UAlg FCT

UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Programação Imperativa

Lição n.º 12

Arrays ordenados

Arrays ordenados

- Renque.
- Busca dicotómica.
- Método da bisseção.

Problema do renque

- O renque de um valor num array é o número de elementos do array cujo valor é menor que esse valor:

```
int ints_rank_general(const int *a, int n, int x);
```

```
void unit_test_rank_general(void)
{
  int a[10] = {8,3,9,8,4, 4,2,7,5,7};
  assert(ints_rank_general(a, 10, 5) == 4);
  assert(ints_rank_general(a, 10, 12) == 10);
  assert(ints_rank_general(a, 10, 1) == 0);
  assert(ints_rank_general(a, 10, 2) == 0);
  assert(ints_rank_general(a, 10, 3) == 1);
  assert(ints_rank_general(a, 10, 7) == 5);
  assert(ints_rank_general(a, 5, 7) == 2);
  assert(ints_rank_general(a, 5, 3) == 0);
  assert(ints_rank_general(a, 5, 9) == 4);
  assert(ints_rank_general(a, 1, 5) == 0);
  assert(ints_rank_general(a, 1, 9) == 1);
  assert(ints_rank_general(a, 0, 5) == 0);
}
```

Chamamos renque geral porque se pode aplicar a um array qualquer. Para arrays ordenados, usaremos funções especializadas.

Renque geral

- Programa-se nas calmas:

```
int ints_rank_general(const int *a, int n, int x)
{
  int result = 0;
  for (int i = 0; i < n; i++)
    if (a[i] < x)
      result++;
  return result;
}
```

É uma variante da função de contagem.

- A versão recursiva também é interessante:

```
int ints_rank_general_r(const int *a, int n, int x)
{
  int result = 0;
  if (n > 0)
    result = (*a < x) + ints_rank_general_r(a+1, n-1, x);
  return result;
}
```

Note bem: o valor aritmético das expressões lógicas é 0 ou 1.

Arrays ordenados

- Um array está ordenado se o valor de cada elemento é menor ou igual ao valor do elemento seguinte.
- Essa propriedade é representada pela seguinte função lógica:

```
int ints_is_sorted(int *a, int n);
```

```
void unit_test_ints_is_sorted(void)
{
  int a[10] = {1,2,5,5,5, 6,8,8,9,9};
  assert(ints_is_sorted(a, 10));
  assert(ints_is_sorted(a, 1));
  assert(ints_is_sorted(a, 0));
  int b[10] = {3,5,5,2,4, 4,8,8,2,5};
  assert(!ints_is_sorted(b, 10));
  assert(ints_is_sorted(b, 3));
  assert(!ints_is_sorted(b, 5));
  assert(ints_is_sorted(b+3, 5));
  assert(!ints_is_sorted(b+5, 5));
}
```

Função is_sorted

- Procura-se o primeiro par de elementos consecutivos fora de ordem:

```
int ints_is_sorted(int *a, int n)
{
  for (int i = 1; i < n; i++)
    if (a[i-1] > a[i])
      return 0;
  return 1;
}
```

- Também a versão recursiva:

```
int ints_is_sorted_r(int *a, int n)
{
  return n <= 1 || (*a <= a[1] && ints_is_sorted_r(a+1, n-1));
}
```

Um array está ordenado se o seu tamanho for menor ou igual a 1 ou, sendo maior que 1, se o valor do primeiro elemento for menor ou igual ao do segundo e o resto do array (tirando o primeiro elemento) estiver ordenado.

Renque em arrays ordenados

- O renque geral inspeciona todos os elementos do array e conta aqueles que são menores que o valor dado.
- Se o array estiver ordenado, conseguimos calcular o renque sem inspecionar os elementos todos.
- Aliás, conseguimos fazê-lo inspecionando relativamente poucos elementos.
- Vejamos como.
- Por hipótese, temos um array a, ordenado, com tamanho n, e é dado um valor x: queremos calcular o número de elementos de a cujo valor é menor que x.

Calculando o renque em arrays ordenados

- Tomemos um elemento qualquer de a, a[m].
- Se x<=a[m], então x<=a[m+1], x<=a[m+2], etc., pois o array está ordenado; logo, todos os elementos de a cujo valor é menor que x estão à esquerda de a[m].
- Inversamente, se x>a[m], então x>a[m-1], x>a[m-2], etc., porque o array está ordenado; logo, o valor de cada um dos elementos à esquerda de a[m] é menor que x e o valor de a[m] também.
- Sendo assim, no primeiro caso, basta contar os elementos de valor menor que x no subarray inicial com m elementos; no segundo, há pelo menos m+1 elementos com valor menor que x, a que se juntam os elementos menores que x no subarray a+(m+1).

Função ints_rank, recursiva

- Veja com atenção:

```
int ints_rank_r(const int *a, int n, int x)
{
  int result = 0;
  if (n > 0)
  {
    int m = n / 2;
    if (x <= a[m])
      result = ints_rank_r(a, m, x);
    else
      result = m+1 + ints_rank_r(a+m+1, n-(m+1), x);
  }
  return result;
}
```

Por uma questão de simetria, que convém computacionalmente, escolhemos o elemento a[m] a meio do array.

Função ints_rank, iterativa

- Veja com muita atenção:

```
int ints_rank(const int *a, int n, int x)
{
  int result = 0;
  while (n > 0)
  {
    int m = n / 2;
    if (x <= a[m])
      n = m;
    else
    {
      result += m+1;
      a += m+1;
      n -= m+1;
    }
  }
  return result;
}
```

Aqui, a contagem parcial mantém-se e o array encolhe, por assim dizer, pois a segunda metade não interessa.

Aqui, entram na contagem os m+1 elementos à esquerda, pois são todos menores que x, e esses m+1 elementos são excluídos do processamento futuro, por assim dizer (pois já foram contabilizados).

Busca dicotómica

- Queremos agora calcular o índice da primeira ocorrência de um valor dado num array ordenado, devolvendo -1, se não houver.
- Baseamo-nos no renque:

```
int ints_bfind(const int *a, int n, int x)
{
  int r = ints_rank(a, n, x);
  return r < n && a[r] == x ? r : -1;
}
```

Se existirem no array ordenado elementos com valor **x**, eles virão todos de seguida e o primeiro deles estará na posição correspondente ao renque de **x** no array; inversamente, se o valor do elemento nessa posição não for igual a **x**, concluímos que não existe no array nenhum elemento com valor **x**.

Complexidade da busca linear

- A busca linear, implementada pela função ints_find, precisa de inspecionar todos os elementos do array, no pior caso, isto é, quando o valor procurado não existe. Por isso, o trabalho computacional e, consequentemente, o tempo de execução são proporcionais ao tamanho do array, no pior caso.
- Para a busca linear, o melhor caso é aquele em que o elemento procurado é o que está na primeira posição, mas esse caso é contrabalançado por aquele em que o elemento procurado está na última posição.
- Portanto, se o array tiver 1000 elementos, por exemplo, então, nos casos em que o elemento procurado não existe, a comparação a[i] == x realizar-se-á 1000 vezes; nos casos em que existe, realizar-se-á, em média, 500 vezes.

```
int ints_find(const int *a, int n, int x)
{
  for (int i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

Complexidade da busca dicotómica

- Na busca dicotómica, o tamanho do array em observação é dividido ao meio, sensivelmente, em cada passo do ciclo.
- Por exemplo, se o array tiver 1000 elementos, após o primeiro passo do ciclo só nos interessa um subarray com 500 elementos; após o segundo passo, só nos interessa o subarray com 250 elementos; depois 125, 62, 31, 15, 8, 4, 2, 1.
- Quer dizer: o ciclo while terá dado 10 voltas e a comparação `x <= a[m]` terá sido realizada 10 vezes.
- Conclusão: a busca dicotómica é muito melhor que a busca linear.
- No entanto, só dá com array ordenados.

Complexidade logarítmica

- Portanto, em geral, o tempo usado para encontrar um elemento no array, ou decidir que ele não existe, é proporcional ao logaritmo do número de elementos: $T = K * \log_2 N$.
- Por exemplo, se uma busca dicotómica num array com 1000 elementos demorar 200 nanossegundos, num array de 2000 elementos demorará 220 nanossegundos ($\log_2 1000 \approx 10$, $\log_2 2000 \approx 11$) e num array de 1000000 elementos demorará 400 nanossegundos ($\log_2 1000000 \approx 20$).
- Dizemos que a busca dicotómica tem complexidade logarítmica.
- A busca linear tem complexidade linear.

Busca dicotómica clássica

- Frequentemente, a busca dicotómica é programada delimitando o intervalo de busca, por meio de dois índices.
- Em cada passo, o intervalo de busca encolhe para pouco menos de metade:

```
int ints_bfind_classic(const int *a, int n, int x)
{
  int i = 0;
  int j = n-1;
  while (i <= j)
  {
    int m = i + (j-i) / 2;
    if (x < a[m])
      j = m-1;
    else if (x > a[m])
      i = m+1;
    else
      return m;
  }
  return -1;
}
```

Em cada passo, o intervalo de busca é delimitado pelos valores das variáveis i e j.

Note bem: esta função só é equivalente à outra no caso em que os arrays não tem valores duplicados. Para arrays com valores duplicados, a outra dá a posição da primeira ocorrência enquanto esta dá a posição de uma ocorrência, não necessariamente a primeira.

Método da bisseção

- A ideia de dividir ao meio o intervalo de busca em cada passo ocorre em vários problemas.
- Por exemplo: calcular a raiz quadrada de um número positivo, x, maior que 1.
- A raiz existe no intervalo entre 1 e x.
- Observamos o ponto médio do intervalo, m.
- Se m*m for igual a x, ou estiver muito perto de x, tomamos m como raiz quadrada de x.
- Se não, se m*m for maior que x, a raiz existirá no intervalo de 1 a m; se m*m for menor que x, a raiz existirá no intervalo de m a x.
- Etc.

Método da bisseção, recursivo

- Calcular recursivamente a raiz de x no intervalo de a a b:

```
double square_root_r(double x, double a, double b)
{
  assert(x > 1);
  assert(a < b);
  assert(a*a - x < 0),
  assert(b*b - x > 0);
  double result = (a + b) / 2;
  if (fabs(result * result - x) >= 0.000001)
  {
    if (result * result < x)
      result = square_root_r(x, result, b);
    else
      result = square_root_r(x, a, result);
  }
  return result;
}
```

Método da bisseção, iterativa

- A versão iterativa deriva-se diretamente da anterior:

```
double square_root_i(double x, double a, double b)
{
  assert(x > 1);
  assert(a < b);
  assert(a*a - x < 0),
  assert(b*b - x > 0);
  double result = (a + b) / 2;
  while (fabs(result * result - x) >= 0.000001)
  {
    if (result * result < x)
      a = result;
    else
      b = result;
    result = (a + b) / 2;
  }
  return result;
}
```

Função principal, raiz quadrada

- Para demonstração, a função principal invoca ou a versão iterativa ou a versão recursiva:

```
double square_root(double x)
{
  assert(x > 0);
  double result;
  if (x > 1)
    result = square_root_r(x, 1.0, x);
//    result = square_root_i(x, 1.0, x);
  else if (x < 1)
    result = 1 / square_root(1/x);
  else
    result = 1;
  return result;
}
```

A raiz de números menores que 1 calcula-se invertendo a raiz do inverso (o qual será maior que 1...)

Função de teste, raiz quadrada

```
void test_square_root(void)
{
  double x;
  while (scanf("%lf", &x) != EOF)
  {
    double z = square_root(x);
    printf("%.6f\n", z);
  }
}
```

```
$ ./a.out
6
2.449490
25
5.000000
0.01
0.100000
2
1.414214
0.5
0.707107
64
8.000000
1.44
1.200000
$
```

Note bem: o método da bisseção é interessante mas há outros métodos melhores para calcular a raiz quadrada e, mais geralmente, para resolver equações não lineares. Esses métodos convergem mais depressa para a solução, pois dividem o intervalo de busca mais "agressivamente".