Análise da Complexidade de Algoritmos Recursivos VI

Joaquim Madeira

16/04/2020

Sumário

- Recap
- Seleção do k-ésimo elemento: o Algoritmo Quickselect
- Cálculo do valor de um polinómio: o Método de Horner
- Multiplicação de matrizes: o Algoritmo de Strassen
- Sugestões de leitura

Recapitulação

Quicksort

- Ordenar o array de modo recursivo, sem usar memória adicional

- Particionar o conjunto de elementos, trocando de posição, se necessário

- Com base no valor de um elemento pivot

Quicksort

- Escolher o valor do element pivot

- Particionar o array
- Elementos da 1ª partição são menores ou iguais do que o pivot
- Elementos da 2ª partição são maiores ou iguais do que o pivot

- Ordenar de modo recursivo a 1ª partição e a 2ª partição

1ª versão

```
void quicksort(int* A, int left, int right) {
  // Casos de base
  if (left >= right) return;

  // Caso recursivo
  // FASE DE PARTIÇÃO
  int pivot = (left + right) / 2;
  int i = left;
  int j = right;

  do {
    while (A[i] < A[pivot]) i++;
    while (A[j] > A[pivot]) j--;

    if (i <= j) {
      trocar(&A[i], &A[j]);
      i++;
      j--;
    }
  } while (i <= j);

  // Chamadas recursivas
  quicksort(A, left, j);
  quicksort(A, i, right);
}
```

Eficiência

- Todas as comparações são feitas na fase de partição !!

- $O(n \log n)$ para o melhor caso e o caso médio
- MAS $O(n^2)$ para o pior caso !!
    - Muito raro, se escolhermos "bem" o pivot
    - Ou se gerarmos uma permutação aleatória do array dado

- Mais comparações do que o Mergesort !
- MAS, na prática, é mais rápido do que o Mergesort !!

K-Selection
– Selecionar o k-ésimo elemento de um conjunto não ordenado

K-Selection

- Dado um array com n elementos : A[0,…,n – 1]
- O array não está ordenado !!

- Qual o valor do elemento na posição de índice k ?
  - Min (k = 0) / Max (k = n – 1) / Mediano (k = n **div** 2)
  - Aplicação: **top k** elementos

- Possíveis soluções ?
- Complexidade ?

K-Selection

- Ideias ? / Eficiência ?

K-Selection – Estratégia direta

- Ordenar por ordem não decrescente os n elementos
- Consultar o elemento na posição k
- Quanto tempo ?
  - 1.000.000 elementos / 10.000.000 elementos / …

K-Selection – Estratégia direta

- Ordenar por ordem não decrescente os n elementos

  $O(n^2)$ ou $O(n \log n)$

- Consultar o elemento na posição k

  $O(1)$

- Quanto tempo ?
  - 1.000.000 elementos / 10.000.000 elementos / …

K-Selection – Estratégia melhorada

- Copiar os primeiros k elementos para um array A

- Ordenar por ordem não decrescente esses k elementos

- Para cada um dos restantes (n – k) elementos
  - Ignorar se maior ou igual que A[k – 1]
  - Caso contrário, inserir ordenadamente em A
    - O elemento A[k – 1] é expulso do array e substituído

K-Selection – Estratégia melhorada

- O que demora mais tempo ?

- Ordenar os primeiros k elementos

  $O(k^2)$ ou $O(k \log k)$

- Para cada um dos restantes (n – k) elementos
  - Ignorar se maior ou igual que A[k – 1] **$O(1)$**
  - Caso contrário, inserir ordenadamente em A **$O(k)$**

K-Selection – Estratégia melhorada

- Ordem de complexidade ?

$$O(k \log k) + (n - k) \times O(k) = O(n \times k)$$

- Encontrar o elemento mediano $O(n^2)$

- Quanto tempo ?
  - 1.000.000 elementos / 10.000.000 elementos / …

K-Selection – Usar uma MIN-HEAP

- Transformar o array numa MIN-Heap com n elementos
- Efetuar k operações deleteMin()
- O último elemento removido é o procurado
- Esta estratégia lembra-nos alguma coisa ?

K-Selection – Usar uma MIN-HEAP

- Transformar o array numa MIN-Heap com n elementos

  O(n)

- Efetuar k operações deleteMin()

  k x **O(log n)**

- O último elemento removido é o procurado

O(n + k x log n)

K-Selection – Usar uma MIN-HEAP

$$O(n + k \times \log n)$$

- Se $k = O(n / \log n)$ então $O(n)$
- Encontrar o elemento mediano $O(n \log n)$

K-Selection – MAX-HEAP “mais pequena”

- Copiar os primeiros k elementos para um array A
- Transformar o array numa MAX-Heap com k elementos
- Para cada um dos restantes (n – k) elementos
  - Comparar com o elemento do topo da heap: A[0]
  - Se for menor, substituir A[0] e reorganizar a heap
- O elemento do topo da heap é o procurado: A[0]

K-Selection – MAX-HEAP "mais pequena"

- Transformar o array numa MAX-Heap com k elementos **O(k)**
- Para cada um dos restantes (n – k) elementos
  - Comparar com o elemento do topo da heap: A[0] **O(1)**
  - Se for menor, substituir A[0] e reorganizar a heap **O(log k)**

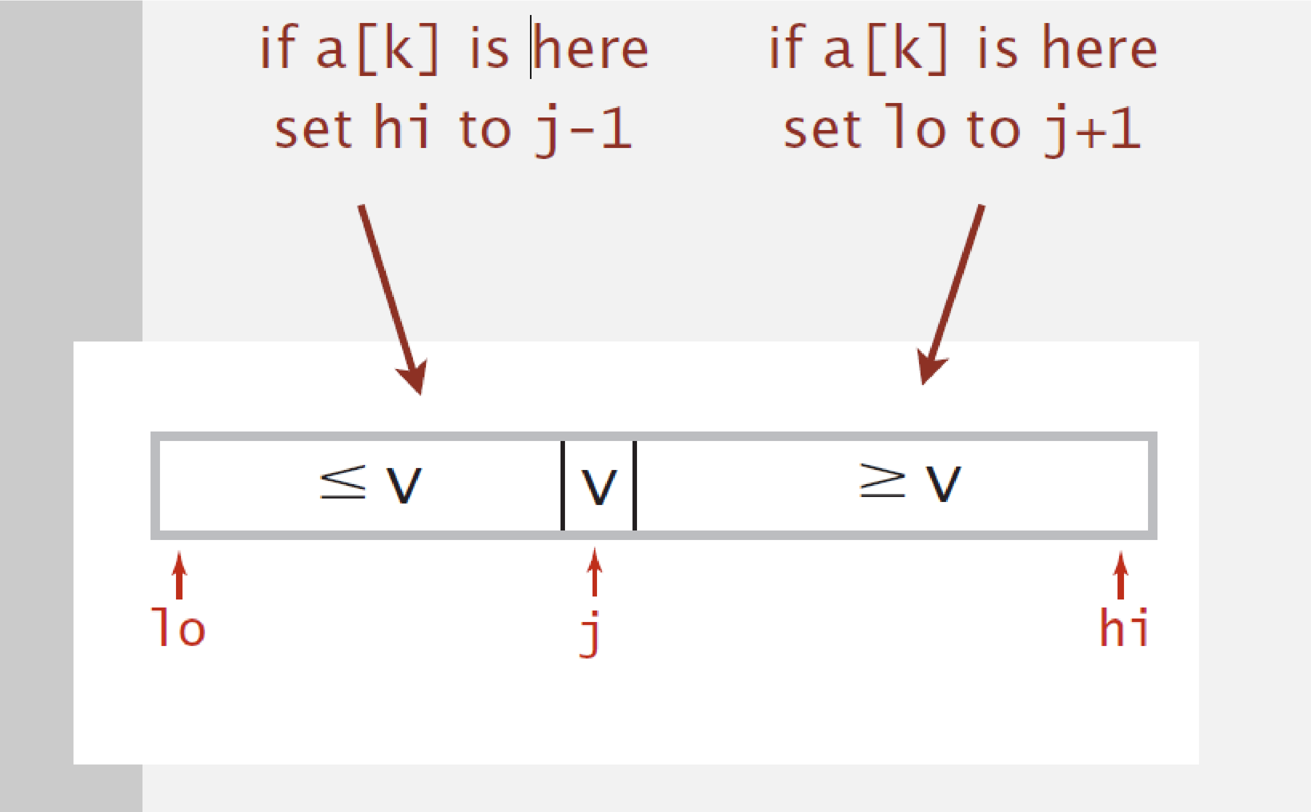
$$(k + (n - k) \times \log k) = O(n \times \log k)$$

K-Selection – Algoritmo Recursivo de Partição

- Como podemos aproveitar a ideia de partição do Quicksort ?

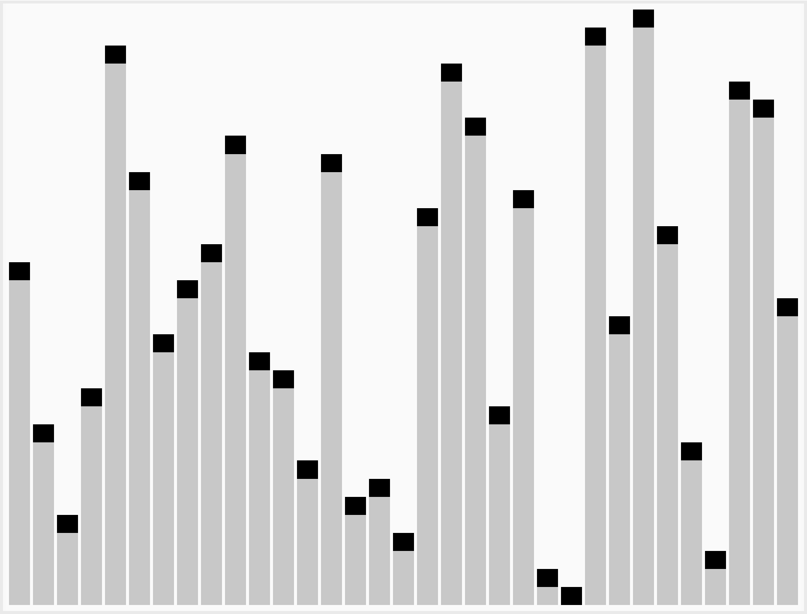
- Particionar o array de modo que:
  - Elemento a[j] esteja no seu lugar
  - Não há elementos maiores à sua esquerda
  - Não há elementos menores à sua direita
- Proceder de modo recursivo
  - MAS, processar uma só das partições

K-Selection – Algoritmo Recursivo de Partição



[Sedgewick & Wayne]

K-Selection – Algoritmo Recursivo de Partição



[Wikipedia]

Eficiência

- No caso médio, algoritmo LINEAR !!

- Cada partição é dividida em 2 “metades”

- Nº total de comparações aprox. igual a

  $$n + n/2 + n/4 + \ldots + 1 \sim 2 \times n \text{ comparações}$$

- Ex: 3.38 x n comparações para encontrar o mediano

Eficiência

- No pior caso, algoritmo QUADRÁTICO !!
  - Random shuffling no início, para evitar que aconteça

- A partição seguinte só tem menos 1 elemento
  - Tal como no pior caso do Quicksort

- Nº total de comparações aprox. igual a

$$n + (n - 1) + \ldots + 1 \sim n^2/2 \text{ comparações}$$

K-Selection – Algoritmo Recursivo de Partição

```c
int qselect(int *v, int len, int k)
{
#	define SWAP(a, b) { tmp = v[a]; v[a] = v[b]; v[b] = tmp; }
	int i, st, tmp;

	for (st = i = 0; i < len - 1; i++) {
		if (v[i] > v[len-1]) continue;
		SWAP(i, st);
		st++;
	}

	SWAP(len-1, st);

	return k == st	?v[st]
			:st > k ? qselect(v, st, k)
				: qselect(v + st, len - st, k - st);
}
```

[rosettacode.org]

Cálculo do valor de um polinómio – O Método de Horner

Calcular o valor de um polinómio

- $P(x) = ?$

- $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$

- Algoritmo directo e ingénuo – Quantas multiplicações ?

- Tarefa: fazer em casa !!

Método de Horner

- $P(x) = ?$

- $P(x) = a_0 + x\,(a_1 + x\,(a_2 + x\,(a_3 + \ldots + x\,(a_{n-1} + x\,a_n) \ldots)$

- Método de Horner – Quantas multiplicações ?

- Tarefa: fazer em casa – V. iterativa + V. recursiva !!

Multiplicação de matrizes
– O Algoritmo de Strassen

Multiplicação de matrizes

```
for( int i = 0; i < n; ++i )    // Initialization
    for( int j = 0; j < n; ++j )
        c[ i ][ j ] = 0;

for( int i = 0; i < n; ++i )
    for( int j = 0; j < n; ++j )
        for( int k = 0; k < n; ++k )
            c[ i ][ j ] += a[ i ][ k ] * b[ k ][ j ];
```

[Weiss]

Multiplicação de matrizes

- Caso mais simples ?
  - Multiplicar matrizes (2 x 2) !!
- Algoritmo direto
  - 8 multiplicações
  - 4 adições
- Como fazer menos multiplicações ?
- MAS, não se fazem omeletas sem partir ovos…

Multiplicação de matrizes

- Como multiplicar matrizes de grande dimensão ?
- O algoritmo direto é $\Theta(m \times n \times p) = \Theta(n^3)$
- Podemos fazer melhor ?
- Divide-and-Conquer
  - Subdividir cada matriz em 4 sub-matrizes
  - Multiplicar recursivamente sub-matrizes
  - Combinar resultados intermédios para obter a matriz final

1ª estratégia

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

[Weiss]

1ª estratégia

$$\mathbf{AB} = \begin{bmatrix} 3 & 4 & 1 & 6 \\ 1 & 2 & 5 & 7 \\ 5 & 1 & 2 & 9 \\ 4 & 3 & 5 & 6 \end{bmatrix} \begin{bmatrix} 5 & 6 & 9 & 3 \\ 4 & 5 & 3 & 1 \\ 1 & 1 & 8 & 4 \\ 3 & 1 & 4 & 1 \end{bmatrix}$$

[Weiss]

we define the following eight $N/2$-by-$N/2$ matrices:

$$A_{1,1} = \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \quad A_{1,2} = \begin{bmatrix} 1 & 6 \\ 5 & 7 \end{bmatrix} \quad B_{1,1} = \begin{bmatrix} 5 & 6 \\ 4 & 5 \end{bmatrix} \quad B_{1,2} = \begin{bmatrix} 9 & 3 \\ 3 & 1 \end{bmatrix}$$

$$A_{2,1} = \begin{bmatrix} 5 & 1 \\ 4 & 3 \end{bmatrix} \quad A_{2,2} = \begin{bmatrix} 2 & 9 \\ 5 & 6 \end{bmatrix} \quad B_{2,1} = \begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix} \quad B_{2,2} = \begin{bmatrix} 8 & 4 \\ 4 & 1 \end{bmatrix}$$

1ª estratégia

- Quantas multiplicações são efetuadas no total ?

  $M(1) = 1$

  $M(n) = 8\ M(n / 2)$

  $M(n) \text{ in } \Theta(n^{\log_2 8}) = \Theta(n^3)$!!??!!

- Como melhorar a ordem de complexidade ?
- Como reduzir o nº de multiplicações recursivas ?

Algoritmo de Strassen

- $C = A \times B$
  - matrizes (2 x 2)

$c_{00} = m_1 + m_4 - m_5 + m_7$
$c_{01} = m_3 + m_5$
$c_{10} = m_2 + m_4$
$c_{11} = m_1 + m_3 - m_2 + m_6$

$m_1 = (a_{00} + a_{11}) \times (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) \times b_{00}$
$m_3 = a_{00} \times (b_{01} - b_{11})$
$m_4 = a_{11} \times (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) \times b_{11}$
$m_6 = (a_{10} - a_{00}) \times (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) \times (b_{10} + b_{11})$

- 7 multiplicações (!)
- 18 adições / subtrações

Algoritmo de Strassen

- C = A x B
  - matrizes (n x n)
  - $n = 2^k$
- Subdividir cada matriz em 4 sub-matrizes !
- Usar as mesmas "fórmulas" para processar cada sub-matriz
  - Adicionar, subtrair e multiplicar sub-matrizes
- As multiplicações são efetuadas de modo recursivo !!

Algoritmo de Strassen

- Eficiência ?
  - Multiplicações ? / Adições / subtrações ?

$M(1) = 1$

$M(n) = 7\ M(n / 2)$

$M(n) \text{ in } \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$

$A(1) = 0$

$A(n) = 7\ A(n / 2) + 18 \times (n / 2)^2$

$A(n) \text{ in } \Theta(n^{2.807})$

Algoritmo de Strassen

- Há detalhes de implementação a ter em conta…

- Acrescentar "zeros" para transformar em matrizes quadradas de tamanho apropriado

- Só é vantajoso para matrizes muito grandes !

Algoritmo de Strassen

- Melhor do que o algoritmo direto !!
- Menos multiplicações, MAS mais adições / subtrações !!
- Hoje em dia, há algoritmos mais eficientes
  - E.g., Coppersmith and Winograd's is $\Theta(n^{2.376})$
  - Implementações complexas
  - Pouca aplicabilidade

Sugestões de leitura

Sugestões de leitura

- A. Levitin, Introduction to the Design and Analysis of Algorithms, 3rd Edition, 2012
  - Capítulo 4: secção 4.5
  - Capítulo 5: secção 5.4
  - Capítulo 6: secção 6.5

- M. A. Weiss, Data Structures and Algorithm Analysis in C++, 4th Edition, 2014
  - Capítulo 1: secção 1.1
  - Capítulo 6: secção 6.4
  - Capítulo 7: secção 7.7
  - Capítulo 10: secção 10.2