Projeto e Análise de Algoritmos

A. G. Silva, R. de Santiago

Baseado nos materiais de
Souza, Silva, Lee, Rezende, Miyazawa – Unicamp
Ribeiro – FCUP • Mariani – UFSC
Manber, Introduction to Algorithms (1989) – Livro

05 de abril de 2024

Cronograma

- **15mar** – Apresentação da disciplina. Introdução. Apresentação da L_1.
- **22mar** – Prova de proficiência/validação.
- **29mar** – *Dia não letivo*
- **05abr** – Notação assintótica. Indução matemática.
- **12abr** – Indução matemática. Recorrências. Divisão e conquista.
- **19abr** – Divisão e conquista. Ordenação.
- **26abr** – Ordenação em tempo linear. Multiplicação de inteiros. Multiplicação de matrizes.
- **03mai** – Estatística de ordem. Dúvidas.
- **10mai** – Primeira avaliação Q_1. Entrega de L_1.
- **17mai** – Grafos. Buscas. Apresentação da L_2.
- **24mai** – Algoritmos gulosos.
- **31mai** – *Dia não letivo*
- **07jun** – *Semana Acadêmica do PPGCC.*
- **14jun** – Programação dinâmica.
- **21jun** – NP-Completude e reduções.
- **28jun** – Algoritmos aproximados e heurísticas.
- **05jul** – Segunda avaliação Q_2. Entrega de L_2.
- **12jul** – Dúvidas e fechamento.

Algoritmo

- Um algoritmo é um **método** para resolver um problema (computacional)
- Um algoritmo é uma **ideia** por trás de um programa e é independente de linguagem de programação, máquina, etc
- **Propriedades** de um algoritmo:

Correção

Deve resolver corretamente **todas as instâncias** do problema

Eficiência

O desempenho (**tempo** e **memória**) deve ser adequado

- Este curso é sobre a **concepção** e **análise** de algoritmos corretos e eficientes

Preocupações

- Importância da análise do tempo de execução

Predição

Quanto tempo um algoritmo precisa para resolver um problema? Qual a escala? Podemos ter garantias sobre o tempo de funcionamento?

Comparação

Um algoritmo *A* é melhor que um algoritmo *B*? Qual é a melhor forma de resolvermos um determinado problema?

- Estudaremos uma **metodologia** para responder a essas questões

Velocidade de computadores

Desempenho algorítmico $\times$ Velocidade de computação

Um algoritmo melhor em um computador mais lento **sempre vencerá** um algoritmo pior em um computador mais rápido, para instâncias suficientemente grandes

- O que realmente importa é a **taxa de crescimento** do tempo de execução!

Random Access Machine (RAM)

- Precisamos de um **modelo genérico** e **independente** de linguagem e de máquina.
- *Random Access Machine* (RAM)
  - Cada **operação simples** (ex.: $+$, $-$, $\leftarrow$, **If**) leva **1 passo**
  - Ciclos e procedimentos, por exemplo, não são instruções simples
  - Cada **acesso à memória** leva também **1 passo**
- Podemos medir o tempo de execução contando o número de passos como uma função do tamanho de entrada: $T(n)$
- Operações são **simplificadas**, mas isto é útil
  Ex.: a soma de dois inteiros não custa o mesmo que dividir dois reais mas, para uma visão global, esses valores específicos não são importantes

Tipos de análise de algoritmos

Pior caso (análise mais comum de ser feita):

- $T(n)$ = quantidade máxima de tempo para qualquer entrada de tamanho n

Caso médio (análise feita de vez em quando):

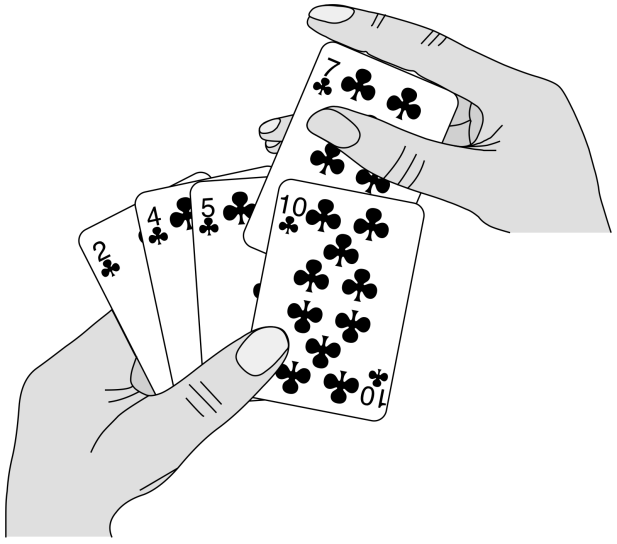
- $T(n)$ = tempo médio para qualquer entrada de tamanho n
- Implica em conhecimento sobre a distribuição estatística das entradas

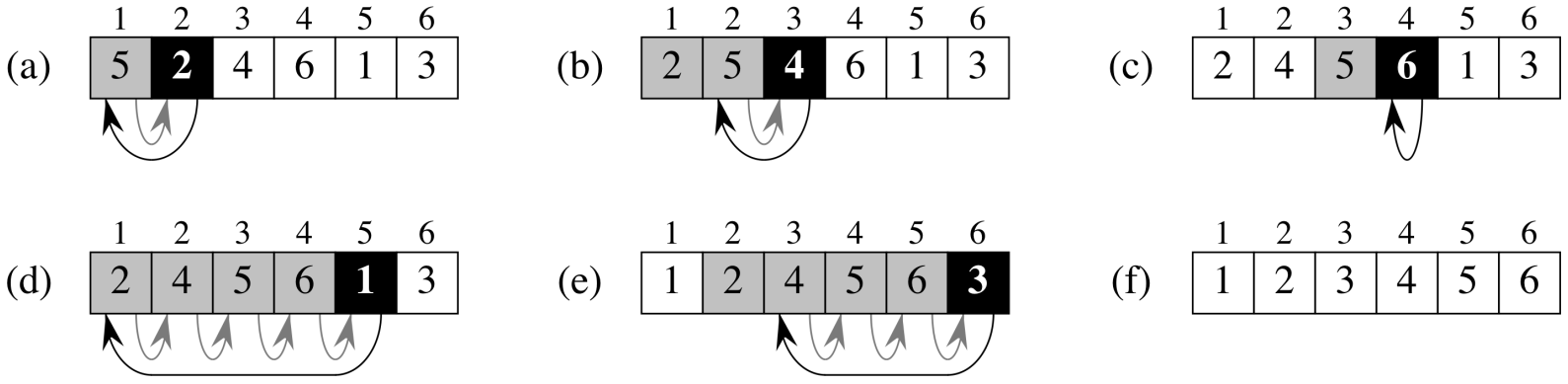
Melhor caso (apenas uma curiosidade):

- Quando o algoritmo é rápido apenas para algumas das entradas

Ordenação por inserção *(revisão)*

Ordenação por inserção





Vamos contar ?

ORDENA-POR-INSERÇÃO(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	$n-1$

A constante c_k representa o custo (tempo) de cada execução da linha k.

Denote por t_j o número de vezes que o teste no laço **enquanto** na linha 5 é feito para aquele valor de j.

Tempo de execução total

Logo, o tempo total de execução $T(n)$ de Ordena-Por-Inserção é a soma dos tempos de execução de cada uma das linhas do algoritmo, ou seja:

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

Como se vê, entradas de tamanho igual (i.e., mesmo valor de n), podem apresentar tempos de execução diferentes já que o valor de $T(n)$ depende dos valores dos t_j.

Pior Caso

Quando o vetor A está em ordem decrescente, ocorre o pior caso para Ordena-Por-Inserção. Para inserir a *chave* em $A[1 \ldots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$ para $j = 2, \ldots, n$.

Lembre-se que:

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

e

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

Pior caso – continuação

Temos então que

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

O tempo de execução no pior caso é da forma $an^2 + bn + c$ onde a, b, c são constantes que dependem apenas dos c_i.

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

Complexidade assintótica de algoritmos

- Como dito anteriormente, na maior parte desta disciplina, estaremos nos concentrando na **análise de pior caso** e no **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- O algoritmo Ordena-Por-Inserção tem como complexidade (de pior caso) uma função quadrática $an^2 + bn + c$, onde a, b, c são constantes absolutas que dependem apenas dos custos c_i.
- O estudo assintótico nos permite "jogar para debaixo do tapete" os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de a, b e c).
- **Por que podemos fazer isso ?**

Análise assintótica de funções quadráticas

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

n	$3n^2 + 10n + 50$	$3n^2$	Diferença percentual
64	12978	12288	5,32%
128	50482	49152	2,63%
512	791602	786432	0,65%
1024	3156018	3145728	0,33%
2048	12603442	12582912	0,16%
4096	50372658	50331648	0,08%
8192	201408562	201326592	0,04%
16384	805470258	805306368	0,02%
32768	3221553202	3221225472	0,01%

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

De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Notação assintótica

- Usando notação assintótica, dizemos que o algoritmo Ordena-Por-Inserção tem complexidade de tempo de pior caso $\Theta(n^2)$.
- Isto quer dizer duas coisas:
  - a complexidade de tempo é limitada (superiormente) assintoticamente por algum polinômio da forma an^2 para alguma constante a,
  - para todo n suficientemente grande, existe alguma instância de tamanho n que consome tempo pelo menos dn^2, para alguma contante positiva d.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

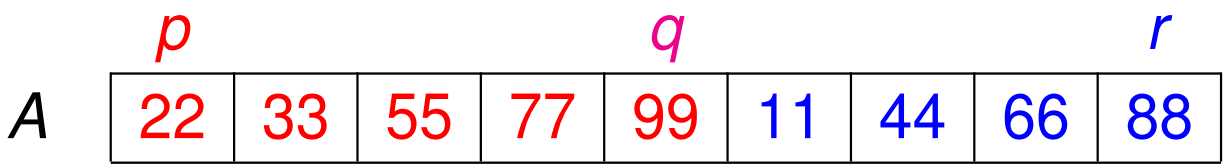
Ordenação por intercalação

Ordenação por intercalação

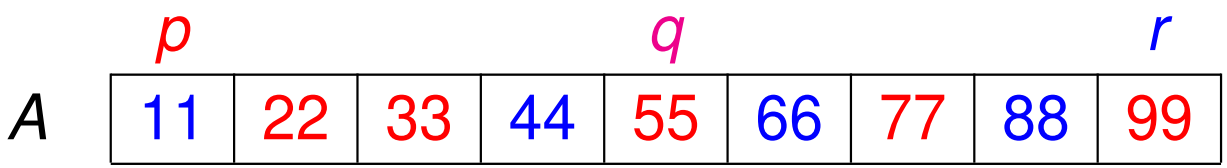
Q que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p \ldots q]$ e $A[q+1 \ldots r]$ crescentes, rearranjar $A[p \ldots r]$ de modo que ele fique em ordem crescente.

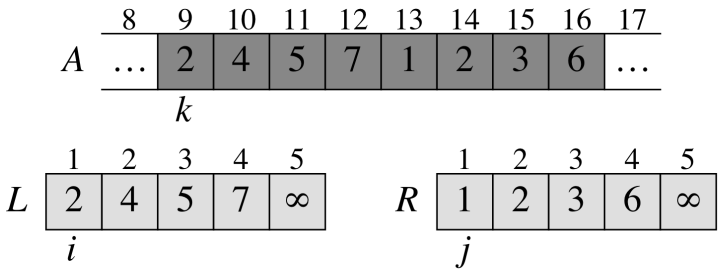
Entrada:



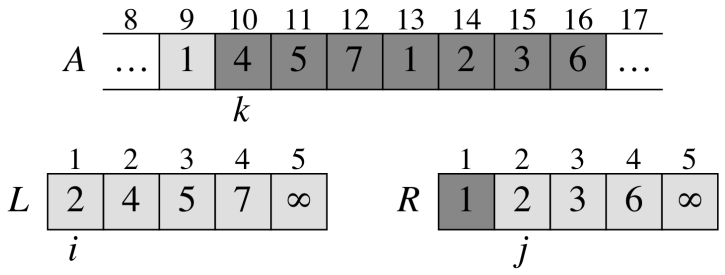
Saída:



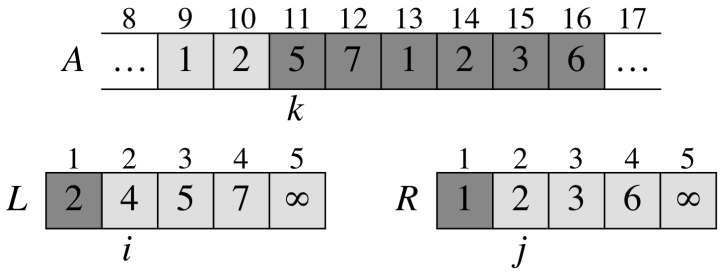
Intercalação com sentinela



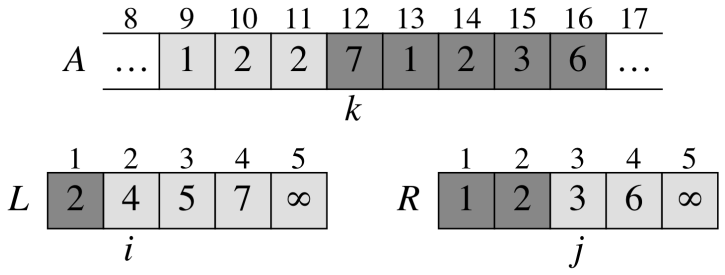
(a)



(b)

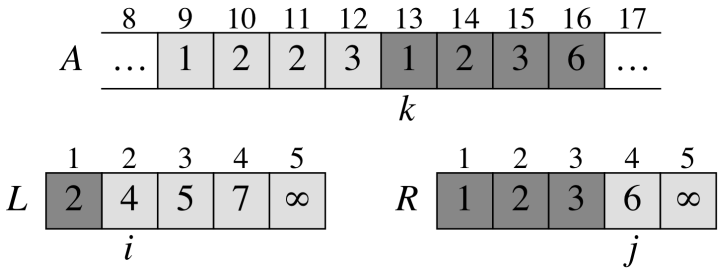


(c)

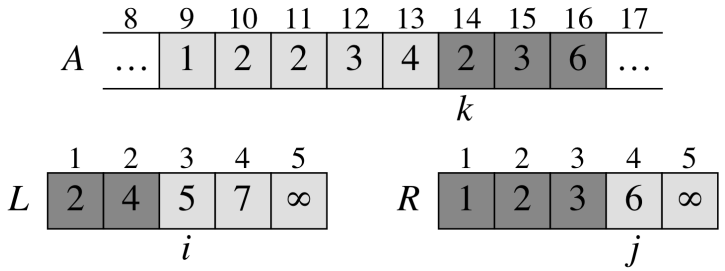


(d)

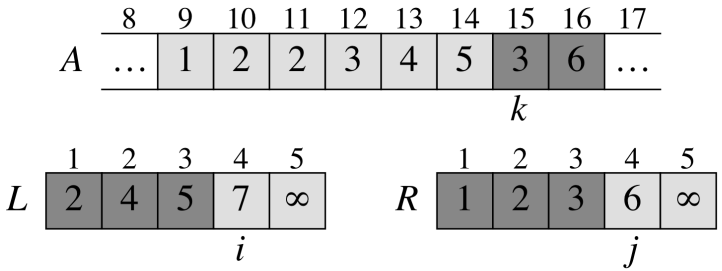
Intercalação com sentinela



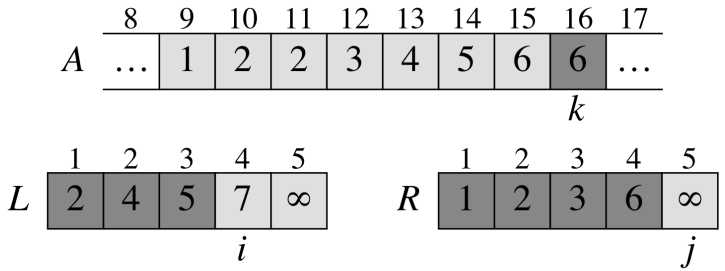
(e)



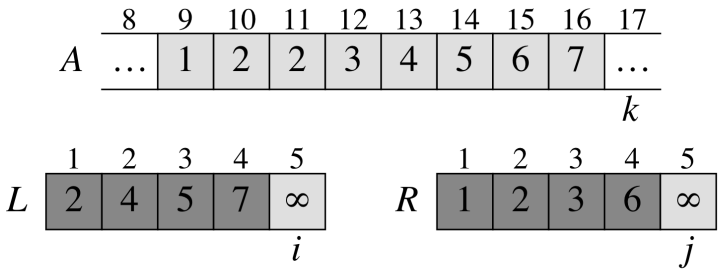
(f)



(g)



(h)



(i)

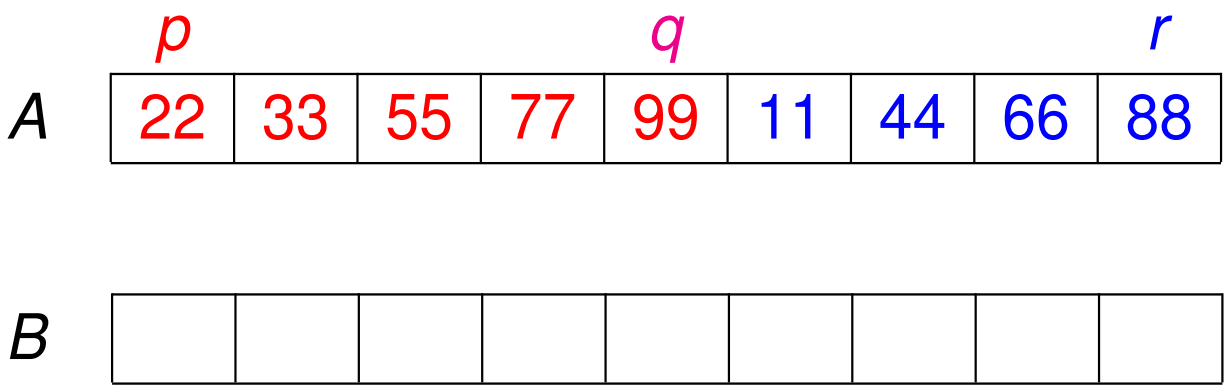
Intercalação com sentinela

INTERCALA(A, p, q, r)

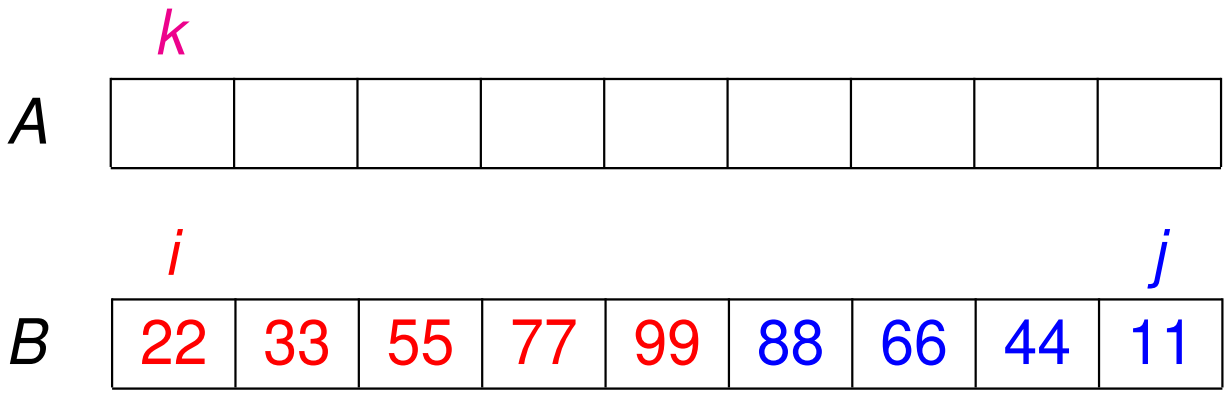
1: $n_1 \leftarrow q - p + 1$
2: $n_2 \leftarrow r - q$
3: sejam $L[1..n_1 + 1]$ e $R[1..n_2 + 1]$ novos vetores
4: **para** $i \leftarrow 1$ **até** n_1 **faça**
5: $\quad L[i] \leftarrow A[p + i - 1]$
6: **para** $j \leftarrow 1$ **até** n_2 **faça**
7: $\quad R[j] \leftarrow A[q + j]$
8: $L[n_1 + 1] \leftarrow \infty$
9: $R[n_2 + 1] \leftarrow \infty$
10: $i \leftarrow 1$
11: $j \leftarrow 1$
12: **para** $k \leftarrow p$ **até** r **faça**
13: $\quad$ **se** $L[i] \leq R[j]$ **então**
14: $\quad\quad A[k] \leftarrow L[i]$
15: $\quad\quad i \leftarrow i + 1$
16: $\quad$ **senão**
17: $\quad\quad A[k] = R[j]$
18: $\quad\quad j \leftarrow j + 1$

Outro algoritmo de intercalação (sem sentinela)

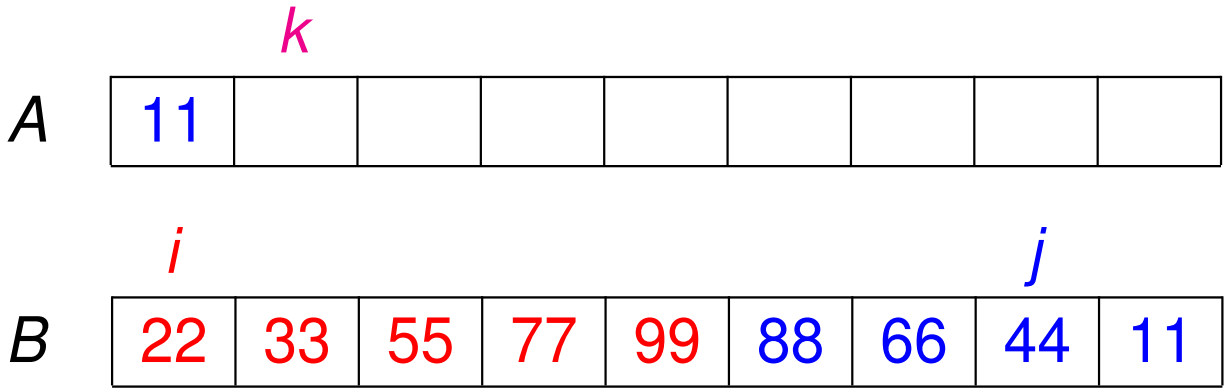
Intercalação



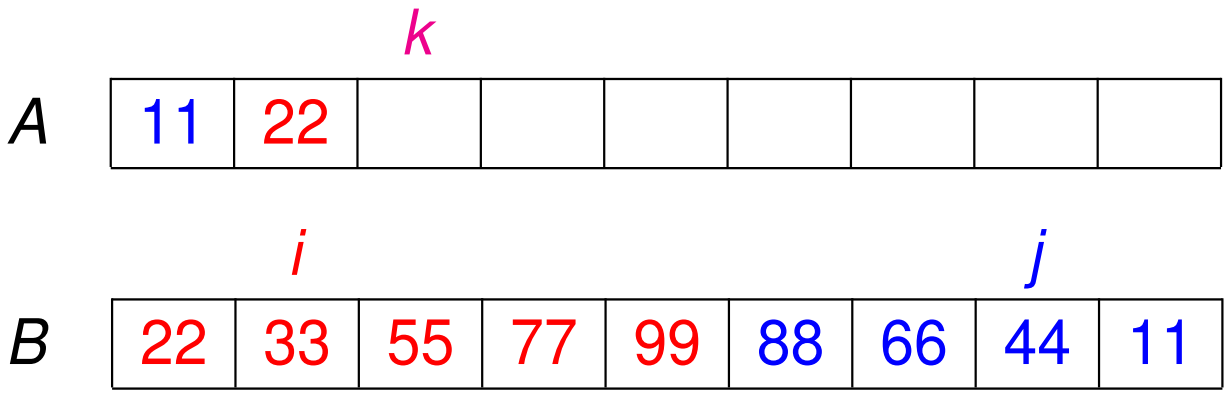
Intercalação



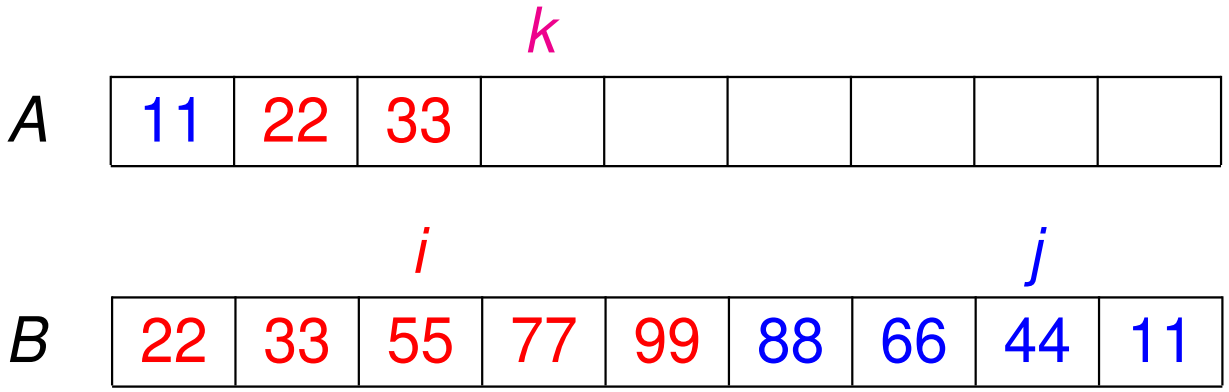
Intercalação



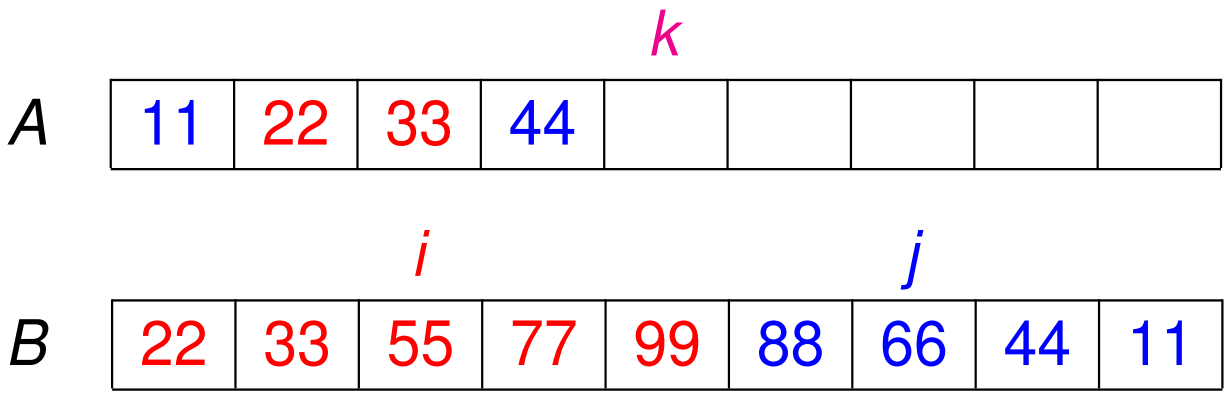
Intercalação



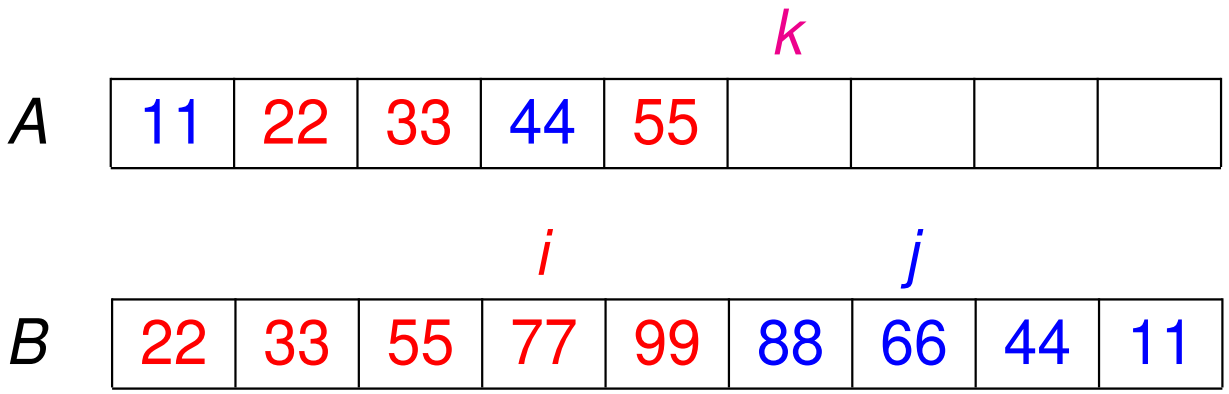
Intercalação



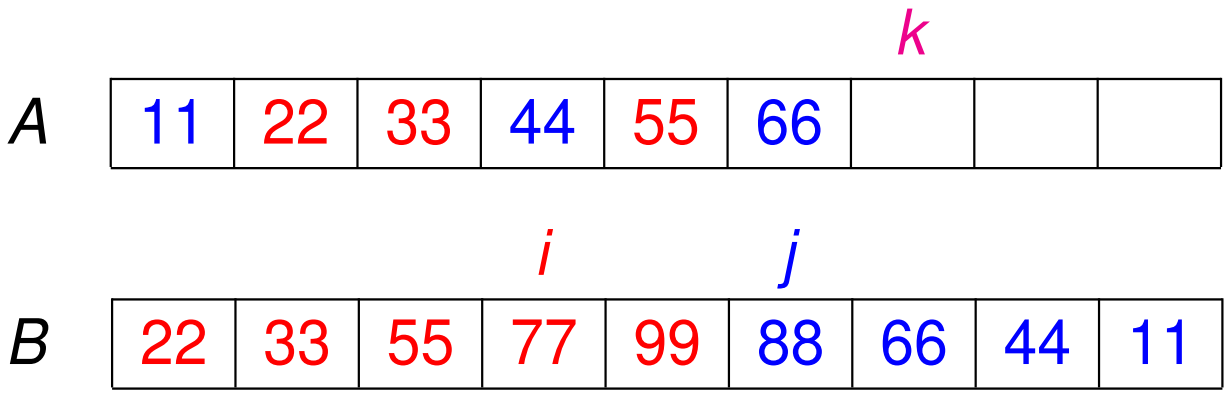
Intercalação



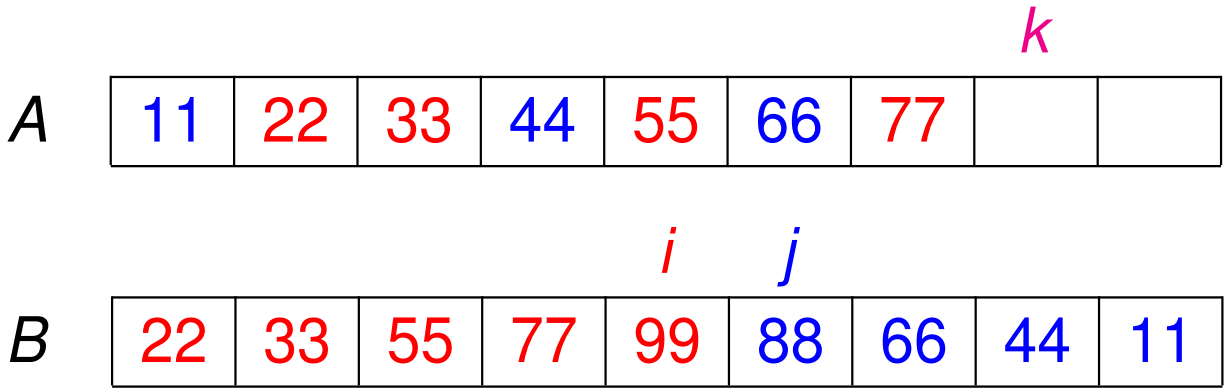
Intercalação



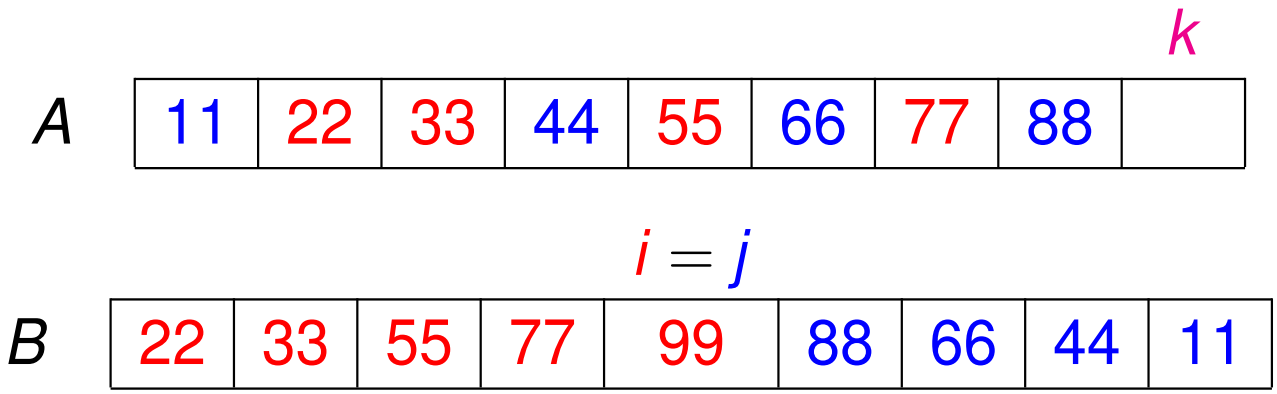
Intercalação



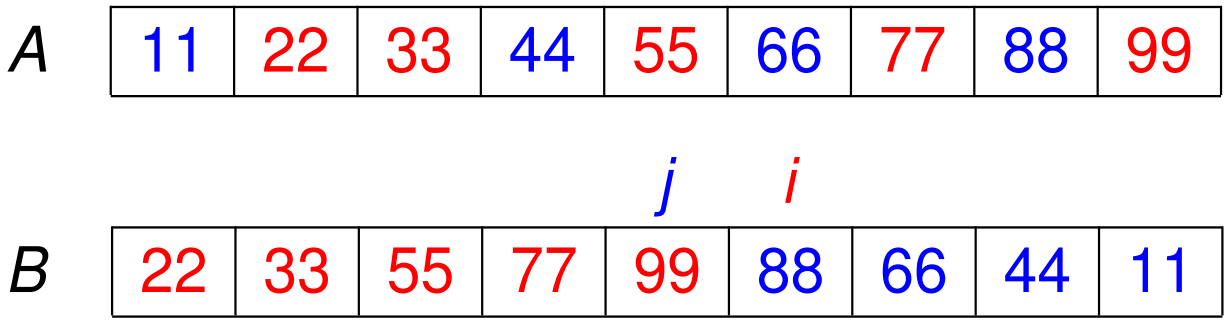
Intercalação



Intercalação



Intercalação

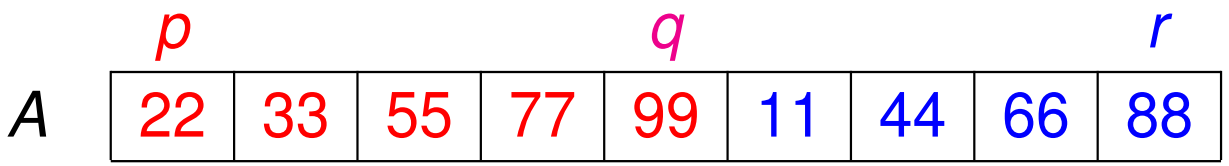


Pseudo-código

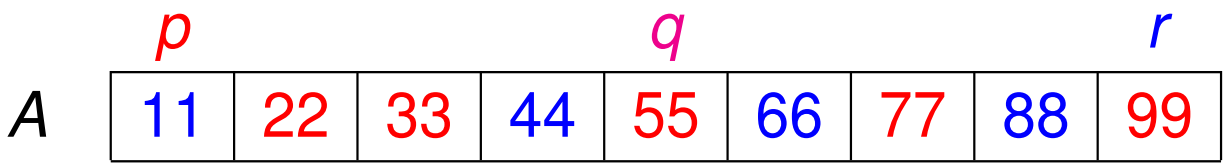
```
INTERCALA(A, p, q, r)
 1   para i ← p até q faça
 2       B[i] ← A[i]
 3   para j ← q + 1 até r faça
 4       B[r + q + 1 − j] ← A[j]
 5   i ← p
 6   j ← r
 7   para k ← p até r faça
 8       se B[i] ≤ B[j]
 9           então A[k] ← B[i]
10                 i ← i + 1
11           senão A[k] ← B[j]
12                 j ← j − 1
```

Complexidade de Intercala

Entrada:



Saída:



Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

Corretude de Intercala

Invariante principal de Intercala:

No começo de cada iteração do laço das linhas 7–12, vale que:

1. $A[p \ldots k-1]$ está ordenado,
2. $A[p \ldots k-1]$ contém todos os elementos de $B[p \ldots i-1]$ e de $B[j+1 \ldots r]$,
3. $B[i] \geq A[k-1]$ e $B[j] \geq A[k-1]$.

Exercício. Prove que a afirmação acima é de fato um invariante de INTERCALA.

Exercício. (fácil) Mostre usando o invariante acima que INTERCALA é correto.

Projeto por indução e algoritmos recursivos

> “To understand recursion, we must first understand recursion.”
>
> *(anônimo)*

- Um algoritmo recursivo obtém a saída para uma instância de de um problema chamando a si mesmo para resolver instâncias menores deste mesmo problema (trata-se de um projeto por indução).
- A resolução por projeto de indução, deve reduzir um problema a subproblemas menores do mesmo tipo. E problemas suficientemente pequenos devem ser resolvidos de maneira direta.

Algoritmos recursivos

- O que é o paradigma de **divisão-e-conquista**?
- Como mostrar a corretude de um algoritmo recursivo?
- Como analisar o consumo de tempo de um algoritmo recursivo?
- O que é uma **fórmula de recorrência**?
- O que significa *resolver* uma fórmula de recorrência?

Recursão e o paradigma de divisão-e-conquista

- Algoritmos de divisão-e-conquista possuem as seguintes etapas em cada nível de recursão:
  1. **Problemas pequenos:** Quando os problemas são suficientemente pequenos, então o algoritmo recursivo deve resolver o problema de maneira direta.
  2. **Problemas que não são pequenos:**
     1. **Divisão:** o problema é dividido em subproblemas semelhantes ao problema original, porém tendo como entrada instâncias de tamanho menor.
     2. **Conquista:** cada subproblema é resolvido recursivamente a menos que o tamanho de sua entrada seja suficientemente "pequeno", quando este é resolvido diretamente.
     3. **Combinação:** as soluções dos subproblemas são combinadas para obter uma solução do problema original.

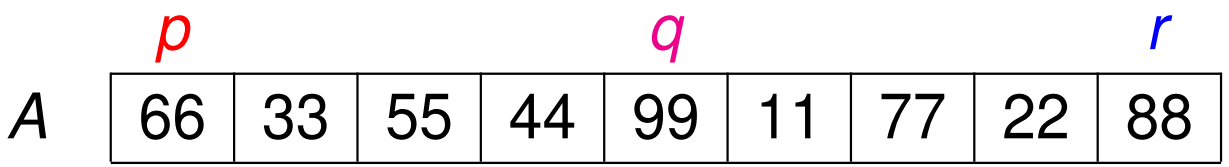
Exemplo de divisão-e-conquista: *Mergesort*

- Mergesort é um algoritmo para resolver o problema de ordenação e um exemplo clássico do uso do paradigma de divisão-e-conquista. (*to merge* = intercalar)
- Descrição do Mergesort em alto nível:
  1. **Divisão**: divida o vetor com n elementos em dois subvetores de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, respectivamente.
  2. **Conquista**: ordene os dois vetores recursivamente usando o Mergesort;
  3. **Combinação**: intercale os dois subvetores para obter um vetor ordenado usando o algoritmo Intercala.

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

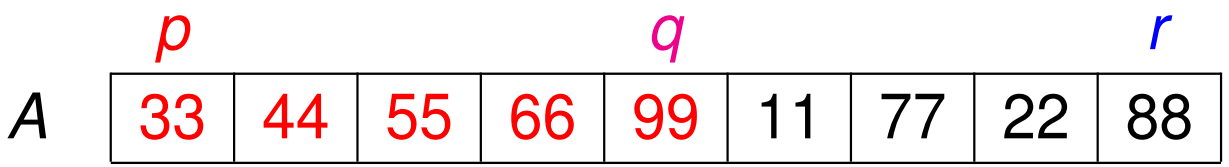
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

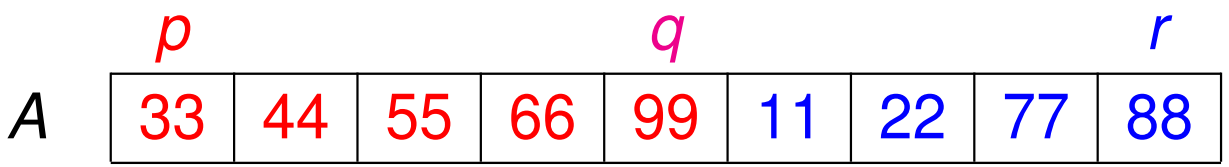
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

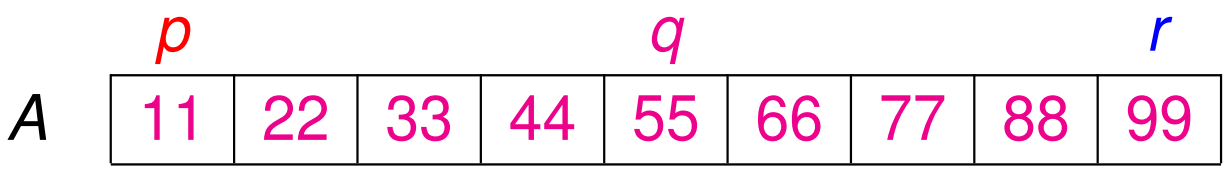
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

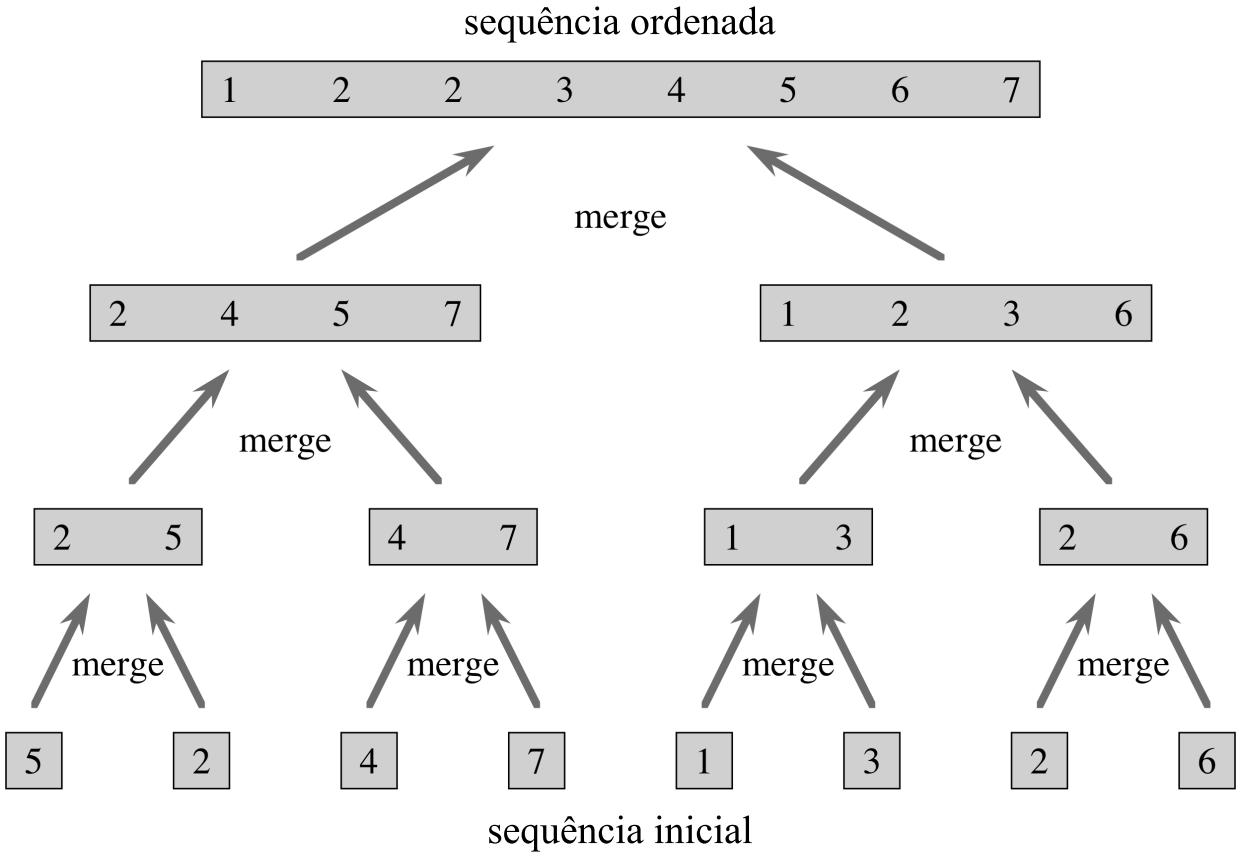
Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort – exemplo do livro

- Exemplo do livro (CLRS)
- Visualização de cada “merge” do algoritmo



Corretude do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

O algoritmo está correto?

A corretude do algoritmo Mergesort apoia-se na corretude do algoritmo Intercala e pode ser demonstrada **por indução** em $n := r - p + 1$.

Aprenderemos como fazer provas por indução mais adiante.

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

Qual é a complexidade de MERGESORT?

Seja $T(n)$:= o consumo de tempo máximo (pior caso) em função de $n = r - p + 1$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	?
2	?
3	?
4	?
5	?

$T(n) = ?$

Complexidade do Mergesort

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

linha	consumo de tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$$

Complexidade do Mergesort

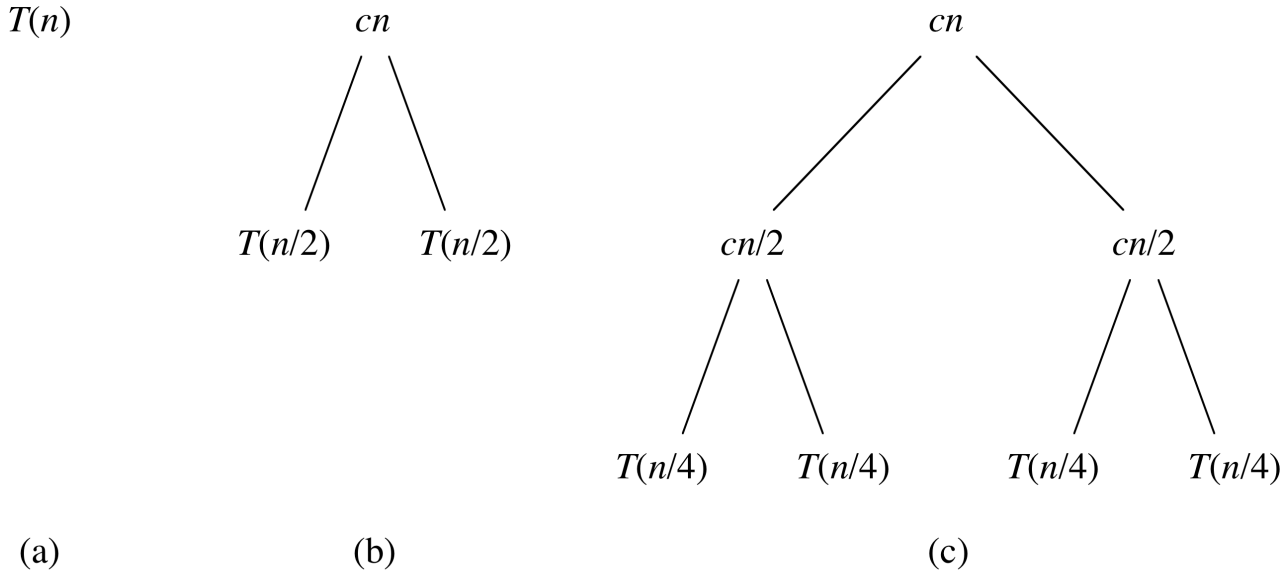
- Obtemos o que chamamos de fórmula de recorrência (i.e., uma fórmula definida em termos de si mesma).

$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

- Em geral, ao aplicar o paradigma de divisão-e-conquista, chega-se a um algoritmo recursivo cuja complexidade $T(n)$ é uma fórmula de recorrência.
- É necessário então resolver a recorrência! Mas, o que significa resolver uma recorrência?
- Significa encontrar uma "fórmula fechada" para $T(n)$.
- No caso, $T(n) = \Theta(n \lg n)$. Assim, o consumo de tempo do Mergesort é $\Theta(n \lg n)$ no pior caso.
- Veremos mais tarde como resolver recorrências.

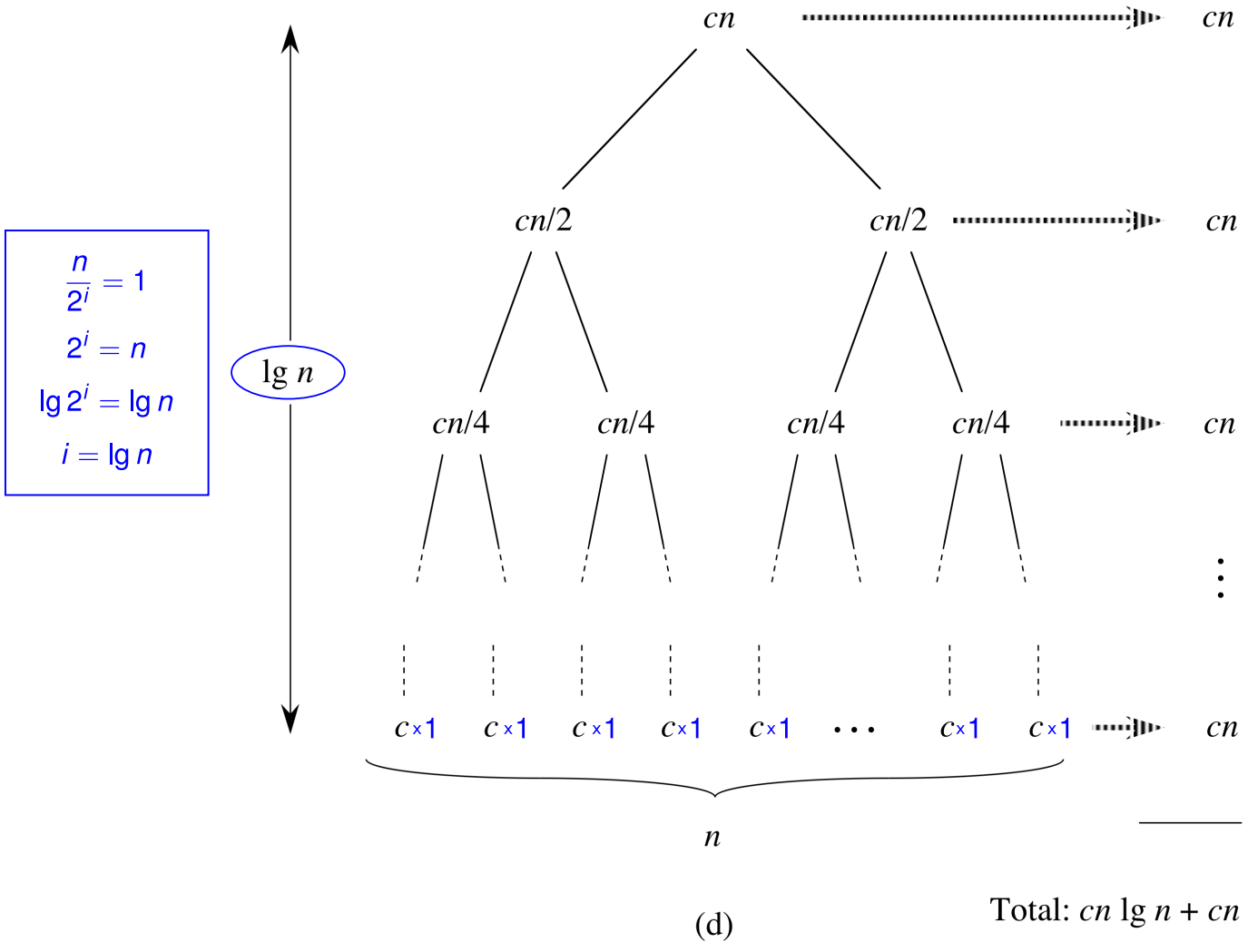
Mergesort – árvore de recursão

- Árvore de recursão do Mergesort



Mergesort – árvore de recursão

- Árvore de recursão do Mergesort



(d)

Crescimento de funções

Notação Assintótica

- Vamos expressar complexidade através de funções em variáveis que descrevam o tamanho de instâncias do problema. Exemplos:
  - Problemas de aritmética de precisão arbitrária: número de bits (ou bytes) dos inteiros.
  - Problemas em grafos: número de vértices e/ou arestas
  - Problemas de ordenação de vetores: tamanho do vetor.
  - Busca em textos: número de caracteres do texto ou padrão de busca.
- Vamos supor que funções que expressam complexidade são sempre positivas, já que estamos medindo número de operações.

Comparação de Funções

- Vamos comparar funções assintoticamente, ou seja, para valores grandes, desprezando constantes multiplicativas e termos de menor ordem.

	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

Análise assintótica

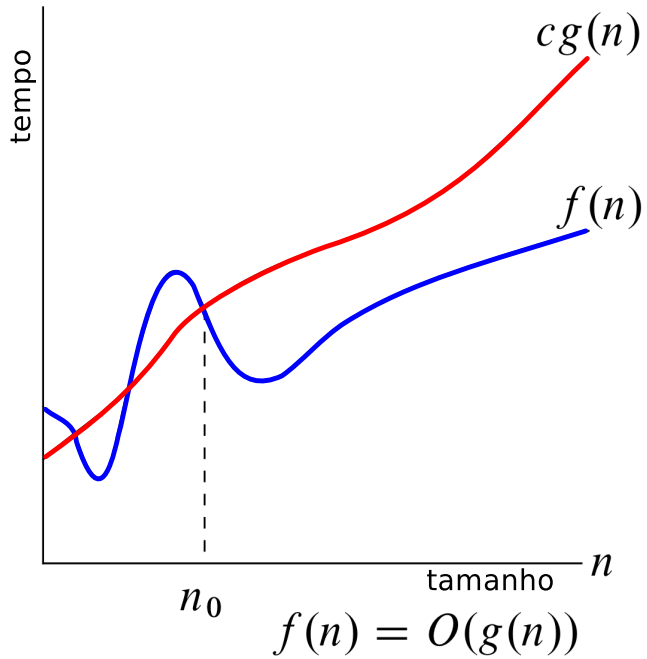
- Precisamos de uma ferramenta matemática para **comparar funções**
- Para a análise de algoritmo será feita uma **análise assintótica**:
  - Matematicamente: estudando o comportamento de **limites** ($n \to \infty$)
  - Computacionalmente: estudando o comportamento para entrada arbitrariamente grande ou descrevendo **taxa de crescimento**
- Para isso, uma **notação** específica é usada: O, Ω, Θ, o, ω
- O foco está nas **ordens de crescimento**

Classe O

Definição:

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

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.



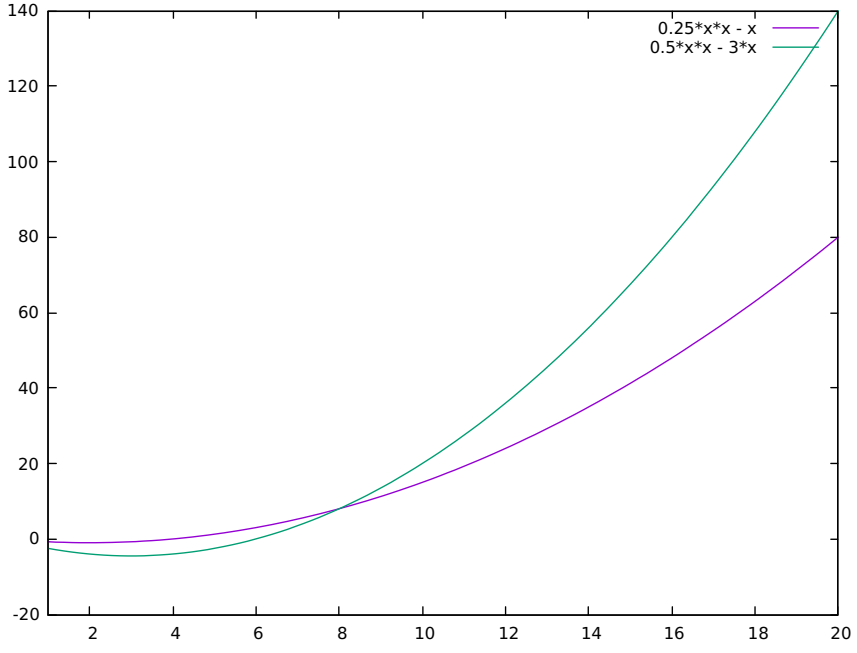
Classe O

Exemplo

$$f(n) = \frac{1}{4}n^2 - n$$
$$g(n) = n^2 - 6n$$

Valores de c e n_0 que satisfazem $f(n) \in O(g(n))$:

$$c = \frac{1}{2} \quad \text{e} \quad n_0 = 8$$

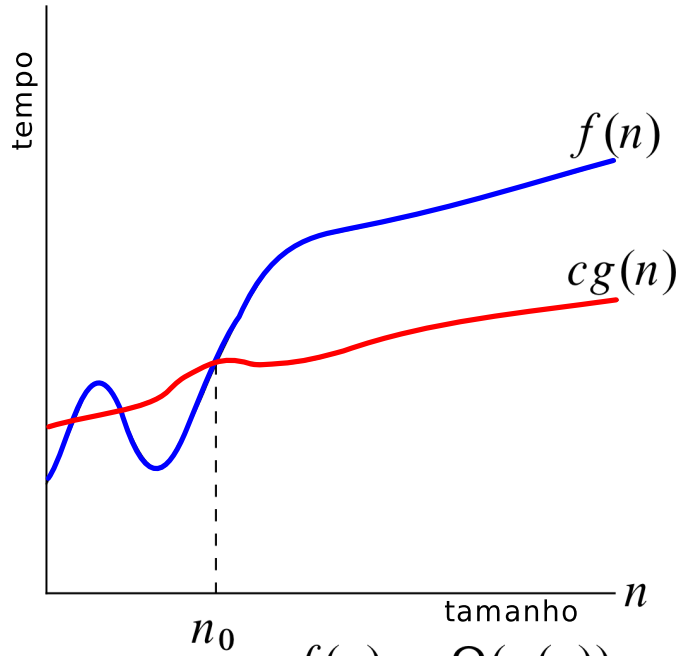


Classe Ω

Definição:

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

Informalmente, dizemos que, se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão lentamente quanto $g(n)$.



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

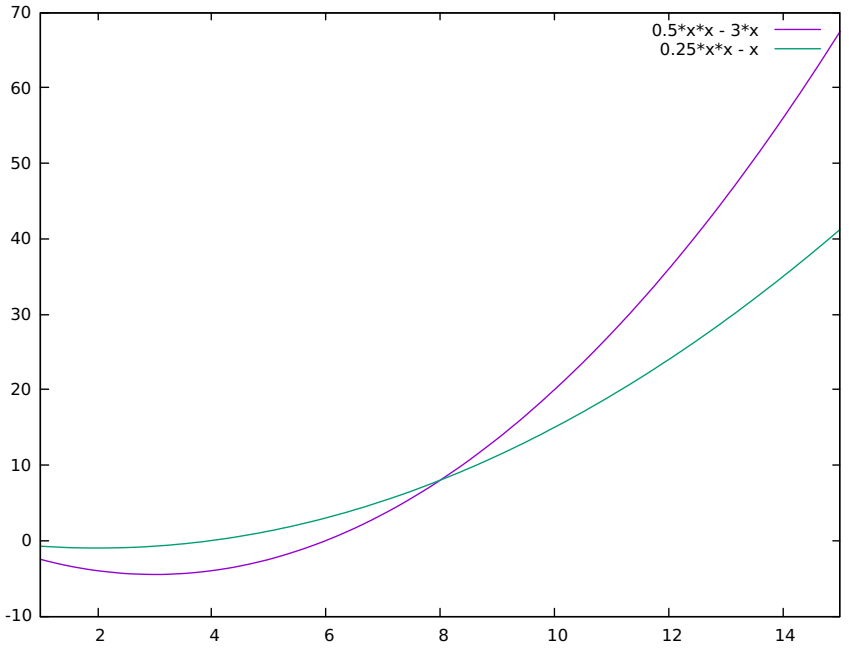
Classe Ω

Exemplo

$$f(n) = \frac{1}{2}n^2 - 3n$$
$$g(n) = \frac{1}{2}n^2 - 2n$$

Valores de c e n_0 que satisfazem $f(n) \in \Omega(g(n))$:

$$c = \frac{1}{2} \quad \text{e} \quad n_0 = 8$$

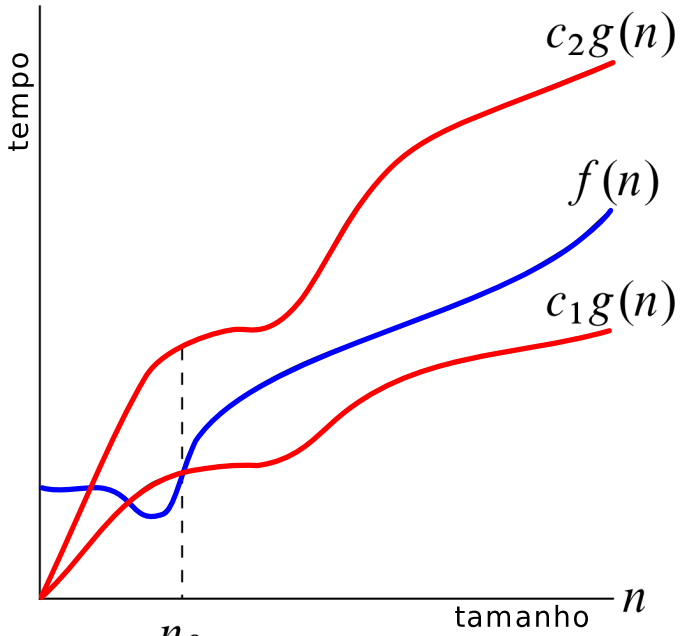


Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ 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 $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.



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

Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ 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 $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Theta(n^2)$

Valores de c_1, c_2 e n_0 que satisfazem a definição são

$$c_1 = \frac{1}{14},\ c_2 = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe o

Definição:

$o(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in o(g(n))$, então $f(n)$ cresce mais lentamente que $g(n)$.

Exemplo:

$1000n^2 \in o(n^3)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \left\lceil \frac{1000}{c} \right\rceil + 1.$$

Classe ω

Definição:

$\omega(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$, para todo $n \geq n_0$.$\}$

Informalmente, dizemos que, se $f(n) \in \omega(g(n))$, então $f(n)$ cresce mais rapidamente que $g(n)$.

Exemplo:

$\frac{1}{1000}n^2 \in \omega(n)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \lceil 1000c \rceil + 1.$$

Notação assintótica – resumo

- $f(n) \in O(g(n))$ se houver constantes positivas n_0 e c tal que $f(n) \leq c\, g(n)$ para todo $n \geq n_0$
- $f(n) \in \Omega(g(n))$ se houver constantes positivas n_0 e c tal que $f(n) \geq c\, g(n)$ para todo $n \geq n_0$
- $f(n) \in \Theta(g(n))$ se houver constantes positivas n_0, c_1 e c_2 tal que $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ para todo $n \geq n_0$
- $f(n) \in o(g(n))$ se, para qualquer constante positiva c, existe n_0 tal que $f(n) < c\, g(n)$ para todo $n \geq n_0$
- $f(n) \in \omega(g(n))$ se, para qualquer constante positiva c, existe n_0 tal que $f(n) > c\, g(n)$ para todo $n \geq n_0$

Notação assintótica – analogia

Analogia entre duas funções f e g e dois números a e b:

- $f(n) \in O(g(n)) \quad \approx \quad a \leq b$
- $f(n) \in \Omega(g(n)) \quad \approx \quad a \geq b$
- $f(n) \in \Theta(g(n)) \quad \approx \quad a = b$
- $f(n) \in o(g(n)) \quad \approx \quad a < b$
- $f(n) \in \omega(g(n)) \quad \approx \quad a > b$

Definições equivalentes

$f(n) \in o(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$

$f(n) \in O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$

$f(n) \in \Theta(g(n))$ se $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$

$f(n) \in \Omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0.$

$f(n) \in \omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$

Propriedades das Classes

Transitividade:

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$.

Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$.

Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$.

Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$.

Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$.

Propriedades das Classes

Reflexividade:

$f(n) \in O(f(n))$.

$f(n) \in \Omega(f(n))$.

$f(n) \in \Theta(f(n))$.

Simetria:

$f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$.

Simetria Transposta:

$f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$.

$f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$.

Notação assintótica – algumas regras práticas

- **Multiplicação por uma constante:**
  $\Theta(c\, f(n)) = \Theta(f(n))$
  $99\, n^2 = \Theta(n^2)$

- **Mais alto expoente** de um polinômio
  $a_x n^x + a_{x-1} n^{x-1} + \cdots + a_2 n^2 + a_1 n + a_0$:
  $3\mathbf{n^3} - 5n^2 + 100 = \Theta(n^3)$
  $6\mathbf{n^4} - 20n^2 = \Theta(n^4)$
  $0.8\mathbf{n} + 224 = \Theta(n)$

- **Termo dominante:**
  $\mathbf{2^n} + 6n^3 = \Theta(2^n)$
  $\mathbf{n!} - 3n^2 = \Theta(n!)$
  $n \log n + 3\mathbf{n^2} = \Theta(n^2)$

Notação assintótica – dominância

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

- Se queremos reduzir o tempo, funções “menores” são melhores
- Uma função domina sobre outra se, a medida que n cresce, a função continua “maior”
- Matematicamente: $f(n) \gg g(n)$ se $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Relações de dominância

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

Notação assintótica – visão prática

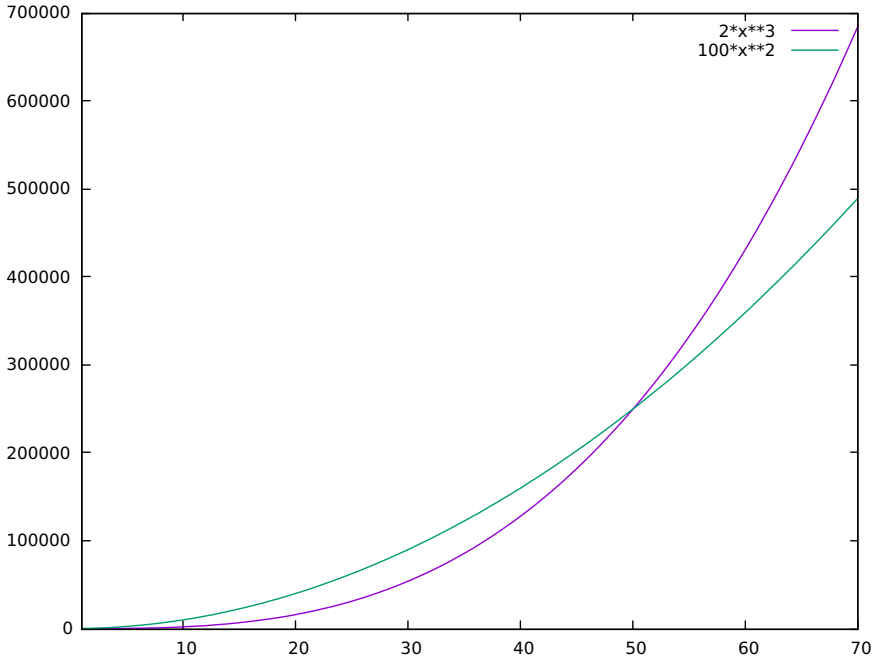
Se uma operação leva 10^{-9} segundos

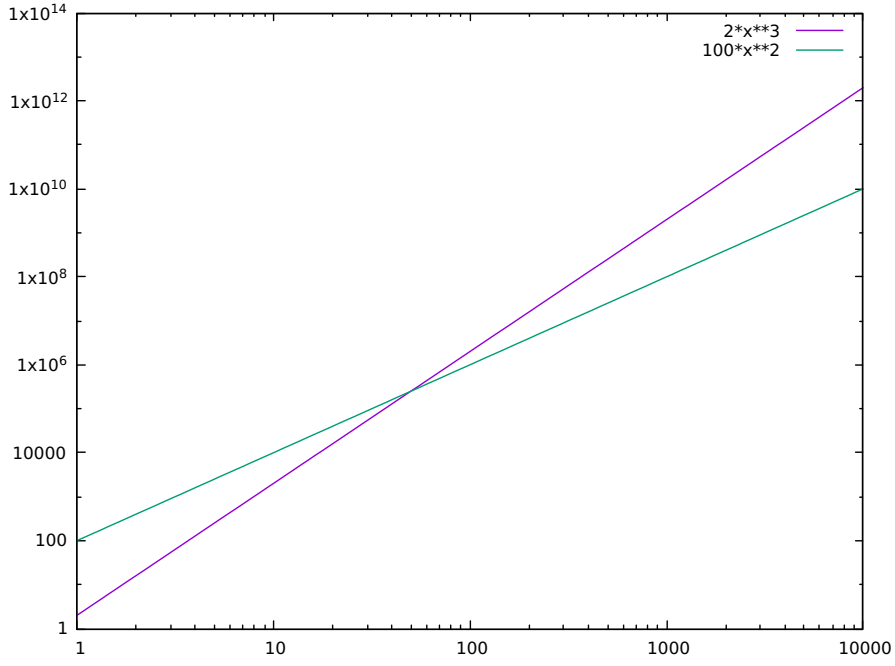
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 anos
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3 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			

Desenhando funções

- Comparando $2n^3$ com $100n^2$ usando o gnuplot:

```
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
```

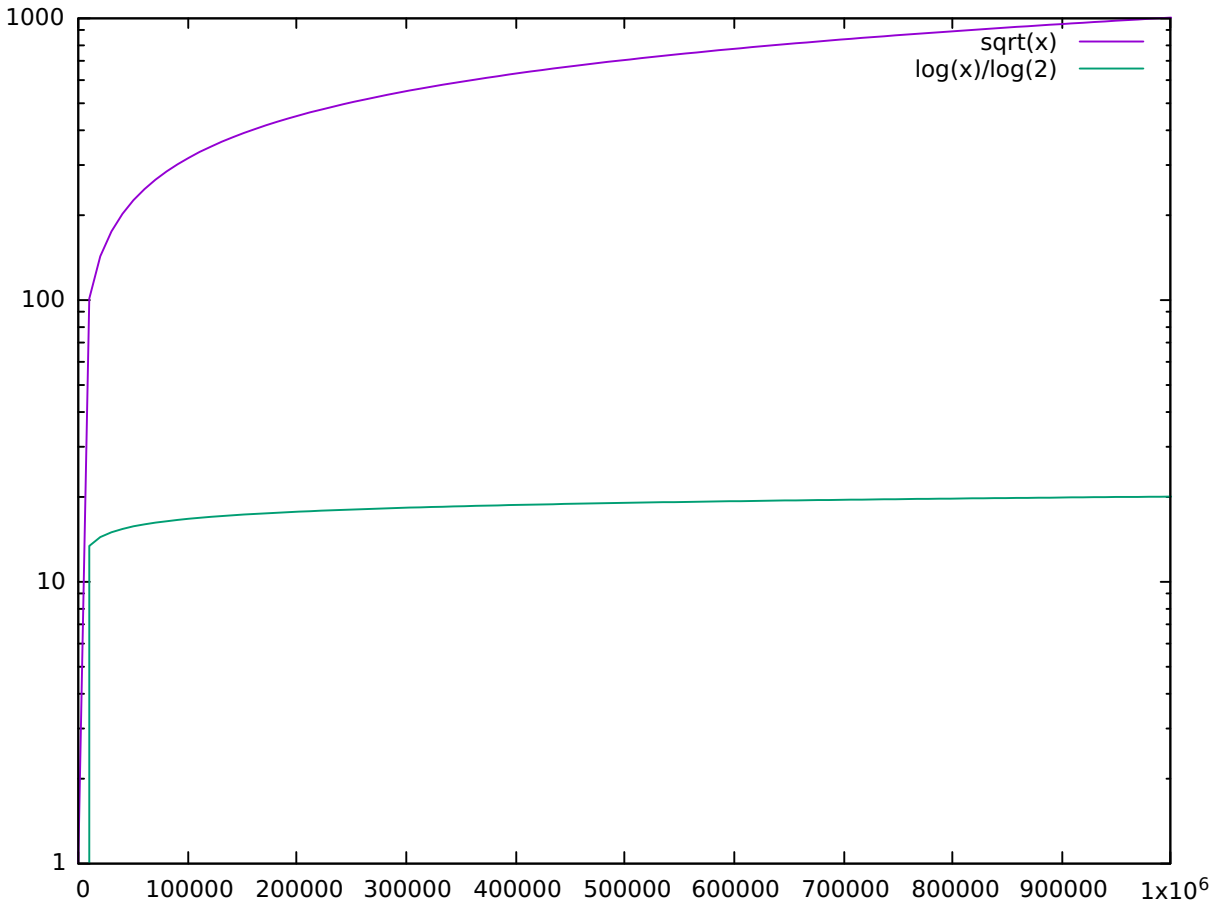




Desenhando funções

- Comparando $\sqrt{n}$ e $\log_2 n$:

```
gnuplot> set logscale y 10
gnuplot> plot [1:1000000] sqrt(x), log(x)/log(2)
```



Indução matemática

Demonstração por Indução

Na *Demonstração por Indução*, queremos demonstrar a validade de $P(n)$, uma propriedade P com um parâmetro natural n associado, para todo valor de n.
Há um número infinito de casos a serem considerados, um para cada valor de n. Demonstramos os infinitos casos de uma só vez:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que a soma dos n primeiros naturais ímpares é n^2.

Demonstração por Indução

Outra forma equivalente:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução:** Supomos que $P(n-1)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que a soma dos n primeiros naturais ímpares é n^2.

Demonstração por Indução

Às vezes queremos provar que uma proposição $P(n)$ vale para $n \geq n_0$ para algum n_0.

- **Base da Indução:** Demonstramos $P(n_0)$.
- **Hipótese de Indução:** Supomos que $P(n-1)$ é verdadeiro.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro $n \geq 2$ pode ser fatorado como um produto de primos.

Indução Fraca $\times$ Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.
No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior, ou seja:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução Forte:** Supomos que $P(k)$ é verdadeiro, para todo $1 \leq k < n$.
- **Passo de Indução:** Provamos que $P(n)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:

Prove que todo inteiro $n \geq 2$ pode ser fatorado como um produto de primos.

Exemplo 1

Demonstre que a inequação

$$(1 + x)^n \geq 1 + nx$$

vale para todo natural n e real x tal que $(1 + x) > 0$.

Demonstração:

- A **base da indução** é $n = 1$. Nesse caso ambos os lados da inequação são iguais a $1 + x$, mostrando a sua validade. Isto encerra a prova do caso base.

Exemplo 1 (cont.)

- A **hipótese de indução** é: *Suponha que a inequação vale para n, isto é,* $(1+x)^n \geq 1+nx$ *para todo real x tal que* $(1+x) > 0$.
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que a inequação vale para o valor* $n+1$*, isto é,* $(1+x)^{n+1} \geq 1+(n+1)x$ *para todo x tal que* $(1+x) > 0$. A dedução é simples:

$$\begin{aligned}
(1+x)^{n+1} &= (1+x)^n(1+x) \\
&\geq (1+nx)(1+x) \text{ (pela h.i. e } (1+x) > 0) \\
&= 1+(n+1)x+nx^2 \\
&\geq 1+(n+1)x \text{ (já que } nx^2 \geq 0)
\end{aligned}$$

A última linha mostra que a inequação vale para $n+1$, completando a demonstração. ■

Exemplo 2

Demonstre que o número T_n de regiões no plano criadas por n retas em posição geral é igual a

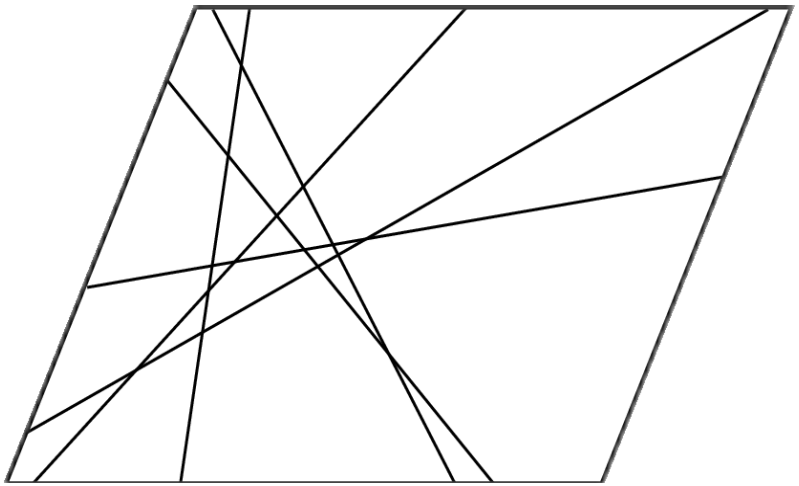
$$T_n = \frac{n(n+1)}{2} + 1.$$

Um conjunto de retas está em posição geral no plano se

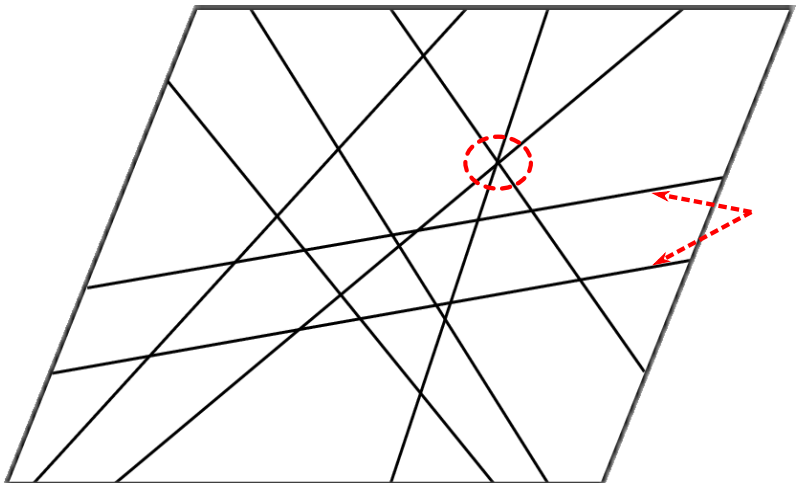
- todas as retas são concorrentes, isto é, não há retas paralelas e
- não há três retas interceptando-se no mesmo ponto.

Exemplo 2 (cont.)

Antes de prosseguirmos com a demonstração vejamos exemplos de um conjunto de retas que está em posição geral e outro que não está.



Em posição geral



Não estão em posição geral

Exemplo 2 (cont.)

Demonstração: A idéia que queremos explorar para o passo de indução é a seguinte: supondo que a fórmula vale para n, adicionar uma nova reta em posição geral e tentar assim obter a validade de $n+1$.

- A **base da indução** é, naturalmente, $n = 1$. Uma reta sozinha divide o plano em duas regiões. De fato,

$$T_1 = (1 \times 2)/2 + 1 = 2.$$

Isto conclui a prova para $n = 1$.

Exemplo 2 (cont.)

- A **hipótese de indução** é: *Suponha que* $T_n = (n(n+1)/2) + 1$ *para* n.
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que para* $n+1$ *retas em posição geral vale que*

$$T_{n+1} = \frac{(n+1)(n+2)}{2} + 1.$$

Considere um conjunto L de $n+1$ retas em posição geral no plano e seja r uma dessas retas. Então, as retas do conjunto $L' = L \setminus \{r\}$ obedecem à hipótese de indução e, portanto, o número de regiões distintas do plano definidas por elas é $(n(n+1))/2 + 1$.

Exemplo 2 (cont.)

- Além disso, r intersecta as outras n retas em n pontos distintos. O que significa que, saindo de uma ponta de r no infinito e após cruzar as n retas de L', a reta r terá cruzado $n+1$ regiões, dividindo cada uma destas em duas outras.
- Assim, podemos escrever que

$$\begin{aligned} T_{n+1} &= T_n + n + 1 \\ &= \frac{n(n+1)}{2} + 1 + n + 1 \text{ (pela h.i.)} \\ &= \frac{(n+1)(n+2)}{2} + 1. \end{aligned}$$

Isso conclui a demonstração. ■

Demonstração por Indução

Exemplos: Apesar da reconhecida validade dos seguintes somatórios, efetue provas por indução matemática da

1. Soma dos n termos de uma progressão aritmética (PA):

$$a_1 + (a_1 + r) + (a_1 + 2r) + \cdots + [a_1 + (n-1)r] =$$

$$= \sum_{i=0}^{n-1} (a_1 + i \cdot r) = \frac{n(a_1 + [a_1 + (n-1)r])}{2}$$

2. Soma dos n termos de uma progressão geométrica (PG):

$$a_1 + (a_1 \cdot q) + (a_1 \cdot q^2) + \cdots + (a_1 \cdot q^{n-1}) =$$

$$= \sum_{i=0}^{n-1} (a_1 \cdot q^i) = \frac{a_1(q^n - 1)}{q - 1}$$