UTFPR - Estrutura de Dados I

Profª Renata Luiza Stange Carneiro Gomes

Atividade Formativa 05: Métodos Ordenação

1 INSTRUÇÕES

Os exercícios devem ser implementados utilizando a linguagem de programação Java. Busque soluções eficientes que eliminem passos desnecessários. Tenha especial enfoque nas dúvidas e dificuldades que se apresentarem. Anote-as e sane-as em aula. Caso sejam dúvidas menos superficiais, utilize do horário de atendimento. Tratam-se de conceitos importantes para a sequência da disciplina.

2 MÉTODOS DE ORDENAÇÃO

1. Seja $v[1..n]$ um vetor cujos n itens aparecem numa ordem aleatória. A ordenação crescente consiste em se determinar uma permutação dos itens em v tal que tenhamos $v[1] \leq v[2] \leq v[3] \leq ... \leq v[n]$.

   1º PASSO: Implemente em uma classe `Teste`, um algoritmo que gere n números inteiros e armazene em um vetor v. Esta classe deverá ser utilizada para testar os demais algoritmos da avaliação.

2.1 ORDENAÇÃO POR TROCAS

A ordenação por trocas consiste em comparar pares itens consecutivos e permutá-los, caso estejam fora de ordem. Se o vetor for assim processado, sistematicamente, da esquerda para a direita, um item máximo será deslocado para sua última posição. Em cada fase desse método, um item de maior valor é deslocado para sua posição definitiva no vetor ordenado. Então, se um vetor tem n itens, após a primeira fase haverá $n - 1$ itens a ordenar. Usando a mesma estratégia, após a segunda fase haverá $n - 2$ itens a ordenar, depois $n - 3$ e assim sucessivamente até que reste um único item.

O algoritmo **Bubble-Sort**, apresentado em aula [1], implementa uma versão **iterativa** da estratégia de ordenação por trocas.

[1] Disponível em: `https://moodle.utfpr.edu.br/pluginfile.php/1388682mod_resource/content/1/Sort.java`

```java
public int[] bubbleSort(int vetor[]){
        int i, j,aux;

        for(i=1; i < vetor.length; i++)
        {
            for(j=1; j<=(vetor.length)-i; j++)
            {
                if(vetor[j-1]>vetor[j])
                {
                    aux = vetor[j];
                    vetor[j] = vetor[j-1];
                    vetor[j-1] = aux;
                }
            }
        }
            return vetor;
    }
}
```

Exemplo 1: Veja agora a ordenação completa de um vetor pelo método da bolha. A variável i indica a fase da ordenação e j indica a posição do par de itens consecutivos que serão comparados e, eventualmente, permutados.

fase	*i*	*j*	*v*[1]	*v*[2]	*v*[3]	*v*[4]	*v*[5]
1º	1	1	**46**	**39**	55	14	27
		2	39	**46**	**55**	14	27
		3	39	46	**55**	**14**	27
		4	39	46	14	**55**	**27**
			39	46	14	27	**55**
2º	2	1	**39**	**46**	14	27	**55**
		2	39	**46**	**14**	27	**55**
		3	39	14	**46**	**27**	**55**
			39	14	27	**46**	**55**
3º	3	1	**39**	**14**	27	**46**	**55**
		2	14	**39**	**27**	**46**	**55**
			14	27	**39**	**46**	**55**
4º	4	1	**14**	**27**	**39**	**46**	**55**
			14	**27**	**39**	**46**	**55**

Figure 1: Simulação do algoritmo de ordenação por trocas

2.1.1 Com base no método de ordenação por troca siga as seguintes instruções:

1. Modifique o método `bubbleSort(int vetor[])` de modo que o vetor seja ordenado de forma **decrescente**.
2. Simule a execução do algoritmo para um vetor de $n \geq 10$ elementos e apresente o resultado em uma tabela conforme a Figura 1;
3. Quantas **comparações** e quantas **trocas** o algoritmo fez?

2.2 ORDENAÇÃO POR SELEÇÃO E BUSCA SEQUENCIAL

A estratégia básica desse método é, em cada fase, selecionar um menor item ainda não ordenado e permutá-lo com aquele que ocupa a sua posição na sequência ordenada. Mais precisamente, isso pode ser descrito assim:

- Para ordenar uma sequência $< a_i, a_{i+1}, ..., a_n >$, selecione uma valor k tal que a $k = min\{a_i, a_{i+1}, ..., a_n\}$, permute os elementos a_i e a_k e, se $i + 1 < n$, repita o procedimento para ordenar a subsequencia $< a_{i+1}, ..., a_n >$.

O algoritmo **Select-Sort**, apresentado em aula, implementa uma versão da estratégia de ordenação por seleção.

```
1   public int[] selectionSort(int[] vetor) {
2       int i, j, eleito, menor, posicao=0;
3
4       for(i=0; i<=(vetor.length)-2; i++)
5       {
6           eleito = vetor[i];
7           menor = vetor[i+1];
8           posicao = i + 1;
9
10          for(j=i+2; j<=(vetor.length)-1;j++)
11          {
12              if(vetor[j] < menor)
13              {
14                  menor = vetor[j];
15                  posicao = j;
16              }
17          }
18          if(menor < eleito){
19              vetor[i] = vetor[posicao];
20              vetor[posicao] = eleito;
21          }
22      }
23      return vetor;
24  }
```

Fase	*i*	*k*	a_1	a_2	a_3	a_4	a_5	a_6
1º	1	4	**46**	55	59	**14**	38	27
2º	2	6	**14**	**55**	59	46	38	**27**
3º	3	5	**14**	**27**	**59**	46	**38**	55
4º	4	4	**14**	**27**	**38**	**46**	59	55
5º	5	6	**14**	**27**	**38**	**46**	**59**	**55**
			14	**27**	**38**	**46**	**55**	59

Figure 2: Simulação do algoritmo de ordenação por seleção

Exemplo 2: Veja a ordenação da sequência $< 46, 55, 59, 14, 38, 27 >$ pelo método por seleção. A variável i indica a fase da ordenação e k indica a posição do menor valor.

Exemplo 3: Veja a busca pelo valor mínimo para a primeira iteração de i. A variável i indica a fase da ordenação e k indica a posição parcial do menor valor. Note que na primeira iteração o valor retornado pelo método é $k = 4$, isto é, o elemento de menor valor encontrado é $v[4] = 14$;

k	*j*	v_1	v_2	v_3	v_4	v_5	v_6
1	1	**46**	**55**	59	14	38	27
1	2	**46**	55	**59**	14	38	27
1	3	**46**	55	59	**14**	38	27
4	4	46	55	59	**14**	**38**	27
4	5	46	55	59	**14**	38	**27**
4	–	46	55	59	**14**	38	27

Figure 3: Simulação do método de busca do menor valor

2.2.1 Com base nos exemplos 2 e 3 siga as seguintes instruções:

1. Separe a lógica do algoritmo em dois métodos distintos: Um método principal `selectionSort(int[] vetor)` que retorna o vetor ordenado, tal como o apresentado em aula. Entretanto, este método executa uma chamada de um método `getMin(v,i,n)` que retorna a posição do item mínimo dentro da sequência $< v_i, v_{i+1}, ..., v_n >$. Observe que este método implementa uma **busca sequêncial**. Inicialmente, o item v_1 é assumido como o mínimo ($k = 1$). Então v_1 é comparado aos demais itens da sequência até que v_4 é encontrado. Como $v_4 < v_1$, o item v_4 passa a ser o mínimo ($k = 4$) e o processo continua analogamente

2. Simule a execução do algoritmo para um vetor de $n \geq 10$ elementos e apresente o resultado em uma tabela conforme a Figura 2 e 3;

3. Quantas **comparações** e quantas **trocas** o algoritmo fez?

2.3 ORDENAÇÃO POR INSERÇÃO

Ao ordenar a sequência $V =< a_1, a_2, ..., a_n >$, esse método considera uma subsequência ordenada $< V_0 =< a_1, ..., a_{i-1} >$ e outra desordenada $V_d =< a_i, ..., a_n >$. Então, em cada fase, um item é removido de V_d e inserido em sua posição correta dentro de V_o. À medida em que o processo se desenvolve, a subsequência desordenada vai diminuindo, enquanto a ordenada vai aumentando.

O algoritmo **Insert-Sort**, apresentado em aula, implementa uma versão da estratégia de ordenação por inserção.

```
1         public int[] insertionSort(int[] vetor) {
2         int i, j, eleito;
3
4         for (i=1; i<=(vetor.length)-1;i++)
5         {
6           eleito = vetor[i];
7           j=i-1;
8
9          while((j>=0) && (vetor[j]>eleito))
10         {
11              vetor[j+1] = vetor[j];
12             j = j-1;
13        }
14         vetor[j+1] = eleito;
15        }
16
17        return vetor;
18    }
```

Exemplo 4. Veja a ordenação da sequência $< 34, 17, 68, 29, 50, 47 >$ pelo método por inserção. A variável i indica a fase da ordenação; a aparte cinza indica o vetor ordenado V_o e a parte branca o vator desordenado V_d.

i	v_1	v_2	v_3	v_4	v_5	v_6
1	**34**	**17**	68	29	50	47
2	**17**	**34**	**68**	29	50	47
3	**17**	**34**	**68**	**29**	50	47
4	**17**	**29**	**34**	**68**	**50**	47
5	**17**	**29**	**34**	**50**	**68**	**47**
	17	**29**	**34**	**47**	**50**	**68**

Figure 4: Simulação do algoritmo de ordenação por inserção

Inicialmente, a parte ordenada [2] é $< 34 >$ e a desordenada é $< 17, 68, 29, 50, 47 >$. Então, o item 17 é removido da parte desordenada e a posição liberada fica disponível no final da parte ordenada. Em seguida, o item 17 é comparado ao 34 que, sendo maior, é deslocado para a direita. Como não há mais itens, o 17 é inserido na posição liberada pelo item 34 . Ao final dessa fase, a parte ordenada tem um item a mais e a desordenada, um item a menos.

2.3.1 Com base no exemplos 4 siga as seguintes instruções:

1. Modifique o método `insertionSort(int[] vetor)` para ordenar um vetor de strings. **DICA:** Você pode utilizar o método `compareTo` da classe String para comparar os elemntos do vetor, entretanto não é obrigatório utilizá-lo na solução.

2. Simule a execução do algoritmo para um vetor de $n \geq 5$ elementos e apresente o resultado em uma tabela conforme a Figura 4;

2.4 BUSCA BINÁRIA

A busca é o processo em que se determina se um particular elemento x é membro de uma determinado vetor V. Dizemos que a busca tem sucesso se $x \in V$ e que fracassa em caso contrário. Ao contrário da busca sequencial, a busca binária somente funciona corretamente se o vetor estiver ordenado. Esse método é semelhante àquele que usamos quando procuramos uma palavra num dicionário: primeiro abrimos o dicionário numa página aproximadamente no meio; se tivermos sorte de encontrar a palavra nessa página, ótimo; senão, verificamos se a palavra procurada ocorre antes ou depois da página em que abrimos e então continuamos, mais ou menos do mesmo jeito, procurando a palavra na primeira ou na segunda metade do dicionário... Como a cada comparação realizada o espaço de busca reduz-se aproximadamente à metade, esse método é denominado busca binária.

Exemplo 5: Vamos simular o funcionamento do algoritmo de busca binária para determinar se o item $x = 63$ pertence ao vetor $V =< 14, 27, 39, 46, 55, 63, 71, 80, 92 >$. Para indicar o intervalo em que será feita a busca, usaremos dois índices, i e f, e para indicar a posição central desse intervalo, o índice m.

Passo	x	i	f	m	$V[1]$	$V[2]$	$V[3]$	$V[4]$	$V[5]$	$V[6]$	$V[7]$	$V[8]$	$V[9]$
1º	63	1	9	5	14	27	39	46	**55**	63	71	80	92
2º	63	6	9	7						63	**71**	80	92
3º	63	6	6	6						**63**			

Figure 5: Simulação do método de busca binária, quando o item é encontrado

No primeiro passo o intervalo de busca compreende todo o vetor, desde a posição 1 até a posição 9 , e o item central ocupa a posição 5 . Como $x > V[5]$, a busca deve

[2] Evidentemente, qualquer sequência contendo um único item está trivialmente ordenada.

continuar na segunda metade do vetor. Para indicar isso, atualizamos o índice i com o valor 6 e repetimos o procedimento. No segundo passo, o intervalo de busca foi reduzido aos itens que ocupam as posições de 6 a 9 e o meio [3] do vetor é a posição 7. Agora $x < V[7]$ e, portanto, a busca deve continuar na primeira metade; então o índice f é atualizado com o valor 6 . No terceiro passo, resta um único item no intervalo de busca e o meio do vetor é a posição 6. Finalmente, temos $x = V[6]$ e a busca termina com sucesso.

Como em cada passo o intervalo de busca reduz-se aproximadamente à metade, é evidente que o processo de busca binária sempre termina, mesmo que o item procurado não conste do vetor. Nesse caso, porém, o processo somente termina quando não há mais itens a examinar, ou seja, quando o intervalo de busca fica vazio. Como o início e o final do intervalo são representados pelos índices i e f, o intervalo estará vazio se e somente se tivermos $i > f$.

Exemplo 6. Seja $x = 40$ e $L =< 14, 27, 39, 46, 55, 63, 71, 80, 92 >$. A tabela a seguir mostra o que acontece quando o item procurado não consta do vetor.

Passo	x	i	f	m	$L[1]$	$L[2]$	$L[3]$	$L[4]$	$L[5]$	$L[6]$	$L[7]$	$L[8]$	$L[9]$
1º	40	1	9	5	14	27	39	46	**55**	63	71	80	92
2º	40	1	4	1	14	**27**	39	46					
3º	40	3	4	3			**39**	46					
4º	40	4	4	4				46					
5º	40	**4**	**3**	?									

Figure 6: Simulação do método de busca binária, quando o item não é encontrado

Note que, durante a busca, o valor que indica o início do intervalo fica maior que aquele que indica o seu final (veja 5 o passo na tabela acima). Isso significa que não há mais itens a considerar e que, portanto, o item procurado não consta do vetor. Nesse caso, a busca deve terminar com fracasso.

2.4.1 Com base nos exemplos 5 e 6 siga as seguintes instruções:

1. Implemente um algoritmo que implemente uma busca binária. (0,2).

2. Simule a execução do algoritmo para um vetor de $n \geq 5$ elementos e apresente o resultado em uma tabela conforme a Figura 5, quando um item é encontrado (0,2);

3. Simule a execução do algoritmo para um vetor de $n \geq$ elementos e apresente o resultado em uma tabela conforme a Figura 6, quando um item não é encontrado (0,1);

[3] O meio do vetor é dado pelo quociente inteiro da divisão $(i + f)/2$, isto é, a média é truncada.

3 Check-List para submissão da atividade

A seguir está uma lista dos critérios mínimos necessários que sua atividade deve atender para ser considerada satisfatória. Observe que, embora todos esses sejam necessários, atender todos eles ainda pode não ser suficiente para considerar sua submissão é satisfatória.

1. Comente seu código! Se algum de seus métodos não for comentado corretamente, em relação ao propósito do método (a qual exercício ele corresponde) e parâmetros de entrada/saída, você perderá pontos por falta de documentação.

2. Teste cada uma de suas funções independentemente, sempre que possível, ou escreva scripts curtos para testá-los.

3. Envie em um arquivo compactado os arquivos `.java` (com a implementação dos algoritmos solicitados) e um arquivo `.pdf` com as simulação e questões a serem respondidas. O arquivo compactado deve conter o seu NOME

4 Referências

1. Pereira, S. L. Estruturas de Dados em C - Uma Abordagem Didática, 1ª ed., Editora Saraiva, 2016