Projeto e Análise de Algoritmos

A. G. Silva e R. de Santiago

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

05 de abril de 2019

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (12 horas/aula)
- Buscas (4 horas/aula)
- Grafos (4 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Busca Heurística (6 horas/aula)

Cronograma

- **15mar** – Apresentação da disciplina. Introdução.
- **22mar** – *Prova de proficiência/validação.*
- **29mar** – Notação assintótica. Recorrências.
- **05abr** – Recorrências. Divisão e conquista. Ordenação.
- **12abr** – Ordenação em tempo linear. Multiplicação de inteiros.
- **19abr** – *Dia não letivo.* Exercícios.
- **26abr** – Estatística de ordem.
- **03mai** – **Primeira avaliação.**
- **10mai** – Grafos. Buscas.
- **17mai** – Algoritmos gulosos.
- **24mai** – Algoritmos gulosos. Programação dinâmica.
- **31mai** – Programação dinâmica.
- **07jun** – **Segunda avaliação.**
- **14jun** – *Semana Acadêmica.* NP-Completude e reduções.
- **21jun** – *Dia não letivo.* Exercícios.
- **28jun** – **Avaliação substitutiva** *(opcional)*

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)
    se $p < r$
        então $q \leftarrow \lfloor (p+r)/2 \rfloor$
            MERGESORT(A, p, q)
            MERGESORT($A, q+1, r$)
            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

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

- 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.

Resolução pelo método da **substituição**

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:

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

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

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

(Yeeeeeeesssss!)

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

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

(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. (Yeeeeeeeesssssss!)

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

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

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

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

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!)

Resolução pelo método da **iteração**

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

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

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 pelo método da iteração

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* $T(n/2)$:
     $T(n) = 2(2(T(n/4)) + 2) + 2$
     $T(n) = 2^2T(n/2^2) + 2^2 + 2$
   - *agora substituindo o valor de* $T(n/4)$:
     $T(n) = 2^2(2(T(n/8) + 2) + 2^2 + 2$
     $T(n) = 2^3T(n/2^3) + 2^3 + 2^2 + 2$

Resolução pelo método da iteração

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

5. Identifique a fóruma do i-ésimo passo

$$T(n) = 2^i T(n/2^i) + \underbrace{2^i + 2^{i-1} + \cdots + 2^2 + 2}_{\text{Soma de PG: } S_n = \frac{a_1(q^n-1)}{q-1}}$$

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

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 pelo método da iteração

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
     $T(n) = 3n - 2 \quad = \quad 3 - 2 \quad = \quad 1$ *(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 pelo método da iteração

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

(Torres 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* $T(n-2)$:
     $T(n) = 2^2T(n-2) + 2 + 1$
     $T(n) = 2^2(2T(n-3) + 1) + 2 + 1$
     $T(n) = 2^3T(n-3) + 2^2 + 2 + 1$

Resolução pelo método da iteração

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

5. Identifique a fóruma do *i*-ésimo passo

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

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 pelo método da iteração

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
     $T(n) = 2^n - 1 \quad = \quad 2 - 1 \quad = \quad 1 \qquad$ *(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 $T(n-1) = 2^{n-1} - 1$
     $T(n) = 2\ T(n-1) + 1$
     $T(n) = 2\ (2^{n-1} - 1) + 1$
     $T(n) = 2^n - 2 + 1$
     $T(n) = 3^n - 1 \qquad$ *(passo indutivo provado)*

   - Demonstrado que $2T(n-1) + 1 = 2^n - 1$ para $n \geq 1$

Resolução pelo método da iteração

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

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

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

Resolução pelo método da **árvore de recorrência**

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.

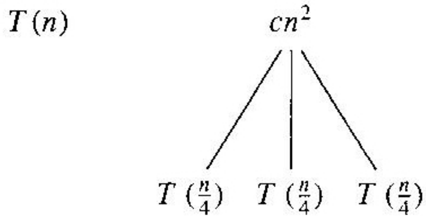
Simplificação

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

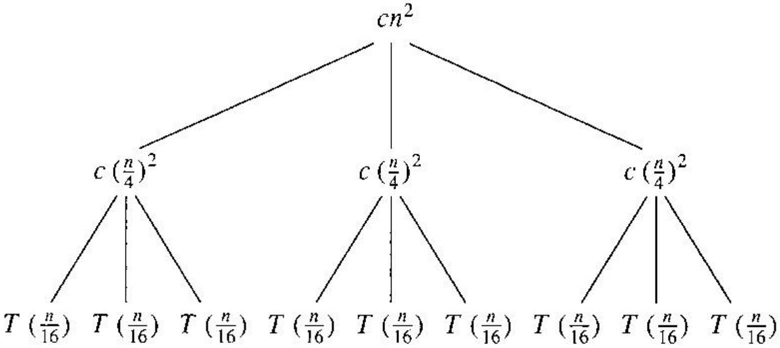
$$
\begin{aligned}
T(n) &= \Theta(1) && \text{para } n = 1, \\
T(n) &= 3T(n/4) + cn^2 && \text{para } n = 4, 16, \ldots, 4^i, \ldots.
\end{aligned}
$$

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

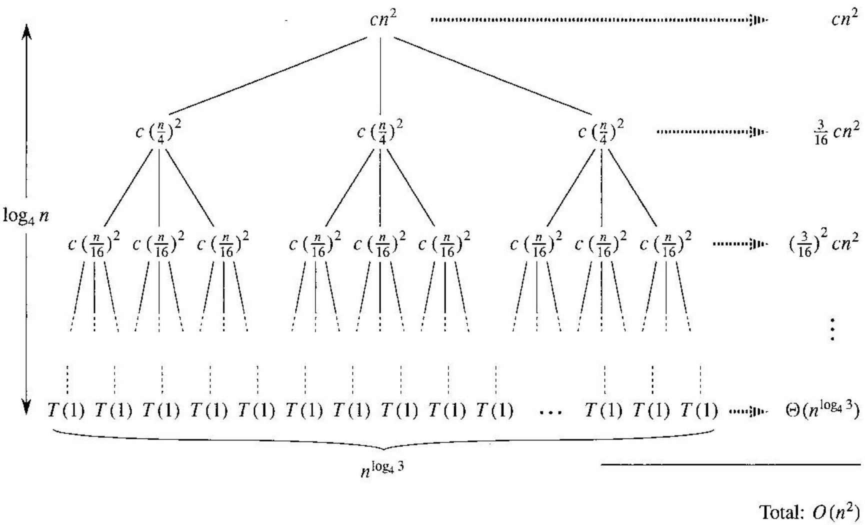
Árvore de recorrência



Árvore de recorrência



Árvore de recorrência



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,

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \cdots + \\
&\quad + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\
&\leq cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}),
\end{aligned}
$$

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”

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

representa todas as recorrências da forma

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

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

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

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

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

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 $T(n) \leq cn$.

Resolução pelo **teorema master**

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,*

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

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 $a^m \left(\frac{(b^k/a)^{m+1}}{(b^k/a)} \right) = b^{km}$ 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: $\boxed{T(n) = 4T\left(\frac{n}{2}\right) + n}$

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

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

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

  $f(n) \;=\; n$

  $f(n) \;\in\; O(n^{\log_b a - \epsilon}) \;=\; O(n^{2-\epsilon}),$ 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: $\boxed{T(n) = 4T\left(\frac{n}{2}\right) + n^2}$

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

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

  $$\begin{aligned} f(n) &= n^2 \\ f(n) &\in \Theta(n^{\log_b a}) = \Theta(n^2) \end{aligned}$$

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

Divisão e Conquista

Projeto de Algoritmos por Divisão e Conquista

- **Dividir para conquistar:** uma tática de guerra aplicada ao projeto de algoritmos.
- Um algoritmo de divisão e conquista é aquele que resolve o problema desejado combinando as soluções parciais de (um ou mais) subproblemas, obtidas recursivamente.
- É mais um paradigma de projeto de algoritmos baseado no princípio da indução.
- Informalmente, podemos dizer que o paradigma incremental representa o projeto de algoritmos por indução fraca, enquanto o paradigma de divisão e conquista representa o projeto por indução forte.
- É natural, portanto, demonstrar a corretude de algoritmos de divisão e conquista por indução.

Algoritmo Genérico

DivisaoConquista(x)

▷ **Entrada:** A instância x
▷ **Saída:** Solução y do problema em questão para x

1. **se** x é suficientemente pequeno **então**
   ▷ *Solucao*(x) algoritmo para pequenas instâncias
2. **retorne** *Solucao*(x)
3. **senão**
   ▷ divisão
4. decomponha x em instâncias menores $x_1, x_2, \cdots, x_k$
5. **para** i **de** 1 **até** k **faça** $y_i := \mathit{DivisaoConquista}(x_i)$
   ▷ conquista
6. combine as soluções y_i para obter a solução y de x.
7. **retorne**(y)

Projeto por Divisão e Conquista - Exemplo 1
Exponenciação

Problema:

Calcular a^n, para todo real a e inteiro $n \geq 0$.

Primeira solução, por indução fraca:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^{n-1}.*
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução, sei calcular a^{n-1}. Então, calculo a^n multiplicando a^{n-1} por a.

Exemplo 1 - Solução 1 - Algoritmo

Exponenciacao(a, n)

$\triangleright$ **Entrada:** A base a e o expoente n.
$\triangleright$ **Saída:** O valor de a^n.
1. **se** $n = 0$ **então**
2.   **retorne**(1) {caso base}
3. **senão**
4. $an' := Exponenciacao(a, n-1)$
5. $an := an' * a$
6. **retorne**(an)

Exemplo 1 - Solução 1 - Complexidade

Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular a^n.

Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n-1) + c_2, & n > 0, \end{cases}$$

onde c_1 e c_2 representam, respectivamente, o tempo (constante) executado na atribuição da base e multiplicação do passo.

Neste caso, não é difícil ver que

$$T(n) = c_1 + \sum_{i=1}^{n} c_2 = c_1 + nc_2 = \Theta(n).$$

Este algoritmo é linear no tamanho da entrada ?

Exemplo 1 - Solução 2 - Divisão e Conquista

Vamos agora projetar um algoritmo para o problema usando indução forte de forma a obter um algoritmo de divisão e conquista.

Segunda solução, por indução forte:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^k, para todo $k < n$.*
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução sei calcular $a^{\lfloor \frac{n}{2} \rfloor}$. Então, calculo a^n da seguinte forma:

$$a^n = \begin{cases} \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ par;} \\ a \cdot \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ ímpar.} \end{cases}$$

Exemplo 1 - Solução 2 - Algoritmo

ExponenciacaoDC(a, n)

▷ **Entrada:** A base a e o expoente n.

▷ **Saída:** O valor de a^n.

1. **se** $n = 0$ **então**
2. **retorne**(1) {caso base}
3. **senão**

   ▷ divisão
4. $an' := ExponenciacaoDC(a, n\ div\ 2)$

   ▷ conquista
5. $an := an' * an'$
6. **se** $(n\ mod\ 2) = 1$
7. $an := an * a$
8. **retorne**(an)

Exemplo 1 - Solução 2 - Complexidade

- Seja $T(n)$ o número de operações executadas pelo algoritmo de divisão e conquista para calcular a^n.
- Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & n > 0, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?

Projeto por Divisão e Conquista - Exemplo 2
Busca Binária

Problema:

Dado um vetor ordenado A com n números reais e um real x, determinar a posição $1 \leq i \leq n$ tal que $A[i] = x$, ou que não existe tal i.

- O projeto de um algoritmo para este problema usando indução simples, nos leva a um algoritmo incremental de complexidade de pior caso $\Theta(n)$. Pense em como seria a indução !
- Se utilizarmos indução forte para projetar o algoritmo, podemos obter um algoritmo de divisão e conquista que nos leva ao algoritmo de busca binária. Pense na indução !
- Como o vetor está ordenado, conseguimos determinar, com apenas uma comparação, que *metade* das posições do vetor não pode conter o valor x.

Exemplo 2 - Algoritmo

BuscaBinaria(A, e, d, x)

▷ **Entrada:** Vetor A, delimitadores e e d do subvetor e x.
▷ **Saída:** Índice $1 \leq i \leq n$ tal que $A[i] = x$ ou $i = 0$.

se $e = d$ **então se** $A[e] = x$ **então retorne**(e)
        senão retorne(0)
senão
    $i := (e + d)\ div\ 2$
    se $A[i] = x$ **retorne**(i)
    senão se $A[i] > x$
        $i := BuscaBinaria(A, e, i - 1, x)$
        senão $\{A[i] < x\}$
        $i := BuscaBinaria(A, i + 1, d, x)$
retorne(i)

Exemplo 2 - Complexidade

- O número de operações $T(n)$ executadas na busca binária no pior caso é:

$$T(n) = \begin{cases} c_1, & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + c_2, & n > 1, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?
- O algoritmo de busca binária (divisão e conquista) tem complexidade de pior caso $\Theta(\log n)$, que é assintoticamente melhor que o algoritmo de busca linear (incremental).
- E se o vetor não estivesse ordenado, qual paradigma nos levaria a um algoritmo assintoticamente melhor ?

Projeto por Divisão e Conquista - Exemplo 3 - Máximo e Mínimo

Problema:

Dado um conjunto S de $n \geq 2$ números reais, determinar o maior e o menor elemento de S.

- Um algoritmo incremental para esse problema faz $2n - 3$ comparações: fazemos uma comparação no caso base e duas no passo.
- Será que um algoritmo de divisão e conquista seria melhor ?
- Um possível algoritmo de divisão e conquista seria:
  - Divida S em dois subconjuntos de mesmo tamanho S_1 e S_2 e solucione os subproblemas.
  - O máximo de S é o máximo dos máximos de S_1 e S_2 e o mínimo de S é o mínimo dos mínimos de S_1 e S_2.

Exemplo 3 - Complexidade

- Qual o número de comparações $T(n)$ efetuado por este algoritmo?

$$T(n) = \begin{cases} 1, & n = 2 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2, & n > 2, \end{cases}$$

- Supondo que n é uma potência de 2, temos:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(\frac{n}{2}) + 2, & n > 2, \end{cases}$$

- Neste caso, podemos provar que $T(n) = \frac{3}{2}n - 2$ usando o método da substituição (indução !).

Exemplo 3 - Complexidade

- **Caso Base:** $T(2) = 1 = 3 - 2$.
- **Hipótese de Indução:** Suponha, para $n = 2^{k-1}$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.
- **Passo de Indução:** Queremos provar para $n = 2^k$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 2 \\ &= 2(\tfrac{3}{4}n - 2) + 2 \text{ (por h. i.)} \\ &= \tfrac{3}{2}n - 2. \end{aligned}$$

- É possível provar que $T(n) = \frac{3}{2}n - 2$ quando n não é potência de 2 ?

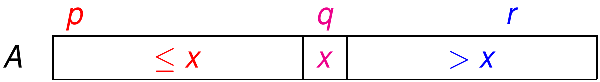
Exemplo 3 - Complexidade

- Assintoticamente, os dois algoritmos para este problema são equivalentes, ambos $\Theta(n)$.
- No entanto, o algoritmo de divisão e conquista permite que menos comparações sejam feitas. A estrutura hierárquica de comparações no retorno da recursão evita comparações desnecessárias.

QuickSort

O algoritmo QUICKSORT segue o paradigma de divisão-e-conquista.

Divisão: divida o vetor em dois subvetores $A[p \dots q-1]$ e $A[q+1 \dots r]$ tais que



$$A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$$

Conquista: ordene os dois subvetores recursivamente usando o QUICKSORT;

Combinação: nada a fazer, o vetor está ordenado.

Partição

Problema: Rearranjar um dado vetor $A[p \ldots r]$ e devolver um índice q, $p \leq q \leq r$, tais que

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r]$$

Entrada:

	p									r
A	99	33	55	77	11	22	88	66	33	44

Saída:

	p				q					r
A	33	11	22	33	44	55	99	66	77	88

Particione

	p									r
A	99	33	55	77	11	22	88	66	33	44
i	j									x
A	99	33	55	77	11	22	88	66	33	44
i		j								x
A	99	33	55	77	11	22	88	66	33	44
	i		j							x
A	33	99	55	77	11	22	88	66	33	44
	i			j						x
A	33	99	55	77	11	22	88	66	33	44
	i				j					x
A	33	99	55	77	11	22	88	66	33	44
		i				j				x
A	33	11	55	77	99	22	88	66	33	44

Particione

		i				j				x
A	33	11	55	77	99	22	88	66	33	44
			i				j			x
A	33	11	22	77	99	55	88	66	33	44
			i					j		x
A	33	11	22	77	99	55	88	66	33	44
			i						j	x
A	33	11	22	77	99	55	88	66	33	44
				i						j
A	33	11	22	33	99	55	88	66	77	44
	p				q					r
A	33	11	22	33	44	55	88	66	77	99

Particione

Rearranja $A[p \dots r]$ de modo que $p \leq q \leq r$ e
$A[p \dots q{-}1] \leq A[q] < A[q{+}1 \dots r]$

```
PARTICIONE(A, p, r)
1   x ← A[r]   ▷ x é o “pivô”
2   i ← p−1
3   para j ← p até r − 1 faça
4       se A[j] ≤ x
5           então i ← i + 1
6                 A[i] ↔ A[j]
7   A[i+1] ↔ A[r]
8   devolva i + 1
```

Invariantes:

No começo de cada iteração da linha 3 vale que:
(1) $A[p \dots i] \leq x$ (2) $A[i{+}1 \dots j{-}1] > x$ (3) $A[r] = x$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)		Tempo
1	$x \leftarrow A[r]$ $\quad \triangleright$ x é o "pivô"	?
2	$i \leftarrow p-1$	?
3	**para** $j \leftarrow p$ **até** $r-1$ **faça**	?
4	**se** $A[j] \leq x$	?
5	**então** $i \leftarrow i+1$	?
6	$A[i] \leftrightarrow A[j]$	?
7	$A[i+1] \leftrightarrow A[r]$	?
8	**devolva** $i+1$	?

$T(n) =$ complexidade de tempo no pior caso sendo

$$n := r - p + 1$$

Complexidade de PARTICIONE

PARTICIONE(A, p, r)	Tempo
$x \leftarrow A[r]$ ▷ x é o "pivô"	$\Theta(1)$
$i \leftarrow p-1$	$\Theta(1)$
para $j \leftarrow p$ **até** $r - 1$ **faça**	$\Theta(n)$
se $A[j] \leq x$	$\Theta(n)$
então $i \leftarrow i + 1$	$O(n)$
$A[i] \leftrightarrow A[j]$	$O(n)$
$A[i+1] \leftrightarrow A[r]$	$\Theta(1)$
devolva $i + 1$	$\Theta(1)$

$$T(n) = \Theta(2n + 4) + O(2n) = \Theta(n)$$

Conclusão:
A complexidade de PARTICIONE é $\Theta(n)$.

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

QUICKSORT(A, p, r)
```
se p < r
    então q ← PARTICIONE(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```

	p									r
A	99	33	55	77	11	22	88	66	33	44

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

QUICKSORT(A, p, r)
1 **se** $p < r$
2 **então** $q \leftarrow$ PARTICIONE(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

	p				q					r
A	33	11	22	33	44	55	88	66	77	99

No começo da linha 3,

$$A[p \dots q{-}1] \leq A[q] < A[q{+}1 \dots r]$$

QuickSort

Rearranja um vetor $A[p \ldots r]$ em ordem crescente.

QUICKSORT(A, p, r)
se $p < r$
  então $q \leftarrow$ PARTICIONE(A, p, r)
    QUICKSORT$(A, p, q-1)$
    QUICKSORT$(A, q+1, r)$

	p				q					r
A	11	22	33	33	44	55	88	66	77	99

QuickSort

Rearranja um vetor $A[p \dots r]$ em ordem crescente.

$\text{QUICKSORT}(A, p, r)$

```
se p < r
    então q ← PARTICIONE(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```

	p				q					r
A	11	22	33	33	44	55	66	77	88	99

Complexidade de QUICKSORT

QUICKSORT(A, p, r)	Tempo
se $p < r$	?
então $q \leftarrow$ PARTICIONE(A, p, r)	?
QUICKSORT$(A, p, q-1)$	?
QUICKSORT$(A, q+1, r)$	?

$T(n) :=$ complexidade de tempo no pior caso sendo

$$n := r - p + 1$$

Complexidade de QUICKSORT

QUICKSORT(A, p, r)	Tempo
1 **se** $p < r$	$\Theta(1)$
2 **então** $q \leftarrow$ PARTICIONE(A, p, r)	$\Theta(n)$
3 QUICKSORT$(A, p, q-1)$	$T(k)$
4 QUICKSORT$(A, q+1, r)$	$T(n-k-1)$

$$T(n) = T(k) + T(n-k-1) + \Theta(n+1)$$

$$0 \le k := q - p \le n - 1$$

Recorrência

$T(n) :=$ consumo de tempo no pior caso

$$\begin{aligned} T(0) &= \Theta(1) \\ T(1) &= \Theta(1) \\ T(n) &= T(k) + T(n-k-1) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots \end{aligned}$$

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(???)$.

Recorrência grosseira:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

$T(n)$ é $\Theta(n^2)$.

Recorrência cuidadosa

$T(n)$:= complexidade de tempo no pior caso

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = \max_{0 \le k \le n-1} \{T(k) + T(n-k-1)\} + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

$$T(n) = \max_{0 \le k \le n-1}\{T(k) + T(n-k-1)\} + bn$$

Quero mostrar que $T(n) = \Theta(n^2)$.

Demonstração – $T(n) = O(n^2)$

Vou provar que $T(n) \leq cn^2$ para n grande.

$$
\begin{aligned}
T(n) &= \max_{0\leq k\leq n-1}\left\{T(k) + T(n-k-1)\right\} + bn \\
&\leq \max_{0\leq k\leq n-1}\left\{ck^2 + c(n-k-1)^2\right\} + bn \\
&= c\max_{0\leq k\leq n-1}\left\{k^2 + (n-k-1)^2\right\} + bn \\
&= c(n-1)^2 + bn \qquad \triangleright \text{ exercício} \\
&= cn^2 - 2cn + c + bn \\
&\leq cn^2\,,
\end{aligned}
$$

se $c > b/2$ e $n \geq c/(2c-b)$.

Continuação – $T(n) = \Omega(n^2)$

Agora vou provar que $T(n) \geq dn^2$ para n grande.

$$\begin{aligned}
T(n) &= \max_{0 \leq k \leq n-1} \left\{ T(k) + T(n-k-1) \right\} + bn \\
&\geq \max_{0 \leq k \leq n-1} \left\{ dk^2 + d(n-k-1)^2 \right\} + bn \\
&= d \max_{0 \leq k \leq n-1} \left\{ k^2 + (n-k-1)^2 \right\} + bn \\
&= d(n-1)^2 + bn \\
&= dn^2 - 2dn + d + bn \\
&\geq dn^2 \,,
\end{aligned}$$

se $d < b/2$ e $n \geq d/(2d-b)$.

Conclusão

$T(n)$ é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT no pior caso é $\Theta(n^2)$.

A complexidade de tempo do QUICKSORT é $O(n^2)$.

QuickSort no melhor caso

$M(n) :=$ complexidade de tempo no melhor caso

$$M(0) = \Theta(1)$$
$$M(1) = \Theta(1)$$
$$M(n) = \min_{0 \le k \le n-1} \{M(k) + M(n-k-1)\} + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

Mostre que, para $n \geq 1$,

$$M(n) \geq \frac{(n-1)}{2} \lg \frac{n-1}{2}.$$

Isto implica que no melhor caso o QUICKSORT é $\Omega(n \lg n)$.

Que é o mesmo que dizer que o QUICKSORT é $\Omega(n \lg n)$.

QuickSort no melhor caso

No melhor caso k é aproximadamente $(n-1)/2$.

$$R(n) = R(\left\lfloor \frac{n-1}{2} \right\rfloor) + R(\left\lceil \frac{n-1}{2} \right\rceil) + \Theta(n)$$

Solução: $R(n)$ é $\Theta(n \lg n)$.

Humm, lembra a recorrência do MERGESORT...

Mais algumas conclusões

$M(n)$ é $\Theta(n \lg n)$.

O consumo de tempo do QUICKSORT no melhor caso é $\Omega(n \log n)$.

Mais precisamente, a complexidade de tempo do QUICKSORT no melhor caso é $\Theta(n \log n)$.

Caso médio

Apesar da complexidade de tempo do QUICKSORT no pior caso ser $\Theta(n^2)$, na prática ele é o algoritmo mais eficiente.

Mais precisamente, a complexidade de tempo do QUICKSORT no caso médio é mais próximo do melhor caso do que do pior caso.

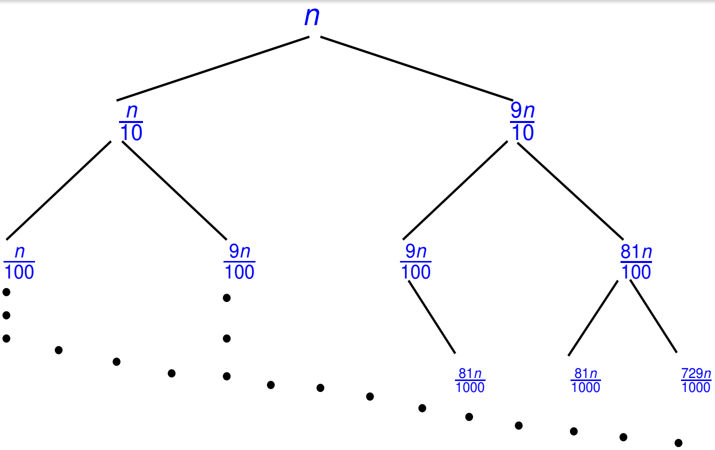
Por quê??

Suponha que (por sorte) o algoritmo PARTICIONE sempre divide o vetor na proporção $\frac{1}{9}$ para $\frac{9}{10}$. Então

$$T(n) = T(\left\lfloor \frac{n-1}{9} \right\rfloor) + T(\left\lceil \frac{9(n-1)}{10} \right\rceil) + \Theta(n)$$

Solução: $T(n)$ é $\Theta(n \lg n)$.

Árvore de recorrência



Número de níveis $\leq \log_{10/9} n$.

Em cada nível o custo é $\leq n$.

Custo total é $O(n \log n)$.

QuickSort Aleatório

O pior caso do QUICKSORT ocorre devido a uma escolha infeliz do pivô.

Um modo de minimizar este problema é usar aleatoriedade.

```
PARTICIONE-ALEATÓRIO(A, p, r)
    i ← RANDOM(p, r)
    A[i] ↔ A[r]
    devolva PARTICIONE(A, p, r)
```

```
QUICKSORT-ALEATÓRIO(A, p, r)
    se p < r
        então q ← PARTICIONE-ALEATÓRIO(A, p, r)
              QUICKSORT-ALEATÓRIO(A, p, q − 1)
              QUICKSORT-ALEATÓRIO(A, q + 1, r)
```

Análise do caso médio

Recorrência para o caso médio do algoritmo QUICKSORT-ALEATÓRIO.

$T(n)$ = consumo de tempo médio do algoritmo QUICKSORT-ALEATÓRIO.

PARTICIONE-ALEATÓRIO rearranja o vetor A e devolve um índice q tal que $A[p \dots q-1] \leq A[q]$ e $A[q+1 \dots r] > A[q]$.

$$
\begin{aligned}
T(0) &= \Theta(1) \\
T(1) &= \Theta(1) \\
T(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1}(T(k)+T(n-1-k))\right)+\Theta(n).
\end{aligned}
$$

$T(n)$ é $\Theta(???)$.

Análise do caso médio

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1}(T(k)+T(n-1-k))\right)+cn \\
&= \frac{2}{n}\sum_{k=0}^{n-1} T(k)+cn.
\end{aligned}
$$

Vou mostrar que $T(n)$ é $O(n \lg n)$.

Vou mostrar que $T(n) \leq an \lg n + b$ para $n \geq 1$ onde $a, b > 0$ são constantes.

Demonstração

$$\begin{aligned} T(n) &\leq \frac{2}{n}\sum_{k=0}^{n-1} T(k) + cn \\ &\leq \frac{2}{n}\sum_{k=0}^{n-1} (ak \lg k + b) + cn \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + cn \end{aligned}$$

Lema

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2.$$

Demonstração

$$
\begin{aligned}
T(n) &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + cn \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + cn \\
&= an \lg n - \frac{a}{4}n + 2b + cn \\
&= an \lg n + b + \left(cn + b - \frac{a}{4}n\right) \\
&\leq an \lg n + b,
\end{aligned}
$$

escolhendo a de modo que $\frac{a}{4}n \geq cn + b$ para $n \geq 1$.

Prova do Lema

$$
\begin{aligned}
\sum_{k=1}^{n-1} k \lg k &= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\
&\leq (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\
&= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\
&\leq \frac{1}{2} n(n-1) \lg n - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \\
&\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}
$$

Conclusão

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $O(n \lg n)$.

Exercício Mostre que $T(n) = \Omega(n \lg n)$.

Conclusão:

O consumo de tempo de QUICKSORT-ALEATÓRIO no caso médio é $\Theta(n \lg n)$.

Ordenação – outros métodos importantes

Algoritmos de ordenação

Algoritmos de ordenação:

- Insertion sort ✓
- Selection sort ✓
- Mergesort ✓
- Quicksort ✓
- Heapsort

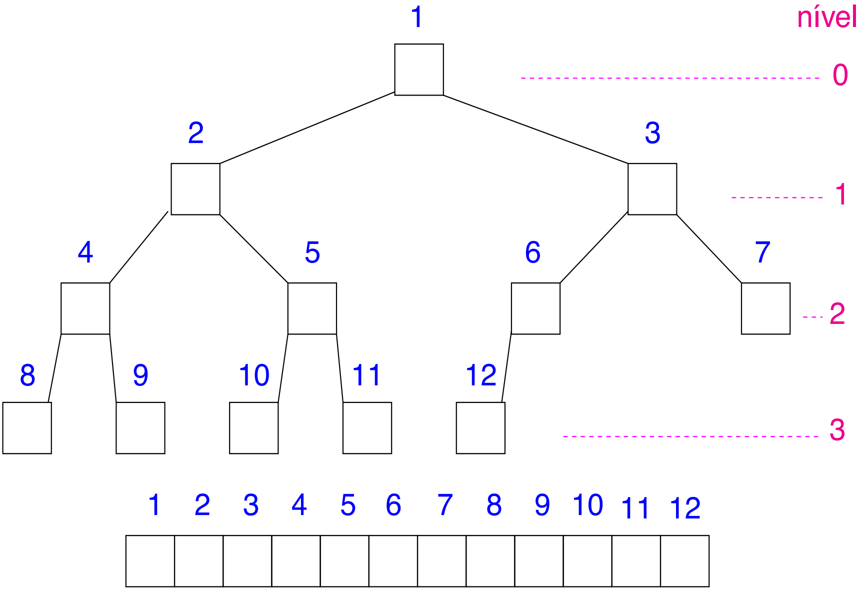
Algoritmos lineares:

- Counting sort
- Radix sort

Heapsort

- O *Heapsort* é um algoritmo de ordenação que usa uma estrutura de dados sofisticada chamada *heap*.
- A complexidade de pior caso é $\Theta(n \lg n)$.
- *Heaps* podem ser utilizados para implementar filas de prioridade que são extremamente úteis em outros algoritmos.
- Um *heap* é um vetor A que simula uma árvore binária completa, com exceção possivelmente do último nível.

Heaps



Heaps

Considere um vetor $A[1 \ldots n]$ representando um heap.

- Cada posição do vetor corresponde a um nó do heap.
- O pai de um nó i é $\lfloor i/2 \rfloor$.
- O nó 1 não tem pai.

Heaps

- Um nó i tem
  $2i$ como filho esquerdo e
  $2i + 1$ como filho direito.

- Naturalmente, o nó i
  tem filho esquerdo apenas se $2i \leq n$ e
  tem filho direito apenas se $2i + 1 \leq n$.

- Um nó i é uma folha se não tem filhos, ou seja, se $2i > n$.
- As folhas são $\lfloor n/2 \rfloor + 1, \ldots, n - 1, n$.

Níveis

Cada nível p, exceto talvez o último, tem exatamente 2^p nós e esses são

$$2^p, 2^p + 1, 2^p + 2, \ldots, 2^{p+1} - 1.$$

O nó i pertence ao nível ???.
O nó i pertence ao nível $\lfloor \lg i \rfloor$.

Prova: Se p é o nível do nó i, então

$$\begin{array}{rcccll} 2^p & \leq & i & < & 2^{p+1} & \Rightarrow \\ \lg 2^p & \leq & \lg i & < & \lg 2^{p+1} & \Rightarrow \\ p & \leq & \lg i & < & p + 1 & \end{array}$$

Logo, $p = \lfloor \lg i \rfloor$.

Portanto o número total de níveis é ???.
Portanto, o número total de níveis é $1 + \lfloor \lg n \rfloor$.

Altura

A altura de um nó i é o maior comprimento de um caminho de i a uma folha.

Os nós que têm altura zero são as folhas.

Qual é a altura de um nó i?

Altura

A altura de um nó i é o comprimento da seqüência

$$2\,i, 2^2\,i, 2^3\,i, \ldots, 2^h\,i$$

onde $2^h\,i \leq n < 2^{(h+1)}\,i$.

Assim,

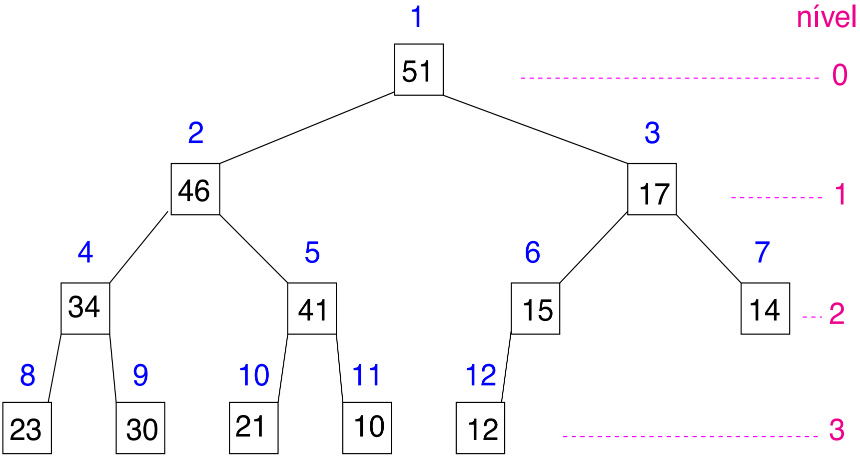
$$\begin{array}{rcccll} 2^h\,i & \leq & n & < & 2^{h+1}\,i & \Rightarrow \\ 2^h & \leq & n/i & < & 2^{h+1} & \Rightarrow \\ h & \leq & \lg(n/i) & < & h+1 & \end{array}$$

Portanto, a altura de i é $\lfloor \lg(n/i) \rfloor$.

Max-heaps

- Um nó *i* satisfaz a propriedade de (max-)heap se $A[\lfloor i/2 \rfloor] \geq A[i]$ (ou seja, pai $\geq$ filho).
- Uma árvore binária completa é um *max-heap* se todo nó distinto da raiz satisfaz a propriedade de heap.
- O máximo ou maior elemento de um max-heap está na raiz.

Max-heap

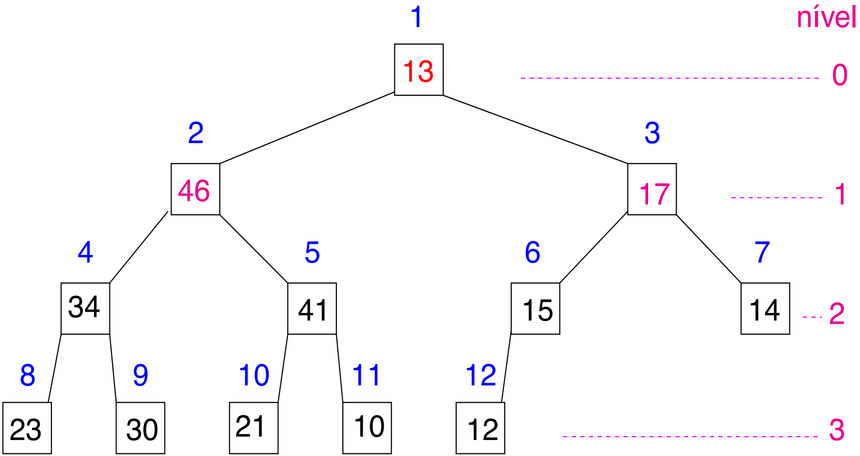


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	12

Min-heaps

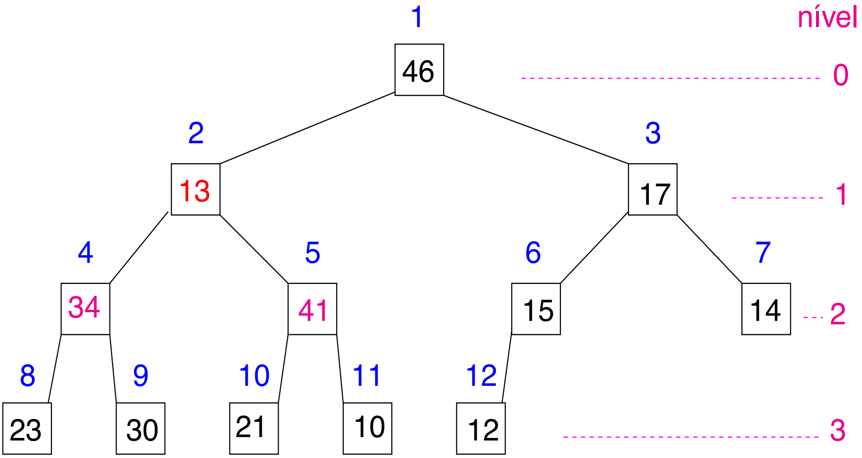
- Um nó i satisfaz a propriedade de (min-)heap se $A[\lfloor i/2 \rfloor] \leq A[i]$ (ou seja, pai $\leq$ filho).
- Uma árvore binária completa é um *min-heap* se todo nó distinto da raiz satisfaz a propriedade de min-heap.
- Vamos nos concentrar apenas em max-heaps.
- Os algoritmos que veremos podem ser facilmente modificados para trabalhar com min-heaps.

Manipulação de max-heap



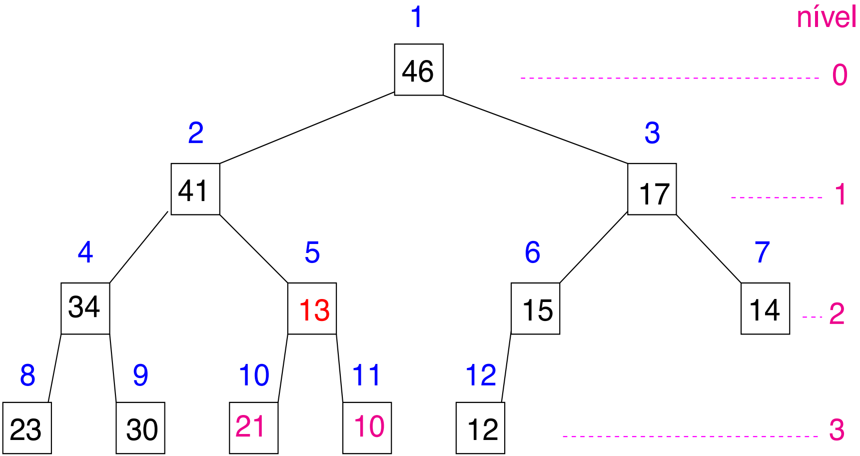
1	2	3	4	5	6	7	8	9	10	11	12
13	46	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



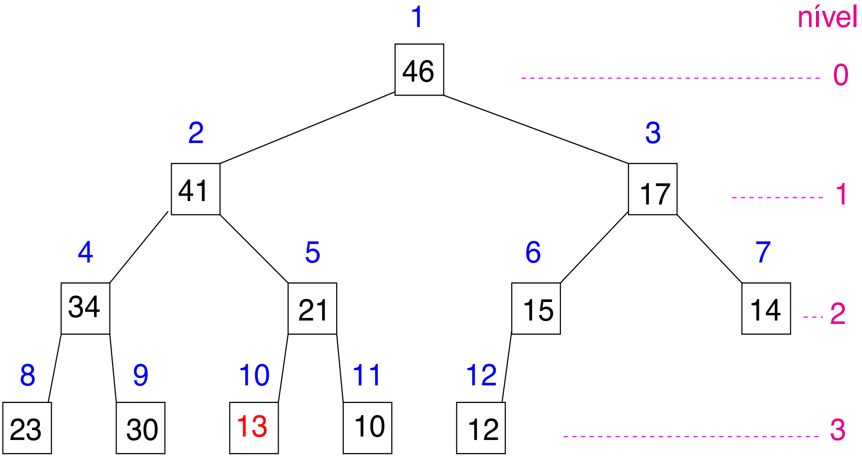
1	2	3	4	5	6	7	8	9	10	11	12
46	13	17	34	41	15	14	23	30	21	10	12

Manipulação de max-heap



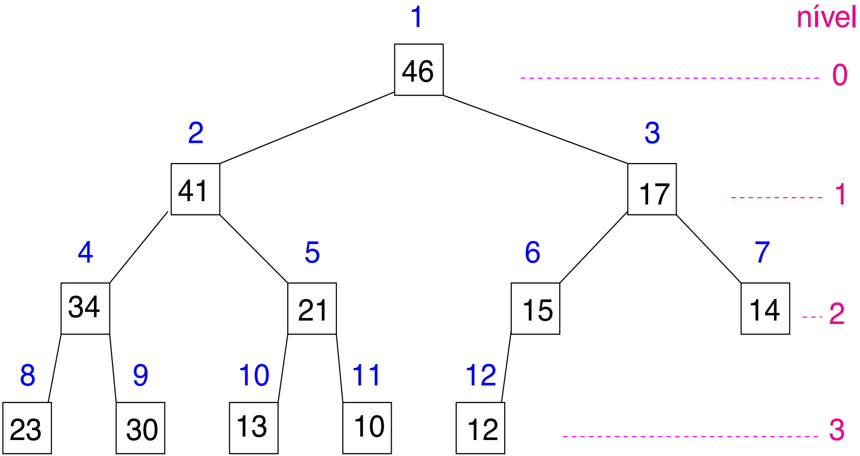
1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	13	15	14	23	30	21	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	17	34	21	15	14	23	30	13	10	12

Manipulação de max-heap

Recebe $A[1 \dots n]$ e $i \geq 1$ tais que subárvores com raízes $2i$ e $2i+1$ são max-heaps e rearranja A de modo que subárvore com raiz i seja um max-heap.

MAX-HEAPIFY(A, n, i)

$e \leftarrow 2i$
$d \leftarrow 2i + 1$
se $e \leq n$ **e** $A[e] > A[i]$
    então $\text{maior} \leftarrow e$
    senão $\text{maior} \leftarrow i$
se $d \leq n$ e $A[d] > A[\text{maior}]$
    então $\text{maior} \leftarrow d$
se $\text{maior} \neq i$
    então $A[i] \leftrightarrow A[\text{maior}]$
            MAX-HEAPIFY(A, n, maior)

Corretude de MAXHEAPIFY

A corretude de MAX-HEAPIFY segue por indução na altura *h* do nó *i*.

Base: para $h = 1$, o algoritmo funciona.

Hipótese de indução: MAX-HEAPIFY funciona para heaps de altura $< h$.

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
Após a troca na linha 9, temos $A[2i], A[2i+1] \leq A[i]$.

O algoritmo MAX-HEAPIFY transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

Corretude de MaxHeapify

Passo de indução:

A variável maior na linha 8 guarda o índice do maior elemento entre $A[i]$, $A[2i]$ e $A[2i+1]$.
Após a troca na linha 9, temos $A[2i], A[2i+1] \leq A[i]$.

O algoritmo Max-Heapify transforma a subárvore com raiz maior em um max-heap (hipótese de indução).

A subárvore cuja raiz é o irmão de maior continua sendo um max-heap.

Logo, a subárvore com raiz i torna-se um max-heap e portanto, o algoritmo Max-Heapify está correto.

Complexidade de MaxHeapify

Max-Heapify(A, n, i)	Tempo
$e \leftarrow 2i$	?
$d \leftarrow 2i + 1$	?
se $e \leq n$ **e** $A[e] > A[i]$	?
então $\text{maior} \leftarrow e$	?
senão $\text{maior} \leftarrow i$	?
se $d \leq n$ e $A[d] > A[\text{maior}]$	?
então $\text{maior} \leftarrow d$	?
se $\text{maior} \neq i$	?
então $A[i] \leftrightarrow A[\text{maior}]$	?
Max-Heapify(A, n, maior)	?

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

Complexidade de MaxHeapify

MAX-HEAPIFY(A, n, i)	Tempo
$e \leftarrow 2i$	$\Theta(1)$
$d \leftarrow 2i + 1$	$\Theta(1)$
se $e \leq n$ **e** $A[e] > A[i]$	$\Theta(1)$
então maior $\leftarrow e$	$O(1)$
senão maior $\leftarrow i$	$O(1)$
se $d \leq n$ e $A[d] > A[\text{maior}]$	$\Theta(1)$
então maior $\leftarrow d$	$O(1)$
se maior $\neq i$	$\Theta(1)$
então $A[i] \leftrightarrow A[\text{maior}]$	$O(1)$
MAX-HEAPIFY(A, n, maior)	$T(h-1)$

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h) \leq T(h-1) + \Theta(5) + O(2)$.

Complexidade de MaxHeapify

h := altura de i = $\lfloor \lg \frac{n}{i} \rfloor$

$T(h)$:= complexidade de tempo no pior caso

$T(h) \leq T(h-1) + \Theta(1)$

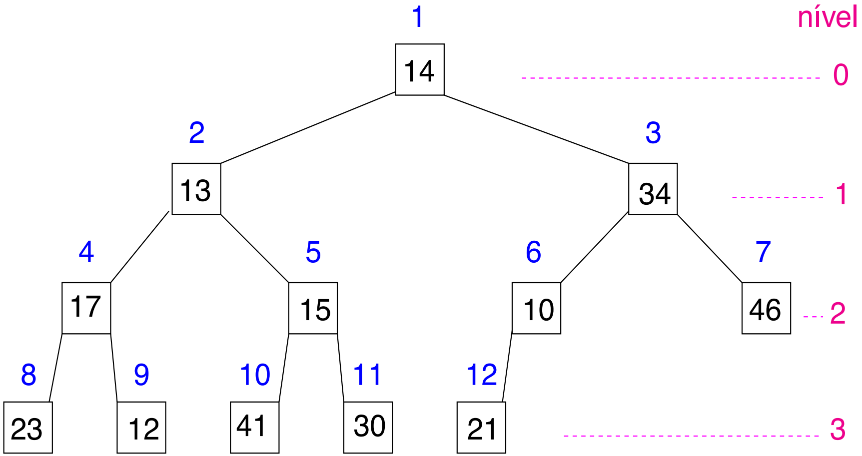
Solução assintótica: $T(n)$ é ???.

Solução assintótica: $T(n)$ é $O(h)$.

Como $h \leq \lg n$, podemos dizer que:

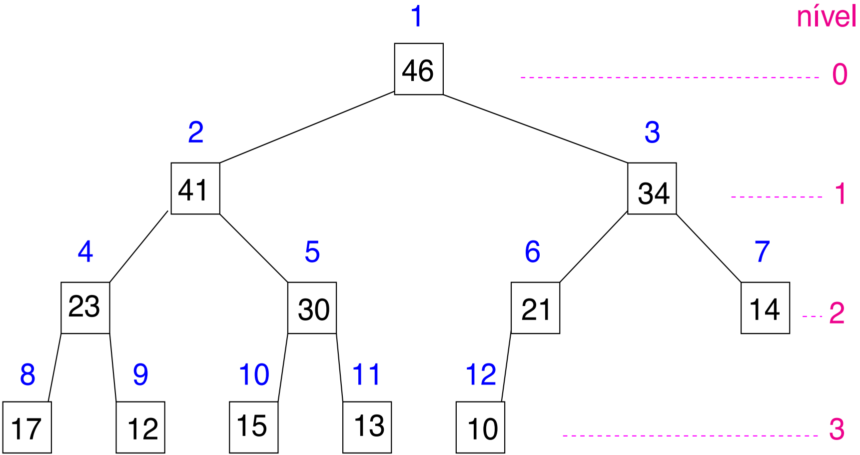
O consumo de tempo do algoritmo Max-Heapify é $O(\lg n)$ (ou melhor ainda, $O(\lg \frac{n}{i})$).

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
14	13	34	17	15	10	46	23	12	41	30	21

Construção de um max-heap



1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

Construção de um max-heap

Recebe um vetor $A[1\,.\,.\,n]$ e rearranja A para que seja max-heap.

```
BUILDMAXHEAP(A, n)
   para i ← ⌊n/2⌋ decrescendo até 1 faça
       MAX-HEAPIFY(A, n, i)
```

Invariante:

No início de cada iteração, $i+1, \ldots, n$ são raízes de max-heaps.

$T(n) =$ complexidade de tempo no pior caso

Análise grosseira: $T(n)$ é $\frac{n}{2}\,O(\lg n) = O(n \lg n)$.

Construção de um max-heap

Análise mais cuidadosa: $T(n)$ é $O(n)$.

- Na iteração i são feitas $O(h_i)$ comparações e trocas no pior caso, onde h_i é a altura da subárvore de raiz i.
- Seja $S(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- A altura de um heap é $\lfloor \lg n \rfloor + 1$.

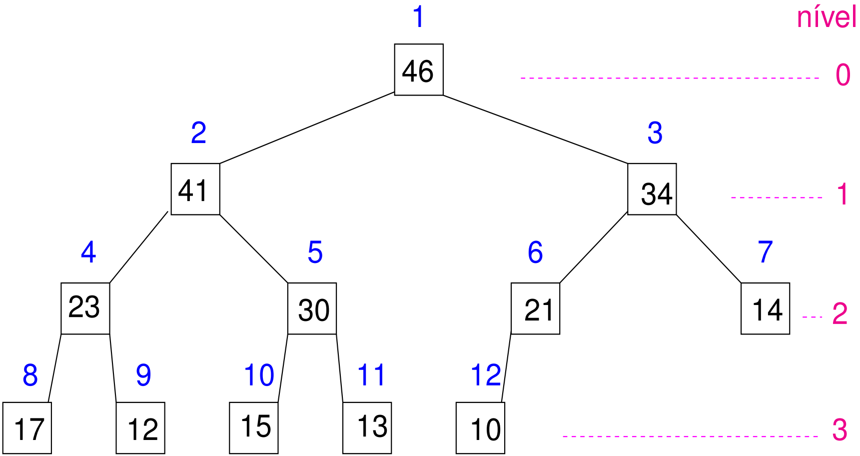
  A complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n))$.

Construção de um max-heap

- Pode-se provar por indução que $S(h) = 2^{h+1} - h - 2$.
- Logo, a complexidade de BUILDMAXHEAP é $T(n) = O(S(\lg n)) = O(n)$.

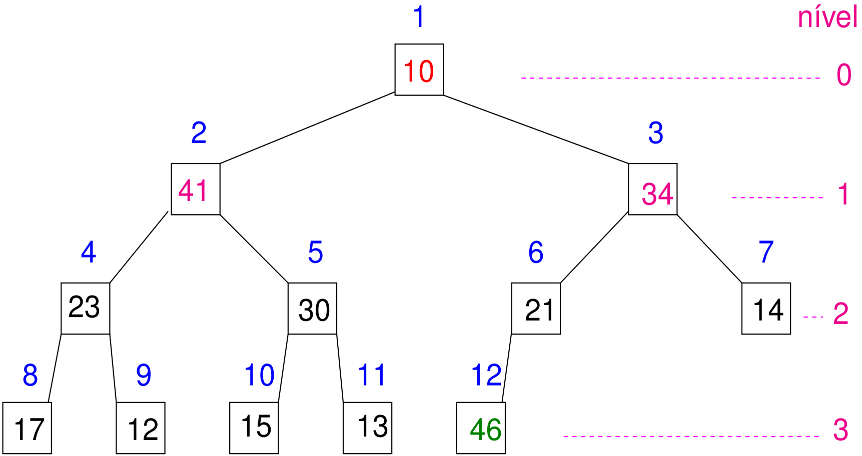
  Mais precisamente, $T(n) = \Theta(n)$. (Por quê?)
- Veja no CLRS uma prova diferente deste fato.

HeapSort



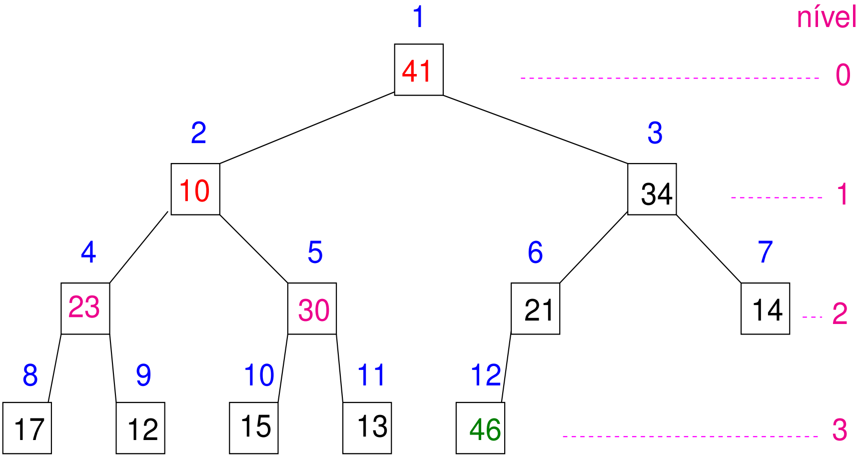
1	2	3	4	5	6	7	8	9	10	11	12
46	41	34	23	30	21	14	17	12	15	13	10

HeapSort



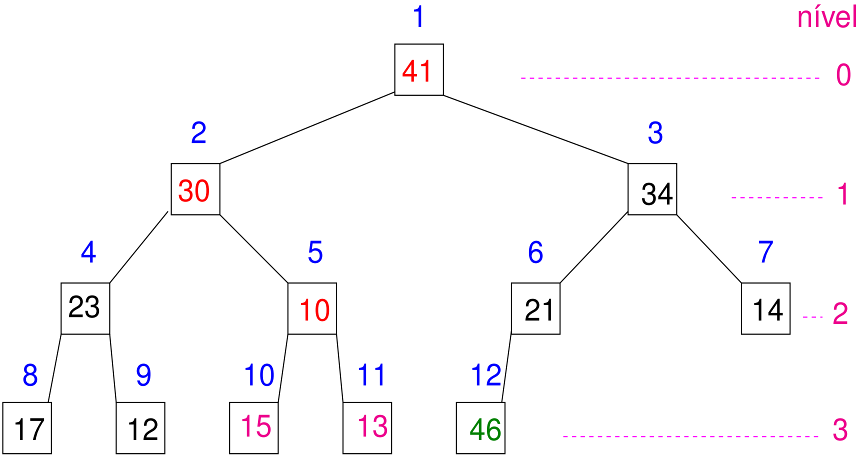
1	2	3	4	5	6	7	8	9	10	11	12
10	41	34	23	30	21	14	17	12	15	13	46

HeapSort



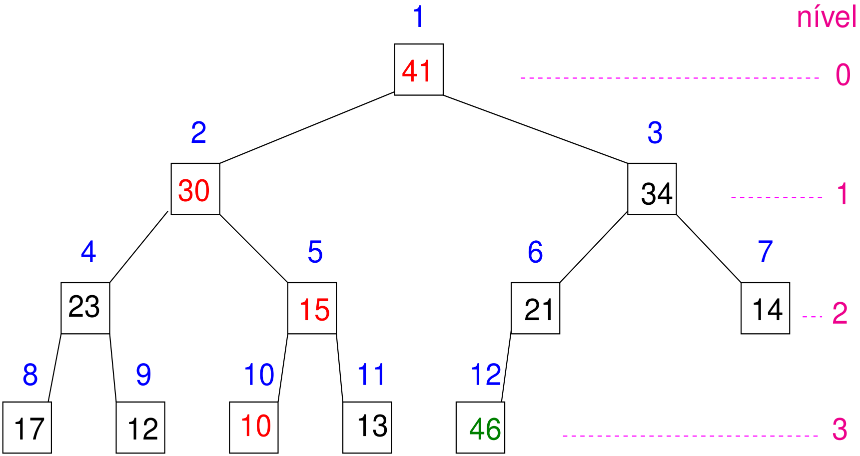
1	2	3	4	5	6	7	8	9	10	11	12
41	10	34	23	30	21	14	17	12	15	13	46

HeapSort



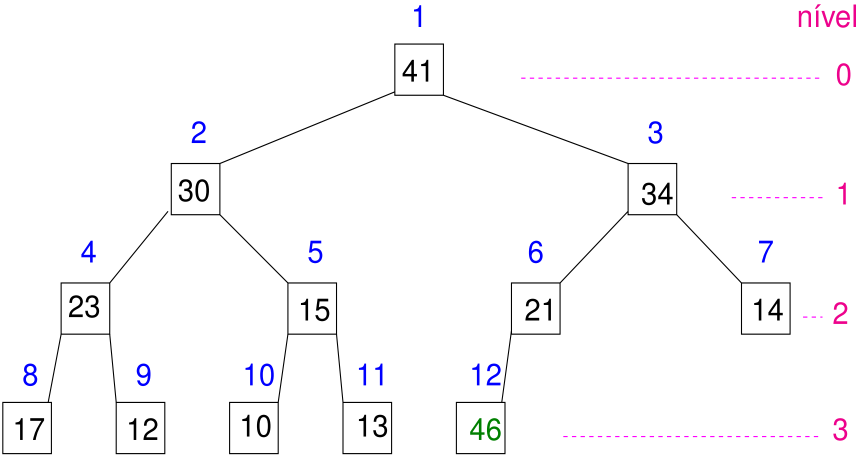
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	10	21	14	17	12	15	13	46

HeapSort



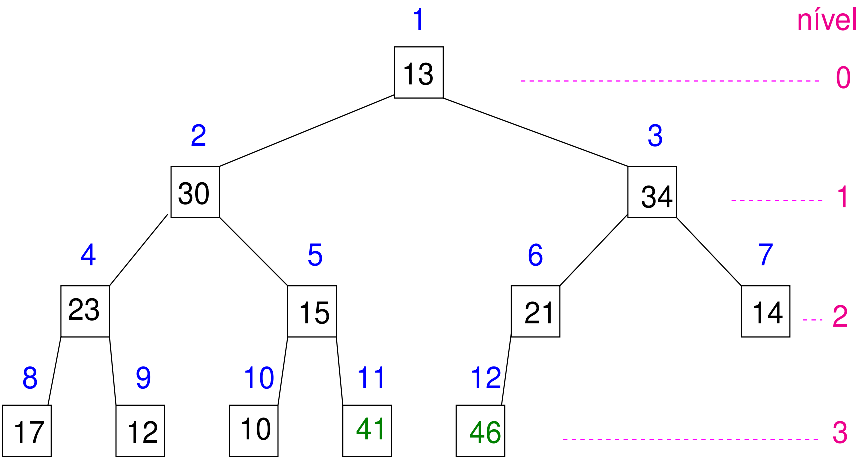
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



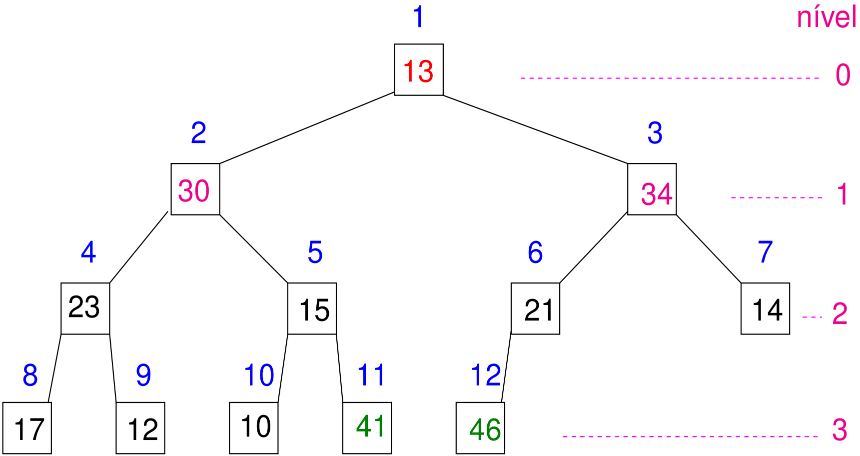
1	2	3	4	5	6	7	8	9	10	11	12
41	30	34	23	15	21	14	17	12	10	13	46

HeapSort



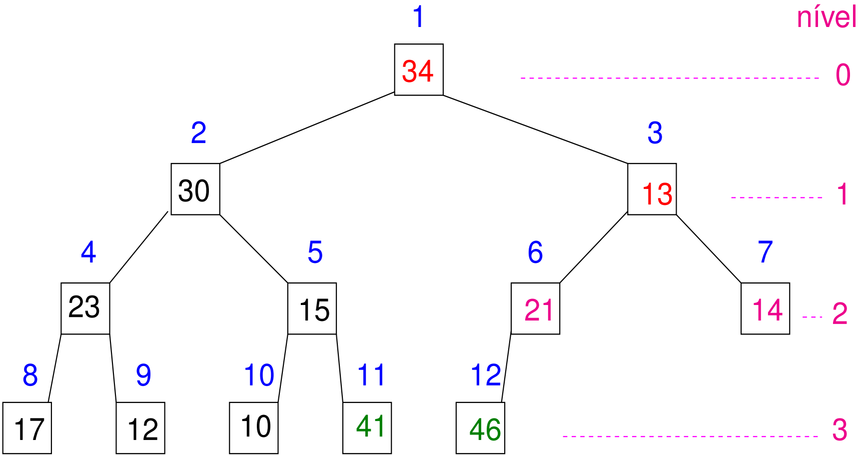
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



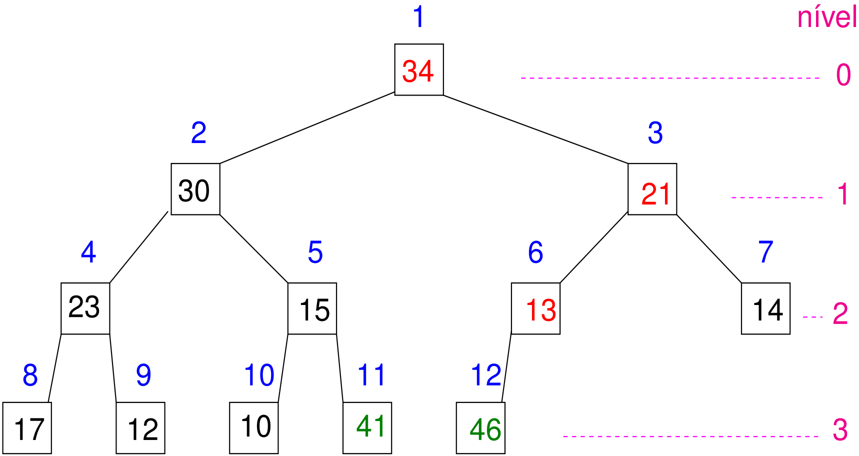
1	2	3	4	5	6	7	8	9	10	11	12
13	30	34	23	15	21	14	17	12	10	41	46

HeapSort



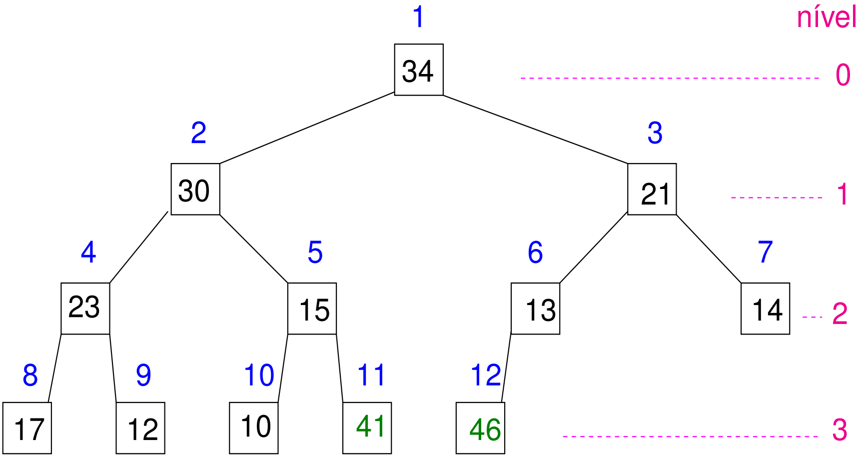
1	2	3	4	5	6	7	8	9	10	11	12
34	30	13	23	15	21	14	17	12	10	41	46

HeapSort



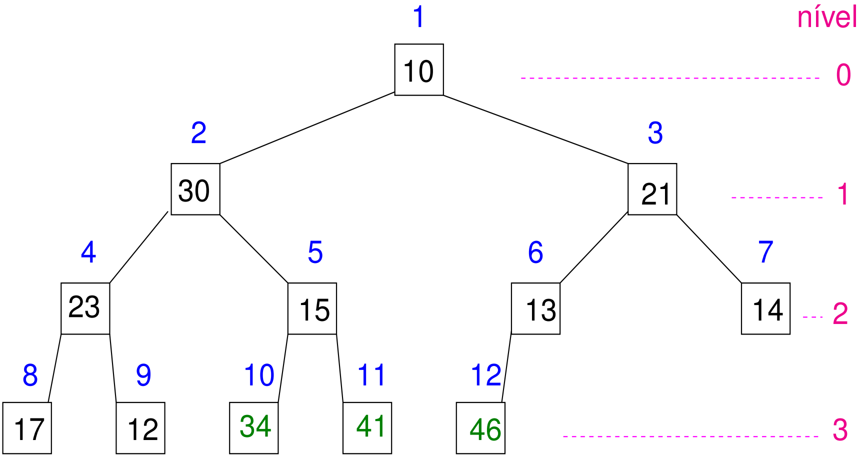
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



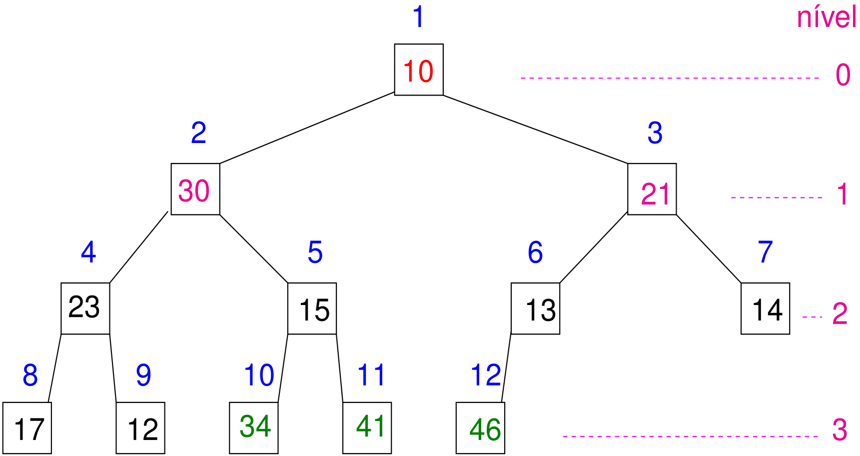
1	2	3	4	5	6	7	8	9	10	11	12
34	30	21	23	15	13	14	17	12	10	41	46

HeapSort



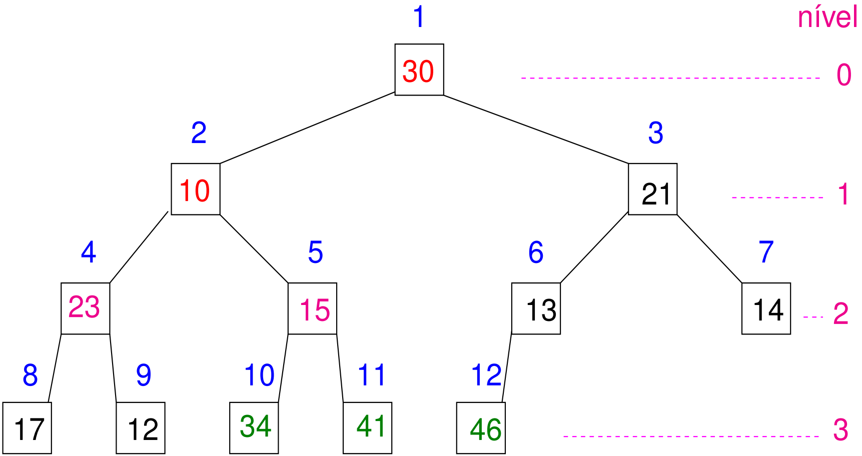
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



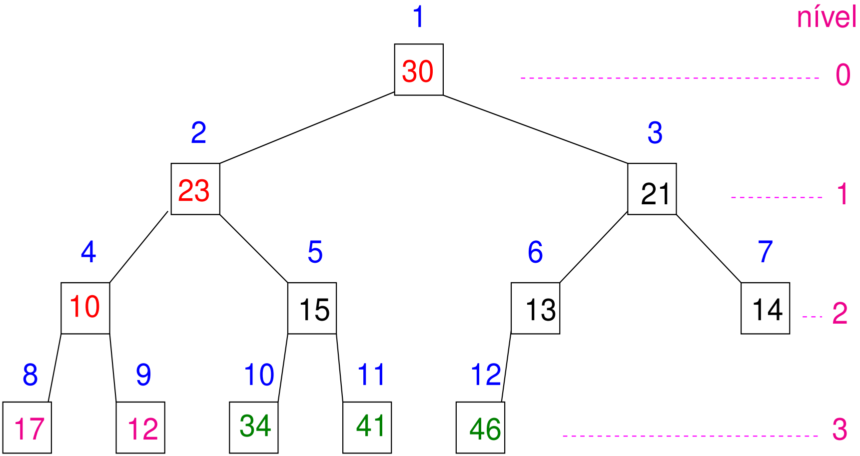
1	2	3	4	5	6	7	8	9	10	11	12
10	30	21	23	15	13	14	17	12	34	41	46

HeapSort



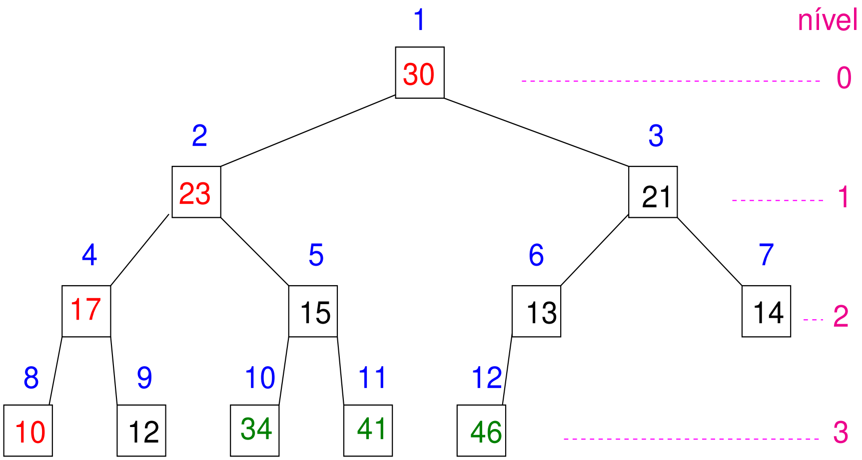
1	2	3	4	5	6	7	8	9	10	11	12
30	10	21	23	15	13	14	17	12	34	41	46

HeapSort



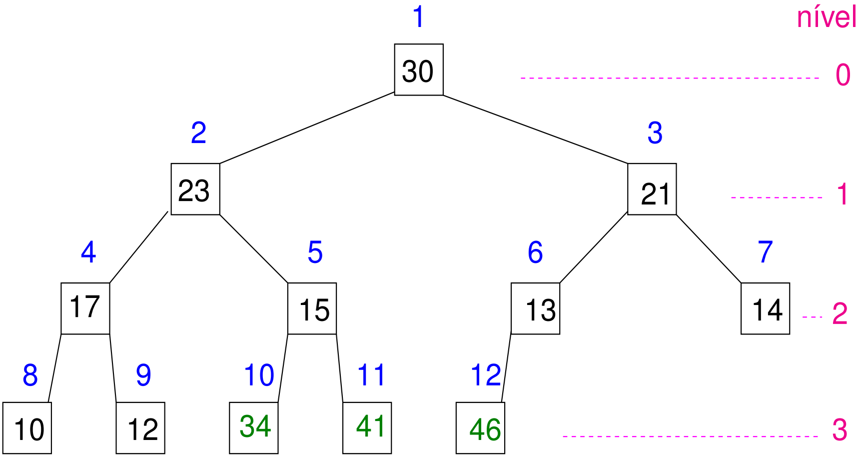
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	10	15	13	14	17	12	34	41	46

HeapSort



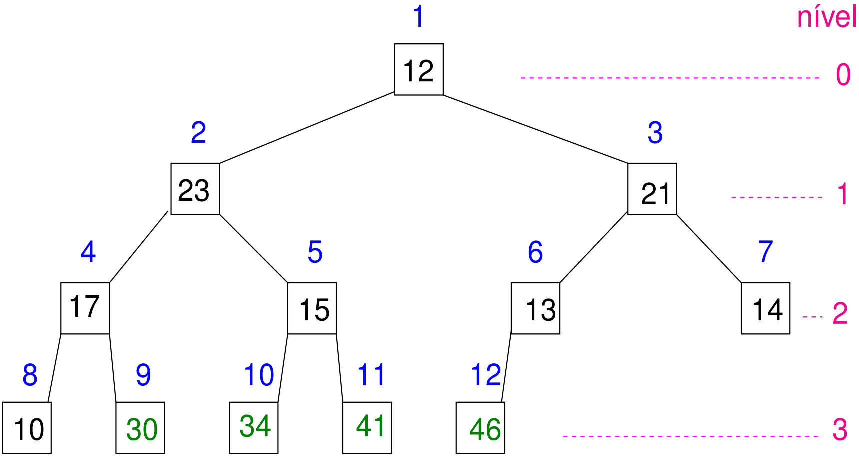
1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
30	23	21	17	15	13	14	10	12	34	41	46

HeapSort



1	2	3	4	5	6	7	8	9	10	11	12
12	23	21	17	15	13	14	10	30	34	41	46

HeapSort

Algoritmo rearranja $A[1 \ldots n]$ em ordem crescente.

HEAPSORT(A, n)
1 BUILD-MAX-HEAP(A, n)
2 $m \leftarrow n$
3 **para** $i \leftarrow n$ **decrescendo até** 2 **faça**
4 $\quad A[1] \leftrightarrow A[i]$
5 $\quad m \leftarrow m - 1$
6 $\quad$ MAX-HEAPIFY$(A, m, 1)$

Invariantes:

No início de cada iteração na linha 3 vale que:

1. $A[m \ldots n]$ é crescente;
2. $A[1 \ldots m] \leq A[m+1]$;
3. $A[1 \ldots m]$ é um max-heap.

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

HeapSort(A, n)	Tempo
Build-Max-Heap(A, n)	?
$m \leftarrow n$	?
para $i \leftarrow n$ **decrescendo até** 2 **faça**	?
$\quad A[1] \leftrightarrow A[i]$	?
$\quad m \leftarrow m - 1$	?
$\quad$ Max-Heapify$(A, m, 1)$	?

$T(n) =$ complexidade de tempo no pior caso

HeapSort

Algoritmo rearranja $A[1 \dots n]$ em ordem crescente.

$\textsc{HeapSort}(A, n)$	Tempo
$\textsc{Build-Max-Heap}(A, n)$	$\Theta(n)$
$m \leftarrow n$	$\Theta(1)$
para $i \leftarrow n$ **decrescendo até** 2 **faça**	$\Theta(n)$
$\quad A[1] \leftrightarrow A[i]$	$\Theta(n)$
$\quad m \leftarrow m - 1$	$\Theta(n)$
$\quad \textsc{Max-Heapify}(A, m, 1)$	$nO(\lg n)$

$T(n) = ??$ $\quad T(n) = nO(\lg n) + \Theta(4n + 1) = O(n \lg n)$

A complexidade de $\textsc{HeapSort}$ no pior caso é $O(n \lg n)$.

Como seria a complexidade de tempo no melhor caso?

Filas com prioridades

Uma fila com prioridades é um tipo abstrato de dados que consiste de uma coleção S de itens, cada um com um valor ou prioridade associada.

Algumas operações típicas em uma fila com prioridades são:

MAXIMUM(S): devolve o elemento de S com a maior prioridade;

EXTRACT-MAX(S): remove e devolve o elemento em S com a maior prioridade;

INCREASE-KEY(S, x, p): aumenta o valor da prioridade do elemento x para p; e

INSERT(S, x, p): insere o elemento x em S com prioridade p.

Implementação com max-heap

HEAP-MAX(A, n)
1 **devolva** $A[1]$

Complexidade de tempo: $\Theta(1)$.

HEAP-EXTRACT-MAX(A, n)
1 $\triangleright\ n \geq 1$
2 max $\leftarrow A[1]$
3 $A[1] \leftarrow A[n]$
4 $\texttt{cor} \leftarrow n - 1$
5 MAX-HEAPIFY$(A, n, 1)$
6 **devolva** max

Complexidade de tempo: $O(\lg n)$.

Implementação com max-heap

HEAP-INCREASE-KEY$(A, i, prior)$

$\triangleright$ Supõe que $prior \geq A[i]$
$A[i] \leftarrow prior$
enquanto $i > 1$ e $A[\lfloor i/2 \rfloor] < A[i]$ **faça**
    $A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$
    $i \leftarrow \lfloor i/2 \rfloor$

Complexidade de tempo: $O(\lg n)$.

MAX-HEAP-INSERT$(A, n, prior)$

$n \leftarrow n + 1$
$A[n] \leftarrow -\infty$
HEAP-INCREASE-KEY$(A, n, prior)$

Complexidade de tempo: $O(\lg n)$.