MC202 - Estruturas de Dados

Guilherme P. Telles

IC

18 de abril de 2023

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides usam português anterior à reforma ortográfica de 2009.

Análise assintótica de algoritmos

Análise de algoritmos

- Podemos analisar um algoritmo para
  - determinar se ele é correto, isto é, sempre pára com o resultado esperado.
  - determinar o desempenho em termos dos recursos que ele usa, como tempo, memória, coqmunicação etc.

Desempenho de algoritmos

- Análise experimental.
- Análise assintótica.

Análise experimental

- Depende da implementação dos algoritmos e é influenciada pela experiência do programador, linguagem de programação, compilador, sistema operacional e hardware.
- O projeto dos experimentos deve ser prever um número adequado de repetições, escolha adequada de parâmetros para o funcionamento dos programas e para comparações de resultados.
- A análise dos resultados deve levar a conclusões que sejam estatisticamente significativas.
- A adição de um novo algoritmo ou mudanças tecnológicas normalmente exigem que todos os experimentos sejam refeitos.

Análise experimental

- Apesar das dificuldades, a análise experimental tem valor e em algumas situações a experimentação é a única forma de analisar eficiência.

Análise assintótica

- Para medir tempo sem implementar um algoritmo usamos como medida uma **contagem simplificada do número de operações que ele executa em função do tamanho da entrada**.
- Nosso tratamento da análise assintótica será introdutório.

Tamanho da entrada

- Definimos o tamanho da entrada de acordo com o problema que estamos resolvendo, por exemplo:
  - busca e ordenação: número de elementos no conjunto
  - calcular o valor de uma função numérica: número de bits necessários para representar um número.
  - caminhos mínimos em grafos: número de vértices + número de arestas no grafo.
- Quase todo problema tem soluções triviais para entradas pequenas. Então **sempre** vamos supor que o tamanho da entrada é suficientemtente grande.

SUM

- O algoritmo abaixo soma os elementos de um vetor.
- O tamanho da entrada é n, o número de elementos do vetor.

```
SUM(A[1..n])
  sum = 0
  for i = 1 to n
      sum = sum + A[i]
  return sum
```

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	
for $i = 1$ **to** n	
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n+1$
$\quad sum = sum + A[i]$	
return sum	

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n+1$
$\quad sum = sum + A[i]$	n
return sum	1

- Contamos todas as operações:

$\text{SUM}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$sum = sum + A[i]$	n
return sum	1

- Somando temos: $T(n) = 1 + (n + 1) + n + 1 = 2n + 3$.

- Contamos todas as operações:

$\textsc{Sum}(A[1\,..\,n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n+1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $T(n) = 1 + (n+1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.

- Contamos todas as operações:

SUM($A[1\,..\,n]$)	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$sum = sum + A[i]$	n
return sum	1

- Somando temos: $T(n) = 1 + (n + 1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.
- Dizemos que o número de operações de SUM é proporcional a n.

- Contamos todas as operações:

$\textsc{Sum}(A[1 \,.\,.\, n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n + 1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $T(n) = 1 + (n + 1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.
- Dizemos que o número de operações de $\textsc{Sum}$ é proporcional a n.
- Também dizemos que o **tempo de execução** de $\textsc{Sum}$, $T(n)$, é proporcional a n.

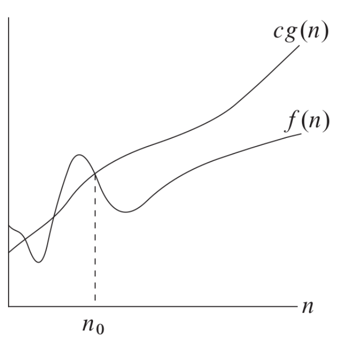
- Contamos todas as operações:

$\textsc{Sum}(A[1..n])$	operações
$sum = 0$	1
for $i = 1$ **to** n	$n+1$
$\quad sum = sum + A[i]$	n
return sum	1

- Somando temos: $T(n) = 1 + (n+1) + n + 1 = 2n + 3$.
- Descartamos os coeficientes e termos de menor grau.
- Dizemos que o número de operações de $\textsc{Sum}$ é proporcional a n.
- Também dizemos que o **tempo de execução** de $\textsc{Sum}$, $T(n)$, é proporcional a n.
- Denotamos $T(n) = O(n)$.

Notação O

- $f(n) = O(g(n))$ se existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0$.



- Intuitivamente, se $f(n) = O(g(n))$ então
  ▷ g é um limite superior assintótico para f

- $n = O(n)$

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

- $n = O(n)$
- $n = O(n \log n)$
- $n = O(n^2)$

- $n = O(n)$
- $n = O(n \log n)$
- $n = O(n^2)$
- $n = O(n^3)$

- $n = O(n)$
- $n = O(n \log n)$
- $n = O(n^2)$
- $n = O(n^3)$
- $n = O(n^n)$

- $3n^3 + 10n^2 + 1000 = O(n^3)$

- $3n^3 + 10n^2 + 1000 = O(n^3)$
- $3n^3 + 10n^2 + 1000 = O(n^4)$

- $3n^3 + 10n^2 + 1000 = O(n^3)$
- $3n^3 + 10n^2 + 1000 = O(n^4)$
- $3n^3 + 10n^2 + 1000 \neq O(n^2)$

- $3n^3 + 10n^2 + 1000 = O(n^3)$
- $3n^3 + 10n^2 + 1000 = O(n^4)$
- $3n^3 + 10n^2 + 1000 \neq O(n^2)$
- $3n^3 + 10n^2 + 1000 \neq O(n^{2.99999})$

- $3n^3 + 10n^2 + 1000 = O(n^3)$
- $3n^3 + 10n^2 + 1000 = O(n^4)$
- $3n^3 + 10n^2 + 1000 \neq O(n^2)$
- $3n^3 + 10n^2 + 1000 \neq O(n^{2.99999})$
- $\frac{1}{300}n^3 - 1000n^2 - 100000n \neq O(n^{2.99999})$

- $\pi = O(1)$

- $\pi = O(1)$
- $10^{80} = O(1)$

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.
- As constantes podem fazer uma grande diferença. Mas tendem a ser similares para algoritmos que resolvem o mesmo problema.

Não são simplificações demais?

- Quando n tende ao infinito, os termos de menor grau são dominados pelo de maior grau.
- As constantes podem fazer uma grande diferença. Mas tendem a ser similares para algoritmos que resolvem o mesmo problema.
- São simplificações razoáveis e o método funciona bem na prática para fornecer uma **aproximação** do desempenho do algoritmo.

Valores da entrada vs. número de operações

- Freqüentemente o tempo de execução de um algoritmo depende não só do tamanho da entrada, mas também dos **valores** dela.
- Não foi o caso de SUM: independentemente dos valores que estão em A o algoritmo faz a mesma quantidade de trabalho.

- O tempo de execução de INSERT depende dos valores na entrada.
- Ele recebe um vetor em que $A[1\,..\,n-1]$ está ordenado e posiciona $A[n]$ de forma que $A[1\,..\,n]$ fique ordenado.

INSERT$(A[1\,..\,n])$

$key = A[n]$
$i = n - 1$
while $i > 0$ and $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
$A[i+1] = key$

- Em casos como esses há três tipos de análises: de pior caso, de caso médio e de melhor caso.

Análise de pior caso

- Na análise de pior caso, o tempo de execução de um algoritmo é o número máximo de instruções que ele pode executar dentre todas as instâncias válidas, em função do tamanho da entrada.
- É boa por fornecer um limite superior para o tempo de execução do algoritmo.
- Pode ser muito pessimista.

- Para INSERT o pior caso é quando $A[n] < A[1 \,..\, n-1]$.

```
INSERT(A[1..n])
key = A[n]
i = n − 1
while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
A[i + 1] = key
```

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	
$i = n - 1$	
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1 \mathinner{.\,.} n-1]$.

INSERT($A[1 \mathinner{.\,.} n]$)	operações
$key = A[n]$	1
$i = n - 1$	
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	
$A[i+1] = A[i]$	
$i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$\quad A[i+1] = A[i]$	$n - 1$
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1 \,.\,.\, n-1]$.

INSERT$(A[1\,.\,.\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n - 1$
$i = i - 1$	$n - 1$
$A[i+1] = key$	

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n - 1$
$i = i - 1$	$n - 1$
$A[i+1] = key$	1

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$\quad A[i+1] = A[i]$	$n - 1$
$\quad i = i - 1$	$n - 1$
$A[i+1] = key$	1

- Somando temos $3n + 1 = O(n)$.

- Para INSERT o pior caso é quando $A[n] < A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	n
$A[i+1] = A[i]$	$n-1$
$i = i - 1$	$n-1$
$A[i+1] = key$	1

- Somando temos $3n + 1 = O(n)$.
- INSERT é $O(n)$ no pior caso.

Análise de melhor caso

- Na análise de melhor caso, o tempo de execução de um algoritmo é o número mínimo de operações que ele pode executar dentre todas as instâncias válidas, em função do tamanho da entrada.
- Costuma ser otimista demais, quase todo algoritmo tem um caso trivial que não representa a dificuldade típica do problema.

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)

$key = A[n]$
$i = n - 1$
while $i > 0$ and $A[i] > key$
    $A[i+1] = A[i]$
    $i = i - 1$
$A[i+1] = key$

- Para Insert o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

Insert($A[1\,..\,n]$)	operações
$key = A[n]$	
$i = n - 1$	
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para $\textsc{Insert}$ o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	
$\quad A[i+1] = A[i]$	
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

INSERT($A[1\,..\,n]$)	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$A[i+1] = A[i]$	
$i = i - 1$	
$A[i+1] = key$	

- Para $\textsc{Insert}$ o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$\quad A[i+1] = A[i]$	0
$\quad i = i - 1$	
$A[i+1] = key$	

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$\quad A[i+1] = A[i]$	0
$\quad i = i - 1$	0
$A[i+1] = key$	

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$A[i+1] = A[i]$	0
$i = i - 1$	0
$A[i+1] = key$	1

- Para $\textsc{Insert}$ o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

$\textsc{Insert}(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$\quad A[i+1] = A[i]$	0
$\quad i = i - 1$	0
$A[i+1] = key$	1

- Somando temos $4 = O(1)$.

- Para INSERT o melhor caso é quando $A[n] > A[1\,..\,n-1]$.

INSERT$(A[1\,..\,n])$	operações
$key = A[n]$	1
$i = n - 1$	1
while $i > 0$ and $A[i] > key$	1
$\quad A[i+1] = A[i]$	0
$\quad i = i - 1$	0
$A[i+1] = key$	1

- Somando temos $4 = O(1)$.
- INSERT é $O(1)$ no melhor caso.

Análise de caso médio

- Na análise de caso médio computamos a média do tempo de execução para todas as instâncias de um certo tamanho considerando a distribuição de probabilidades para as instâncias daquele tamanho.
- Fornece uma idéia do comportamento esperado de um algoritmo.
- Não costuma ser fácil formalizar a distribuição das instâncias e fazer esse tipo de análise.

- Se supusermos que $A[n]$ pode ocupar qualquer posição entre 1 e n com a mesma probabilidade, então o número médio de execuções do while de INSERT é

$$\frac{2+3+\ldots+n+1}{n} = \frac{n(n+3)}{2n} = \frac{n+3}{2} = O(n)$$

- INSERT é $O(n)$ no caso médio.

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

```
SELECTION-SORT(A[1..n])
for i = n downto 2
    max = 1
    for j = 2 to i
        if A[j] > A[max]
            max = j
    exchange A[i] and A[max]
```

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	
$max = 1$	
for $j = 2$ **to** i	
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	
for $j = 2$ **to** i	
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n-1$
for $j = 2$ **to** i	
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

SELECTION-SORT$(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n - 1$
for $j = 2$ **to** i	$\sum_{k=2}^{n} k$
if $A[j] > A[max]$	
$max = j$	
exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

SELECTION-SORT($A[1\,..\,n]$)	operações
for $i = n$ **downto** 2	n
$\quad max = 1$	$n-1$
$\quad$ **for** $j = 2$ **to** i	$\sum_{k=2}^{n} k$
$\quad\quad$ **if** $A[j] > A[max]$	$\sum_{k=2}^{n} k - (n-1)$
$\quad\quad\quad max = j$	
$\quad$ exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n-1$
for $j = 2$ **to** i	$\sum_{k=2}^{n} k$
if $A[j] > A[max]$	$\sum_{k=2}^{n} k - (n-1)$
$max = j$	$\sum_{k=2}^{n} k - (n-1)$
exchange $A[i]$ and $A[max]$	

Outro exemplo: selection-sort

- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$\quad max = 1$	$n-1$
$\quad$ **for** $j = 2$ **to** i	$\sum_{k=2}^{n} k$
$\quad\quad$ **if** $A[j] > A[max]$	$\sum_{k=2}^{n} k - (n-1)$
$\quad\quad\quad max = j$	$\sum_{k=2}^{n} k - (n-1)$
$\quad$ exchange $A[i]$ and $A[max]$	$n-1$

Outro exemplo: selection-sort

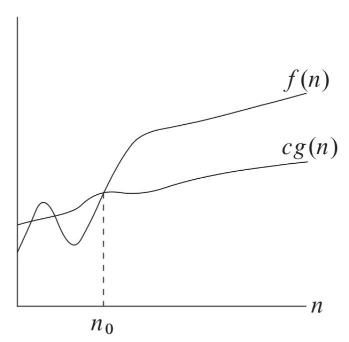
- Ordena um vetor encontrando o máximo n-1 vezes.
- O tempo de execução não depende dos valores em A.

$\textsc{Selection-Sort}(A[1\,..\,n])$	operações
for $i = n$ **downto** 2	n
$max = 1$	$n-1$
for $j = 2$ **to** i	$\sum_{k=2}^{n} k$
if $A[j] > A[max]$	$\sum_{k=2}^{n} k - (n-1)$
$max = j$	$\sum_{k=2}^{n} k - (n-1)$
exchange $A[i]$ and $A[max]$	$n-1$

- $\textsc{Selection-Sort}$ é $O(n^2)$.

Ω

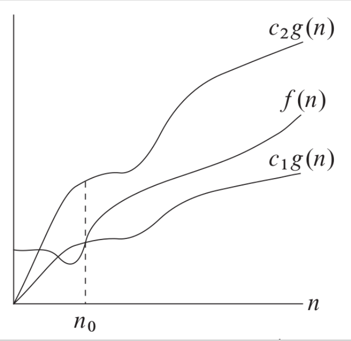
- $f(n) = \Omega(g(n))$ se existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0$.



- Intuitivamente, se $f(n) = \Omega(g(n))$ então
  $\triangleright$ g é um limite inferior assintótico para f

Θ

- $f(n) = \Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0$



- Intuitivamente, se $f(n) \in \Theta(g(n))$ então
  $\triangleright$ g é limite inferior e superior assintótico para f.

- Também há as notações o e ω que representam limites estritamente superior e estritamente inferior.

Memória

- Para a memória podemos usar a mesma técnica, contando o número de posições de memória usadas pelo algoritmo.
- Não contamos a memória usada pela entrada e pela saída.

Algumas funções

ops \ n	10	20	50	100	500	1000
$100000 \cdot \log n$	0.0003	0.0004	0.0005	0.0006	0.0008	0.0009
$10000 \cdot n$	0.0001	0.0002	0.0005	0.001	0.005	0.01
$1000 \cdot n \log n$	0.00003	0.00009	0.0003	0.0007	0.004	0.01
$100 \cdot n^2$	0.00001	0.00004	0.0003	0.001	0.03	0.1
$10 \cdot n^3$	0.00001	0.00008	0.001	0.01	1.3	10
$2 \cdot n^4$	0.00002	0.0003	0.01	0.2	125	0.5 horas
n^5	0.0001	0.00320	0.31250	10	8.68 horas	11.57 dias
$n^{\log n}$	0.000002	0.0004	4	5.4 horas	10^5 séc.	
2^n	0.000001	0.001	13 dias	10^{11} séc.		
3^n	0.00006	3	10^5 séc.			
$n!$	0.004	77 anos				

Tempo supondo 10^9 operações de algoritmo por segundo.

- Mais exemplos ao longo do semestre.