Estruturas de Dados

A. G. Silva

Revisado em 19 de setembro de 2018

Objetivos

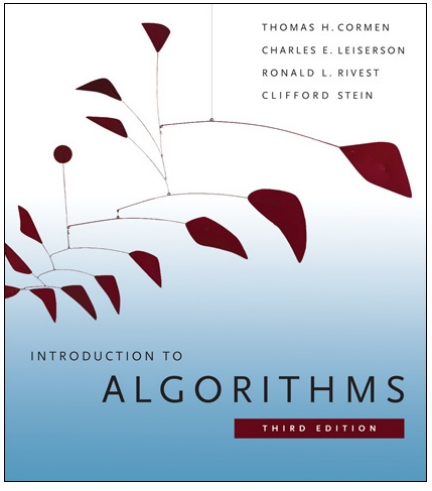
- **Geral:** Compreender o processo de análise de complexidade de algoritmos e conhecer as principais técnicas para o desenvolvimento de algoritmos, aplicações e análises de complexidade.

- **Específicos:**
  - Compreender o processo de análise de complexidade de algoritmos
  - Conhecer as principais técnicas para o desenvolvimento de algoritmos e suas análises
  - Compreender a diferença entre complexidade de problemas e complexidade de soluções
  - Conhecer e compreender as classes de complexidade de problemas
  - Conhecer algoritmos para tratar problemas complexos

Bibliografia

Básica:

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein , Introduction to Algorithms. 3^{rd} edition, The MIT Press, 2009.



- S. Dasgupta, C.H. Papadimitriou, U.V. Vazirani, Algorithms, 1^{st} edition, McGraw-Hill, 2006.
- Jon Kleinberg, Éva Tardos, Algorithm Design, 1^{st} edition, Pearson, 2005.

Bibliografia (cont...)

Complementar:

- N.C. Ziviani, Projeto de Algoritmos com Implementações em Java e C++, Thompson Learning, 2007.
- H.R. Lewis, C.H. Papadimitriou, Elementos de Teoria da Computação, 2 a Edição, Bookman, 2000.
- T.A. Sudkamp, Languages and Machines, Addison-Wesley, 1988.
- *Artigos selecionados*

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

Introdução

O que veremos nesta disciplina?

- Como provar a "corretude" de um algoritmo
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo $=$ análise de complexidade
- Técnicas e idéias gerais de projeto de algoritmos: indução, divisão-e-conquista, programação dinâmica, algoritmos gulosos etc
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínseca de vários problemas: inexistência de soluções eficientes

Algoritmos

O que é um algoritmo (computacional) ?

Informalmente, um algoritmo é um procedimento computacional bem definido que:

- recebe um conjunto de valores como entrada,
- produz um conjunto de valores como saída,
- através de uma sequência de passos em um modelo computacional.

Equivalentemente, um algoritmo é uma ferramenta para resolver um problema computacional. Este problema define a relação precisa que deve existir entre a entrada e a saída do algoritmo.

Exemplos de problemas: teste de primalidade

Problema: determinar se um dado número é primo.

Exemplo:

Entrada: 9411461

Saída: É primo.

Exemplo:

Entrada: 8411461

Saída: Não é primo.

Exemplos de problemas: ordenação

Definição: um vetor $A[1 \ldots n]$ é crescente se $A[1] \leq \ldots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \ldots n]$ de modo que fique crescente.

Entrada:

1										n
33	55	33	44	33	22	11	99	22	55	77

Saída:

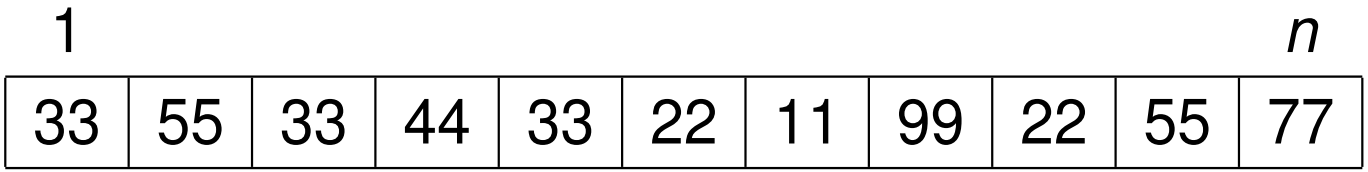
1										n
11	22	22	33	33	33	44	55	55	77	99

Instância de um problema

Uma instância de um problema é um conjunto de valores que serve de entrada para esse.

Exemplo:
Os números 9411461 e 8411461 são instâncias do problema de primalidade.

Exemplo:
O vetor



é uma instância do problema de ordenação.

A importância dos algoritmos para a computação

- Exemplos de aplicações para o uso/desenvolvimento de algoritmos “eficientes”?
  - projetos de genoma de seres vivos
  - rede mundial de computadores
  - comércio eletrônico
  - planejamento da produção de indústrias
  - logística de distribuição
  - computação científica, imagens médicas
  - *games* e filmes, ...

Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhece** algoritmos eficientes capazes de resolvê-los. Exemplos são os problemas $\mathcal{NP}$-completos.

  Curiosamente, **não foi provado** que tais algoritmos não existem! Interprete isso como um desafio para inteligência humana.

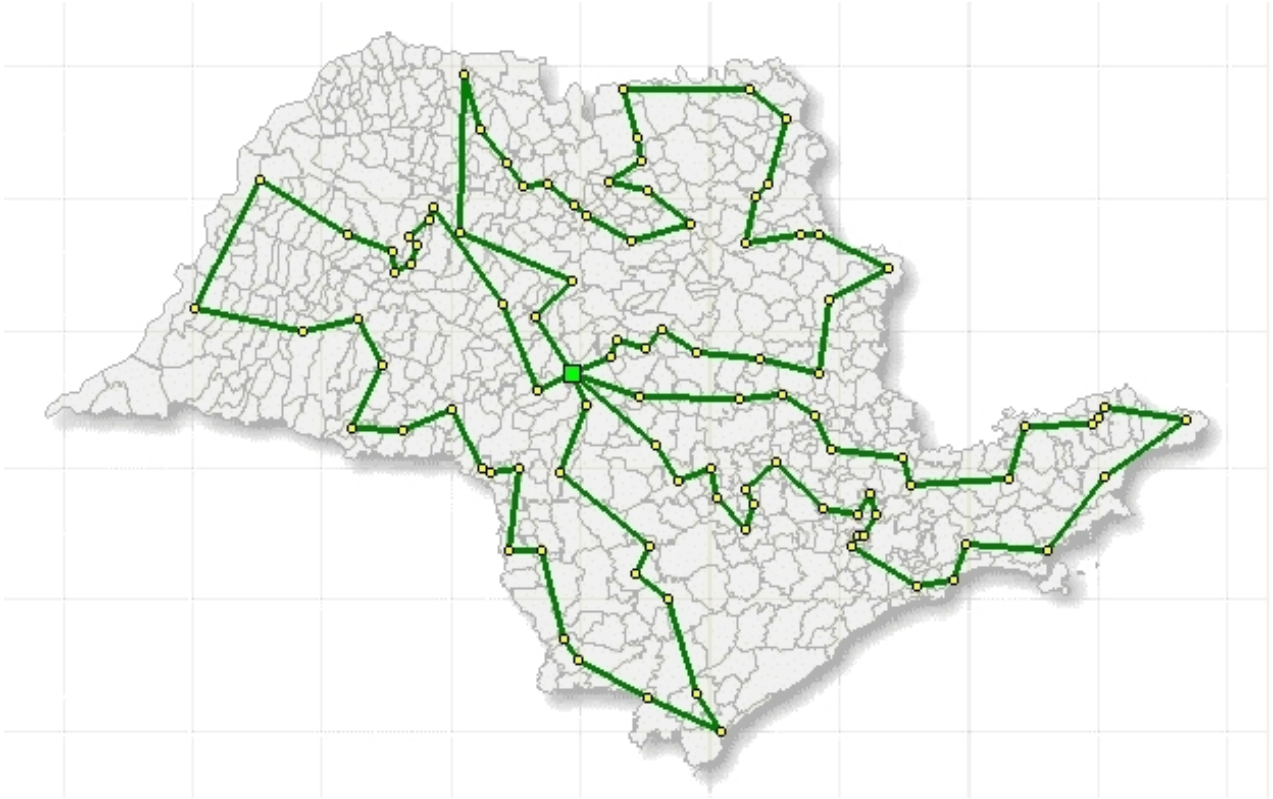
- Esses problemas tem a característica notável de que se um deles admitir um algoritmo "eficiente" então todos admitem algoritmos "eficientes".

- Por que devo me preocupar com problemas $\mathcal{NP}$-sei-lá-o-quê?

  Problemas dessa classe surgem em inúmeras situações práticas, como problemas $\mathcal{NP}$-difíceis.

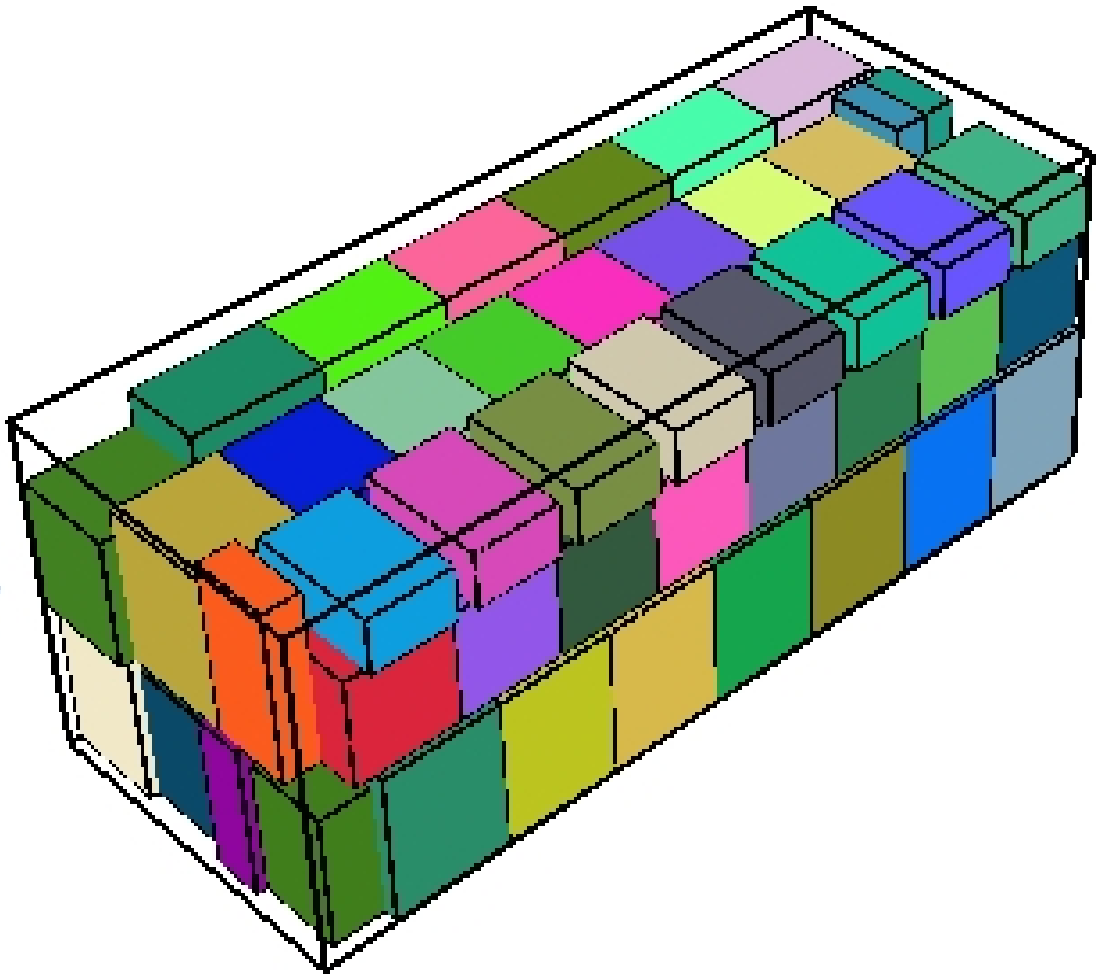
Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida. (vehicle routing)



Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular o número mínimo de *containers* para transportar um conjunto de caixas com produtos. (bin packing 3D)



Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica. (facility location)

e muito mais...

É importante saber indentificar quando estamos lidando com um problema $\mathcal{NP}$-difícil!

Algoritmos e tecnologia

- Condição ideal (irreal): os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta disciplina é inútil!
- O mundo real: há computadores com velocidade de processamento na ordem de bilhões de instruções por segundo e trilhões de bytes em memória.
- Mas ainda assim temos uma limitação na velocidade de processamento e memória dos computadores.

Neste caso faz **muita** diferença ter um **bom** algoritmo.

Algoritmos e tecnologia

Exemplo: **ordenação de um vetor de n elementos**

- Suponha que os computadores A e B executam $1G$ e $10M$ instruções por segundo, respectivamente. Ou seja, A é **100 vezes mais rápido** que B.
- Algoritmo 1: implementado em A por um excelente programador em linguagem de máquina (ultra-rápida). Executa $2n^2$ instruções.
- Algoritmo 2: implementado na máquina B por um programador mediano em linguagem de alto nível dispondo de um compilador "meia-boca". Executa $50n \log n$ instruções.

Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um milhão de elementos? **Qual algoritmo é mais rápido?**
- Algoritmo 1 na máquina *A*:
$$\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000 \text{ segundos}$$
- Algoritmo 2 na máquina *B*:
$$\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos}$$
- Ou seja, *B* foi **VINTE VEZES** mais rápido do que *A*!
- Se o vetor tiver 10 milhões de elementos, esta razão será de 2.3 dias para 20 minutos!

Algoritmos e tecnologia

E se tivermos os tais problemas $\mathcal{NP}$-difíceis ?

$f(n)$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
n	$2{,}0\times10^{-11}$seg	$4{,}0\times10^{-11}$seg	$6{,}0\times10^{-11}$seg	$8{,}0\times10^{-11}$seg	$1{,}0\times10^{-10}$seg
n^2	$4{,}0\times10^{-10}$seg	$1{,}6\times10^{-9}$seg	$3{,}6\times10^{-9}$seg	$6{,}4\times10^{-9}$seg	$1{,}0\times10^{-8}$seg
n^3	$8{,}0\times10^{-9}$seg	$6{,}4\times10^{-8}$seg	$2{,}2\times10^{-7}$seg	$5{,}1\times10^{-7}$seg	$1{,}0\times10^{-6}$seg
n^5	$2{,}2\times10^{-6}$seg	$1{,}0\times10^{-4}$seg	$7{,}8\times10^{-4}$seg	$3{,}3\times10^{-3}$seg	$1{,}0\times10^{-2}$seg
2^n	$1{,}0\times10^{-6}$seg	1,0seg	13,3dias	$1{,}3\times10^{5}$séc	$1{,}4\times10^{11}$séc
3^n	$3{,}4\times10^{-3}$seg	140,7dias	$1{,}3\times10^{7}$séc	$1{,}7\times10^{19}$séc	$5{,}9\times10^{28}$séc

Supondo um computador com velocidade de 1 Terahertz (mil vezes mais rápido que um computador de 1 Gigahertz).

Algoritmos e tecnologia – Conclusões

- O uso de um algoritmo adequado pode levar a ganhos extraordinários de desempenho.
- Isso pode ser tão importante quanto o projeto de *hardware*.
- A melhora obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.
- As melhorias nos algoritmos produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, etc).

Descrição de algoritmos

Podemos descrever um algoritmo de várias maneiras:

- usando uma linguagem de programação de alto nível: `C`, `Pascal`, `Java` etc
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro do CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.

Exemplo de pseudo-código

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1 \ldots n]$ de modo que fique crescente.

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

Corretude de algoritmos

- Um algoritmo (que resolve um determinado problema) é determinístico se, para toda instância do problema, ele pára e devolve uma resposta correta.
- Algoritmos probabilísticos são algoritmos que utilizam passos probabilísticos (como obter um número aleatório).

  Estes algoritmos podem errar ou gastar muito tempo, mas neste caso, queremos que a probabilidade de errar ou de executar por muito tempo seja muuuuito pequena.
- O curso será focado principalmente em algoritmos determinísticos, mas veremos alguns exemplos de algoritmos probabilísticos.

Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo pára. Se ele for muito leeeeeeeeeeeeento terá pouca utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (rápidos).
- Mas o que seria uma boa medida de eficiência de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem qual a máquina foi usada!
- Queremos um critério uniforme para comparar algoritmos.

Modelo Computacional

- Uma possibilidade é definir um modelo computacional de um máquina.
- O modelo computacional estabelece quais os recursos disponíveis, as instruções básicas e quanto elas custam (= tempo).
- Dentre desse modelo, podemos estimar através de uma análise matemática o tempo que um algoritmo gasta em função do tamanho da entrada
  (= análise de complexidade).
- A análise de complexidade depende sempre do modelo computacional adotado.

Tamanho da entrada

Problema: Primalidade

Entrada: inteiro n

Tamanho: número de bits de $n \approx \lg n = \log_2 n$

Problema: Ordenação

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

Tamanho: $n \lg U$ onde U é o maior número em $A[1 \ldots n]$

Medida de complexidade e eficiência de algoritmos

- A complexidade de tempo (= eficiência) de um algoritmo é o número de instruções básicas que ele executa em função do tamanho da entrada.
- Adota-se uma "atitude pessimista" e faz-se uma análise de pior caso.
  Determina-se o tempo máximo necessário para resolver uma instância de um certo tamanho.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE = análise assintótica.

Medida de complexidade e eficiência de algoritmos

- Um algoritmo é chamado eficiente se a função que mede sua complexidade de tempo é limitada por um **polinômio** no tamanho da entrada.
  Por exemplo: n, $3n - 7$, $4n^2$, $143n^2 - 4n + 2$, n^5.
- Mas por que polinômios?
  Resposta padrão: (polinômios são funções bem "comportadas").

Vantagens do método de análise proposto

- O modelo RAM é robusto e permite prever o comportamento de um algoritmo para instâncias GRANDES.
- O modelo permite comparar algoritmos que resolvem um mesmo problema.
- A análise é mais robustas em relação às evoluções tecnológicas .

Desvantagens do método de análise proposto

- Fornece um limite de complexidade pessimista sempre considerando o pior caso.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma freqüência e é possível que as “instâncias ruins” ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no caso médio.
- A análise de complexidade de algoritmos no caso médio é complicada e depende do conhecimento da distribuição das instâncias.

Começando a trabalhar

Ordenação

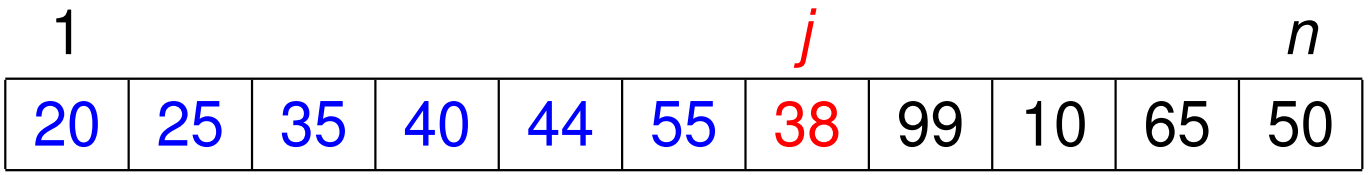
Problema: ordenar um vetor em ordem crescente

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

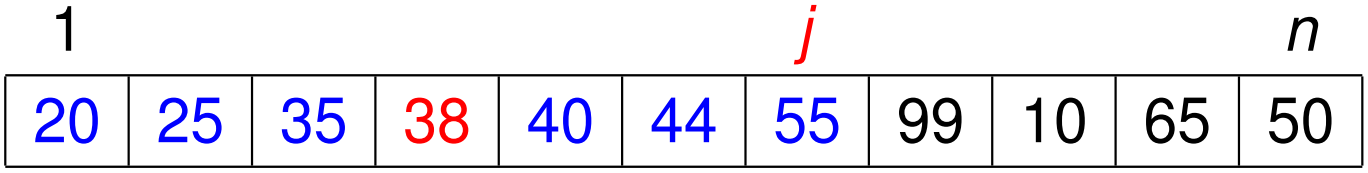
Saída: vetor $A[1 \ldots n]$ rearranjado em ordem crescente

Vamos começar estudando o algoritmo de ordenação baseado no método de inserção.

Inserção em um vetor ordenado

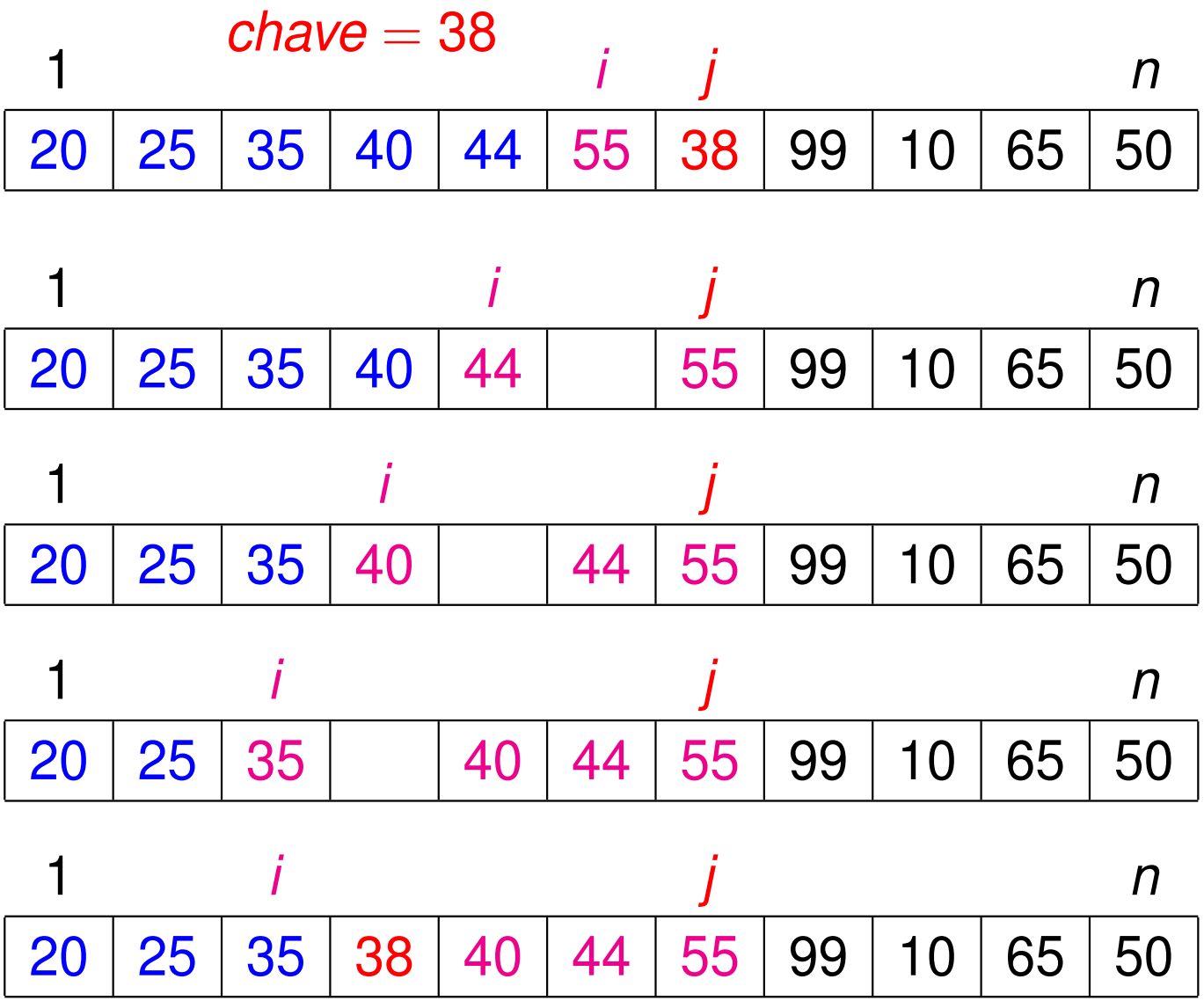


- O subvetor $A[1 \ldots j-1]$ está ordenado.
- Queremos inserir a $chave = 38 = A[j]$ em $A[1 \ldots j-1]$ de modo que no final tenhamos:

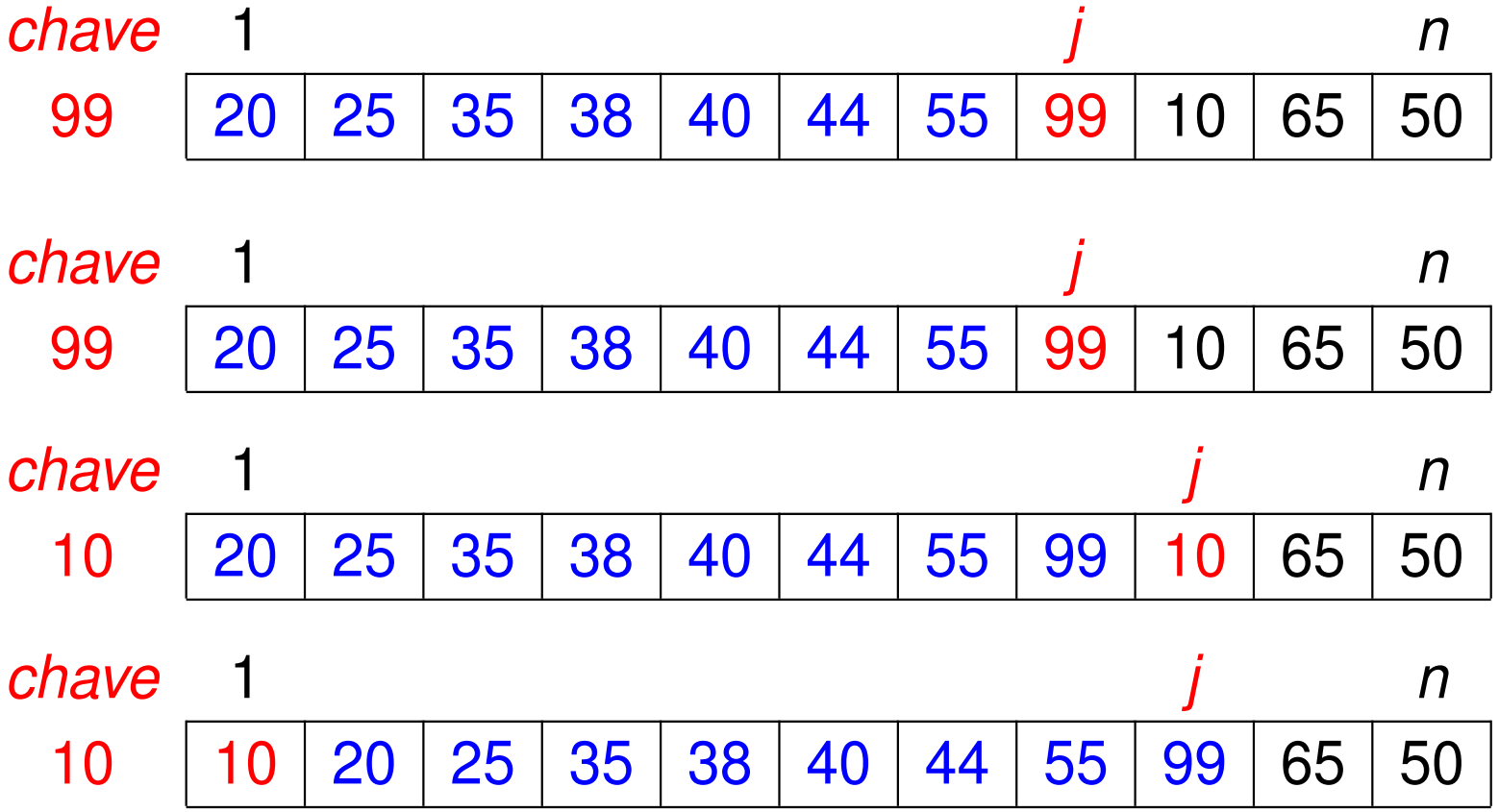


- Agora $A[1 \ldots j]$ está ordenado.

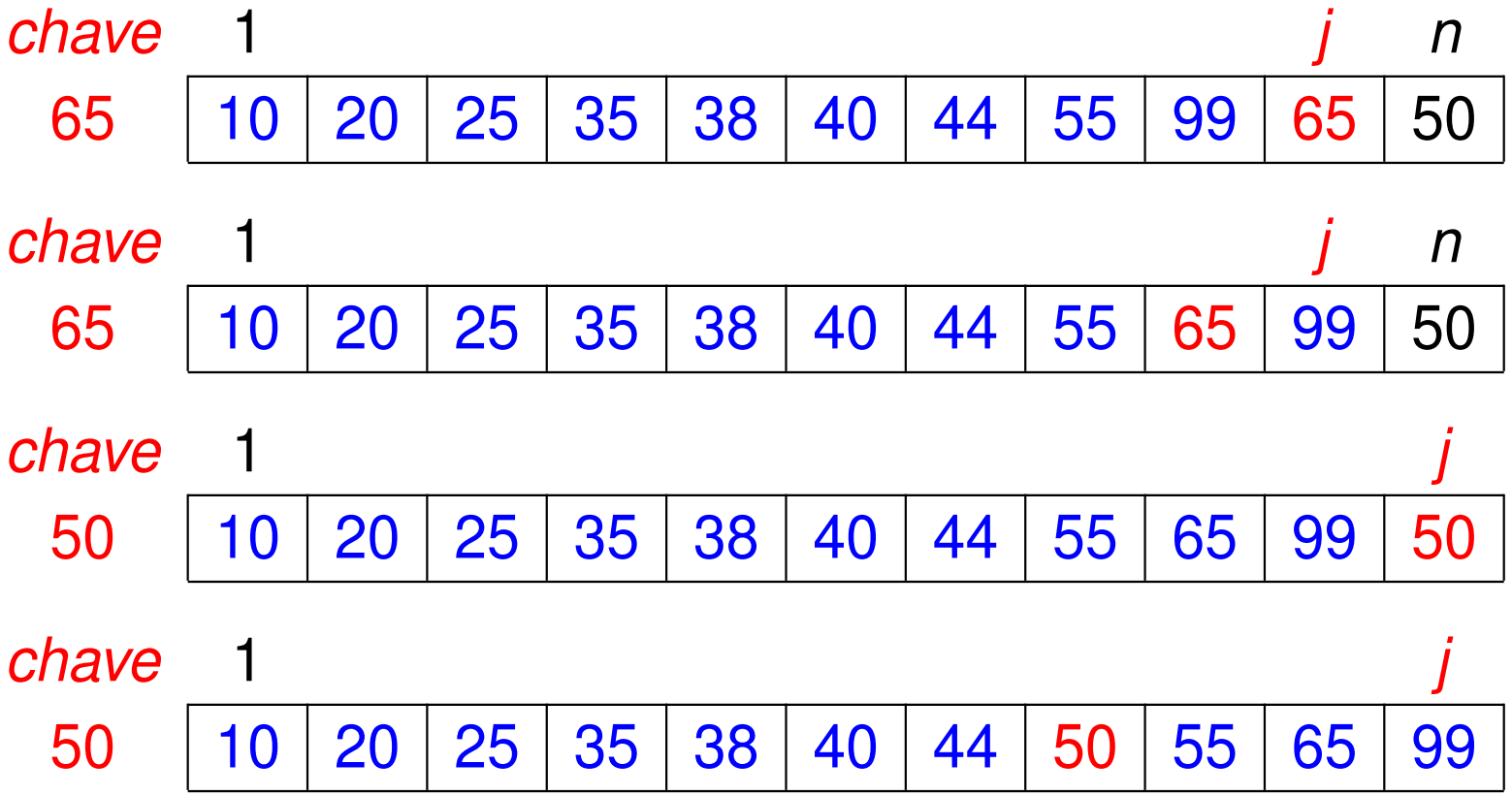
Como fazer a inserção



Ordenação por inserção



Ordenação por inserção



Ordena-Por-Inserção

Pseudo-código

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

Análise do algoritmo

O que é importante analisar ?

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

O algoritmo pára

```
ORDENA-POR-INSERÇÃO(A, n)
1   para j ← 2 até n faça
    ...
4       i ← j − 1
5       enquanto i ≥ 1  e A[i] > chave faça
6           ...
7           i ← i − 1
8       ...
```

No laço **enquanto** na linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução pára em algum momento por causa do teste condicional $i \geq 1$.

O laço na linha 1 evidentemente pára (o contador j atingirá o valor $n + 1$ após $n - 1$ iterações).

Portanto, o algoritmo **pára**.

Ordena-Por-Inserção

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

O que falta fazer ?

- Verificar se ele produz uma resposta correta.
- Analisar sua complexidade de tempo.

Invariantes de laço e provas de corretude

- Definição: um invariante de um laço é uma propriedade que relaciona as variáveis do algoritmo a cada execução completa do laço.
- Ele deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- A prova de corretude de um algoritmo requer que sejam encontrados e provados invariantes dos vários laços que o compõem.
- Em geral, é mais difícil descobrir um invariante apropriado do que mostrar sua validade se ele for dado de bandeja. . .

Exemplo de invariante

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

Invariante principal de ORDENA-POR-INSERÇÃO: (i1)

No começo de cada iteração do laço **para** das linha 1–8, o subvetor $A[1 \ldots j-1]$ está ordenado.

Corretude de algoritmos por invariantes

A estratégia “típica” para mostrar a corretude de um algoritmo iterativo através de invariantes segue os seguintes passos:

1. Mostre que o invariante vale no início da primeira iteração (trivial, em geral)
2. Suponha que o invariante vale no início de uma iteração qualquer e prove que ele vale no ínicio da próxima iteração
3. Conclua que se o algoritmo pára e o invariante vale no ínicio da última iteração, então o algoritmo é correto.

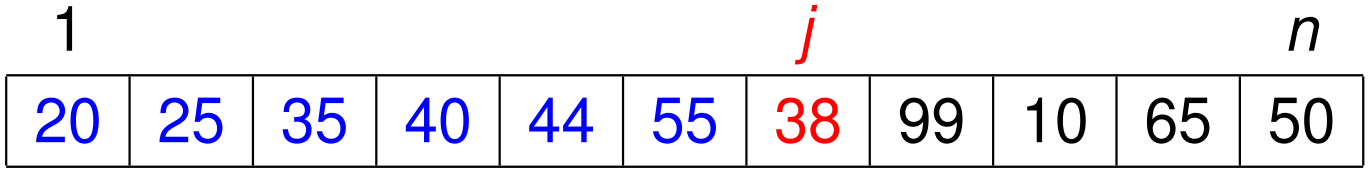
Note que (1) e (2) implicam que o invariante vale no início de qualquer iteração do algoritmo. Isto é similar ao método de indução matemática ou indução finita!

Corretude da ordenação por inserção

Vamos verificar a corretude do algoritmo de ordenação por inserção usando a técnica de **prova por invariantes de laços**.

Invariante principal: (i1)

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1 \ldots j-1]$ está ordenado.



- Suponha que o invariante vale.
- Então a corretude do algoritmo é "evidente". **Por quê?**
- No ínicio da última iteração temos $j = n + 1$. Assim, do invariante segue que o (sub)vetor $A[1 \ldots n]$ está ordenado!

Melhorando a argumentação

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

Um invariante mais preciso: **(i1')**

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1 \ldots j-1]$ é uma permutação ordenada do subvetor original $A[1 \ldots j-1]$.

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1 \ldots 1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

   Uma demonstração mais formal deste fato exige invariantes auxiliares para o laço interno **enquanto**.
3. Corretude do algoritmo: na última iteração, temos $j = n + 1$ e logo $A[1 \ldots n]$ está ordenado com os elementos originais do vetor. Portanto, o algoritmo é correto.

Invariantes auxiliares

No início da linha 5 valem os seguintes invariantes:

(i2) $A[1 \ldots i]$, *chave* e $A[i+2 \ldots j]$ contém os elementos de $A[1 \ldots j]$ antes de entrar no laço que começa na linha 5.

(i3) $A[1 \ldots i]$ e $A[i+2 \ldots j]$ são crescentes.

(i4) $A[1 \ldots i] \leq A[i+2 \ldots j]$

(i5) $A[i+2 \ldots j] > \textit{chave}$.

Invariantes (i2) a (i5)
+condição de parada na linha 5
+atribuição da linha 7
$\Longrightarrow$ invariante (i1')

Demonstração? Mesma que antes.

Complexidade do algoritmo

- Vamos tentar determinar o tempo de execução (ou complexidade de tempo) de ORDENA-POR-INSERÇÃO em função do tamanho de entrada.
- Para o problema de Ordenação vamos usar como tamanho de entrada a dimensão do vetor e ignorar o valores dos seus elementos (modelo RAM).
- A complexidade de tempo de um algoritmo é o número de instruções básicas (operações elementares ou primitivas) que executa a partir de uma entrada.
- Exemplo: comparação e atribuição entre números ou variáveis numéricas, operações aritméticas, etc.

Vamos contar ?

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

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.

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.

Melhor caso

O melhor caso de Ordena-Por-Inserção ocorre quando o vetor A já está ordenado. Para $j = 2, \ldots, n$ temos $A[i] \leq chave$ na linha 5 quando $i = j - 1$. Assim, $t_j = 1$ para $j = 2, \ldots, n$.

Logo,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Este tempo de execução é da forma $an + b$ para constantes a e b que dependem apenas dos c_i.

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

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.

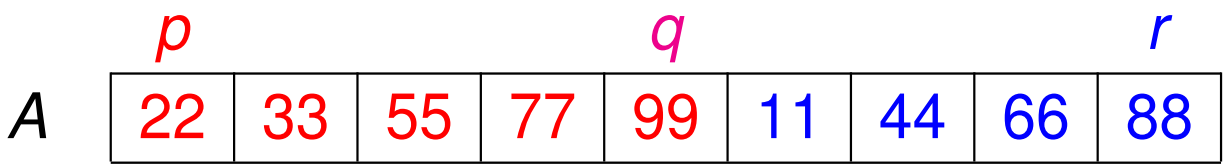
Ordenação por intercalação

Ordenação por intercalação

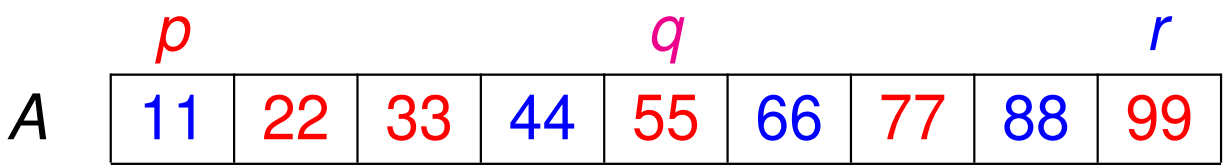
Q que significa intercalar dois (sub)vetores ordenados?

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.

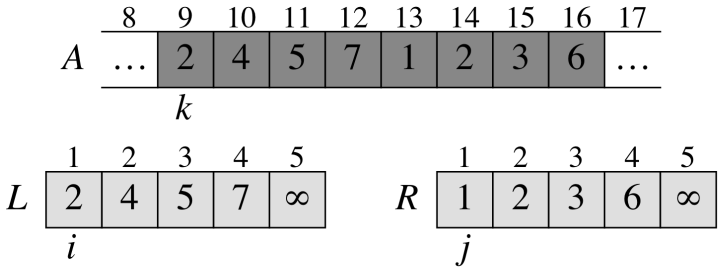
Entrada:



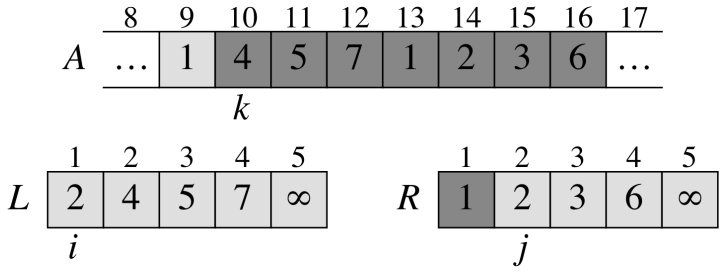
Saída:



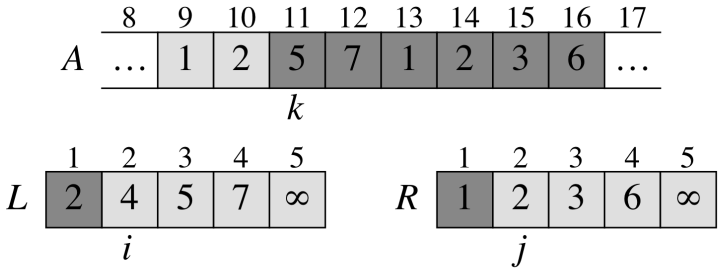
Intercalação com sentinela



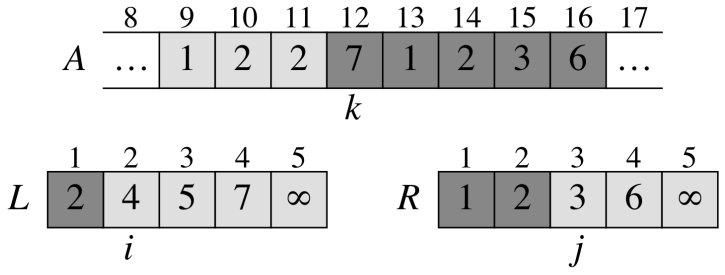
(a)



(b)

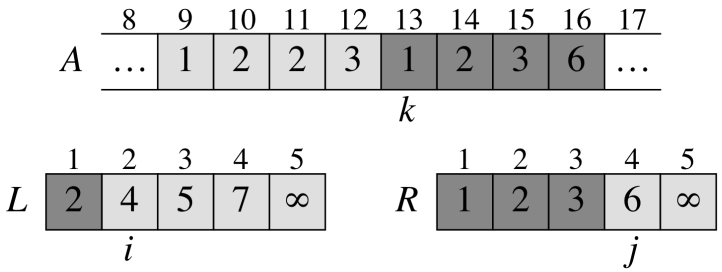


(c)

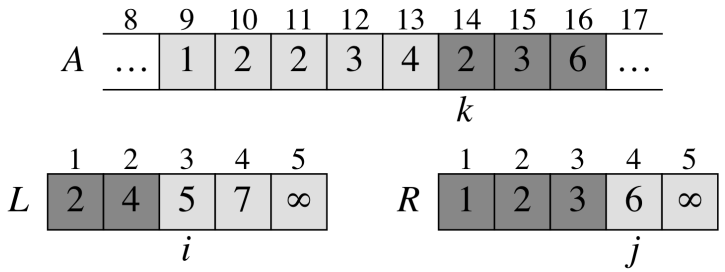


(d)

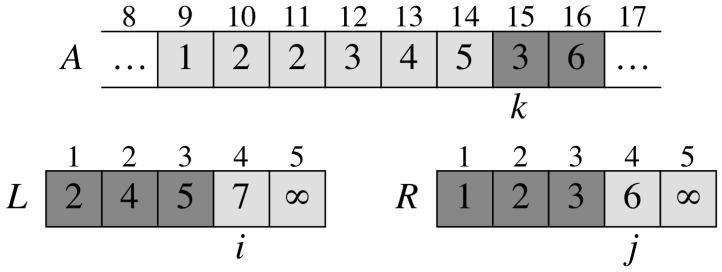
Intercalação com sentinela



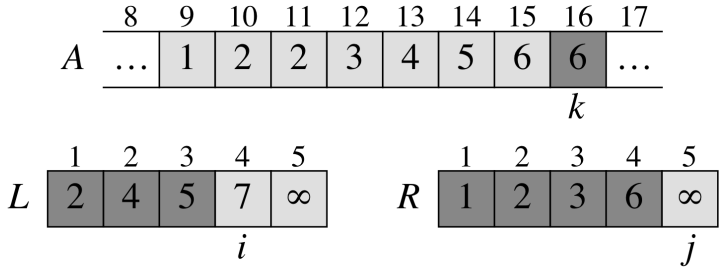
(e)



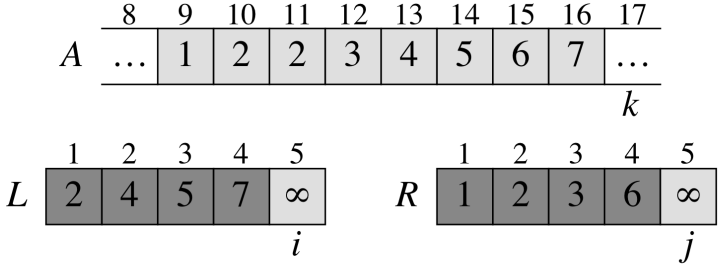
(f)



(g)



(h)



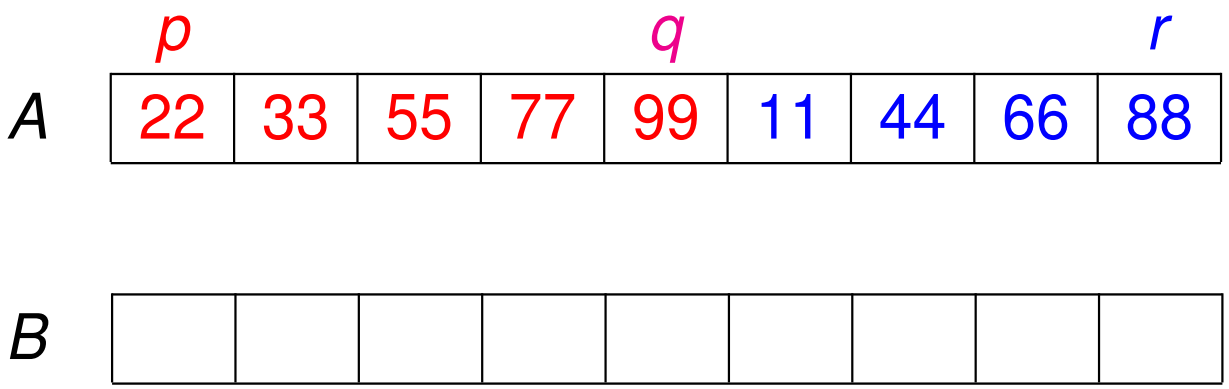
(i)

Intercalação com sentinela

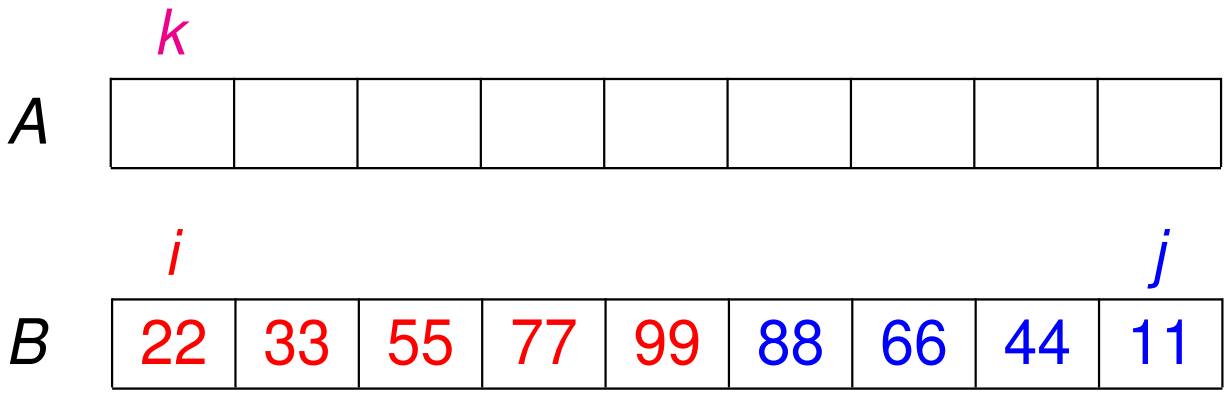
```
INTERCALA(A, p, q, r)
 1: n1 ← q − p + 1
 2: n2 ← r − q
 3: sejam L[1..n1 + 1] e R[1..n2 + 1] novos vetores
 4: para i ← 1 até n1 faça
 5:     L[i] ← A[p + i − 1]
 6: para j ← 1 até n2 faça
 7:     R[j] ← A[q + j]
 8: L[n1 + 1] ← ∞
 9: R[n2 + 1] ← ∞
10: i ← 1
11: j ← 1
12: para k ← p até r faça
13:     se L[i] ≤ R[j] então
14:         A[k] ← L[i]
15:         i ← i + 1
16:     senão
17:         A[k] = R[j]
18:         j ← j + 1
```

Outro algoritmo de intercalação (sem sentinela)

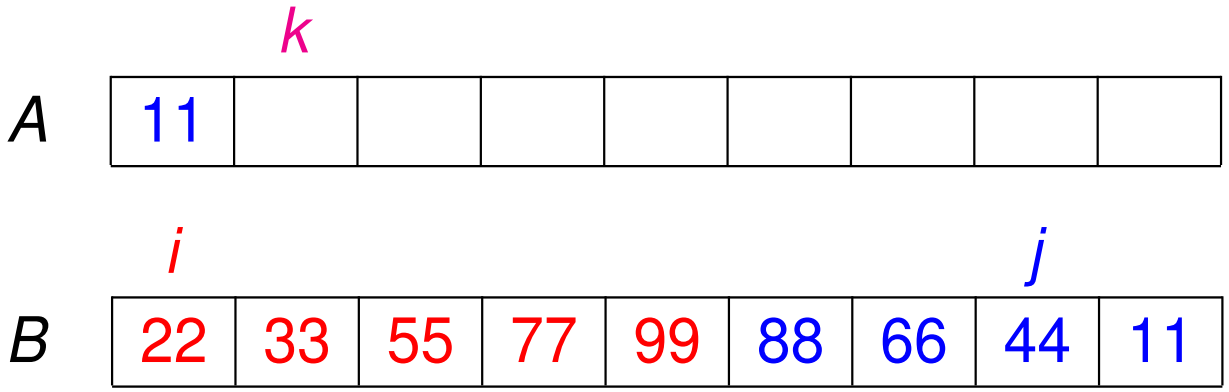
Intercalação



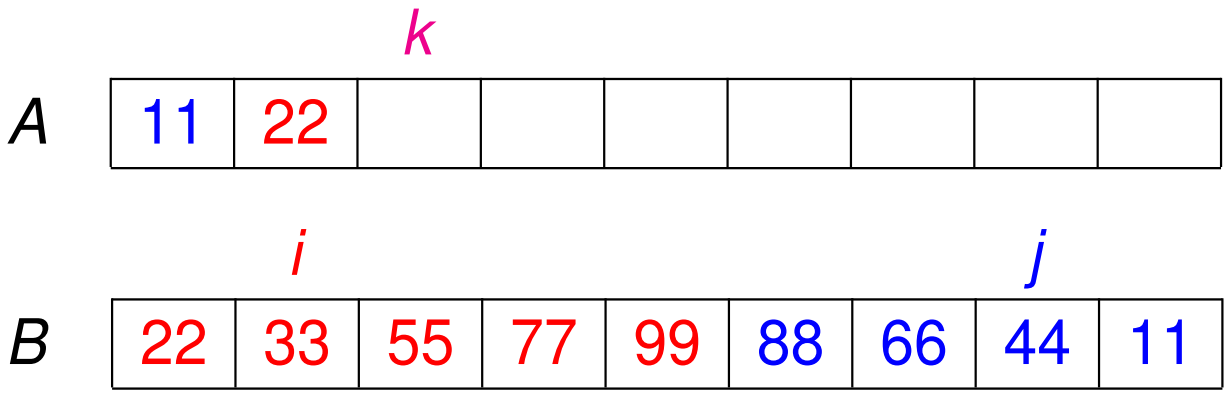
Intercalação



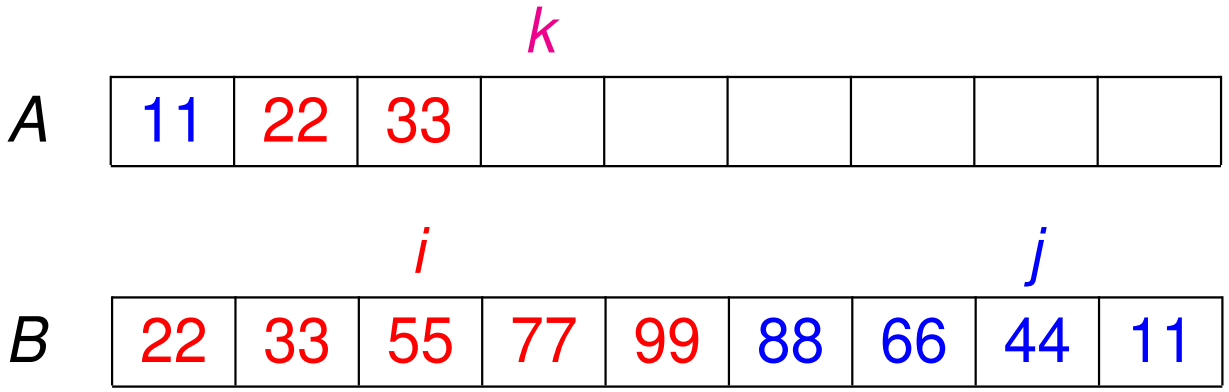
Intercalação



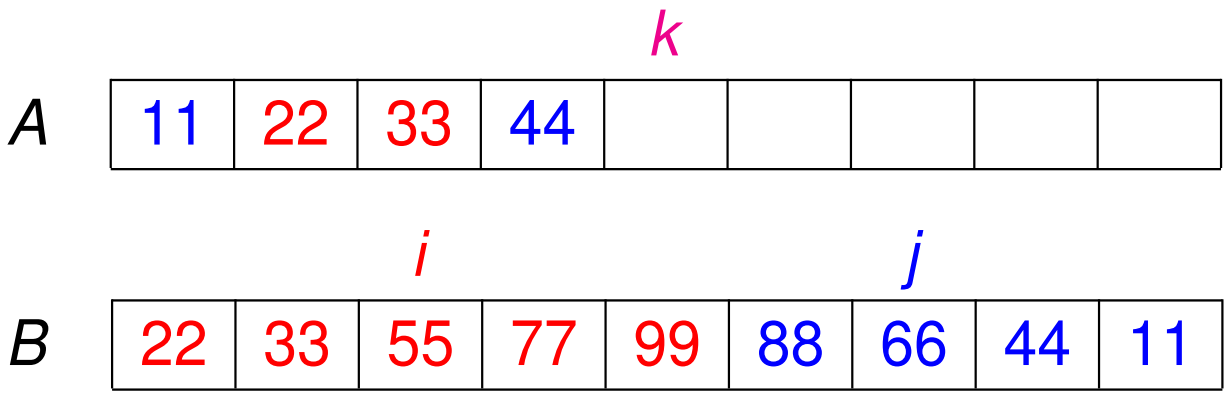
Intercalação



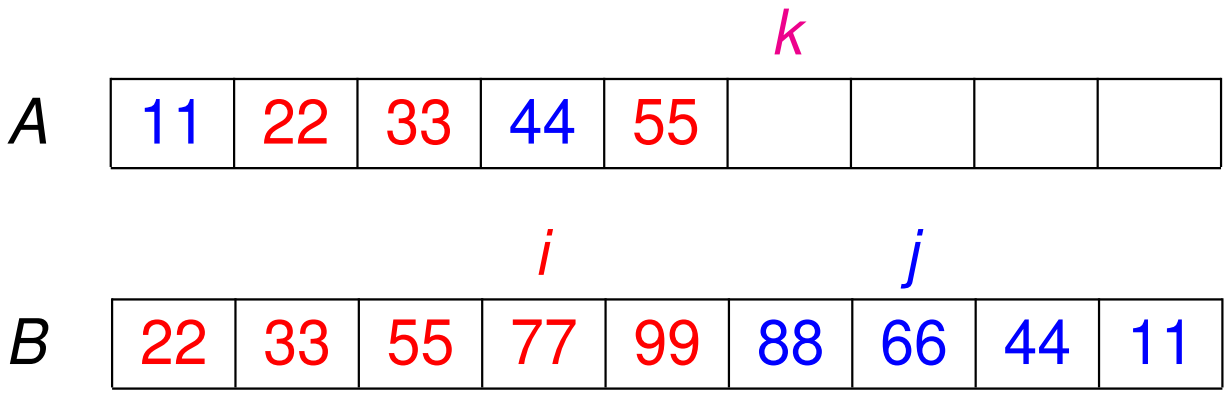
Intercalação



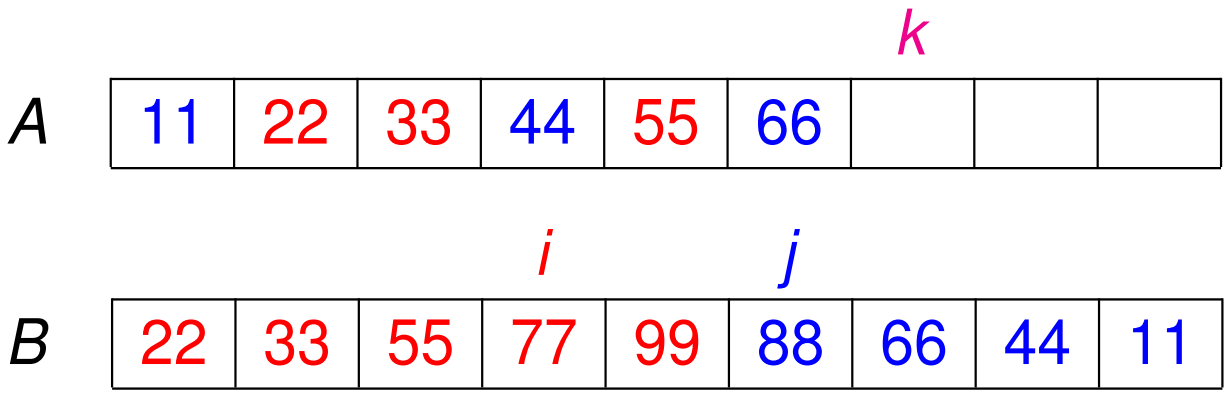
Intercalação



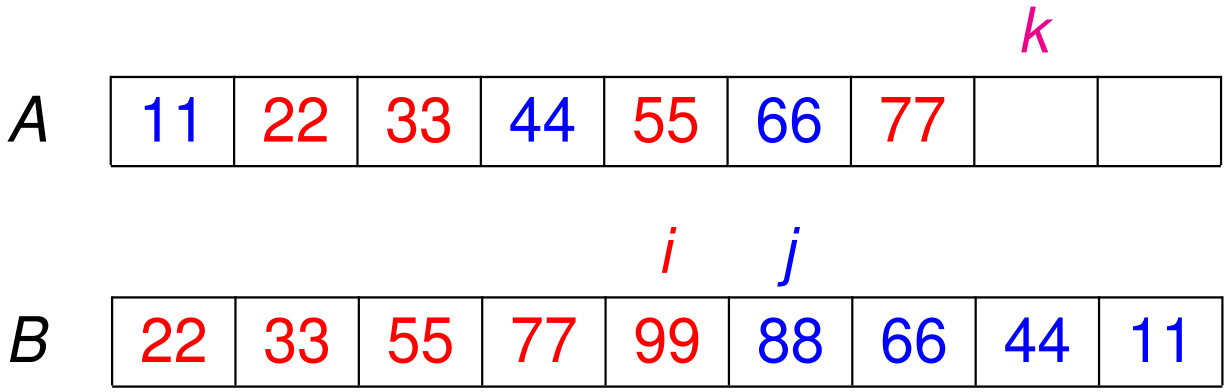
Intercalação



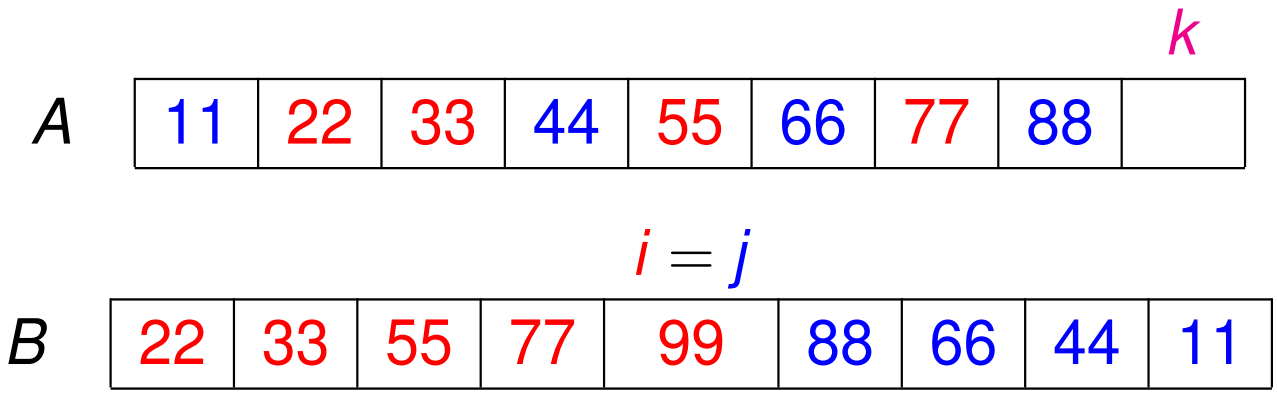
Intercalação



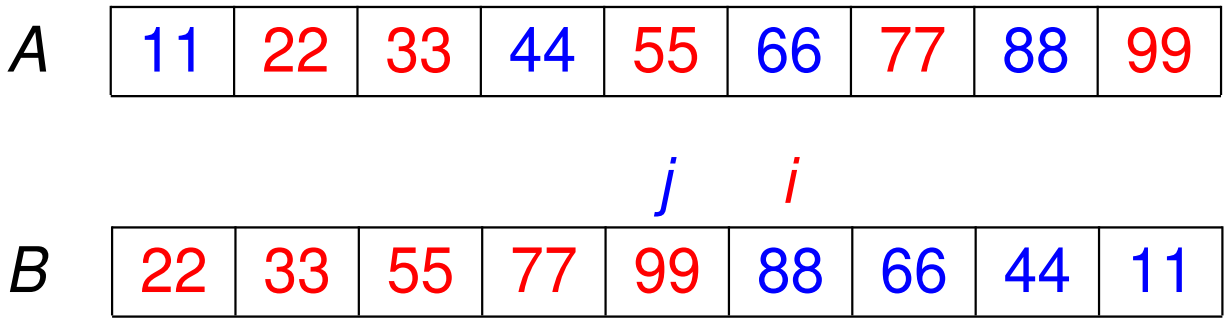
Intercalação



Intercalação



Intercalação

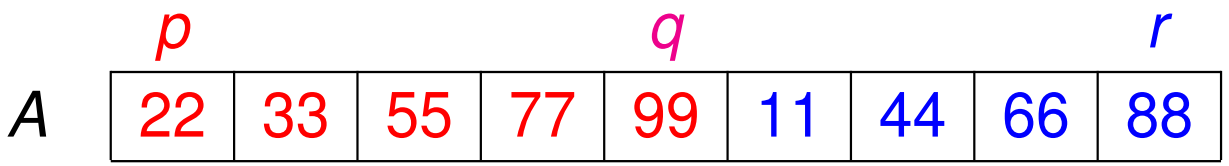


Pseudo-código

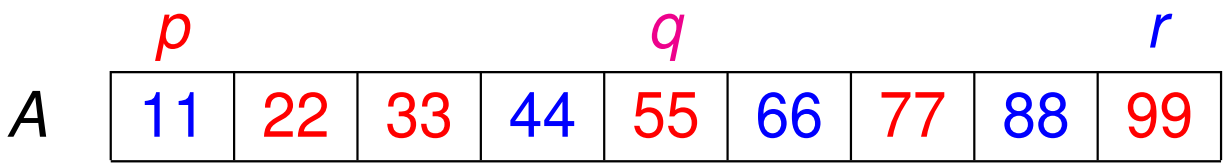
```
INTERCALA(A, p, q, r)
 1   para i ← p até q faça
 2       B[i] ← A[i]
 3   para j ← q + 1 até r faça
 4       B[r + q + 1 − j] ← A[j]
 5   i ← p
 6   j ← r
 7   para k ← p até r faça
 8       se B[i] ≤ B[j]
 9           então A[k] ← B[i]
10                 i ← i + 1
11           senão A[k] ← B[j]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

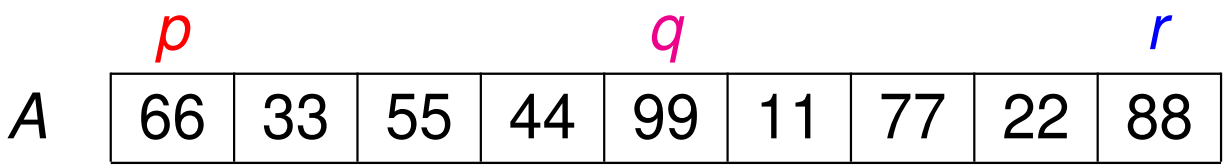
Exemplo de divisão-e-conquista: *Mergesort*

- 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.

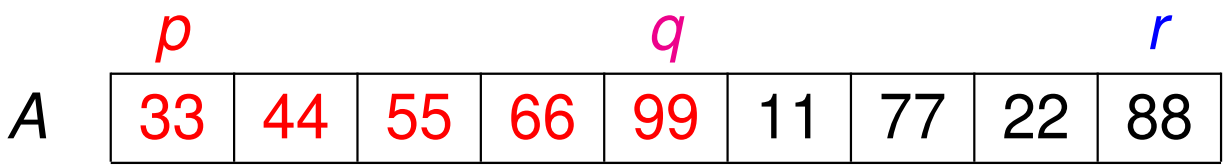
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

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

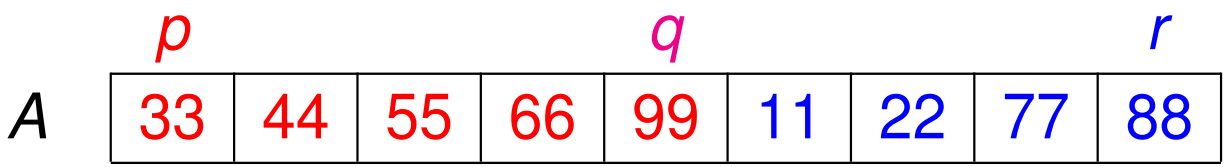
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

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

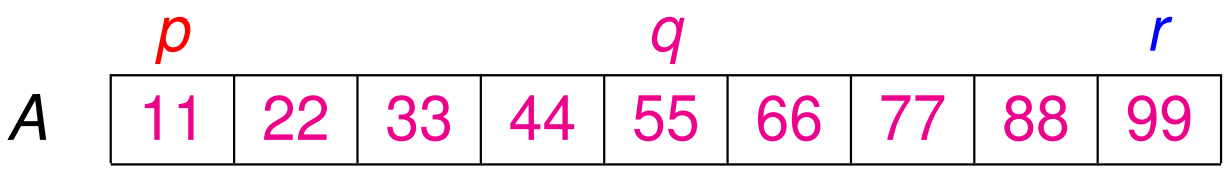
```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```



Mergesort

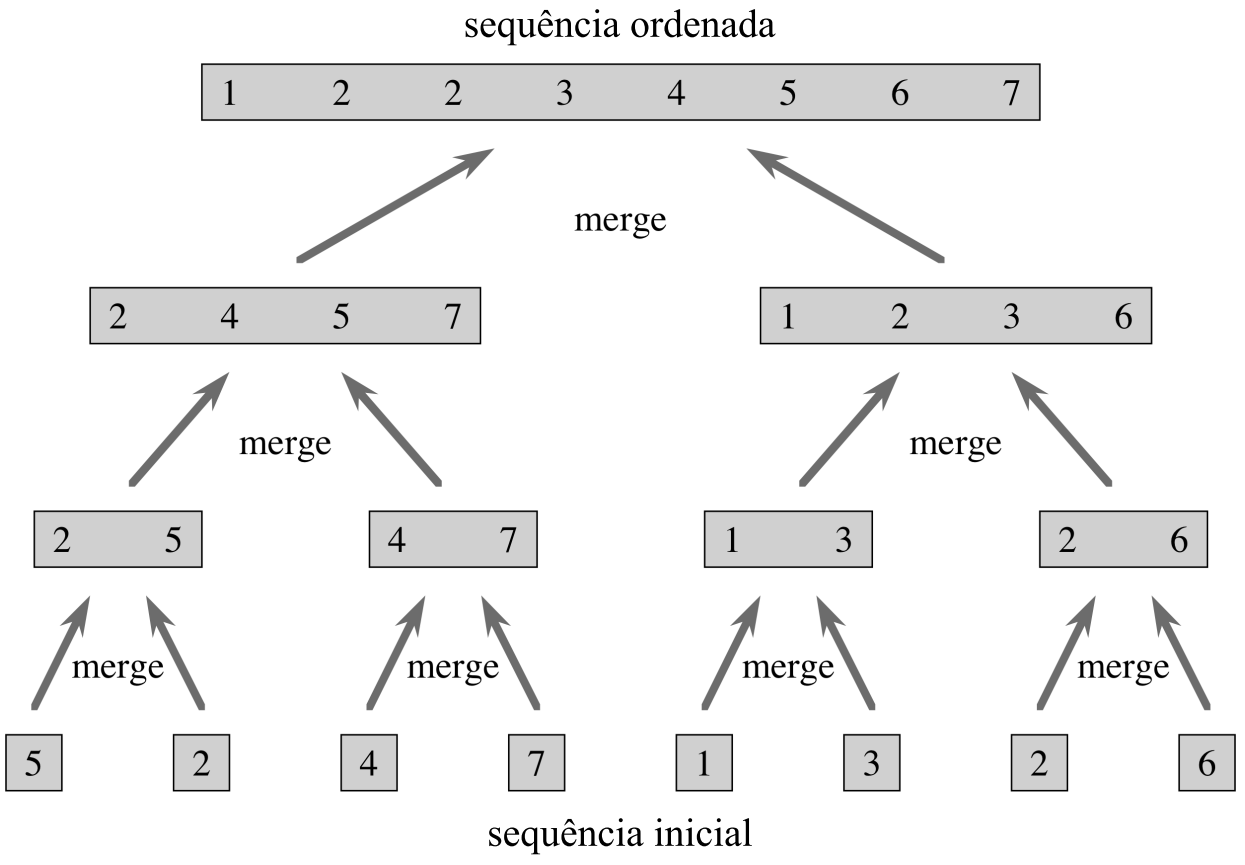
Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
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

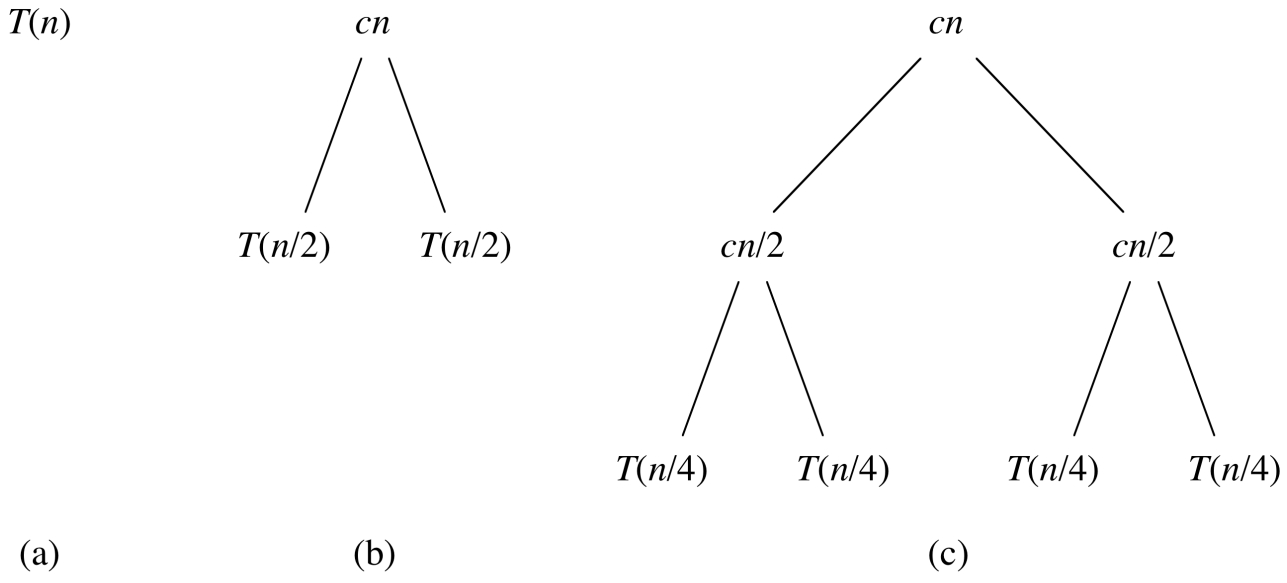
- Obtemos o que chamamos de fórmula de recorrência (i.e., uma fórmula definida em termos de si mesma).

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

- 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.

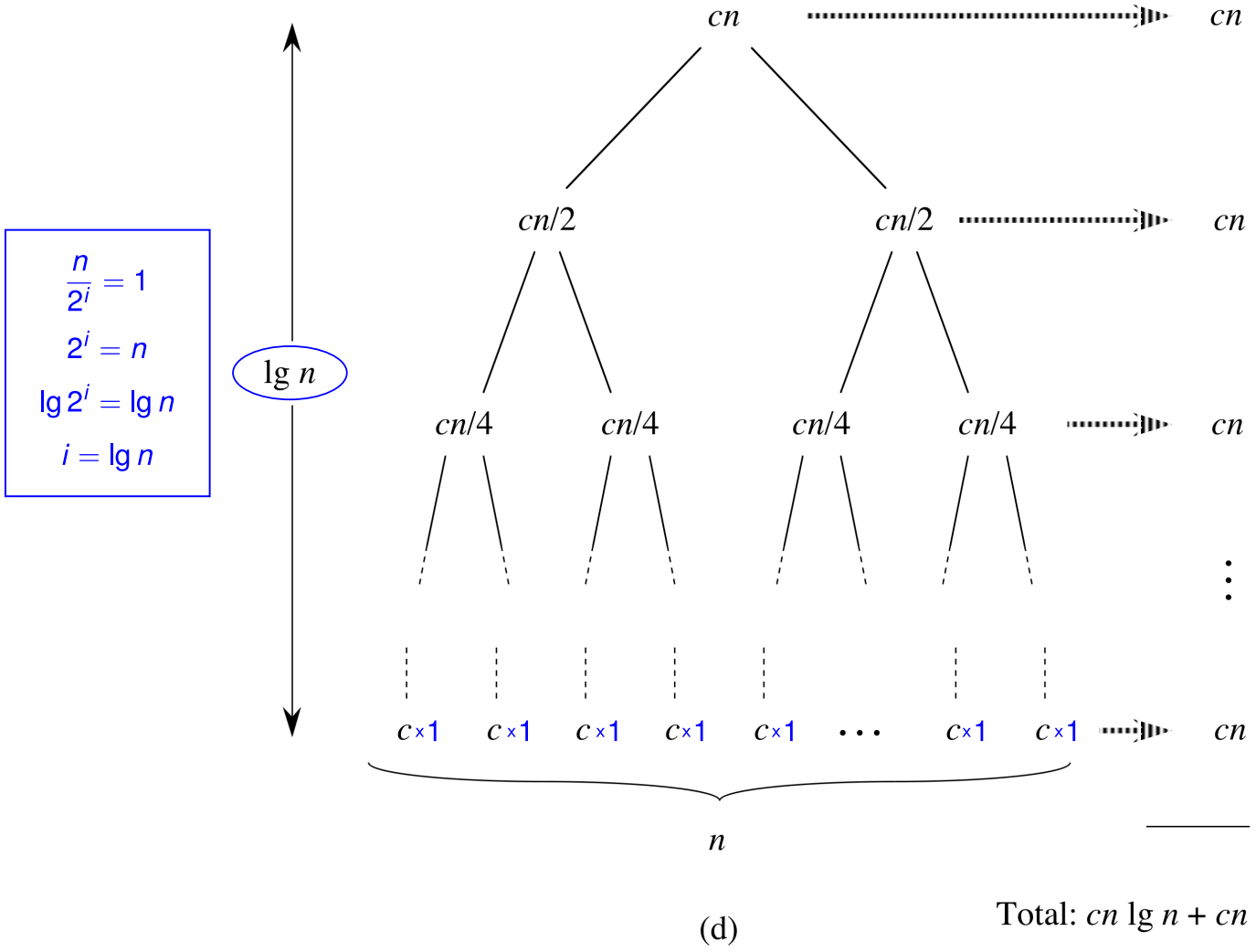
Mergesort – árvore de recursão

- Árvore de recursão do Mergesort



Mergesort – árvore de recursão

- Árvore de recursão do Mergesort



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

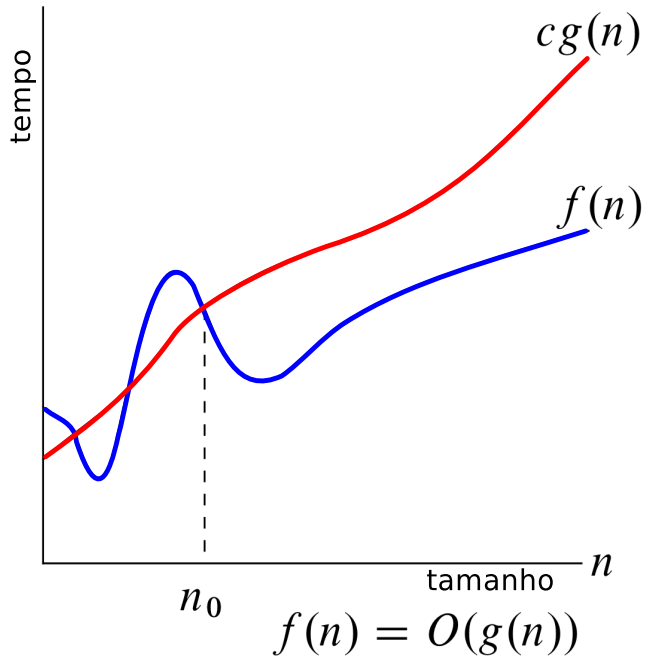
- Precisamos de uma ferramenta matemática para **comparar funções**
- Para a análise de algoritmo será feita uma **análise assintótica**:
  - Matematicamente: estudando o comportamento de **limites** ($n \to \infty$)
  - Computacionalmente: estudando o comportamento para entrada arbitrariamente grande 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 $g(n)$.



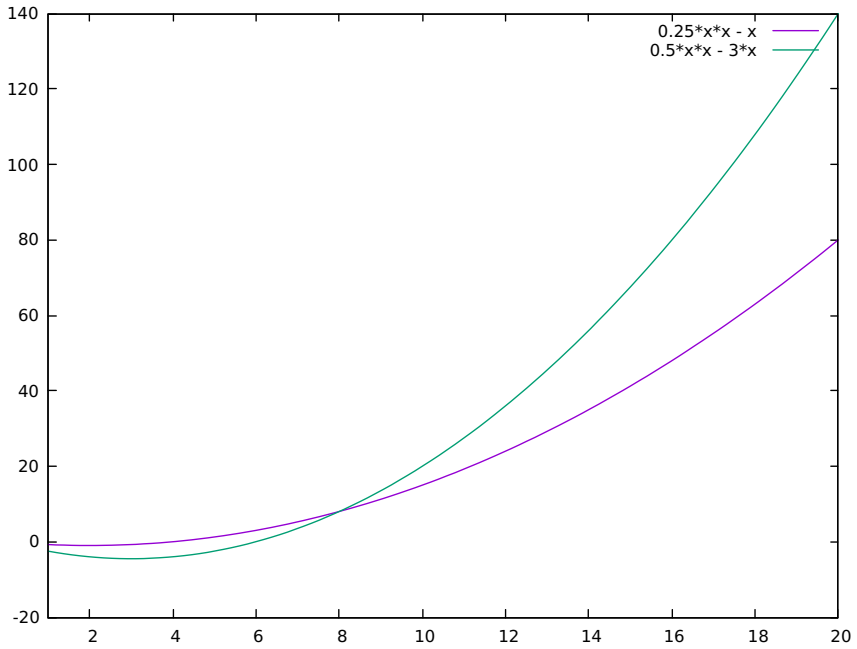
Classe O

Exemplo

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

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

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

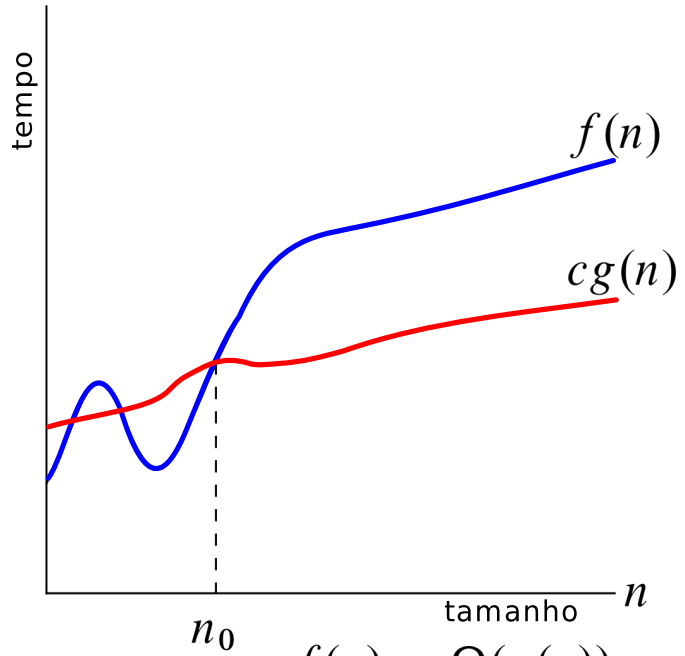


Classe Ω

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 $g(n)$.



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

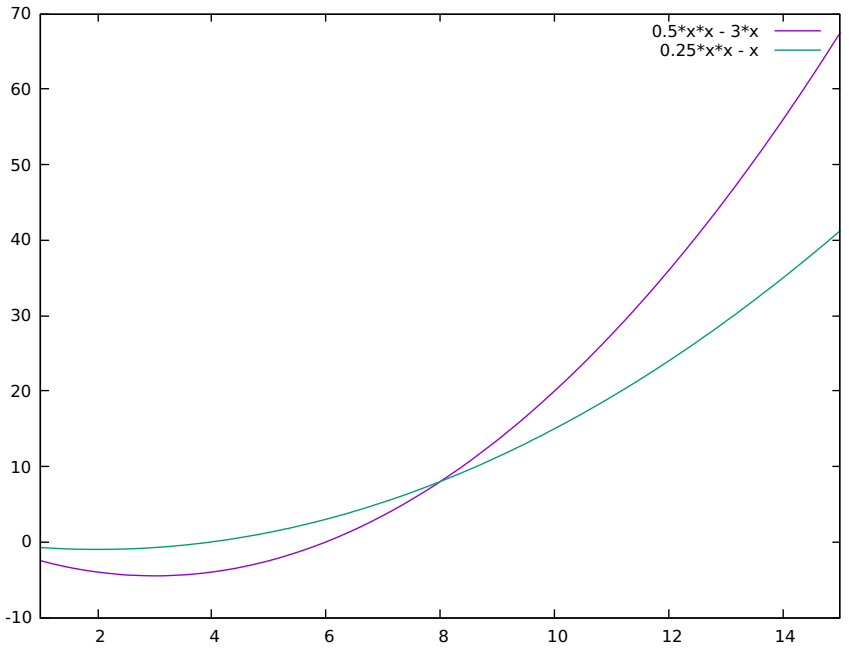
Classe Ω

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 $f(n) \in \Omega(g(n))$:

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

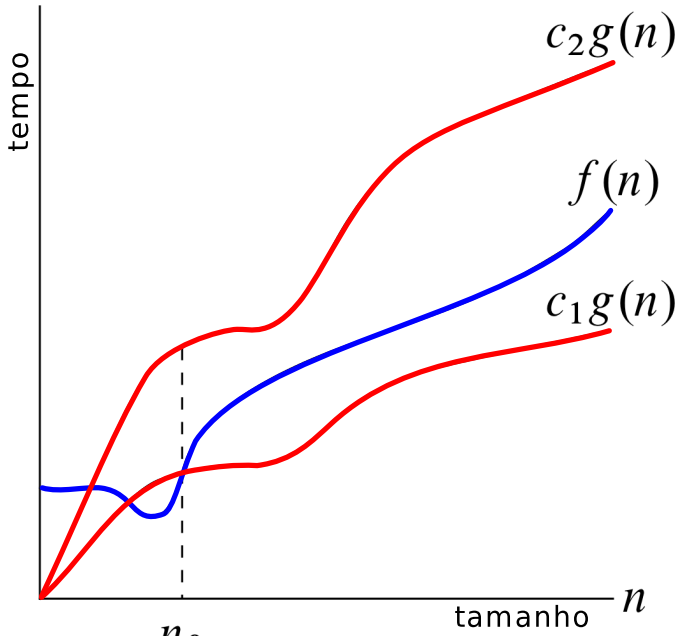


Classe Θ

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

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

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

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

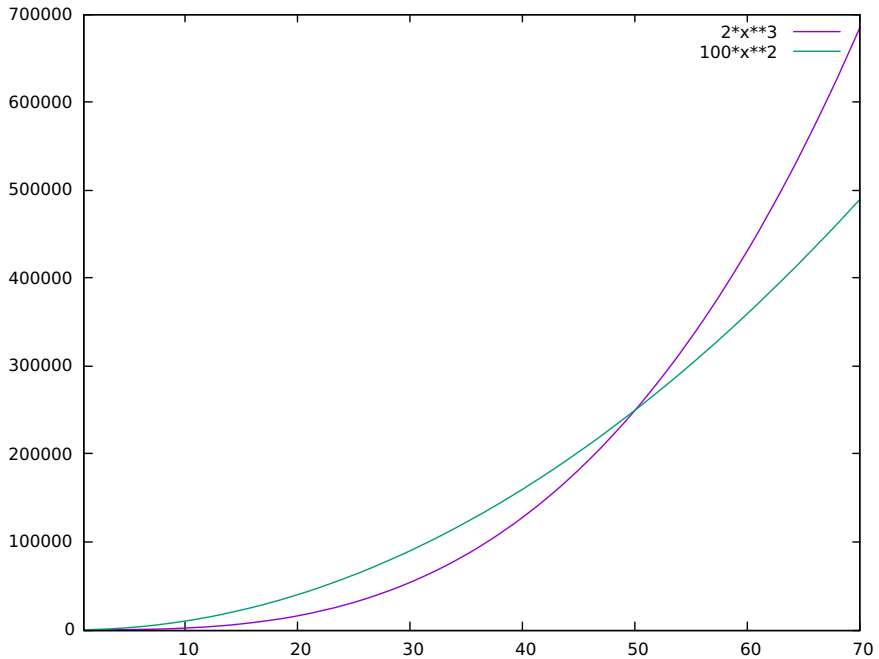
Se uma operação leva 10^{-9} segundos

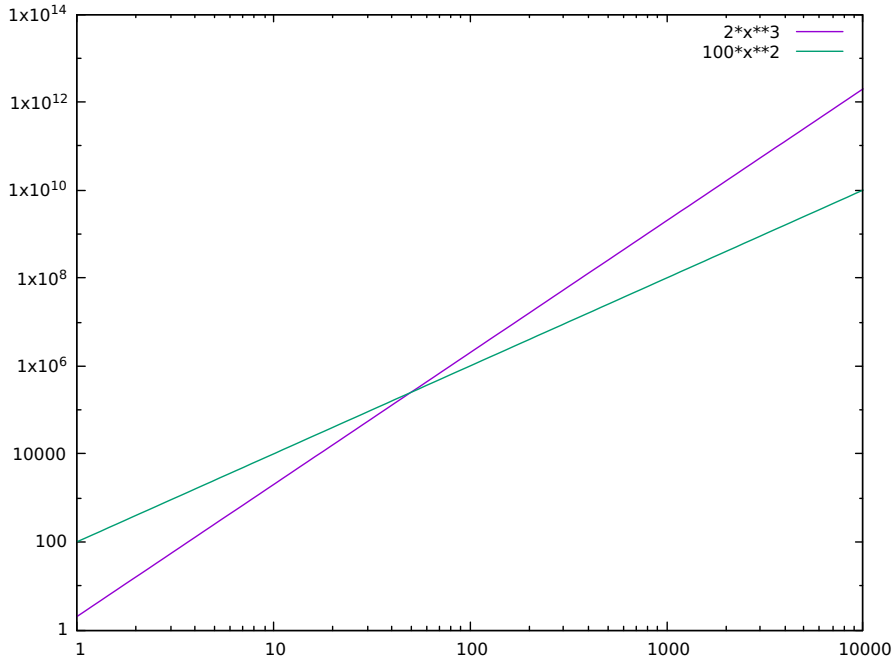
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$
20	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	77 anos
30	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$1.07s$	
40	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	$< 0.01s$	18.3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

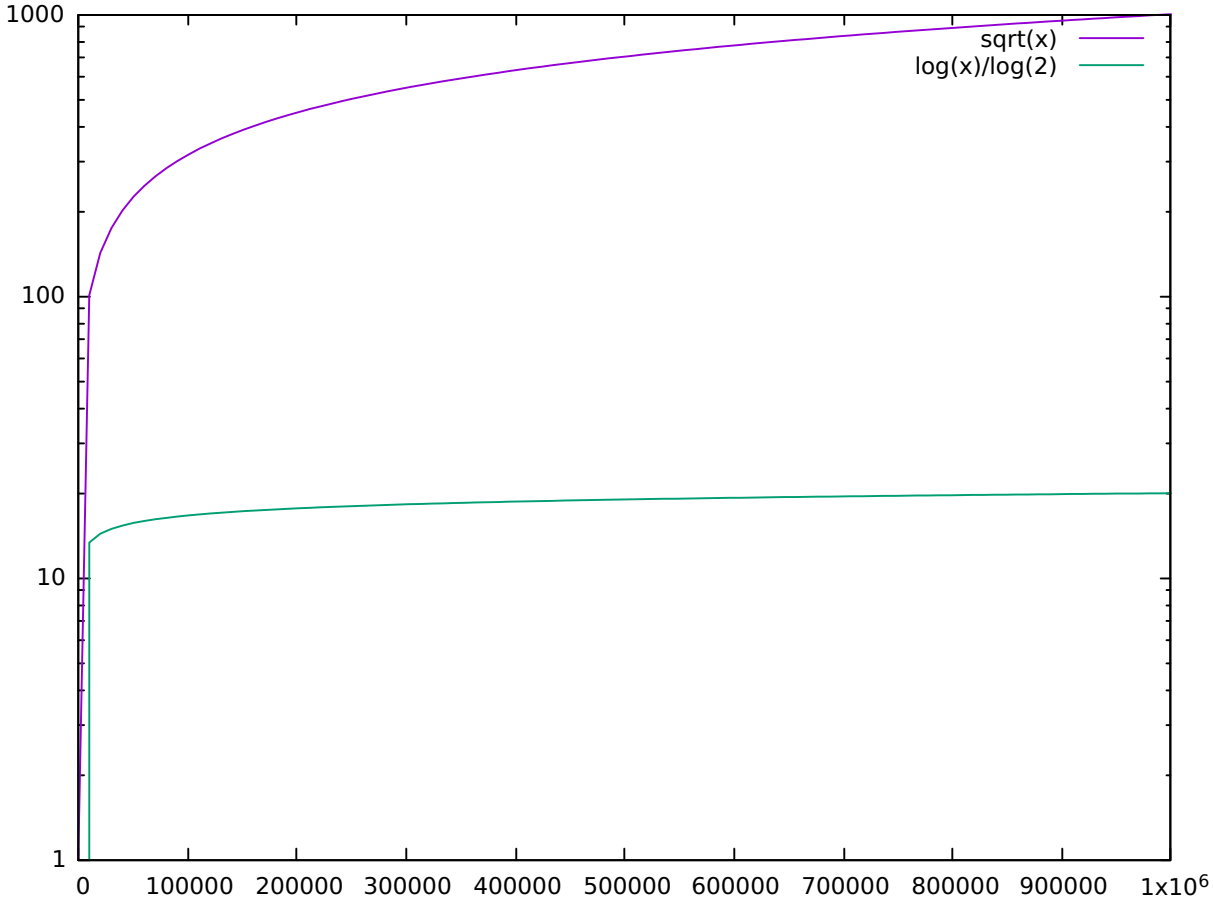




Desenhando funções

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

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



Recorrências

Resolução de Recorrências

- Relações de recorrência expressam a complexidade de algoritmos recursivos como, por exemplo, os algoritmos de divisão e conquista.
- É preciso saber resolver as recorrências para que possamos efetivamente determinar a complexidade dos algoritmos recursivos.

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	b_0
2	b_1
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	an

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + an + (b_0 + b_1)$$

Resolução de recorrências

- Queremos resolver a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + an + b \quad \text{para } n \geq 2. \end{aligned}$$

- Resolver uma recorrência significa encontrar uma fórmula fechada para $T(n)$.
- Não é necessário achar uma solução exata. Basta encontrar uma função $f(n)$ tal que $T(n) \in \Theta(f(n))$.

Resolução de recorrências

Alguns métodos para resolução de recorrências:

- substituição
- iteração
- árvore de recorrência

Veremos também um resultado bem geral que permite resolver várias recorrências: Master theorem.

Método da substituição

- Idéia básica: “adivinhe” qual é a solução e prove por indução que ela funciona!
- Método poderoso mas nem sempre aplicável (obviamente).
- Com prática e experiência fica mais fácil de usar!

Exemplo

Considere a recorrência:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

Chuto que $T(n) \in O(n \lg n)$.

Mais precisamente, chuto que $T(n) \leq 3n \lg n$.

(Lembre que $\lg n = \log_2 n$.)

Exemplo

$$
\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\
&\leq 3 \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + 3 \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\leq 3 \left\lceil \frac{n}{2} \right\rceil \lg n + 3 \left\lfloor \frac{n}{2} \right\rfloor (\lg n - 1) + n \\
&= 3 \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n - 3 \left\lfloor \frac{n}{2} \right\rfloor + n \\
&= 3n \lg n - 3 \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\leq 3n \lg n.
\end{aligned}
$$

(Yeeeeeeesssssss!)

Exemplo

- Mas espere um pouco!
- $T(1) = 1$ e $3.1.\lg 1 = 0$ e a base da indução não funciona!
- Certo, mas lembre-se da definição da classe $O(\ )$.

  Só preciso provar que $T(n) \leq 3n \lg n$ para $n \geq n_0$ onde n_0 é alguma constante.

  Vamos tentar com $n_0 = 2$. Nesse caso

  $$T(2) = T(1) + T(1) + 2 = 4 \leq 3.2.\lg 2 = 6,$$

  e estamos feitos.

Exemplo

- Certo, funcionou para $T(1) = 1$.
- Mas e se por exemplo $T(1) = 8$?

  Então $T(2) = 8 + 8 + 2 = 18$ e $3.2.\lg 2 = 6$.
  Não deu certo...
- Certo, mas aí basta escolher uma constante maior.
  Mostra-se do mesmo jeito que $T(n) \leq 10n \lg n$ e para esta escolha $T(2) = 18 \leq 10.2.\lg 2 = 20$.
- De modo geral, se o passo de indução funciona ($T(n) \leq cn \lg n$), é possível escolher c e a base da indução (n_0) de modo conveniente!

Como achar as constantes?

- Tudo bem. Dá até para chutar que $T(n)$ pertence a classe $O(n \lg n)$.
- Mas como descobrir que $T(n) \leq 3n \lg n$? Como achar a constante 3?
- Eis um método simples: suponha como hipótese de indução que $T(n) \leq cn \lg n$ para $n \geq n_0$ onde c e n_0 são constantes que vou tentar determinar.

Primeira tentativa

$$
\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\
&\leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\leq c \left\lceil \frac{n}{2} \right\rceil \lg n + c \left\lfloor \frac{n}{2} \right\rfloor \lg n + n \\
&= c \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n + n \\
&= cn \lg n + n
\end{aligned}
$$

(Hummm, não deu certo. . .)

Segunda tentativa

$$
\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \\
&\leq c \left\lceil \frac{n}{2} \right\rceil \lg \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\leq c \left\lceil \frac{n}{2} \right\rceil \lg n + c \left\lfloor \frac{n}{2} \right\rfloor (\lg n - 1) + n \\
&= c \left(\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor \right) \lg n - c \left\lfloor \frac{n}{2} \right\rfloor + n \\
&= cn \lg n - c \left\lfloor \frac{n}{2} \right\rfloor + n \\
&\leq cn \lg n.
\end{aligned}
$$

Para garantir a última desigualdade basta que $-c\lceil n/2 \rceil + n \leq 0$ e $c = 3$ funciona. (Yeeeeeeesssssss!)

Completando o exemplo

Mostramos que a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

satisfaz $T(n) \in O(n \lg n)$.

Mas quem garante que $T(n)$ não é "menor"?

O melhor é mostrar que $T(n) \in \Theta(n \lg n)$.

Resta então mostrar que $T(n) \in \Omega(n \lg n)$. A prova é similar. (Exercício!)

Como chutar?

Não há nenhuma receita genérica para adivinhar soluções de recorrências. A experiência é o fator mais importante.

Felizmente, há várias idéias que podem ajudar.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \quad \text{para } n \geq 2. \end{aligned}$$

Ela é quase idêntica à anterior e podemos chutar que $T(n) \in \Theta(n \lg n)$. Isto de fato é verdade. (Exercício ou consulte o CLRS)

Como chutar?

Considere agora a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor + 17) + n \quad \text{para } n \geq 2. \end{aligned}$$

Ela parece bem mais difícil por causa do "17" no lado direito.

Intuitivamente, porém, isto não deveria afetar a solução. Para n **grande** a diferença entre $T(\lfloor n/2 \rfloor)$ e $T(\lfloor n/2 \rfloor + 17)$ não é tanta.

Chuto então que $T(n) \in \Theta(n \lg n)$. (Exercício!)

Truques e sutilezas

Algumas vezes adivinhamos corretamente a solução de uma recorrência, mas as contas aparentemente não funcionam! Em geral, o que é necessário é fortalecer a hipótese de indução.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \quad \text{para } n \geq 2. \end{aligned}$$

Chutamos que $T(n) \in O(n)$ e tentamos mostrar que $T(n) \leq cn$ para alguma constante c.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \\ &\leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 \\ &= cn + 1. \end{aligned}$$

(Humm, falhou...)

E agora? Será que erramos o chute? Será que $T(n) \in \Theta(n^2)$?

Truques e sutilezas

Na verdade, adivinhamos corretamente. Para provar isso, é preciso usar uma hipótese de indução mais forte.

Vamos mostrar que $T(n) \leq cn - b$ onde $b > 0$ é uma constante.

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1 \\ &\leq c\lceil n/2 \rceil - b + c\lfloor n/2 \rfloor - b + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

onde a última desigualdade vale se $b \geq 1$.
(Yeeeessss!)

Método da iteração

- Não é necessário adivinhar a resposta!
- Precisa fazer mais contas!
- Idéia: expandir (iterar) a recorrência e escrevê-la como uma somatória de termos que dependem apenas de n e das condições iniciais.
- Precisa conhecer limitantes para várias somatórias.

Método da iteração

Considere a recorrência

$$\begin{aligned} T(n) &= b && \text{para } n \leq 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + n && \text{para } n \geq 4. \end{aligned}$$

Iterando a recorrência obtemos

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) \\ &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor). \end{aligned}$$

Certo, mas quando devo parar?
O i-ésimo termo da série é $3^i \lfloor n/4^i \rfloor$. Ela termina quando $\lfloor n/4^i \rfloor \leq 3$, ou seja, $i \geq \log_4 n$.

Método da iteração

Como $\lfloor n/4^i \rfloor \leq n/4^i$ temos que

$$
\begin{aligned}
T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + 3^j b \\
T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \cdots + d.3^{\log_4 n} \\
&\leq n.(1 + 3/4 + 9/16 + 27/64 + \cdots) + dn^{\log_4 3} \\
&= n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + dn^{\log_4 3} \\
&= 4n + dn^{\log_4 3}
\end{aligned}
$$

pois $3^{\log_4 n} = n^{\log_4 3}$ e $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$ para $0 < q < 1$.

Como $\log_4 3 < 1$ segue que $n^{\log_4 3} \in o(n)$ e logo, $T(n) \in O(n)$.

Método de iteração

- As contas ficam mais simples se supormos que a recorrência está definida apenas para potências de um número, por exemplo, $n = 4^i$.
- Note, entretanto, que a recorrência deve ser provada para todo natural suficientemente grande.
- Muitas vezes, é possível depois de iterar a recorrência, adivinhar a solução e usar o método da substituição!

Método de iteração

$$\begin{aligned} T(n) &= b && \text{para } n \leq 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + n && \text{para } n \geq 4. \end{aligned}$$

Chuto que $T(n) \leq cn$.

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + n \\ &\leq 3c\lfloor n/4 \rfloor + n \\ &\leq 3c(n/4) + n \\ &\leq cn \end{aligned}$$

onde a última desigualdade vale se $c \geq 4$.
(Yeeessss!)

Resolução pelo método da iteração

- A ideia da resolução pelo **método da iteração** (ou **expansão telescópica**) é expandir a relação de recorrência até que possa ser detectado seu comportamento no caso geral.
- Passos para resolver um equação de recorrência:
  1. Copie a fórmula original
  2. Descubra o passo (se $T(n)$ estiver escrito em função de $T(n/2)$, a cada passo o parâmetro é dividido por 2)
  3. Isole as equações para "os próximos passos"
  4. Substitua os valores isolados na fórmula original
  5. Identifique a fórmula do i-ésimo passo
  6. Descubra o valor de i de forma a igualar o parâmetro de $T(x)$ ao parâmetro (valor de n) no caso base
  7. Substitua o valor de i na fórmula do i-ésimo caso
  8. Identifique a complexidade dessa fórmula
  9. Prove por indução que a equação foi corretamente encontrada

Resolução por expansão telescópica

Exemplo 1: $\boxed{T(n) = 2T(n/2) + 2}$ $\boxed{T(1) = 1}$

1. $T(n) = 2T(n/2) + 2$ *(fórmula original)*
2. $T(n)$ está escrito em função de $T(n/2)$
3. Isole as equações para $T(n/2)$ e $T(n/4)$:
   $$T(n/2) = 2(T(n/4)) + 2$$
   $$T(n/4) = 2(T(n/8)) + 2$$
4. Substitua $T(n/2)$ pelo valor que foi isolado acima e, em seguida, o mesmo para $T(n/4)$
   - *substituindo o valor isolado de* $T(n/2)$:
     $T(n) = 2(2(T(n/4)) + 2) + 2$
     $T(n) = 2^2 T(n/2^2) + 6$
   - *agora substituindo o valor de* $T(n/4)$:
     $T(n) = 2^2(2(T(n/8) + 2) + 6$
     $T(n) = 2^3 T(n/2^3) + 2^3 + 6$
     $T(n) = 2^3 T(n/2^3) + 2^4 - 2$

Resolução por expansão telescópica

Exemplo 1: $\boxed{T(n) = 2T(n/2) + 2}$ $\boxed{T(1) = 1}$

5. Identifique a fóruma do i-ésimo passo
   $$T(n) = 2^i T(n/2^i) + 2^{i+1} - 2$$
6. Descubra o valor de i de forma a igualar o parâmetro de $T(x)$ ao parâmetro (valor de n) no caso base
   $$T(n/2^i) \quad \Leftrightarrow \quad T(1)$$
   $$n/2^i = 1$$
   $$n = 2^i$$
   $$i = \lg(n)$$
7. Substitua o valor de i na fórmula do i-ésimo caso
   $$T(n) = 2^{\lg(n)} T(1) + 2^{\lg(n)+1} - 2$$
   $$T(n) = n + 2n - 2$$
   $$T(n) = 3n - 2$$
8. Identifique a complexidade dessa fórmula
   $$\boxed{T(n) \in \Theta(n)}$$

Resolução por expansão telescópica

Exemplo 1: $\boxed{T(n) = 2T(n/2) + 2}$ $\boxed{T(1) = 1}$

9. Prova por indução
   - **Passo base:** para $n = 1$, o resultado esperado é 1
     $T(n) = 3n - 2 \;=\; 3 - 2 \;=\; 1$ *(correto)*
   - **Passo indutivo:** por hipótese de indução, assumimos que a fórmula está correta para $n/2$, isto é, $T(n/2) = 3\,n/2 - 2$. Então, temos que verificar se $T(n) = 3n - 2$, sabendo-se que $T(n) = 2T(n/2) + 2$ e partindo da *H.I.* que
     $T(n/2) = 3n/2 - 2$
     $T(n) = 2\,T(n/2) + 2$
     $T(n) = 2\,(3n/2 - 2) + 2$
     $T(n) = 2 \cdot 3 \cdot n/2 - 2 \cdot 2 + 2$
     $T(n) = 3n - 4 + 2$
     $T(n) = 3n - 2$ *(passo indutivo provado)*
   - Demonstrado que $2T(n/2) + 2 = 3n - 2$ para $n \geq 1$

Resolução por expansão telescópica

Exemplo 2: $\boxed{T(n) = 2T(n-1) + 1}$ $\boxed{T(1) = 1}$

(Torre de Hanoi)

1. $T(n) = 2T(n-1) + 1$ *(fórmula original)*
2. $T(n)$ está escrito em função de $T(n-1)$
3. Isole as equações para $T(n-1)$ e $T(n-2)$:
   $$T(n-1) = 2T(n-2) + 1$$
   $$T(n-2) = 2T(n-3) + 1$$
4. Substitua $T(n-1)$ pelo valor que foi isolado acima e, em seguida, o mesmo para $T(n-2)$
   - *substituindo o valor isolado de* $T(n-1)$:
     $T(n) = 2(2T(n-2) + 1) + 1$
   - *agora substituindo o valor de* $T(n-2)$:
     $T(n) = 2^2 T(n-2) + 2 + 1$
     $T(n) = 2^2(2T(n-3) + 1) + 2 + 1$
     $T(n) = 2^3 T(n-3) + 2^2 + 2 + 1$
     $T(n) = 2^3 T(n-3) + 2^3 - 1$

Resolução por expansão telescópica

Exemplo 2: $\boxed{T(n) = 2T(n-1) + 1}$ $\boxed{T(1) = 1}$

5. Identifique a fóruma do i-ésimo passo

$$T(n) = 2^i T(n-1) + 2^i - 1$$

6. Descubra o valor de i de forma a igualar o parâmetro de $T(x)$ ao parâmetro (valor de n) no caso base

$$T(n-i) \quad \Leftrightarrow \quad T(1)$$
$$n - i = 1$$
$$i = n - 1$$

7. Substitua o valor de i na fórmula do i-ésimo caso

$$T(n) = 2^{n-1} T(1) + 2^{n-1} - 1$$
$$T(n) = 2^{n-1} + 2^{n-1} - 1$$
$$T(n) = 2 \cdot 2^{n-1} - 1$$
$$T(n) = 2^n - 1$$

8. Identifique a complexidade dessa fórmula

$$\boxed{T(n) \in \Theta(2^n)}$$

Resolução por expansão telescópica

Exemplo 2: $\boxed{T(n) = 2T(n-1) + 1}$ $\boxed{T(1) = 1}$

9. Prova por indução
   - **Passo base:** para $n = 1$, o resultado esperado é 1
     $T(n) = 2^n - 1 \quad = \quad 2 - 1 \quad = \quad 1$ *(correto)*
   - **Passo indutivo:** por hipótese de indução, assumimos que a fórmula está correta para $n-1$, isto é, $T(n-1) = 2^{n-1} - 1$. Então, temos que verificar se $T(n) = 2^n - 1$, sabendo-se que $T(n) = 2^n - 1$ e partindo da *H.I.* que $T(n-1) = 2^{n-1} - 1$
     $T(n) = 2\ T(n-1) + 1$
     $T(n) = 2\ (2^{n-1} - 1) + 1$
     $T(n) = 2^n - 2 + 1$
     $T(n) = 3^n - 1$ *(passo indutivo provado)*
   - Demonstrado que $2T(n-1) + 1 = 2^n - 1$ para $n \geq 1$

Resolução por expansão telescópica

- **Exercícios** – Repita o procedimento para as seguintes equações de recorrência:

  1. $T(n) = 3T(n-1) + 1$
     $T(1) = 1$

  2. $T(n) = 4T(n/2) + n$
     $T(1) = 1$

Árvore de recorrência

- Permite visualizar melhor o que acontece quando a recorrência é iterada.
- É mais fácil organizar as contas.
- Útil para recorrências de algoritmos de divisão-e-conquista.

Árvore de recorrência

Considere a recorrência

$$\begin{aligned} T(n) &= \Theta(1) && \text{para } n = 1, 2, 3, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 && \text{para } n \geq 4, \end{aligned}$$

onde $c > 0$ é uma constante.

Costuma-se (CLRS) usar a notação $T(n) = \Theta(1)$ para indicar que $T(n)$ é uma constante.

Árvore de recorrência

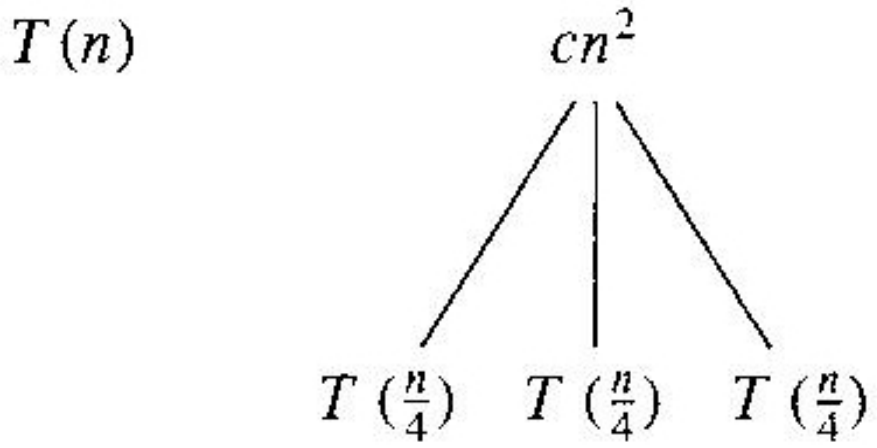
Simplificação

Vamos supor que a recorrência está definida apenas para potências de 4

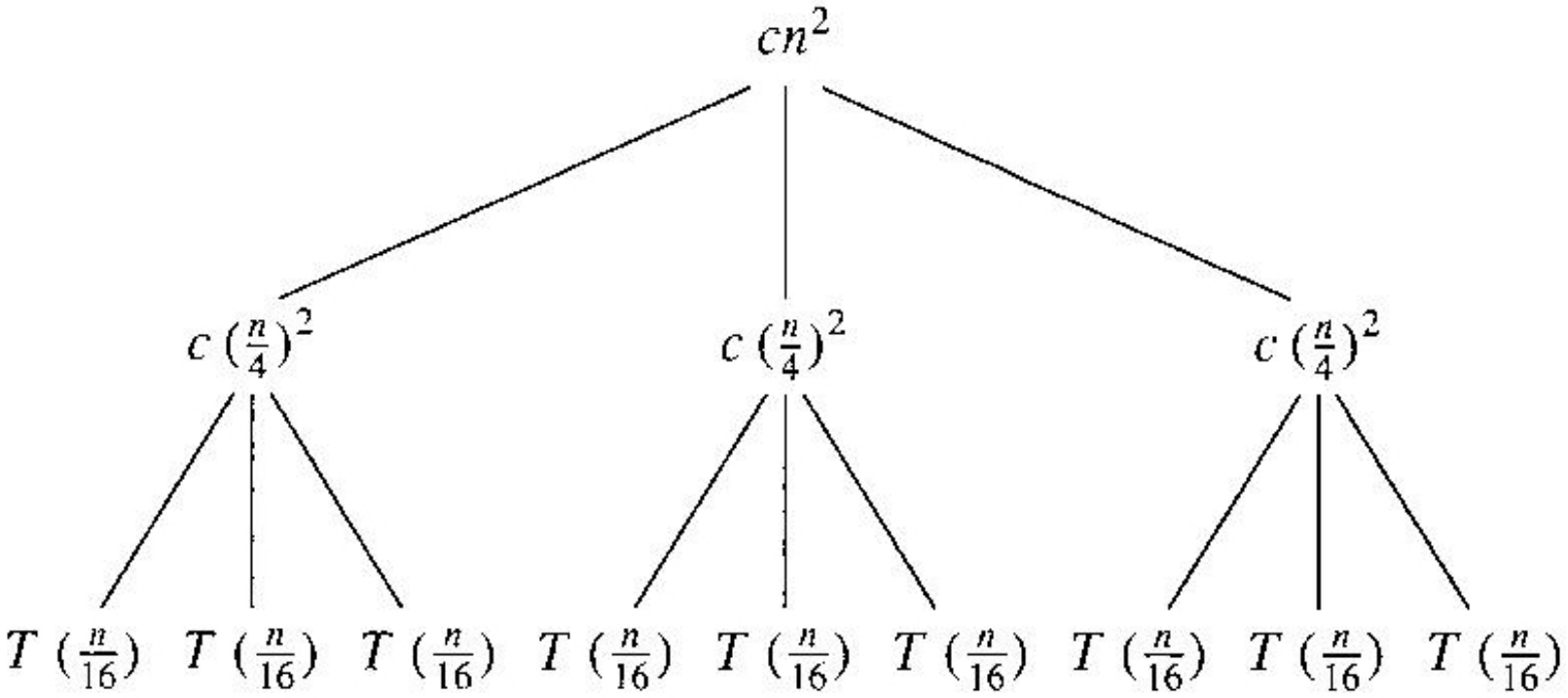
$$\begin{aligned} T(n) &= \Theta(1) && \text{para } n = 1, \\ T(n) &= 3T(n/4) + cn^2 && \text{para } n = 4, 16, \ldots, 4^i, \ldots. \end{aligned}$$

Isto permite descobrir mais facilmente a solução. Depois usamos o método da substituição para formalizar.

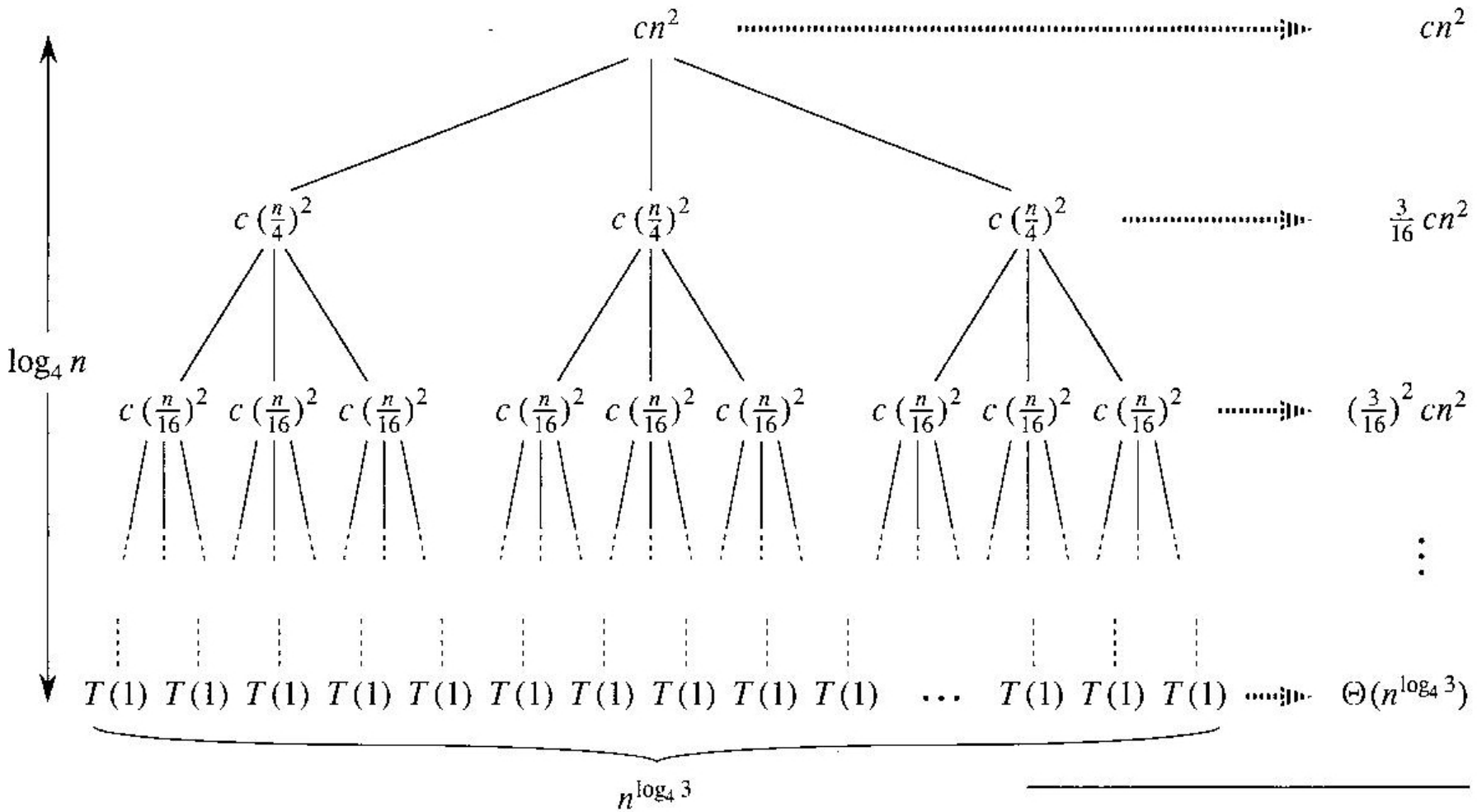
Árvore de recorrência



Árvore de recorrência



Árvore de recorrência



Total: $O(n^2)$

Árvore de recorrência

- O número de níveis é $\log_4 n + 1$.
- No nível i o tempo gasto (sem contar as chamadas recursivas) é $(3/16)^i cn^2$.
- No último nível há $3^{\log_4 n} = n^{\log_4 3}$ folhas. Como $T(1) = \Theta(1)$ o tempo gasto é $\Theta(n^{\log_4 3})$.

Árvore de recorrência

Logo,

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \cdots + \\
&\quad + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\
&\leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) = \frac{3}{16}cn^2 + \Theta(n^{\log_4 3}),
\end{aligned}
$$

e $T(n) \in O(n^2)$.

Árvore de recorrência

Mas $T(n) \in O(n^2)$ é realmente a solução da recorrência original?

Com base na árvore de recorrência, chutamos que $T(n) \leq dn^2$ para alguma constante $d > 0$.

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

onde a última desigualdade vale se $d \geq (16/13)c$.
(Yeeesssss!)

Árvore de recorrência

Resumo

- O número de nós em cada nível da árvore é o número de chamadas recursivas.
- Em cada nó indicamos o "tempo" ou "trabalho" gasto naquele nó que **não** corresponde a chamadas recursivas.
- Na coluna mais à direita indicamos o tempo total naquele nível que **não** corresponde a chamadas recursivas.
- Somando ao longo da coluna determina-se a solução da recorrência.

Vamos tentar juntos?

Eis um exemplo um pouco mais complicado.

Vamos resolver a recorrência

$$\begin{aligned} T(n) &= 1 && \text{para } n = 1, 2, \\ T(n) &= T(\lceil n/3 \rceil) + T(\lfloor 2n/3 \rfloor) + n && \text{para } n \geq 3. \end{aligned}$$

Qual é a solução da recorrência?

Resposta: $T(n) \in O(n \lg n)$. (Resolvido em aula)

Recorrências com O à direita (CLRS)

Uma "recorrência"

$$T(n) = \Theta(1) \qquad \text{para } n = 1, 2,$$
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \qquad \text{para } n \geq 3$$

representa todas as recorrências da forma

$$T(n) = a \qquad \text{para } n = 1, 2,$$
$$T(n) = 3T(\lfloor n/4 \rfloor) + bn^2 \qquad \text{para } n \geq 3$$

onde a e $b > 0$ são constantes.

As soluções exatas dependem dos valores de a e b, mas estão todas na mesma classe Θ.

A "solução" é $T(n) = \Theta(n^2)$, ou seja, $T(n) \in \Theta(n^2)$.

As mesmas observações valem para as classes O, Ω, o, ω.

Recorrência do Mergesort

Podemos escrever a recorrência de tempo do Mergesort da seguinte forma

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \qquad \text{para } n \geq 2. \end{aligned}$$

A solução da recorrência é $T(n) = \Theta(n \lg n)$.

A prova é essencialmente a mesma do primeiro exemplo. (Exercício!)

Cuidados com a notação assintótica

A notação assintótica é muito versátil e expressiva. Entretanto, deve-se tomar alguns cuidados.

Considere a recorrência

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\lfloor n/2 \rfloor) + n \qquad \text{para } n \geq 2. \end{aligned}$$

É similar a recorrência do Mergesort!

Mas eu vou "provar" que $T(n) = O(n)$!

Cuidados com a notação assintótica

Vou mostrar que $T(n) \leq cn$ para alguma constante $c > 0$.

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2c\lfloor n/2 \rfloor + n \\ &\leq cn + n \\ &= O(n) \qquad \Longleftarrow \textbf{ERRADO!!!} \end{aligned}$$

Por quê?

Não foi feito o passo indutivo, ou seja, não foi mostrado que $T(n) \leq cn$.

Teorema Master

- Veremos agora um resultado que descreve soluções para recorrências da forma

$$T(n) = aT(n/b) + f(n),$$

onde $a \geq 1$ e $b > 1$ são constantes.

- O caso base é omitido na definição e convenciona-se que é uma constante para valores pequenos.
- A expressão n/b pode indicar tanto $\lfloor n/b \rfloor$ quanto $\lceil n/b \rceil$.
- O Teorema Master **não** fornece a resposta para todas as recorrências da forma acima.

Teorema Master (Manber)

Teorema (**Teorema Master (Manber)**)

Dada uma relação de recorrência da forma

$$T(n) = aT(n/b) + cn^k,$$

onde $a, b \in \mathbb{N}$, $a \geq 1$, $b \geq 2$, $c > 0$ *e* $k \geq 0$ *são constantes,*

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^k \\ \Theta(n^k \log n), & \text{se } a = b^k \\ \Theta(n^k), & \text{se } a < b^k \end{cases}$$

Teorema Master (Manber)

Prova: Por simplicidade, assumimos que $n = b^m$ de modo que n/b é sempre inteiro. Com isso temos

$$T(n) = aT(n/b) + cn^k$$

é equivalente a

$$T(n) = aT(b^{m-1}) + cb^{mk}$$

Vamos começar expandindo a relação de recorrência:

$$\begin{aligned}
T(n) &= aT(b^{m-1}) + cb^{mk} \\
&= a(aT(b^{m-2}) + cb^{(m-1)k}) + cb^{mk} \\
&= a^2T(b^{m-2}) + cab^{(m-1)k} + cb^{mk} \\
&= a^3T(b^{m-3}) + ca^2b^{(m-2)k} + cab^{(m-1)k} + cb^{mk} \\
&= \ldots \\
&= a^mT(b^0) + ca^{m-1}b^k + ca^{m-2}b^2k + \ldots + cab^{(m-1)k} + cb^{mk}
\end{aligned}$$

Teorema Master (Manber)

Assumindo que $T(1) = c$, ficamos com:

$$\begin{aligned} T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \ldots + cb^{mk} \\ &= c\sum_{i=0}^{m} a^{m-i}b^{ik} \\ &= ca^m \sum_{i=0}^{m} (b^k/a)^i. \end{aligned}$$

Na última linha podemos ver os casos do enunciado, com base em como séries geométricas se comprotam quando b^k/a é maior, menor ou igual a zero.

Teorema Master (Manber)

$$T(n) = ca^m \sum_{i=0}^{m} (b^k/a)^i.$$

Caso 1: $a > b^k$
Neste caso, o somatório $\sum_{i=0}^{m} (b^k/a)^i$ converge para uma constante. Daí, temos que $T(n) \in \Theta(ca^m)$. Como $n = b^m$, então $m = \log_b n$, consequentemente, $T(n) \in \Theta(n^{\log_b a})$.

Teorema Master (Manber)

$$T(n) = ca^m \sum_{i=0}^{m} (b^k/a)^i.$$

Caso 2: $a = b^k$
Como $b^k/a = 1$, temos $\sum_{i=0}^{m} (b^k/a)^i = m + 1$. Daí, temos que $T(n) \in \Theta(ca^m m)$. Como $m = \log_b n$ e $a = b^k$, então $ca^m m = cn^{\log_b a} \log_b n = cn^k \log_b n$, o que nos leva à conclusão que $T(n) \in \Theta(n^k \log_b n)$.

Teorema Master (Manber)

$$T(n) = ca^m \sum_{i=0}^{m} (b^k/a)^i.$$

Caso 3: $a < b^k$

Neste caso, a série não converge quando m vai para infinito, mas é possível calcular sua soma para um número finito de termos.

$$\begin{aligned} T(n) &= ca^m \sum_{i=0}^{m} (b^k/a)^i \\ &= ca^m \left(\frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1} \right). \end{aligned}$$

Desprezando as constantes na última linha da expressão acima e sabendo que $a^m \left(\frac{(b^k/a)^{m+1}-1}{(b^k/a)-1} \right) = b^{km}$ e $b^m = n$, concluímos que $T(n) \in \Theta(n^k)$. CQD

Teorema Master

Teorema (**Teorema Master (CLRS)**)

Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida para os inteiros não-negativos pela relação de recorrência

$$T(n) = aT(n/b) + f(n).$$

Então $T(n)$ pode ser limitada assintoticamente da seguinte maneira:

1. *Se $f(n) \in O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$*
2. *Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$*
3. *Se $f(n) \in \Omega(n^{\log_b a + \epsilon})$, para alguma constante $\epsilon > 0$ e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e para n suficientemente grande, então $T(n) \in \Theta(f(n))$*

Resolução por Teorema Master

Exemplo 1: $\boxed{T(n) = 4T\left(\frac{n}{2}\right) + n}$

- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$a = 4 \quad ; \quad b = 2$$
$$\log_b a \;=\; \log_2 4 \;=\; 2$$

$$f(n) \;=\; n$$
$$f(n) \;\in\; O(n^{\log_b a - \epsilon}) \;=\; O(n^{2-\epsilon}), \quad \text{sendo } \epsilon = 1 \ (\epsilon > 0)$$

- Portanto, se encaixa no caso 1 do Teorema Master:

$$T(n) \;\in\; \Theta(n^{\log_b a}) \;=\; \Theta(n^2)$$

Resolução por Teorema Master

Exemplo 2: $\boxed{T(n) = T\left(\frac{9n}{10}\right) + n}$

- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

  $a = 1 \quad ; \quad b = \frac{10}{9} \quad ; \quad \log_b a = \log_{\frac{10}{9}} 1 = 0$

  $f(n) = n$

  $f(n) \in \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{0+\epsilon}),$ sendo $\epsilon = 1$ $(\epsilon > 0)$

- Será caso 3 se satisfizer a condição de regularidade:
  Para todo n, $af\left(\frac{n}{b}\right) = \frac{9n}{10} \leq \frac{9}{10}n = cf(n)$ para
  $c = \frac{9}{10} < 1$.

- Portanto, se encaixa no caso 3 do Teorema Master:
  $T(n) \in \Theta(f(n)) = \Theta(n)$

Resolução por Teorema Master

Exemplo 3: $\boxed{T(n) = 4T\left(\frac{n}{2}\right) + n^2}$

- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

  $a = 4 \quad ; \quad b = 2 \quad ; \quad \log_b a = \log_2 4 = 2$

  $f(n) = n^2$

  $f(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$

- Portanto, se encaixa no caso 2 do Teorema Master:

  $T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(n^2 \log n)$

Exemplos de Recorrências

Exemplos onde o Teorema Master se aplica:

- Caso 1:
  $T(n) = 9T(n/3) + n$
  $T(n) = 4T(n/2) + n \log n$
- Caso 2:
  $T(n) = T(2n/3) + 1$
  $T(n) = 2T(n/2) + (n + \log n)$
- Caso 3:
  $T(n) = T(3n/4) + n \log n$

Exemplos de Recorrências

Exemplos onde o Teorema Master **não se aplica**:

- $T(n) = T(n-1) + n$
- $T(n) = T(n-a) + T(a) + n$, ($a \geq 1$ inteiro)
- $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, ($0 < \alpha < 1$)
- $T(n) = T(n-1) + \log n$
- $T(n) = 2T(\frac{n}{2}) + n \log n$

Agradecimentos

- Esses slides são fruto de um trabalho iniciado pelos Profs. Cid C. de Souza e Cândida N. da Silva. Desde então, várias partes foram motificadas, melhoradas ou colocadas ao gosto particular de cada docente. Em particular pelos Profs. Orlando Lee, Pedro J. de Rezende, Flávio K. Miyazawa e, mais recentemente, pelo Prof. Alexandre. G. Silva.
- Vários outros professores colaboraram direta ou indiretamente para a preparação do material desses slides. Agradecemos, especialmente (em ordem alfabética): Célia P. de Mello, José C. de Pina, Paulo Feofiloff, Ricardo Dahab e Zanoni Dias.
- Como foram feitas modificações de conteúdo teórico e filosófico, os erros/imprecisões que se encontram nesta versão devem ser comunicados a alexandre.goncalves.silva@ufsc.br.