3. Vectores: Algoritmos de Pesquisa

João Pascoal Faria (versão original)
Ana Paula Rocha (versão 2004/2005)
Luís Paulo Reis (versão 2005/2006)

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.

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 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.h>
#include <stdlib.h>   // para usar rand() e srand()
#include <time.h>     // para usar time()

// 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
}
```

Sugestão: começar a ler o programa do fim para o princípio!

Exemplo de aplicação: totoloto (cont.)

```
// Verifica se existe o valor x nas primeiras n posições
// do array v
// (inline: explicado adiante)
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) );
}
```

7

Exemplo de aplicação: totoloto (concl.)

```
// 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 << '\n';
}

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;
}
```

8

Nota sobre 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!

9

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);
```

10

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 efectuadas (instanciando os parâmetros do padrão de acordo com os tipos dos argumentos actuais 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)
```

11

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
}
```

12

Pesquisa Binária

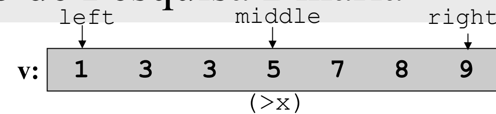
- Problema (*pesquisa de valor em array <u>ordenado</u>*): verificar se um valor (*x*) existe num array (*v*) previamente ordenado e, no caso de existir, indicar a sua posição
  – no caso de arrays com valores repetidos, consideramos a variante em que basta indicar a posição de uma ocorrência qualquer (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.

13

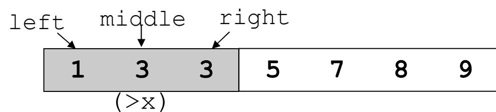
Exemplo de Pesquisa Binária

x: 2

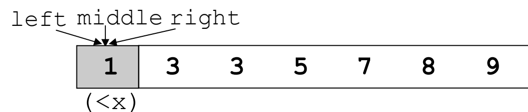
<u>1ª iteração</u> **v:**



<u>2ª iteração</u>



<u>3ª iteração</u>



<u>4ª iteração</u>

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

14

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
}
```