Ordenação - Projeto por Indução Simples

Demonstração por Indução

- Demonstrar a validade de $P(n)$ (propriedade P com um parâmetro natural n associado) para todo valor de n.
- Há um número infinito de casos a serem considerados (para todo n). Demonstramos todos de uma só vez:
  - **Base da Indução:** demonstramos $P(1)$.
  - **Hipótese de Indução:** supomos que $P(n)$ é verdadeiro.
  - **Passo de Indução:** provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Indução Fraca

Provar que $\sum_{i=1}^{k} 2i-1 = k^2$, agora por indução.

Base da Indução: Para $i = 1$, temos que $2-1 = 1 = 1^2$.

Hipótese de Indução: Vamos supor que $\sum_{i=1}^{k} 2i-1 = k^2$.

Passo de Indução: Vamos provar que $\sum_{i=1}^{k+1} 2i-1 = (k+1)^2$.

$$\begin{aligned}
\sum_{i=1}^{k+1} 2i-1 &= \sum_{i=1}^{k} 2i-1+2(k+1)-1 \\
&= k^2+2k+1 \text{ (pela hipótese de indução)} \\
&= (k+1)^2.
\end{aligned}$$

Indução Fraca $\times$ Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.

No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior. É necessária quando a demonstração do passo envolve supor a validade da proposição para um caso menor que n, mas que não necessariamente é o anterior.

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

Indução Forte

Prove que todo inteiro n pode ser escrito como a soma de diferentes potências de 2.

Base da Indução: Para $i = 1$, temos que $2^0 = 1$.

Hipótese de Indução: $1 \leq i \leq k$, i pode ser escrito como a soma de diferentes potências de 2.

Passo de Indução: Seja m a maior potência de 2 que "cabe" em k, isto é: $2^m <= k < 2^(m+1)$. Vamos provar que $k = A + 2^m$ pode ser escrito como soma de diferentes potências de 2. Se $A = 2^m$, fim da prova. Senão $A = k - 2^m$. Se $A < 2^(m+1) - 2^m = 2^m$, então $k - 2^m < 2^m$ e portanto, A pode ser escrito como soma de diferentes potências de 2.

Motivação

- ordenar faz parte do nosso dia-a-dia
- ordenação de dados pode facilitar e aumentar a eficiência das operações de pesquisa sobre um conjunto de dados
- como ordenar? Objetivo: minimizar a tarefa computacional de movimentação de dados.
- busca – tarefa muito comum

Questões como: o que tenho para fazer hoje? Qual a primeira tarefa que vou realizar? Por que escolhi essa tarefa como sendo a primeira? são respondidas ordenando, de alguma forma as tarefas do dia.

Em geral, um conjunto de itens é classificado para para simplificar a recuperação manual das informações ou para tornar eficiente o acesso da máquina aos dados.

Como a operação de busca é uma tarefa muito comum em computação, o conhecimento desses métodos é um passo importante para que você se torne um bom programador.

Objetivos

- estudar métodos de ordenação de arquivos de dados em que cada elemento (item) é caracterizado por uma chave ("key")
- chaves são usadas para controlar a ordenação
- tempo de execução usualmente proporcional ao número de comparações número de movimentações e/ou trocas

Ordenação Interna e Externa

- Interna – conjunto de todos os dados a ordenar cabem na memória.
  - qualquer registro pode ser imediatamente acessado
  - foco nesta disciplina
- Externa – necessita de memória secundária
  - registros são acessados sequencialmente ou em grandes blocos
  - ordenar dados de disco, por exemplo

Complexidade de Algoritmos

Seja E_n o conjunto de entradas de tamanho n para o problema P. Seja I um elemento de E_n; $p(I)$ a probabilidade de que I ocorra e $t(I)$ o número de operações básicas executadas pelo algoritmo com entrada I.

- Análise do pior caso: $\max_{I \in E_n} t(I)$
- Análise do melhor caso: $\min_{I \in E_n} t(I)$
- Análise do caso médio: $\sum_{I \in E_n} p(I)t(I)$

Uma possível maneira de expressar os resultados de um algoritmo é computar o comportamento médio do algoritmo, isto é, computar o número de operações básicas executadas para cada entrada de tamanho n e, então, fazer a média. O comportamento médio pode ser definido como:

$$A(n) = \sum_{I \in E_n} p(I)t(I)$$

Uma outra abordagem para descrever o comportamento de um algoritmo é calcular a sua complexidade de **pior-caso**.

$$W(n) = \max_{I \in E_n} t(I)$$

$W(n)$ é o número máximo de operações básicas executadas pelo algoritmo com qualquer entrada de tamanho n. Observe que a complexidade de pior-caso é um limitante superior para o número de operações básicas executadas pelo algorimo.

A complexidade de **melhor-caso** é dada pela função:

$$f(n) = \min_{I \in E_n} t(I)$$

O Problema da Ordenação

Problema: Ordenar um conjunto de $n \geq 1$ inteiros.

- Podemos projetar por indução diversos algoritmos para o problema da ordenação.
- Na verdade, todos os algoritmos básicos de ordenação surgem de projetos por indução sutilmente diferentes.
- Começaremos usando indução simples no projeto do algoritmo de ordenação.

Projeto por Indução Simples

Hipótese de Indução Simples: Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Primeira Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.
- Esta indução dá origem ao algoritmo incremental *Insertion Sort*.

Exemplo

```
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A O S R T I N G E X A M P L E
A O R S T I N G E X A M P L E
A O R S T I N G E X A M P L E
A I O R S T N G E X A M P L E
A I N O R S T G E X A M P L E
A G I N O R S T E X A M P L E
A E G I N O R S T X A M P L E
A E G I N O R S T X A M P L E
A A E G I N O R S T X M P L E
A A E G I M N O P R S T X L E
A A E G I M N O P R S T X L E
A A E G I L M N O P R S T X E
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Insertion Sort - Versão Iterativa

OrdenacaoInsercao(A**)**
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.

1. **para** $i := 2$ **até** n **faça**
2. $\quad v := A[i]$
3. $\quad j := i - 1$
4. $\quad$ **enquanto** $(j > 0)$ e $(A[j] > v)$ **faça**
5. $\qquad A[j+1] := A[j]$
6. $\qquad j := j - 1$
7. $\quad A[j+1] := v$

Insertion Sort - Versão Recursiva

OrdenacaoInsercao(A, n**)**
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.

1. **se** $n \geq 2$ **faça**
2. $\quad$ OrdenacaoInsercao$(A, n-1)$
3. $\quad v := A[n]$
4. $\quad j := n - 1$
5. $\quad$ **enquanto** $(j > 0)$ e $(A[j] > v)$ **faça**
6. $\qquad A[j+1] := A[j]$
7. $\qquad j := j - 1$
8. $\quad A[j+1] := v$

Complexidade

1. Pior caso – vetor com os elementos em ordem invertida:
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \Rightarrow \mathcal{O}(n^2)$$
2. Melhor caso – vetor ordenado:
$$\sum_{i=1}^{n-1} 1 = n - 1 \Rightarrow \mathcal{O}(n)$$
3. Caso médio – supondo igualmente prováveis:
$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2}\frac{n(n-1)}{2} \Rightarrow \mathcal{O}(n^2)$$

Insertion Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Insertion Sort* executa no pior caso ?
- Tanto o número de comparações quanto o de trocas é dado pela recorrência:

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

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.

Características

- número mínimo de comparações e movimentos – itens estão originalmente em ordem
- número máximo – itens estão originalmente na ordem reversa
- método a ser utilizado quando o arquivo está "quase" ordenado
- bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado
- algoritmo estável

Um algoritmo é dito estável se a ordem relativa dos itens com chaves iguais mantém-se inalterada pelo processo de ordenação.

Segunda Alternativa

Hipótese de Indução Simples: Sabemos ordenar um conjunto de $n-1 \geq 1$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Segunda Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x o menor elemento de S. Então x certamente é o primeiro elemento da seqüência ordenada de S e basta ordenarmos os demais elementos de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$ e assim obtemos S ordenado.
- Esta indução dá origem ao algoritmo incremental *Selection Sort*.

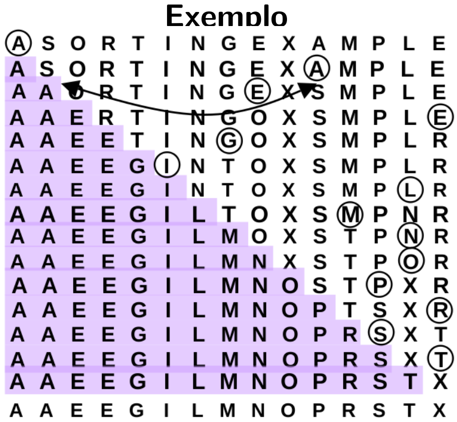
Selection Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Selection Sort* executa no pior caso ?
- O número de comparações é dado pela recorrência:

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

- Portanto, $\Theta(n^2)$ comparações são executadas no pior caso.

Exemplo



Selection Sort - Análise de Complexidade - Cont.

- Já o número de trocas é dado pela recorrência:

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

- Portanto, $\Theta(n)$ trocas são executadas no pior caso.
- Apesar dos algoritmos *Insertion Sort* e *Selection Sort* terem a mesma complexidade assintótica, em situações onde a operação de troca é muito custosa, é preferível utilizar *Selection Sort*.

Selection Sort - Recursiva

OrdenacaoSelecao(A, i, n)
Entrada: Vetor A de n números inteiros e os índices de início e término da seqüência a ser ordenada.
Saída: Vetor A ordenado.

```
se i < n faça
    min := i
    para j := i + 1 até n faça
        se A[j] < A[min] então min := j
    t := A[min]
    A[min] := A[i]
    A[i] := t
    OrdenacaoSelecao(A, i + 1, n)
```

Selection Sort - Versão Iterativa

OrdenacaoSelecao(A)
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.

```
para i := 1 até n − 1 faça
    min := i
    para j := i + 1 até n faça
        se A[j] < A[min] então min := j
    t := A[min]
    A[min] := A[i]
    A[i] := t
```

Complexidade

1. Comparações:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \Rightarrow \mathcal{O}(n^2)$$

2. Trocas:

$$\sum_{i=1}^{n-1} 1 = n-1 \Rightarrow \mathcal{O}(n)$$

3. A atribuição `min := j` é executada em média $n \log n$ vezes, Knuth (1973).

Note que o número de movimentação é $3(n-1)$.

Outra observação é que o desempenho é independente da ordenação inicial dos dados. A única coisa que depende desta ordenação é o número de vezes que `min` é atualizado (quadrático no pior caso, $n \log n$ em média).

Vantagens

- custo linear no tamanho da entrada para o número de movimentos de registros (bom para arquivos com registros grandes)
- é muito interessante para arquivos pequenos

Desvantagens

- arquivo ordenado não ajuda em nada.
- não é estável

Terceira Alternativa

- **Passo da Indução (Terceira Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x o maior elemento de S. Então x certamente é o último elemento da seqüência ordenada de S e basta ordenarmos os demais elementos de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$ e assim obtemos S ordenado.
- Em princípio, esta indução dá origem a uma variação do algoritmo *Selection Sort*.
- No entanto, se implementamos de uma forma diferente a seleção e o posicionamento do maior elemento, obteremos o algoritmo *Bubble Sort*.

Exemplo

8	5	4	3	9	6
5	8				
	4	8			
		3	8		
			8	9	
				6	9
4	3	5	6	8	9
3	4	5	6	8	9
3	4	5	6	8	9

Bubble Sort - Versão Iterativa

BubbleSort(A)
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.

```
para i := n decrescendo até 1 faça
    para j := 2 até i faça
        se A[j-1] > A[j] então
            t := A[j-1]
            A[j-1] := A[j]
            A[j] := t
```

- em cada passo, cada elemento é comparado com o próximo elemento no vetor
- se o elemento estiver fora de ordem, a troca é realizada
- uma possível otimização é repetir os dois passos acima até que não ocorram mais trocas

Bubble Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Bubble Sort* executa no pior caso ?
- O número de comparações e de trocas é dado pela recorrência:

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

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.
- Ou seja, algoritmo *Bubble Sort* executa mais trocas que o algoritmo *Selection Sort* !

Complexidade

1. Número de comparações:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \Rightarrow \mathcal{O}(n^2)$$

2. Algoritmo termina no passo k quando não há mais trocas:
   $(n-1)+(n-2)+\cdots+(n-k) =$

$$\sum_{i=k}^{n-1} i = \frac{2kn-k^2-k}{2}$$

   Considerando que o número de iterações k é $\mathcal{O}(n)$ $\Rightarrow \mathcal{O}(n^2)$

Vantagens

- algoritmo simples
- algoritmo estável

Desvantagem

- o fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.

Comparando...

Algoritmo	Comparações	Trocas
Insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Bubble	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

Projeto por Indução Forte

- **Hipótese de Indução Forte:** Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.
- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Primeira Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros. Podemos particionar S em dois conjuntos, S_1 e S_2, de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$. Como $n \geq 2$, ambos S_1 e S_2 possuem menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Podemos então obter S ordenado intercalando os conjuntos ordenados S_1 e S_2.

Também podemos usar indução forte para projetar algoritmos para o problema da ordenação.

Projeto por Indução Forte

- Dá origem ao algoritmo de divisão e conquista *Mergesort*.
- O conjunto S é um vetor de tamanho n.
- A operação de divisão é imediata, o vetor é dividido em dois vetores com metade do tamanho do original, que são ordenados recursivamente.
- O trabalho do algoritmo está concentrado na conquista: a intercalação dos dois subvetores ordenados.
- Para simplificar a implementação da operação de intercalação e garantir sua complexidade linear, usamos um vetor auxiliar.

Mergesort - Intercalação

	k								
A	33	44	55	66	99	11	22	77	88

	i							j	
B	33	44	55	66	99	88	77	22	11

Mergesort - Intercalação

		k							
A	11	44	55	66	99	11	22	77	88
	i							j	
B	33	44	55	66	99	88	77	22	11

Mergesort - Intercalação

			k						
A	11	22	55	66	99	11	22	77	88
		i						j	
B	33	44	55	66	99	88	77	22	11

Vamos continuar...

Mergesort - Pseudo-código

OrdenacaoIntercalacao(A, e, d)
Entrada: Vetor A de n números inteirosos índices e e d que delimitam início e fim do subvetor a ser ordenado.
Saída: Subvetor de A de e a d ordenado.

se $d > e$ **então**
  $m := (e + d)\ div\ 2$
  OrdenacaoIntercalacao(A, e, m)
  OrdenacaoIntercalacao($A, m + 1, d$)
  para i **de** e **até** m **faça**
    $B[i] := A[i]$
  para j **de** $m + 1$ **até** d **faça**
    $B[d + m + 1 - j] := A[j]$

Mergesort - Pseudo-código (cont.)

OrdenacaoIntercalacao(A, e, d):cont.

  $i := e$; $j := d$
  para k **de** e **até** d **faça**
    se $B[i] < B[j]$ **então**
      $A[k] := B[i]$
      $i := i + 1$
    senão
      $A[k] := B[j]$
      $j := j - 1$

O motivo para copiar o trecho do segundo subvetor em ordem reversa é pois assim o último elemento do primeiro subvetor serve de sentinela quando varremos o segundo subvetor e vice-versa.

Mergesort

	e				m				d
A	55	33	66	44	99	11	77	22	88

Mergesort

	e				m				d
A	55	33	66	44	99	11	77	22	88
	e		m		d				
A	55	33	66	44	99	11	77	22	88

Vamos continuar....

Mergesort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Mergesort* executa no pior caso ?
- A etapa de intercalação tem complexidade $\Theta(n)$, logo, o número de comparações e de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n \log n)$

Mergesort - Análise de Complexidade

- Ou seja, algoritmo *Mergesort* é assintoticamente mais eficiente que todos os anteriores.
- Em contrapartida, o algoritmo *Mergesort* usa o dobro de memória. Ainda assim, assintoticamente o gasto de memória é equivalente ao dos demais algoritmos.
- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?

Mergesort - Análise de Complexidade

- Sim! Basta deslocarmos os elementos de um dos subvetores, quando necessário, para dar lugar ao mínimo dos dois subvetores.
- No entanto, a etapa de intercalação passa a ter complexidade $\Theta(n^2)$, resultando na seguinte recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n^2, & n > 1, \end{cases}$$

Mergesort - Análise de Complexidade

- Ou seja, a complexidade do *Mergesort* passa a ser $\Theta(n^2)$!
- Como era de se esperar, a eficiência da etapa de intercalação é crucial para a eficiência do *Mergesort*.

Segunda Alternativa

- **Hipótese de Indução Forte:** Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.
- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Segunda Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Sejam S_1 e S_2 os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente. Ambos S_1 e S_2 possuem menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Podemos obter S ordenado concatenando S_1 ordenado, x e S_2 ordenado.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Quicksort*.
- Na implementação do algoritmo, o conjunto S é um vetor de tamanho n novamente.
- Em contraste ao *Mergesort*, no *Quicksort* é a operação de divisão que é a operação mais custosa: depois que escolhemos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.

Projeto por Indução Forte

- Conseguimos fazer essa divisão com $\Theta(n)$ operações: basta varrer o vetor com dois apontadores, um varrendo da direita para a esquerda e outro da esquerda para a direita, em busca de elementos situados na parte errada do vetor, e trocar um par de elementos de lugar quando encontrado.
- Após essa etapa basta ordenarmos os dois trechos do vetor recursivamente para obtermos o vetor ordenado, ou seja, a conquista é imediata.

Quicksort

	e								d
A	33	66	44	99	11	77	22	88	55

i $\rightarrow$ $\leftarrow$ j

- i procura por elementos $\geq$ ao pivot
- j procura por elementos $\leq$ ao pivot

Quicksort

	e								d
A	33	66	44	99	11	77	22	88	55
		i					j		

- troca!

A	33	22	44	99	11	77	66	88	55

Quicksort

	e								d
A	33	22	44	99	11	77	66	88	55
				i	j				

- troca!

A	33	22	44	11	99	77	66	88	55

Quicksort

	e								d
A	33	22	44	11	99	77	66	88	55
				j	i				

- i e j se cruzaram!
- hora de colocar o pivot em seu lugar

A	33	22	44	11	55	77	66	88	99

Quicksort - Pseudo-código

Quicksort(A, e, d**)**
Entrada: Vetor A de números inteiros e os índices e e d que delimitam início e fim do subvetor a ser ordenado.
Saída: Subvetor de A de e a d ordenado.

se $d > e$ **então**
  $v := A[d]$ {escolhe pivot}
  $i := e - 1$
  $j := d$

repita
  repita $i := i + 1$ **até que** $A[i] \geq v$
  repita $j := j - 1$ **até que** $A[j] \leq v$ **ou** $j = e$
  $t := A[i]$
  $A[i] := A[j]$
  $A[j] := t$
até que $j \leq i$
$A[j] := A[i]$
$A[i] := A[d]$
$A[d] := t$
Quicksort($A, e, i - 1$)
Quicksort($A, i + 1, d$)

Quicksort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Quicksort* executa no pior caso ?
- Certamente a operação de divisão tem complexidade $\Theta(n)$, mas o tamanho dos dois subproblemas depende do *pivot* escolhido.
- No pior caso, cada divisão sucessiva do *Quicksort* separa um único elemento dos demais, recaindo na recorrência:

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

- Portanto, $\Theta(n^2)$.

Quicksort - Análise de Complexidade

- Então, o algoritmo *Quicksort* é assintoticamente menos eficiente que o *Mergesort* no pior caso.
- Porém, no caso médio, o *Quicksort* efetua $\Theta(n \log n)$ comparações e trocas.
- Assim, na prática, o *Quicksort* é bastante eficiente, com uma vantagem adicional em relação ao *Mergesort*: é *in place*, isto é, não utiliza um vetor auxiliar.

Quicksort - Análise de Caso Médio

- Considere que i é o índice da posição do *pivot* escolhido no vetor ordenado.
- Supondo que qualquer elemento do vetor tem igual probabilidade de ser escolhido como o *pivot*, então, na média, o tamanho dos subproblemas resolvidos em cada divisão sucessiva será:

$$\frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)), \text{ para } n \geq 2$$

- Sabemos que:

$$\sum_{i=1}^{n} T(i-1) = \sum_{i=1}^{n} T(n-i) = \sum_{i=1}^{n-1} T(i), \text{ supondo } T(0) = 0$$

Quicksort - Análise de Caso Médio

- Assim, no caso médio, o número de operações efetuadas pelo *Quicksort* é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 0 \text{ ou } n = 1 \\ \frac{2}{n}\sum_{i=1}^{n-1} T(i) + n - 1, & n \geq 2, \end{cases}$$

- Esta é uma recorrência de história completa conhecida ! Sabemos que $T(n) \in \Theta(n \log n)$.
- Portanto, na média, o algoritmo *Quicksort* executa $\Theta(n \log n)$ trocas e comparações.

Heapsort

- O projeto por indução que leva ao *Heapsort* é essencialmente o mesmo do *Selection Sort*: selecionamos e posicionamos o maior (ou menor) elemento do conjunto e então aplicamos a hipótese de indução para ordenar os elementos restantes.
- A diferença importante é que no *Heapsort* utilizamos a estrutura de dados *heap* para selecionar o maior (ou menor) elemento eficientemente.
- Um *heap* é um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de *heap*.

Considerando o nó i

- filho esquerdo de i: $2i$
- filho direito de i: $2i + 1$
- pai de i: $\lfloor i/2 \rfloor$
- nível da raiz: 0
- nível de i:
- altura da árvore:
- altura de i:

Heapsort - Estrutura do Heap

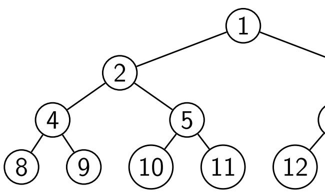
- Na simulação da árvore binária completa com um vetor, definimos que o nó i tem como filhos esquerdo e direito os nós $2i$ e $2i + 1$ e como pai o nó $\lfloor i/2 \rfloor$.
- Uma árvore com estrutura de *heap* é aquela em que, para toda subárvore, o nó raiz é maior ou igual (ou menor ou igual) às raízes das subárvores direita e esquerda.
- Assim, o maior (ou menor) elemento do *heap* está sempre localizado no topo, na primeira posição do vetor.

Considerando o nó i – `v[1..m]`

- filho esquerdo de i: $2i$
- filho direito de i: $2i + 1$
- pai de i: $\lfloor i/2 \rfloor$
- nível da raiz: 0
- nível de i: $\lg i$
- altura da árvore: $\lg m + 1$
- altura de i: $\lg(m/i)$

Estrutura do Heap

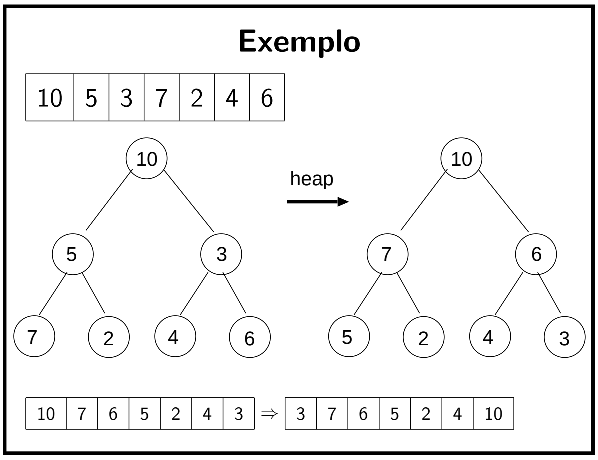
- pai(i) $= \lfloor i/2 \rfloor$
- esq(i) $= 2 * i$
- dir(i) $= 2 * i + 1$

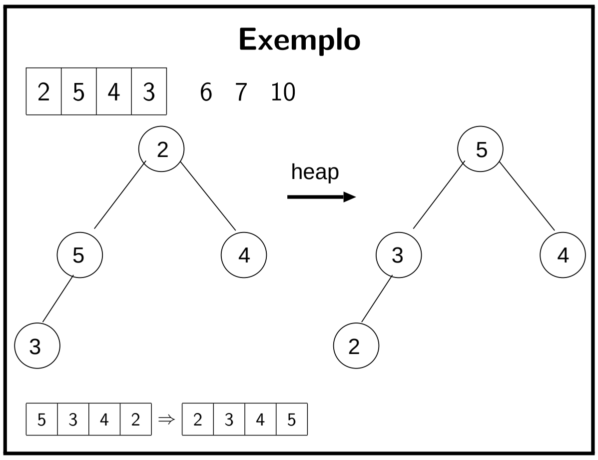


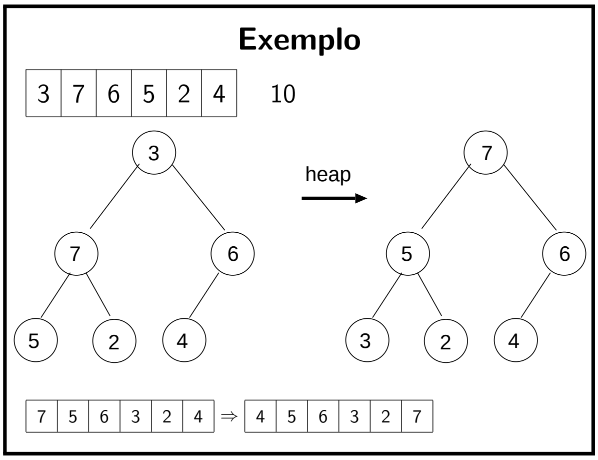
1	2	3	4	5	6	7	8	9	10	11	12
i	$2i$	$2i+1$	$4i$	$4i+1$							

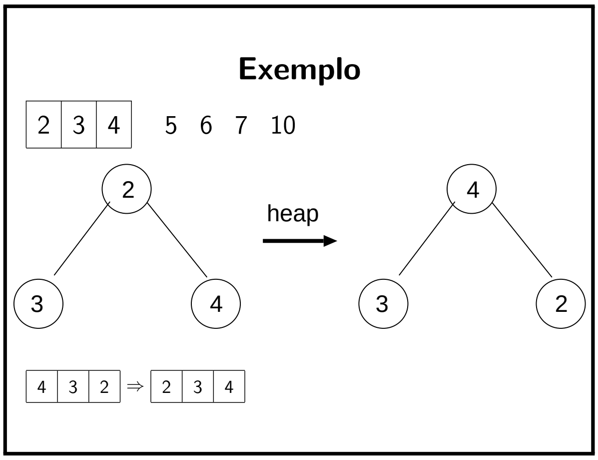
Heapsort - O Algoritmo

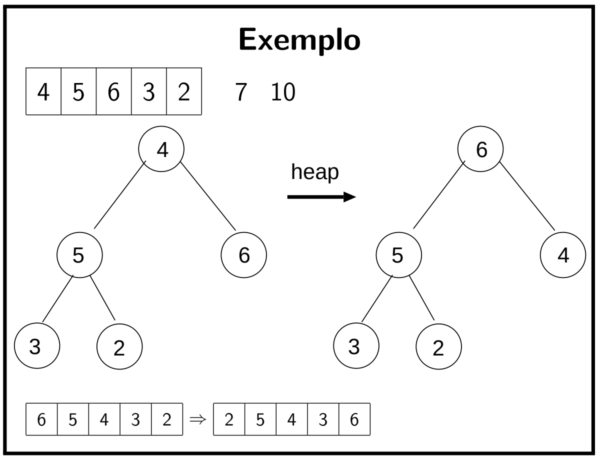
- Então, o uso da estrutura *heap* permite que:
  - O elemento máximo do conjunto seja determinado e corretamente posicionado no vetor em tempo constante, trocando o primeiro elemento do *heap* com o último.
  - O trecho restante do vetor (do índice 1 ao $n - 1$), que pode ter deixado de ter a estrutura de heap, volte a tê-la com número de trocas de elementos proporcional à altura da árvore.
- *Heapsort* – construção de um *heap* com os elementos a serem ordenados, seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do *heap*.

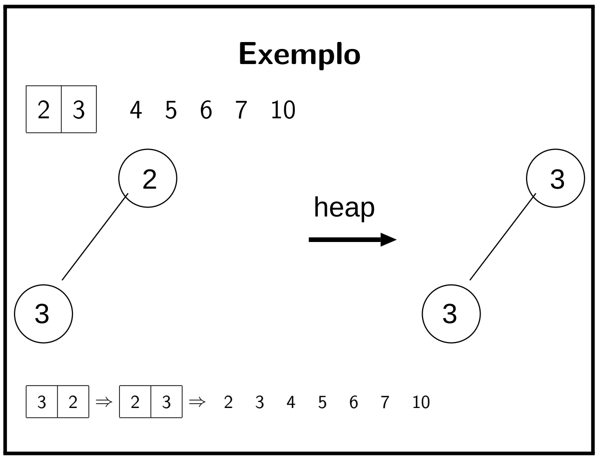

Exemplo
10 5 3 7 2 4 6
heap
10 7 6 5 2 4 3 ⇒ 3 7 6 5 2 4 10


Exemplo
2 5 4 3 6 7 10
heap
5 3 4 2 ⇒ 2 3 4 5


Exemplo
3 7 6 5 2 4 10
heap
7 5 6 3 2 4 ⇒ 4 5 6 3 2 7


Exemplo
2 3 4 5 6 7 10
heap
4 3 2 ⇒ 2 3 4


Exemplo
4 5 6 3 2 7 10
heap
6 5 4 3 2 ⇒ 2 5 4 3 6


Exemplo
2 3 4 5 6 7 10
heap
3 2 ⇒ 2 3 ⇒ 2 3 4 5 6 7 10

Heapsort - Pseudo-código

Heapsort(A)
Entrada: Vetor A de n números inteiros.
Saída: Vetor A ordenado.

```
ConstroiHeap(A, n)
para tamanho de n decrescendo até 2 faça
    ▷ {troca elemento do topo do heap com o último}
    t := A[tamanho]; A[tamanho] := A[1]; A[1] := t
    ▷ {rearranja A para ter estrutura de heap}
    AjustaHeap(A, 1, tamanho)
```

Heapsort - Análise de Complexidade

- Logo, a complexidade da etapa de ordenação do *Heapsort* é:

$$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n.$$

- Portanto, na etapa de ordenação do *Heapsort* são efetuadas $O(n \log n)$ comparações e trocas no pior caso.
- Na verdade $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$
- No entanto, também temos que computar a complexidade de construção do *heap*.

Heapsort - Rearranjo - Pseudo-código

AjustaHeap(A, i, n)
Entrada: Vetor A de n números inteiros com estrutura de heap, exceto, talvez, pela subárvore de raiz i.
Saída: Vetor A com estrutura de heap.

```
se 2i ≤ n e A[2i] ≥ A[i]
    então maximo := 2i senão maximo := i
se 2i + 1 ≤ n e A[2i + 1] ≥ A[maximo]
    então maximo := 2i + 1
se maximo ≠ i então
    t := A[maximo]; A[maximo] := A[i]; A[i] := t
    AjustaHeap(A, maximo, n)
```

Heapsort - Construção do Heap - Top-down

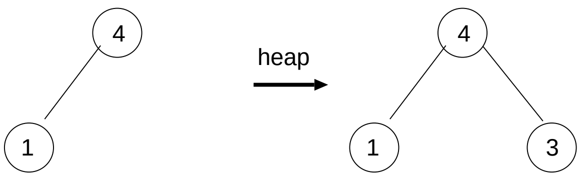
- Mas, como construímos o *heap* ?
- Se o trecho de 1 a i do vetor tem estrutura de *heap*, é fácil adicionar a folha $i + 1$ ao *heap* e em seguida rearranjá-lo, garantindo que o trecho de 1 a $i + 1$ tem estrutura de *heap*.
- Esta é a abordagem *top-down* para construção do *heap*.

Heapsort - Análise de Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na etapa de ordenação do algoritmo *Heapsort* ?
- A seleção e posicionamento do elemento máximo é feita em tempo constante.
- No pior caso, a função **AjustaHeap** efetua $\Theta(h)$ comparações e trocas, onde h é a altura do *heap* que contém os elementos que resta ordenar.
- Como o *heap* representa uma árvore binária completa, então $h \in \Theta(\log i)$, onde i é o número de elementos do *heap*.

Heap - Construção Top Down

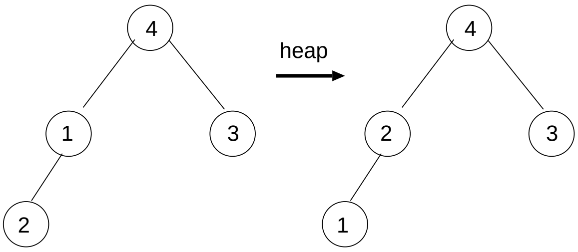
4	1	3	2	16	9	10	14	8	7



4	1	3	2	16	9	10	14	8	7

Heap - Construção Top Down

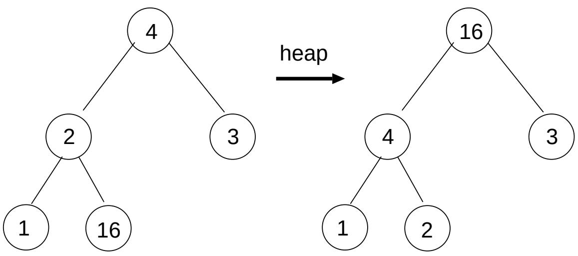
4	1	3	2	16	9	10	14	8	7



4	2	3	1	16	9	10	14	8	7

Heap - Construção Top Down

4	2	3	1	16	9	10	14	8	7



16	4	3	1	2	9	10	14	8	7

Vamos continuar...

Construção do Heap - Top-down

ConstroiHeap(A, n)
Entrada: Vetor A de n números inteiros.
Saída: Vetor A com estrutura de *heap*.

para i **de** 2 **até** n **faça**
    $v := A[i]$
    $j := i$
    enquanto $j > 1$ **e** $A[j \ div \ 2] < v$ **faça**
        $A[j] := A[j \ div \ 2]$
        $j := j \ div \ 2$
    $A[j] := v$

Top-down - Complexidade

- Quantas comparações e trocas são executadas no pior caso na construção do *heap* pela abordagem *top-down* ?
- O rearranjo do *heap* na iteração i efetua $\Theta(h)$ comparações e trocas no pior caso, onde h é a altura da árvore representada pelo trecho do *heap* de 1 a i. Logo, $h \in \Theta(\log i)$.
- Portanto, o número de comparações e trocas efetuadas construção do *heap* por esta abordagem é:

$$\sum_{i=1}^{n} \log i \in \Theta(n \log n).$$

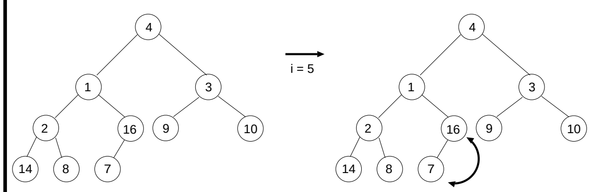
- $\Theta(n \log n)$ comparações e trocas no pior caso.

Construção do Heap - Bottom-up

- É possível construir o *heap* de forma mais eficiente.
- Suponha que o trecho de i a n do vetor é tal que, para todo j, $i \leq j \leq n$, a subárvore de raiz j representada por esse trecho do vetor tem estrutura de *heap*.
- Note que, em particular, o trecho de $\lfloor n/2 \rfloor + 1$ a n do vetor satisfaz a propriedade, pois inclui apenas folhas da árvore binária de n elementos.
- Podemos então executar **AjustaHeap**$(A, i-1, n)$, garantindo assim que o trecho de $i-1$ a n satisfaz a propriedade.
- Esta é a abordagem *bottom-up* para construção do *heap*.

Árvore Heap (construção)

4	1	3	2	16	9	10	14	8	7



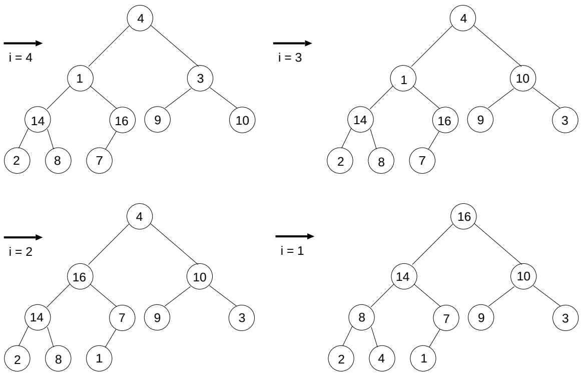
A construção de um heap máximo é realizada de acordo com os seguintes passos:

- Inicialmente considera que cada folha da árvore do heap principal a ser construído já é um heap, uma vez que as folhas não possuem filhos, atendendo necessariamente à propriedade do heap.

- O algoritmo começa processando o nó na posição $\lfloor (n/2) \rfloor$, pois este é o primeiro nó do heap (partindo-se dos níveis mais baixos para os mais altos, da direita para a esquerda) que possui algum filho e que pode não atender à propriedade do heap.

- A partir desta posição inicial ($\lfloor (n/2) \rfloor$), a repetição se aplica para cada nó percorrendo o vetor da direita para a esquerda e a árvore de baixo para cima.

Árvore Heap (construção)



Construção do Heap - Bottom-up - Pseudo-código

ConstroiHeap(A, n**)**
Entrada: Vetor A de n números inteiros.
Saída: Vetor A com estrutura de *heap*.

para i **de** n div 2 **decrescendo até** 1 **faça**
    $AjustaHeap(A, i, n)$

Bottom-up - Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na construção do *heap* pela abordagem *bottom-up* ?
- O rearranjo do *heap* na iteração i efetua $\Theta(h)$ comparações e trocas no pior caso, onde h é a altura da subárvore de raiz i.
- Seja $T(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- O número de operações efetuadas na construção do *heap* pela abordagem *bottom-up* é $T(\log n)$.

Bottom-up - Complexidade

- A expressão de $T(h)$ é dada pela recorrência:

$$T(h) = \begin{cases} 0, & h = 0 \\ 2T(h-1) + h, & h > 1, \end{cases}$$

- É possível provar (por indução) que $T(h) = 2^{h+1} - (h+2)$.
- Então, $T(\log n) \in \Theta(n)$ e a abordagem *bottom-up* para construção do *heap* apenas efetua $\Theta(n)$ comparações e trocas no pior caso.
- Ainda assim, a complexidade do *Heapsort* no pior caso é $\Theta(n \log n)$.

O método construção do *heap* descrito acima é o método *bottom-up*, no qual consideramos inicialmente que cada folha do heap é um heap por em si e construímos heaps de altura $h+1$ unindo dois *heaps* de altura h por uma raiz comum. Essa união exige que a função **AjustaHeap** seja executada para o nó raiz da nova subárvore para garantir que o novo *heap* tenha de fato a estrutura de *heap*.

Também poderíamos usar o método de construção *top-down*, no qual consideramos a inserção de um novo elemento por vez no *heap*, no final do vetor, seguida de uma operação de rearranjo do *heap*, semelhante a **AjustaHeap** para posicionar corretamente o elemento inserido. A única diferença no método de arranjo do *heap* é que, nesse caso, queremos reposicionar novo nó subindo no *heap*, enquanto que na função **AjustaHeap** estamos reposicionando o novo nó descendo no *heap*. No pior caso, a adição de um elemento dessa forma tem complexidade $O(\log n)$, onde n é o número de elementos no *heap* no momento da adição. Logo, a construção do *heap* dessa forma tem complexidade de pior caso

$$\sum_{i=1}^{n} \log i \in O(n \log n).$$

Do ponto de vista teórico ambas as formas de construção do *heap* são boas, pois a complexidade assintoticamente pior de construção *top-down* do *heap* não afeta a complexidade $O(n \log n)$ do algoritmo como um todo. No entanto, do ponto de vista prático, a implementação da construção *bottom-up* do *heap* é mais interessante.

Ordenação - Cota Inferior

- os algoritmos vistos têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos
- o resultado da comparação de x_i com x_j, $i \neq j$, define se x_i será posicionado antes ou depois de x_j
- todos os algoritmos dão uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o problema da ordenação.
- a menor cota superior é dada pelos algoritmos *Mergesort* e o *Heapsort*, que efetuam $\Theta(n \log n)$ comparações no pior caso.

Ordenação - Cota Inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente ?
- Veremos a seguir que não...
- É possível provar que qualquer algoritmo que ordena n elementos baseado apenas em comparações de elementos efetua no mínimo $\Omega(n \log n)$ comparações no pior caso.
- Para demonstrar esse fato, vamos representar os algoritmos de ordenação em um modelo computacional abstrato, denominado árvore (binária) de decisão.

Árvores de Decisão - Modelo Abstrato

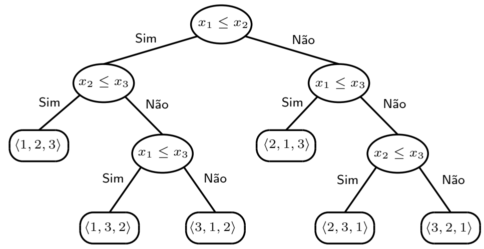
- Os nós internos de uma árvore de decisão representam comparações feitas pelo algoritmo.
- As subárvores de cada nó interno representam possibilidades de continuidade das ações do algoritmo após a comparação.
- Cada nó possui apenas duas subárvores. Tipicamente, as duas subárvores representam os caminhos a serem seguidos conforme o resultado (verdadeiro ou falso) da comparação efetuada.
- As folhas são as respostas possíveis após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Árvores de Decisão para a Ordenação

- **Problema da Ordenação:** Dado um conjunto de n inteiros $x_1, x_2, \ldots, x_n$, encontre uma permutação p dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.
- É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.
  - As ramificações representam os possíveis resultados da comparação: verdadeiro se $x_i \leq x_j$, ou falso se $x_i > x_j$.
  - As folhas representam possíveis soluções: as diferentes permutações dos n índices.

Árvores de Decisão para a Ordenação

Veja a árvore de decisão que representa o comportamento do *Insertion Sort* para um conjunto de 3 elementos:



Árvores de Decisão para a Ordenação

- Para um alg. de ordenação baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- A árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (duas seqüências distintas de decisões terminem no mesmo resultado).
- Pior caso – caminho mais longo da raiz a uma folha
- A altura mínima de uma árvore binária de decisão com pelo menos $n!$ folhas dá o número mínimo de comparações que o melhor algoritmo de ordenação baseado em comparações deve efetuar.

A Cota Inferior

- Qual a altura mínima, em função de n, de uma árvore binária de decisão com pelo menos $n!$ folhas ?
- Uma árvore binária de decisão T com altura h tem, no máximo, 2^h folhas.
- Portanto, se T tem pelo menos $n!$ folhas, $n! \leq 2^h$, ou seja, $h \geq \log_2 n!$.

- Mas,

$$\begin{aligned}
\log_2 n! &= \textstyle\sum_{i=1}^{n} \log i \\
&\geq \textstyle\sum_{i=\lceil n/2 \rceil}^{n} \log i \\
&\geq \textstyle\sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\
&\geq (n/2 - 1)\log n/2 \\
&= n/2 \log n - n/2 - \log n + 1 \\
&\geq n/4 \log n, \text{ para } n \geq 16.
\end{aligned}$$

- Então, $h \in \Omega(n \log n)$.

A demonstração de que $\log n! \in \Omega(n \log n)$ também pode ser feita da seguinte forma (supondo n par):

$$\begin{aligned}
\log(n!) &= \log \prod_{i=1}^{n} i \\
&= \sum_{i=1}^{n} \log i \\
&= \log 1 + \log 2 + \log 3 + \cdots + \log\left(\frac{n}{2}\right) + \\
&\quad + \log n + \log(n-1) + \log(n-2) + \cdots + \log\left(\frac{n}{2}+1\right) \\
&= \log n + \log(2n-2) + \log(3n-6) + \cdots + \log\left(\frac{n^2}{4}+\frac{n}{2}\right)
\end{aligned}$$

Se conseguirmos demonstrar que, para $n \geq n_0$, cada parcela dessa soma é maior ou igual a $c \log n$, onde c é uma constante positiva, a demonstração estará concluída. Ou seja, queremos demonstrar que, para algum $c > 0$,

$$\log k + \log(n-k+1) \geq c \log n,$$

para $1 \leq k \leq \frac{n}{2}$. Vejamos:

$$\begin{aligned}
\log k + \log(n-k+1) &= \log(nk - k^2 + k) \\
&\geq \log(nk - k^2) \\
&= \log(k(n-k)) \\
&\geq \log\left(\frac{n}{2}\right), \text{ pois } k \geq 1 \text{ e } n-k \geq \tfrac{n}{2} \\
&= \log n - 1 \\
&\geq \frac{\log n}{2}, \text{ para } n \geq 4
\end{aligned}$$

Portanto, a inequação é de fato verdadeira para $c = \frac{1}{2}$ e $n \geq 4$. Logo, temos que, para $n \geq n_0 = 4$,

$$\begin{aligned}
\sum_{i=1}^{n} \log i &\geq \sum_{i=1}^{\frac{n}{2}} \frac{\log n}{2} \\
&= \frac{1}{2}\sum_{i=1}^{\frac{n}{2}} \log n \\
&= \frac{1}{4}\sum_{i=1}^{n} \log n \\
&= \frac{1}{4} n \log n.
\end{aligned}$$

Observações finais

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o problema da ordenação.
- Portanto, os algoritmos *Mergesort* e *Heapsort* são algoritmos ótimos.

Animação

Vejam:

http://math.hws.edu/eck/js/sorting/xSortLab.html

Referências

- Cid Carvalho de Souza e Cândida Nunes da Silva, notas de aula da disciplina de MC448 — Análise de Algoritmos I, IC-Unicamp.
- Figuras retiradas do material de apoio do curso de Algoritmos e Estruturas de Dados do Instituto Superior Técnico
- N. Ziviani, *Projeto de algoritmos com implementação em pascal e C*. São Paulo, Ed. Pioneira, 2004.