Ordenação e Pesquisa

Luís Lopes

DCC-FCUP

Estruturas de Dados

Pesquisa de Informação

A pesquisa eficiente de informação é extremamente relevante, seja:

- num catálogo indexado por uma relação de ordem, e.g. alfabética, como seja uma lista telefónica;
- nos registos de uma base de dados
- informação em páginas Web
- em ficheiros binários, e.g., imagens

A pesquisa sequencial é dos casos mais simples e é normalmente usada quando não se tem uma relação de ordem na informação a pesquisar.

Pesquisa Sequencial

- pesquisa sequencial consiste em procurar um valor `val` numa sequência de valores `v` com dimensão `v.length`;
- em seguida está definida a pesquisa num array de objectos de tipo `T`;
- nota: os objectos de tipo `T` têm de implementar a interface `Comparable`.

```
int sequentialSearch(T[] values, T val) {
  for (int i = 0 ; i < values.length ; i++)
    if ( isEqual(val,values[i]) )
      return i;
  return -1;
}

boolean isEqual ( T val1, T val2 ) {
  return ( val1.compareTo(val2) == 0 );
}
```

Pesquisa Binária

- é uma estratégia eficiente de pesquisa de um valor `val` numa sequência ordenada de valores `v` entre os índices `low` e `high`
- Ideia do algoritmo:
  - seja `m` a posição média de `v`
  - comparar `val` com `v[m]`
  - se `val == v[m]` então encontrou
  - senão se `val < v[m]`, procura `val` no intervalo `[low,m-1]`
  - senão (`val>v[m]`), procura `val` no intervalo `[m+1,high]`

Pesquisa Binária: versão recursiva

```
int binarySearchRec(T[] values, T val ) {
  return binarySearchRec(values, val, 0, values.length - 1);
}

int binarySearchRec(T[] values, T val, int low, int high) {
  if( high < low )
    return -1;
  else {
    int half = low + ( high - low ) / 2;
    if ( isEqual( val, values[half] ) )
      return half;
    else if ( isLess( val, values[half] ) )
      return binarySearchRec(values, val, low, half - 1);
    else
      return binarySearchRec(values, val, half + 1, high);
  }
}
```

Pesquisa Binária: versão iterativa

```
int binarySearch(T[] values, T val ) {
  return binarySearch(values, val, 0, values.length - 1);
}

int binarySearch(T[] values, T val, int low, int high) {
  while ( low <= high ) {
    int half = low + ( high - low ) / 2;
    if ( isEqual( val, values[half] ) )
      return half;
    else if ( isLess( val, values[half] ) )
      high = half - 1;
    else
      low =  half + 1;
  }
  return -1;
}
```

Algoritmos de Ordenação

Problemas em que a ordenação é relevante:

- unicidade / repetições dos elementos de uma sequência
- atribuição de prioridades
- selecção de elementos numa sequência
- frequência de um elemento numa sequência
- intersecção / reunião de sequências
- procura eficiente de informação

Método de ordenação: selecção de máximo/mínimo

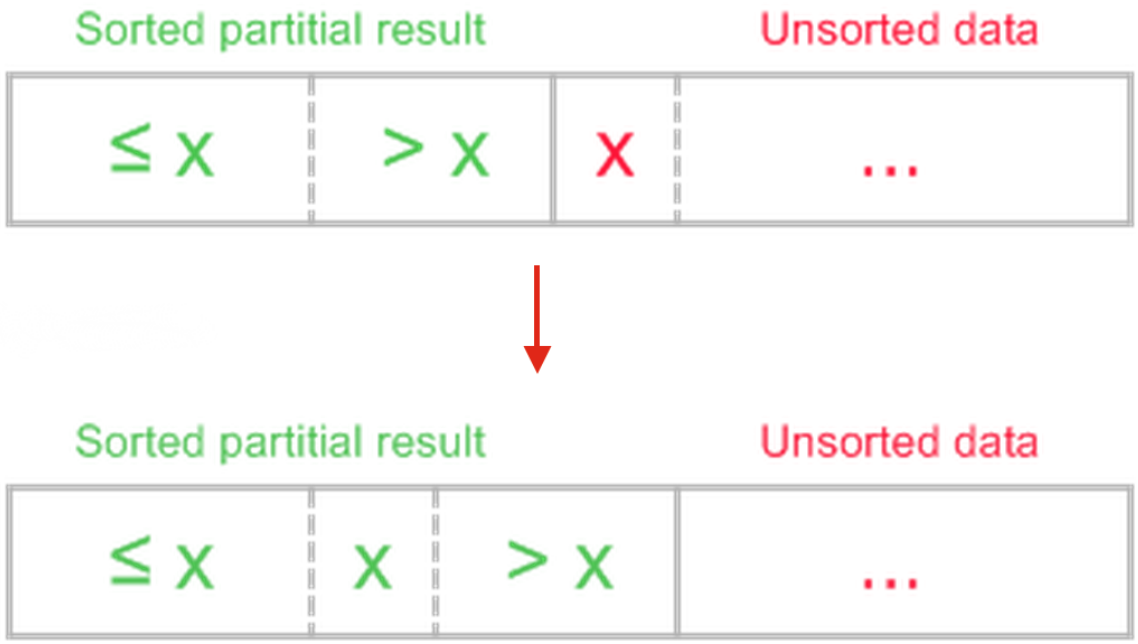
Na posição `i` do vector `v`, coloca-se o menor elemento entre os ainda não ordenados, i.e. que se encontram entre `i + 1` e `v.length - 1`.

```
void selectMinimumSort(T[] values) {
  for (int i = 0 ; i < values.length ; i++) {
    int min = i;
    for (int j = i + 1 ; j < values.length ; j++)
      if ( isLess(values[j], values[min]) )
        min = j;
    swap(values, i, min);
  }
}

void swap ( T[] values, int i, int j ) {
  T tmp      = values[i];
  values[i] = values[j];
  values[j] = tmp;
}
```

Método de ordenação: Inserção

Este método de ordenação é designado de "inserção" porque divide um array em parte ordenada e parte não ordenada. Depois, retira os elementos um a um da parte não ordenada e insere-os na posição correcta entre os elementos ordenados.



Método de ordenação: Inserção

```
void insertionSort(T[] values) {
  for ( int i = 1 ; i < values.length ; i++ ) {
    int j = i;
    while ( j > 0 && isGreater(values[j - 1],values[j]) ) {
      swap( values, j, j - 1 );
      j--;
    }
  }
}

boolean isGreater ( T val1, T val2 ) {
  return ( val1.compareTo(val2) > 0 );
}
```

Método de ordenação: Bolha (Bubble Sort)

Este método consiste em comparar os elementos do vector dois a dois e trocar os elementos que não estiverem na ordem desejada. O nome vem do facto de em cada iteração as comparações sucessivas de elementos deslocarem o maior (ou menor) elemento para as últimas posições do vector como se fosse o deslizar de uma bolha.

```
void bubbleSort(T[] values) {
  for ( int i = 0 ; i < values.length - 1 ; i++ )
    for ( int j = 0 ; j < values.length - 1 - i ; j++ )
      if ( isGreater( values[j], values[j+1] ) )
          swap( values, j, j+1 );
}

boolean isGreater ( T val1, T val2 ) {
  return ( val1.compareTo(val2) > 0 );
}
```

Método de ordenação: Bolha (Bubble Sort)

O algoritmo 1, faz sempre $n(n-1)/2$ comparações. Se o vector ficar ordenado a meio das iterações, faz comparações desnecessárias. Uma optimização consiste em usar uma flag que sinaliza a ocorrência de trocas numa iteração. O algoritmo termina se não ocorrerem trocas na iteração anterior.

```
void bubbleSort(T[] values) {
  boolean swapped = true;
  for ( int i = 0 ; i < values.length - 1 && swapped ; i++ ){
    swapped = false;
    for ( int j = 0 ; j < values.length - 1 - i ; j++ )
      if ( isGreater( values[j], values[j+1] ) ) {
        swap( values, j, j+1 );
        swapped = true;
      }
  }
}
```

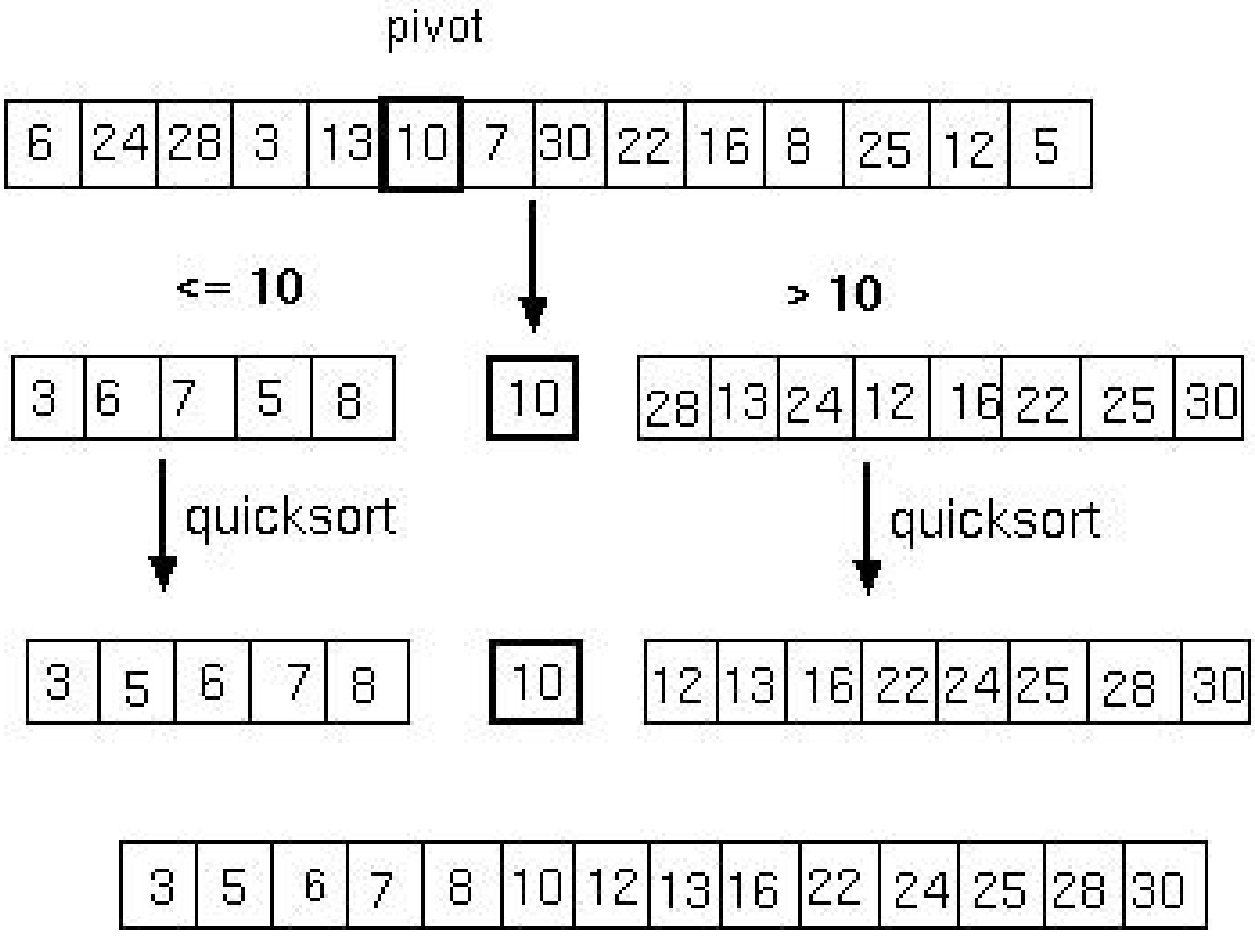
Ordenação: Método Quick-Sort

O **Quick-Sort** é um dos algoritmos de ordenação mais eficientes ($O(n \log n)$).
Baseia-se numa estratégia recursiva dividir-para-conquistar.
Supondo que se pretende ordenar um vector S, consiste em:

1. Se o número de elementos em S é 0 ou 1, S está ordenado;
2. Escolher um elemento $v \in S$. Chamamos-lhe *pivot*.
3. Dividir $S - \{v\}$ em dois vectores:
   $S_1 = \{x \in S - \{v\} \mid x < v\}$, elementos menores que o *pivot*
   $S_2 = \{x \in S - \{v\} \mid x \geq v\}$, elementos maiores que o *pivot*
   Note-se que os elementos em S_1 e S_2 não estão necessariamente ordenados.
4. Aplicar recursivamente o algoritmo a S_1 e S_2

Neste algoritmo, o esforço está na partição do vector, sendo a estratégia de escolha do pivot importante. A junção dos sub-vectores já ordenados é uma simples colagem.

Ilustração do Quick-Sort



Implementação do Quick-Sort

```
void quickSort(T[] values) {
  quickSort(values, 0, values.length - 1);
}

void quickSort(T[] values, int low, int high) {
  if (high > low) {
    int index = partition(values, low, high);
    quickSort(values, low, index - 1);
    quickSort(values, index + 1, high);
  }
}
```

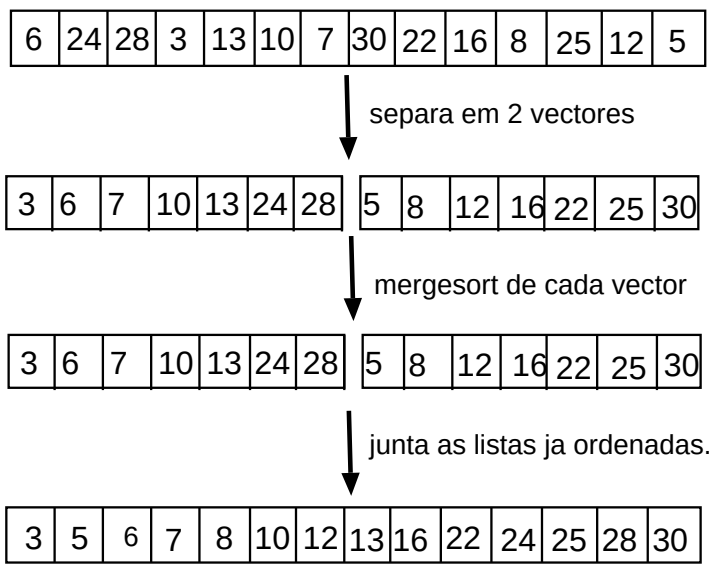
Implementação do Quick-Sort (cont.)

```
int partition(T[] values, int low, int high) {
    T pivot = values[low];
    int i   = low;
    int j   = high;
    while ( i < j ) {
      // Keep incrementing from the start of the range so long as the
      // values are less than the pivot.
      while ( i < high && isLessOrEqual( values[i], pivot ) )
        i++;
      // Keep decrementing from the end of the range so long as the
      // values are greater than the pivot.
      while ( j > low && isGreater( values[j], pivot ) )
        j--;
      // So long at the indexes have not crossed, swap the pivot with
      // the value that was out of place.
      if ( i < j )
        swap( values, i, j );
    }
    swap( values, j, low );
    return j;
}
```

Ordenação: Método Merge-Sort

A ideia base do algoritmo consiste em:

- subdivide o vector em duas partes,
- aplicar o algoritmo a cada parte,
- quando as duas partes estiverem ordenadas (2 vectores ordenados), faz-se a junção para produzir um vector final ordenado.



Implementação do Merge-Sort

```
void mergeSort( T[] v, T[] u ) {
  mergeSort( v, u, 0, v.length - 1 );
}

void mergeSort( T[] v, T[] u, int low, int high ) {
  if ( low < high ) {
    int half = low + (high - low) / 2;
    mergeSort( v, u, low, half );
    mergeSort( v, u, half + 1, high );
    merge( v, u, low, half, high );
  }
}
```

Mergesort - junção das partes

```
void merge(T v[], T u[], int low, int half, int high) {
  int i = low, j = half + 1, k = low;
  // Alternately copy to u the smallest elements by order
  while( i <= half && j <= high )
    if( isLess( v[i], v[j] ) )
      u[k++] = v[i++];
    else
      u[k++] = v[j++];
  // if you reached j = high first, copy rest of first half
  while( i <= half )
    u[k++] = v[i++];
  // if you reached i = half first, copy rest of second half
  while( j <= high )
    u[k++] = v[j++];
  // Copy u back to v
  for( i = 0 ; i < k ; i++ )
    v[i] = u[i];
}
```

Funções de ordenação do Java

- Usar a class `java.util.Arrays` que contém vários métodos para ordenar, e.g.

```
static void sort(Object[] a)
static void sort(Object[] a, Comparator c)
```

  Existem outros métodos específicos para inteiros ou strings.

- Em Java, a classe Arrays permite a invocação do método `sort()` para ordenar não apenas inteiros mas outro tipo de objectos. Para tal, os objectos têm de pertencer a classes que implementem a interface `Comparable`:

```
public interface Comparable {
  int compareTo(Object o);
}
```