

ACH2002

Aula 3

Análise de Algoritmos: introdução

Aulas passadas

- Programação elegante (leiaute, documentação)
- Conceitos básicos de C (revisão – para uso em aula e EPs)
- Análise de complexidade
- Técnicas de desenvolvimento de algoritmos: recursão, programação dinâmica, tentativa e erro (backtracking), algoritmos gulosos, heurísticas, ...
- Algoritmos de ordenação (comparação dos algoritmos e suas complexidades, de tempo e espaço)

Dica:

- Site com resumo dos principais pontos e **exercícios** com respostas!

<http://www.cs.ecu.edu/karl/3300/spr16/Notes/C/>

Aulas de hoje

- Programação elegante (leiaute, documentação)
- Conceitos básicos de C (revisão – para uso em aula e EPs)
- **Análise de complexidade**
- Técnicas de desenvolvimento de algoritmos: recursão, divisão e conquista, programação dinâmica, tentativa e erro (backtracking), algoritmos gulosos, heurísticas, ...
- Algoritmos de ordenação (comparação dos algoritmos e suas complexidades, de tempo e espaço)

O que é um algoritmo?

O que é um algoritmo?

Informalmente (Cormen *et al.*, 2002):

- Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores com **saída**.
- Sequência de passos computacionais que transformam a entrada na saída.

O que é um algoritmo?

Informalmente (Cormen *et al.*, 2002):

- Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores com **saída**.
- Sequência de passos computacionais que transformam a entrada na saída.

Pode existir mais de um algoritmo para resolver um mesmo problema?

O que é um algoritmo?

Informalmente (Cormen *et al.*, 2002):

- Qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores com **saída**.
- Sequência de passos computacionais que transformam a entrada na saída.

Pode existir mais de um algoritmo para resolver um mesmo problema?

SIM!!!

Ex: problema de ordenar um vetor

Exemplo

Problema da ordenação

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$.

Saída: Uma permutação da sequência de entrada a'_1, \dots, a'_n tal que $a'_i \leq a'_j$ para todo $i \leq j$.

Para a instância 3, 42, 17, 2, -1 deste problema, a saída esperada é -1, 2, 3, 17, 42.

Qual a diferença entre...

- Problema:
- Algoritmo:
- Programa:

Qual a diferença entre...

- **Problema:** O QUE queremos resolver
- **Algoritmo:** COMO iremos resolver
 - Descrição de um número finito de passos elementares, que vale para um amplo conjunto de entradas possíveis
 - Existe um algoritmo para todos os problemas?
- **Programa:** implementação de um algoritmo em uma determinada linguagem de programação

Qual a diferença entre...

- **Problema:** O QUE queremos resolver
- **Algoritmo:** COMO iremos resolver
 - Descrição de um número finito de passos elementares, que vale para um amplo conjunto de entradas possíveis
 - Existe um algoritmo para todos os problemas? Não!
- **Programa:** implementação de um algoritmo em uma determinada linguagem de programação

Qual a diferença entre...

- **Problema:** O QUE queremos resolver
- **Algoritmo:** COMO iremos resolver
 - Descrição de um número finito de passos elementares, que vale para um amplo conjunto de entradas possíveis
 - Existe um algoritmo para todos os problemas? Não!
 - Nesta disciplina: foco nos algoritmos (Aula e provas)
- **Programa:** implementação de um algoritmo em uma determinada linguagem de programação

Algoritmos

- Um algoritmo é correto se

Algoritmos

- Um **algoritmo** é **correto** se ele produz a saída esperada para **todas** as entradas possíveis
 - Dizemos que o algoritmo **resolve** o problema
- Três formas distintas para saber se um algoritmo está correto:
 - **Prova de corretude (do algoritmo)** **Vamos ver um pouco nesta disciplina**
 - Teste de software (do programa, não é prova de corretude, é teste empírico)
 - Verificação de software: problema de dada uma especificação e um programa, saber se o programa está correto (ex. de problema sem algoritmo para o caso geral)
- Mas além de ser correto, de quantos recursos ele precisa? (tempo, espaço em memória, número de acesso ao disco, etc...)
 - **Análise (de complexidade) do algoritmo** **Foco desta disciplina**

Análise de algoritmos

- Permite prever os recursos necessários
- Permite comparar diferentes algoritmos para o mesmo problema e decidir qual o melhor (em relação ao uso de recurso(s) mais importante(s) para você...)
 - Qual o mais **eficiente**

Exemplo

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

Ex: Entrada: {3, 5, 16, 17, -1}, 5
Saída: 2

Exemplo

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

```
BUSCASEQUENCIAL( $A, b$ )  
1  for  $i \leftarrow 1$  até  $n$   
2      do if  $a_i = b$   
3          then return  $i$   
4  return  $\perp$ 
```

Obs: vetor em pseudocódigo começa em 1

Exemplo

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

Algoritmo
(pseudocódigo)

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++)
        if(array[i] == n)
            return i;
    return -1;
}
```

Programa
(em C)



Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

$\text{BUSCASEQUENCIAL}(A, b)$

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

**Programa
(em C)**

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++)
        if(array[i] == n)
            return i;
    return -1;
}
```

Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Opção 1: teste empírico

O que fazer?

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

**Programa
(em C)**

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++){
        if(array[i] == n)
            return i;
    }
    return -1;
}
```

Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Opção 1: teste empírico

O que fazer?

- Implementar
- Testar para várias entradas

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

**Programa
(em C)**

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++){
        if(array[i] == n)
            return i;
    }
    return -1;
}
```

Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Opção 1: teste empírico

O que fazer?

- Implementar
- Testar para várias entradas

Desvantagens:

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

Programa (em C)

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++){
        if(array[i] == n)
            return i;
    }
    return -1;
}
```

Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Opção 1: teste empírico

O que fazer?

- Implementar
- Testar para várias entradas

Desvantagens:

- Tem que implementar
- Normalmente não dá para testar todas as entradas possíveis

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

**Programa
(em C)**

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++){
        if(array[i] == n)
            return i;
    }
    return -1;
}
```


Como provar que esse algoritmo está correto? (usando pseudo-código ou programa)

Opção 2: prova formal de corretude

Problema da busca

Entrada: Uma sequência de n valores a_1, \dots, a_n em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou \perp caso contrário.

**Algoritmo
(pseudocódigo)**

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

**Programa
(em C)**

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
    int i;
    for(i = 0; i < size; i++){
        if(array[i] == n)
            return i;
    }
    return -1;
}
```

Prova de corretude de algoritmos

- Vamos aprender algumas técnicas durante a disciplina
- Por hora, vamos aprender a usar a noção de “**invariantes**” para algoritmos **iterativos**:
 - Algoritmos baseados em **loops** (iterações)
 - Usualmente usando **for**, **while**

Invariantes

- **Invariante**: relação entre os valores das variáveis envolvidas dentro do laço, que é sempre **verdadeira no início de cada iteração**
- Interessante explicitar na forma de comentário

Ex:

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

Invariantes

- O que isso tem a ver com prova de corretude?

- Ex:

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

Invariantes

- O que isso tem a ver com prova de corretude?

- Podemos fazer uma **prova por indução** para mostrar que o x retornado SEMPRE será o valor máximo para TODA entrada v e n ;-)

```
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

Prova por indução (fraca)

- Objetivo: prova que uma determinada **propriedade (P)** é válida para **todos** os elementos de um conjunto **potencialmente infinito**
- 3 partes:
 - **Base da indução:** prova/mostra que P é verdadeira para o primeiro elemento (0, ou 1, ou 2, ...) desse conjunto
 - **Hipótese da indução:** assume que P é verdadeira para o n-ésimo elemento desse conjunto (é na verdade o enunciado de P)
 - **Passo da indução:** usa a hipótese para provar que P é verdadeira para o (n+1)-ésimo elemento desse conjunto

Prova por indução (fraca)

- Objetivo: prova que uma determinada **propriedade (P)** é válida para **todos** os elementos de um conjunto **potencialmente infinito**
- 3 partes:
 - **Base da indução:** prova/mostra que P é verdadeira para o primeiro elemento (0, ou 1, ou 2, ...) desse conjunto
 - **Hipótese da indução:** assume que P é verdadeira para o n -ésimo elemento desse conjunto (é na verdade o enunciado de P)
 - **Passo da indução:** usa a hipótese para provar que P é verdadeira para o $(n+1)$ -ésimo elemento desse conjunto

ou $n-1$

ou n

Prova por indução

Ex: prova por indução que $1 + 2 + 3 + \cdots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Base: P é verdadeira para $n = 1$: $1 = \frac{1(1+1)}{2}$

Hipótese: P é verdadeira para um dado n qualquer: $1 + 2 + 3 + \cdots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Passo: Dado que P vale para n , P é verdadeira para $n+1$:

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + n + 1 &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{n^2 + n + 2n + 2}{2} \\ &= \frac{n^2 + 3n + 2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

Prova por indução

Ex: prova por indução que $1 + 2 + 3 + \cdots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Base: P é verdadeira para $n = 1$: $1 = \frac{1(1+1)}{2}$

Hipótese: P é verdadeira para um dado n qualquer: $1 + 2 + 3 + \cdots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$

Passo: Dado que P vale para n , P é verdadeira para $n+1$:

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + n + 1 &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{n^2 + n + 2n + 2}{2} \\ &= \frac{n^2 + 3n + 2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

Note que um loop é uma série...

Prova de corretude por indução

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

x, o valor retornado, é o elemento máximo do vetor

Base:

Hipótese:

Passo da indução:

Prova de corretude por indução

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

Base: no início da primeira iteração ($j = 1$), x é o elemento máximo de $v[0]$

Hipótese:

x , o valor retornado, é o elemento máximo do vetor

Passo da indução:

Prova de corretude por indução

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

x, o valor retornado, é o elemento máximo do vetor

Base: no início da primeira iteração ($j = 1$), x é o elemento máximo de $v[0]$

Hipótese: assuma que é verdadeiro para um $j < n$, que x é o elemento máximo de $v[0..j-1]$

Passo da indução:

Prova de corretude por indução

```
int Max (int v[], int n) {  
    int j, x;  
    x = v[0];  
    for (j = 1; j < n; j++)  
        /* x é um elemento máximo de v[0..j-1] */  
        if (x < v[j]) x = v[j];  
    return x;  
}
```

x, o valor retornado, é o elemento máximo do vetor

Base: no início da primeira iteração ($j = 1$), x é o elemento máximo de $v[0]$

Hipótese: assumo que é verdadeiro para um $j < n$, que x é o elemento máximo de $v[0..j-1]$

Passo da indução: se x é o elemento máximo de $v[0..j-1]$ no início da iteração para o valor j , então na próxima instrução (que é única no loop):

- se $x < v[j]$, x será substituído por $v[j]$, o que o torna o elemento máximo de $v[0..j]$ no início da próxima iteração (valor $j+1$)
- se $x \geq v[j]$, x não mudará de valor, pois continua sendo o elemento máximo de $v[0..j]$ no início da próxima iteração (valor $j+1$)

Como seria a prova de corretude para esse nosso problema?

- Qual a invariante?
- O que você quer provar?

BUSCASEQUENCIAL(A, b)

1 **for** $i \leftarrow 1$ até n

2 **do if** $a_i = b$

3 **then return** i

4 **return** \perp

Como seria a prova de corretude para esse nosso problema?

- Qual a invariante?
- O que você quer provar? Que o programa retorna i se b for o i -ésimo elemento, e nil c.c.

b não ocorre em a_1, \dots, a_{i-1}

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

Base:

Hipótese:

Passo:

Como seria a prova de corretude para esse nosso problema?

- Qual a invariante?
- O que você quer provar? Que o programa retorna i se b for o i -ésimo elemento, e nil c.c.

b não ocorre em a_1, \dots, a_{i-1}

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

Base: quando $i = 1$, não há elementos entre a_1 e a_0 , logo b não pode ocorrer lá.

Hipótese:

Passo:

Como seria a prova de corretude para esse nosso problema?

- Qual a invariante?
- O que você quer provar? Que o programa retorna i se b for o i -ésimo elemento, e nil c.c.

b não ocorre em a_1, \dots, a_{i-1}

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

Base: quando $i = 1$, não há elementos entre a_1 e a_0 , logo b não pode ocorrer lá.

Hipótese: b não ocorre em a_1, \dots, a_{i-1}
(e o algoritmo dá a resposta correta)

Passo:

Como seria a prova de corretude para esse nosso problema?

- Qual a invariante?
- O que você quer provar? Que o programa retorna i se b for o i -ésimo elemento, e nil c.c.

b não ocorre em a_1, \dots, a_{i-1}

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

Base: quando $i = 1$, não há elementos entre a_1 e a_0 , logo b não pode ocorrer lá.

Hipótese: b não ocorre em a_1, \dots, a_{i-1}
(e o algoritmo dá a resposta correta)

Passo: O que acontece com $i+1$? Pela hipótese, no passo i é verdade que b não ocorre em a_1, \dots, a_{i-1} . A próxima instrução (ainda no passo i) é comparar b com a_i .

- Se forem iguais, pela hipótese de indução, é a primeira vez que isso acontece, e a execução da linha 3 fará com que a linha 2 não volte a ser executada (**invariante continuou válido**, pois não há outra iteração) e o **algoritmo dá a resposta correta**

- se forem diferentes, a linha 3 não é executada, há uma nova iteração (se $i \leq n$), e o **invariante é válido** (pois o b não foi achado em a_1, \dots, a_i)

Por fim, se a linha 4 é executada, é porque a linha 3 não foi executada nenhuma vez, e na última iteração $i = n$. Logo, b não foi encontrado em a_1, \dots, a_n . Logo, aqui o algoritmo também dá a **resposta correta**.

Exercício

- Prove a corretude desse algoritmo, que assume o vetor A está ordenado crescentemente:

BUSCABINARIA(A, b)

```
1   $i \leftarrow 1$ 
2   $j \leftarrow |A|$ 
3  while  $i \leq j$ 
4      do  $m \leftarrow \lfloor \frac{j+i}{2} \rfloor$ 
5          if  $b < a_m$ 
6              then  $j \leftarrow m - 1$ 
7          else if  $b > a_m$ 
8              then  $i \leftarrow m + 1$ 
9          else return  $m$ 
10 return  $\perp$ 
```

Solução no cap 3 da apostila
do Prof. Márcio Moreto

Como analisar um algoritmo

- Opção 1: testes empíricos – como fazer?

Como analisar um algoritmo

- Opção 1: testes empíricos – como fazer?
 - Implementar o algoritmo
 - Testar com diferentes entradas
 - Diferentes tamanhos
 - Medir o recurso (como?)
 - Para cada tamanho de entrada deveríamos testar várias vezes (para diluir o efeito de outras variáveis, como uso compartilhado da CPU)
 - Analisar o uso do recurso em função do tamanho

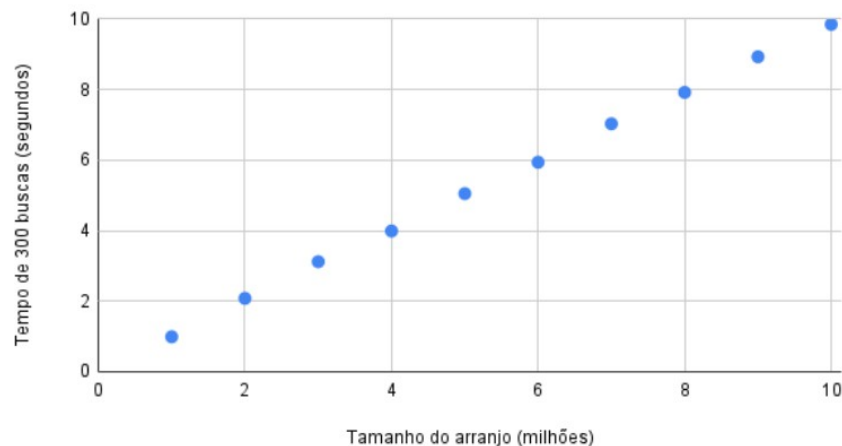
Nosso exemplo

BUSCASEQUENCIAL(A, b)

```
1  for  $i \leftarrow 1$  até  $n$ 
2      do if  $a_i = b$ 
3          then return  $i$ 
4  return  $\perp$ 
```

tamanho do arranjo em milhões	tempo de 300 buscas em segundos
1	0,99
2	2,08
3	3,12
4	3,99
5	5,05
6	5,94
7	7,03
8	7,92
9	8,93
10	9,85

Busca Simples



Busca Simples **Fazendo uma regressão linear:**

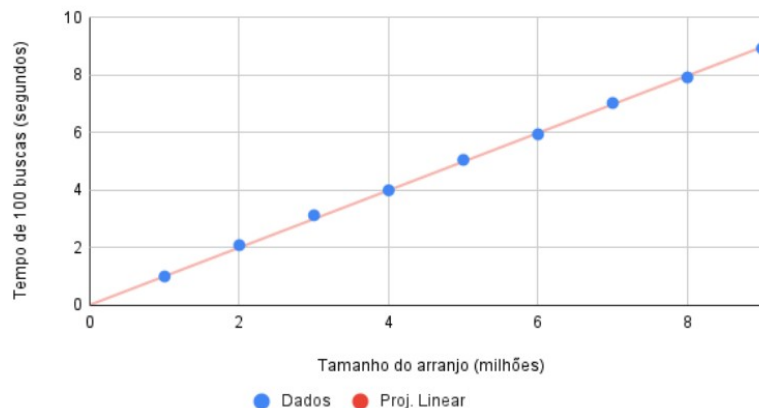


Figura 2.2: Gráfico ilustrando a hipótese de que o tempo de processamento da busca sequencial segue a função linear $t(x) = 0,997x$.

Prevendo o tempo para outros tamanhos...

Que legal!!!

tamanho do arranjo em milhões	tempo previsto	tempo observado
11	10,97	10,87
12	11,97	11,81
13	12,97	12,78
14	13,96	14,00
15	14,96	14,74

E para busca binária?

BUSCABINARIA(A, b)

```
1   $i \leftarrow 1$ 
2   $j \leftarrow |A|$ 
3  while  $i \leq j$ 
4      do  $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$ 
5          if  $b < a_m$ 
6              then  $j \leftarrow m - 1$ 
7          else if  $b > a_m$ 
8              then  $i \leftarrow m + 1$ 
9          else return  $m$ 
10 return  $\perp$ 
```


tamanho do arranjo em milhões	tempo de 300 buscas em segundos
1	0,00
2	0,00
3	0,00
4	0,00
5	0,00
6	0,00
7	0,00
8	0,00
9	0,00
10	0,00

NADA????

E para busca binária?

BUSCABINARIA(A, b)

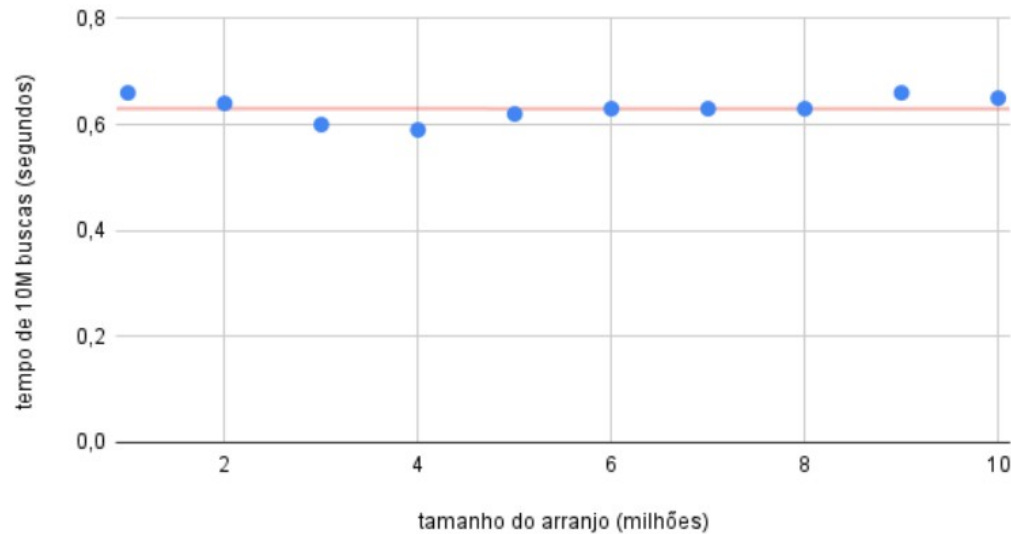
```
1   $i \leftarrow 1$ 
2   $j \leftarrow |A|$ 
3  while  $i \leq j$ 
4      do  $m \leftarrow \lfloor \frac{i+j}{2} \rfloor$ 
5          if  $b < a_m$ 
6              then  $j \leftarrow m - 1$ 
7          else if  $b > a_m$ 
8              then  $i \leftarrow m + 1$ 
9          else return  $m$ 
10 return  $\perp$ 
```



tamanho do arranjo em milhões	tempo de 10M buscas em segundos
1	0,66
2	0,64
3	0,60
4	0,59
5	0,62
6	0,63
7	0,63
8	0,63
9	0,66
10	0,65

E para busca binária?

Busca Binária



ões	tempo de 10M buscas em segundos
	0,66
	0,64
	0,60
	0,59
	0,62
	0,63
	0,63
	0,63
	0,66
	0,65

Figura 2.3: Gráfico ilustrando a hipótese de que o tempo de processamento da busca binária segue a função constante $t(x) = 0,63$.

Tempo previsto... será?

tamanho do arranjo em milhões	tempo previsto	tempo observado
20	0,63	0,68
100	0,63	0,75
500	0,63	0,81
1000	0,63	0,89

Tabela 2.2: Tempo de processamento previsto e observado para tamanhos maiores de arranjos.

E se eu aumentar o tamanho exponencialmente?

tamanho do arranjo em milhões	tempo de 10M buscas em segundos
1	0,59
2	0,62
4	0,69
8	0,69
16	0,74
32	0,76
64	0,79
128	0,83
256	0,96
512	1,01

$$t(2^y) = a.y + b$$

$$t(x) = a.\log_2(x) + b = 0,043.\log_2(x) + 0,529$$

Como analisar um algoritmo

- Opção 1: testes empíricos – **desvantagens:**
 - além do trabalho que dá (lembre do que precisa ser feito):
 - tem que ajustar uma curva
 - variabilidade devido à execução em diferentes máquinas, SO, linguagem de programação, memória disponível
 - pode ser demorado....

Como analisar um algoritmo

- Opção 2:

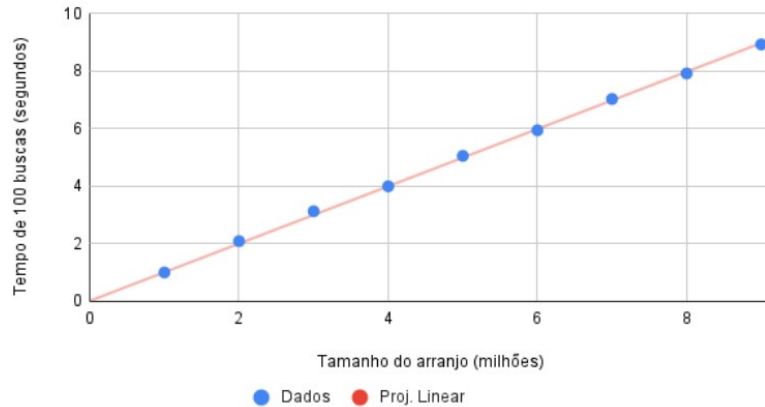
Como analisar um algoritmo

- Opção 2: análise formal, matemática ! ;-)

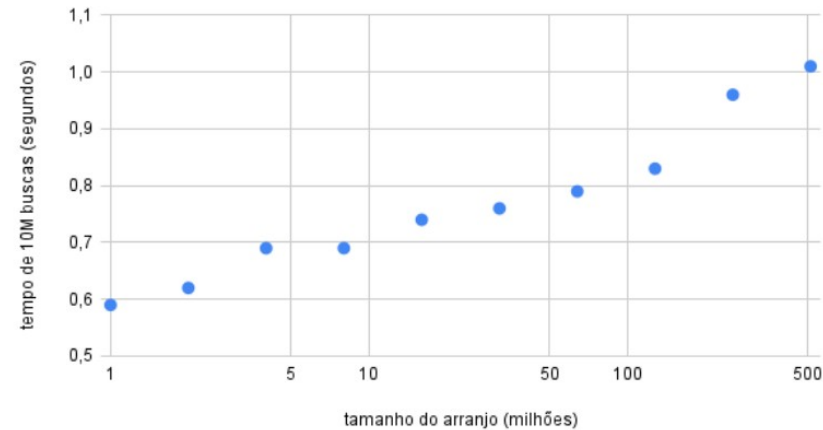
Análise assintótica (motivação)

- Primeiro: consciência que a complexidade (tempo, memória, etc) normalmente depende do tamanho da entrada

Busca Simples



Busca Binária



$$t(x) = a \cdot \log_2(x) + b = 0,043 \cdot \log_2(x) + 0,529$$

Figura 2.2: Gráfico ilustrando a hipótese de que o tempo de processamento da busca sequencial segue a função linear $t(x) = 0,997x$.

Complexidade em função do tamanho

- Tamanho de entrada (n):

- depende do problema estudado
- maioria dos problemas: número de itens de entrada
- exemplo da busca (quantidade de elementos do arranjo)

- Tempo de execução:

- quantidade de operações primitivas ou etapas executadas para uma determinada entrada
- vamos considerar que cada linha i leva um tempo constante c_i

➤ Função de custo de um algoritmo

- engloba o custo de tempo de cada instrução e o número de vezes que cada instrução é executada
- Exemplo: *insertion-sort(A)* (entrada: array A que tem tamanho n)

Insertion sort (inserção direta)

- Como você ordena as cartas do baralho, **se pegá-las uma de cada vez?**

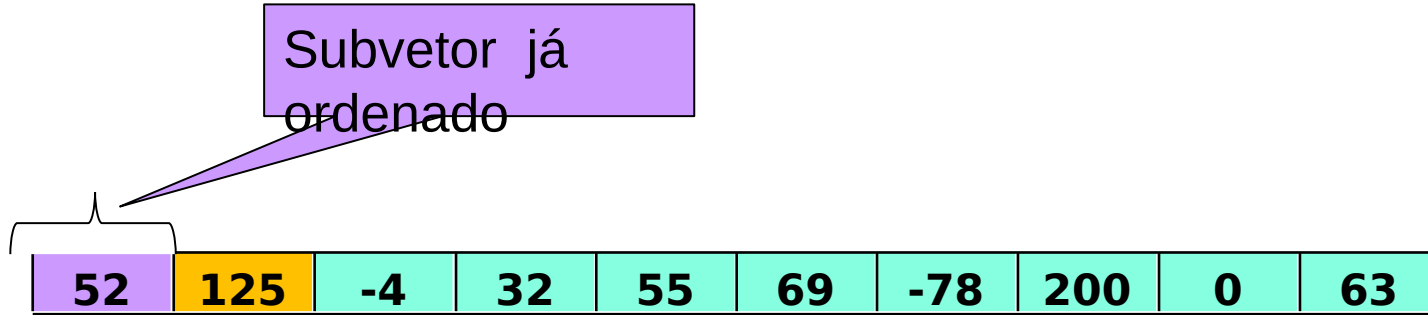


Insertion sort (inserção direta)

- Percorre o array (do início) e, em cada passo:
 - Aumenta a parte ordenada do array em uma posição, inserindo um novo elemento na posição correta **e deslocando os demais para a direita**

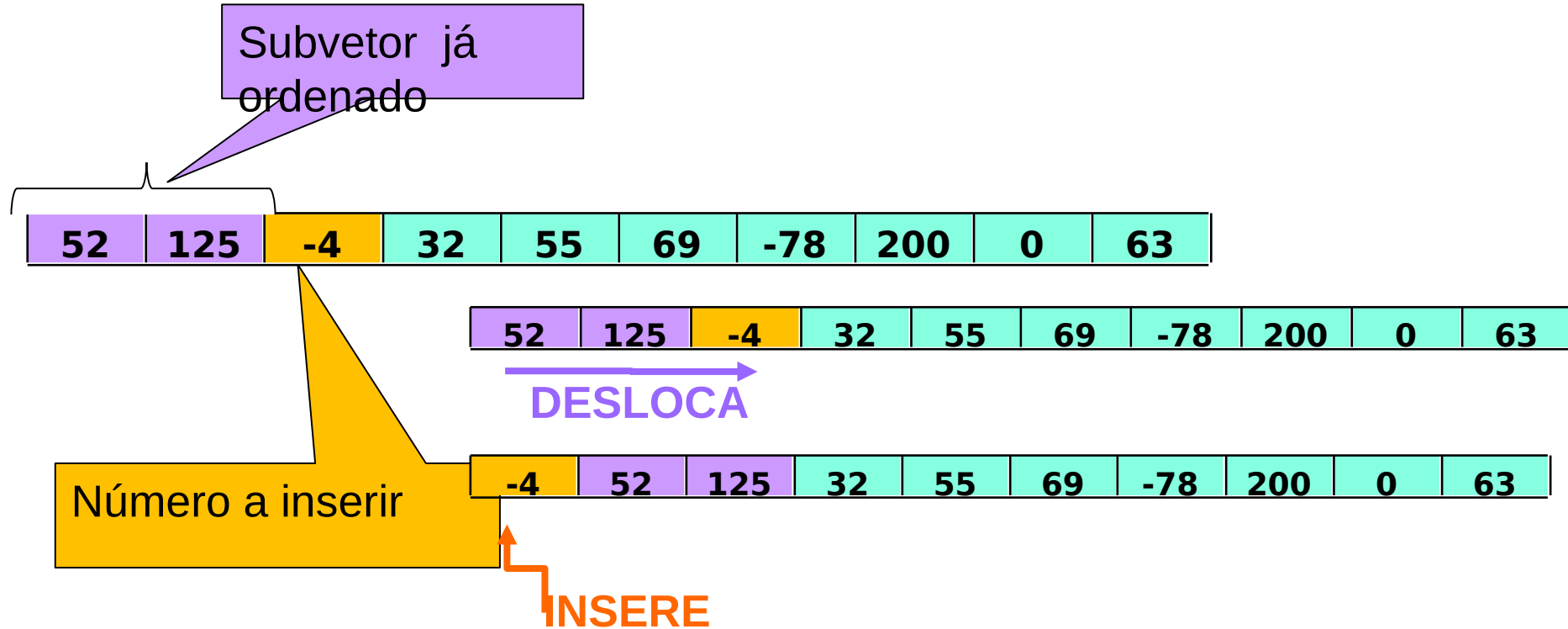
Insertion sort (inserção direta)

1)



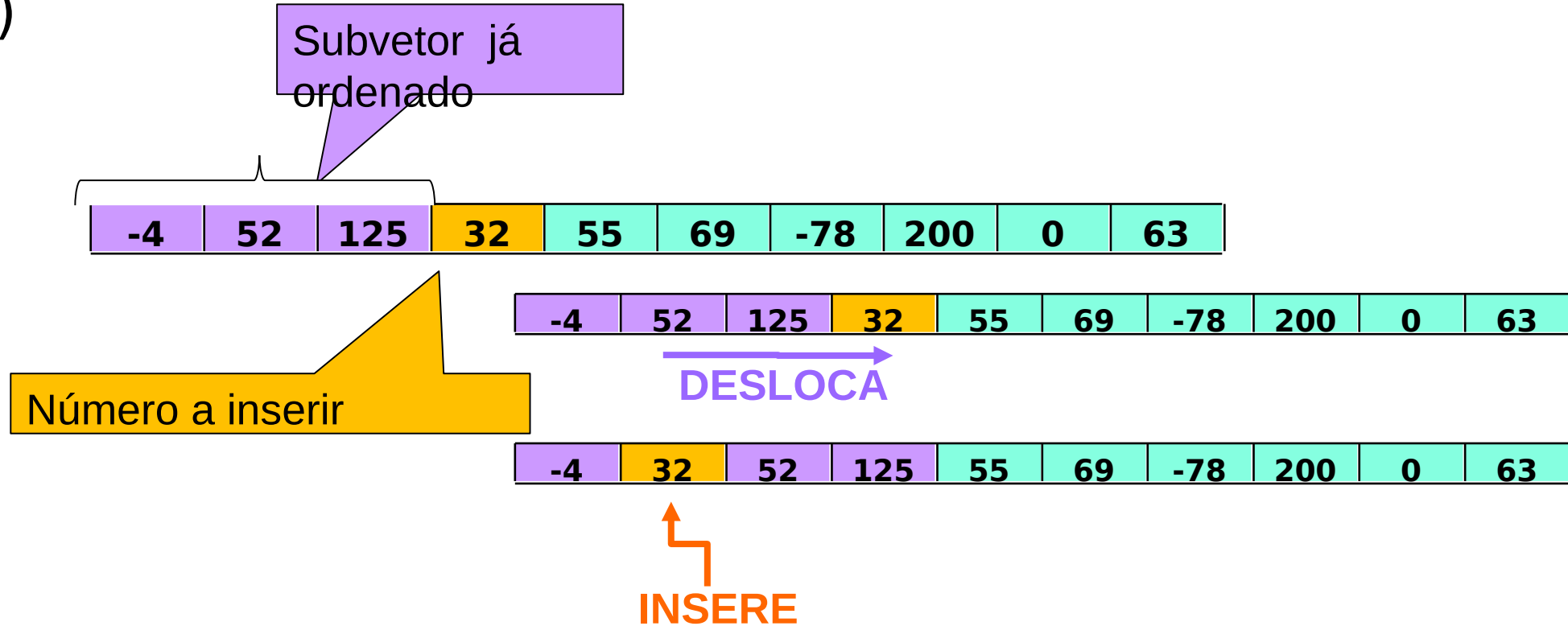
Insertion sort (inserção direta)

2)



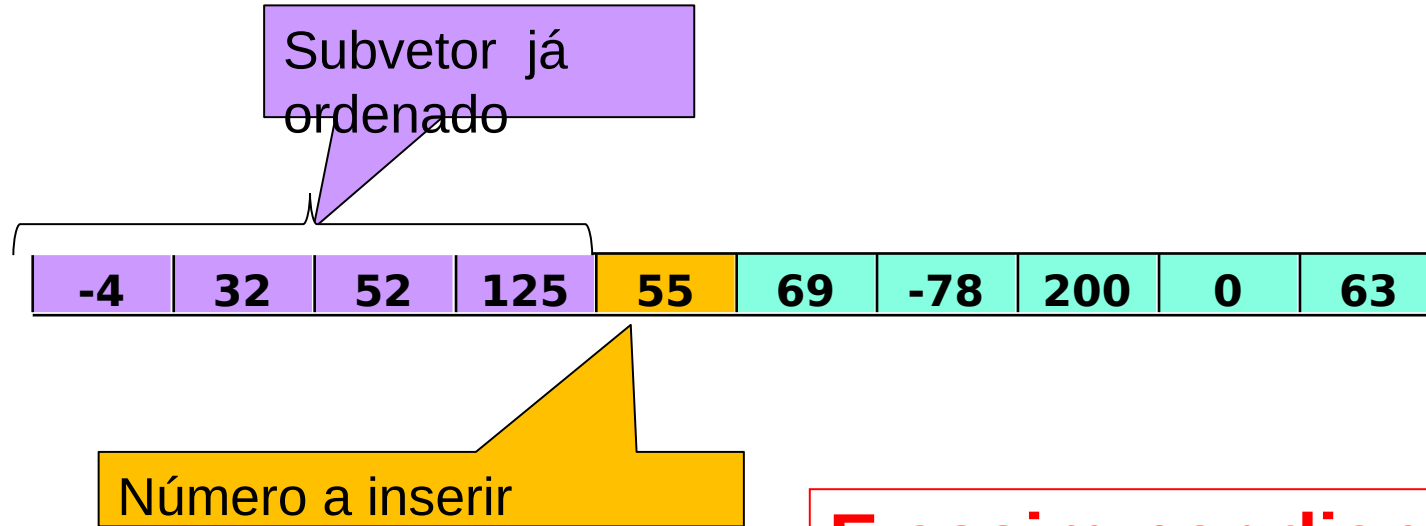
Insertion sort (inserção direta)

3)



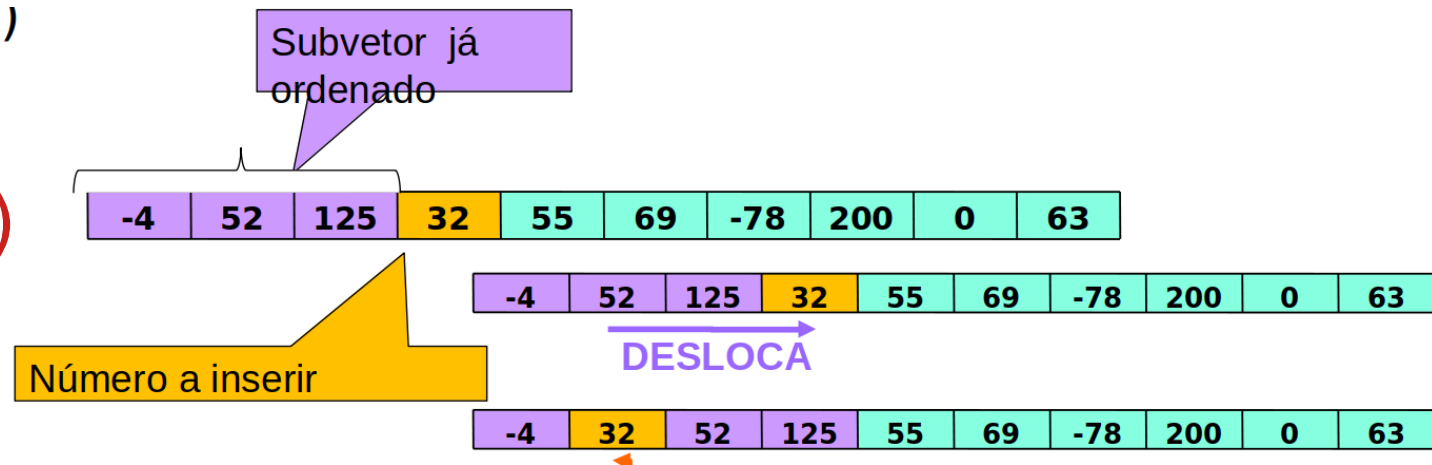
3)

Insertion sort (inserção direta)



E assim por diante...

Insertion sort (inserção direta)



```
1 para j = 2 até tamanho[A] faça
2   chave = A[j] // "número a inserir"
3   // ordenando elementos à esquerda
4   i = j - 1
5   enquanto i > 0 e A[i] > chave faça
6     A[i+1] = A[i]
7     i = i - 1
8   fim enquanto
9   A[i+1] = chave
10 fim para
```

➤ Função de custo de um algoritmo

- engloba o custo de tempo de cada instrução e o número de vezes que cada instrução é executada

- Exemplo: *insertion-sort*(A) (entrada: array A que tem tamanho n)

custo vezes

1	para j = 2 até tamanho[A] faça	c_1	n	
2	chave = A[j] // “número a inserir”	c_2	n-1	
3	// ordenando elementos à esquerda	0	n-1	
4	i = j - 1	c_4	n-1	
5	enquanto i > 0 e A[i] > chave faça	c_5	$\sum_{j=2}^n t_j$	→ t_j – número de vezes que a linha é executada para um dado j (depende do j)
6	A[i+1] = A[i]	c_6	$\sum_{j=2}^n (t_j - 1)$	
7	i = i - 1	c_7	$\sum_{j=2}^n (t_j - 1)$	
8	fim enquanto			
9	A[i+1] = chave	c_8	n-1	
10	fim para			

➤ Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução

➤
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

```
1 para j = 2 até tamanho[A] faça
```

```
2     chave = A[j]
```

```
3 // ordenando elementos à esquerda
```

```
4     i = j - 1
```

```
5     enquanto i > 0 e A[i] > chave faça
```

```
6         A[i+1] = A[i]
```

```
7         i = i - 1
```

```
8     fim enquanto
```

```
9     A[i+1] = chave
```

```
10 fim para
```

custo vezes

c_1 n

c_2 n-1

0 n-1

c_4 n-1

c_5 $\sum_{j=2}^n t_j \longrightarrow t_j$ – número de vezes

c_6 $\sum_{j=2}^n (t_j - 1)$

c_7 $\sum_{j=2}^n (t_j - 1)$

que a linha é executada para um dado j (depende do j)

c_8 n-1

➤ Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução

➤
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

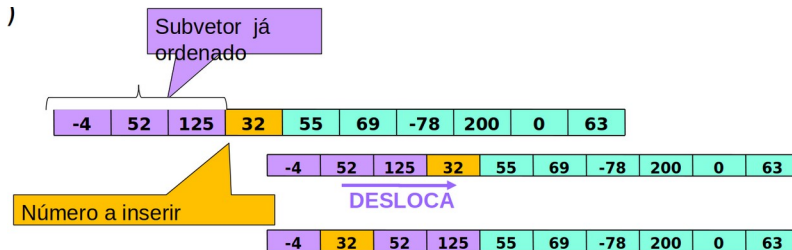
```

1  para j = 2 até tamanho[A] faça
2    chave = A[j]
3    // ordenando elementos à esquerda
4    i = j - 1
5    enquanto i > 0 e A[i] > chave faça
6      A[i+1] = A[i]
7      i = i - 1
8    fim enquanto
9    A[i+1] = chave
10 fim para
  
```

custo vezes

c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n-1$

Melhor caso ?



➤ Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução

➤
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

```

1 para j = 2 até tamanho[A] faça
2     chave = A[j]
3     // ordenando elementos à esquerda
4     i = j - 1
5     enquanto i > 0 e A[i] > chave faça
6         A[i+1] = A[i]
7         i = i - 1
8     fim enquanto
9     A[i+1] = chave
10 fim para
  
```

custo vezes

c_1 n

c_2 n-1

0 n-1

c_4 n-1

$c_5 \sum_{j=2}^n t_j$

$c_6 \sum_{j=2}^n (t_j - 1)$

$c_7 \sum_{j=2}^n (t_j - 1)$

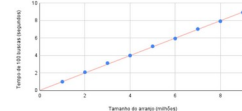
c_8 n-1

Melhor caso: vetor já ordenado ($A[i] \leq$ chave na linha 5 $\rightarrow t_j=1$ para $j=2,3,\dots,n$)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$$

$$(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Tempo de execução, neste caso, pode ser expresso como $an + b$ para constantes **a** e **b** que dependem dos custos de instrução $c_i \rightarrow$ **função linear de n**



➤ Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução

➤
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

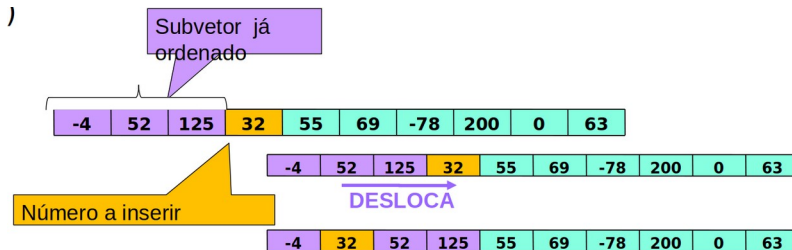
```

1 para j = 2 até tamanho[A] faça
2   chave = A[j]
3   // ordenando elementos à esquerda
4   i = j - 1
5   enquanto i > 0 e A[i] > chave faça
6     A[i+1] = A[i]
7     i = i - 1
8   fim enquanto
9   A[i+1] = chave
10 fim para
  
```

custo vezes

c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n-1$

PIOR caso ?



➤ Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução

➤
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$$

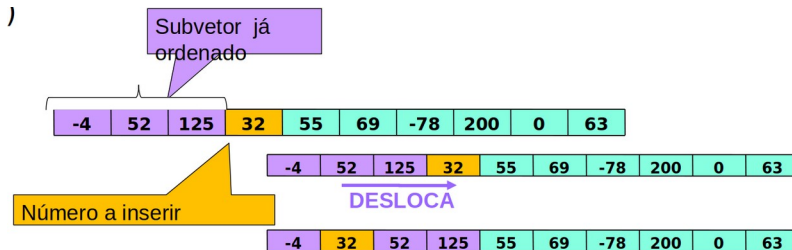
```

1 para j = 2 até tamanho[A] faça
2   chave = A[j]
3   // ordenando elementos à esquerda
4   i = j - 1
5   enquanto i > 0 e A[i] > chave faça
6     A[i+1] = A[i]
7     i = i - 1
8   fim enquanto
9   A[i+1] = chave
10 fim para
  
```

custo vezes

c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^n t_j$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$\sum_{j=2}^n (t_j - 1)$
c_8	$n-1$

PIOR caso ?



Pior caso: vetor em ordem inversa (deve comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado $A[0 \dots j-1]$)
 ➔ $t_j = j$ para $j=2, 3, \dots, n$

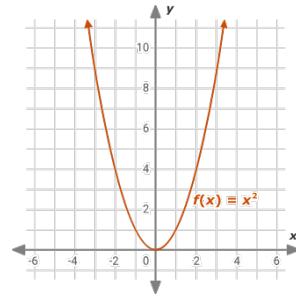
- Tempo de execução do algoritmo = soma dos tempos de execução para cada instrução
- $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1)$
- **Pior caso:** vetor em ordem inversa (deve comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado $A[j \dots j-1]$ → $t_j = j$ para $j=2, 3, \dots, n$)

$$\sum_{j=2}^n (j) = \frac{n(n-1)}{2} + 1 \quad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2} + c_7 \frac{n(n-1)}{2} + c_8(n-1) =$$

$$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - (c_2 + c_4 + c_5 + c_8)$$

Tempo de execução, neste caso, pode ser expresso como $an^2 + bn + c$ para constantes **a**, **b** e **c** que dependem dos custos de instrução c_i → **função quadrática** de n



Análise de algoritmos

- Em geral:
 - tempo de execução de um algoritmo é fixo para uma determinada entrada
 - analisamos apenas o **pior caso** dos algoritmos:
 - é um limite superior sobre o tempo de execução de qualquer entrada;
 - pior caso ocorre com muita frequência para alguns algoritmos. Exemplo: busca de registro inexistente em um banco de dados;
 - muitas vezes, o **caso médio** é quase tão ruim quanto o pior caso

- Nas análises anteriores, foram feitas algumas simplificações em relação às constantes, chegando à função linear e à função quadrática
- **Taxa de crescimento** ou **ordem de crescimento**:
 - considera apenas o termo inicial de uma fórmula (exemplo: an^2), pois os termos de mais baixa ordem são relativamente insignificantes para grandes valores de n ;
 - ignora o coeficiente constante do termo inicial também por ser menos significativo para grandes entradas;
 - Portanto, dizemos que: a ordenação por inserção, por exemplo, tem um tempo de execução do pior caso igual a $\Theta(n^2)$ (*lê-se “theta de n ao quadrado”*);
 - Em geral, consideramos um algoritmo mais eficiente que outro se o tempo de execução do seu pior caso apresenta uma ordem de crescimento mais baixa.

Exercícios (Indução matemática)

1. Prove que $1^2 + 2^2 + 3^2 + \dots + n^2 = (2n^3 + 3n^2 + n)/6$, $\forall n \geq 1$
2. Prove que $1 + 3 + 5 + \dots + 2n - 1 = n^2$, $\forall n \geq 1$
3. Prove que $1^3 + 2^3 + 3^3 + \dots + n^3 = (n^4 + 2n^3 + n^2)/4$, $\forall n \geq 1$
4. Prove que $1^3 + 3^3 + 5^3 + \dots + (2n - 1)^3 = 2n^4 - n^2$, $\forall n \geq 1$
5. Prove que $1 + 2 + 2^2 + 2^3 + \dots + 2^n = 2^{n+1} - 1$, $\forall n \geq 0$
6. Prove que $2^n \geq n^2$, $\forall n \geq 4$
7. Prove que $\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \dots + \frac{1}{2n-1} - \frac{1}{2n} = \frac{1}{n+1} + \frac{1}{n+2} + \dots + \frac{1}{2n}$
8. Prove que a soma dos cubos de três números naturais positivos sucessivos é divisível por 9.
9. Prove que todo número natural $n > 1$ pode ser escrito como o produto de primos (indução forte).
10. Prove que todo número natural positivo pode ser escrito como a soma de diferentes potências de 2 (indução forte).

Referências

- Apostila do Prof Márcio Moretto – Cap 1, 2, 3
- Paulo Feofiloff. Algoritmos em C. Cap 1.2 (**tem exercícios!!!**) <https://www.ime.usp.br/~pf/algoritmos-livro/>