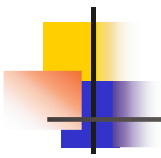
QuickSort

Algoritmos e Estuturas de Dados

Cátia Vaz



QuickSort

- Algoritmo do tipo ***dividir para conquistar***
- **Ideia do algoritmo:** efectuar partição dos dados e ordenar as várias partes independentemente (de forma recursiva)
  - posicionamento da partição a efectuar depende dos dados de entrada
  - processo de partição é crítico
  - uma vez efectuada a partição, cada uma das partes pode por sua vez ser ordenada pelo mesmo algoritmo

.

Cátia Vaz

QuickSort-partição

```
public static void quicksort(int a[], int left, int right){
   int i;
   if (right <= left) return;
   i = partition(a, left, right);
   quicksort(a, left, i-1);
   quicksort(a, i+1, right);
}
```

QUICKSORT(A, p, r)
if $p < r$
  then $q \leftarrow$ PARTITION(A, p, r)
    QUICKSORT$(A, p, q-1)$
    QUICKSORT$(A, q+1, r)$

- Processo de partição rearranja os dados de forma a que as três condições seguintes sejam válidas (de **a[left]** a **a[right]**):
  - o elemento **a[i]**, para algum **i**, fica, após a partição, na sua posição final;
  - nenhum dos elementos em **a[left]** … **a[i-1]** é maior do que **a[i]**;
  - nenhum dos elementos em **a[i+1]** … **a[right]** é menor do que **a[i]**.

QuickSort-partição

- Para cada processo de partição, **pelo menos um elemento** fica na sua **posição final**.
- Após partição, o *array* fica sub-dividido em duas partes
  - que podem ser ordenadas separadamente
- A ordenação completa é conseguida através de **partição** + aplicação **recursiva** do algoritmo aos dois subconjuntos de dados daí resultantes.

QuickSort-partição

- **Estratégia para a partição** escolher **a[right]** arbitrariamente para ser o elemento de partição
  - percorrer o *array* a partir da esquerda até encontrar um elemento maior que ou igual ao elemento de partição (**a[right]**)
  - percorrer o *array* a partir da direita até encontrar um elemento menor que ou igual ao elemento de partição (**a[right]**)
    - trocamos as posições destes dois elementos!
  - procedimento continua até nenhum elemento à esquerda de **a[right]** ser maior que ele, e nenhum elemento à direita de **a[right]** ser menor que ele
    - completa-se trocando **a[right]** com o elemento mais à esquerda do sub-*array* da direita

QuickSort-partição

```
public static int partition(int a[], int left, int right){
   int i, j;//i da esquerda para a direita, j ao contrário
   int v; //elemento de partição
   v = a[right]; i = left-1; j = right;
   for (;;) {
        while (less(a[++i], v)) ;
        while (less(v, a[--j]))
        if (j == left) break;
        if (i >= j) break;
        exch(a[i], a[j]);
   }
   exch(a[i], a[right]);
return i;
}
```

QuickSort-partição

- Processo de partição não é estável
  - qualquer chave pode ser movida para trás de várias outras chaves iguais a si (que ainda não foram examinadas)
- A eficiência do processo de ordenação depende de como a partição divide os dados (**do elemento de partição**)
  - será tanto mais equilibrada quanto mais perto este elemento estiver do meio do array na sua posição final

QuickSort-características

Pode ser muito ineficiente em casos patológicos!

- **Propriedade: *quicksort* usa cerca de $N^2/2$ comparações no pior caso.**

  Demonstração: se o array já estiver ordenado, **todas as partições degeneram e o programa chama-se a si próprio *N* vezes**;
  o número de comparações é de
  $N + (N-1) + (N-2) + \ldots + 2 + 1 = (N + 1) N / 2$
  (mesma situação se o ficheiro estiver ordenado por ordem inversa)

  Nota: Não apenas o tempo necessário para a execução do algoritmo cresce quadraticamente como o espaço necessário para o processo recursivo é de cerca de **N** o que é inaceitável para ficheiros grandes.

QuickSort-características

Melhor caso: quando cada partição divide o ficheiro de entrada **exactamente em metade**

- **número de comparações** usadas por *quicksort* satisfaz a recursão de dividir para conquistar

$$C(N) = 2\,C(N/2) + N$$

e logo $C(N) = O(N \log N)$

QuickSort-características

- **Propriedade**: *quicksort* usa cerca de **2N lg N** comparações em média

Demonstração: A **fórmula de recorrência exacta para o número de comparações** utilizado por *quicksort* para ordenar **N** números distintos aleatoriamente posicionados é

$$C(N) = N + 1 + \frac{1}{N}\sum_{k=1}^{N}(C(K-1) + C(N-K)), \qquad N \geq 2 \quad \text{e} \quad C(0) = C(1) = 0$$

o termo **N+1** cobre o custo de comparar o elemento de partição com os restantes (2 comparações extra: i e j cruzam-se)

cada elemento tem probabilidade **1/N** de ser o elemento de partição após o que ficamos com duas sub-arrays de tamanhos **k-1** e **N-k**

QuickSort-características

- Esta análise assume que os dados estão **aleatoriamente ordenados** e têm **chaves diferentes**
  - pode ser lento em situações em que as chaves não são distintas ou que os dados não estão aleatoriamente ordenados
- O algoritmo pode ser melhorado para reduzir a probabilidade que um **caso mau** ocorra.

Quick Sort- análise caso médio

$$C_N = (N-1) + \frac{1}{N}\sum_{k=1}^{N}(C_{K-1} + C_{N-K}), \quad N \geq 2 \quad \text{e} \quad C(0) = C(1) = 0$$

$$= (N-1) + \frac{2}{N}\sum_{k=1}^{N} C_{K-1}$$

Multiplicando por N:

$$N\,C_N = N(N-1) + 2\sum_{k=1}^{N} C_{K-1}$$

Subtraindo a mesma fórmula para N-1:

$$NC_N - (N-1)\,C_{N-1} = N(N-1) - (N-1)(N-2) + 2\sum_{k=1}^{N} C_{K-1} - 2\sum_{k=1}^{N-1} C_{K-1}$$

$$= 2(N-1) + 2\,C_{N-1}$$

$$NC_N = (N+1)\,C_{N-1} + 2(N-1)$$

$$= (N+1)\,C_{N-1} + 2N - 2$$

$$= (N+1)\,C_{N-1} + 2N$$

Quick Sort- análise caso médio

Dividindo ambos os lados por N(N+1): $N C_N = (N+1)\, C_{N-1} + 2N$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Usando a recorrência:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

$$= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \frac{C_2}{3} + \sum_{k=3}^{N} \frac{2}{K+1}$$

como: $\sum_{k=1}^{N} \frac{1}{K} = O(\lg n)$

Quick sort usa aproximadamente **2 N lg N** comparações em média

QuickSort-características

- **Questões mais relevantes:**
  - possível redução de desempenho devido ao uso de recursão
  - tempo de execução dependente dos dados de entrada
  - tempo de execução quadrático no pior caso
  - **espaço/memória necessário no pior caso é linear**
    - um problema sério (para uma estrutura de dados de grandes dimensões)
- **Problema do espaço está associado ao uso de recursão:**
  - recursão implica chamada a função e logo a carregar dados na pilha/stack do computador

QuickSort- versão não recursiva

- Usamos uma pilha (stack) explícita
  - pilha contém "sub arrays" a serem processados (sub arrays a ordenar)
  - quando precisamos de um sub-array para processar tiramo-la da pilha (i.e.fazemos um ***pop()*** do stack)
  - por cada partição criamos duas sub-arrays e colocamos ambas na pilha (i.e. fazemos dois ***push()*** para o stack)
  - substitui a pilha do computador que é usado na implementação recursiva
- Conduz a uma versão não recursiva de ***quicksort()***

QuickSort

- versão não recursiva de ***quicksort()***
  - verifica os tamanhos dos dois sub-arrays e coloca o maior deles primeiro na pilha (e o menor depois; logo o menor é retirado e tratado primeiro)
  - ordem de processamento dos sub-arrays não afecta a correcta operação da função ou o tempo de processamento mas afecta o tamanho da pilha

QuickSort-versão não recursiva

```
static void quicksort(int a[], int left, int right){
   Stack<Integer> s = new Stack<Integer>();
   s.push(left);s.push(right)
   while (!s.empty()) {
        right = s.pop(); left = s.pop();
        if (right <= left) continue;
        i = partition(a, left, right);
        if (i-left > right-i){
               s.push(left); s.push(i-1);}
        s.push(i+1);s.push(right);
        if (i-left <= right-i){
               s.push(left);s.push(i-1);}
   }
}
```


QuickSort-versão não recursiva

- *A estratégia de colocar o maior dos sub-arrays primeiro na pilha*
  - garante que cada entrada na pilha não é maior do que metade da que estiver antes dela na pilha
  - pilha apenas ocupa **lg N** no pior caso
    - que ocorre agora quando a partição for sempre no meio da tabela
    - em ficheiros aleatórios o tamanho máximo da pilha é bastante menor
  - Estratégia que pode ser também na **versão recursiva**.
- **Propriedade:** *se a menor dos dois sub-arrays for ordenado primeiro, a pilha nunca necessita mais do que* **lg N** *entradas quando quicksort é usado para ordenar* **N** *elementos.*

  Demonstração: no pior caso o tamanho da pilha é inferior a T(N) em que T(N) satisfaz a recorrência

  $T(N) = T(\lfloor N/2 \rfloor) + 1$ $(T(0) = T(1) = 0)$

QuickSort-versão não recursiva

- **Propriedade**: se a menor dos dois sub-*arrays* for ordenado primeiro, a pilha nunca necessita mais do que **lg N** entradas quando *quicksort* é usado para ordenar **N** elementos.

  Demonstração: no pior caso o tamanho da pilha é inferior a **T(N)** em que **T(N)** satisfaz a recorrência

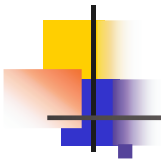
  T(N) = T (⌊N/2⌋) + 1 (T(0) = T(1) = 0)

QuickSort- melhoramentos

- Algoritmo pode ainda ser melhorado:
  - porquê colocar ambas os sub-*arrays* na pilha se um deles é de imediato retirado
    - teste de **right <= left** é feito assim que os sub-*arrays* saem da pilha
      - seria melhor nunca as lá ter colocado!
  - ordenação de **sub-arrays de pequenas dimensões** pode ser efectuada de forma mais eficiente
    - ao mudar o teste no início da função recursiva para uma chamada

      `if (right-left <= M) insertionSort(a, left, right)`

      em que **M** é um parâmetro a definir na implementação
    - Algoritmo híbrido, para *arrays* pequenos usar o *insertion sort.*



QuickSort- melhoramentos

- Utilizar um elemento de partição que com alta probabilidade divida o *array* pela metade
  - pode-se usar um elemento aleatoriamente escolhido
    - evita o pior caso (i.e. pior caso tem baixa probabilidade de acontecer)
    - é um exemplo de um algoritmo probabilístico
  - pode-se escolher alguns (ex: três) elementos do *array* e usar a **mediana dos três** como elemento de partição
    - escolhendo os três elementos da esquerda, meio e direita da tabela podemos incorporar sentinelas na ordenação
    - ordenamos os três elementos, depois trocamos o do meio com **a[right-1]** e corremos o algoritmo de partição em **a[left+1]** … **a[right-2]**
    - este melhoramento chama-se o método da **mediana de três**.

QuickSort- melhoramentos

- Método da mediana de três melhora *quicksort* por três razões
  - o pior caso é mais improvável de acontecer na prática
    - dois dos três elementos teriam de ser dos maiores ou menores do *array* e isto teria de acontecer constantemente a todos os níveis de partição
  - reduz o tempo médio de execução do algoritmo embora apenas por cerca de 5%
  - caso particular de métodos em que se faz amostragem dos dados para estimar as suas propriedades
- Conjuntamente com o método de tratar de pequenos *arrays* pode dar ganhos de 20 a 25%

QuickSort- estudo empírico

		Quicksort básico			Quicksort melhorado		
N	shellsort	M=0	M=10	M=20	M=0	M=10	M=20
12500	6	2	2	2	3	2	3
25000	10	5	5	5	5	4	6
50000	26	11	10	10	2	9	14
100000	58	24	22	22	25	20	28
200000	126	53	48	50	52	44	54
400000	278	116	105	110	114	97	118
800000	616	255	231	241	252	213	258

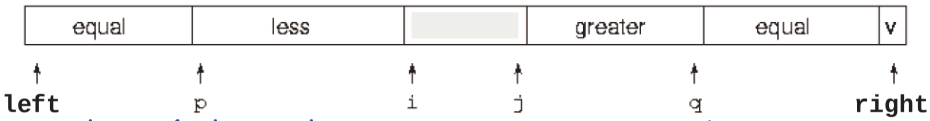
- *Quicksort* é cerca de 2 vezes mais rápido que *shellsort* para *arrays* grandes aleatoriamente ordenados.
- Usando *insertion* para **pequenos *arrays*** e a **estratégia de mediana-de-três** melhoram cada um a eficiência por um factor de 10%

QuickSort- chaves duplicadas

- *Arrays* com um grande número de chaves duplicadas são frequentes na prática
  - desempenho de *quicksort* pode ser substancialmente melhorado se
    - todas as chaves forem iguais
      - *quicksort* mesmo assim faz **N lg N** comparações
    - houver duas chaves distintas
      - reduz-se ao problema anterior para cada sub-*array*
  - natureza recursiva de *quicksort* garante que haverá frequentemente sub-*arrays* de items com poucas chaves
    - uma possibilidade é dividir o *array* em três partes
      - cada uma para chaves menores, iguais e maiores que o elemento de partição

QuickSort- chaves duplicadas

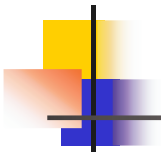
- Solução: fazer uma partição em três partes
  - manter **chaves iguais** ao elemento de partição que são encontradas no **sub-*array* da esquerda** do **lado esquerdo do *array***
  - manter **chaves iguais** ao elemento de partição que são encontradas no **sub-*array* da direita** do **lado direito do *array***



  - Quando os índices de pesquisa se cruzam sabemos onde estão os elementos iguais ao de partição e é fácil colocá-los em posição
    - trabalho extra para chaves duplicadas é proporcional ao número de chaves duplicadas: funciona bem se não houver chaves duplicadas
    - **linear** quando há um número constante de chaves

QuickSort- partição em três

```
static void quicksort(int a[], int left, int right){
   int i, j, k, p, q, v;
   if (right <= left) return;
   v = a[right]; i = left-1; j = right; p = left-1; q = right;
   for (;;) {
        while (less(a[++i], v)) ;
        while (less(v, a[--j])) if (j == left) break;
        if (i >= j) break;
        exch(a[i], a[j]);
        if (equal(a[i],v)) { p++; exch(a[p],a[i]);}
        if (equal(v,a[j])) { q--; exch(a[q],a[j]);}
   }
   exch(a[i], a[right]); j = i-1; i = i+1;
   for (k = left ; k < p; k++, j--) exch(a[k], a[j]);
   for (k = right-1; k > q; k--, i++) exch(a[k], a[i]);
   quicksort(a, left, j);
   quicksort(a, i, right);
}
```



Junção versus Partição

- *Quicksort* é baseado na operação de partição
  - é efectuada uma partição e quando as duas metades do *array* estão ordenadas, o *array* está ordenado
- Operação complementar é de **junção (*merge*)**
  - combinar dois arrays para obter um, maior, ordenado
  - dividir os arrays em duas partes para serem ordenados e depois combinar as partes de forma a que o array total fique ordenado
  - *mergesort*
- *Mergesort:* de **N** elementos é feito em tempo proporcional a **N log N, independentemente dos dados**!!