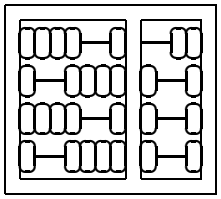

MC-102 Algoritmos e Programação de Computadores
IC-UNICAMP

Aula 17 - Aula sobre métodos de busca e ordenação

1 Objetivos

Apresentar os diferentes métodos de busca e ordenação, destacando suas diferenças e as vantagens desvantagens de cada um.

2 Motivação

- Permitir que a busca por um elemento seja feita com eficiência, através da escolha de um método adequado.
- Mostrar os métodos de ordenação, seus pontos fracos e também onde cada método é mais adequado, permitindo assim que a escolha do método possa atender a aplicação.

3 Métodos de Busca

Problema

Dada uma coleção de n elementos, pretende-se saber se um determinado elemento x existe nessa coleção. Para efeitos práticos, vamos supor que essa coleção é implementada como sendo um vetor a[0...n-1] de n elementos inteiros.

3.1 Solução 1 - Pesquisa Seqüencial

Uma solução possível é percorrer o vetor desde a primeira posição até à última. Para cada posição i, comparamos a[i] com x. Se forem iguais dizemos que x existe. Se chegarmos ao fim do vetor dizemos que x não existe. Vamos traduzir estas palavras para um algoritmo.

Algoritmo 1: pesquisa seqüencial

```
(1) i <-- 0
(2) Se i = n, escreve "não existe" e termina.
(3) Se x = a[i], escreve "existe" e termina.
(4) i <-- i+1. Volta ao passo 2.
```

Este algoritmo chama-se algoritmo de pesquisa seqüencial porque varre os elementos da nossa coleção seqüencialmente um após outro. O algoritmo é independente da linguagem de programação em que possa vir a ser implementado. Se quisermos implementar este algoritmo em C teríamos de escrever qualquer coisa deste estilo:

```
i = 0;
do
 {
    if( i == n ) { printf("não existe"); break; }
    if( x == a[i] ) { printf("existe"); break; }
    i++;
 }
while( 1 );
```

Pergunta

Quanto tempo é que este algoritmo demora a executar?

Por outras palavras, quantas vezes é que a comparação x == a[i] é executada?

Resposta

```
* caso x não exista no vetor, n vezes.
* caso x exista no vetor,
      o 1 vez no melhor caso (x está na primeira posição).
      o n vezes no pior caso (x está na última posição).
      o n/2 vezes no caso médio.
```

E se o vetor estivesse ordenado?

3.2 Solução 2 - Pesquisa Binária

Vamos supor agora que o vetor inicial estava ordenado por ordem crescente (se fosse por ordem decrescente o raciocínio era semelhante). Será que é possível resolver o problema de modo mais eficiente?

A resposta é positiva. O problema que estamos a ver é semelhante ao problema de encontrar o telefone de uma pessoa numa lista telefônica. Se a lista não estivesse ordenada, não teríamos outra hipótese senão percorrer a lista do principio até ao fim (uma tarefa em tudo semelhante a encontrar uma agulha num palheiro). Esse algoritmo seria o equivalente à pesquisa seqüencial que vimos à pouco.

Conhecem alguém que procura a lista telefônica do princípio até ao fim? Claro que não. Como a lista está ordenada alfabeticamente, conseguimos descobrir rapidamente o telefone de uma pessoa. Abrimos a lista no meio e depois decidimos pelo lado esquerdo ou pelo lado direito. E assim sucessivamente até encontrarmos o nome da pessoa. A cada passo vamos eliminando grande parte das páginas da lista.

Se o vetor estiver ordenado podemos implementar um algoritmo semelhante ao da lista telefônica. A idéia é manter duas variáveis, esq e dta, que significam os limites esquerdo e direito do vetor. A

idéia é que se o elemento x existe numa posição a[i] do vetor, então i tem de estar compreendido entre esq e dta ($esq \leq i \leq dta$).

Algoritmo 2: pesquisa binária

```
    (1) esq <-- 0 , dta <-- n-1
    (2) Se esq > dta, escreve "não existe" e termina.
    (3) i <-- parte inteira de (esq+dta)/2
    (4) Se x = a[i], escreve "existe" e termina.
        Se x < a[i], dta <-- i-1. Volta ao passo 2.
        Se x > a[i], esq <-- i+1. Volta ao passo 2.
{
```

Vamos programar o algoritmo de pesquisa binária em C.

```
esq = 0;
dta = n-1;
do
 {
     if( esq > dta )
     {
        printf("nao existe");
        break;
     }
    i = (esq+dta)/2;
    if( x == a[i] )
     {
        printf("existe");
        break;
     }
    else if( x < a[i] )
             dta = i-1;
         else
             esq = i+1;
 }
while( 1 );
```

Pergunta

Quanto tempo é que o algoritmo de pesquisa binária demora?

Por outras palavras, quantas vezes é que a comparação x == a[i] é executada?

Resposta

```
 * no melhor caso, 1 vez.
 * no pior caso, log2(n).

   (nota: log2 significa logaritmo de base 2)
```

Vejamos um exemplo. Se n=1000, o algoritmo de pesquisa seqüencial irá executar 1000 comparações no pior caso. Por sua vez, o algoritmo de pesquisa binária irá executar 10 comparações no pior caso. O logaritmo de base 2 aparece porque estamos sempre a dividir o intervalo ao meio: 1000, 500, 250, 125, 63, 32, 16, 8, 4, 2, 1. E se n=1000000000 (1 bilhão) ? Aqui vai a resposta:

```
* Pesquisa binária faz 30 comparações.
* Pesquisa seqüencial faz 1000000000 comparações!
```

Qual dos dois algoritmos é melhor? Não se pode dizer que um é melhor do que o outro. Depende ... Há vários pontos que se deve ter em consideração.

1. Se n é pequeno (inferior a 100 ou coisa parecida) não vale a pena estar a fazer pesquisas binárias porque o computador é muito rápido a varrer uma coleção de 100 elementos.
2. O algoritmo de pesquisa binária assume que o vetor está ordenado. Ordenar um vetor também tem um custo (quantas comparações são necessárias para ordenar um vetor de dimensão n utilizando o algoritmo da aula passada?)
3. Se for para fazer uma só pesquisa, não vale a pena estar a ordenar o vetor. Por outro lado, se pretendermos fazer muitos pesquisas, o esforço da ordenação já poderá valer a pena.

4 Métodos de Ordenação

O problema da ordenação é fundamental na computação, como vimos nos métodos de pesquisa a eficiência da busca sempre é melhor quando trabalhamos com conjuntos ordenados.

Existem muitos algoritmos de ordenação, a escolha do mais eficiente vai depender de vários fatores: número de itens a ser classificado; se os valores já estão agrupados em subconjuntos ordenados; etc.

4.1 Ordenação por Inserção

A proposta da ordenação por inserção é:

```
Para cada elemento do vetor faça
    insira-o na sua posição correspondente;
```

Durante o processo de ordenação por inserção a lista fica dividida em duas sub listas, uma com os elementos já ordenados e a outra com elementos ainda por ordenar.

No início, a sub lista ordenada é formada trivialmente apenas pelo primeiro elemento da lista.

Nos métodos de ordenação por inserção, a cada etapa i, o i-ésimo elemento é inserido em seu lugar apropriado entre os(i-1) elementos já ordenados. Os índices dos itens a serem inseridos variam 2 a m.

Veja no exemplo abaixo o resultado das varreduras:

Varredura	X[0]	X[1]	X[2]	X[3]	X[4]
Vetor original	9	8	7	6	5
1	{8	9}	{7	6	5}
2	{7	8	9}	{6	5}
3	{6	7	8	9}	{5}
4	{5	6	7	8	9}

4.1.1 Função de Inserção

```
void Insercao(int m,int vet[])
{
   int i,j,aux;
   for(i = 1; i < m; i++)
    {
      aux = vet[i];
      for(j = i-1; (j >= 0) && (aux < vet[j]); j--)
          vet[j + 1] = vet[j];
          vet[j + 1] = aux;
    }
}
```

Número de Comparações:

Em cada etapa i são executadas no máximo, i comparações pois o loop interno é executado para j de i-1 até 0. Como i varia de 1 ate m temos, pela formula da soma dos termos de uma P.A.:

$\sum_{i=1}^{m} a_i = (1+m) * \frac{(m+1)}{2} = O(m^2)$

4.2 Ordenação por Seleção

A ordenação por seleção consiste, em cada etapa, em selecionar o maior (ou o menor)elemento e alocá-lo em sua posição correta dentro da futura lista ordenada.

Durante a aplicação do método de seleção a lista com m registros fica decomposta em duas sub listas, uma contendo os itens já ordenados e a outra com os restantes ainda não ordenados. No início a sub lista ordenada é vazia e a outra contém todos os demais.No final do processo a sub lista ordenada apresentará (m-1) itens e a outra apenas 1.

As etapas(ou varreduras) como já descrito acima consiste em buscar o maior elemento da lista não ordenada e colocá-lo na lista ordenada. Veja no exemplo abaixo o resultado das etapas da ordenação de um vetor de inteiros:

Etapa	X[0]	X[1]	X[2]	X[3]	X[4]
Vetor original	5	9	1	4	3
1	{5	3	1	4}	{9}
2	{4	3	1}	{5	9}
3	{1	3}	{ 4	5	9}
4	{1}	{3	4	5	9}

4.2.1 Função de Seleção(Buscando o maior elemento)

```
void Selecao(int m,int x[])
{
  int aux,j,i,maior;
  for(i=0;i<m-1;i++)
  {
    maior=0;
    for(j=0;j<m-i;j++)
      if (x[j] > x[maior])
          maior=j;

   aux=x[m-i-1];
   x[m-i-1]=x[maior];
   x[maior]=aux;
  }
}
```

Número de Comparações:

A comparação é feita no *loop* mais interno for(j=0;i¡m-i;j++) .Para cada valor de i são feitas (m-i) comparações dentro do loop mais interno. Como i varia de até m-1, o numero total de comparações para ordenar a lista toda é:

$\sum_{i=0}^{m-1}(m-1) = m + (m-1) + (m-2) + ... + 1 = \frac{m(m-1)}{2} = O(m^2)$

4.3 Ordenação por troca - *Bubble Sort*

Um método simples de ordenação por troca é a estratégia conhecida como bolha que consiste, em cada etapa "borbulhar" o maior elemento para o fim da lista. Inicialmente percorre-se a lista dada da esquerda para a direita, comparando pares de elementos consecutivos, trocando de lugar os que estão fora da ordem.No exemplo abaixo, em cada troca, o maior elemento é deslocado uma posição para a direita.

Varredura	X[0]	X[1]	X[2]	X[3]	Troca
1	10	9	7	6	0 e 1
	9	10	7	6	1 e 2
	9	7	10	6	2 e 3
	9	7	6	10	Fim da Varredura 1

Após a primeira varredura o maior elemento encontra-se alocado em sua posição definitiva na lista ordenada. Podemos deixá-lo de lado e efetuar a segunda varredura na sub lista v[0],v[1],v[2]. Veja a continuação do exemplo:

Varredura	X[0]	X[1]	X[2]	X[3]	Troca
2	9	7	6	10	0 e 1
	7	9	6	10	1 e 2
	7	6	9	10	Fim da Varredura 2

Após a segunda varredura o maior elemento da sub lista v[0],v[1],v[2] encontra-se alocado em sua posição definitiva. A próxima sub lista a ser ordenada é v[0],v[1]. Veja a continuação do exemplo:

Varredura	X[0]	X[1]	X[2]	X[3]	Troca
3	7	6	9	10	0 e 1
	6	7	6	10	Fim da Varredura 3

4.3.1 Função Bubble Sort

```
void BubbleSort(int m,int x[])
{
  int aux,j,i;
  for(i=0;i<m-1;i++)
  {
    for(j=0;j<m-i-1;j++)
      if (x[j] > x[j+1]) {
        aux=x[j];
        x[j]=x[j+1];
        x[j+1]=aux;
      }
  }
}
```

Número de Comparações:

No algoritmo da bolha observamos que existem m-1 varreduras(etapas) e que em cada varredura o número de de comparações diminui de uma unidade , variando de m-1 até 1. Portanto :

$\sum_{i=1}^{m-1}(m-1) = (m-1) + (m-2) + ... + 1 = \frac{m(m-1)}{2} = O(m^2)$

4.4 Exercícios

1. Dado o vetor ordenado v=1,5,6,7,12,17,22,45,65,98,100,120,150,200,231. Mostre passo a passo a pesquisa binária pelo elemento 100. Quantas comparações foram feitas? E se a pesquisa fosse seqüencial, quantas comparações seriam feitas?

   R.

   ```
   n=15
   inicio=0  fim=14 meio=7   Elem[7] = 45  < 100
   inicio=8  fim=14 meio=11  Elem[11]= 120 > 100
   inicio=8  fim=10 meio=9   Elem[9] = 98  < 100
   inicio=10 fim=10 meio=10  Elem[10]=100  - Elemento Encontrado.

   P.B. realizou 4 comparações. Uma pesquisa seqüencial faria 11 comparações.
   ```

2. Para o vetor acima, de um exemplo onde a pesquisa seqüencial faria menos comparações que a pesquisa binária? **R.**

```
Busca pelo elemento 1.

n=15
inicio=0  fim=14 meio=7  Elem[7] = 45  > 1
inicio=0  fim=6  meio=3  Elem[3] = 7   > 1
inicio=0  fim=2  meio=1  Elem[1] = 5   > 1
inicio=0  fim=0  meio=0  Elem[0] = 1   - Elemento Encontrado.

PB faz 4 comparações e a pesquisa seqüencial faz apenas uma comparação.
```

3. Ordene o vetor v=20,12,28,05,10,18 usando o método de inserção. Mostre o vetor a cada passo do *loop*.

   R.

	V[0]	**V[1]**	**V[2]**	**V[3]**	**V[4]**	**V[5]**	i
V	20	12	28	05	10	18	0
	20						1
	12	20					2
	12	20	28				3
	05	12	20	28			4
	05	10	12	20	28		5
	05	10	12	18	20	28	6

4. Ordene o vetor v=20,12,28,05,10,18 usando o método de seleção. Mostre o vetor a cada passo do *loop*.

   R.

	V[0]	**V[1]**	**V[2]**	**V[3]**	**V[4]**	**V[5]**	i
V	20	12	28	05	10	18	0
	20	12	05	10	18	28	1
	12	05	10	18	20	28	2
	12	05	10	18	20	28	3
	05	10	12	18	20	28	4
	05	10	12	18	20	28	5

5. Ordene o vetor v=20,12,28,05,10,18 usando o método de bolha. Mostre o vetor a cada passo do *loop*.

   R.

	V[0]	**V[1]**	**V[2]**	**V[3]**	**V[4]**	**V[5]**	i
V	20	12	28	05	10	18	
1 Passo	12	20	05	28			
				10	28		
					28	28	
2 Passo	12	05	10	18	20	28	
3 Passo	05	10	12	18	20	28	
4 Passo							

5 Bibliografia Utilizada

Apostila do **prof. Flávio Keidi Miyazawa**.

Paulo Feofiloff - Projeto de Algoritmos. disponível em: http://www.ime.usp.br/ pf/algoritmos

Fernando Lobo. Programação Imperativa, 2003/04. Disponível em :http://www.adeec.fct.ualg.pt/PI_flobo/

Material disponível em: http://www.inf.pucpcaldas.br/biblioteca_virtual/c/metodos_de_ordenacao.htm

Material do Curso de C da UFMG - Disponível em:http://ead1.eee.ufmg.br/cursos/C/