Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 8

Pesquisa e Ordenação de Vetores

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Reis, Rocha e Faria, 2007)

Escola de Engenharia
Universidade do Minho

Estrutura

- **Algoritmos de Pesquisa em Vetores:**
  - Pesquisa Sequencial
  - Pesquisa Binária
- **Algoritmos de Ordenação de Vetores:**
  - Ordenação por Inserção
  - Ordenação por Selecção
  - BubbleSort
  - ShellSort
  - MergeSort
  - Ordenação por Partição (QuickSort)
  - BucketSort

Escola de Engenharia
Universidade do Minho

Introdução

- **Algoritmo: conjunto claramente especificado de instruções a seguir para resolver um problema**
  - Noção de algoritmo muito próxima da noção de programa (imperativo)
  - O mesmo algoritmo pode ser "implementado" por muitos programas de computador diferentes
  - O mesmo problema pode ser resolvido por muitos algoritmos diferentes
- **Descrição de algoritmos:**
  - em linguagem natural, pseudo-código, numa linguagem de programação, etc.

Escola de Engenharia
Universidade do Minho

Pesquisa Sequencial

- **Problema (pesquisa de valor em array):**
  - Verificar se um valor existe num array e, no caso de existir, indicar a sua posição.
  - Possíveis variantes para o caso de arrays com valores repetidos:
    - (a) indicar a posição da primeira ocorrência
    - (b) indicar a posição da última ocorrência
    - (c) indicar a posição de uma ocorrência qualquer

Pesquisa Sequencial

- **Algoritmo (pesquisa sequencial):**
  - Percorrer sequencialmente todas as posições do array, da primeira para a última [(a)] ou da última para a primeira [(b)], até encontrar o valor pretendido ou chegar ao fim do array
    - [(a)] caso se pretenda saber a posição da primeira ocorrência
    - [(b)] caso se pretenda saber a posição da última ocorrência
- **Adequado para arrays não ordenados ou pequenos**

Implementação da Pesquisa Sequencial em C++

```
/* Procura um valor x num array v de n inteiros (n>=0).
Retorna o índice da primeira ocorrência de x em v, se
encontrar; senão, retorna -1. */

int SequentialSearch(const int v[], int n, int x)
{
   for (int i = 0; i < n; i++)
      if (v[i] == x) return i; // encontrou

   return -1; // não encontrou
}
```

Exemplo de aplicação: totoloto

```
// Programa para gerar aleatoriamente uma aposta do totoloto

#include <iostream>
#include <cstdlib>   // para usar rand() e srand()

// Constantes
const int TAMANHO_APOSTA = 6;
const int MAIOR_NUMERO = 49;

// Gera inteiro aleatório entre a e b
int rand_between(int a, int b)
{
   return (rand() % (b - a + 1)) + a;
   // Nota: rand() gera um inteiro entre 0 e RAND_MAX
}
```


Exemplo de aplicação: totoloto

```
// Verifica se existe o valor x nas primeiras n posições
// do array v
inline bool existe(int x, const int a[], int n)
{
   return SequentialSearch(a, n, x) != -1;
}

// Preenche a aposta ap com valores aleatórios sem repetições
void geraAposta(int ap[])
{
    for (int i = 0; i <= TAMANHO_APOSTA - 1; i++)
       do
          ap[i] = rand_between(1, MAIOR_NUMERO);
       while ( existe(ap[i], ap, i) );
}
```

Exemplo de aplicação: totoloto

```
// Imprime a aposta ap
void imprimeAposta(const int ap[])
{
    cout << "A aposta gerada é: ";
    for (int i = 0; i <= TAMANHO_APOSTA - 1; i++)
       cout << ap[i] << ' ';
    cout << endl;
}

main()
{
    int aposta[TAMANHO_APOSTA];
    srand(time(0)); /* Inicializa gerador de numeros pseudo-
                       aleatórios com valor diferente de
                       cada vez que o programa corre */
    geraAposta(aposta);
    imprimeAposta(aposta);
    return 0;
}
```


Funções `inline` em C++

- Qualificador `inline` antes do nome do tipo retornado por uma função: "aconselha" o compilador a gerar uma cópia do código da função no lugar em que é chamada
- Evita o peso do mecanismo de chamada de funções
- Exemplo: estando definida uma função

```
inline int max(int a, int b)
    { return a > b? a : b; }
```

  o compilador pode converter uma chamada do género

```
x = max(z, y);
```

  em algo do género:

```
x = z > y ? z : y;
```

- Usar só com funções pequenas, porque o código da função é replicado em todos os sítios em que a função é chamada!
- Dispensa uso de macros!

Nomes de funções sobrecarregados
(overloaded)

- Podem-se definir várias funções com o mesmo nome, desde que as suas listas de argumentos sejam suficientemente diferentes (em número ou tipo) para que se possa determinar a que função se refere cada chamada
  - Diz-se que o nome está sobrecarregado ou "overloaded"
  - Não basta que as funções difiram no tipo de retorno

```
// Protótipos
double abs(double);
int abs(int);
double abs(double x, double y); // sqrt(sqr(x)+sqr(y))

// Chamadas
abs(1);      // chama abs(int);
abs(1.0);    // chama abs(double);
abs(2, 3.0); // chama abs(double, double);
```


Templates de funções

- **Um *template* de funções define uma família ilimitada de funções com o mesmo nome (*overloaded*), que só diferem em nomes de tipos (parâmetros da definição)**
  - Sintaxe: Preceder cabeçalho da função da palavra chave `template` seguida da lista de parâmetros formais entre `< >`. Cada parâmetro é precedido da palavra chave `class` (no sentido de "tipo de dados").
  - O compilador gera funções individuais de acordo com as chamadas efetuadas (instanciando os parâmetros do padrão de acordo com os tipos dos argumentos atuais passados)

```
// Dá o maior de 2 valores do tipo T (comparáveis com ">")
template <class T>  T  max(T a, T b)
{ return a > b? a : b; }

// Chamadas
max(1, 2);     // gera e chama max(int, int)
max('a', 'b'); // gera e chama max(char, char)
max(1, 'a');   // Erro: não consegue gerar max(int, char)
```

Implementação Genérica da Pesquisa Sequencial em C++

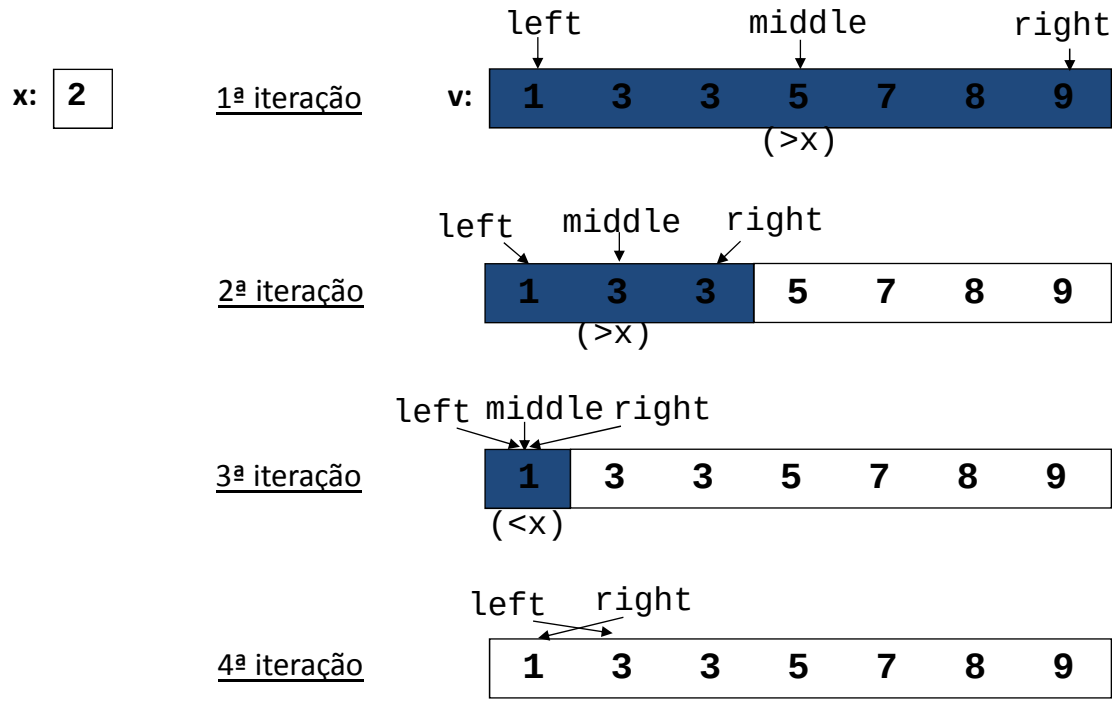
- ***Template* de função em C++, na variante (a):**

```
/* Procura um valor x num array v de n elementos (n>=0)
comparáveis com os operadores de comparação. Retorna o
índice da primeira ocorrência de x em v, se encontrar;
senão, retorna -1. */

template <class T>
int SequentialSearch(const T v[], int n, T x)
{
   for (int i = 0; i < n; i++)
      if (v[i] == x) return i; // encontrou

   return -1; // não encontrou
}
```


Pesquisa Binária

- **Problema (pesquisa de valor em array ordenado): verificar se um valor (*x*) existe num array (*v*) previamente ordenado e, no caso de existir, indicar a sua posição**
  - no caso de arrays com valores repetidos, consideramos a variante em que basta indicar a posição de uma ocorrência qualquer (as outras ocorrências do mesmo valor estão em posições contíguas)
- **Algoritmo (pesquisa binária):**
  - Comparar o valor que se encontra a meio do array com o valor procurado, podendo acontecer uma de três coisas:
    - é igual ao valor procurado⇒ está encontrado
    - é maior do que o valor procurado ⇒ continuar a procurar (do mesmo modo) no sub-array à esquerda da posição inspeccionada
    - é menor do que o valor procurado ⇒ continuar a procurar (do mesmo modo) no sub-array à direita da posição inspeccionada.
  - Se o array a inspeccionar se reduzir a um array vazio, conclui-se que o valor procurado não existe no array inicial.

Exemplo de Pesquisa Binária



array a inspeccionar vazio ⇒ o valor 2 não existe no array inicial!

Implementação da Pesquisa Binária em C++

```
/* Procura um valor x num array v de tamanho n previamente
ordenado. Retorna o índice de uma ocorrência de x em v, se
encontrar; senão, retorna -1. Supõe que os elementos do array
são comparáveis com os operadores de comparação. */

template <class T> int BinarySearch(const T v[], int n, T x)
{
    int left = 0, right = n - 1;
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if (x == v[middle]) return middle; // encontrou
        else if (x > v[middle]) left = middle + 1;
        else right = middle - 1;
    }
    return -1; // não encontrou
}
```

Ordenação de Vetores

- **Problema (*ordenação de vector*)**
  - Dado um vector (v) com **N** elementos, rearranjar esses elementos por ordem crescente (ou melhor, por ordem não decrescente, porque podem existir valores repetidos)
- **Ideias base:**
  - Existem diversos algoritmos de ordenação ("sorting") com complexidade **O(N^2) -** por exemplo Ordenação por Inserção, BubbleSort ou Shellsort que são muito simples
  - Existem algoritmos de ordenação mais difíceis de codificar que têm complexidade **O(N log N)**

Algoritmos de Ordenação de Vetores

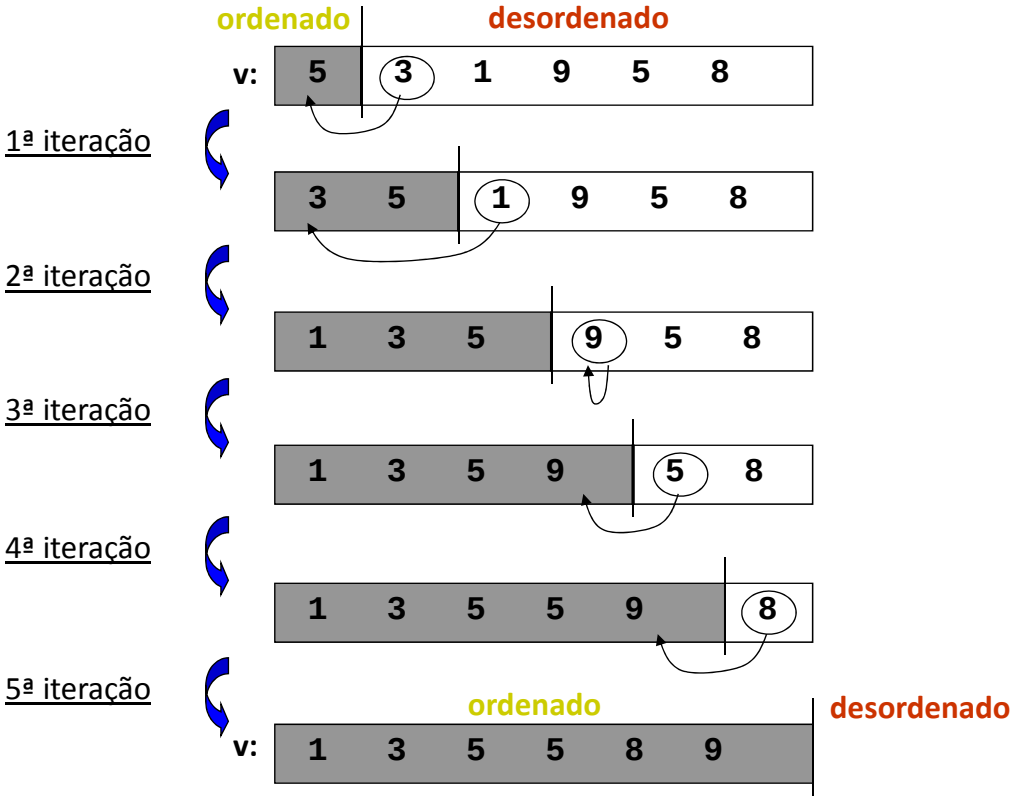
- **Algoritmos:**
  - Ordenação por Inserção
  - Ordenação por Selecção
  - BubbleSort
  - ShellSort
  - MergeSort
  - Ordenação por Partição (QuickSort)
  - BucketSort

Ordenação por Inserção

- **N-1 passos**
- **Em cada passo *p* coloca um elemento na ordem, sabendo que elementos dos índices inferiores (entre 0 e p-1) já estão ordenados**

- **Algoritmo (*ordenação por inserção*):**
  - Considera-se o vetor dividido em dois sub-vetores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
  - Começa-se com um elemento apenas no sub-vetor da esquerda
  - Move-se um elemento de cada vez do sub-vetor da direita para o sub-vetor da esquerda, inserindo-o na posição correta por forma a manter o sub-vetor da esquerda ordenado
  - Termina-se quando o sub-vetor da direita fica vazio

Exemplo de Ordenação por Inserção



Inserção de valor em vetor ordenado

- **Sub-problema: inserir um valor num vetor ordenado (mantendo-o ordenado)**
- **Solução em C++ (função InsertSorted) :**

```
/* Insere um valor x num array vec com n elementos ordenados.
Após a inserção, o array fica com n+1 elementos ordenados.
Retorna a posição em que inseriu o valor (entre 0 e n). */
template <class T>
int InsertSorted(T vec[], int n, T x)
{
   int j;
   // procura posição (j) de inserção da direita (posição n)
   // para a esquerda (posição 0), e ao mesmo tempo chega
   // elementos à direita (podia usar o próprio n em vez de j)
   for (j = n; j > 0 && x < vec[j-1]; j--)
      vec[j] = vec[j-1];
   vec[j] = x; // insere agora
   return j; // retorna a posição em que inseriu
}
```


Implementação da Ordenação por Inserção em C++

- A função em C++ para ordenar um array pelo método de inserção é agora trivial

```
// Ordena array vec de n elementos: vec[0]≤... ≤vec[n-1]
template <class T> void InsertionSort(T vec[], int n)
{
    // i - tamanho do sub-array esquerdo ordenado
    for (int i = 1; i < n; i++)
        InsertSorted(vec, i , vec[i]);
}
```

- **Juntando tudo (para ser mais eficiente) ...**

```
template <class T> void InsertionSort(T vec[], int n)
{
   for (int i = 1; i < n; i++) {
      T x = vec[i];
      for (int j = i; j > 0 && x < vec[j-1]; j--)
         vec[j] = vec[j-1];
      vec[j] = x;
   }
}
```

Ordenação por Inserção usando `vector`

```
// Ordena elementos do vector vec. Comparable: deve conter
// construtor cópia, operadores igualdade (=,<,>)

template <class Comparable>
void InsertionSort(vector<Comparable> &vec)
{
    for (int p = 1; p < vec.size(); p++)
    {
       Comparable tmp = vec[p];
       int j;
       for (j = p; j > 0 && tmp < vec[j-1]; j--)
          vec[j] = vec[j-1];
       vec[j] = tmp;
    }
}
```


Análise da Ordenação por Inserção

- 2 ciclos encaixados, cada um pode ter N iterações :
  - $O(N^2)$
- Caso mais desfavorável: vector em ordem inversa
  - $O(N^2)$
- Caso mais favorável: vector já ordenado
  - $O(N)$
- Conclusão:
  - Só pode ser utilizado para vetores pequenos…
    (ver slides sobre complexidade de algoritmos)

Ordenação por Seleção

- **Provavelmente é o algoritmo mais intuitivo:**
  - Encontrar o mínimo do vetor
  - Trocar com o primeiro elemento
  - Continuar para o resto do vetor (excluindo o primeiro)
- **2 ciclos encaixados, cada um pode ter N iterações :**
  - **Complexidade $O(N^2)$**
- **Variantes:**
  - "Stable Sort" – Insere mínimo na primeira posição (em vez de realizar a troca)
  - "Shaker Sort" – Procura máximo e mínimo em cada iteração (Selecção bidireccional)

Ordenação por Seleção

Algoritmo em C:

```
// Ordena array vec de n elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]
// usando ordenação por seleção

void selectionSort(int vec[], int tam)
{
 int i, j, min, aux;
 for (i=0; i<tam-1; i++) {
   min = i;
   for (j=i+1; j<tam; j++) {
     if (vec[j] < vec[min]) min = j;
   }
   aux = vec[i];
   vec[i] = vec[min];
   vec[min] = aux;
 }
}
```

Ordenação por Seleção

Algoritmo em C++:

// Ordena array vec de n elementos, ficando vec[0]≤.. ≤vec[n-1]
// usando ordenação por seleção

```
template<class T>
void selection_sort(vector<T> &vec )
{
   vector<T>::iterator it1;
  for(it1 = vec.begin(); it1 != vec.end()-1; ++it1 ){
       iter_swap(it1, min_element(it1, vec.end()));
  }
}
```


BubbleSort

- **Algoritmo de Ordenação "BubbleSort" ("Exchange Sort"):**
  - Compara elementos adjacentes. Se o segundo for menor do que o primeiro, troca-os
  - Fazer isto desde o primeiro até ao último par
  - Repetir para todos os elementos excepto o último (que já está correcto)
  - Repetir, usando menos um par em cada iteração até não haver mais pares (ou não haver trocas)

- **Diversas variantes**
- **2 ciclos encaixados, cada um pode ter N iterações:**
  - **Complexidade $O(N^2)$**

BubbleSort

Algoritmo em C:

// Ordena array vec de n elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]
// usando BubbleSort

```
void bubble(int vec[], int n)
  {
    int troca, i, j, aux;
    for (i = n-1; i > 0; i--) {
              // o maior valor entre vec[0] e vec[i] vai para a posição vec[i]
      troca = 0;
      for (j =0 ; j < i; j++)  {
        if (vec[j] > vec[j+1]) {
          aux = vec[j]; vec[j] = vec[j+1]; vec[j+1] = aux;
          troca = 1;
        }
      }
      if (!troca) return;
    }
  }
```


BubbleSort

Algoritmo em C++:

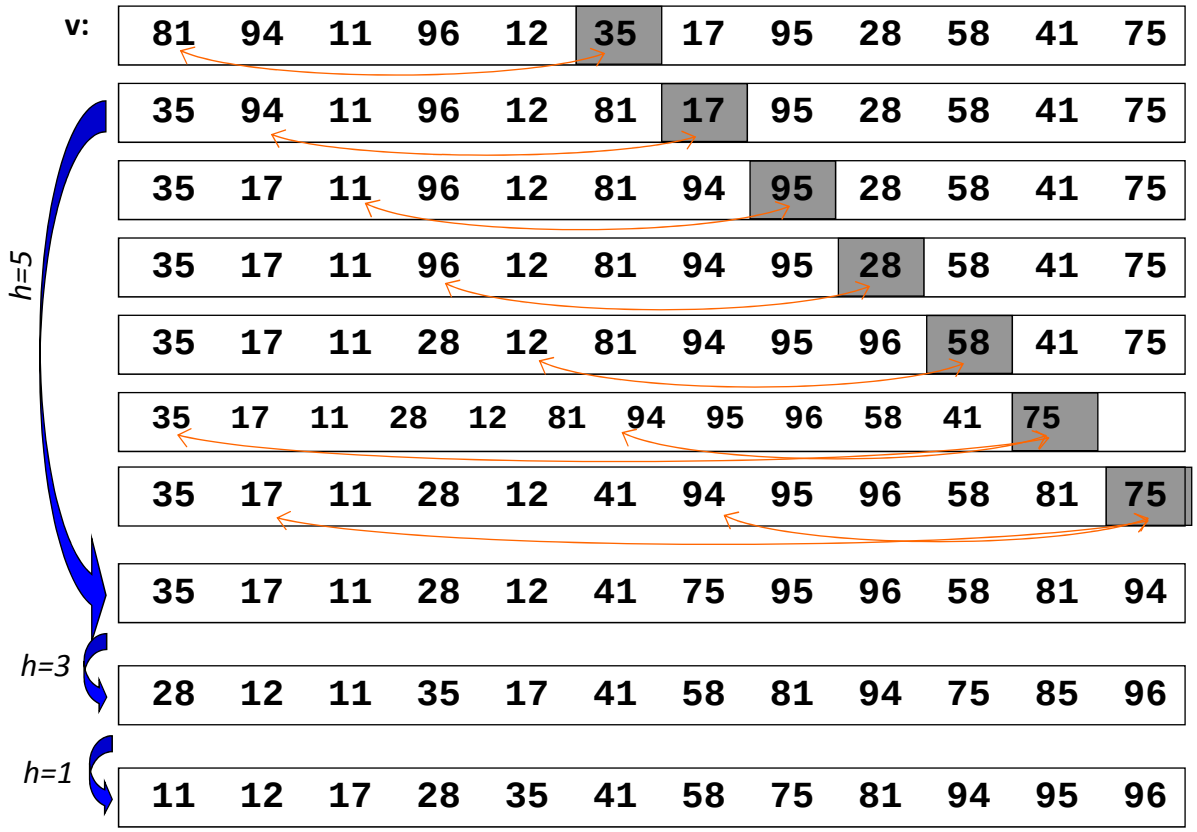
// Ordena array vec de n elementos inteiros, ficando vec[0]≤.. ≤vec[n-1]
// usando BubbleSort

```
template<class T>
void bubble_sort(vector<T> &vec )
{
  vector<T>::reverse_iterator it1;
  for(it1 = vec.rbegin(); it1 != vec.rend(); ++it1 ) {
    vector<T>::iterator it2;
    bool troca = false;
    for(it2 = vec.begin(); &*it2 != &*it1; ++it2 )  {
      if (*it2 > *(it2+1)) {
        iter_swap(it2, it2+1);
        troca = true;
      }
    }
    if (!troca) return;
  }
}
```

ShellSort

- **Compara elementos distantes**
- **Distância entre elementos comparados vai diminuindo, até que a comparação seja sobre elementos adjacentes**
  - Usa a sequência $h_1, h_2, \ldots, h_t$ $(h_1=1)$
  - Em determinado passo, usando incremento h_k, todos os elementos separados da distância h_k estão ordenados, $vec[i] \leq vec[i+h_k]$
- **Sequência de incrementos:**
  - Shell: mais popular, não mais eficiente $O(N^2)$
    - $h_t = N/2$, $h_k = h_{k+1}/2$
  - Hibbard: incrementos consecutivos não têm factores comuns
    - $h = 1, 3, 7, \ldots, 2^k-1$ $O(N^{5/4})$

ShellSort (h = {5, 3, 1})



ShellSort

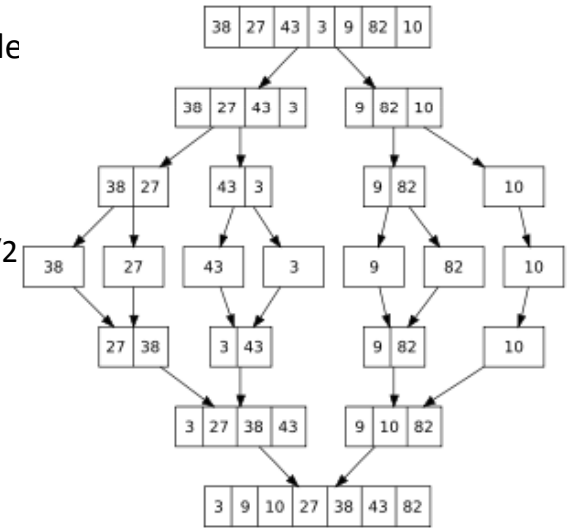
```
// ShellSort - com incrementos de Shell
template <class Comparable>
void ShellSort(vector<Comparable> &vec)
{
    int j;
    for (int gap = vec.size()/2; gap > 0; gap /= 2)
       for (int i = gap; i < vec.size(); i++)
       {
          Comparable tmp = vec[i];
          for (j = i; j>gap && tmp<vec[j-gap]; j -= gap)
             vec[j] = vec[j-gap];
          vec[j] = tmp;
       }
}
```


ShellSort

```
// ShellSort genérico. Realiza Ordenação por Inserção nos elementos
   de vec[] com uma dada distância – gap. Se gap=1 faz Ordenação por
   inserção normal

void shellSortPhase(int vec[], int tam, int gap) {
   for (int i = gap; i < tam; ++i) {
        int value = vec[i];
        for (int j = i - gap; j >= 0 && vec[j] > value; j -= gap)
               vec[j + gap] = vec[j];
       vec[j + gap] = value;
   }
}

void shellSort(int vec[], size_t length) {
// gaps[ ] deve ser aproximadamente uma progressão geométrica. A
   sequência apresentada é a melhor

   static const int gaps[] = {1, 4, 10, 23, 57, 132, 301, 701};
   for (int sInd = sizeof(gaps)/sizeof(gaps[0])-1; sInd>=0; --sInd)
        shellSortPhase(vec, length, gaps[sInd]);
}
```

MergeSort

- **Abordagem recursiva**
  - Divide-se o vetor ao meio
  - Ordena-se cada metade (usando MergeSort recursivamente)
  - Fundem-se as duas metades já ordenadas
- **Divisão e Conquista:**
  - Problema é dividido em dois de metade do tamanho
- **Análise**
  - Tempo execução: O(N logN)
    - 2 chamadas recursivas de tamanho N/2
    - Operação de junção de vectores: O(N)
- **Inconveniente**
  - fusão de vectores requer espaço extra linear



Fonte: Wikipedia

MergeSort

```
//Algoritmo em C++:
template <class Comparable>
void mergeSort(vector <Comparable> &vec)
{
    vector<Comparable> tmpVec(vec.size());
    mergeSort(vec, tmpVec, 0, vec.size()-1);
}

/* Método que realiza chamadas recursivas:
   vec é um vector de elementos do tipo Comparable.
   tmpVec é um vector para colocar o resultado da fusão (merge)
   left (right) é o elemento mais à esquerda(direita) do sub vector */
template <class Comparable>
void mergeSort(vector <Comparable> &vec,
        vector<Comparable> &tmpVec, int left, int right)
{
    if (left < right) {
       int center = (left + right)/2;   //divide ao meio
       mergeSort(vec, tmpVec, left, center);  // ordena 1ª metade
       mergeSort(vec, tmpVec, center + 1, right); // ordena 2ª metade
       merge(vec, tmpVec, left, center +1, right); // Junta metades
    }
}
```

MergeSort

```
// Algoritmo de fusão dos dois vectores do MergeSort
template <class Comparable>
void merge(vector <Comparable> &vec, vector<Comparable> &tmpVec,
           int leftPos, int rightPos, int rightEnd)
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    while (leftPos <= leftEnd && rightPos <= rightEnd )
       if (vec[leftPos] <= vec[rightPos] )
          tmpVec[tmpPos++] = vec[leftPos++];
       else
          tmpVec[tmpPos++] = vec[rightPos++];
    while (leftPos<=leftEnd) tmpVec[tmpPos++] = vec[leftPos++];
    while (rightPos<=rightEnd) tmpVec[tmpPos++] = vec[rightPos++];
    for ( int i = 0; i < num Elements; i++, rightEnd-- )
       vec[rightEnd] = tmpVec[rightEnd];
}
```


Ordenação por Partição (Quick Sort)

- **Algoritmo (ordenação por partição):**
  1. Caso básico: Se o número (n) de elementos do vector (v) a ordenar for 0 ou 1, não é preciso fazer nada
  2. Passo de partição:
     2.1. Escolher um elemento arbitrário (**x**) do vector (chamado pivot)
     2.2. Partir o vector inicial em dois sub-vectores (esquerdo e direito), com valores $\leq$ **x** no sub-vector esquerdo e valores $\geq$ **x** no sub-vector direito (podendo existir um 3º sub-vector central com valores =**x**)
  3. Passo recursivo: Ordenar os sub-vectores esquerdo e direito, usando o mesmo método recursivamente
- **Algoritmo recursivo baseado na técnica *divisão e conquista***

Refinamento do Passo de Partição

(Supõe-se excluído pelo menos o caso em que n = 0)

2. <u>Passo de partição:</u>

2.1. Escolher para pivot (x) o elemento do meio do vector (*vec[n/2]*)

2.2. Inicializar *i = 0* (índice da 1º posição do vector)

Inicializar *j = n-1* (índice da última posição do vector)

2.3. Enquanto *i ≤ j*, fazer:

2.3.1. Enquanto *vec[i] < x* (é sempre *i≤n-1*!), incrementar *i*

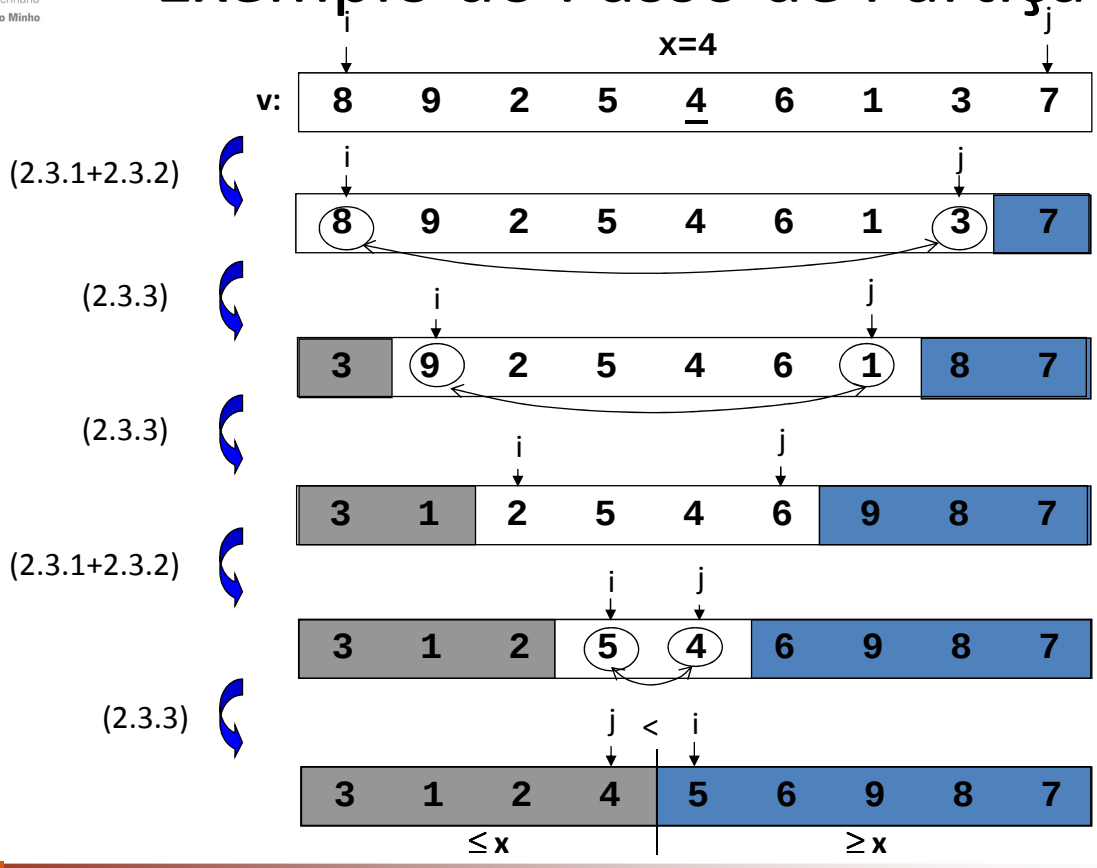
2.3.2. Enquanto *vec[j] > x* (é sempre *j≥0*!), decrementar *j*

2.3.3. Se *i ≤ j então* trocar *vec[i]* com *vec[j]*, incrementar *i* e decrementar *j*

2.4. O sub-vector esquerdo (com valores *≤x*) é *vec[0], ..., vec[j]* (vazio se *j<0*)

2.5. O sub-vector direito (com valores *≥x*) é *vec[i], ..., vec[n-1]* (vazio se *i>n-1*)

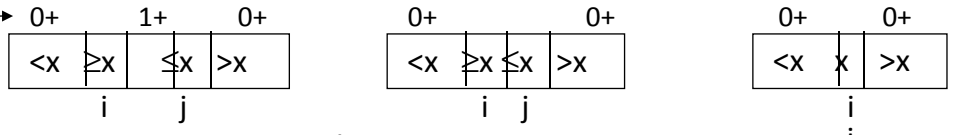
Exemplo do Passo de Partição



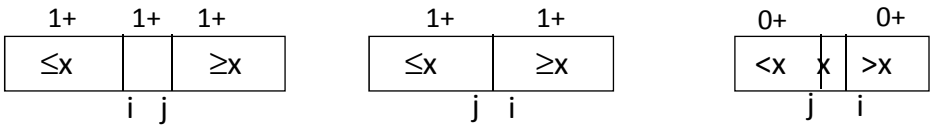
Passo de Partição

- **Da 1ª vez, a condição em 2.3. (i ≤ j) é verdadeira, e acontece o seguinte:**
  - o ciclo 2.3.1. pára com i na posição do pivot ou à esquerda
  - o ciclo 2.3.2. pára com j na posição do pivot ou à direita
  - assim, antes do passo 2.3.3., 0≤ i ≤ j ≤ n-1
  - ilustram-se a seguir os vários situações possíveis antes do passo 2.3.3.:

tamanho de cada segmento (1 por omissão) →



  - uma vez que i≤ j, a troca é realizada, resultando respectivamente as seguintes situações após o passo 2.3.3.:



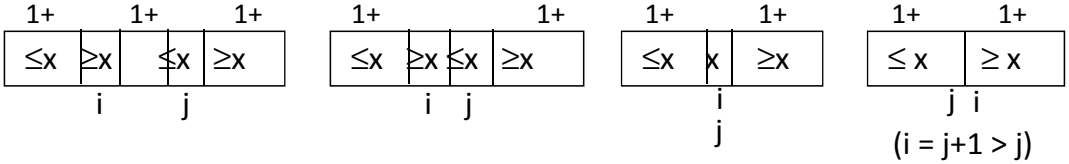
(i ≤ j) CONTINUA!

(i = j+1 > j) FIM!

(i = j+2 > j) FIM!

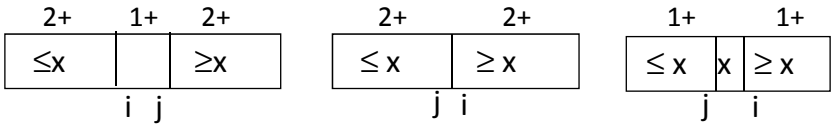
Passo de Partição

- **No caso da esquerda, o ciclo 2.3. é executado 2ª vez, e acontece o seguinte:**
  - o ciclo 2.3.1. pára de certeza, devido à existência de valores ≥ x à direita
  - o ciclo 2.3.2. pára de certeza, devido à existência de valores ≤ x à esquerda
  - antes do passo 2.3.3. verifica-se uma das situações seguintes:



(i = j+1 > j) FIM!

  - nos três casos da esquerda, a troca é realizada, resultando respectivamente as seguintes situações após o passo 2.3.3:



(i ≤ j) CONTINUA!

(i = j+1 > j) FIM!

(i = j+2 > j) FIM!

Passo de Partição (conc.) e Passo de Recursão

- **No caso da esquerda, o ciclo 2.2.3. é executado 3ª vez, e o esquema repete-se.**
- **Conclui-se (por indução) que realmente nunca são acedidos elementos fora do vector durante o ciclo 2.3.**

- **No final do passo de partição, qualquer dos sub-vectores (esquerdo e direito) é de tamanho <n**
  - além disso, os sub-vectores são disjuntos, como convém
- **Assim, as chamadas recursivas do mesmo algoritmo destinam-se a ordenar vectores cada vez mais pequenos e, portanto, cada vez mais próximos do caso básico. Isto garante que a recursão tem fim.**

Implementação da Ordenação por Partição em C++

```
/* Ordena array(vec) entre 2 posições (a e b). Supõe que os elementos
do array são comparáveis com "<" e ">" e copiáveis com "=". */
template <class T> void QuickSort(T vec[], int a, int b)
{
  if (a >= b) return;  // caso básico (tamanho <= 1)
  T x = vec[(a+b)/2];
  int i = a, j = b;
   // passo de partição
   do {
        while (vec[i] < x) i++;
        while (vec[j] > x) j--;
        if (i > j) break;
        T tmp=vec[i]; vec[i]=vec[j]; vec[j]=tmp;
        i++; j--; //troca
    } while (i <= j);
   // passo recursivo
    QuickSort(vec, a, j);
    QuickSort(vec, i, b);
}
```

Programa de teste

```
#include <iostream>
#include <stdlib>  // Para usar rand()
// inserir aqui a função QuickSort implementada

void imprime(const int vec[], int n)
{
    for (int i = 0; i < n; i++)
        cout << vec[i] << ' ';
    cout << '\n';
}
main()
{
    const int SIZE = 10000;
    int vec[SIZE];
    for (int i = 0; i < SIZE; i++)
         vec[i] = rand();
    imprime(vec, SIZE);
    QuickSort(vec, 0, SIZE - 1);
    imprime(vec, SIZE);
    return 0;
}
```


Análise da Ordenação por Partição

- **Escolha pivot determina eficiência**
  - *pior caso*: pivot é elemento mais pequeno
    O(N^2)
  - *melhor caso*: pivot é elemento médio
    O(N logN)
  - *caso médio*: pivot corta vector arbitrariamente
    O(N logN)

- **Escolha do pivot**
  - má escolha: extremos do vector (O(N^2) se vector ordenado)
  - aleatório: envolve mais uma função pesada
  - recomendado: mediana de três elementos (extremos do vector e ponto médio)

BucketSort (BinSort)

- **Ordenação linear**
  - usa informação adicional sobre entrada
- **Algoritmo**
  - vector de entrada: inteiros positivos inferiores a **M**

    Vec = [V_1, V_2, …, V_N] ; $V_i < M$
  - inicializar um vector de **M** posições a **0's**

    count = [c_1, c_2, …, c_M] ; $c_j = 0$
  - Ler vector entrada (*Vec*) e para cada valor incrementar a posição respectiva no vector *count* : *count[Vi]++*
  - Produzir saída lendo o vector *count*
- **Eficiência**
  - tempo linear

Algoritmos de Ordenação

- **Cada ponto corresponde à ordenação de 100 vetores de inteiros gerados aleatoriamente** (Fonte: Sahni, "Data Structures, Algorithms and Applications in C++")
- **Método de ordenação por partição (*quickSort) é na prática o mais* eficiente, excepto para arrays pequenos (até cerca 20 elementos), em que o método de ordenação por inserção (*insertionSort) é melhor!***

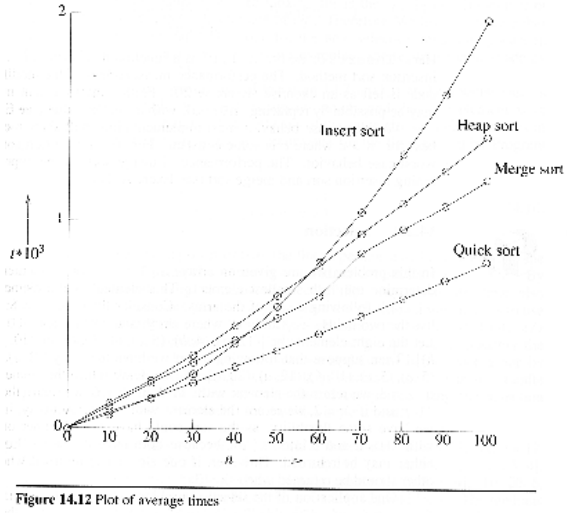


Figure 14.12 Plot of average times

Algoritmos Find da STL

- **Pesquisa sequencial em vectores:**

```
iterator find(iterator start, iterator end, const TYPE& val);
```

  - procura a 1ª ocorrência entre *[start, end[de um elemento idêntico a val* (comparação efectuada pelo operador ==).
    - sucesso, retorna iterador para o elemento encontrado
    - não sucesso, retorna iterador para fim do vector (*v.end())*
- **iterator find_if(iterator start, iterator end, Predicate up);**
  - procura a 1ª ocorrência entre *[start, end[para a qual o predicado unário up retorna verdadeiro.*
- **Pesquisa binária em vectores:**

```
bool binary_search(iterator start, iterator end, const TYPE& val );
bool binary_search(iterator start, iterator end, const TYPE& val, Comp f);
```

    - necessário implementar o operador <

Algoritmos Sort da STL

- **Ordenação de vectores:**
  - `void sort(iterator start, iterator end);`
    - ordena os elementos do vector entre *[start, end[por ordem ascendente,* usando o operador <
  - `void sort(iterator start, iterator end, StrictWeakOrdering cmp);`
    - ordena os elementos do vector entre *[start, end[por ordem ascendente,* usando a função *StrictWeakOrdering*
  - Algoritmo de ordenação implementado em *sort() é o algoritmo introsort, possui complexidade O(N logN)*

Algoritmos Sort e Find da STL (exemplo)

```
class Pessoa {
   string BI;
   string nome;
   int idade;
public:
   Pessoa (string BI, string nm="", int id=0);
   string getBI() const;
   string getNome() const;
   int getIdade() const;
   bool operator < (const Pessoa & p2) const;
   bool operator == (const Pessoa & p2) const;
};

Pessoa::Pessoa(string b, string nm, int id):
   BI(b),nome(nm), idade(id) {}

string Pessoa::getBI() const { return BI; }
string Pessoa::getNome() const { return nome; }
int Pessoa::getIdade() const { return idade; }
```


Algoritmos Sort e Find da STL (exemplo)

```
bool Pessoa::operator < (const Pessoa & p2) const
{
   return nome < p2.nome;
}

bool Pessoa::operator == (const Pessoa & p2) const
{
   return BI == p2.BI;
}

ostream & operator << (ostream &os, const Pessoa & p)
{
   os << "(BI: " << p.getBI() << ", nome: " << p.getNome()
   << ", idade: " << p.getIdade() << ")";
   return os;
}
```

Algoritmos Sort e Find da STL (exemplo)

```
template <class T>
void write_vector(vector<T> &v)
{
   for (unsigned int i=0 ; i < v.size(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
   cout << endl;
}

bool compPessoa(const Pessoa &p1, const Pessoa &p2)
{
   return p1.getIdade() < p2.getIdade();
}

bool eAdolescente(const Pessoa &p1)
{
   return p1.getIdade() <= 20;
}
```


Algoritmos Sort e Find da STL (exemplo)

```
template <class T>
void write_vector(vector<T> &v)
{
   for (unsigned int i=0 ; i < v.size(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
   cout << endl;
}

bool compPessoa(const Pessoa &p1, const Pessoa &p2)
{
   return p1.getIdade() < p2.getIdade();
}

bool eAdolescente(const Pessoa &p1)
{
   return p1.getIdade() <= 20;
}
```

Algoritmos Sort e Find da STL (exemplo)

```
int main()
{
   vector<Pessoa> vp;
   vp.push_back(Pessoa("6666666","Rui Silva",34));
   vp.push_back(Pessoa("7777777","Antonio Matos",24));
   vp.push_back(Pessoa("1234567","Maria Barros",20));
   vp.push_back(Pessoa("7654321","Carlos Sousa",18));
   vp.push_back(Pessoa("3333333","Fernando Cardoso",33));
   vector<Pessoa> vp1=vp;
   vector<Pessoa> vp2=vp;
   cout << "vector inicial:" << endl;
   write_vector(vp);
}
```

```
vector inicial:
v[0] = (BI: 6666666, nome: Rui Silva, idade: 34)
v[1] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[2] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[3] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[4] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
```


Algoritmos Sort e Find da STL (exemplo)

```
sort(vp1.begin(),vp1.end());
cout << "Apos 'sort' usando 'operador <':“ << endl;
write_vector(vp1);
```

```
Apos 'sort' usando 'operador <':
v[0] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[1] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[2] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
v[3] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[4] = (BI: 6666666, nome: Rui Silva, idade: 34)
```

```
sort(vp2.begin(),vp2.end(),compPessoa);
cout << "Apos 'sort' usando funcao de comparacao:" << endl;
write_vector(vp2);}
```

```
Apos 'sort' usando funcao de comparacao:
v[0] = (BI: 7654321, nome: Carlos Sousa, idade: 18)
v[1] = (BI: 1234567, nome: Maria Barros, idade: 20)
v[2] = (BI: 7777777, nome: Antonio Matos, idade: 24)
v[3] = (BI: 3333333, nome: Fernando Cardoso, idade: 33)
v[4] = (BI: 6666666, nome: Rui Silva, idade: 34)
```

Algoritmos Sort e Find da STL (exemplo)

```
Pessoa px("7654321");
vector<Pessoa>::iterator it = find(vp.begin(),vp.end(),px);
if (it==vp.end())
   cout << "Pessoa " << px << " nao existe no vector " << endl;
else
   cout << "Pessoa " << px << " existe no vector como: " << *it
        << endl;
```

```
Pessoa (BI: 7654321, nome: , idade: 0) existe no vector
como: (BI: 7654321, nome: Carlos Sousa, idade: 18)
```

```
it = find_if(vp.begin(),vp.end(),eAdolescente);
if (it==vp.end())
   cout << "Pessoa adolescente nao existe no vector " << endl;
else
   cout << "pessoa adolescente encontrada: " << *it << endl;
}
```

```
pessoa adolescente encontrada: (BI: 1234567, nome: Maria
Barros, idade: 20)
```


Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 8

Pesquisa e Ordenação de Vetores

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Reis, Rocha e Faria, 2007)