor de *i*, de 1 a *n* + 1, ou seja, *n* + 1 vezes.

- $T(n) = 1 + (n + 1) + \dots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A comparação da linha 24 será executada para cada valor de 1 a *n*, ou seja, *n* vezes.

- $T(n) = 1 + (n + 1) + (n) + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A linha 26 não será executada.

- $T(n) = 1 + (n + 1) + (n) + 0 + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A linha 28 será executada *n* vez.

- $T(n) = 1 + (n + 1) + (n) + 0 + (n) \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23  
24          if (vetor[i] == k) {
25  
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A linha 30 será executada 1 vez.

- $T(n) = 1 + (n + 1) + (n) + 0 + (n) + 1$

Exemplo de Análise de Algoritmo

- E como seria a análise do mesmo algoritmo no melhor caso?

```
int busca(int[] vetor, int k) {
    int i = 1;
    while (i <= n) {

        if (vetor[i] == k) {

            return i;
        }
        i++;
    }
    return -1;
}
```

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23  
24          if (vetor[i] == k) {
25  
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

Neste caso, a atribuição da linha 21 será executada 1 vez.

- $T(n) = 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20    int busca(int[] vetor, int k) {
21        int i = 1;
22        while (i <= n) {
23
24            if (vetor[i] == k) {
25
26                return i;
27            }
28            i++;
29        }
30        return -1;
31    }
```

A comparação linha 22
será executada 1 vez,
pois o número que
gostaríamos está na primeira
posição.

- $T(n) = 1 + 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A comparação linha 24
será executada 1 vez,
pois o número que
gostaríamos está na primeira
posição.

- $T(n) = 1 + 1 + 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A comparação linha 26 será executada 1 vez.

- $T(n) = 1 + 1 + 1 + 1$
- O(n) = c

Análise de Algoritmos Recursivos

- Vamos considerar o algoritmo recursivo abaixo para calcular o fatorial de um número inteiro não-negativo (*n*).

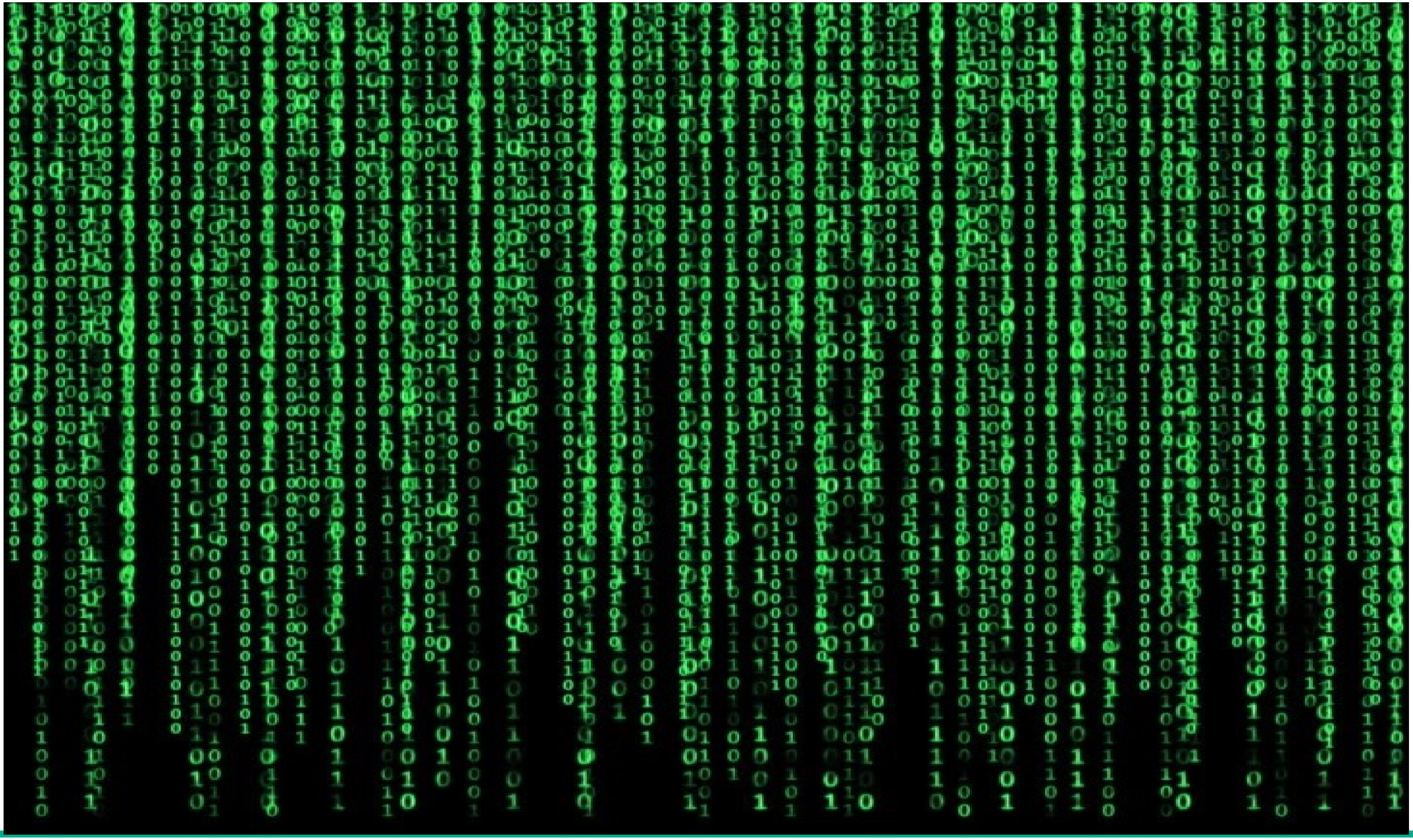
```java
private static int fatorial(int numero) {

    //Quando o numero for 0, o resultado do fatorial sempre vai ser 1
    if (numero == 0) {
        return 1;
    } else {
        return numero * fatorial(numero - 1);
    }
}
```

Métodos de Busca

Método de Busca

- Atualmente, armazenar dados e guardar informações não é problema:
  - Grande capacidade de armazenamento
  - Discos ficaram muito mais baratos
  - *Cloud Computing*
- Porém, recuperar um dado ou informação de qualidade de maneira precisa e eficiente pode ser um problema.
  - Existem diversos métodos de busca.

O Problema da Busca



O Problema da Busca

"Dado um ***conjunto*** de elementos, onde cada um é identificado por uma chave, o objetivo da busca é ***localizar***, nesse conjunto, o elemento que corresponde a uma ***chave específica***."

Busca Sequencial

- Dado o vetor abaixo com 10 números inteiros, é possível localizar algum valor específico?

85	56	72	93	0	24	62	48	12	37

- Exemplo:
  - Existe o valor 62 no vetor?
  - Em qual posição ele está?
  - Como você fez para localizar esse valor em termos de algoritmos?

Busca Sequencial

- Método mais simples de busca;
- É aplicável a vetores **desordenados**;
- Percorre-se **todos** os registros até encontrar a chave da busca.

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

Não

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- Existe o valor 62 no vetor?

SIM!!!!

Valor	85	56	72	93	0	24	62	48	12	37
Índice	0	1	2	3	4	5	6	7	8	9

É o número 62?

Busca Sequencial

- A implementação do método sequencial consiste em:
  - Um **loop** que navega por todo o vetor;
  - Uma **regra de comparação**, que analisa se o valor do vetor naquela posição é igual ao valor procurado

Algoritmo Busca Sequencial

```
Algoritmo buscaSequencial (int X)
    Para i ← 0 até n
        Se vetor[i] == X então
            retorne i
        Fim-se
    Fim-Para
    retorne -1
Fim
```

Complexidade - Busca Sequencial

- Melhor caso
  - O elemento procurado se encontra na primeira posição da lista
  - A comparação será executada apenas uma vez
  - A complexidade portanto é:

$$O(n) = 1$$

Complexidade - Busca Sequencial

- Pior caso
  - O elemento procurado se encontra na última posição da lista ou não se encontra na lista
  - A comparação será uma vez para cada elemento existente no vetor
  - A complexidade portanto é

$$O\ (n) = n$$

Complexidade - Busca Sequencial

- Caso médio
  - O elemento procurado se encontra em uma posição qualquer da lista
  - A comparação será executada a quantidade de vezes representada pela posição do elemento no vetor
  - A complexidade portanto é

$$O(n) = \frac{n+1}{2}$$

Algoritmo Busca Sequencial

- Conseguimos otimizar esse código?

```
Algoritmo buscaSequencial (int X)
    Para i ← 0 até n
        Se vetor[i] == X então
            retorne i
        Fim-se
    Fim-Para
    retorne -1
Fim
```

E se comparassemos com *vetor*[*i*] == *n* ?

Otimizações da Busca Sequencial

- Alteramos o *loop* de busca para que cheque somente se `v[n] == x` (ao invés de checar se `i < n`)
- ***Sentinela***
  - Adicionamos o valor procurado na última posição do vetor;
    - com isso garantimos que em algum momento a condição `v[n] == x` será verdadeira

85	56	72	93	0	24	62	48	12	37	*k*

Sentinela - Otimizações da Busca Sequencial

```
Algoritmo buscaSequencial_Sentinela (int X)
    vetor[ultimaPosicao] ← X
    i ← 0
    Enquanto vetor[i] != X Faça
        i ← i + 1
    Fim-Enquanto
    Se ( i == n ) então
        retorne -1
    senão
        retorne i
        Fim-se
    Fim
```

Lista Ordenada - Otimizações da Busca Sequencial

- Com a lista ordenada, eu tenho a garantia que o próximo valor será obrigatoriamente igual ou superior ao valor atual.
  - com isso, realizamos um teste extra para verificar se $v[n] > x$
  - caso seja verdadeiro, retorna falso

Lista Ordenada - Otimizações da Busca Sequencial

```
Algoritmo buscaSequencial_ListaOrdenada (int X)
    Para i ← 0 até n
        Se vetor[i] == X então
            retorne i
        senão
            Se    vetor[i] > X então
                Retorne -1
            Fim-se
        Fim-se
    Fim-Para
    retorne -1
    Fim
```

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Não.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?

Lista Ordenada - Otimizações da Busca Sequencial

Não.
Sim.

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

É o número 65?
É maior que 65?
Então, finaliza a busca

Lista Ordenada - Otimizações da Busca Sequencial

- Essa é a melhor busca para uma lista ordenada?

Valor	0	12	24	37	48	56	62	79	85	93
Índice	0	1	2	3	4	5	6	7	8	9

Busca Binária

- A busca binária se aproveita do fato do vetor estar ordenado, para realizar a busca em menos iterações.
- Ela usa a estratégia **dividir para conquistar**, para dividir o problema da busca em problemas menores.

Busca Binária

- Buscando o elemento 56

Valores	0	12	24	37	48	56	62	79	85	93
Índice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Início Centro Fim

Centro = $\frac{\text{Inicio + Fim}}{2}$

56	62	79	85	93
5	6	7	8	9

Início Centro Fim

Elemento Encontrado

56	62
5	**6**

Início Centro Fim

Comparação com a Busca Sequencial

- Buscando o 56

Valores	0	12	24	37	48	56	62	79	85	93
Índice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- Devemos buscar um a um
  - Quantas comparações na busca sequencial devemos fazer??
    - 6
  - E na Busca binária?
    - 3

Busca Binária

- Buscando o elemento 73

Valores	0	12	24	37	48	56	62	79	85	93
Índice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Início Centro Fim

56	62	79	85	93
5	6	7	8	9

Início Centro Fim

56	62
5	**6**

Início
Centro
Fim

62

Início = fim = centro
Elemento não encontrado

Comparação com a Busca Sequencial

- Buscando o 72

Valores	0	12	24	37	48	56	62	79	85	93
Índice	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- Devemos buscar um a um
  - Quantas comparações na busca sequencial devemos fazer??
    - 8
  - E na Busca binária?
    - 4

Busca Binária Iterativa

```
Algoritmo buscaBinariaIterativa (int X)
   inicio ← 0
   fim ← n-1
   enquanto inicio <= fim faça
      centro ← (inicio+fim)/2
      se x == v[centro] então
         retorna centro
      Fim-se
      se x < v[centro] então
         fim ← centro-1
      Fim-se
      se x > v[centro]então
         inicio ← centro+1
      Fim-se
   Fim-enquanto
   Retorna -1
fim
```

Conseguimos implementar de outra maneira?

Busca Binária Recursiva

```
Algoritmo buscaBinariaRecursiva (int X, int inicio, int fim)
      centro ← (inicio+fim)/2
      se inicio > fim então
          Retorna -1
      Fim-se
      se x == v[centro] então
         retorna centro
      Fim-se
      se x < v[centro] então
         retorna buscaBinariaRecursiva (x, inicio, centro-1)
      Fim-se
      se x > v[centro] então
         retorna buscaBinariaRecursiva (x, centro + 1, fim)
      Fim-se
   fim
```

Complexidade – Busca Binária

- Melhor caso
  - O elemento procurado se encontra no centro
  - A comparação será executada apenas uma vez
- A complexidade portanto é

$$O\ (n) = 1$$

Complexidade – Busca Binária

- Pior caso
  - O elemento procurado não se encontra na lista
  - Para cada comparação realizada, o tamanho da lista será dividido por 2
- A complexidade portanto é

$$O\ (n) = \log(n)$$

Complexidade – Busca Binária

- Pior Caso
  - No início da primeira iteração, *esq* − *dir* vale aproximadamente n.
  - No início da segunda, vale aproximadamente $n/2$.
  - No início da terceira, $n/4$.
  - No início da (k+1)-ésima, $n/2^k$.
  - Quando k atinge ou ultrapassa log n, o valor da expressão $n/2^k$ fica menor ou igual a 1 e a execução do algoritmo para. Logo, o número de iterações é aproximadamente

$$O\ (n) = \log(n)$$

Complexidade – Busca Binária

- Caso médio
  - O elemento procurado se encontra em uma posição qualquer da lista
  - A comparação será executada entre uma e log(n) vezes com diferentes probabilidades
- A complexidade portanto é

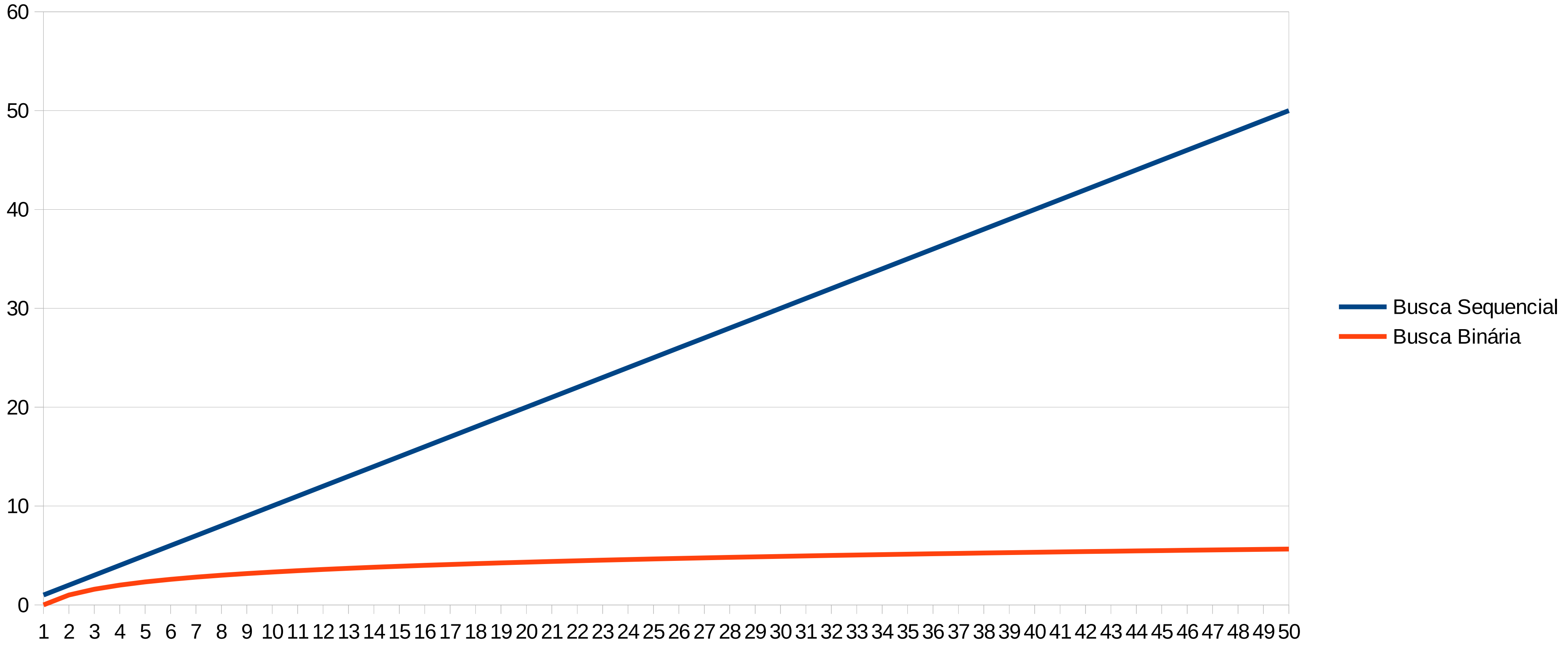
$$O\ (n) = \log(n)$$

Busca Binária

- Alto custo para manter a tabela ordenada.

- Não deve ser usada em aplicações muito dinâmicas.

Busca Sequencial X Busca Binária



Busca por Interpolação

- Ainda assumindo que temos um vetor ordenado;
- Assumindo ainda que as distribuições de valores nesse vetor é uniforme.
  - ex: {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}

Busca por Interpolação

- Se as chaves estiverem uniformemente distribuídas, esse método pode ser ainda mais eficiente do que a busca binária
- Com chaves uniformemente distribuídas, pode-se esperar que *x* esteja aproximadamente na posição:

$$i = esq + dta - esq \; * \; \frac{X - a[esq]}{a[dta] - a[esq]}$$

- sendo que *esq* e *dir* são redefinidos iterativamente como na busca binária.

Busca por Interpolação

```
Algoritmo buscaInterpolacao (int X)
esq ← 0
dir ← n-1
enquanto esq <= dir faça
    i ← esq + dir -(esq *((X - vetor[esq])/ (vetor[dir]-vetor[esq])))
    se x == vetor[i] então
        retorna i
    Fim-se
    se x < vetor[i] então
        dir ← i -1
    Fim-se
    se x > a[i]então
        esq ← i + 1
    Fim-se
Fim-enquanto
Retorna -1
fim
```

Complexidade - Busca por Interpolação

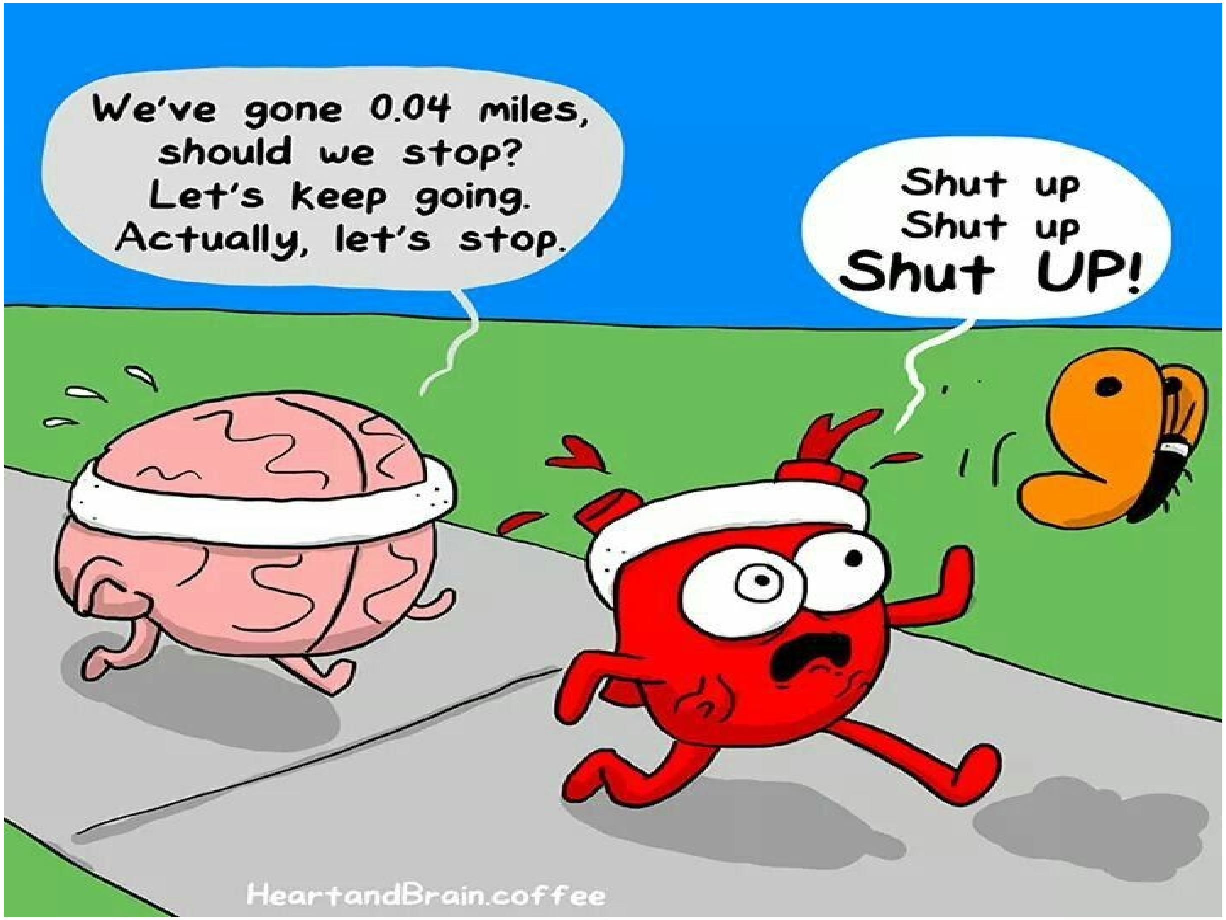
- Complexidade: O(log(log(n))) se as chaves estiverem uniformemente distribuídas:
  - Raramente precisará de mais comparações.
- Se as chaves não estiverem uniformemente distribuídas, a busca por interpolação pode ser tão ruim quanto uma busca sequencial.
- Desvantagens:
  - Em situações práticas, as chaves tendem a se aglomerar em torno de determinados valores e não são uniformemente distribuídas:
    - Exemplo: há uma quantidade maior de nomes começando com “S” do que com “Q”.

Dúvidas

Referência

- DOBRUSHKIN, Vladimir A. Métodos para Análise de Algoritmos. LTC, 03/2012

Exercícios


We've gone 0.04 miles, should we stop? Let's keep going. Actually, let's stop.
Shut up Shut up Shut UP!
HeartandBrain.coffee

Exercícios

1) Implemente a busca sequencial (iterativa e recursiva) e teste com vetores de tamanhos 10, 25 e 50.
2) Implemente a busca binária (iterativa e recursiva) e teste com vetores de tamanhos 10, 25 e 50.

Obrigado!
Bom Dia!
Boa Tarde!
Boa Noite!

bruno.moritani@anhembi.br