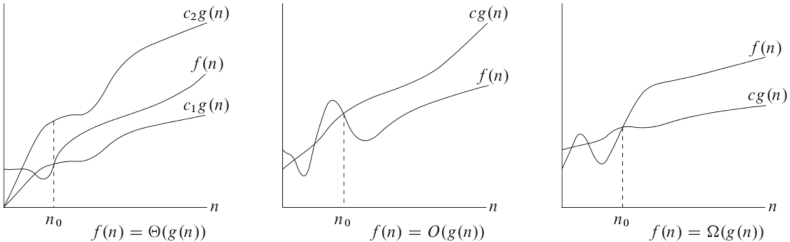
Noções de Complexidade Algorítmica

Pedro Ribeiro

DCC/FCUP

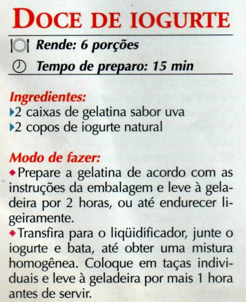
2021/2022



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



DOCE DE IOGURTE

Rende: 6 porções

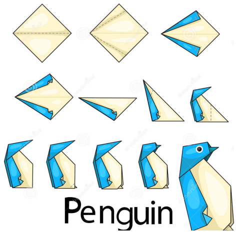
Tempo de preparo: 15 min

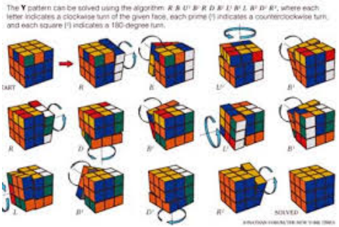
Ingredientes:

- 2 caixas de gelatina sabor uva
- 2 copos de iogurte natural

Modo de fazer:

- Prepare a gelatina de acordo com as instruções da embalagem e leve à geladeira por 2 horas, ou até endurecer ligeiramente.
- Transfira para o liqüidificador, junte o iogurte e bata, até obter uma mistura homogênea. Coloque em taças individuais e leve à geladeira por mais 1 hora antes de servir.





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?

- Os métodos das estruturas de dados que temos implementado também são (pequenos) algoritmos. Exemplos:
  - *addFirst*, *addLast*, *removeFirst* e *removeLast* nas listas ligadas
  - *contains*, *add* e *remove* no TAD Conjunto
  - *push* e *pop* no TAD Pilha
  - *enqueue* e *dequeue* no TAD Fila

- Outros algoritmos podem usar as estruturas de dados como "legos básicos". Exemplos:
  - O algoritmo de escalonamento *round-robin* de processos usa uma pilha
  - O algoritmo para verificar se uma expressão tem os parenteses bem balanceados usa uma pilha
  - Um algoritmo para simular o atendimento em balcões de um banco, de um aeroporto ou de uma loja do cidadão usa uma fila

Propriedades desejadas num algoritmo

Correção

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

Eficiência

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

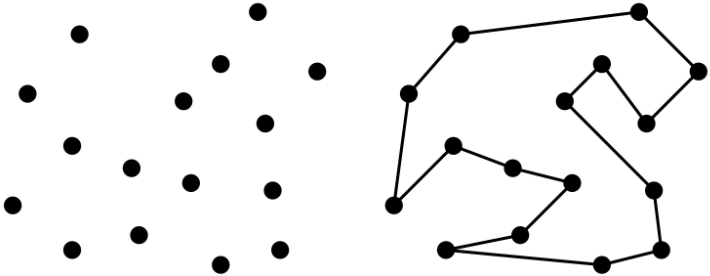
Correção de um algoritmo

- **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

Problema do Caixeiro Viajante (Euclidean TSP)

Input: um conjunto S de n pontos no plano
Output: Um caminho que começa num ponto, visita todos os outros pontos de S, e regressa ao ponto inicial.

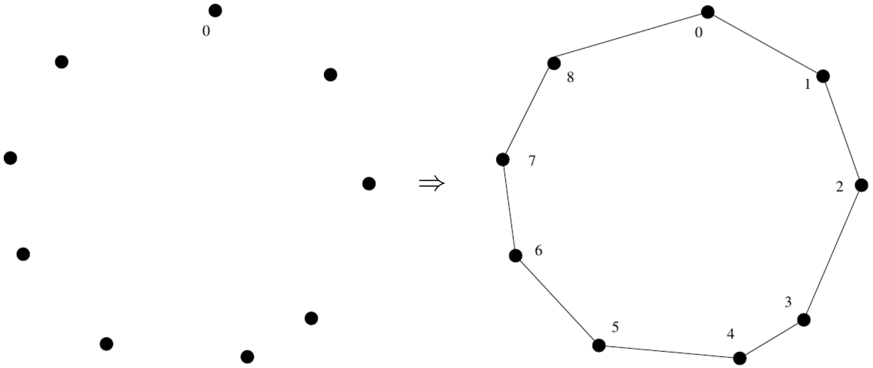
Um exemplo:



Um 1º possível algoritmo (vizinho mais próximo)

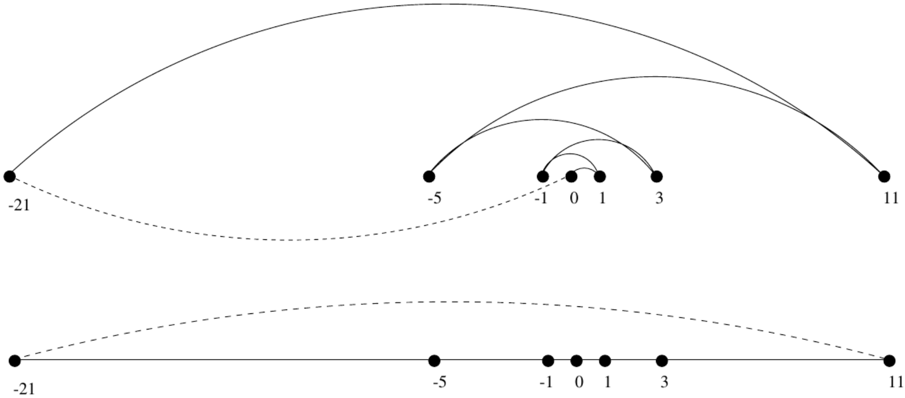
$p_1 \leftarrow$ ponto inicial escolhido aleatoriamente
$i \leftarrow 1$
Enquanto (existirem pontos por visitar) **fazer**
    $i \leftarrow i + 1$
    $p_i \leftarrow$ vizinho não visitado mais próximo de p_{i-1}
retorna caminho $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \rightarrow p_1$

Parece funcionar...



Mas não funciona para todas as instâncias!

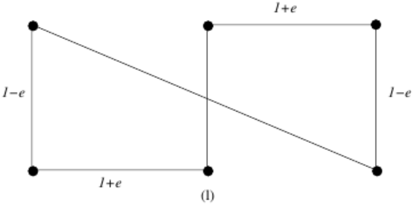
(Nota: começar pelo ponto mais à esquerda não resolveria o problema)

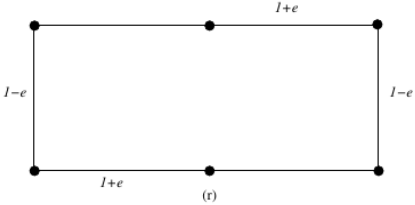


Um 2º possível algoritmo (par mais próximo)

Para $i \leftarrow 1$ **até** $(n-1)$ **fazer**
    Adiciona ligação ao par de pontos mais próximo tal que os pontos estão em componentes conexas (cadeias de pontos) diferentes
Adiciona ligação entre dois pontos dos extremos da cadeia ligada
retorna o ciclo que formou com os pontos

Também não funciona para todas as instâncias!





Como resolver então o problema?

Um 3º 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?)!

- 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 nas próximas aulas)
- 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?)

- 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**)

n ≤ 7: $< 0.001s$
n $= 8$: $0.001s$
n $= 9$: $0.016s$
n $= 10$: $0.185s$
n $= 11$: $2.204s$
n $= 12$: $28.460s$
...
n $= 20$: 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

Eficiência de um algoritmo

Perguntas

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

- Vamos ver 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: $+$, $-$, $\leftarrow$, **If**) 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: $T(n)$**
  (n é o tamanho do input)
- 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

Exemplo com um pequeno programa:

```
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)

Exemplo com um pequeno programa:

```
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

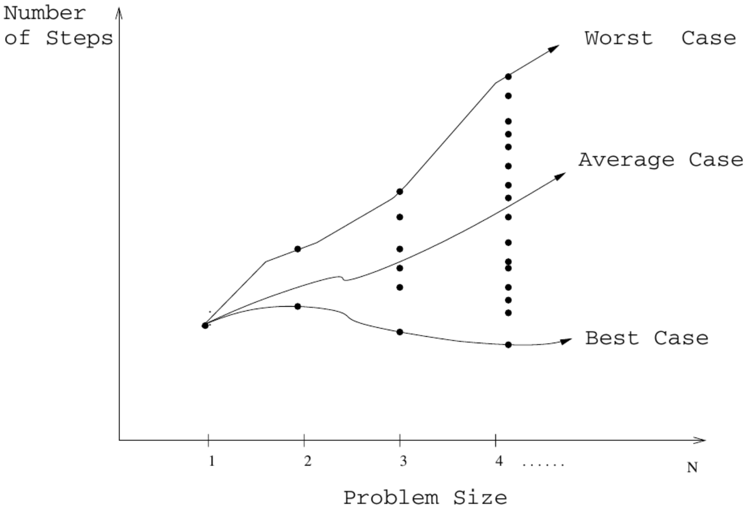
Análise **Caso Médio**: **(por vezes)**

- $T(n) =$ 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, ω)
- Permite "simplificar" expressões como a anteriormente mostrada focando apenas nas **ordens de grandeza**

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

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

$f(n) \in \Omega(g(n))$ **(minorante)**

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

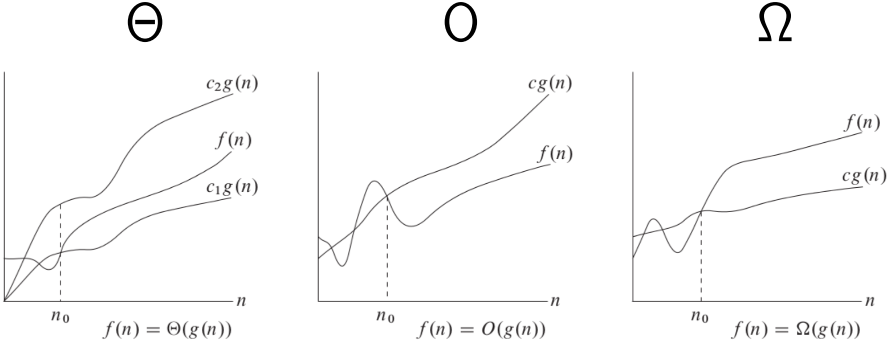
$f(n) \in \Theta(g(n))$ **(limite "apertado" - majorante e minorante)**

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 $f(n)$ *(a partir de um dado n)*

Onde c, c_1 e c_2 são constantes

Notação

Uma ilustração



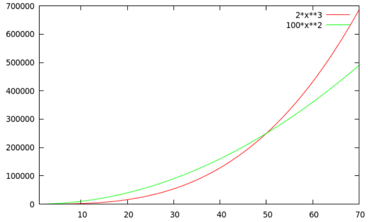
As definições implicam um n a partir do qual a função é majorada e/ou minorada. Valores pequenos de n "não importam".

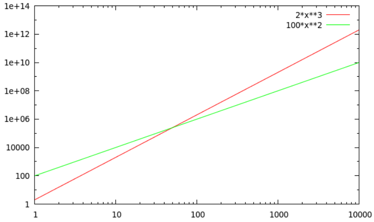
Nota: Alguma bibliografia usa $=$ em vez de $\in$
Exemplo: $f(n) = \mathcal{O}(g(n))$ é o mesmo que $f(n) \in \mathcal{O}(g(n))$

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:

- 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 $n \geq n_0$
- 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) $\in \Theta$(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:

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

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:

$f(n) \in O(g(n))$	é como	$a \leq b$	limite superior	pelo menos tão bom como
$f(n) \in \Omega(g(n))$	é como	$a \geq b$	limite inferior	pelo menos tão mau como
$f(n) \in \Theta(g(n))$	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**:
  $3\text{n}^3 - 5n^2 + 100 \in \Theta(n^3)$
  $6\text{n}^4 - 20^2 \in \Theta(n^4)$
  $0.8\text{n} + 224 \in \Theta(n)$

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

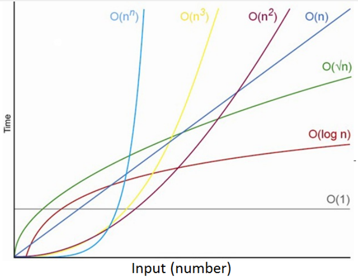
Crescimento Assintótico

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

- Se queremos minimizar o tempo (ou espaço), **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

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



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			

Previsão do tempo de execução

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)

$\mathrm{tempo}(\mathrm{n}_2) = \mathrm{f}(\mathrm{n}_2)/\mathrm{f}(\mathrm{n}_1) \times \mathrm{tempo}(\mathrm{n}_1)$

Previsão do tempo de execução

Um exemplo:

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

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

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

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) × tempo(n)

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

- 2$^\mathrm{n}$: $2^{2n}/2^n = 2^{2n-n} = 2^n$. 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!

- **log**$_2$(n) : $\log_2(2n)/\log_2(n)$. Aumenta $\frac{\log_2(2n)}{\log_2(n)}$ 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!

- **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, $\log_2(n) \in \mathcal{O}(\sqrt{n})$

- **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, $\log_2 n \in \Theta(\log_3 n)$

- 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.

- `SinglyLinkedList<T>` (tem atributos *first* e *size*)
  - `size()`: $\Theta(1)$
  - `isEmpty()`: $\Theta(1)$
  - `getFirst()`: $\Theta(1)$
  - `removeFirst()`: $\Theta(1)$
  - `getLast()`: $\Theta(n)$
  - `removeLast()`: $\Theta(n)$

As complexidades são as mesmas para `CircularLinkedList` e `DoublyLinkedList` com excepção do `getLast()` e `removeLast()`

- `CircularLinkedList`: (tem atributos *last* e *size*)
  - `getLast()`: $\Theta(1)$
  - `removeLast()`: $\Theta(n)$
- `DoublyLinkedList`: (nós têm *prev* e *next*)
  - `getLast()`: $\Theta(1)$
  - `removeLast()`: $\Theta(1)$

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

```java
int count = 0;
for (int i=0; i<1000; i++)
   for (int j=i; j<1000; j++)
      count++;
System.out.println(count);
```

(a complexidade temporal é proporcional ao valor de *count* no final)

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 x_1.

- $1, 2, 3, 4, 5, \ldots \ldots$ $(r = 1, x_1 = 1)$
- $3, 5, 7, 9, 11, \ldots \ldots$ $(r = 2, x_1 = 3)$

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 x_a a x_b

$$S(a, b) = \sum_{i=a}^{b} x_i = \frac{(b-a+1) \times (x_a + x_b)}{2}$$

Somatório dos primeiros n termos

$$S_n = \sum_{i=1}^{n} x_i = \frac{n \times (x_1 + x_n)}{2}$$

Ciclos e Somatórios

```java
int count = 0;
for (int i=0; i<1000; i++)
   for (int j=i; j<1000; j++)
      count++;
System.out.println(count);
```

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

```java
int count = 0;
for (int i=0; i<n; i++)
   for (int j=i; j<n; j++)
      count++;
System.out.println(count);
```

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 $\Theta(n^c)$!

```
for (int i=0; i<n; i++)
   for (int j=1; j<5; j++)
```

$\Theta(n)$

```
for (int i=1; i<=n; i++)
   for (int j=1; j<=i*i; j++)
```

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

```
i = n;
while (i>0) i = i/2;
```

$\Theta(\log(n))$ (de cada vez i fica reduzido a metade)

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

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

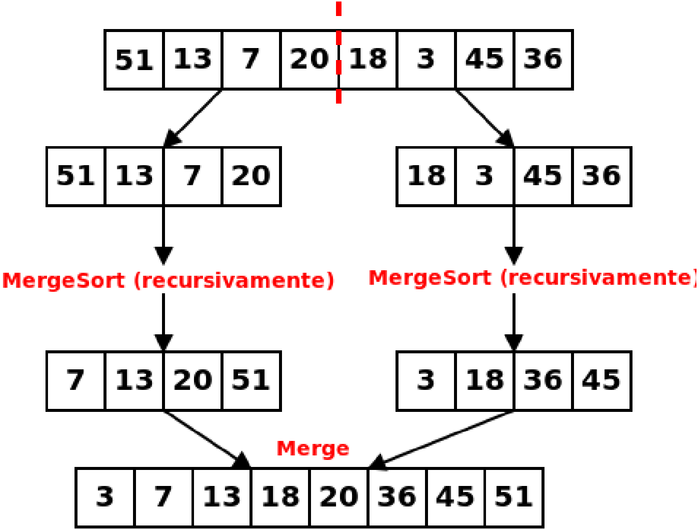
MergeSort

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

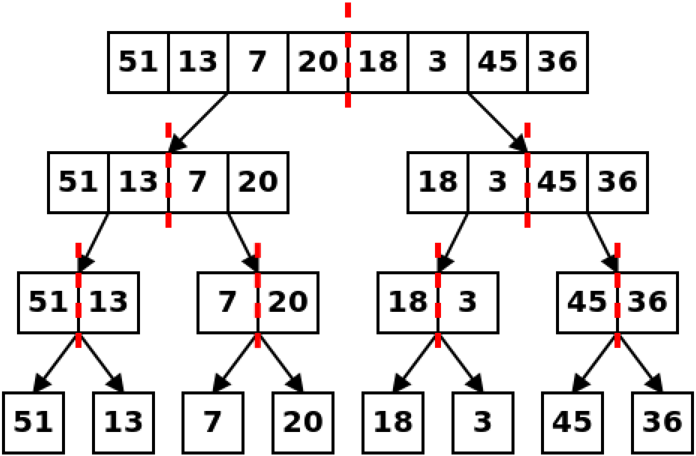
Conquistar: ordenar recursivamente as 2 metades. Se o problema for ordenar um array de apenas 1 elemento, basta devolvê-lo.

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

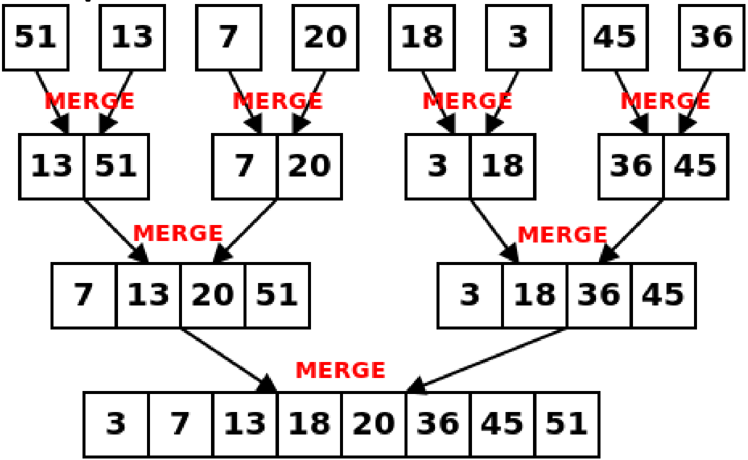
O que acontece do ponto de vista da primeira chamada à função recursiva:



Dividir:


51 13 7 20 18 3 45 36
51 13 7 20
18 3 45 36
51 13
7 20
18 3
45 36
51
13
7
20
18
3
45
36

Conquistar:


51
13
7
20
18
3
45
36
MERGE
MERGE
MERGE
MERGE
13 51
7 20
3 18
36 45
MERGE
MERGE
7 13 20 51
3 18 36 45
MERGE
3 7 13 18 20 36 45 51

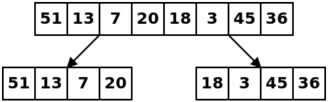
Qual o **tempo de execução** deste algoritmo?

- 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)

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

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



Não preciso de criar uma cópia do array!

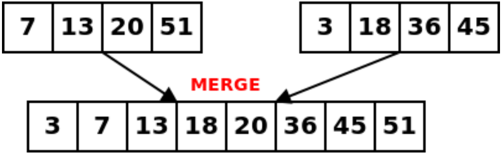
Usemos uma função com 2 argumentos:
`mergesort(a,b)`: (ordenar desde a posição *a* até posição *b*)

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)`

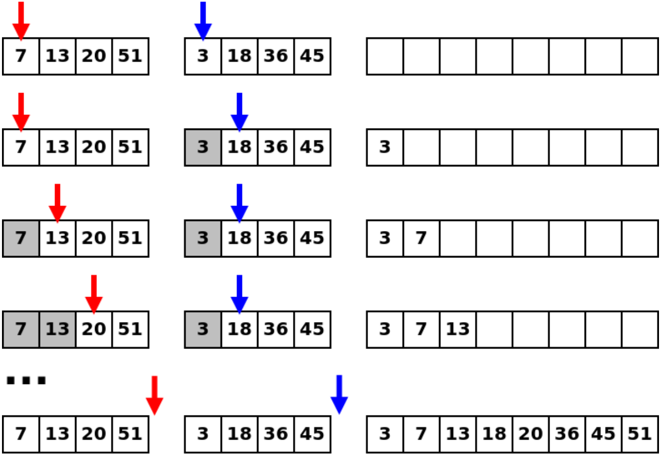
Só preciso de fazer uma conta (soma + divisão)
Consigo fazer divisão em $\Theta(1)$ (tempo constante!)

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



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

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



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

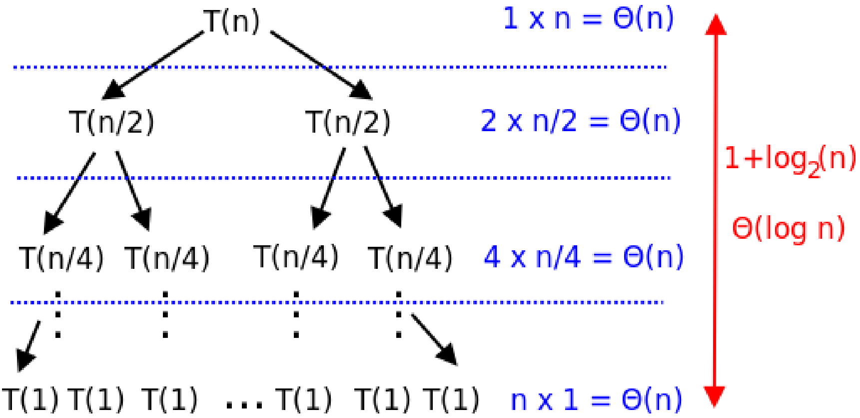
Qual é então o **tempo de execução** do MergeSort?

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

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

Como **resolver** esta recorrência?

Vamos desenhar a **árvore de recorrência**:



O total é o somatório disto tudo: **MergeSort é** $\Theta(n \log n)$ **!**

Nem sempre um algoritmo recursivo tem complexidade **linearítmica**!

Vamos ver um outro exemplo. Imagine que tem um array de n elementos e quer **descobrir o máximo**.

Uma simples **pesquisa linear** chegava, mas vamos desenhar um algoritmo seguindo as ideias do dividir para conquistar.

Descobrir o máximo

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

Conquistar: calcular recursivamente o máximo de cada uma das metades

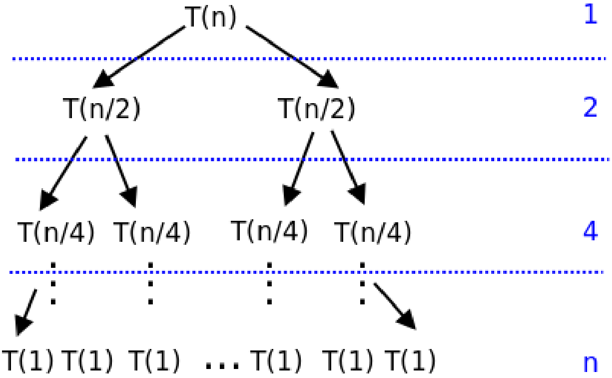
Combinar: comparar o máximo de cada uma das metades e ficar com o maior deles

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

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

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

O que tem esta recorrência de diferente da do MergeSort?
Como a **resolver**?



No total gasta $1 + 2 + 4 + \ldots + n = \sum_{i=0}^{\log_2(n)} 2^i$

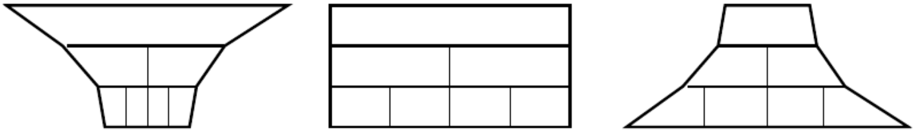
O que domina a soma? Note que $2^k = 1 + \sum_{i=0}^{k-1} 2^i$.

O último nível domina o peso da árvore e logo, o algoritmo é $\Theta(n)$!

Nem todas as recorrências de um algoritmo de **dividir para conquistar** dão origem a complexidades **logarítmicas** ou **linearítmicas**.

Na realidade, temos tipicamente **três tipos de casos**:

- O tempo é repartido de maneira mais ou menos **uniforme por todos os níveis** da recursão (ex: mergesort)
- O tempo é dominado pelo **último nível** da recursão (ex: máximo)
- O tempo é dominado pelo **primeiro nível** da recursão (ex: multiplicação de matrizes "naive")



(para saber mais podem espreitar o Master Theorem)

É usual assumir que $T(1) = \Theta(1)$. Nesses casos podemos escrever apenas a parte de $T(n)$ para descrever uma recorrência.

- **MergeSort:** $T(n) = 2T(n/2) + \Theta(n)$
- **MáximoD&C:** $T(n) = 2T(n/2) + \Theta(1)$

Por vezes temos um algoritmo que reduz um problema a um único subproblema.

Nesses casos podemos dizer que usamos **diminuir e conquistar** (decrease and conquer).

- **Pesquisa Binária:**
  Num array ordenado de tamanho n, comparar com o elemento do meio e procurar na metade correspondente
  $T(n) = T(n/2) + \Theta(1)$ [$\Theta(logn)$]

- **Máximo com "tail recursion":** Num array de tamanho n, recursivamente descobrir o máximo do array excepto o primeiro elemento e depois comparar com o primeiro elemento.
  $T(n) = T(n-1) + \Theta(1)$ [$\Theta(n)$]