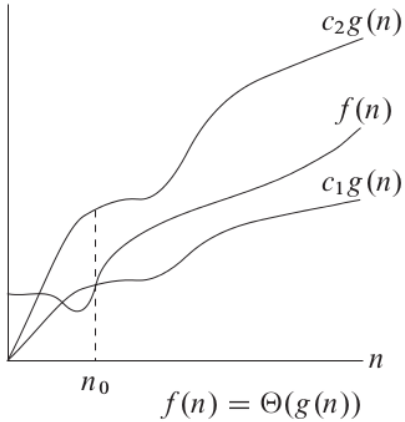
# Análise Assintótica de Algoritmos

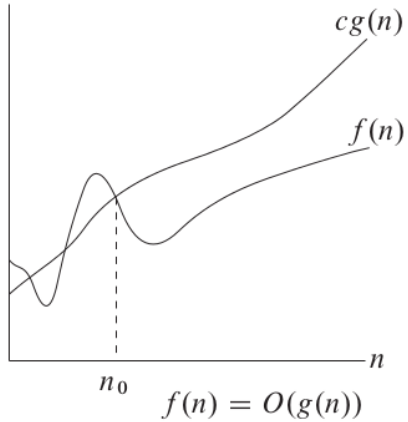
Pedro Ribeiro

DCC/FCUP

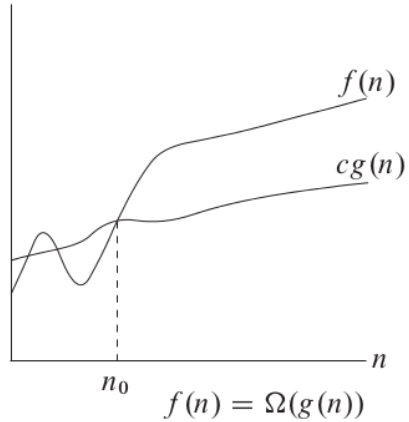
2022/2023




$f(n) = \Theta(g(n))$




$f(n) = O(g(n))$



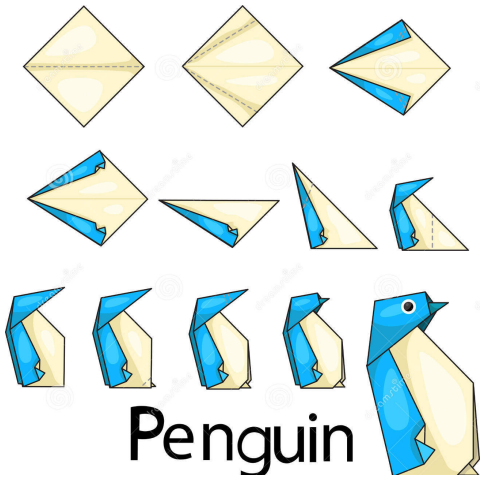

$f(n) = \Omega(g(n))$

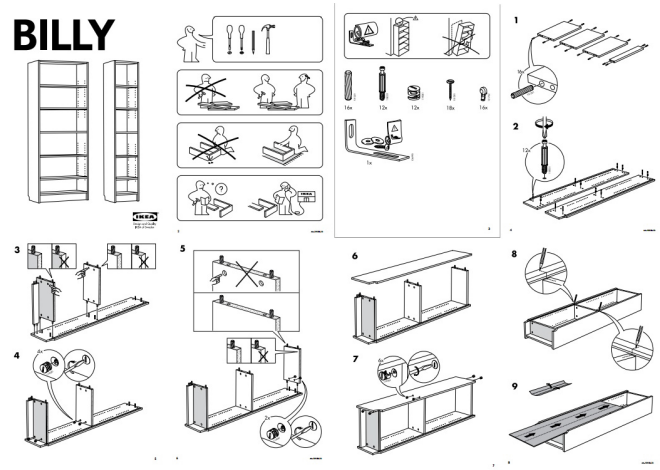
# Introdução e Motivação

# O que é um algoritmo?

Um conjunto de **instruções** executáveis para resolver um **problema**

- O problema é a **motivação** para o algoritmo
- As instruções têm de ser **executáveis**
- Geralmente existem **vários algoritmos** para um mesmo problema [Como escolher?]
- **Representação**: descrição das instruções suficiente para que a audiência o entenda

# O que é um algoritmo?

## Versão "Ciência de Computadores"

- Os algoritmos são as **ideias** por detrás dos programas
  São independentes da linguagem de programação, da máquina, ...
- Um algoritmo serve para resolver um **problema**
- Um problema é caracterizado pela descrição do **input** e **output**

Um exemplo clássico:

### Problema de Ordenação

**Input:** uma sequência $\langle a_1, a_2, \ldots, a_n \rangle$ de $n$ números
**Output:** uma permutação dos números $\langle a_1', a_2', \ldots, a_n' \rangle$ tal que $a_1' \leq a_2' \leq \ldots \leq a_n'$

### Exemplo para Problema de Ordenação

**Input:** 6 3 7 9 2 4
**Output:** 2 3 4 6 7 9

# O que é um algoritmo?

## Como representar um algoritmo

- Vamos usar preferencialmente **pseudo-código** *(nos slides)*
- Por vezes usaremos **C/C++/Java** ou frases em **português**
- O pseudo-código é baseado em linguagens imperativas e é "legível"

**Pseudo-Código**

```
a ← 0
i ← 0
Enquanto (i < 5) fazer
        a ← a + i

escrever(a)
```

**Código em C**

```
a = 0;
i = 0;
while (i<5) {
    a += i;
}
printf("%d\n", a);
```

# Propriedades desejadas num algoritmo

## Correção

Tem de resolver correctamente **todas as instâncias** do problema

## Eficiência

Performance (**tempo** e **memória**) tem de ser adequada

# Correção de um algoritmo

- **Instância**: Exemplo concreto de input válido
- Um algoritmo correto resolve **todas as instâncias** possíveis
  Exemplos para ordenação: números já ordenados, repetidos, ...
- Nem sempre é fácil **provar** a correção de um algoritmo e muito menos é óbvio se um algoritmo está correcto

# Correção de um algoritmo

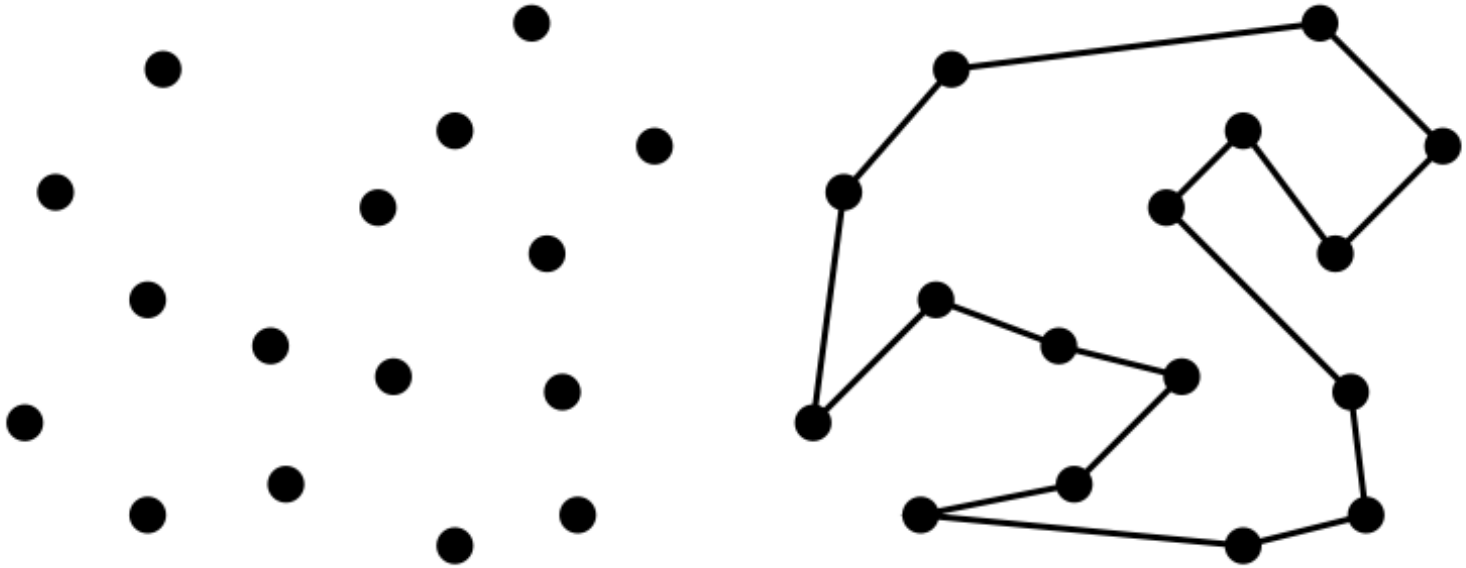
## Um problema exemplo

**Problema do Caixeiro Viajante (*Euclidean TSP*)**

**Input**: um conjunto $S$ de $n$ pontos no plano

**Output**: O **caminho mais curto** que começa num ponto, visita todos os outros pontos de $S$, e regressa ao ponto inicial.

Um exemplo:



# Correção de um algoritmo

## Um problema exemplo - Caixeiro Viajante

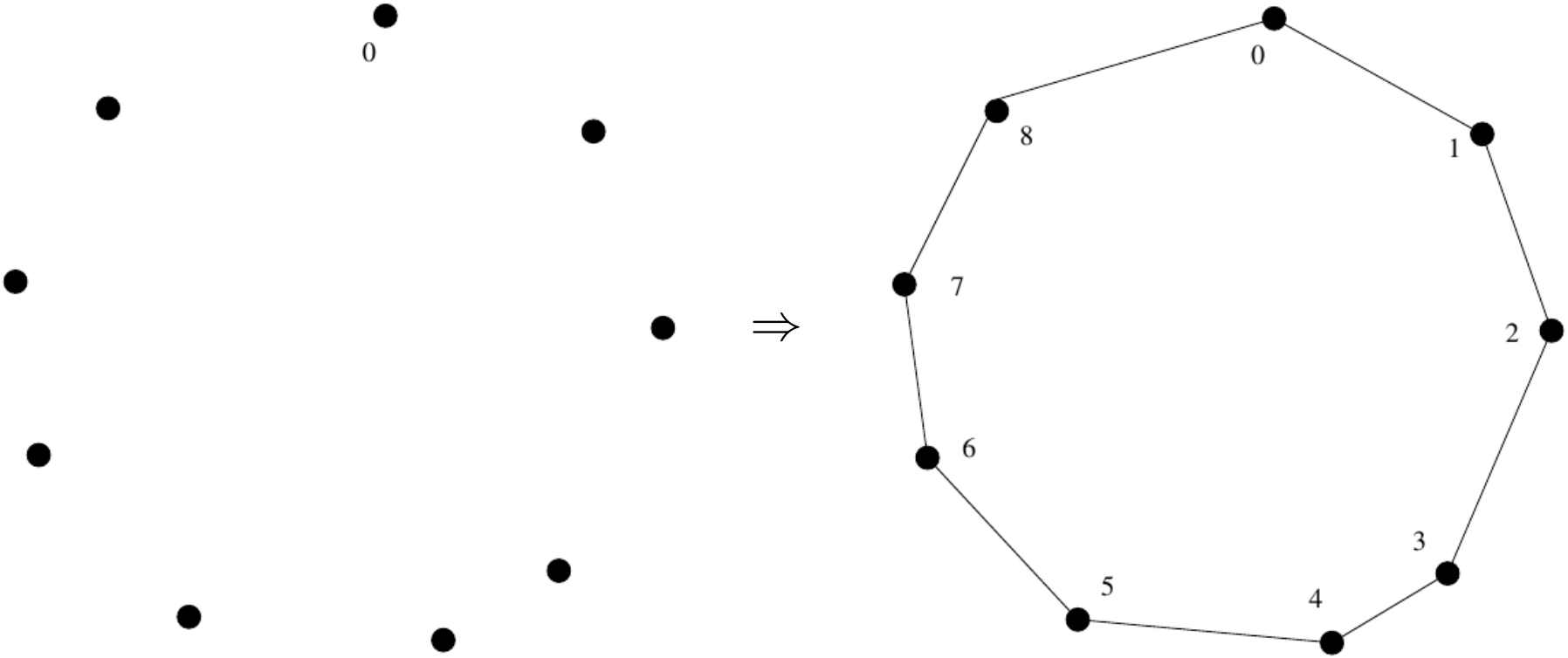
### Um 1º possível algoritmo (vizinho mais próximo)

$p_1 \leftarrow$ ponto inicial escolhido aleatoriamente
$i \leftarrow 1$
**Enquanto** (existirem pontos por visitar) **fazer**
    $i \leftarrow i + 1$
    $p_i \leftarrow$ vizinho não visitado mais próximo de $p_{i-1}$
**retorna** caminho $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \rightarrow p_1$

# Correção de um algoritmo

## Um problema exemplo - Caixeiro Viajante - vizinho mais próximo

Parece funcionar...

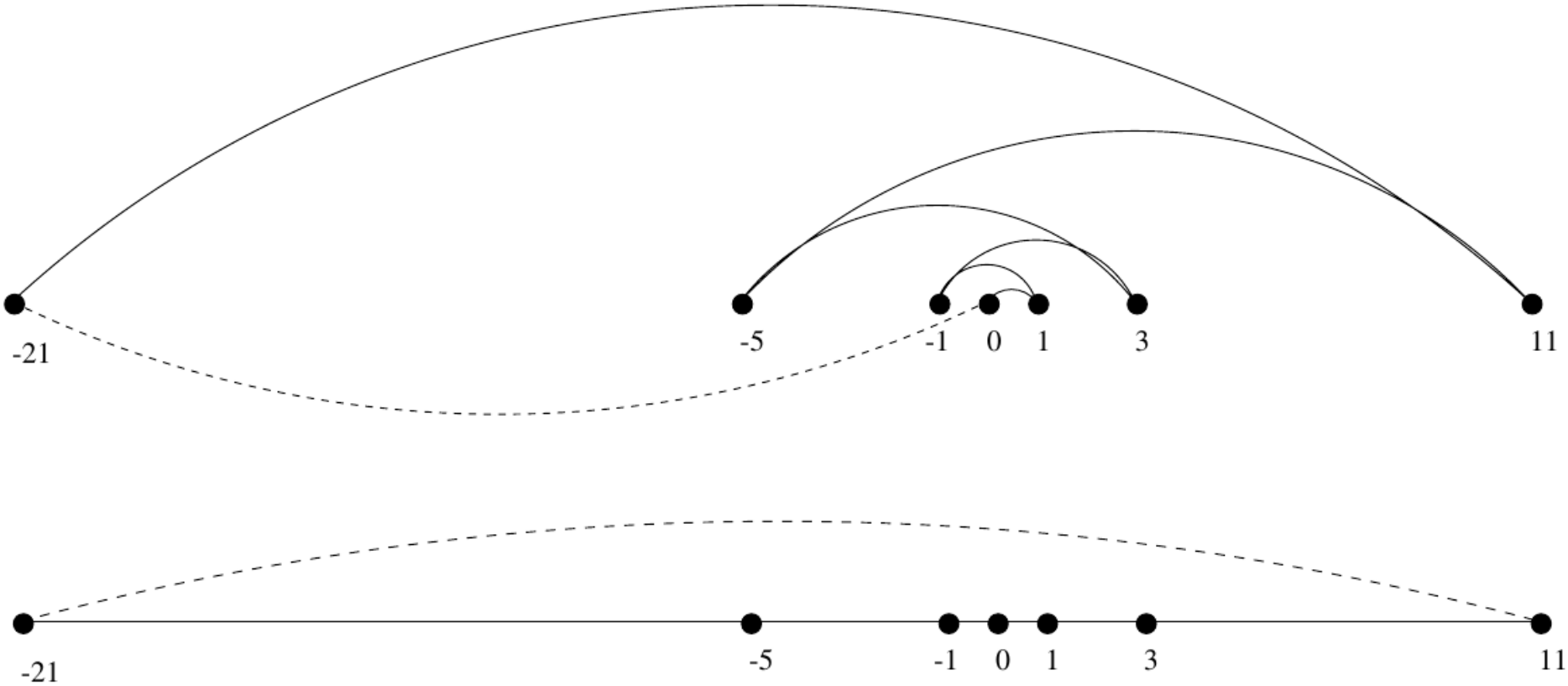


# Correção de um algoritmo

## Um problema exemplo - Caixeiro Viajante - vizinho mais próximo

Mas não funciona para todas as instâncias!

(Nota: começar pelo ponto mais à esquerda não resolveria o problema)



# Correção de um algoritmo

## Um problema exemplo - Caixeiro Viajante

Como resolver então o problema? *(tentar todos os caminhos possíveis)*

**Um 2º possível algoritmo (pesquisa exaustiva *aka* força bruta)**

$P_{min} \leftarrow$ uma qualquer permutação dos pontos de $S$
**Para** $P_i \leftarrow$ cada uma das permutações de pontos de $S$
    **Se** $(custo(P_i) < custo(P_{min}))$ **Então**
        $P_{min} \leftarrow P_i$
**retorna** Caminho formado por $P_{min}$

O algoritmo é correto, mas extremamente lento!

- $P(n) = n! = n \times (n-1) \times \ldots \times 1$
- Por exemplo, $P(20) = 2,432,902,008,176,640,000$
- Para uma instância de tamanho 20, o computador mais rápido do mundo não resolvia— (quanto tempo demoraria?)

# Correção de um algoritmo

## Um problema exemplo - Caixeiro Viajante

- O problema apresentado é uma versão restrita (euclideana) de um dos problemas mais "clássicos", o **Travelling Salesman Problem** (**TSP**)
- Este problema tem **inúmeras aplicações** (mesmo na forma "pura")
  Ex: análise genómica, produção industrial, routing de veículos, ...
- Não é conhecida **nenhuma solução eficiente** para este problema
  (que dê resultados ótimos, e não apenas "aproximados")
- A solução apresentada tem complexidade temporal $\mathcal{O}(n!)$
  O algoritmo de Held-Karp tem complexidade $\mathcal{O}(2^n n^2)$
  (iremos falar deste tipo de análise: *big O notation*)
- O TSP pertence à classe dos problemas **NP-hard**
  A versão de decisão pertence à classes dos problemas **NP-completos**
  (vão falar disto noutras UCs)

# Eficiência de um algoritmo

## Uma experiência - instruções

- Quantas instruções simples faz um computador actual por segundo?
  (apenas uma aproximação, uma ordem de grandeza)

  No meu portátil umas $10^9$ instruções
- A esta velocidade quanto tempo demorariam as seguintes quantidades de instruções?

| **Quant.** | **100** | **1000** | **10000** |
|---|---|---|---|
| $N$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ |
| $N^2$ | $< 0.01s$ | $< 0.01s$ | $0.1s$ |
| $N^3$ | $< 0.01s$ | $1.00s$ | 16 min |
| $N^4$ | $0.1s$ | 16 min | 115 dias |
| $2^N$ | $10^{13}$ anos | $10^{284}$ anos | $10^{2993}$ anos |
| $n!$ | $10^{141}$ anos | $10^{2551}$ anos | $10^{35642}$ anos |

# Eficiência de um algoritmo

## Uma experiência - permutações

- Voltemos à ideia das **permutações**

**Exemplo: as 6 permutações de $\{1, 2, 3\}$**

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

- Recorda que o número de permutações pode ser calculado como:
  $P(n) = n! = n \times (n-1) \times \ldots \times 1$
  (consegues perceber a fórmula?)

# Eficiência de um algoritmo

## Uma experiência - permutações

- Quanto tempo demora um programa que passa por todas as permutações de $n$ números?
  (os seguintes tempos são aproximados, no meu portátil)
  (o que quero mostrar é a **taxa de crescimento**)

$\mathbf{n} \leq \mathbf{7}$: $< 0.001s$
$\mathbf{n} = \mathbf{8}$: $0.001s$
$\mathbf{n} = \mathbf{9}$: $0.016s$
$\mathbf{n} = \mathbf{10}$: $0.185s$
$\mathbf{n} = \mathbf{11}$: $2.204s$
$\mathbf{n} = \mathbf{12}$: $28.460s$
...
$\mathbf{n} = \mathbf{20}$: 5000 anos !

Quantas permutações por segundo?
Cerca de $10^7$

# Eficiência de um algoritmo

## Sobre a rapidez do computador

- Um **computador mais rápido** adiantava alguma coisa? **Não!**
  Se $n = 20 \rightarrow 5000$ anos, hipoteticamente:
  - 10x mais rápido ainda demoraria 500 anos
  - 5,000x mais rápido ainda demoraria 1 ano
  - 1,000,000x mais rápido demoraria quase dois dias mas
    $n = 21$ já demoraria mais de um mês
    $n = 22$ já demoraria mais de dois anos!
  - A **taxa de crescimento do algoritmo** é muito importante!

### Algoritmo vs Rapidez do computador

Um algoritmo melhor num computador mais lento ***ganhará sempre*** a um algoritmo pior num computador mais rápido, para instâncias suficientemente grandes

# Uma metodologia para comparar algoritmos

# Eficiência de um algoritmo

## Perguntas

- Como conseguir **prever** o tempo que um algoritmo demora?
- Como conseguir **comparar** dois algoritmos diferentes?

- Vamos estudar uma **metodologia** para conseguir responder
- Vamos focar a nossa atenção no **tempo de execução**
  Podíamos por exemplo querer medir o espaço (memória)

# Random Access Machine (RAM)

- Precisamos de um **modelo** que seja **genérico** e **independente** da máquina/linguagem usada.
- Vamos considerar uma *Random Access Machine* (**RAM**)
  - Cada **operação simples** (ex: $+$, $-$, $\leftarrow$, **Se**) demora **1 passo**
  - Ciclos e procedimentos, por exemplo, não são instruções simples!
  - Cada **acesso à memória** custa também 1 passo
- Medir tempo de execução... **contando o número de passos consoante o tamanho do input:** $T(n)$
- As operações estão **simplificadas**, mas mesmo assim isto é útil
  Ex: somar dois inteiros não custa o mesmo que dividir dois reais, mas veremos que esses valores, numa visão global, não são importantes.

# Random Access Machine (RAM)

## Um exemplo de contagem

### Um programa simples

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++;
```

Vamos contar o número de operações simples:

| | |
|---|---|
| Declarações de variáveis | 2 |
| Atribuições: | 2 |
| Comparação "menor que": | $n+1$ |
| Comparação "igual a": | $n$ |
| Acesso a um array: | $n$ |
| Incremento: | entre $n$ e $2n$ (depende dos zeros) |

# Random Access Machine (RAM)

## Um exemplo de contagem

### Um programa simples

```
int count = 0;
for (int i=0; i<n; i++)
    if (v[i] == 0) count++
```

Total de operações no **pior** caso:
$T(n) = 2 + 2 + (n + 1) + n + n + 2n = 5 + 5n$

Total de operações no **melhor** caso:
$T(n) = 2 + 2 + (n + 1) + n + n + n = 5 + 4n$

# Tipos de Análises de um Algoritmo

Análise do **Pior Caso**: **(o mais usual)**

- $T(n)$ = máximo tempo do algoritmo para um qualquer input de tamanho $n$

(vamos sempre assumir que é esta a nossa análise excepto se for dito explicitamente o contrário)

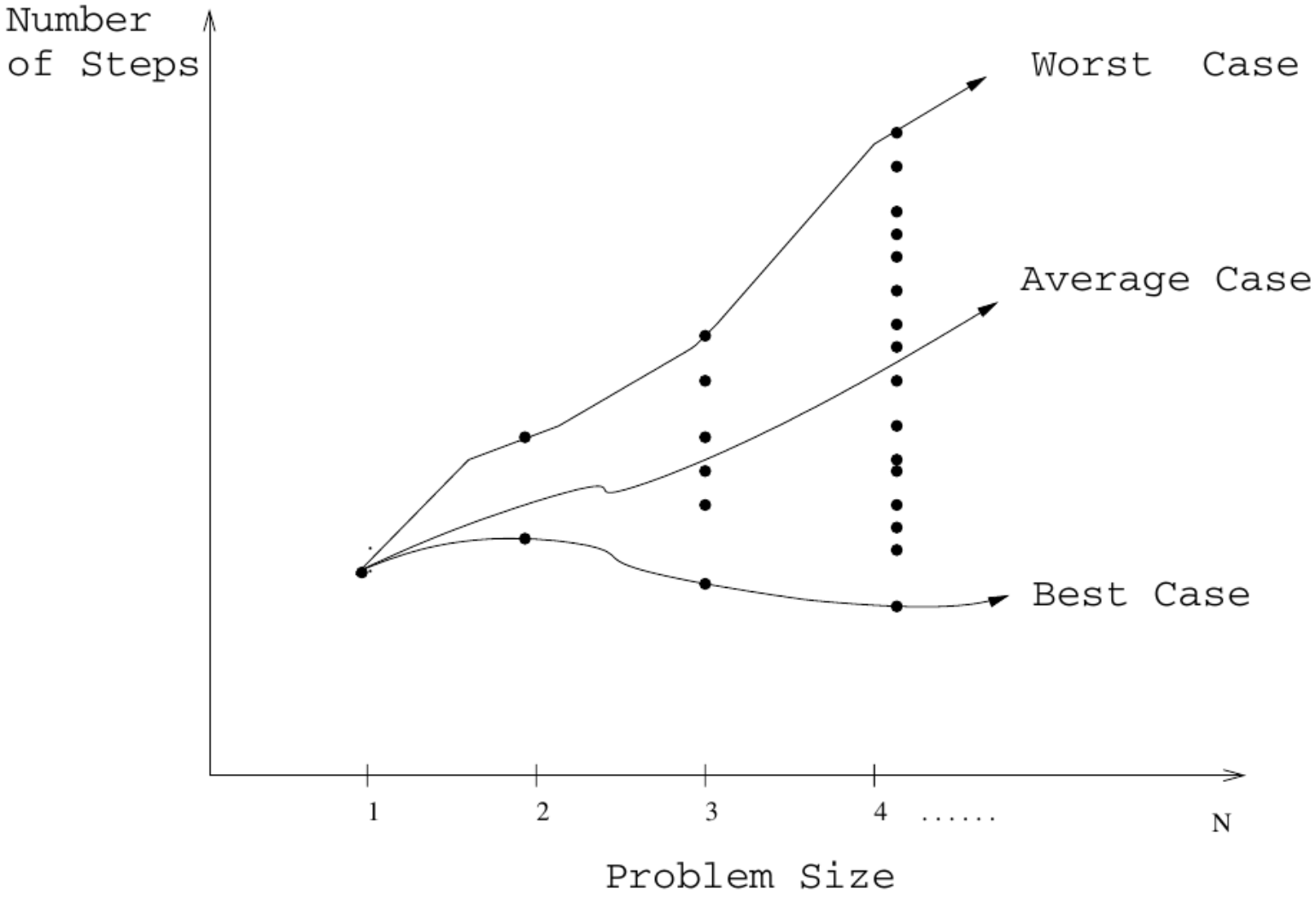
Análise **Caso Médio**: **(por vezes)**

- $T(n)$ = tempo médio do algoritmo para todos os inputs de tamanho $n$
- Implica conhecer a distribuição estatística dos inputs

Análise do **Melhor Caso**: **("enganador")**

- Fazer "batota" com um algoritmo que é rápido para *alguns* inputs

# Tipos de Análises de um Algoritmo



# Análise Assintótica

Precisamos de **ferramenta matemática** para comparar funções

Na análise de algoritmos usa-se a **Análise Assintótica**

- "Matematicamente": estudo dos comportamento dos **limites**
- CC: estudo do comportamento para input arbitrariamente grande
  ou
  "descrição" da **taxa de crescimento**

- Usa-se uma **notação** específica: $\mathcal{O}, \Omega, \Theta$ (e também $o, \omega$)
- Permite "simplificar" expressões como a anteriormente mostrada focando apenas nas **ordens de grandeza**

# Notação

## Definições

### $\mathbf{f(n) \in \mathcal{O}(g(n))}$ (majorante)

Significa que $c \times g(n)$ é um **limite superior** de $f(n)$

### $\mathbf{f(n) \in \Omega(g(n))}$ (minorante)

Significa que $c \times g(n)$ é um **limite inferior** de $f(n)$

### $\mathbf{f(n) \in \Theta(g(n))}$ (limite "apertado" - majorante e minorante)

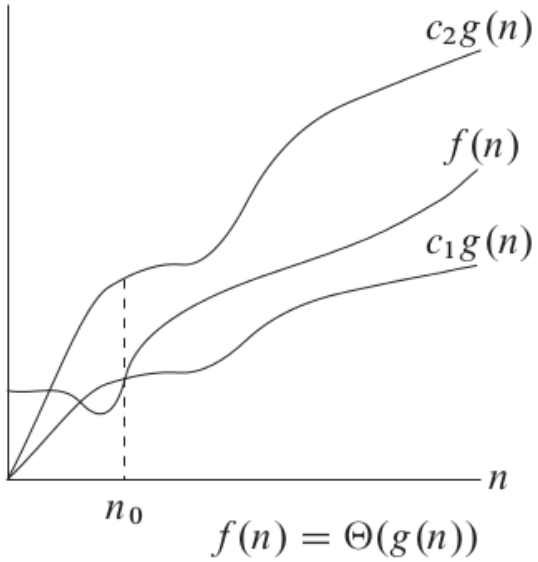
Significa que $c_1 \times g(n)$ é um **limite inferior** de $f(n)$ e $c_2 \times g(n)$ é um **limite superior** de $f(n)$

Onde $c$, $c_1$ e $c_2$ são constantes

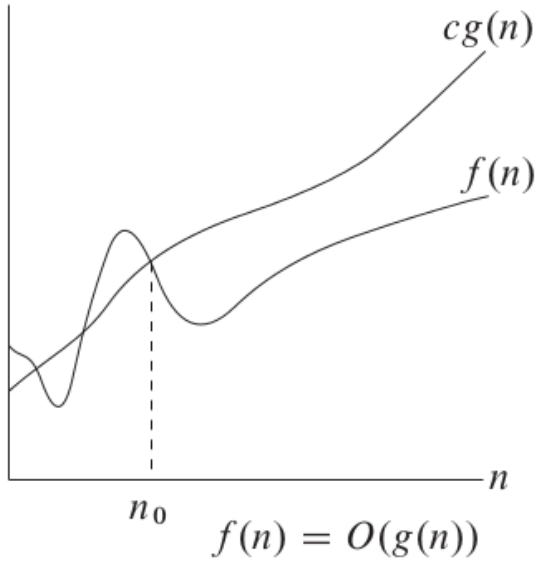
# Notação

## Uma ilustração

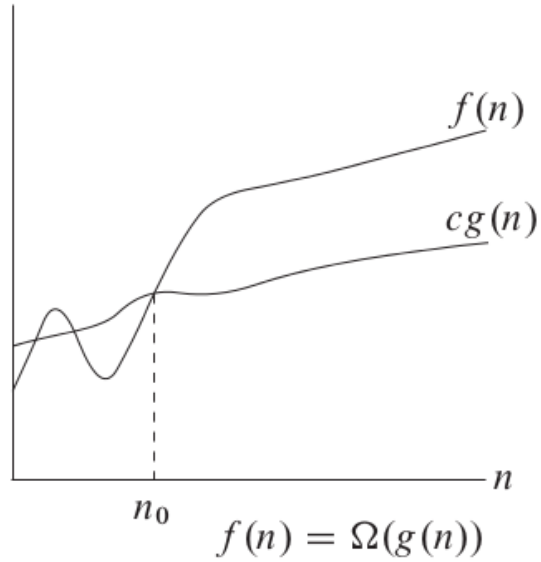
Θ




O




Ω




As definições implicam um $n$ a partir do qual a função é majorada e/ou minorada. Valores pequenos de $n$ "não importam".

**Nota:** Alguma bibliografia usa $=$ em vez de $\in$
Exemplo: $f(n) = \mathcal{O}(g(n))$ é o mesmo que $f(n) \in \mathcal{O}(g(n))$

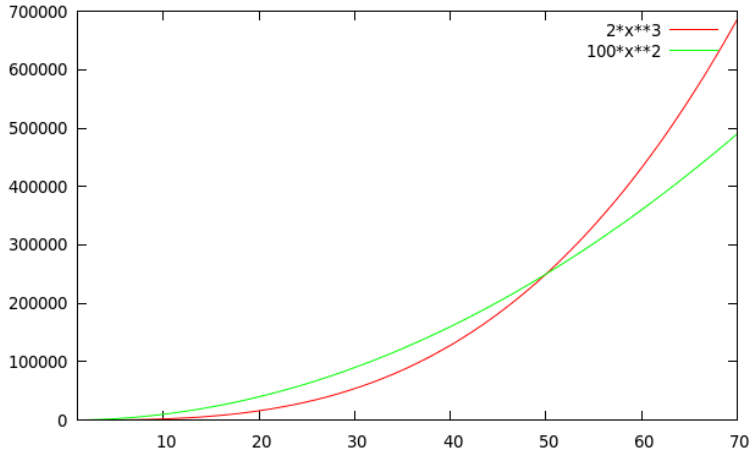
# Crescimento Assintótico

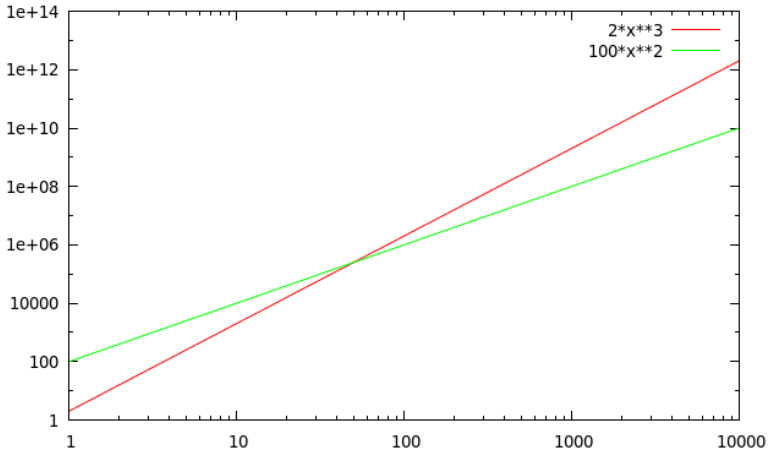
## Desenhando funções com gnuplot

Um programa útil para desenhar gráficos de funções é o `gnuplot`.

(comparando $2n^3$ com $100n^2$)

```
gnuplot> plot [1:70] 2*x**3, 100*x**2
gnuplot> set logscale xy 10
gnuplot> plot [1:10000] 2*x**3, 100*x**2
```



# Notação

## Formalização

- $\mathbf{f(n)} \in \mathcal{O}(\mathbf{g(n)})$ se existem constantes positivas $n_0$ e $c$ tal que $f(n) \leq c \times g(n)$ para todo o $n \geq n_0$

$f(n) = 3n^2 + 5n + 6$

$f(n) \in \mathcal{O}(n^2)$, para $c = 4$, temos que $cn^2 \geq f(n)$ para $n \geq 6$

$f(n) \in \mathcal{O}(n^3)$, para $c = 1$, temos que $cn^3 \geq f(n)$ para $n \geq 4.5$

$f(n) \notin \mathcal{O}(n)$, para qualquer $c$, temos que $cn < f(n)$ para $n$ suficientemente grande

(experimente usar o gnuplot para desenhar as funções)

# Notação

## Formalização

- $\mathbf{f(n) \in \Omega(g(n))}$ se existem constantes positivas $n_0$ e $c$ tal que $f(n) \geq c \times g(n)$ para todo o $n \geq n_0$

$f(n) = 3n^2 + 5n + 6$

$f(n) \in \Omega(n^2)$, para $c = 1$, temos que $cn^2 \leq f(n)$ para $n \geq 0$

$f(n) \notin \Omega(n^3)$, para qualquer $c$, temos que $cn^3 > f(n)$ para $n$ suficientemente grande

$f(n) \in \Omega(n)$, para $c = 1$, temos que $cn^2 \leq f(n)$ para $n \geq 0$

(experimente usar o gnuplot para desenhar as funções)

# Notação

## Formalização

- $\mathbf{f(n) \in \Theta(g(n))}$ se existem constantes positivas $n_0$, $c_1$ e $c_2$ tal que $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ para todo o $n \geq n_0$

$f(n) = 3n^2 + 5n + 6$

$f(n) \in \Theta(n^2)$, porque $f(n) = \mathcal{O}(n^2)$ e $f(n) = \Omega(n^2)$

$f(n) \notin \Theta(n^3)$, porque $f(n) = \mathcal{O}(n^3)$, mas $f(n) \neq \Omega(n^3)$

$f(n) \notin \Theta(n)$, porque $f(n) = \Omega(n)$, mas $f(n) \neq \mathcal{O}(n)$

- $f(n) \in \Theta(g(n))$ implica $f(n) \in \mathcal{O}(g(n))$ e $f(n) \in \Omega(g(n))$

(experimente usar o gnuplot para desenhar as funções)

# Notação

**Formalização:**

- $\mathbf{f(n) \in \mathcal{O}(g(n))}$ se existem constantes positivas $n_0$ e $c$ tal que $f(n) \leq c \times g(n)$ para todo o $n \geq n_0$
- $\mathbf{f(n) \in \Omega(g(n))}$ se existem constantes positivas $n_0$ e $c$ tal que $f(n) \geq c \times g(n)$ para todo o $n \geq n_0$
- $\mathbf{f(n) \in \Theta(g(n))}$ se existem constantes positivas $n_0$, $c_1$ e $c_2$ tal que $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ para todo o $n \geq n_0$

**Algumas Consequências:**

- $f(n) \in \mathbf{\Theta}(g(n)) \longleftrightarrow f(n) \in \mathbf{O}(g(n))$ e $f(n) \in \mathbf{\Omega}(g(n))$
- $f(n) \in \mathbf{\Theta}(g(n)) \longleftrightarrow g(n) \in \mathbf{\Theta}(f(n))$
- $f(n) \in \mathbf{O}(g(n)) \longleftrightarrow g(n) \in \mathbf{\Omega}(f(n))$

# Notação: uma analogia

Para ser mais fácil "relembrar", fica aqui uma analogia para se recordarem.

Comparação entre duas funções $f$ e $g$, e entre dois números $a$ e $b$:

| | | | | |
|---|---|---|---|---|
| $f(n) \in \mathbf{O}(g(n))$ | é como | $a \leq b$ | limite superior | pelo menos tão bom como |
| $f(n) \in \mathbf{\Omega}(g(n))$ | é como | $a \geq b$ | limite inferior | pelo menos tão mau como |
| $f(n) \in \mathbf{\Theta}(g(n))$ | é como | $a = b$ | "iguais" | tão bom (ou mau) como |

# Notação: Algumas regras práticas

- **Multiplicação por uma constante** não altera o comportamento:
  $\Theta(c \times f(n)) \in \Theta(f(n))$
  $99 \times n^2 \in \Theta(n^2)$

- Num polinómio $a_x n^x + a_{x-1} n^{x-1} + \ldots + a_2 n^2 + a_1 n + a_0)$
  podemos focar-nos na parcela com o **maior expoente**:
  $3\mathbf{n^3} - 5n^2 + 100 \in \Theta(n^3)$
  $6\mathbf{n^4} - 20^2 \in \Theta(n^4)$
  $0.8\mathbf{n} + 224 \in \Theta(n)$

- Numa soma/subtracção podemos focar-nos na parcela **dominante**:
  $\mathbf{2^n} + 6n^3 \in \Theta(2^n)$
  $\mathbf{n!} - 3n^2 \in \Theta(n!)$
  $n \log n + 3\mathbf{n^2} \in \Theta(n^2)$

# Crescimento Assintótico

## Quando uma função domina a outra

Quando é que uma função **é melhor** que outra?

- Se queremos minimizar o tempo, **funções "mais pequenas" são melhores**
- Uma função **domina** outra se à medida que $n$ cresce ela fica "infinitamente maior"
- Matematicamente: $f(n) \gg g(n)$ se $\lim_{n\to\infty} g(n)/f(n) = 0$

### Relações de Domínio

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n!$$

# Crescimento Assintótico

## Funções Usuais

| Função | Nome | Exemplos |
|---|---|---|
| 1 | constante | somar dois números |
| $\log n$ | logarítmica | pesquisa binária, inserir elemento numa heap |
| $n$ | linear | 1 ciclo para encontrar o máximo |
| $n \log n$ | linearítmica | ordenação (ex: mergesort, heapsort) |
| $n^2$ | quadrática | 2 ciclos (ex: verificar pares, bubblesort) |
| $n^3$ | cúbica | 3 ciclos (ex: Floyd-Warshall) |
| $2^n$ | exponencial | pesquisa exaustiva (ex: subconjuntos) |
| $n!$ | factorial | todas as permutações |

$n$ na base $\rightarrow$ função **polinomial**

$n$ no expoente $\rightarrow$ função **exponencial**

# Crescimento Assintótico

## Uma visão prática

Se uma operação demorar $10^{-9}$ segundos

| | $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| 10 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ |
| 20 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | 77 anos |
| 30 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $1.07s$ | |
| 40 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | 18.3 min | |
| 50 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | 13 dias | |
| 100 | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $10^{13}$ *anos* | |
| $10^3$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $1s$ | | |
| $10^4$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $0.1s$ | 16.7 min | | |
| $10^5$ | $< 0.01s$ | $< 0.01s$ | $< 0.01s$ | $10s$ | 11 dias | | |
| $10^6$ | $< 0.01s$ | $< 0.01s$ | $0.02s$ | 16.7 min | 31 anos | | |
| $10^7$ | $< 0.01s$ | $0.01s$ | $0.23s$ | 1.16 dias | | | |
| $10^8$ | $< 0.01s$ | $0.1s$ | $2.66s$ | 115 dias | | | |
| $10^9$ | $< 0.01s$ | $1s$ | $29.9s$ | 31 anos | | | |

# Crescimento Assintótico

## Funções menos usuais - Exemplos com gnuplot

- **Qual cresce mais rápido: $\sqrt{n}$ ou $\log_2 n$?**

```
gnuplot> plot [1:60] sqrt(x), log(x)/log(2)
```

$\sqrt{n}$ cresce mais rápido, logo é pior, ou seja, $\sqrt{n} \in \Omega(\log_2 n)$

- **Qual cresce mais rápido: $\log_2 n$ ou $\log_3 n$?**

```
gnuplot> plot [1:100] log(x)/log(2), log(x)/log(3),
2*log(x)/log(3)
```

crescem ao "mesmo" ritmo, ou seja, $\log_2 n \in \Theta(\log_3 n)$

# Análise Assintótica

## Mais alguns exemplos

- Um programa tem dois pedaços de código $A$ e $B$, executados um a seguir ao outro, sendo que $A$ corre em $\Theta(n \log n)$ e $B$ em $\Theta(n^2)$.
  O programa corre em $\Theta(n^2)$, porque $n^2 \gg n \log n$
- Um programa chama $n$ vezes uma função $\Theta(\log n)$, e de seguida volta a chamar novamente $n$ vezes outra função $\Theta(\log n)$
  O programa corre em $\Theta(n \log n)$
- Um programa tem 5 ciclos, chamados sequencialmente, cada um deles com complexidade $\Theta(n)$
  O programa corre em $\Theta(n)$
- Um programa $P_1$ tem tempo de execução proporcional a $100 \times n \log n$. Um outro programa $P_2$ tem $2 \times n^2$.
  Qual é o programa mais eficiente?
  $P_1$ é mais eficiente porque $n^2 \gg n \log n$. No entanto, para um $n$ pequeno, $P_2$ é mais rápido e pode fazer sentido ter um programa que chama $P_1$ ou $P_2$ consoante o $n$.

# Previsão do tempo de execução de um algoritmo

# Notação Assintótica

## Como usar para previsão

Se eu tiver um algoritmo com complexidade assintótica definida pela função $f(n)$ como posso **prever** o tempo que vai demorar?

- **Caso 1** Depois de ter uma implementação a funcionar onde possa testar com um $n$ pequeno
- **Caso 2** Ainda antes de começar a implementar qualquer algoritmo

# Previsão do tempo de execução

## Quando tenho uma implementação

Pré-requisitos:

- Uma implementação com complexidade $f(n)$
- Um caso de teste (pequeno) com input de tamanho $n_1$
- O tempo que o programa demora nesse input: $tempo(n_1)$

Agora queremos estimar quanto tempo demora para um input (parecido) de tamanho $n_2$. **Como fazer?**

### Estimando o tempo de execução

$f(n_2)/f(n_1)$ é a taxa de crescimento da função (de $n_1$ para $n_2$)

$$\mathbf{tempo(n_2) = f(n_2)/f(n_1) \times tempo(n_1)}$$

# Previsão do tempo de execução

## Quando tenho uma implementação

Um exemplo:

- Tenho um programa de complexidade $\mathbf{\Theta(n^2)}$ que demora **1 segundo** para um input de tamanho **5,000**. Quanto tempo demora para um input de tamanho **10,000?**

$$f(n) = n^2$$
$$n_1 = 5,000$$
$$tempo(n_1) = 1$$
$$n_2 = 10,000$$

$$tempo(n_2) = f(n_2)/f(n1) \times tempo(n_1) =$$
$$= 10,000^2/5,000^2 \times 1 = 4 \text{ segundos}$$

# Previsão do tempo de execução

## Sobre a taxa de crescimento

Vejamos o que acontece quando se **duplica o tamanho do input** para algumas das funções habituais (independentemente da máquina!):

**tempo(2n)** $=$ **f(2n)/f(n)** $\times$ **tempo(n)**

- **n** : $2n/n = 2$. O tempo duplica!
- **n²** : $(2n)^2/n^2 = 4n^2/n^2 = 4$. O tempo aumenta 4x!
- **n³** : $(2n)^3/n^3 = 8n^3/n^3 = 8$. O tempo aumenta 8x!

- **2ⁿ** : $2^{2n}/2^n = 2^{2n-n} = 2^n$. O tempo aumenta $2^n$ vezes!
  Exemplo: Se $n = 5$, o tempo para $n = 10$ vai ser **32x** mais!
  Exemplo: Se $n = 10$, o tempo para $n = 20$ vai ser **1024x** mais!

- **log₂(n)** : $\log_2(2n)/\log_2(n) = 1 + 1/\log_2(n)$. Aumenta $1 + \frac{1}{\log_2(n)}$ vezes!
  Exemplo: Se $n = 5$, o tempo para $n = 10$ vai ser **1.43x** mais!
  Exemplo: Se $n = 10$, o tempo para $n = 20$ vai ser **1.3x** mais!

# Previsão do tempo de execução

## Quando não tenho uma implementação

Pré-requisitos:

- A complexidade da minha ideia algorítmica: $f(n)$
- O tamanho $n$ do input para o qual quero estimar o tempo

Se eu tivesse o tempo para um dado $n_0$ podia fazer o seguinte:
$tempo(n) = f(n)/f(n_0) \times tempo(n_0) = f(n) \times \frac{tempo(n_0)}{f(n_0)}$
$\frac{tempo(n_0)}{f(n_0)} = op$: tempo para analisar uma "operação"/possível solução

O valor de *op* é dependente do problema e da máquina, mas não deixa de ser apenas um **factor constante**!

Se eu tiver o valor de *op*, calcular o tempo para $n$ passa a ser apenas:

**Estimando o tempo de execução**

*op* é quanto "custa" analisar uma possível solução

$\mathbf{tempo(n) = f(n) \times op}$

# Previsão do tempo de execução

## Quando não tenho uma implementação

Precisamos de uma estimativa de $op$, ainda que muito por alto, para ter uma ideia do tempo de execução.

Exemplos (vindos de aulas anteriores, no meu portátil):

- Uma operação simples demorava $10^{-9}$ segundos
- Cada permutação demorava $10^{-7}$ segundos

Vou dar-vos a regra de usarem $op = 10^{-8}$ para uma estimativa inicial.

No futuro podem ter de actualizar este valor, mas não é assim tão importante, porque é factor constante!

# Previsão do tempo de execução

## Tabelas

$op = 10^{-8}$

| | $n$ máximo | |
|---|---|---|
| $f(n)$ | 1s | 1min |
| $\log_2 n$ | | |
| $n$ | | |
| $n \log_2 n$ | | |
| $n^2$ | | |
| $n^3$ | | |
| $2^n$ | | |
| $n!$ | | |

$op = 10^{-8}/2$

| | $n$ máximo | |
|---|---|---|
| $f(n)$ | 1s | 1min |
| $\log_2 n$ | | |
| $n$ | | |
| $n \log_2 n$ | | |
| $n^2$ | | |
| $n^3$ | | |
| $2^n$ | | |
| $n!$ | | |

# Previsão do tempo de execução

## Tabelas

$op = 10^{-8}$

| | $n$ máximo | |
|---|---|---|
| $f(n)$ | 1s | 1min |
| $\log_2 n$ | $\sim \infty$ | $\sim \infty$ |
| $n$ | $10^8$ | $6 \times 10^9$ |
| $n \log_2 n$ | $\sim 4 \times 10^6$ | $\sim 2 \times 10^8$ |
| $n^2$ | $10,000$ | $77,459$ |
| $n^3$ | 464 | 1,817 |
| $2^n$ | 26 | 32 |
| $n!$ | 11 | 12 |

$op = 10^{-8}/2$

| | $n$ máximo | |
|---|---|---|
| $f(n)$ | 1s | 1min |
| $\log_2 n$ | $\sim \infty$ | $\sim \infty$ |
| $n$ | $2 \times 10^8$ | $\sim 10^{10}$ |
| $n \log_2 n$ | $\sim 9 \times 10^6$ | $\sim 4 \times 10^8$ |
| $n^2$ | 14,142 | 109,544 |
| $n^3$ | 584 | 2289 |
| $2^n$ | 27 | 33 |
| $n!$ | 11 | 13 |

- $\log n$ serve "virtualmente" para tudo
- $n$ para "quase" tudo (só ler já demora $\Theta(n)$)
- $n \log n$ pelo até um milhão não dá problemas
- $n^2$ para cima de $10,000$ já começa a demorar
- $n^3$ para cima de 500 já começa a demorar
- $2^n$ e $n!$ crescem muito rápido e só podem ser usados para um $n$ (mesmo) muito pequeno

# Previsão do tempo de execução

## Algumas Considerações

- A constante *op* não é o (mais) importante, mas sim a taxa de crescimento.
- Notem que isto só dá "estimativas"! Não tempos exactos...
- As "constantes escondidas" podem influenciar muito um programa
  - Ex: Ler elementos (scanf/scanner) demora muito mais tempo que operação simples
- O comportamento do programa pode depender do tipo de input
  - Ex: quicksort "naive" é bom num input aleatório, mas mau num quase ordenado

# Complexidade de programas em concreto

# Analisando complexidade de programas

Vamos agora ver um pouco de como calcular a complexidade de pedaços de código em concreto.

- **Caso 1 Ciclos** (e somatórios)
- **Caso 2 Funções Recursivas** (e recorrências)

# Ciclos e Somatórios

## Um ciclo habitual

```
contador ← 0
Para i ← 1 até 1000 fazer
    Para j ← i até 1000 fazer
        contador ← contador + 1
escrever(contador)
```

O que escreve o programa?

$1000 + 999 + 998 + 997 + \ldots.. + 2 + 1$

# Ciclos e Somatórios

**Progressão aritmética**: é uma sequência numérica em que cada termo, a partir do segundo, é igual à soma do termo anterior com uma constante $r$ (a **razão dessa sequência numérica**). Ao primeiro termo chamaremos $a_1$.

- $1, 2, 3, 4, 5, \ldots\ldots$ $(r = 1, a_1 = 1)$
- $3, 5, 7, 9, 11, \ldots\ldots$ $(r = 2, a_1 = 3)$

Como fazer um somatório de uma progressão aritmética?

$$1+2+3+4+5+6+7+8 = (1+8)+(2+7)+(3+6)+(4+5) = 4 \times 9$$

## Somatório de $a_p$ a $a_q$

$$S(p,q) = \sum_{i=p}^{q} a_i = \frac{(q-p+1)\times(a_p+a_q)}{2}$$

## Somatório dos primeiros $n$ termos

$$S_n = \sum_{i=1}^{n} a_i = \frac{n\times(a_1+a_n)}{2}$$

# Ciclos e Somatórios

## Um ciclo habitual

```
contador ← 0
Para i ← 1 até 1000 fazer
    Para j ← i até 1000 fazer
        contador ← contador + 1
escrever(contador)
```

O que escreve o programa?

$1000 + 999 + 998 + 997 + \ldots.. + 2 + 1$

Escreve $S_{1000} = \frac{1000 \times (1000+1)}{2} = 500500$

# Ciclos e Somatórios

## Um ciclo habitual

```
contador ← 0
Para i ← 1 até n fazer
    Para j ← i até n fazer
        contador ← contador + 1
escrever(contador)
```

Qual o tempo de execução?

Vai fazer $S_n$ passos:

$$S_n = \sum_{i=1}^{n} a_i = \frac{n \times (1+n)}{2} = \frac{n+n^2}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$$

O programa faz $\Theta(n^2)$ passos

# Ciclos e Somatórios

Quem quiser saber mais sobre somatórios interessantes para CC, pode espreitar o *Appendix A* do *Introduction to Algorithms*.

Notem que $c$ ciclos não implicam $\Theta(n^c)$!

## Ciclos

```
Para i ← 1 até n fazer
    Para j ← 1 até 5 fazer
```

$\Theta(n)$

## Ciclos

```
Para i ← 1 até n fazer
    Para j ← 1 até i × i fazer
```

$\Theta(n^3)$ $(1^2 + 2^2 + 3^2 + \ldots + n^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6})$

# Dividir para Conquistar

Muitos algoritmos podem ser expressos de forma **recursiva**,

Vários destes algoritmos seguem o paradigma de **dividir para conquistar**:

**Dividir para Conquistar**

**Dividir** o problema num conjunto de subproblemas que são instâncias mais pequenas do mesmo problema

**Conquistar** os subproblemas resolvendo-os recursivamente. Se o problema for suficientemente pequeno, resolvê-lo diretamente

**Combinar** as soluções dos problemas mais pequenos numa solução para o problema original

# Dividir para Conquistar

## Um Exemplo - MergeSort

Algoritmo **MergeSort** para ordenar um array de tamanho $n$

### MergeSort

**Dividir:** partir o array inicial em 2 arrays com metade do tamanho inicial

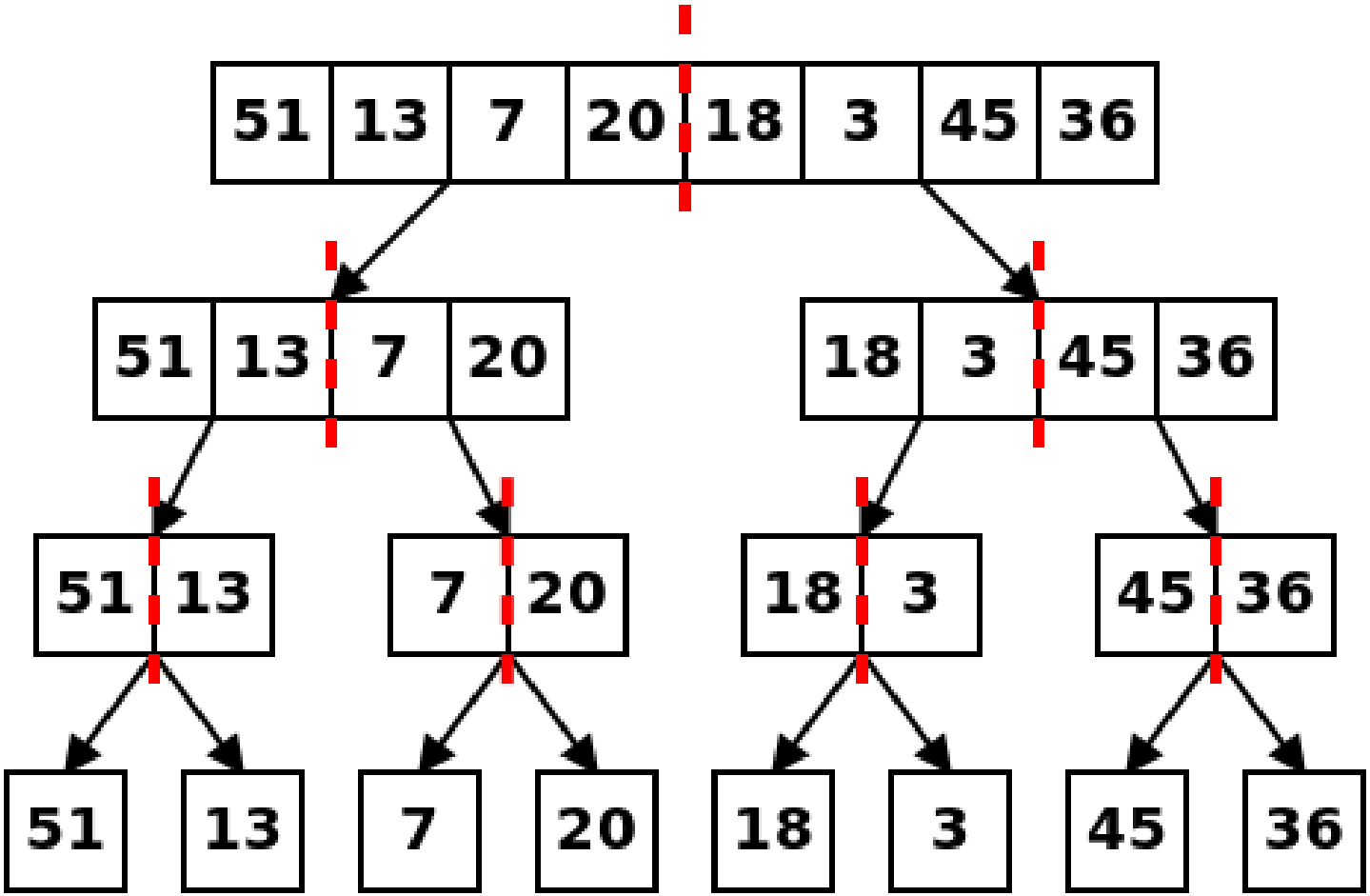
**Conquistar:** ordenar recursivamente as 2 metades. Se o problema for ordenar um array de apenas 1 elemento, basta devolvê-lo.

**Combinar:** fazer uma junção (*merge*) das duas metades ordenadas para um array final ordenado.

# Dividir para Conquistar

## Um Exemplo - MergeSort

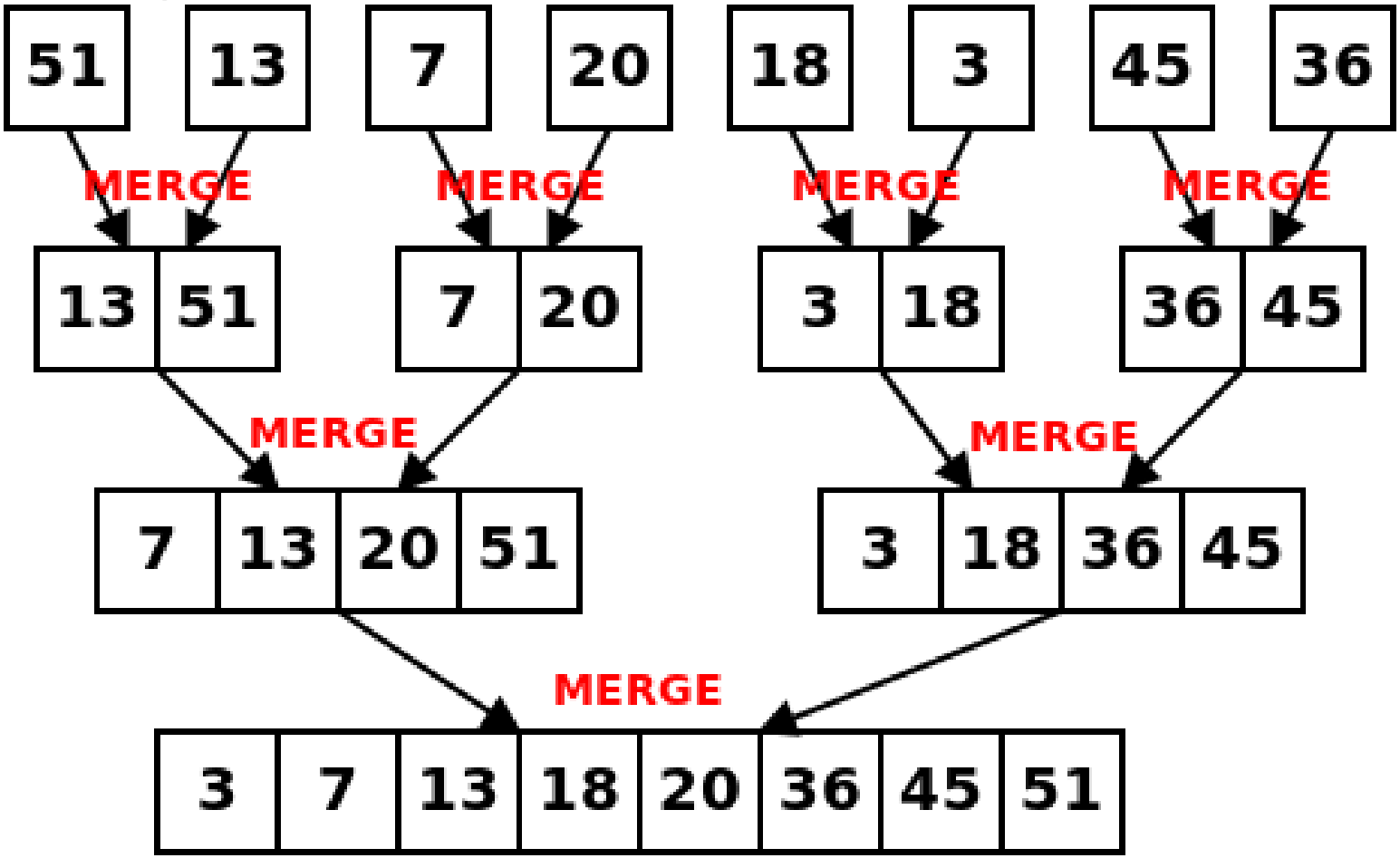
**Dividir:**



# Dividir para Conquistar

## Um Exemplo - MergeSort

### Conquistar:



# Dividir para Conquistar

## Um Exemplo - MergeSort

Qual o **tempo de execução** deste algoritmo?

- **D(n)** - Tempo para partir um array de tamanho $n$ em 2
- **M(n)** - Tempo para fazer um *merge* de 2 arrays de tamanho $n/2$
- **T(n)** - Tempo total para um MergeSort de um array de tamanho $n$

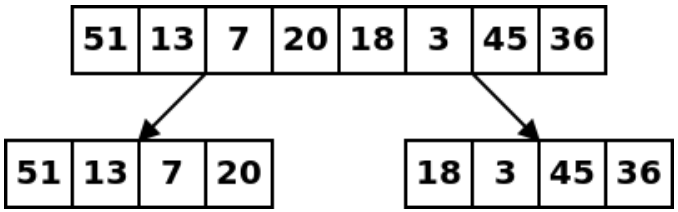
Para simplificar vamos assumir que $n$ é uma potência de 2.
(as contas são muito parecidas nos outros casos)

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{se } n > 1 \end{cases}$$

# Dividir para Conquistar

## Um Exemplo - MergeSort

**D(n)** - Tempo para partir um array de tamanho $n$ em 2




Não preciso de criar uma cópia do array!

Usemos uma função com 2 argumentos:
`mergesort(a,b)`: (ordenar desde a posição $a$ até posição $b$)

No início, `mergesort(0, n-1)` (com arrays começados em 0)

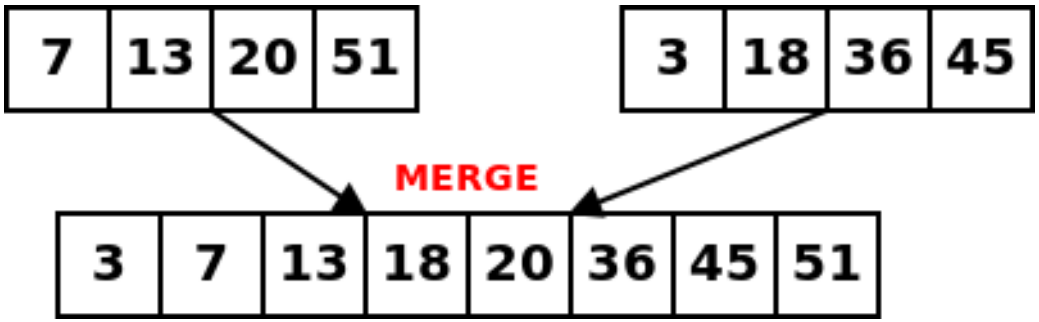
Seja $m = \lfloor (a+b)/2 \rfloor$ a posição do meio.
Chamadas a `mergesort(a,m)` e `mergesort(m+1,b)`

Só preciso de fazer uma conta (soma + divisão)
Consigo fazer divisão em $\Theta(1)$ (tempo constante!)

# Dividir para Conquistar

## Um Exemplo - MergeSort

**M(n)** - Tempo para fazer um *merge* de 2 arrays de tamanho $n/2$

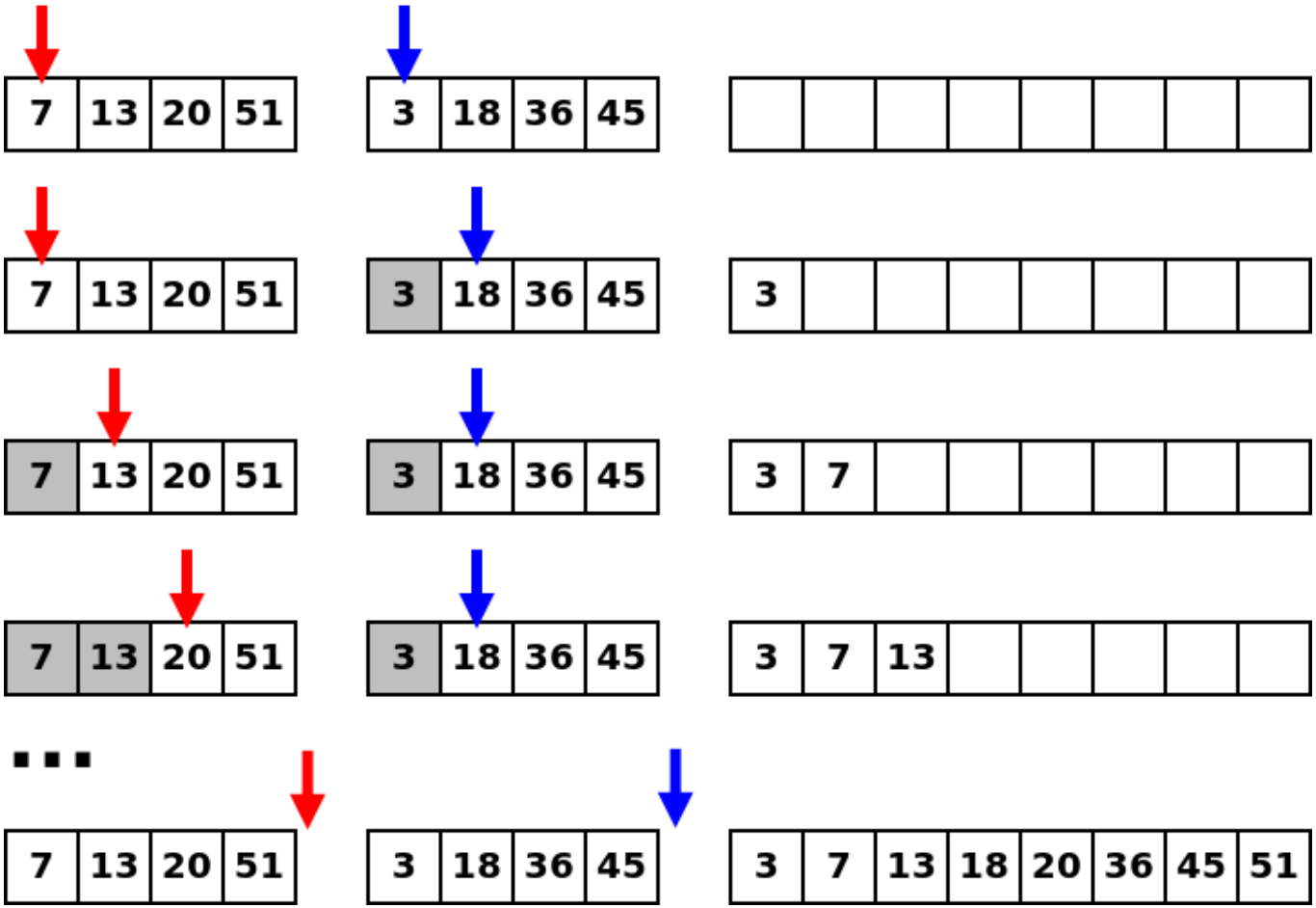



Em tempo constante não é possível. E em tempo **linear**?

# Dividir para Conquistar

## Um Exemplo - MergeSort

**M(n)** - Tempo para fazer um *merge* de 2 arrays de tamanho $n/2$




No final fiz $n$ comparações+cópias. Gasto $\Theta(n)$ (tempo linear!)

# Dividir para Conquistar

## Um Exemplo - MergeSort

De volta à recorrência do MergeSort:

- **D(n)** - Tempo para partir um array de tamanho $n$ em 2
- **M(n)** - Tempo para fazer um *merge* de 2 arrays de tamanho $n/2$
- **T(n)** - Tempo total para um MergeSort de um array de tamanho $n$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{se } n > 1 \end{cases}$$

fica

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

# Recorrências

## Detalhes

Para inputs suficientemente pequenos, um algoritmo demora normalmente tempo constante.

Isto significa que para um $n$ pequeno, temos que $T(n) = \Theta(1)$

Por conveniência, podemos geralmente **omitir o caso base da recorrência**.

Exemplos:

- Mergesort: $T(n) = 2T(n/2) + \Theta(n)$
- Pesquisa Binária: $T(n) = T(n/2) + \Theta(1)$
- Descobrir o máximo com *tail recursion*: $T(n) = T(n-1) + \Theta(1)$

Como resolver recorrências como esta?

# Recorrências

## Resolvendo

Vamos falar de 2 métodos (existem outros como "desenrolar" a recorrência ou "adivinhar" o resultado e depois provar por indução):

- **Árvore de Recursão:** desenhar uma árvore representando a recursão e somar toda a computação feita nos seus nós
- ***Master Theorem*:** Se a recorrência for da forma $\mathbf{aT(n/b) + cn^k}$, a resposta segue um padrão

# Recorrências

## Árvore de Recursão

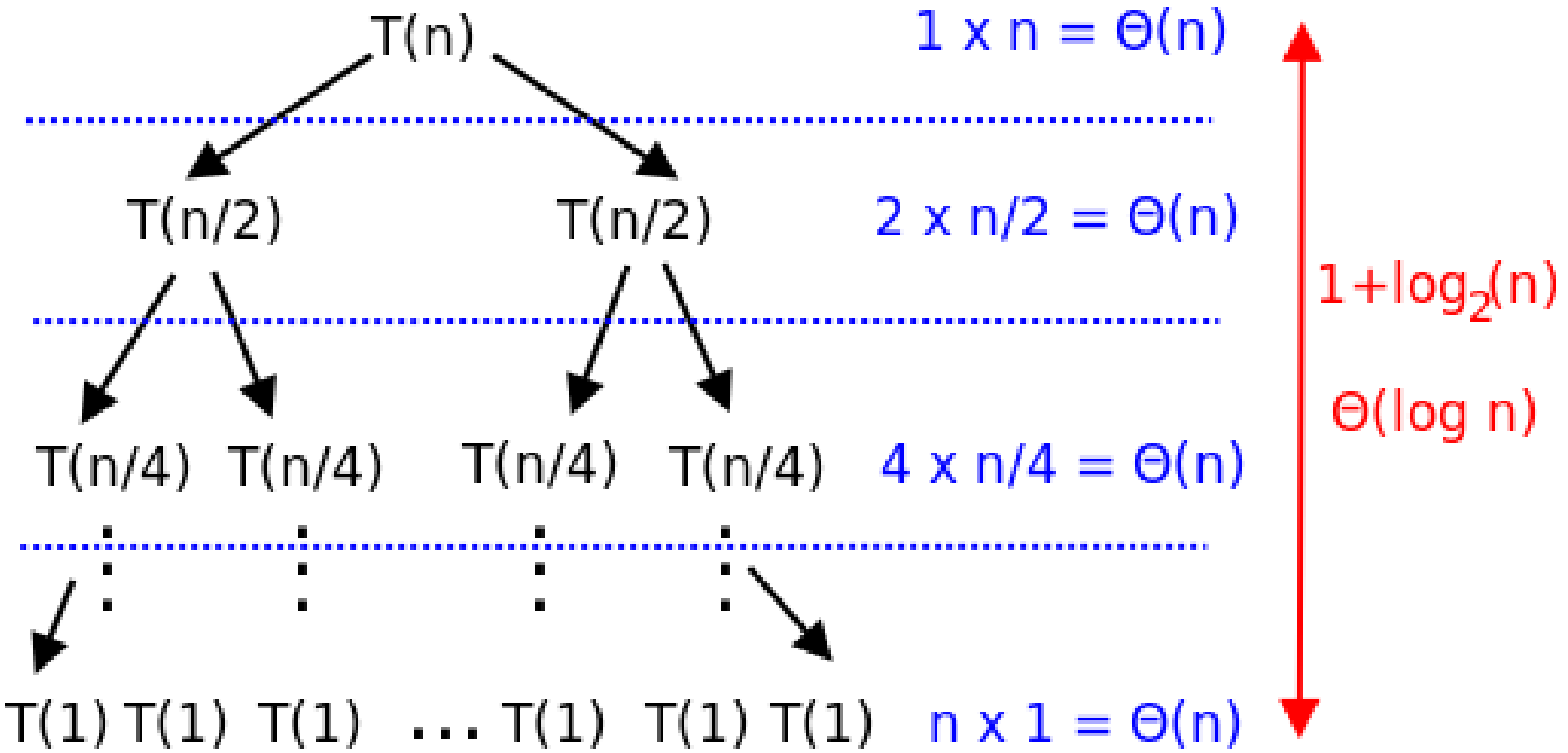
Um método possível **é desenhar a árvore de recursão** e analisá-la, somando todo a computação feita em cada um dos seus nós

Vamos tentar usar este método com o MergeSort: $T(n) = 2(n/2) + n$

(para uma explicação mais simples vamos assumir que $n = 2^k$, mas os resultados são válidos para qualquer $n$)

# Recorrências

## Árvore de Recursão




Somando tudo temos que o **MergeSort** é $\Theta(n \log_2 n)$

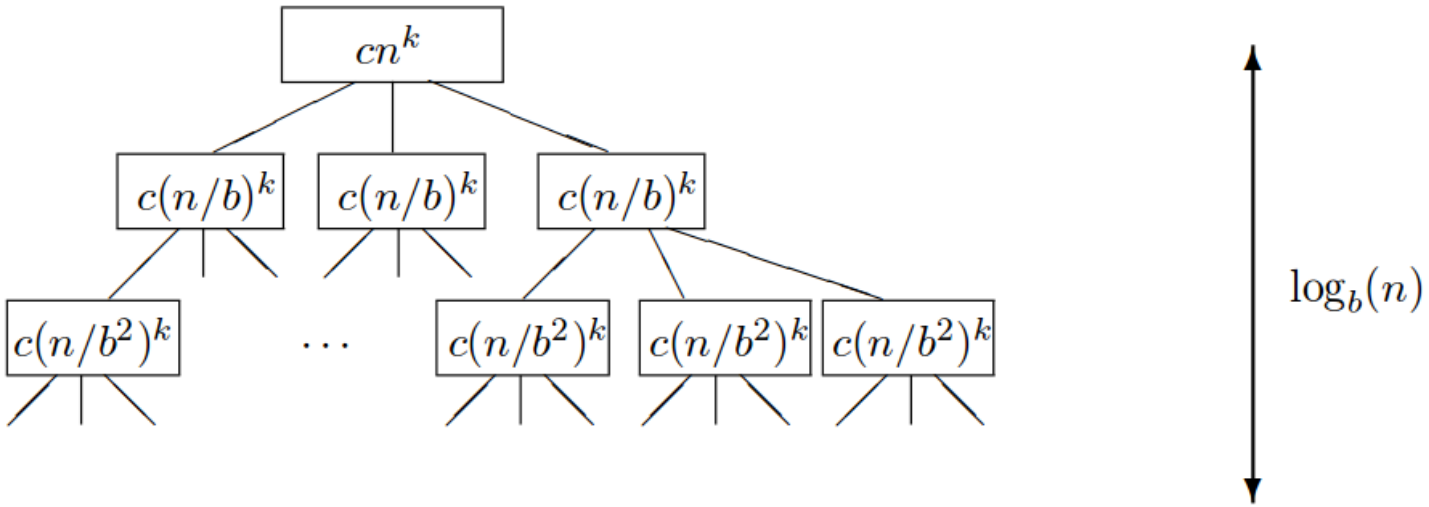
# Recorrências

## Master Theorem

Podemos usar o **master theorem** para recorrências da forma:

$$\mathbf{T(n) = aT(n/b) + cn^k}$$

Isto é muito útil para recorrências vindas de um algoritmo de dividir para conquistar que divide o problema em **a** pedaços, cada um de tamanho **n/b** e que demora tempo $\mathbf{cn^k}$ para dividir+combinar.

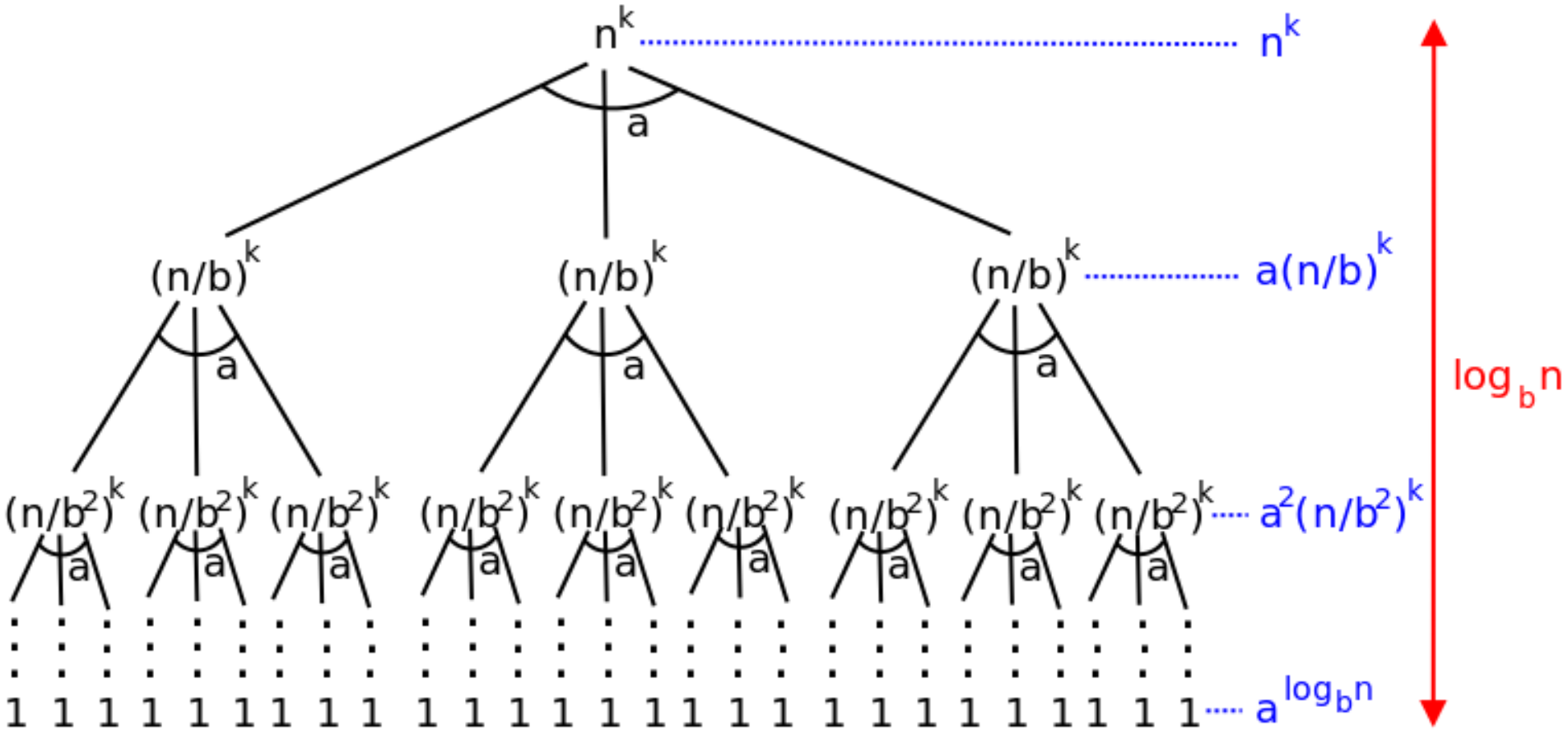



No caso do MergeSort temos que $a = 2, b = 2, k = 1$.

# Master Theorem

## Intuição

$\mathbf{aT(n/b) + n^k}$ (assumimos $c = 1$ para uma explicação mais simples)



# Master Theorem

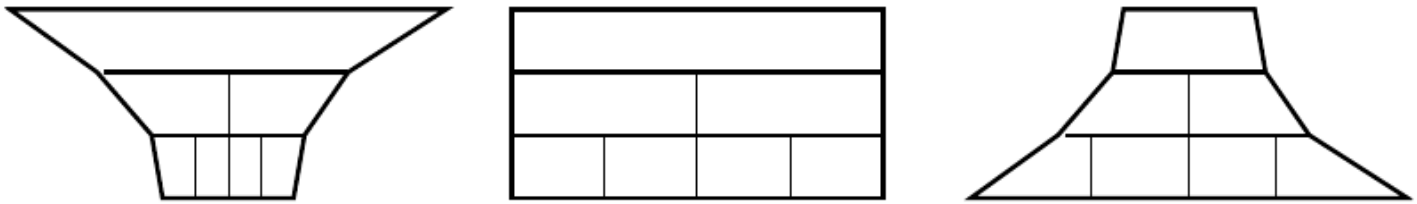
## Intuição

- **Raíz (primeiro nível):** $n^k$
- **Profundidade $i$ (meio da árvore):**
  $a^i(n/b^i)^k = a^i/b^{ik}n^k = (a/b^k)^i n^k$
- **Folhas (último nível):** $a^{\log_b n} = n^{\log_b a}$

Portanto o peso da profundidade $i$ é: $\mathbf{(a/b^k)^i n^k}$

(1) $a < b^k$ implica que $a/b^k$ é menor que 1 (peso vai diminuindo)
(2) $a = b^k$ implica que $a/b^k$ é igual a 1 (peso é constante)
(3) $a > b^k$ implica que $a/b^k$ é maior que 1 (peso vai crescendo)

- (1) O tempo é dominado pelo **primeiro nível**
- (2) O tempo está (uniformemente) **distribuído** por todos os níveis
- (3) O tempo é dominado pelo **último nível**



# Master Theorem

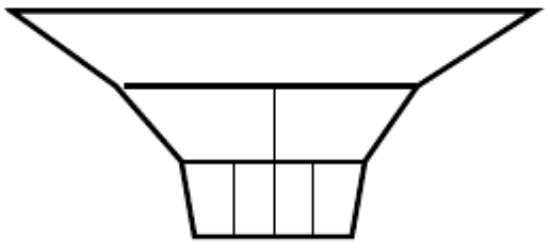
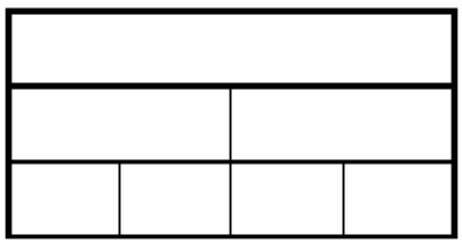
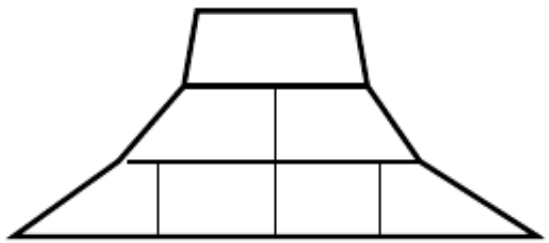
## Master Theorem - Uma versão prática

Uma recorrência $\mathbf{aT(n/b) + cn^k}$ ($a \geq 1, b > 1, c$ e $k$ são constantes positivas) resolve para:

(1) $T(n) = \Theta(n^k)$ se $a < b^k$
(2) $T(n) = \Theta(n^k \log n)$ se $a = b^k$
(3) $T(n) = \Theta(n^{\log_b a})$ se $a > b^k$

Se pensarem na árvore de recursão, intuitivamente, estes 3 casos correspondem a:

- (1) O tempo é dominado pelo **primeiro nível**
- (2) O tempo está (uniformemente) **distribuído** por todos os níveis
- (3) O tempo é dominado pelo **último nível**

# Master Theorem

## Master Theorem - Uma versão prática

Uma recorrência $\mathbf{aT(n/b) + cn^k}$ ($a \geq 1, b > 1, c$ e $k$ são constantes positivas) resolve para:

(1) $T(n) = \Theta(n^k)$ se $a < b^k$
(2) $T(n) = \Theta(n^k \log n)$ se $a = b^k$
(3) $T(n) = \Theta(n^{\log_b a})$ se $a > b^k$

**Exemplo de um caso (1)**:

$T(n) = 2T(n/2) + n^2$

$a = 2, b = 2, k = 2,\ a < b^k$ porque $2 < 4$.

A recorrência resolve para $\mathbf{\Theta(n^2)}$

# Master Theorem

## Master Theorem - Uma versão prática

Uma recorrência $\mathbf{aT(n/b) + cn^k}$ ($a \geq 1, b > 1, c$ e $k$ são constantes positivas) resolve para:

(1) $T(n) = \Theta(n^k)$ se $a < b^k$
(2) $T(n) = \Theta(n^k \log n)$ se $a = b^k$
(3) $T(n) = \Theta(n^{\log_b a})$ se $a > b^k$

**Exemplo de um caso (2)**:

$T(n) = 2T(n/2) + n$ *(ex: MergeSort)*

$a = 2, b = 2, k = 1$, $a = b^k$ porque $2 = 2$.

A recorrência resolve para $\mathbf{\Theta(n \log n)}$ (como já sabíamos).

# Master Theorem

## Master Theorem - Uma versão prática

Uma recorrência $\mathbf{aT(n/b) + cn^k}$ ($a \geq 1, b > 1, c$ e $k$ são constantes positivas) resolve para:

(1) $T(n) = \Theta(n^k)$ se $a < b^k$
(2) $T(n) = \Theta(n^k \log n)$ se $a = b^k$
(3) $T(n) = \Theta(n^{\log_b a})$ se $a > b^k$

**Exemplo de um caso (3)**:

$T(n) = 2T(n/2) + 1$

$a = 2, b = 2, k = 0$, $a > b^k$ porque $2 > 1$.

A recorrência resolve para $\mathbf{\Theta(n)}$

# Master Theorem

## Revisitando os exemplos

Exemplos:

(1) $T(n) = 2T(n/2) + n^2 = \Theta(n^2)$
$n^2 + n^2/2 + n^2/4 + \ldots + n \leftarrow$ ($n^2$ domina, ou seja, a raíz)

(2) $T(n) = 2T(n/2) + n = \Theta(n \log n)$
$n + n + \ldots + n \leftarrow$ (distribuído por todos os níveis)

(3) $T(n) = 2T(n/2) + 1 = \Theta(n)$
$1 + 2 + 4 + \ldots + n \leftarrow$ ($n$ domina, ou seja, as folhas)

# Master Theorem

Apenas por completude, aqui fica a versão do *master theorem* apresentada no livro **"Introduction to Algorithms"**.

## Master Theorem - uma versão mais geral

Uma recorrência $\mathbf{aT(n/b) + f(n)}$ ($a \geq 1, b > 1$ são constantes) resolve para:

(1) Se $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$

(2) Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$

(3) Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e para todos os $n$ suficientemente grandes, então $T(n) = \Theta(f(n))$

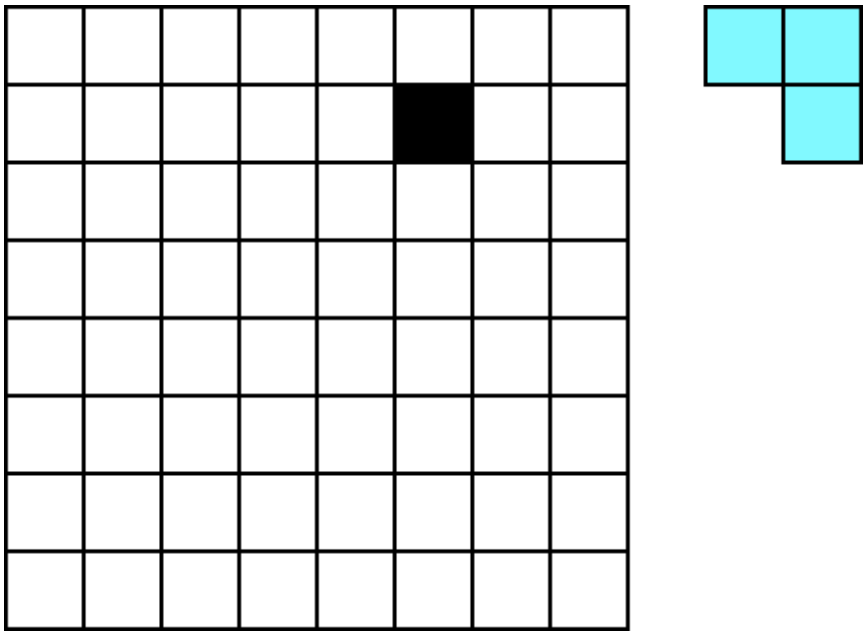
(os casos 1 e 3 estão invertidos em relação à versão prática que mostrei)

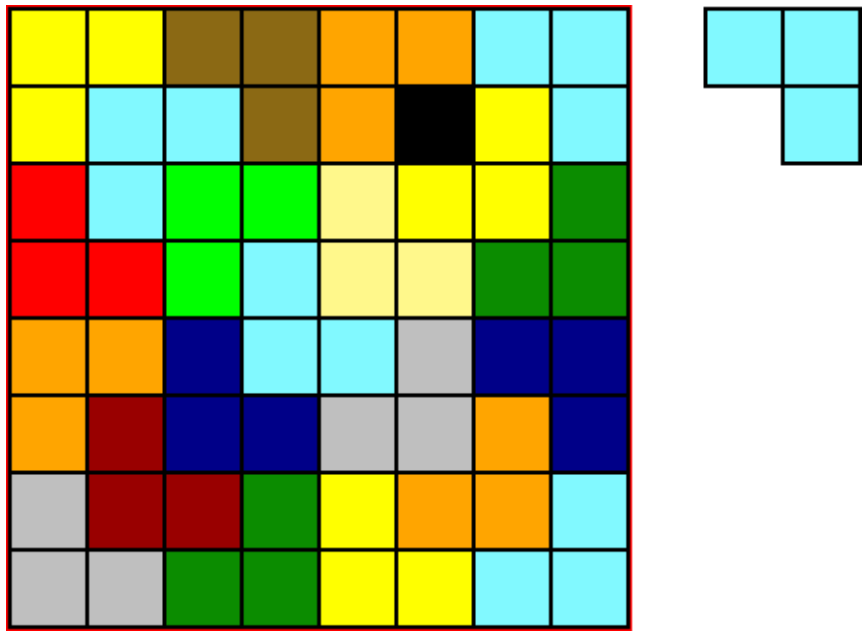
# Dividir para Conquistar

## Um Puzzle

Até "manualmente" se pode usar esta técnica de desenho algorítmico.

Imagine que tem uma grelha (ou matriz) de $2^n \times 2^n$ e quer **preencher todas as quadrículas** com peças com o formato de um L.
As peças podem ser rodadas e a grelha inicial tem um casa "proibida".





Uma ideia é **dividir em 4** quadrados mais pequenos... e colocar uma peça!