FIAP ON

ALGORITMOS:
APRENDA A PROGRAMAR

BUSCA

JORGE L. SURIAN

8

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

8 BUSCA

A busca de valores num vetor é uma situação algorítmica bastante interessante. Vamos explorar algumas técnicas e variantes a elas associadas.

8.1 Pesquisa sequencial direta

Nesse tipo de pesquisa, iremos buscar dentro de um vetor determinado valor. Assim, considerando um array unidimensional (matriz de uma dimensão ou vetor) *A* contendo *n* valores nas posições *A*[0] .. *A*[*n*-1] e um valor *v*, descobrir:

- Se *v* pertence ao vetor.
- Qual posição de *A* contêm *v* (normalmente assume-se que todos os dados em *A* são distintos).

A busca sequencial é uma técnica que lembra o conhecido conceito de busca de "uma agulha num palheiro". Ora, dada a semelhança existente entre a agulha e cada ramo de palha, parece natural que devamos testar elemento a elemento, pois na pior das hipóteses, se eliminarmos toda palha teremos, finalmente, a **agulha!**

Vamos escrever uma função que de posse de um vetor "A", de um valor a ser procurado "v", num total de elementos "n" devolva ou a posição do elemento procurado ou o número "-1", indicativo da inexistência do valor.

```
algoritmo psq_seq_01;
início
fim
função buscasequencial (v : inteiro, n : inteiro, A : matriz[n-1] de inteiros) : inteiro
    achei : inteiro;
início
  achei := 0;
  i := 0;
  enquanto i < n e achei = 0 faça
    se A[i] = v então
       achei := 1;
    senão
       i := i + 1;
    fim-se
  fim-enquanto
  se achei então
     retorne i;
  senão
     retorne -1;
  fim-se
fim
```

Código-fonte 8.1 – Função I
Fonte: Elaborado pelo autor (2015)

Nossa estratégia consiste num índice *i* que varre a lista (nosso vetor) integralmente. A variável *achei* é uma “flag”, pois quando assume o valor um indica que o item foi encontrado. O valor zero indica que o valor não foi encontrado.

Naturalmente, se o item for encontrado ou se chegarmos ao final do vetor sem encontrá-lo, retornamos à posição do vetor em que o número procurado foi encontrado ou o valor negativo, indicativo da inexistência do registro.

8.2 Pesquisa sequencial com sentinela

Nesse tipo de pesquisa, usa-se uma posição adicional no final do vetor *A* que é carregada com uma cópia do dado que está sendo buscado. Dessa forma, garante-se que qualquer que seja o valor, esse sempre será encontrado!

Note que se esse valor estiver numa posição qualquer, essa será devolvida. Já se o valor não fizer parte do vetor, simplesmente a última posição é que constará na resposta.

```
algoritmo psq_seq_02;
início
fim
função buscasequencial (v : inteiro, n : inteiro, A : matriz[n] de inteiros) : inteiro
    achei : inteiro;
início
  A[n] := v;
  i := 0;
  enquanto A [i] <> v faça
    i := i + 1;
  fim-enquanto
  se i < n então
    retorne i;
  senão
    retorne -1;
  fim-se
fim
```

Código-fonte 8.2 – Função II
Fonte: Elaborado pelo autor (2015)

8.3 Aplicando análise do algoritmo

Vamos estudar brevemente a questão do desempenho do algoritmo de busca sequencial estudado.

A análise de pior caso de ambos os algoritmos para busca seqüencial são obviamente $O(n)$ (a lista toda...), embora a busca com sentinela seja mais rápida, apenas por realizar menos testes.

- A análise de caso médio requer que estipulemos um modelo probabilístico para as entradas. Sejam: E_0 , E_1 , … E_{n-1} as entradas v correspondentes às situações onde v=A[0], v=A[1], … v=A[n-1]
- E_n entradas v tais que v não pertence ao array A
- $p(E_i)$ a probabilidade da entrada E_i ocorrer
- t (E_i) a complexidade do algoritmo quando recebe a entrada E_i

Assumimos:

- $p(E_i) = q/n$ para $i < n$
- $p(E_n) = 1-q$
- Se admitirmos t (E_i) = $i+1$, então temos como complexidade média:

$$\sum_{i=0}^{n} p(E_i)\, t(E_i) = (n+1)(1-q) + \frac{q}{n}\left(\sum_{i=0}^{n-1} i + 1\right)$$
$$= (n+1)(1-q) + \frac{q}{n}\frac{n(n+1)}{2}$$
$$= \frac{(n+1)(2-q)}{2}$$

Assim se:

- q=1/2, temos complexidade média ≈ 3n/4
- q=0, temos complexidade média ≈ n
- q=1, temos complexidade média ≈ n/2

Todavia, nem seria necessário pensarmos tudo isso, pois nossos conhecimentos anteriores já nos levam a questionar um tipo de busca que realiza o processo, posição por posição. Ora, a árvore binária serviu anteriormente para maximizar o tempo em ordenações, talvez possa contribuir significativamente de alguma maneira nessa situação.

8.4 Pesquisa binária

Vamos supor um conjunto de “n” elementos armazenados em um vetor. Como notamos, na busca sequencial podemos percorrer o vetor inteiro e não encontrar a informação procurada.

Claramente, se o vetor for muito grande a busca sequencial pode demorar um tempo inaceitável (tempo de resposta “vira” prazo de entrega...), pois cada comparação da busca sequencial **elimina apenas um elemento do vetor**.

Mas, não seria possível melhorar esse processo aplicando o conceito de dividir o problema em um problema menor? Numa árvore binária balanceada, notamos que todos os elementos inferiores a algum termo estão sempre a *sua direita*. Cuidado para não confundir Árvore Binária Balanceada com Árvore Binária Hierárquica!

Por mais que as árvores binárias sejam um assunto complexo, tem várias aplicações, como vimos na questão da ordenação e veremos na busca. Calvin, tem também uma aplicação da árvore binária, bem a seu feitio.

Figura 8.1 – Exemplo de aplicação de árvore binária
Fonte: Google Images (2015)

Partindo da suposição de que um vetor esteja devidamente ordenado, podemos então aplicar o que chamamos de **busca binária**, que é baseada na ideia de se eliminar a maior quantidade de elementos do vetor com uma única comparação.

Chamando o elemento procurado de *p* e o vetor de *v*, podemos descrever o algoritmo de busca binária do seguinte modo:

- Pegue o elemento central do vetor v, seja *t* este elemento.
- se $t > p$, então pegue novamente o elemento central do vetor considerando todos elementos localizados antes da posição central.
- se $t < p$,*es* então pegue novamente o elemento central do vetor considerando todos elementos localizados depois da posição central.
- Este processo deve ser repetido até que não haja mais elementos do vetor a serem procurados.

Note que a cada comparação executada é eliminada, simplesmente, metade dos elementos do vetor considerado!

Assim, se o vetor tem tamanho *n* na primeira iteração, eliminamos *n/2* de seus elementos. Essa estratégia é também conhecida como “dividir para conquistar”, por razões bastante óbvias.

Devemos notar, por exemplo, que na segunda iteração teremos apenas *n/4* dos elementos originais, na terceira iteração *n/8* elementos e assim por diante. Esse

processo ocorrerá de forma que em cada uma das i-ésimas iterações do algoritmo $n/2^i$ elementos seja eliminada.

Nosso programa irá parar quando achar o elemento procurado ou quando não houver mais elemento a procurar!

Consideremos que na k-ésima comparação teremos apenas um elemento do vetor, ou seja, $n/2^k = 1$.

Ora, o número de comparações necessárias para o algoritmo de busca binária é $k = \log_2 n$.

Assim, apenas para termos uma pálida ideia do que ocorre, na busca simples, se tivermos um vetor com 1.000 elementos, poderemos fazer até **1.000** comparações no algoritmo.

Já na busca binária para um vetor **ordenado** com 1.000 elementos poderemos fazer no máximo **11** comparações!

E o número fica mais surpreendente quando dobramos a quantidade de elementos do vetor:

- busca simples: 2.000 comparações
- busca binária: 12 comparações

Ou seja, se os dados estiverem previamente ordenados (em ordem crescente ou decrescente), a busca pode ser feita de maneira muito mais eficiente. Como ponto negativo, devemos nos lembrar de que a ordenação demanda em tempo, também.

Todavia, se os valores não são alterados ou se essa alteração é pouco frequente, ou ainda quando a série é previamente ordenada, a busca binária passa a ser uma hipótese interessante a considerar, pois é muito mais eficiente que a busca sequencial!

Vamos agora analisar o algoritmo voltado à busca binária:

```
algoritmo psq_binaria;
início
fim
função buscabinaria (p : inteiro, n : inteiro, v : matriz[n] de inteiros) : inteiro
    inf, sup, meio : inteiro;
    achou : lógico;
início
  inf := 0;
  sup := n-1;
  achou := falso;
  enquanto (inf <= sup) e achou = falso faça
     meio := (inf+sup) / 2;
     se v[meio] < p então
        inf := meio + 1;
     senão
        se v[meio] > p então
           sup := meio - 1;
        senão
           achou := verdadeiro;
        fim-se
     fim-se
  fim-enquanto
  se achou então
     retorne meio;
  senão
     retorne -1;
  fim-se
fim
```

Código-fonte 8.3 – Pesquisa binária
Fonte: FIAP (2005)

Primeiramente, é importante observar que as variáveis *inf, sup* e *meio*, trabalham em conjunto visando permitir que a cada comparação, metade dos elementos seja descartada.

Por exemplo, vamos pensar na estrutura representada no desenho a seguir, supondo que estamos buscando o valor 40, que levaria a seguinte “trajeto”:

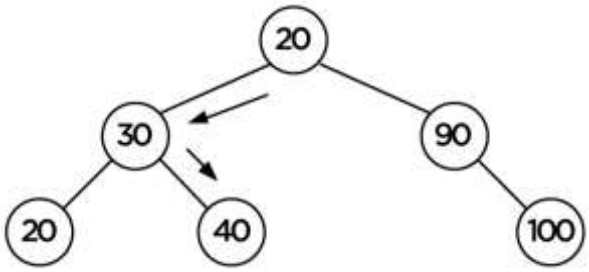

Figura 8.2 – Buscando o valor 40
Fonte: FIAP (2015)

Se admitirmos que as posições do vetor estejam numeradas sempre da esquerda para a direita, ou seja, v[0] vale 20, v[1] vale 30, e assim respectivamente

até v[5] que vale 100, teríamos para nossas variáveis *inf, sup* e *meio* os seguintes valores, para as pesquisas dos números 40 e 30:

p = 40			
Inf	Sup	Meio	Situação
0	5	2	Achou

p = 30			
Inf	Sup	Meio	Situação
0	5	2	
0	1	0	Achou
1	1	1	Achou

V	Inicial
0	20
1	30
2	40
3	50
4	90
5	100

Figura 8.3 – Buscando valores 30 e 40
Fonte: FIAP (2015)

Ou seja, “quebramos” o problema em partes menores e, a partir delas, voltamos a fazer nova pesquisa.

8.4.1 Pesquisa binária – simulando busca de elemento existente

Vamos, agora, assumir a seguinte lista ordenada para pesquisarmos:

v[0] = 10; v[1] = 11; v[2] = 12; v[3] = 13; v[4] = 17; v[5] = 18 e v[6] = 19.

Vamos “buscar” os números 13; 10 (estão na lista) e 15 (que não está), para entendermos o processo.

Inicialmente, vamos detalhar situações em que a pesquisa é bem-sucedida.

V	Inicial
0	10
1	11
2	12
3	13
4	17
5	18
6	19

Figura 8.4 – Busca de elemento existente
Fonte: FIAP (2015)

p = 13						
Inf	Sup	Meio	Achou	N	v[Meio]	Ocorre
0	6	3	FALSO	6	13	v[Meio] =13
			VERDADEIRO			
p = 10						
Inf	Sup	Meio	Achou	N	v[Meio]	Ocorre
0	6	3	FALSO	6	13	v[Meio] > 10
	2	1			11	v[Meio] > 10
	1	0	VERDADEIRO		10	v[Meio] > 10

Figura 8.5 – Busca de elemento existente (2)
Fonte: FIAP (2015)

Observar que meio SEMPRE recebe um valor inteiro resultado da divisão de INF e SUP, ou seja, “arredonda” para baixo!

Nas situações retratadas, temos a variável *achou* sinalizando o que fazer. Note que nos dois casos, ela “quebra” o loop.

Mas, o que ocorre se buscarmos na lista um valor inexistente?

V	Inicial
0	10
1	11
2	12
3	13
4	17
5	18
6	19

Figura 8.6 – Busca de elemento inexistente
Fonte: FIAP (2015)

p = 15							
Inf	Sup	Meio	Achou	N	v[Meio]	Ocorre	inf <= sup
0	6	3	FALSO	6	13	v[Meio] < 15	Sim
4	6	5	FALSO		18	v[Meio] > 15	Sim
4	4	4	FALSO		17	v[Meio] > 15	Sim
4	3						Não

Figura 8.7 – Busca de elemento inexistente (2)
Fonte: FIAP (2015)

Observar que quando o valor é inexistente, o algoritmo caminha para um paradoxo em que SUP (nosso superior) se torna menor que INF (nosso inferior). Ora, nessas situações a conclusão é que o elemento procurado não está no vetor.

8.5 Exercícios

a) Escreva uma função de busca simples. Sua função recebe um vetor v de caracteres e uma letra qualquer. Sua função deverá retornar à posição da letra no vetor ou -1 se a letra não estiver no vetor.

```
Algoritmo "Posição a"

//
// Escreva uma função de busca simples.
// Sua função recebe um vetor v de
// caracteres e uma letra qualquer.
// Sua função deverá retornar à posição
// da letra no vetor ou -1 se a letra
// não estiver no vetor.
//

Var
```

```
  i: inteiro;
  n: inteiro;
  posicao: inteiro;
  texto: caractere;
  letra: caractere;
  v: vetor[1..100] de caractere;

Inicio

  ESCREVA ("Digite o texto onde deseja procurar a letra: ")
  LEIA (texto)
  n := COMPR(texto)
  ESCREVA ("Digite a letra que deseja procurar no texto: ")
  LEIA (letra)
  posicao := -1

  PARA i DE 1 ATE n FACA
    v[i] := COPIA(texto,i,1)
  FIMPARA

  PARA i DE 1 ATE n FACA
    SE (v[i] = letra) E (posicao = -1) ENTAO
      posicao := i
    FIMSE
  FIMPARA
```

```
   ESCREVA ("A letra está na posição ", posicao)

Fimalgoritmo
```

b) No exercício anterior é retornada uma posição em v, mas isso não garante que possa haver mais de uma ocorrência da letra em v. Crie uma outra função que retorne todas as posições de v em que a letra em questão existir.

```
Algoritmo "Posição b"

//
// No exercício anterior é retornada
// uma posição em v, mas isso não garante
// que possa haver mais de uma ocorrência
// da letra em v. Crie uma outra função
// que retorne todas as posições de v
// em que a letra em questão existir.
//

Var

   i: inteiro;
   n: inteiro;
   posicao: inteiro;
   texto: caractere;
   letra: caractere;
```

```
  v: vetor[1..100] de caractere;

Inicio

  ESCREVA ("Digite o texto onde deseja procurar a letra: ")
  LEIA (texto)
  n := COMPR(texto)
  ESCREVA ("Digite a letra que deseja procurar no texto: ")
  LEIA (letra)
  posicao := -1

  PARA i DE 1 ATE n FACA
    v[i] := COPIA(texto,i,1)
  FIMPARA

  PARA i DE 1 ATE n FACA
    SE (v[i] = letra) ENTAO
      ESCREVAL ("A letra está na posição ", i)
      posicao := i
    FIMSE
  FIMPARA

  SE posicao = -1 ENTAO
    ESCREVA ("A letra ", letra, " não existe nessa frase ")
  FIMSE
```

Fimalgoritmo

c) Implemente o algoritmo de busca binária. Suponha que os dados do vetor v estejam ordenados. Em seguida, devolva todas as posições em que a letra aparecer.

Algoritmo "Posição c"

//

// No exercício anterior é retornada

// uma posição em v, mas isso não garante

// que possa haver mais de uma ocorrência

// da letra em v. Crie uma outra função

// que retorne todas as posições de v

// em que a letra em questão existir.

//

Var

i: inteiro;

j: inteiro;

m: inteiro;

n: inteiro;

o: inteiro;

achei: logico;

texto: caractere;

letra: caractere;

c: caractere;

```
  v: vetor[1..100] de caractere;

Inicio

  ESCREVA ("Digite o texto onde deseja procurar a letra: ")
  LEIA (texto)
  ESCREVA ("Digite a letra que deseja procurar no texto: ")
  LEIA (letra)
  n := COMPR(texto)
  achei := falso

  PARA i DE 1 ATE n FACA
    v[i] := COPIA(texto,i,1)
  FIMPARA

  PARA i DE 1 ATE n - 1 FACA
    PARA j DE i + 1 ATE n FACA
      SE v[i] > v[j] ENTAO
        c := v[i]
        v[i] := v[j]
        v[j] := c
      FIMSE
    FIMPARA
  FIMPARA

  m := 1
```

```
    ENQUANTO (m <= n) E (nao achei) FACA
      o := INT((m + n) / 2)
      SE v[o] = letra ENTAO
        achei := verdadeiro
      FIMSE
      SE v[o] > letra ENTAO
        n := o - 1
      SENAO
        m := o + 1
      FIMSE
    FIMENQUANTO

    SE achei ENTAO
      ESCREVA ("A letra ", letra, " está na posição ", o)
    SENAO
      ESCREVA ("Não existe letra ", letra, " neste texto)
    FIMSE

Fimalgoritmo
```

REFERÊNCIAS

SCRIPTOL. Disponível em: <http://www.scriptol.com>. Acesso em: 18 jun. 2020.

FEOFILOFF, Paulo. **Algoritmos em Linguagem C.** Rio de Janeiro: Campus, 2009.

FORBELLONE, André L.V.; EBERSPACHER, Henri F. **Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, Sérgio. **Java 2, Ensino Didático***.* São Paulo: Érica, 2002.

GANE, Chris; SARSON, Trish. **Análise Estruturada de Sistemas***.* São Paulo: LTC-Livros Técnicos e Científicos, 1983.

GONDO, Eduardo. **Apostila: Notas de Aula***.* São Paulo, 2008.

LATORE, Robert. **Aprenda em 24 horas Estrutura de Dados e Algoritmos***.* Rio de Janeiro: Campus, 1999.

MANZANO, José A.N.G.; OLIVEIRA, Jayr F. **Algoritmos:** Lógica para o Desenvolvimento de Programação. 23.ed. São Paulo: Érica, 2010.

PIVA JUNIOR, Dilermando et al. **Algoritmos e Programação de Computadores.** Rio de Janeiro: Campus, 2012.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de Programação e Estrutura de Dados**. São Paulo: Pearson Prentice Hall, 2009.

ROCHA, Antonio Adrego. **Estrutura de Dados e Algoritmos em Java***.* Lisboa: FCA- Editora de Informática, 2011.

RODRIGUES, Rita. **Apostila: Notas de Aula***.* 2008.

SALVETTI, Dirceu Douglas; BARBOSA, Lisbete Madsen. **Algoritmos***.* São Paulo: Makron Books, 1998.

SCHILDT, Herbert. **Linguagem C - Guia Prático***.* São Paulo: McGraw Hill, 1989.

WOOD, Steve. **Turbo Pascal – Guia do Usuário***.* São Paulo: McGraw Hill, 1987.

ZIVIANI, Nivio. **Projeto de Algoritmos com implementações em Pascal e C***.* São Paulo: Pioneira, 1999.