

## MC448 — Análise de Algoritmos I

Cid Carvalho de Souza Cândia Nunes da Silva  
Orlando Lee

9 de outubro de 2009

### Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade  $O(n)$ :

- **Counting Sort:** Elementos são números inteiros “pequenos”; mais precisamente, inteiros  $x \in O(n)$ .
- **Radix Sort:** Elementos são números inteiros de comprimento máximo constante, isto é, independente de  $n$ .
- **Bucket Sort:** Elementos são números reais uniformemente distribuídos no intervalo  $[0..1)$ .

### Ordenação em Tempo Linear

### Counting Sort

- Considere o problema de ordenar um vetor  $A[1 \dots n]$  de inteiros quando se sabe que todos os inteiros estão no intervalo entre 0 e  $k$ .
- Podemos ordenar o vetor simplesmente contando, para cada inteiro  $i$  no vetor, quantos elementos do vetor são menores que  $i$ .
- É exatamente o que faz o algoritmo *Counting Sort*.

## Counting Sort

COUNTING-SORT( $A, B, n, k$ )

```
1 para  $i \leftarrow 0$  até  $k$  faça
2    $C[i] \leftarrow 0$ 

3 para  $j \leftarrow 1$  até  $n$  faça
4    $C[A[j]] \leftarrow C[A[j]] + 1$ 
   ▷  $C[i]$  é o número de  $j$ s tais que  $A[j] = i$ 

5 para  $i \leftarrow 1$  até  $k$  faça
6    $C[i] \leftarrow C[i] + C[i - 1]$ 
   ▷  $C[i]$  é o número de  $j$ s tais que  $A[j] \leq i$ 

7 para  $j \leftarrow n$  decrescendo até 1 faça
8    $B[C[A[j]]] \leftarrow A[j]$ 
9    $C[A[j]] \leftarrow C[A[j]] - 1$ 
```

## Algoritmos *in-place* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é *in-place* se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- **Exemplos:** QUICKSORT e HEAPSORT são métodos de ordenação *in-place*. Note entretanto que o QUICKSORT requer memória adicional  $\omega(\lg n)$  devido à pilha de recursão. MERGESORT e COUNTING-SORT não são métodos *in-place*.
- Um método de ordenação é *estável* se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- **Exemplos:** COUNTING-SORT e QUICKSORT são exemplos de métodos *estáveis* (desde que certos cuidados sejam tomados na implementação). HEAPSORT não é *estável*.

## Counting Sort - Complexidade

- Qual a complexidade do algoritmo COUNTING-SORT?
- O algoritmo não faz comparações entre elementos de  $A$ !
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, a complexidade de COUNTING-SORT é  $O(n + k)$ . Quando  $k \in O(n)$ , ele tem complexidade  $O(n)$ .

Há algo de errado com o limite inferior de  $\Omega(n \log n)$  para ordenação?

## Radix Sort

- Considere agora o problema de ordenar um vetor  $A[1 \dots n]$  inteiros quando se sabe que todos os inteiros podem ser representados com apenas  $d$  dígitos, onde  $d$  é uma constante.
- Por exemplo, os elementos de  $A$  podem ser CEPs, ou seja, inteiros de 8 dígitos.

## Radix Sort

- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
  - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito **mais significativo**.
  - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os  $d - 1$  dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

## Radix Sort

- Podemos evitar o uso excessivo de memória adicional começando pelo dígito **menos significativo**.
- É isso o que faz o algoritmo **Radix Sort**.
- Para que **Radix Sort** funcione corretamente, ele deve usar um método de ordenação **estável**.
- Por exemplo, o **COUNTING-SORT**.

## Radix Sort

Suponha que os elementos do vetor  $A$  a ser ordenado sejam números inteiros de até  $d$  dígitos. O **Radix Sort** é simplesmente:

**RADIX-SORT**( $A, n, d$ )

- 1 **para**  $i \leftarrow 1$  até  $d$  **faça**
- 2 Ordene  $A[1 \dots n]$  pelo  $i$ -ésimo dígito menos significativo usando um método **estável**

## Radix Sort - Exemplo

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	→ 657	→ 355	→ 657
720	329	457	720
355	839	657	839
	↑	↑	↑

## Radix Sort - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- **Hipótese de indução:** os números estão ordenados com relação aos  $i - 1$  dígitos menos significativos.
- O que acontece ao ordenarmos pelo  $i$ -ésimo dígito?
- Se dois números têm  $i$ -ésimo dígitos distintos, o de menor  $i$ -ésimo dígito aparece antes do de maior  $i$ -ésimo dígito.
- Se ambos possuem o mesmo  $i$ -ésimo dígito, então a ordem dos dois também estará correta pois o método de ordenação é **estável** e, pela **HI**, os dois elementos já estavam ordenados segundo os  $i - 1$  dígitos menos significativos.

## Radix Sort - Complexidade

- Em contraste, um algoritmo por comparação como o **MERGESORT** teria complexidade  $\Theta(n \lg n)$ .
- Assim, **RADIX-SORT** é mais vantajoso que **MERGESORT** quando  $d < \lg n$ , ou seja, o número de dígitos for menor que  $\lg n$ .
- Se  $n$  for um **limite superior** para o maior valor a ser ordenado, então  $O(\lg n)$  é uma estimativa para a quantidade de **dígitos** dos números.
- Isso significa que não há diferença significativa entre o desempenho do **MERGESORT** e do **RADIX-SORT**?

## Radix Sort - Complexidade

- Qual é a complexidade de **RADIX-SORT**?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito.
- Se essa complexidade for  $\Theta(f(n))$ , obtemos uma complexidade total de  $\Theta(d f(n))$ .
- Se o algoritmo estável for, como o **COUNTING-SORT**, obtemos a complexidade  $\Theta(d(n + k))$ .
- Se  $k \in O(n)$ , isto resulta em uma complexidade linear em  $n$ .

E o limite inferior de  $\Omega(n \log n)$  para ordenação?

## Radix Sort - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que interpretamos os dígitos.
- A vantagem de se usar **RADIX-SORT** fica evidente quando interpretamos os **dígitos de forma mais geral** que simplesmente **0..9**.
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de  $n = 2^{20}$  números de **64 bits**. Então, **MERGESORT** faria cerca de  $n \lg n = 20 \times 2^{20}$  comparações e usaria um vetor auxiliar de tamanho  $2^{20}$ .

## Radix Sort - Complexidade

- Agora suponha que interpretamos cada número como tendo  $d = 4$  dígitos em base  $k = 2^{16}$ , e usarmos **RADIX-SORT** com o *Counting Sort* como método estável.

Então a complexidade de tempo seria da ordem de  $d(n+k) = 4(2^{20} + 2^{16})$  operações, bem menor que  $20 \times 2^{20}$  do **MERGESORT**. Mas, note que utilizamos dois vetores auxiliares, de tamanhos  $2^{16}$  e  $2^{20}$ .

- Se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
- Note que é possível usar o *Radix Sort* para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma  $d$ -upla ordenada de itens comparáveis.

## Bucket Sort

- Supõe que os  $n$  elementos da entrada estão **distribuídos uniformemente** no intervalo  $[0, 1)$ .
- A ideia é dividir o intervalo  $[0, 1)$  em  $n$  segmentos de mesmo tamanho (*buckets*) e distribuir os  $n$  elementos nos seus respectivos segmentos. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

## Bucket Sort - Pseudocódigo

**BUCKETSORT**( $A, n$ )

- 1 **para**  $i \leftarrow 1$  **até**  $n$  **faça**
- 2     insira  $A[i]$  na lista ligada  $B[\lfloor nA[i] \rfloor]$
- 3 **para**  $i \leftarrow 0$  **até**  $n - 1$  **faça**
- 4     ordene a lista  $B[i]$  com **INSERTION-SORT**
- 5 Concatene as listas  $B[0], B[1], \dots, B[n - 1]$

## Bucket Sort - Exemplo

	1	.78	0	
	2	.17	1	.12, .17
	3	.39	2	.21, .23, .26
	4	.26	3	.39
	5	.72	4	
A =	6	.94	5	
	7	.21	6	.68
	8	.12	7	.72, .78
	9	.23	8	
	10	.68	9	.94

## Bucket Sort - Corretude

- Dois elementos  $x$  e  $y$  de  $A$ ,  $x < y$ , ou terminam na mesma lista ou são colocados em listas diferentes  $B[i]$  e  $B[j]$ .
- A primeira possibilidade implica que  $x$  aparecerá antes de  $y$  na concatenação final, já que cada lista é ordenada.
- No segundo caso, como  $x < y$ , segue que  $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$ . Como  $i \neq j$ , temos  $i < j$ . Assim,  $x$  aparecerá antes de  $y$  na lista final.

## Estatísticas de Ordem

## Bucket Sort - Complexidade

- É claro que o pior caso do *Bucket Sort* é quadrático, supondo-se que as ordenações das listas seja feita com ordenação por inserção.
- Entretanto, o tempo esperado é linear. Intuitivamente, a idéia da demonstração é que, como os  $n$  elementos estão distribuídos uniformemente no intervalo  $[0, 1)$ , então o tamanho esperado das listas é pequeno.
- Portanto, as ordenações das  $n$  listas  $B[i]$  leva tempo total esperado  $\Theta(n)$ .
- Os detalhes podem ser vistos no livro de CLRS.

## Estatísticas de Ordem (Problema da Seleção)

- Estamos interessados em resolver o  
**Problema da Seleção:**  
Dado um conjunto  $A$  de  $n$  números reais e um inteiro  $i$ , determinar o  $i$ -ésimo menor elemento de  $A$ .
- Casos particulares importantes:
  - Mínimo :  $i = 1$
  - Máximo :  $i = n$
  - Mediana :  $i = \lfloor \frac{n+1}{2} \rfloor$  (mediana inferior)
  - Mediana :  $i = \lceil \frac{n+1}{2} \rceil$  (mediana superior)

## Mínimo

Recebe um vetor  $A[1 \dots n]$  e devolve o **mínimo** do vetor.

```
MÍNIMO( $A, n$ )
1  mín ←  $A[1]$ 
2  para  $j \leftarrow 2$  até  $n$  faça
3      se mín >  $A[j]$ 
4          então mín ←  $A[j]$ 
5  devolva mín
```

Número de comparações:  $n - 1 = \Theta(n)$

## Mínimo e máximo

- Processe os elementos em pares. Para cada par, compare o menor com o mínimo atual e o maior com o máximo atual. Isto resulta em 3 comparações para cada 2 elementos.
- Se  $n$  for ímpar, inicialize o mínimo e o máximo como sendo o primeiro elemento.
- Se  $n$  for par, inicialize o mínimo e o máximo comparando os dois primeiros elementos.

Número de comparações:

```
3⌊ $n/2$ ⌋   se  $n$  for ímpar
3⌊ $n/2$ ⌋ - 2 se  $n$  for par
```

Pode-se mostrar que isto é o melhor possível.  
(Exercício \* do CLRS)

## Mínimo e máximo

Recebe um vetor  $A[1 \dots n]$  e devolve o **mínimo** e o **máximo** do vetor.

```
MINMAX( $A, n$ )
1  mín ← máx ←  $A[1]$ 
2  para  $j \leftarrow 2$  até  $n$  faça
3      se  $A[j] < mín$ 
4          então mín ←  $A[j]$ 
5      se  $A[j] > máx$ 
6          então máx ←  $A[j]$ 
7  devolva (mín, máx)
```

Número de comparações:  $2(n - 1) = 2n - 2 = \Theta(n)$

É possível fazer melhor!

## Problema da Seleção – primeira solução

Recebe  $A[1 \dots n]$  e  $i$  tal que  $1 \leq i \leq n$  e devolve o  $i$ -ésimo menor elemento de  $A[1 \dots n]$

```
SELECT-ORD( $A, n, i$ )
1  ORDENE( $A, n$ )
2  devolva  $A[i]$ 
```

ORDENE pode ser *MergeSort* ou *HeapSort*.

A complexidade de tempo de SELECT-ORD é  $O(n \lg n)$ .

Será que não dá para fazer melhor que isso?

Afinal, consigo achar o mínimo e/ou máximo em tempo  $O(n)$ .

## Relembrando – Partição

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

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

Entrada:

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

Saída:

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

## Relembrando – Particione

Reorganiza  $A[p \dots r]$  de modo que  $p \leq q \leq r$  e  $A[p \dots q - 1] \leq A[q] < A[q + 1 \dots r]$ .

**PARTICIONE**( $A, p, r$ )

```
1  $x \leftarrow A[r]$  ▷  $x$  é o "pivô"
2  $i \leftarrow p - 1$ 
3 para  $j \leftarrow p$  até  $r - 1$  faça
4     se  $A[j] \leq x$ 
5         então  $i \leftarrow i + 1$ 
6          $A[j] \leftrightarrow A[i]$ 
7  $A[i + 1] \leftrightarrow A[r]$ 
8 devolva  $i + 1$ 
```

## Problema da Seleção – segunda solução

Suponha que queremos achar o  $i$ -ésimo menor de  $A[1 \dots n]$ .

- Executamos **PARTICIONE** e este reorganiza o vetor e devolve um índice  $k$  tal que

$$A[1 \dots k - 1] \leq A[k] < A[k + 1 \dots n].$$

- Eis a idéia do algoritmo:
  - Se  $i = k$ , então o pivô  $A[k]$  é o  $i$ -ésimo menor! (Yeesss!)
  - Se  $i < k$ , então o  $i$ -ésimo menor está em  $A[1 \dots k - 1]$ ;
  - Se  $i > k$ , então o  $i$ -ésimo menor está em  $A[k + 1 \dots n]$ .

## Problema da Seleção – segunda solução

Recebe  $A[p \dots r]$  e  $i$  tal que  $1 \leq i \leq r - p + 1$  e devolve o  $i$ -ésimo menor elemento de  $A[p \dots r]$ .

**SELECT-NL**( $A, p, r, i$ )

```
1 se  $p = r$ 
2     então devolva  $A[p]$ 
3  $q \leftarrow$  PARTICIONE( $A, p, r$ )
4  $k \leftarrow q - p + 1$ 
5 se  $i = k$  ▷ pivô é o  $i$ -ésimo menor!
6     então devolva  $A[q]$ 
7     senão se  $i < k$ 
8         então devolva SELECT-NL( $A, p, q - 1, i$ )
9         senão devolva SELECT-NL( $A, q + 1, r, i - k$ )
```

## Segunda solução – complexidade

$\text{SELECT-NL}(A, p, r, i)$	Tempo
1 se $p = r$	?
2 então devolva $A[p]$	?
3 $q \leftarrow \text{PARTICIONE}(A, p, r)$	?
4 $k \leftarrow q - p + 1$	?
5 se $i = k$	?
6 então devolva $A[q]$	?
7 senão se $i < k$	?
8 então devolva $\text{SELECT-NL}(A, p, q - 1, i)$	?
9 senão devolva $\text{SELECT-NL}(A, q + 1, r, i - k)$	?

$T(n)$  = complexidade de tempo no pior caso quando  $n = r - p + 1$

## Segunda solução – complexidade

- A complexidade de  $\text{SELECT-NL}$  no pior caso é  $\Theta(n^2)$ .
- Então é melhor usar  $\text{SELECT-ORD}$ ?
- Não,  $\text{SELECT-NL}$  é muito eficiente na prática.
- Vamos mostrar que no caso médio  $\text{SELECT-NL}$  tem complexidade  $O(n)$ .

## Segunda solução – complexidade

$\text{SELECT-NL}(A, p, r, i)$	Tempo
1 se $p = r$	$\Theta(1)$
2 então devolva $A[p]$	$O(1)$
3 $q \leftarrow \text{PARTICIONE}(A, p, r)$	$\Theta(n)$
4 $k \leftarrow q - p + 1$	$\Theta(1)$
5 se $i = k$	$\Theta(1)$
6 então devolva $A[q]$	$O(1)$
7 senão se $i < k$	$O(1)$
8 então devolva $\text{SELECT-NL}(A, p, q - 1, i)$	$T(k - 1)$
9 senão devolva $\text{SELECT-NL}(A, q + 1, r, i - k)$	$T(n - k)$

$T(n) = \max\{T(k - 1), T(n - k)\} + \Theta(n)$

$T(n) \in \Theta(n^2)$  (Exercício)

## SELECT aleatorizado

O pior caso do  $\text{SELECT-NL}$  ocorre devido a uma escolha infeliz do pivô.

Um modo de evitar isso é usar aleatoriedade (como no  $\text{QUICKSORT-ALEATÓRIO}$ ).

```

PARTICIONE-ALEATÓRIO(A, p, r)
1  j ← RANDOM(p, r)
2  A[j] ↔ A[r]
3  devolva PARTICIONE(A, p, r)
    
```

## Algoritmo SELECT-ALEAT

Recebe  $A[p \dots r]$  e  $i$  tal que  $1 \leq i \leq r - p + 1$  e devolve o  $i$ -ésimo menor elemento de  $A[p \dots r]$

```
SELECT-ALEAT( $A, p, r, i$ )
1  se  $p = r$ 
2    então devolva  $A[p]$ 
3   $q \leftarrow$  PARTICIONE-ALEATÓRIO( $A, p, r$ )
4   $k \leftarrow q - p + 1$ 
5  se  $i = k$   $\triangleright$  pivô é o  $i$ -ésimo menor
6    então devolva  $A[q]$ 
7  senão se  $i < k$ 
8    então devolva SELECT-ALEAT( $A, p, q - 1, i$ )
9  senão devolva SELECT-ALEAT( $A, q + 1, r, i - k$ )
```

## Análise do caso médio

Recorrência para o caso médio de SELECT-ALEAT.

$T(n)$  = complexidade de tempo médio de SELECT-ALEAT.

$$T(0) = \Theta(1)$$

$$T(1) = \Theta(1)$$

$$T(n) \leq \frac{1}{n} \sum_{k=1}^n T(\max\{k-1, n-k\}) + \Theta(n).$$

$T(n)$  é  $\Theta(???)$ .

## Análise do caso médio

$$\begin{aligned} T(n) &\leq \frac{1}{n} \sum_{k=1}^n T(\max\{k-1, n-k\}) + an \\ &\leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + an \end{aligned}$$

pois

$$\max\{k-1, n-k\} = \begin{cases} k-1 & \text{se } k > \lceil n/2 \rceil, \\ n-k & \text{se } k \leq \lceil n/2 \rceil. \end{cases}$$

Se  $n$  é par, cada termo de  $T(\lceil n/2 \rceil)$  a  $T(n-1)$  aparece exatamente duas vezes na somatória.

Se  $n$  é ímpar, esses termos aparecem duas vezes e  $T(\lfloor n/2 \rfloor)$  aparece uma vez.

## Demonstração: $T(n) \leq cn$

$$T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + an$$

$$\stackrel{\text{hi}}{\leq} \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + an$$

$$= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \right) + an$$

$$= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lceil n/2 \rceil - 1)\lceil n/2 \rceil}{2} \right) + an$$

## Demonstração: $T(n) \leq cn$

$$\begin{aligned}
 T(n) &= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\
 &\leq \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2 - 2)(n/2 - 1)}{2} \right) + an \\
 &= \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
 &\leq \frac{3cn}{4} + \frac{c}{2} + an \\
 &= cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right) \leq cn.
 \end{aligned}$$

Isto funciona se  $c > 4a$  e  $n \geq 2c/(c - 4a)$ .  
Logo,  $T(n) = O(n)$ .

## Problema da Seleção – terceira solução

### Problema da Seleção:

Dado um conjunto  $A$  de  $n$  números reais e um inteiro  $i$ , determinar o  $i$ -ésimo menor elemento de  $A$ .

Veremos um **algoritmo linear** para o Problema da Seleção.

**BFPRT** = Blum, Floyd, Pratt, Rivest e Tarjan

Para simplificar a exposição, vamos supor que os elementos em  $A$  são distintos.

## Conclusão

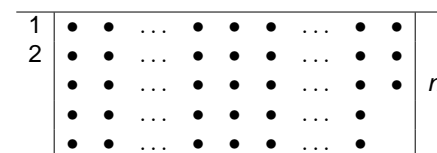
A complexidade de tempo de **SELECT-ALEAT** no **caso médio** é  $O(n)$ .

Na verdade,

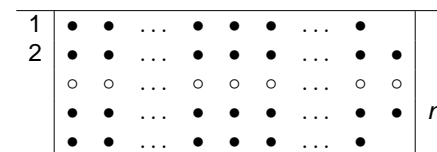
A complexidade de tempo de **SELECT-ALEAT** no **caso médio** é  $\Theta(n)$ .

## Problema da Seleção – terceira solução

- Divida os  $n$  elementos em  $\lfloor \frac{n}{5} \rfloor$  subconjuntos de 5 elementos e um subconjunto de  $n \bmod 5$  elementos.



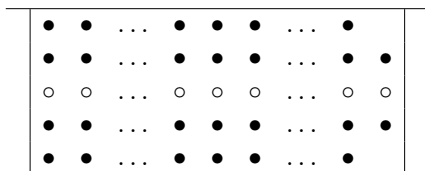
- Encontre a **mediana** de cada um dos  $\lfloor \frac{n}{5} \rfloor$  subconjuntos.



Na figura acima, cada subconjunto está em ordem crescente, de cima para baixo.

## Problema da Seleção – terceira solução

- 3 Determine, recursivamente, a **mediana  $x$**  das **medianas** dos subconjuntos de no máximo 5 elementos.



A figura acima é a mesma que a anterior, com as colunas ordenadas pela mediana de cada grupo. A ordem dos elementos em cada coluna permanece inalterada. Por simplicidade de exposição, supomos que a última coluna permanece no mesmo lugar.

**Note que o algoritmo não ordena as medianas!**

## Problema da Seleção - terceira solução

- 4 Usando  $x$  como pivô, particione o conjunto original  $A$  criando dois subconjuntos  $A_<$  e  $A_>$ , onde
- $A_<$  contém os elementos  $< x$  e
  - $A_>$  contém os elementos  $> x$ .

Se a posição final de  $x$  após o particionamento é  $k$ , então  $|A_<| = k - 1$  e  $|A_>| = n - k$ .

## Problema da Seleção - terceira solução

- 5 Finalmente, para encontrar o  $i$ -ésimo menor elemento do conjunto, compare  $i$  com a posição  $k$  de  $x$  após o particionamento:
- Se  $i = k$ ,  $x$  é o elemento procurado;
  - Se  $i < k$ , então determine recursivamente o  $i$ -ésimo menor elemento do subconjunto  $A_<$ ;
  - Senão, determine recursivamente o  $(i - k)$ -ésimo menor elemento do subconjunto  $A_>$ .

Note que esta parte é idêntica ao feito em **SELECT-NL** e em **SELECT-ALEAT**. O que diferencia este algoritmo dos outros é a escolha do **pivô**. Escolhendo-se a mediana das medianas, vamos poder garantir que nenhum dos lados é muito “grande”.

## Terceira solução – complexidade

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

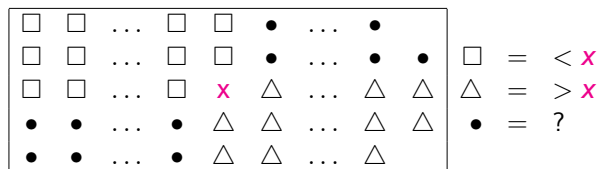
- 1 Divisão em subconjuntos de 5 elementos.  $\Theta(n)$
- 2 Encontrar a mediana de cada subconjunto.  $\Theta(n)$
- 3 Encontrar  $x$ , a mediana das medianas.  $T(\lceil n/5 \rceil)$
- 4 Particionamento com pivô  $x$ .  $O(n)$
- 5 Encontrar o  $i$ -ésimo menor de  $A_<$   $T(k - 1)$   
**OU** encontrar o  $i - k$ -ésimo menor de  $A_>$ .  $T(n - k)$

Temos então a recorrência

$$T(n) = T(\lceil n/5 \rceil) + T(\max\{k - 1, n - k\}) + \Theta(n)$$

## Terceira Solução - Complexidade

O diagrama abaixo classifica os elementos da última figura.



Veja que o número de elementos > x, isto é △s, é no mínimo  $\frac{3n}{10} - 6$ .

Isto porque no mínimo  $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$  grupos contribuem com 3 elementos > x, exceto possivelmente o último e aquele que contém x. Portanto,  $3 \left( \lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right) \geq \frac{3n}{10} - 6$ .

## Solução da recorrência: $T(n) \leq cn$

$$\begin{aligned}
 T(n) &\leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + an \\
 &\stackrel{hi}{\leq} c\lceil n/5 \rceil + c(\lfloor 7n/10 \rfloor + 6) + an \\
 &\leq c(n/5 + 1) + c(7n/10 + 6) + an \\
 &= 9cn/10 + 7c + an \\
 &= cn + (-cn/10 + 7c + an) \\
 &\leq cn,
 \end{aligned}$$

Quero que  $(-cn/10 + 7c + an) \leq 0$ .

Isto equivale a  $c \geq 10a(n/(n-70))$  quando  $n > 70$ . Como  $n > 140$ , temos  $n/(n-70) \leq 2$  e assim basta escolher  $c \geq 20a$ .

## Terceira Solução - Complexidade

Da mesma forma, o número de elementos < x, isto é □s, é no mínimo  $\frac{3n}{10} - 6$ .

Assim, no passo 5 do algoritmo,

$$\max\{k-1, n-k\} \leq n - \left( \frac{3n}{10} - 6 \right) \leq \frac{7n}{10} + 6.$$

A recorrência  $T(n)$  está agora completa:

$$T(n) \leq \begin{cases} \Theta(1), & n \leq 140 \\ T(\lceil n/5 \rceil) + T(\lfloor 7n/10 \rfloor + 6) + \Theta(n), & n > 140, \end{cases}$$

140 é um “número mágico” que faz as contas funcionarem...

A solução é  $T(n) \in \Theta(n)$

## Algoritmo SELECT

Recebe  $A[p \dots r]$  e  $i$  tal que  $1 \leq i \leq r-p+1$  e devolve um índice  $q$  tal que  $A[q]$  é o  $i$ -ésimo menor elemento de  $A[p \dots r]$ .

SELECT( $A, p, r, i$ )

```

1 se  $p = r$ 
2   então devolva  $p$  ▷  $p$  e não  $A[p]$ 
3  $q \leftarrow$  PARTICIONE-BFPRT( $A, p, r$ )
4  $k \leftarrow q - p + 1$ 
5 se  $i = k$ 
6   então devolva  $q$  ▷  $q$  e não  $A[q]$ 
7   senão se  $i < k$ 
8     então devolva SELECT( $A, p, q - 1, i$ )
9     senão devolva SELECT( $A, q + 1, r, i - k$ )
    
```

## PARTICIONE-BFPRT

Rearranja  $A[p \dots r]$  e devolve um índice  $q$ ,  $p \leq q \leq r$ , tal que  $A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$  e

$$\max\{k-1, n-k\} \leq \left\lfloor \frac{7n}{10} \right\rfloor + 6,$$

onde  $n = r - p + 1$  e  $k = q - p + 1$ .

## PARTICIONE-BFPRT

- Divida o vetor em  $\lfloor n/5 \rfloor$  grupos de tamanho 5 e um grupo  $\leq 5$ ,
- ordene cada grupo e determine a mediana de cada um deles,
- determine a mediana das medianas chamando **SELECT** (!!)
- e particione o vetor em torno desse valor.

## PARTICIONE-BFPRT

**PARTICIONE-BFPRT**( $A, p, r$ )  $\triangleright n := r - p + 1$

- 1 **para**  $j \leftarrow p, p+5, p+5 \cdot 2, \dots$  **até**  $p+5(\lceil n/5 \rceil - 1)$  **faça**
- 2     **ORDENE**( $A, j, j+4$ )
- 3     **ORDENE**( $A, p+5\lfloor n/5 \rfloor, n$ )
- 4 **para**  $j \leftarrow 1$  **até**  $\lceil n/5 \rceil - 1$  **faça**
- 5      $A[j] \leftrightarrow A[p+5j-3]$
- 6      $A[\lceil n/5 \rceil] \leftrightarrow A[(p+5\lfloor n/5 \rfloor + n)/2]$
- 7      $k \leftarrow$  **SELECT**( $A, p, p+\lceil n/5 \rceil - 1, \lfloor (\lceil n/5 \rceil + 1)/2 \rfloor$ )
- 8      $A[k] \leftrightarrow A[r]$
- 9 **devolva** **PARTICIONE**( $A, p, r$ )

## Exercícios

**Exercício 1** Mostre como modificar **QUICKSORT** de modo que tenha complexidade de tempo  $\Theta(n \lg n)$  no **pior caso**.

**Exercício 2** Suponha que você tenha uma subrotina do tipo “caixa-preta” que determina a mediana em **tempo linear** (no **pior caso**). Descreva um algoritmo linear simples que resolve o problema da seleção para todo  $i$ .

## Exercícios

**Exercício 3** Dado um conjunto de  $n$  números, queremos imprimir em ordem crescente os  $i$  maiores elementos deste usando um algoritmo baseado em comparações. Compare a complexidade dos seguintes métodos em função de  $n$  e  $i$ .

- Ordene o vetor e liste os  $i$  maiores elementos.
- Construa uma fila de prioridade (max-heap) e chame a rotina EXTRACT-MAX  $i$  vezes.
- Use um algoritmo de seleção para encontrar o  $i$ -ésimo maior elemento, particione o vetor em torno dele e ordene os  $i$  maiores elementos.