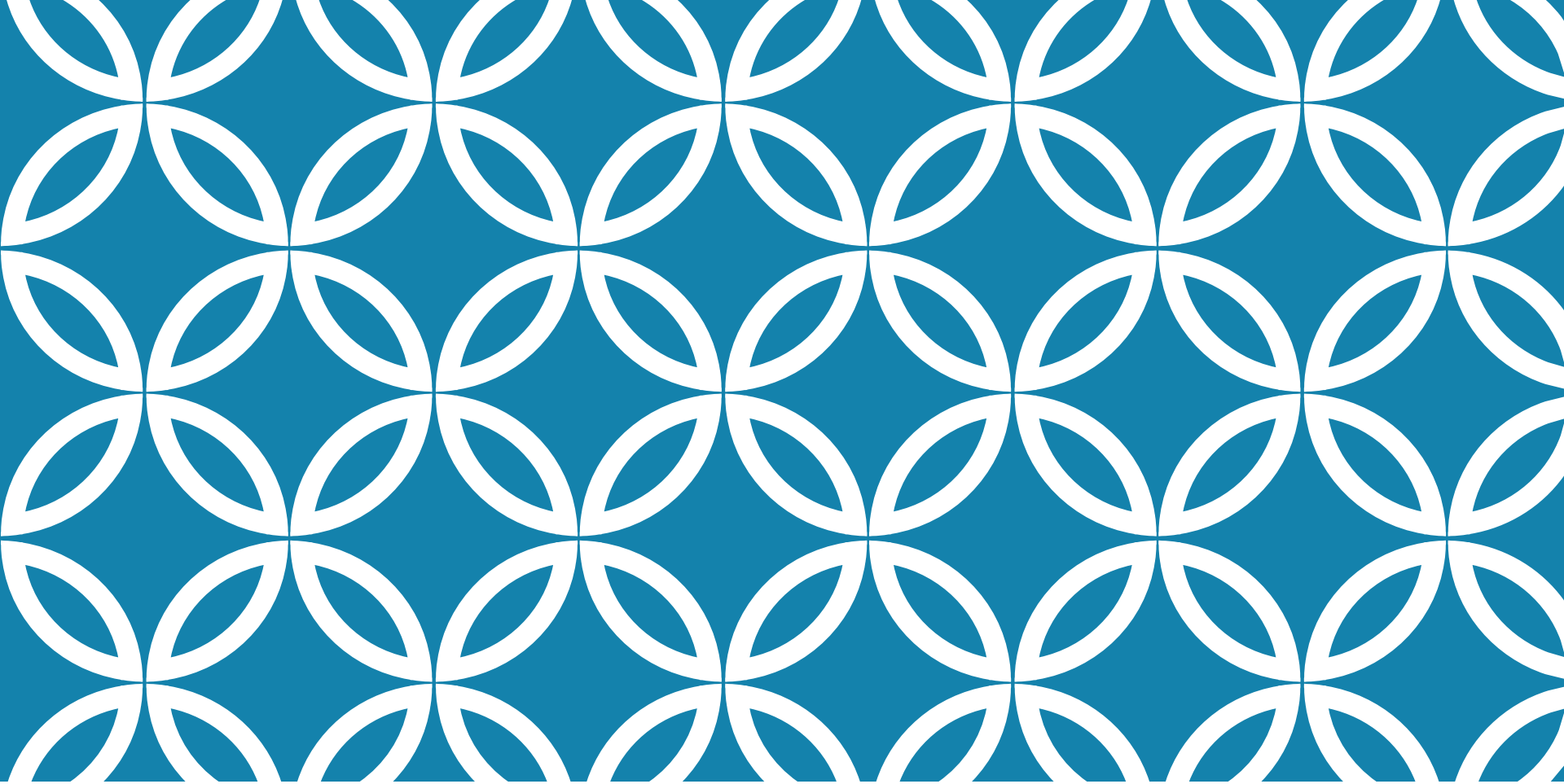


COMPLEXIDADE (INGÊNUA) DE ALGORITMOS ITERATIVOS

Prof. André Vignatti

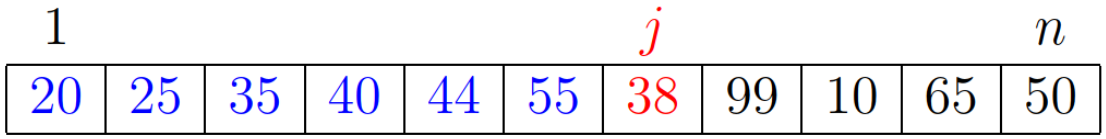
PROBLEMA DA ORDENAÇÃO

Entrada: vetor $A[1 \ldots n]$

Saída: vetor $A[1 \ldots n]$ rearranjado em ordem crescente

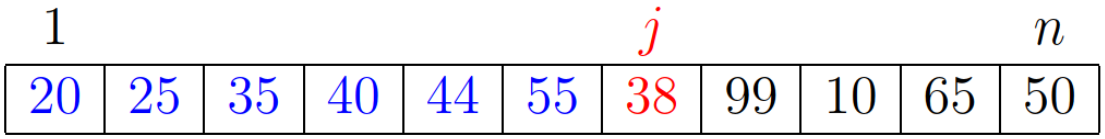
INSERTION SORT - IDEIA

- O subvetor $A[1 \ldots j-1]$ está ordenado

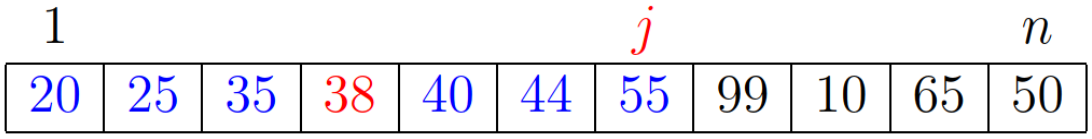


INSERTION SORT - IDEIA

- O subvetor $A[1 \ldots j-1]$ está ordenado

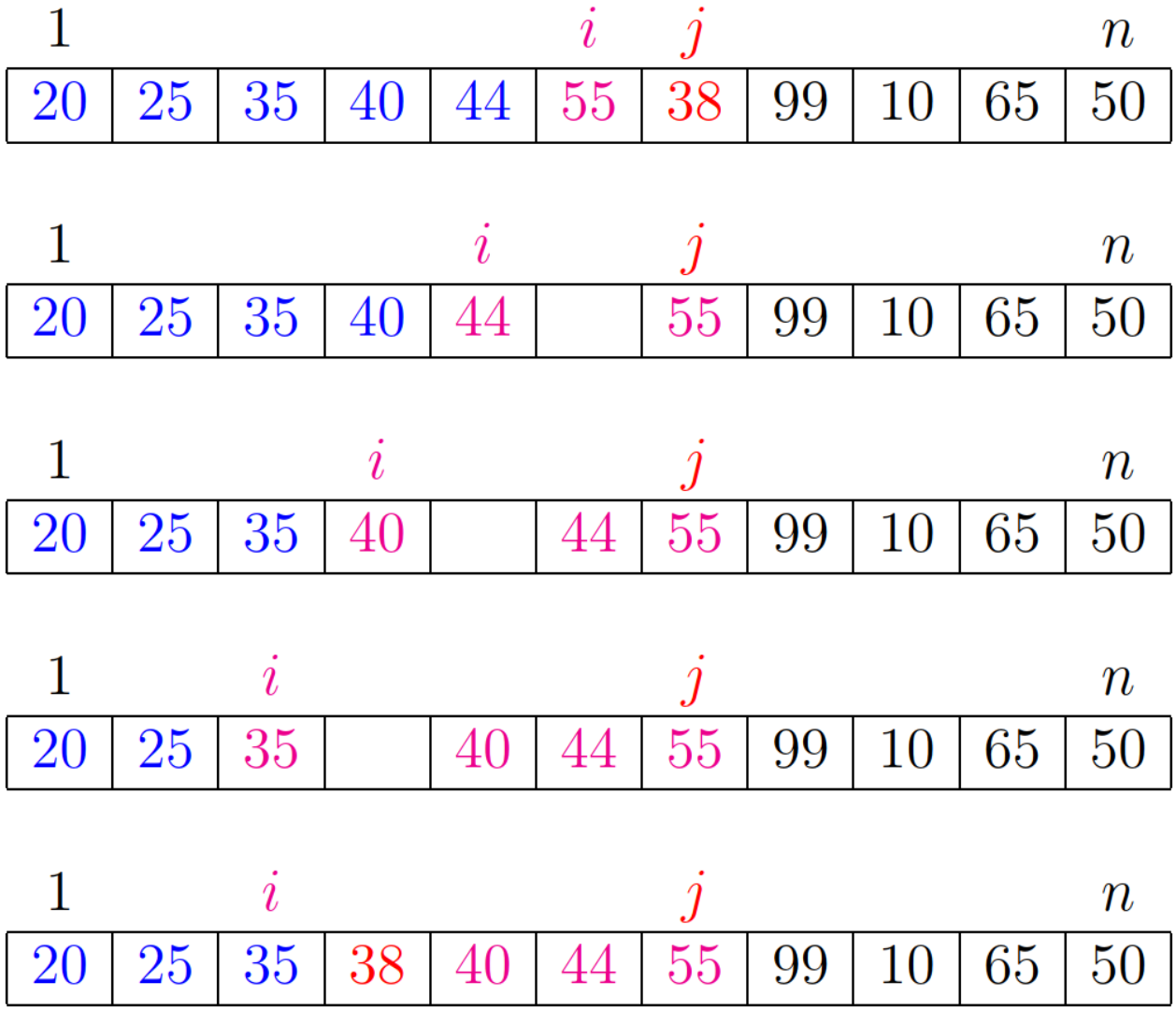


Queremos inserir a $chave = 38 = A[j]$ em $A[1 \ldots j-1]$ para que tenhamos:



Agora $A[1 \ldots j]$ está ordenado

$chave = 38$


1 i j n
20 25 35 40 44 55 38 99 10 65 50
1 i j n
20 25 35 40 44 55 99 10 65 50
1 i j n
20 25 35 40 44 55 99 10 65 50
1 i j n
20 25 35 40 44 55 99 10 65 50
1 i j n
20 25 35 38 40 44 55 99 10 65 50

```
INSERTIONSORT(A, n)
1   para j ← 2 até n faça
2       chave ← A[j]
3       ▷ Insere A[j] no subvetor ordenado A[1..j − 1]
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           A[i + 1] ← A[i]
7           i ← i − 1
8       A[i + 1] ← chave
```

ANÁLISE DE ALGORITMO

O que é importante analisar?

ANÁLISE DE ALGORITMO

O que é importante analisar?

- Finitude: o algoritmo pára?

ANÁLISE DE ALGORITMO

O que é importante analisar?

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?

ANÁLISE DE ALGORITMO

O que é importante analisar?

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?
- Complexidade de tempo: quantas intruções são necessárias no **pior caso** para ordenar os n elementos?

COMPLEXIDADE DE TEMPO

O tempo de execução (ou complexidade de tempo) é dado em função do tamanho de entrada

COMPLEXIDADE DE TEMPO

O tempo de execução (ou complexidade de tempo) é dado em função do tamanho de entrada

- Na ordenação: o tamanho de entrada é a dimensão do vetor (ignora os valores dos seus elementos)

COMPLEXIDADE DE TEMPO

O tempo de execução (ou complexidade de tempo) é dado em função do tamanho de entrada

- Na ordenação: o tamanho de entrada é a dimensão do vetor (ignora os valores dos seus elementos)

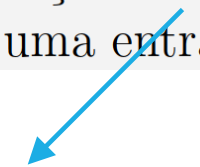
Complexidade de tempo é o número de *instruções básicas* (operações elementares ou primitivas) que executam a partir de uma entrada.

COMPLEXIDADE DE TEMPO

O tempo de execução (ou complexidade de tempo) é dado em função do tamanho de entrada

- Na ordenação: o tamanho de entrada é a dimensão do vetor (ignora os valores dos seus elementos)

Complexidade de tempo é o número de *instruções básicas* (operações elementares ou primitivas) que executam a partir de uma entrada.



Exemplo: comparação, atribuição, operações aritméticas, ...

Vamos contar o número de *instruções básicas* do InsertionSort

Vamos contar o número de *instruções básicas* do InsertionSort

Seja

- c_k: custo (tempo) de cada execução da linha k.
- t_j: número de vezes que a linha 5 é executada para aquele valor de j

Vamos contar o número de *instruções básicas* do InsertionSort

Seja

- c_k: custo (tempo) de cada execução da linha k.
- t_j: número de vezes que a linha 5 é executada para aquele valor de j

InsertionSort(A, n)

```
1 para j ← 2 até n faça
2     chave ← A[j]
3     ▷ Insere A[j] em A[1 . . . j − 1]
4     i ← j − 1
5     enquanto i ≥ 1  e A[i] > chave faça
6         A[i + 1] ← A[i]
7         i ← i − 1
8     A[i + 1] ← chave
```

Vamos contar o número de *instruções básicas* do InsertionSort

Seja

- c_k: custo (tempo) de cada execução da linha k.
- t_j: número de vezes que a linha 5 é executada para aquele valor de j

InsertionSort(A, n)	Custo
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1
2 $\quad chave \leftarrow A[j]$	c_2
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0
4 $\quad i \leftarrow j - 1$	c_4
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6
7 $\quad\quad i \leftarrow i - 1$	c_7
8 $\quad A[i+1] \leftarrow chave$	c_8

Vamos contar o número de *instruções básicas* do InsertionSort

Seja

- c_k: custo (tempo) de cada execução da linha k.
- t_j: número de vezes que a linha 5 é executada para aquele valor de j

InsertionSort(A, n)	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	$n-1$
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad\quad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] \leftarrow chave$	c_8	$n-1$

O tempo total $T(n)$ é a soma dos tempos de cada linha do algoritmo:

O tempo total $T(n)$ é a soma dos tempos de cada linha do algoritmo:

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

O tempo total $T(n)$ é a soma dos tempos de cada linha do algoritmo:

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

Entradas de tamanho igual (mesmo n), podem apresentar tempos de execução diferentes pois $T(n)$ depende dos valores t_j.

MELHOR CASO

O melhor caso ocorre quando o vetor A já está ordenado

MELHOR CASO

O melhor caso ocorre quando o vetor A já está ordenado

Assim, $t_j = 1$ para $j = 2, \ldots, n$

MELHOR CASO

O melhor caso ocorre quando o vetor A já está ordenado

Assim, $t_j = 1$ para $j = 2, \ldots, n$

Logo,

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

MELHOR CASO

O melhor caso ocorre quando o vetor A já está ordenado

Assim, $t_j = 1$ para $j = 2, \ldots, n$

Logo,

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

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

Este tempo de execução é da forma $an + b$

Portanto, no melhor caso, o tempo de execução é uma **função linear** no tamanho da entrada.

PIOR CASO

O pior caso ocorre quando o vetor A está em ordem decrescente

PIOR CASO

O pior caso ocorre quando o vetor A está em ordem decrescente

- Para inserir a *chave* em $A[1 \ldots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$.

PIOR CASO

O pior caso ocorre quando o vetor A está em ordem decrescente

- Para inserir a *chave* em $A[1 \ldots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$.

Lembre-se que

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

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

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

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right)$$
$$+ c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1)$$

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

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1)$$

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

O tempo de execução no pior caso é da forma $an^2 + bn + c$

No pior caso, o tempo é uma **função quadrática** no tamanho da entrada

COMPLEXIDADE ASSINTÓTICA

Na maioria dos casos, estuda-se o comportamento de pior caso, de maneira assintótica (instâncias de tamanho grande).

COMPLEXIDADE ASSINTÓTICA

Na maioria dos casos, estuda-se o comportamento de pior caso, de maneira assintótica (instâncias de tamanho grande).

O estudo assintótico "joga para debaixo do tapete" as **constantes** e **termos de baixa ordem**.

COMPLEXIDADE ASSINTÓTICA

Na maioria dos casos, estuda-se o comportamento de pior caso, de maneira assintótica (instâncias de tamanho grande).

O estudo assintótico "joga para debaixo do tapete" as **constantes** e **termos de baixa ordem**.

Por que podemos fazer isso ?

Considere a função quadrática $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

Como se vê, $3n^2$ é o termo dominante quando n é grande

algoritmo que executa **aproximadamente** $4n^2$ passos

```
if (a > b) {
    result = x;
} else {
    result = y;
}
```

$4n^2$ passos

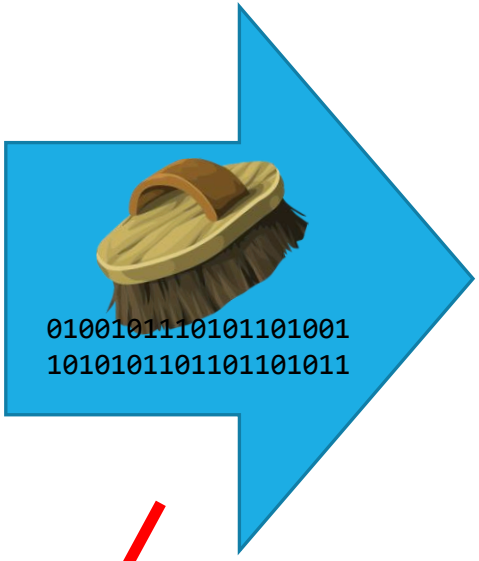

```
result = a > b ? x : y;
```

n^2 passos???

algoritmo que executa **aproximadamente** $4n^2$ passos

```
if (a > b) {
    result = x;
} else {
    result = y;
}
```

$4n^2$ passos




```
result = a > b ? x : y;
```

n^2 passos???

deixe me procurar no manual do modelo RAM

isso é tarefa do criador do algoritmo?

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; ***premature optimization is the root of all evil"*** **–** Donald Knuth ao receber o Prêmio Turing em 1974

Grosseiramente falando, podemos nos concentrar nos termos dominantes e esquecer os demais.

- Usando **notação assintótica**, dizemos que o InsertionSort tem complexidade de tempo de pior caso $\Theta(n^2)$.