MC448 — Análise de Algoritmos I

Cid Carvalho de Souza e Cândida Nunes da Silva

25 de Março de 2008

Ordenação

O Problema da Ordenação

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

O Problema da Ordenação

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

- Podemos projetar por indução diversos algoritmos para o problema da ordenação.

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.

Ordenação por indução: paradigma incremental

OrdenacaoIncremental(A, i, j)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
$n := j - i + 1$
se $n = 1$ **então**
    retorne
se não
    $<$ comandos iniciais $>$
    OrdenacaoIncremental($A, i, j-1$)   (ou ($A, i+1, j$))
    $<$ comandos finais $>$
fim se
retorne

Ordenação por indução: paradigma divisão-e-conquista

OrdenacaoD&C(A, i, j)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
$n := j - i + 1$
se $n = 1$ **então**
    retorne
se não
    $<$ comandos iniciais: a **divisão** $>$ (cálculo de q !)
    OrdenacaoD&C(A, i, q)
    OrdenacaoD&C($A, q+1, j$)
    $<$ comandos finais: a **conquista** $>$
fim se
retorne

Projeto por Indução Simples

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

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.

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.

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*.

Insertion Sort - Pseudo-código - Versão Recursiva

OrdenacaoInsercao(A, n)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
1. **se** $n \geq 2$ **faça**
2. OrdenacaoInsercao($A, n - 1$)
3. $v := A[n]$
4. $j := n - 1$
5. **enquanto** $(j > 0)$ e $(A[j] > v)$ **faça**
6. $A[j + 1] := A[j]$
7. $j := j - 1$
8. $A[j + 1] := v$

Insertion Sort - Pseudo-código - Versão Iterativa

- É fácil eliminar o uso de recursão simulando-a com um laço.

OrdenacaoInsercao(A)

```
▷ Entrada: Vetor A de n números inteiros.
▷ Saída: Vetor A ordenado.
para i := 2 até n faça
    v := A[i]
    j := i − 1
    enquanto (j > 0) e (A[j] > v) faça
        A[j + 1] := A[j]
        j := j − 1
    A[j + 1] := v
```


Insertion Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Insertion Sort* executa no pior caso ?

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:

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:

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

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.

Projeto por Indução Simples - Segunda Alternativa

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

Projeto por Indução Simples - 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.

Projeto por Indução Simples - 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.

Projeto por Indução Simples - 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 - Pseudo-código - 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 - Pseudo-código - Versão Iterativa

- Novamente, é fácil eliminar o uso de recursão simulando-a com um laço.

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** $min := j$
  $t := A[min]$
  $A[min] := A[i]$
  $A[i] := t$

Selection Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Selection Sort* executa no pior caso ?

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:

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:

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

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.

Selection Sort - Análise de Complexidade - Cont.

- Já o número de trocas é dado pela recorrência:

Selection Sort - Análise de Complexidade - Cont.

- Já o número de trocas é dado pela recorrência:

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

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.

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*.

Projeto por Indução Simples - Terceira Alternativa

Projeto por Indução Simples - Terceira Alternativa

- Ainda há uma terceira alternativa para o passo da indução.

Projeto por Indução Simples - Terceira Alternativa

- Ainda há uma terceira alternativa para o passo da indução.
- **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.

Projeto por Indução Simples - Terceira Alternativa

- Ainda há uma terceira alternativa para o passo da indução.
- **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*.

Projeto por Indução Simples - Terceira Alternativa

- Ainda há uma terceira alternativa para o passo da indução.
- **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*.

Bubble Sort - Pseudo-código - Versão Iterativa

BubbleSort(A)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.

1. **para** $i := n$ **decrescendo até** 1 **faça**
2. **para** $j := 2$ **até** i **faça**
3. **se** $A[j-1] > A[j]$ **então**
4. $t := A[j-1]$
5. $A[j-1] := A[j]$
6. $A[j] := t$

Bubble Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Bubble Sort* executa no pior caso ?

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:

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:

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

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.

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* !

Projeto por Indução Forte

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

Hipótese de Indução Forte:
Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.

Projeto por Indução Forte

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

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.

Projeto por Indução Forte

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

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.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.
- Na implementação do algoritmo, o conjunto S é um vetor de tamanho n.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.
- Na implementação do algoritmo, 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.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.
- Na implementação do algoritmo, 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.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.
- Na implementação do algoritmo, 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 - 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($A, m + 1, d$)

Mergesort - Pseudo-código (cont.)

OrdenacaoIntercalacao(A, e, d):cont.

▷ Copia o trecho de e a m para B.
para i **de** e **até** m **faça**
  $B[i] := A[i]$
▷ Copia o trecho de m a d para B em ordem reversa.
para j **de** $m + 1$ **até** d **faça**
  $B[d + m + 1 - j] := A[j]$
$i := e$; $j := d$
para k **de** e **até** d **faça**
  se $B[i] < B[j]$ **então**
    $A[k] := B[i]$
    $i := i + 1$
  se não
    $A[k] := B[j]$
    $j := j - 1$

Mergesort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Mergesort* executa no pior caso ?

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:

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:

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

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, $\Theta(n \log n)$ comparações e trocas são executadas no pior caso.

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, $\Theta(n \log n)$ comparações e trocas são executadas no pior caso.
- Ou seja, algoritmo *Mergesort* é assintoticamente mais eficiente que todos os anteriores.

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

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- Sim! Basta deslocarmos os elementos de um dos subvetores, quando necessário, para dar lugar ao mínimo dos dois subvetores.

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- 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:

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- 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

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- 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:

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

- Ou seja, a complexidade do *Mergesort* passa a ser $\Theta(n^2)$!

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- 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:

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

- 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*.

Projeto por Indução Forte - Segunda Alternativa

Hipótese de Indução Forte:
Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.

Projeto por Indução Forte - 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.

Projeto por Indução Forte - 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*.

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.

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 de escolhemos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.

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 de escolhemos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.
- 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.

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 de escolhemos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.
- 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 - Pseudo-código

Quicksort(A, e, d)

$\triangleright$ **Entrada:** Vetor A de números inteiros e os índices e e d que delimitam início e fim do subvetor a ser ordenado.
$\triangleright$ **Saída:** Subvetor de A de e a d ordenado.

se $d > e$ **então**
    $\triangleright$ escolhe o pivot
    $v := A[d]$
    $i := e - 1$
    $j := d$

Quicksort - Pseudo-código (cont.)

Quicksort(A, e, d): cont.

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 ?

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.

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:

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:

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

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:

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

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

Quicksort - Análise de Complexidade

- Então, o algoritmo *Quicksort* é assintoticamente menos eficiente que o *Mergesort* no pior caso.

Quicksort - Análise de Complexidade

- Então, o algoritmo *Quicksort* é assintoticamente menos eficiente que o *Mergesort* no pior caso.
- Veremos a seguir que, no caso médio, o *Quicksort* efetua $\Theta(n \log n)$ comparações e trocas.

Quicksort - Análise de Complexidade

- Então, o algoritmo *Quicksort* é assintoticamente menos eficiente que o *Mergesort* no pior caso.
- Veremos a seguir que, 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.

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:

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:

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

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:

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

- Não é difícil ver que:

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:

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

- Não é difícil ver 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:

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

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 $T(n) \in \Theta(n \log n)$.

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.

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.

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*.

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 $\lfloor i/2 \rfloor$.

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.

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.

Heapsort - O Algoritmo

Heapsort - O Algoritmo

- Então, o uso da estrutura *heap* permite que:

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.

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 - 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.
- O algoritmo *Heapsort* consiste então da construção de um *heap* com os elementos a serem ordenados, seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do *heap*.

Heapsort - Pseudo-código

Heapsort(A)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
1. *ConstroiHeap*(A, n)
2. **para** *tamanho* **de** n **decrescendo até** 2 **faça**
   ▷ troca elemento do topo do heap com o último
3. $t := A[tamanho]$; $A[tamanho] := A[1]$; $A[1] := t$
   ▷ rearranja A para ter estrutura de heap
4. *AjustaHeap*$(A, 1, tamanho)$

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.
1. **se** $2i \leq n$ e $A[2i] \geq A[i]$
2. **então** $maximo := 2i$ **se não** $maximo := i$
3. **se** $2i + 1 \leq n$ e $A[2i+1] \geq A[maximo]$
4. **então** $maximo := 2i + 1$
5. **se** $maximo \neq i$ **então**
6. $t := A[maximo]$; $A[maximo] := A[i]$; $A[i] := t$
7. *AjustaHeap*$(A, maximo, n)$

Heapsort - Análise de Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na etapa de ordenação do algoritmo *Heapsort* ?

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.

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.

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*.

Heapsort - Análise de Complexidade

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

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

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.

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 $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$, conforme visto em exercício de lista !

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 $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$, conforme visto em exercício de lista !
- No entanto, também temos que computar a complexidade de construção do *heap*.

Heapsort - Construção do *Heap* - *Top-down*

- Mas, como construímos o *heap* ?

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*.

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*.

Construção do *Heap* - *Top-down* - Pseudo-código

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**
        $A[j] := A[j \; div \; 2]$
        $j := j \; div \; 2$
    $A[j] := v$

Construção do *Heap* - *Top-down* - Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na construção do *heap* pela abordagem *top-down* ?

Construção do *Heap* - *Top-down* - Complexidade

- Quantas comparações e quantas 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, $h \in \Theta(\log i)$.

Construção do *Heap* - *Top-down* - Complexidade

- Quantas comparações e quantas 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:

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

Construção do *Heap* - *Top-down* - Complexidade

- Quantas comparações e quantas 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:

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

- Então, o algoritmo *Heapsort* efetua ao todo $\Theta(n \log n)$ comparações e trocas no pior caso.

Heapsort - Construção do *Heap* - *Bottom-up*

- É possível construir o *heap* de forma mais eficiente.

Heapsort - 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*.

Heapsort - 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.

Heapsort - 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.

Heapsort - 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*.

Construção do *Heap* - *Bottom-up* - Pseudo-código

ConstroiHeap(A, n)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A com estrutura de *heap*.
1. **para** i **de** n div 2 **decrescendo até** 1 **faça**
2. *AjustaHeap*(A, i, n)

Construção do *Heap* - *Bottom-up* - Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na construção do *heap* pela abordagem *bottom-up* ?

Construção do *Heap* - *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.

Construção do *Heap* - *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.

Construção do *Heap* - *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 $T(\log n)$.

Construção do *Heap* - *Bottom-up* - Complexidade

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

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

Construção do *Heap* - *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 $T(h) = 2^{h+1} - (h+2)$.

Construção do *Heap* - *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.

Construção do *Heap* - *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 problema da Ordenação - Cota Inferior

- Estudamos diversos algoritmos para o problema da ordenação.

O problema da Ordenação - Cota Inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.

O problema da Ordenação - Cota Inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, 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 no conjunto ordenado.

O problema da Ordenação - Cota Inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, 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 no conjunto ordenado.
- Todos os algoritmos dão uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o problema da ordenação.

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, 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 no conjunto ordenado.
- 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.

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente ?

O problema da Ordenação - Cota Inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente ?
- Veremos a seguir que não...

O problema da 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.

O problema da 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.

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.

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.
- No caso das árvores binárias de decisão, 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.

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.
- No caso das árvores binárias de decisão, 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 do algoritmo após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da 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 $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da 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 da seguinte forma:

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da 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 da seguinte forma:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I — versão 2

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da 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 da seguinte forma:
  - 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 $x_i > x_j$.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I — versão 2

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da ordenação:

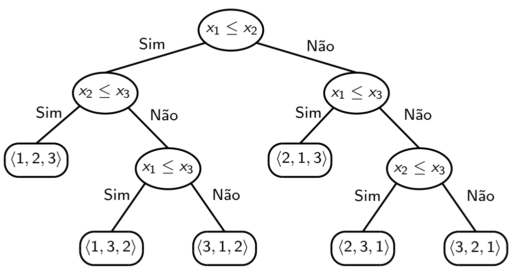
Problema da Ordenação:
Dado um conjunto de n inteiros $x_1, x_2, \ldots, x_n$, encontre uma permutação p dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

- É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão da seguinte forma:
  - 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.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I — versão 2

Árvores de Decisão para o Problema da Ordenação

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



Árvores de Decisão para o Problema da Ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.

Árvores de Decisão para o Problema da Ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas seqüências distintas de decisões terminem no mesmo resultado).

Árvores de Decisão para o Problema da Ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas seqüências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.

Árvores de Decisão para o Problema da Ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas seqüências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.
- 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 ?

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.

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, $h \geq \log_2 n!$.

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,

$$\log_2 n! = \sum_{i=1}^{n} \log i$$

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,

$$\begin{aligned} \log_2 n! &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \end{aligned}$$

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,

$$\begin{aligned} \log_2 n! &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \end{aligned}$$

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,

$$\begin{aligned} \log_2 n! &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\ &\geq (n/2 - 1) \log n/2 \end{aligned}$$

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,

$$\begin{aligned} \log_2 n! &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\ &\geq (n/2-1)\log n/2 \\ &= n/2 \log n - n/2 - \log n + 1 \end{aligned}$$

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,

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

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,

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

- Então, $h \in \Omega(n \log n)$.

Observações finais

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o problema da ordenação.

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.