PESQUISA, ORDENAÇÃO E TÉCNICAS DE ARMAZENAMENTO

UNIDADE 2 - ORDENAÇÃO INTERNA E ALGORITMOS DE ORDENAÇÃO

Andrea Karina Garcia

Introdução

A informação e o conhecimento são simultaneamente causa e efeito entre si. A interação é dinâmica e a sucessão, sua produção, pode ser plenamente invertida, sem gerar nenhuma contradição, proporcionando expansão benéfica a ambos. Portanto, disponibilizar informação é promover a geração de conhecimento. (COSTA e XAVIER, 2010). Mas, como a informação é registrada, armazenada, organizada, selecionada e recuperada por suportes tecnológicos?

Uma instituição financeira (banco), por exemplo, classifica os cheques pelos números de conta, de modo que possa preparar extratos bancários individuais. De maneira geral, todas as empresas têm a necessidade de classificar seus dados, muitas vezes em volumes maciços. O verdadeiro propósito de um sistema de informação deve ser em termos dos usos dados à informação e dos efeitos resultantes desses usos nas atividades dos usuários? A função mais importante do sistema seria a forma como a informação modifica a realização dessas atividades? (LE COADIC, 1994).

No Brasil, o Marco Legal da Ciência, Tecnologia e Inovação, de acordo com Rocha (2018), traz novas possibilidades de interação entre a iniciativa privada e a Administração Pública, com vistas ao aperfeiçoamento dos modelos de organização interna das instituições de Ciência, Tecnologia e Inovação (ICTs). Segundo o autor, a adoção de ferramentas computacionais adequadas possibilita a elucidação das possíveis interpretações sobre o contexto político, social e econômico, e permite identificar novos rumos para o desenvolvimento de projetos estratégicos nas mais diversas áreas. Os resultados, assim, poderão ser consolidados e analisados pelas instâncias de planejamento estratégico das instituições que implementam as tais políticas públicas.

Azevedo Junior e Campos (2008 *apud* ROCHA, 2018) afirmam que, quanto mais rápido um negócio puder alterar seus processos e o sistema de informação, os quais servem a ele de suporte, mais preparado estará para reagir a eventos de concorrência de mercado. Os sistemas de informação, portanto, compreendem requisito imprescindível em um cenário cada vez mais dinâmico.

Classificar dados é o ato de colocar os dados em uma ordem particular e específica, crescente ou decrescente. É uma das aplicações mais importantes da computação. Vale ressaltar, desde já, que independentemente da classificação, ou seja, do algoritmo utilizado para classificar o *array*, o resultado final será o mesmo. Será a escolha do algoritmo, seu tempo de execução e uso de memória do programa, que farão a diferença.

Nesta unidade, você terá a oportunidade de aprender sobre formas de ordenação interna e as distintas possibilidades de escolha de algoritmos de ordenação, tais como: 1) Bolha; 2) Inserção; 3) Seleção; 4) Shellsort e Mergesort; 5) Quicksort; 6) Heaps Binários; 7) Heapsort; 8) Countingsort; 9) Bucketsort e Radixsort.

Classificar dados é um problema instigante, que demanda esforços intensos de investimento financeiro e pesquisas científicas. Convido a você, caro aluno, a mergulhar neste segmento de universo fantástico da computação. Bons estudos!

2.1 Ordenação interna

Do ponto de vista da memória do computador, os algoritmos de ordenação podem ser classificados em (COELHO; FÉLIX, s. d.):

Ordenação INTERNA	Ordenação EXTERNA
Quando os dados a serem ordenados estão na memória principal.	Quando os dados a serem ordenados necessitam de armazenamento em memória auxiliar, como o HD.

Quadro 1 - Conceitos de ordenação Interna e Externa.
Fonte: Elaborado pela autora, 2019.

Quando os dados a serem ordenados estão na memória principal, são chamados de ordenação interna. Quando os dados a serem ordenados necessitam de armazenamento em memória auxiliar, como o HD, são chamados de ordenação externa.
Sendo assim, esta unidade terá seu foco nos métodos de ordenação interna.

2.2 O problema da ordenação

Em um processo de seleção de um algoritmo de ordenação interna, você deve considerar os seguintes aspectos: o tempo gasto pela ordenação e o uso econômico da memória disponível. Segundo Menotti (s. d.), os métodos de ordenação "*in situ*" são os preferidos. A expressão "*in situ*" é usada na computação para definir uma operação que ocorre sem interromper o estado normal do sistema. Ao mesmo tempo, métodos que utilizam listas encadeadas não são muito utilizados. E, para finalizar, métodos que fazem cópias dos itens a serem ordenados, têm menor importância.
Sobre o critério de avaliação, sendo "n" o número de registros no arquivo, as medidas de complexidade relevantes são, ainda de acordo com o autor:

- número de comparações C (n) entre chaves;
- número de movimentações M (n) de itens.

A classificação é um mecanismo que ordena os dados em uma ordem crescente ou decrescente, com base em uma ou mais chaves de classificação. Por exemplo, uma lista de nomes poderia ser classificada alfabeticamente; contas bancárias poderiam ser classificadas pelo número de conta; registros de folhas de pagamento de funcionários poderiam ser classificados pelo CPF; e assim por diante.

```
typedef int ChaveTipo;
typedef struct
{
ChaveTipo Chave;
/* outros componentes */
} Item;
```

Outras características importantes referem-se à estabilidade, relativa "à manutenção da ordem original de itens de chaves iguais". (MENOTTI, s. d.). Em outras palavras, "um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação". (MENOTTI, s. d.). Ressalta-se que, na ordenação interna, o arquivo a ser ordenado cabe todo na memória principal.
Os métodos de classificação de ordenação interna são categorizados dessa forma:

Métodos simples	Adequados para pequenos arquivos. • Requerem O (n2) comparações. • Produzem programas pequenos.
Métodos eficientes:	Adequados para arquivos maiores. • Requerem O (n log n) comparações. • Usam menos comparações. • As comparações são mais complexas nos detalhes. • Métodos simples são mais eficientes para pequenos arquivos.

Quadro 2 - Classificação dos métodos de ordenação interna.
Fonte: MENOTTI, s. d.

Os algoritmos de ordenação podem ser aplicados a diversos tipos de estrutura, tais como: vetores, matrizes e estruturas dinâmicas (COELHO; FÉLIX, s. d.). Dois algoritmos simples de classificação são: classificação por seleção e por inserção. A classificação por intercalação é mais eficiente e, ao mesmo tempo, mais complexa.

2.3 Introdução e exemplo dos algoritmos de ordenação: Bolha; Inserção; Seleção; Shellsort; Mergesort; Quicksort; Heaps Binários; Heapsort; Countingsort; Bucketsort e Radixsort.

Na sociedade contemporânea, o acúmulo de dados torna-se um problema a cada dia mais complexo a ser resolvido. Aprimorar o uso de algoritmos de ordenação, que são processos lógicos para se organizar uma determinada estrutura linear, é imprescindível. (AGUILAR, 2018).
Este estudo concentrará seus esforços em três deles, dentre os diversos métodos expostos: *BubbleSort*, *InsertionSort*, *SelectionSort*. Didaticamente falando, introduzem ideias que servem de base para outros métodos mais eficientes. Esses métodos utilizam como uma de suas operações básicas a comparação de elementos da lista, ou seja, o foco portanto, é didático e não por desempenho.

2.3.1 Bolha (BubbleSort)

A classificação por bolha é um algoritmo de classificação simples. A ideia da ordenação por bolhas é flutuar o maior elemento para o fim; por este motivo, deve-se repetir "n" vezes a flutuação. *Bubble Sort* é um algoritmo de ordenação que pode ser aplicado em *arrays* e listas dinâmicas.
Nesse método, os elementos da lista são movidos para as posições adequadas de forma contínua, "assim como uma bolha move-se num líquido", nas palavras de Cintra *et al.* (2015, p. 24). Se um elemento está inicialmente numa posição 'i' e, para que a lista fique ordenada, ele deve ocupar a posição 'j', ele terá que passar por todas as posições entre 'i' e 'j'. (CINTRA; NOBRE; VIANA, 2015, p. 24).
Ainda de acordo com os autores, em cada iteração do método, percorre-se a lista, partindo de seu início, comparando cada elemento com seu sucessor, trocando-os de posição se houver necessidade. É possível demonstrar que, se a lista tiver n elementos, após no máximo (n–1) iterações, a lista estará em ordem.
O algoritmo *BubbleSort* percorre o vetor muitas vezes; por isso, não é recomendado o uso para aplicações que exigem velocidade ou tratem de uma grande quantidade de dados. Acompanhe o exemplo abaixo, baseado em Cintra *et al.* (2015, p. 24):

```
ALGORITMO BOLHA
      ENTRADA: UM VETOR L COM N POSIÇÕES
      SAÍDA: O VETOR L EM ORDEM CRESCENTE
      PARA i = 1 até n - 1
            PARA j = 0 até n - 1 - i
                  SE L[j] > L[j+1]
                        AUX = L[j] // SWAP
                        L[j] = L[j+1]
                        L[j+1] = AUX
FIM <BOLHA>
```

Primeiro, será realizada a troca, o *swap*, de elementos da lista. Depois, será feito o *swap* usando o procedimento troca. Assim, a chamada troca (L[i], L[j]) serve para trocar o conteúdo das posições i e j do vetor L.
Percebe-se que ao final da primeira iteração do laço mais externo do algoritmo, o maior elemento da lista estará na última posição do vetor. Ao final da segunda iteração, os dois maiores valores da lista estarão nas duas últimas posições do vetor L, em ordem crescente.
Ao final da iteração 'k' do laço mais externo, os 'k' maiores valores da lista estarão nas 'k' últimas posições do vetor, em ordem crescente. Claramente, instrução crítica do algoritmo bolha é a comparação (se). Em cada iteração do laço mais interno é feita uma comparação. A complexidade temporal desse algoritmo pertence a O (n2). Além dos parâmetros de entrada (L e n), o algoritmo utiliza apenas três variáveis escalares (i, j e aux). Dessa forma, a quantidade extra de memória utilizada por ele é constante.
Outro aspecto importante dos métodos de ordenação é a estabilidade. Um método de ordenação é estável se ele preserva a ordem relativa existente entre os elementos repetidos da lista. Por exemplo, se a ordenação da lista (4, 7, 6, 7, 2) resulta em (2, 4, 6, 7, 7), a ordenação foi estável. Observe o conteúdo de um vetor após cada iteração do laço mais externo, no quadro a seguir.

	0	1	2	3	4	5
Lista Original	9	4	5	10	5	8
1ª iteração	4	5	9	5	8	10
2ª iteração	4	5	5	9	9	10
3ª iteração	4	5	5	9	9	10
4ª iteração	4	5	5	9	9	10
5ª iteração	4	5	5	9	9	10

Quadro 3 - Iteração com BubbleSort.
Fonte: CINTRA; NOBRE; VIANA, 2015. p. 26.

De acordo com o exemplo acima, no final da segunda iteração a lista já estava ordenada. Na terceira iteração não houve nenhuma alteração na lista. Desse fato, conclui-se que a lista já estava em ordem, de modo que as duas últimas iterações foram inúteis.
Há outras formas de implementar o algoritmo *BubbleSort*. Entendendo o funcionamento desse código, você poderá implementá-lo de outras formas. Aliás, podemos testar juntos, a partir do VISUALGO.

VOCÊ SABIA?

Os softwares educativos são amplificadores de potencialidade, na capacitação e aperfeiçoamento. Eles são considerados programas educacionais quando projetados por meio de uma metodologia que os contextualizem no processo ensino-aprendizagem. O "VISUALGO" é um exemplo disso; nele você poderá rodar exemplos de ordenação por bolhas. Entenda a teoria a partir da prática. É só clicar neste link <https://visualgo.net/pt/sorting>.

As tecnologias de informação e comunicação propiciam trabalhar com investigação e experimentação do conteúdo teórico, ampliando a sua experiência, de modo a fomentar e construir o seu próprio conhecimento.

2.3.2 Inserção (InsertSort)

A classificação por inserção é um algoritmo de classificação simples. A primeira iteração desse algoritmo seleciona o segundo elemento no *array* e, se for menor que o primeiro elemento, troca-o pelo primeiro elemento. A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos sejam na ordem. Na *i*-ésima interação desse algoritmo, os primeiros elementos *i*, no *array* original, serão classificados.
Observe o exemplo abaixo, conforme Deitel e Deitel (2015. p. 644.):
***Array*:**
34 56 4 10 77 51 93 30 5 52
Um programa que implementa o algoritmo de classificação por inserção analisará os dois primeiros elementos do *array*, 34 e 56. Estes já estão em ordem. Portanto, o programa continua. Caso eles estivessem fora de ordem, o programa iria permutá-los.
Na próxima iteração, o programa examina o terceiro valor, que é o número 4. Esse valor é menor que 56. Portanto, o programa armazena 4 e, sendo assim, move o 34 uma posição para a direita. O programa agora alcançou o começo do *array*, colocando o 4 no zero-ésimo elemento.
4 34 56 10 77 51 93 30 5 52
Na próxima iteração, o programa armazena 10 em uma variável temporária. Então, compara 10 a 56, e move o 56 uma posição para a direita, porque ele é maior que 10. O programa, então, compara 10 com 34, movendo o 34 uma posição para a direita. Quando o programa compara 10 com o 4, ele observa que o 10 é maior que o 4, e o coloca o 10 na posição 1.
4 10 34 56 77 51 93 30 5 52
Usando esse algoritmo, na *i*-ésima interação, os primeiros elementos *i* do *array* são classificados. Contudo, podem ainda não estar nos locais finais. Valores menores podem ser localizados mais tarde no *array*.
Sobre a implementação da classificação por inserção, a classe *InsertionSortTest* é composta pelos métodos descritos abaixo. Clique e confira!

O método static insertionSort, que serve para classificar "ints" usando o algoritmo de classificação por inserção.
O método static printPass, que serve para exibir o conteúdo do array, depois de cada passagem.
"Main" para testar o método insertionSort.

```java
// Figura 19.5: InsertionSortTest.java
// Classificando um array com a classificação por inserção.
import java.security.SecureRandom;
import java.util.Arrays;

public class InsertionSortTest
{
   // classifica o array utilizando a classificação por inserção
   public static void insertionSort(int[] data)
   {
      // faz um loop sobre data.length - 1 elementos
      for (int next = 1; next < data.length; next++)
      {
         int insert = data[next]; // valor a inserir
         int moveItem = next; // local para inserir elemento

         // procura o local para colocar o elemento atual
         while (moveItem > 0 && data[moveItem - 1] > insert)
         {
            // desloca o elemento direito um slot
            data[moveItem] = data[moveItem - 1];
            moveItem--;
         }

         data[moveItem] = insert; // local do elemento inserido
         printPass(data, next, moveItem); // passagem de saída do algoritmo
      }
   }

   // imprime uma passagem do algoritmo
   public static void printPass(int[] data, int pass, int index)
   {
      System.out.printf("after pass %2d: ", pass);

      // gera saída dos elementos até o item trocado
      for (int i = 0; i < index; i++)
         System.out.printf("%d  ", data[i]);

      System.out.printf("%d* ", data[index]); // indica troca

      // termina de gerar a saída do array
      for (int i = index + 1; i < data.length; i++)
         System.out.printf("%d  ", data[i]);

      System.out.printf("%n               "); // para alinhamento

      // indica quantidade do array que é classificado
      for(int i = 0; i <= pass; i++)
         System.out.print("--  ");
      System.out.println();
   }

   public static void main(String[] args)
   {
      SecureRandom generator = new SecureRandom();

      int[] data = new int[10]; // cria o array

      for (int i = 0; i < data.length; i++) // preenche o array
         data[i] = 10 + generator.nextInt(90);

      System.out.printf("Unsorted array:%n%s%n%n",
         Arrays.toString(data)); // exibe o array
      insertionSort(data); // classifica o array

      System.out.printf("Sorted array:%n%s%n%n",
```

```
67              Arrays.toString(data)); // exibe o array
68       }
69    } // fim da classe InsertionSortTest
```

```
Unsorted array:
[34, 96, 12, 87, 40, 80, 16, 50, 30, 45]
after pass  1: 34  96* 12  87  40  80  16  50  30  45
               --  --
after pass  2: 12* 34  96  87  40  80  16  50  30  45
               --  --  --
after pass  3: 12  34  87* 96  40  80  16  50  30  45
               --  --  --  --
after pass  4: 12  34  40* 87  96  80  16  50  30  45
               --  --  --  --  --
after pass  5: 12  34  40  80* 87  96  16  50  30  45
               --  --  --  --  --  --
after pass  6: 12  16* 34  40  80  87  96  50  30  45
               --  --  --  --  --  --  --
after pass  7: 12  16  34  40  50* 80  87  96  30  45
               --  --  --  --  --  --  --  --
after pass  8: 12  16  30* 34  40  50  80  87  96  45
               --  --  --  --  --  --  --  --  --
after pass  9: 12  16  30  34  40  45* 50  80  87  96
               --  --  --  --  --  --  --  --  --  --
Sorted array:
[12, 16, 30, 34, 40, 45, 50, 80, 87, 96]
```

Figura 1 - Classificando um array com o método de inserção.
Fonte: DEITEL, 2015. p. 644 e 645.

Descrição:
Linhas 9 a 28: declaram o método *insertionSort*;
Linhas 12 a 27: fazem um *loop* por itens "*data.lenght* - 1" no *array*;
Linha 14: a cada iteração, esta linha declara e inicializa a variável "*insert*", que contém o valor do elemento que será inserido na parte classificada do *array*;
Linha 15: declara e inicializa a variável "*moveItem*", que monitora onde inserir o elemento;
Linhas 18 a 23: fazem um *loop* para localizar a posição correta onde o elemento deve ser inserido. O *loop* terminará quando o programa alcançar o início do *array*, ou quando alcançar um elemento menor que o valor a ser inserido;
Linha 21: move um elemento para a direita no *array*;
Linha 22: decrementa a posição na qual inserir o próximo elemento;
Linha 25: depois do *loop* terminar, esta linha insere o elemento na posição;
Linhas 31 a 51: a saída do método *printPass* utiliza traços para indicar a parte do array que é classificada após cada passagem. Um asterisco é colocado ao lado da posição do elemento que foi trocado pelo menor emento, nessa passagem.

O algoritmo de classificação por inserção é executado no tempo O (n^2). A implementação da classificação por inserção (linhas 9 a 28) contém dois *loops* "*for*" (linhas 12 a 27) itera "*data.lenght* – 1" vezes, inserindo um elemento na posição apropriada nos elementos classificados até o momento. Para o propósito desse aplicativo, "*data.lenght* – 1" é equivalente a "n – 1" comparações. Cada *loop* individual é executado no tempo O (n). (DEITEL; DEITEL, 2015).

VOCÊ QUER VER?

Ao encerrar o segundo método de ordenação, você ainda está com algumas dúvidas? Assista ao vídeo romeno de dança folclórica, que traz uma demonstração sobre o algoritmo de inserção. Produzido pela Sapientia University e dirigido por Kátai Zoltán e Tóth Lázló. O nome do canal é AlgoRythmics. Tenho certeza de que você irá solucioná-las depois de assistir esses vídeos!Há outros vídeos interessantíssimos sobre o tema. Vale a pena conferir! É só clicar neste link < https://www.youtube.com/watch?time_continue=35&v=ROalU379l3U>.

O recurso audiovisual é uma importante ferramenta, que proporciona o aprendizado por meio do lúdico. As possibilidades de ensino e aprendizagem se ampliam, contribuindo para a sua compreensão e assimilação dos conteúdos. Além de solucionar suas dúvidas sobre o assunto anterior, ainda teve a oportunidade de assistir um vídeo sobre o próximo assunto: o método de Seleção.

2.3.3 Seleção (SelectionSort)

A classificação por seleção é outro algoritmo de classificação simples. Dentro de uma necessidade e escolha de classificação em ordem crescente, a primeira iteração selecionará o menor elemento no *array*, permutando pelo primeiro elemento.

A segunda iteração selecionará o segundo menor item, o menor dos elementos restantes, de modo a trocá-lo pelo segundo elemento. O algoritmo prosseguirá em seu ritmo de trabalho, até que a última iteração selecione o segundo maior elemento, e permute-o pelo penúltimo índice, deixando o maior elemento no ultimo índice.

Depois da *i*-ésima iteração, os menores itens *i* do *array* serão classificados na ordem crescente nos primeiros elementos *i* do *array*. Observe o exemplo abaixo, conforme Deitel e Deitel (2015. p. 641):

Array:

34 56 4 10 77 51 93 30 5 52

Um programa que implementa a classificação por seleção primeiro determina o menor valor, que é o 4 neste *array*, o qual está contido no índice 2. Então, o programa troca 4 por 34, resultando em:

4 56 34 10 77 51 93 30 5 52

O programa, depois disso, determina o menor valor dos elementos restantes, dentre todos, menos o 4. Neste caso, é o 5, presente no índice 8. Então, o programa troca 5 por 56.

4 5 34 10 77 51 93 30 56 52

Na terceira iteração, o programa determina o próximo menor valor, que é o 10, trocando-o pelo 34.

4 5 10 34 77 51 93 30 56 52

O processo continua, até que o *array* seja completamente classificado.

4 5 10 30 34 51 52 56 77 93

Sendo assim, compreende-se que: o menor elemento está na primeira posição. Depois da segunda iteração, os dois menores elementos assumirão, de acordo com a ordem, as duas primeiras posições. E assim por diante.

Sobre a implementação da classificação por seleção (*SelectionSortTest*), é composto pelos seguintes métodos:

- o método *static SelectionSort*, que serve para classificar um "*array int*", usando o algoritmo de classificação por seleção;
- o método *static swap*, que serve para permutar os valores dos dois elementos do *array*;
- o método *static printPass*, que serve para exibir o conteúdo do *array*, depois de cada passagem;
- "*main*" para testar o método *selectionSort*.

No exemplo de pesquisa, baseado em Deitel e Deitel (2015), a seguir, o "*main*" (linhas 57 a 72) cria um *array* de "*ints*" nomeados data, preenchendo-os com "*ints*" aleatórios, no intervalo 10 a 99. Sendo assim, a linha 68 testa o método *selectionSort*.

```java
// Figura 19.4: SelectionSortTest.java
// Classificando um array com classificação por seleção.
import java.security.SecureRandom;
import java.util.Arrays;

public class SelectionSortTest
{
   // classifica o array utilizando a classificação por seleção
   public static void selectionSort(int[] data)
   {
      // faz um loop sobre data.length - 1 elementos
      for (int i = 0; i < data.length - 1; i++)
      {
         int smallest = i; // primeiro índice do array remanescente

         // faz um loop para localizar o índice do menor elemento
         for (int index = i + 1; index < data.length; index++)
            if (data[index] < data[smallest])
               smallest = index;

         swap(data, i, smallest); // troca o menor elemento na posição
         printPass(i + 1, smallest); // passagem de saída do algoritmo
      }
   } // finaliza o método SelectionSort

   // método auxiliar para trocar valores em dois elementos
   private static void swap(int[] data, int first, int second)
   {
      int temporary = data[first]; // armazena o primeiro em temporário
      data[first] = data[second]; // substitui o primeiro pelo segundo
      data[second] = temporary; // coloca o temporário no segundo
   }

   // imprime uma passagem do algoritmo
   private static void printPass(int[] data, int pass, int index)
   {
      System.out.printf("after pass %2d: ", pass);

      // saída de elementos até item selecionado
      for (int i = 0; i < index; i++)
         System.out.printf("%d  ", data[i]);

      System.out.printf("%d* ", data[index]); // indica troca

      // termina de gerar a saída do array
      for (int i = index + 1; i < data.length; i++)
         System.out.printf("%d  ", data[i]);

      System.out.printf("%n               "); // para alinhamento

      // indica quantidade do array que é classificado
      for (int j = 0; j < pass; j++)
         System.out.print("--  ");
      System.out.println();
   }

   public static void main(String[] args)
```

```
57      public static void main(String[] args)
58      {
59         SecureRandom generator = new SecureRandom();
60
61         int[] data = new int[10]; // cria o array
62
63         for (int i = 0; i < data.length; i++) // preenche o array
64            data[i] = 10 + generator.nextInt(90);
65
66         System.out.printf("Unsorted array:%n%s%n%n",
67            Arrays.toString(data)); // exibe o array
68         selectionSort(data); // classifica o array
69
70         System.out.printf("Sorted array:%n%s%n%n",
71            Arrays.toString(data)); // exibe o array
72      }
73   } // fim da classe SelectionSortTest
```

```
Unsorted array:
[40, 60, 59, 46, 98, 82, 23, 51, 31, 36]
after pass  1: 23  60  59  46  98  82  40* 51  31  36
               --
after pass  2: 23  31  59  46  98  82  40  51  60* 36
               --  --
after pass  3: 23  31  36  46  98  82  40  51  60  59*
               --  --  --
after pass  4: 23  31  36  40  98  82  46* 51  60  59
               --  --  --  --
after pass  5: 23  31  36  40  46  82  98* 51  60  59
               --  --  --  --  --
after pass  6: 23  31  36  40  46  51  98  82* 60  59
               --  --  --  --  --  --
after pass  7: 23  31  36  40  46  51  59  82  60  98*
               --  --  --  --  --  --  --
after pass  8: 23  31  36  40  46  51  59  60  82* 98
               --  --  --  --  --  --  --  --
after pass  9: 23  31  36  40  46  51  59  60  82* 98
               --  --  --  --  --  --  --  --  --
Sorted array:
[23, 31, 36, 40, 46, 51, 59, 60, 82, 98]
```

Figura 2 - Classificando um array com o método de seleção.
Fonte: DEITEL, 2015. p. 643.

Descrição:
Linhas 9 a 24: declaram o método *selectionSort*;
Linhas 12 a 23: fazem um *loop* "*data.lenght* - 1" vezes;
Linha 14: declara a inicialização – para o índice *i* atual – a variável *smallest*, a qual armazena o índice do menor elemento no *array* remanescente;
Linhas 17 a 19: fazem um *loop* sobre os elementos restantes no *array*;
Linha 18: para cada um desses elementos, esta linha compara seu valor com o valor do menor elemento.
Linha 19: se o elemento atual for menor que o menor elemento, esta linha atribui o índice do elemento atual a *smallest*. Quando esse *loop* termina, *smallest* conterá o índice do menor elemento no *array* restante.
Linha 21: esta, chama o método *swap*.

Linhas 27 a 32: o método *swap* é demonstrado nas linhas 27 a 32, para colocar o menor elemento restante na próxima área ordenada do *array*.
Linhas 52 e 53: a saída do método *printPass* utiliza traços para indicar a parte do *array* que é classificada após cada passagem.
Linha 43: um asterisco é colocado ao lado da posição do elemento que foi trocado pelo menor emento, nessa passagem. A cada passagem, o elemento ao lado do asterisco, especificado na linha 43, e o elemento acima do conjunto de traços mais à direita foram permutados.

O algoritmo de classificação por seleção é executado no tempo O (n^2). O método *selectionSort* (linhas 9 a 24) tem dois *loops* "*for*". O *loop* externo (linhas 12 a 23), itera pelos primeiros "n -1" elementos do *array* e coloca o menor item restante na sua posição classificada. O *loop* interno (linhas 17 a 19) itera por cada item no *array* restante, em busca do menor elemento. Esse *loop* é executado "n – 1" vezes durante a primeira iteração do *loop* externo, "n – 2" vezes durante a segunda iteração e, então, "n – 3", e assim por diante. O *loop* interno irá iterar, portanto, um total de "n (n – 1) /2 ou (n^2 – n) /2". (DEITEL; DEITEL, 2015). Agora, convido você, caro aluno, a colocar em prática seus conhecimentos. Vamos lá?

VAMOS PRATICAR?

1. Como os algoritmos de ordenação podem ser classificados, do ponto de vista da memória do computador?

Resolução

Os algoritmos de ordenação podem ser classificados da seguinte maneira: ordenação interna (quando os dados a serem ordenados estão na memória principal); ordenação externa (quando os dados a serem ordenados necessitam de armazenamento em memória auxiliar, como o HD). 2. O que é o *BubbleSort*?

Resolução

É um método de classificação, um algoritmo de classificação simples. A ideia da ordenação por bolhas é flutuar o maior elemento para o fim. Por este motivo, deve-se repetir "n" vezes a flutuação. O *Bubble Sort* é um algoritmo de ordenação que pode ser aplicado em *arrays* e listas dinâmicas. No caso de uma ordenação decrescente, por exemplo, a posição atual dos elementos é comparada com a próxima posição. Se a posição atual for maior que a posição posterior, é realizada a troca dos valores nessa posição. Caso contrário, não é realizada a troca, apenas passa-se para o próximo par de comparações. O algoritmo *Bubble Sort* percorre todo o vetor diversas vezes, por isso, não é recomendado o uso dele para aplicações que requerem velocidade ou trabalhem com uma grande quantidade de dados.

3. O que é o *SelectionSort*?

Resolução

É um método de classificação por seleção, um algoritmo de classificação simples. Dentro de uma necessidade e escolha de classificação em ordem crescente, a primeira iteração selecionará o menor elemento no *array*, permutando pelo primeiro elemento. A segunda iteração selecionará o segundo menor item, o menor item dos elementos restantes, de modo a trocá-lo pelo segundo elemento. O algoritmo prosseguirá em seu ritmo de trabalho, até que a última iteração selecione o segundo maior elemento, e permute-o pelo penúltimo índice, deixando o maior elemento no ultimo índice. Depois da *i*-ésima iteração, os menores itens *i* do *array* serão classificados na ordem crescente nos primeiros elementos *i* do *array*.

Ao finalizar as atividades, você teve a oportunidade de fixar o conteúdo e fortalecer sua memória, além de consolidar hábitos de disciplina importantes para sua vida acadêmica. Parabéns!

2.3.4 MergeSort e ShellSort

O *MergeSort* é um método baseado em uma "estratégia de resolução de problemas conhecida como divisão e conquista". (CINTRA, NOBRE, VIANA, 2015, p. 32). Essa técnica consiste em decompor a instância a ser resolvida em instâncias menores do mesmo tipo de problema, e, por fim, utilizar as soluções parciais para obter uma solução da instância original. Para que seja viável aplicar essa técnica a um problema ele deve possuir duas propriedades estruturais; ele precisa se decompor em qualquer instância não trivial do problema, assim como em instâncias menores do mesmo tipo de problema. (CINTRA, NOBRE, VIANA, 2015, p. 32).
Segundo Lima *et al.* (2017, p. 172), o *MergeSort* é um algoritmo de ordenação mediano, "Devido à recursividade ser sua principal ferramenta, seu melhor resultado é com relação às estruturas lineares aleatórias". Entretanto, ao tratar de estrutura pequena e/ou já pré-ordenada (crescente ou decrescente), a recursividade passa a ser uma desvantagem. Essa peculiaridade faz com que o consumo de tempo de processamento seja alto e trocas desnecessárias sejam realizadas. Portanto, o algoritmo é indicado para aplicações com estruturas lineares em que a divisão em estruturas menores sejam o objetivo. Exemplo: em filas para operações bancárias. (LIMA; RICARTE; SOUZA, 2017).

VOCÊ O CONHECE?

John von Neumann foi quem, em 1945 criou o algoritmo de ordenação *MergeSort*. Neumann foi um matemático húngaro, de origem judaica que se naturalizou americano. Contribuiu na Teoria dos Conjuntos, Análise Funcional, Teoria Ergótica, Mecânica Quântica, Teoria dos Jogos, Análise Numérica, Hidrodinâmica, Estatística, Ciência da Computação, entre outras áreas. É considerado um dos mais importantes matemáticos do século XX.

O método *Shell Sort*, segundo Cintra *et al*. (2015. p. 30) "é um exemplo de algoritmo fácil de descrever e implementar, mas difícil de analisar". É considerado uma extensão do algoritmo de ordenação por inserção, pois permite a troca de registros distantes um do outro.
Segundo Lima *et al.* (2017), é um dos métodos que mais apresenta resultados satisfatórios, sobretudo com estruturas maiores e desorganizadas. Por ser considerado uma melhoria do *SelectionSort*, o *ShellSort*, ao ser utilizado com as mesmas finalidades que seu predecessor (recursos que demandem pouca escrita), irá apresentar um melhor desempenho, e, consequentemente, expandir a vida útil dos recursos. (LIMA; RICARTE; SOUZA, 2017).

2.3.5 QuickSort

O algoritmo *QuickSort* é o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações. Provavelmente, é o mais utilizado. Possui complexidade "C (n) = O (n log n)"no melhor e médio caso, "C (n) = O (n^2) no pior caso. Por esse motivo, não é um algoritmo estável.
Bem como o *Mergesort*, "o Quicksort também é baseado na estratégia de divisão e conquista", de acordo com Cintra *et al*. (2015, p. 30). Com relação a esse aspecto, o que os diferencia é o fato de que enquanto no *Mergesort* a fase de divisão é trivial e a fase de conquista é trabalhosa, no *Quicksort* acontece exatamente o contrário.

Basicamente, a ideia do método é dividir um conjunto com "n" itens. Ou seja, dividi-lo em "dois problemas menores". Os problemas menores são ordenados independentemente, e os resultados são combinados para produzir a solução final. A operação do algoritmo divide sua lista de entrada em duas sub-listas, a partir de um pivô. Em seguida, o mesmo procedimento nas duas listas menores, até uma lista unitária é realizado.

2.3.6 Heaps Binários

O segredo do algoritmo *Heapsort* é uma estrutura de dados, conhecida como *heap*, que trata o vetor como uma árvore binária. Há dois tipos estrutura: *max-heap* e *min-heap* (FEOFILOFF, 2018). Um *heap* é um vetor em que o valor de todo pai é maior ou igual ao valor de cada um de seus dois filhos. Portanto, um vetor v [1..m] é um heap se:

$$v\,[f/2] \geq v[f]$$

Obs: para f = 2 ; . . . ; m .

1	2	3	4	5	6	7	8	9	10	11	12	13	14
999	888	777	555	666	777	555	222	333	444	111	333	666	333

Figura 3 - Ilustração de Vetor heap.
Fonte: FEOFILOFF, 2018.

É uma operação relativamente fácil rearranjar os elementos de um vetor de inteiros v [1..m] para que ele se torne um *heap*. O processo deve ser repetido enquanto o valor de um filho for maior que o de seu pai. Então, trocam-se os valores de pai e filho. Assim, um passo em direção à raiz é executado. Mais precisamente, enquanto v[f/2] < v[f], faça troca (v[f/2], v[f]) e em seguida f = f/2 (FEOFILOFF, 2018). A operação de troca é definida assim:
#define troca (A, B) {int t = A; A = B; B = t;}
Para implementar um *heap* binário usando vetor, considere a seguinte estrutura, portanto: 1) armazenamento dos dados (vetor); 2) indicação do tamanho do vetor; 3) indicação da última posição utilizada pelo vetor.

2.3.7 HeapSort

O algoritmo *HeapSort* tem duas fases: a primeira transforma o vetor em *heap* e a segunda rearranja o *heap* em ordem crescente. Observe o esboço a seguir.

```
// Rearranja os elementos do vetor v[1..n]
// de modo que fiquem em ordem crescente.

void
heapsort (int n, int v[])
{
   constroiHeap (n, v);
   for (int m = n; m >= 2; --m) {
      troca (v[1], v[m]);
      peneira (m-1, v);
   }
}
```

Figura 4 - Fases dos vetores.
Fonte: FEOFILOFF, 2018.

No início de cada iteração do *for* {} valem as seguintes propriedades invariantes:

- o vetor v [1..m] é um heap;
- v [1..m] ≤ v[m+1..n];
- v [m+1..n] está em ordem crescente;
- v [1..n] é uma permutação do vetor original.

A expressão v [1..m] ≤ v [m+1..n] é uma abreviatura de todos os elementos de v [1..m] são menores ou iguais a todos os elementos de v [m+1..n]. Dessa forma, observe (FEOFILOFF, 2018):

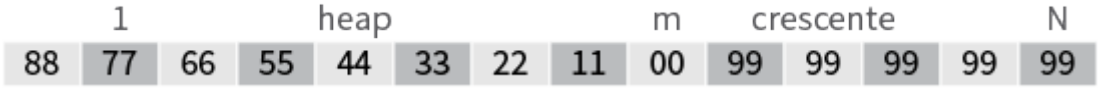

Figura 5 - Exemplo de HeapSort.
Fonte: FEOFILOFF, 2018.

Segue daí que v [1..n] estará em ordem crescente quando m for igual a 1. Portanto, o algoritmo está correto. E por falar em correto, caro aluno, você já pensou na importância de problematizar suas aplicações, também, no contexto social que você está inserido?

VOCÊ QUER LER?

Você, estudante de um curso de tecnologia, já parou para pensar na importância de problematizar suas aplicações? "Admirável mundo novo" foi escrito pelo inglês Aldous Huxley, em 1931. A estória é ambientada em Londres, no ano de 2540. Ele antecipa questões relacionadas à tecnologia reprodutiva, hipnopedia e manipulação psicológica. A obra é considerada uma das 100 melhores do século XX.

O *Heapsort* é portanto, baseado no uso de *heap* binário. Inicialmente, são inseridos os elementos da lista num *heap* binário crescente. Em seguida, são feitas sucessivas remoções do menor elemento do *heap*, colocando os elementos removidos do *heap* de volta na lista. Então, a lista estará em ordem crescente. (CINTRA e VIANA, 2011).

2.3.8 CountingSort, BucketSort e RadixSort

O *CountingSort* é um algoritmo de ordenação estável e de complexidade O (n). Basicamente, o ele tem o objetivo de determinar, para cada entrada "x", o número de elementos menor que "x". Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no *array* de saída.
Clique nas setas abaixo e aprenda mais sobre o tema.
O método *BucketSort* permite ordenar listas. Para uma lista com "n" elementos, é usado um vetor de ponteiros (*bucket*) com "n" posições. O maior elemento é indicado por "k". Cada posição do *bucket* apontará para uma lista encadeada, a partir da qual serão inseridos os elementos da lista que pertencem ao intervalo [i * (k + 1) / n, (i + 1) * (k +1) /n). (CINTRA, NOBRE, VIANA, 2015). Quando o vetor contém valores que são uniformemente distribuídos, o método tem complexidade considerada linear O (n).
O *RadixSort* é um algoritmo de ordenação rápido e estável. É um método que permite ordenar listas, cujos elementos sejam comparados a cada dois. Ele pode ser usado para ordenar itens identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o *RadixSort* ordena estas chaves em qualquer ordem relacionada com a lexicografia. A cada iteração, ordena-se a lista por uma de suas posições, começando pela posição mais significante, pois os elementos da lista devem ser representados como uma mesma quantidade de posições.
Por esse motivo, é um algoritmo de ordenação que opera sobre inteiros, processando dígitos individuais. (CINTRA, NOBRE, VIANA, 2015).
O algoritmo *BubbleSort*, de acordo com Silva (*apud* LIMA *et al*., 2017, p. 172), "apesar de ser o de mais fácil implementação, não apresenta resultados satisfatórios, principalmente no número de comparações". A ineficiência do algoritmo deve-se ao grande consumo de processamento, o que, para máquinas com poucos (ou limitados) recursos computacionais, resulta em lentidão e longos períodos de espera. Sua aplicação é, na opinião dos autores, indicada somente para fins educacionais. (SILVA *apud* LIMA; RICARTE; SOUZA, 2017).
O *SelectionSort* é em estruturas lineares, no entanto, com o número de elementos consideravelmente maior do que o *InsertSort*. O número de trocas é bastante inferior ao número de comparações, consumindo mais tempo de leitura e menos de escrita. A vantagem de seu uso ocorre quando se trata de componentes em que, quanto mais se escreve, ou reescreve, mais se desgasta e, consequentemente, perdem sua eficiência, como é o caso das memórias. (EEPROM; FLASH *apud* LIMA; RICARTE; SOUZA, 2017.).
Segundo Lima *et al*. (2017, p. 172), o *InsertionSort* é útil para estruturas lineares pequenas, geralmente entre 8 e 20 elementos. É amplamente utilizado em sequências de 10 elementos, tendo listas ordenadas de forma decrescente como pior caso, listas em ordem crescente como o melhor caso. Sua principal vantagem é o pequeno número de comparações. O excessivo número de trocas é a sua desvantagem.
Para concluir, o *InsertionSort* e o *SelectionSort*, de acordo com Lima *et al*. (2017, p. 172), são mais utilizados em associação com outros algoritmos de ordenação, como os *MergeSort*, *QuickSort* e o *ShellSort*, os quais tendem a subdividir as listas a serem organizadas em listas menores, fazendo com que sejam mais eficientemente utilizados.
Agora, convido você, caro aluno, a colocar em prática seus conhecimentos. Vamos lá?

VAMOS PRATICAR?

1. Como funciona o *MergeSort*?

Resolução

O Mergesort é um exemplo de algoritmo de ordenação que faz uso da estratégia “dividir para conquistar”. É um método estável e possui complexidade “C (n) = O (n log n) ”, para todos os casos. Esse algoritmo divide o problema em pedaços menores, de modo a resolver um pedaço de cada vez, juntando depois os resultados. O vetor é dividido em duas partes iguais, cada qual divididas em duas partes iguais novamente e assim por diante, até ficar um (ou dois) elementos, cuja ordenação é trivial. Para reunir as partes ordenadas, os dois elementos de cada parte são separados, e o menor deles é retirado de sua parte. Em seguida, os inferiores entre os restantes são comparados. Prossegue-se assim, até ordenar todas as partes.

2. É fácil de implementar o *ShellSort*?

Resolução

O método ShellSort é um exemplo de algoritmo fácil de descrever e implementar, mas difícil de analisar. É uma extensão do algoritmo de ordenação por inserção. Ele permite a troca de registros distantes um do outro, diferentemente do algoritmo de ordenação por inserção que possui a troca de itens adjacentes para determinar o ponto de inserção. É um método não estável.

3. Descreva o mecanismo de funcionamento do *QuickSort*?

Resolução

Basicamente, a ideia do método é dividir um conjunto com “n” itens. Ou seja, dividi-lo em “dois problemas menores”. Os problemas menores são ordenados independentemente, e os resultados são combinados para produzir a solução final. A operação do algoritmo divide sua lista de entrada em duas sub-listas, a partir de um pivô. Em seguida, o mesmo procedimento nas duas listas menores, até uma lista unitária é realizado.

Ao finalizar as atividades, você teve a oportunidade de fixar o conteúdo e fortalecer sua memória, além de consolidar hábitos de disciplina importantes para sua vida acadêmica. Parabéns!

Síntese

Ao longo dessa unidade, você aprendeu que classificar dados é o ato de colocar os dados em uma ordem particular e específica, crescente ou decrescente; que é uma das aplicações mais importantes da computação. Aprendeu, ainda, que independentemente da classificação do algoritmo utilizado para classificar o *array*, o resultado final será o mesmo; entretanto, será a escolha do algoritmo, bem como seu tempo de execução e uso de memória do programa, que farão a diferença.

Nesta unidade, você teve a oportunidade de:

- entender o conceito de ordenação que é a operação de rearranjar os dados disponíveis em uma determinada ordem. A eficiência no manuseio desses dados pode ser aumentada, se eles estiverem dispostos de acordo com algum critério de ordem;
- conhecer os métodos de classificação interna, que do ponto de vista da memória do computador, podem ser classificados em: ordenação interna e ordenação externa; estes, classificados em: métodos simples e métodos eficientes;
- compreender as principais diferenças entre os algoritmos de ordenação: 1) Bolha; 2) Inserção; 3) Seleção; 4) Shellsort e Mergesort; 5) Quicksort; 6) Heaps Binários; 7) Heapsort; 8) Countingsort; 9) Bucketsort e Radixsort.

Bibliografia

AGUILAR, L. J. **Fundamentos de programação**: algoritmos, estruturas de dados e objetos. 3. ed. Porto Alegre: AMGH, 2018.

CINTRA, G. F.; NOBRE, R. H.; VIANA, G. V. R. **Pesquisa e ordenação de dados.** 2. ed. Fortaleza: Editora UECE, 2015.

CINTRA, G. F.; VIANA, G. V. R. **Pesquisa e ordenação de dados**. Fortaleza: UECE, 2011

COELHO, H.; FÉLIX, N. **Métodos de ordenação 1**. 39 slides. s. d. Disponível em: <http://www.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf>. Acesso em: 14/09/2019.

COSTA, R. O.; Xavier, R. C. M. Relações mútuas entre informação e conhecimento: o mesmo conceito? **Ciência da Informação**, Brasília, DF, v. 39, n. 2, p.75-83, maio/ago. 2010. Disponível em: <http://www.scielo.br/pdf/ci/v39n2/06.pdf>. Acesso em: 14/09/2019.

DEITEL, P.; DEITEL, H. **Java como programar**. 8. ed. São Paulo: Pearson, 2015.

FEOFILOFF, P. **Heapsort**. IME. Instituto de Matemática e Estatística. USP. Universidade de São Paulo, 2018. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>. Acesso em: 14/09/2019.

HUXLEY, A.L. Admirável Mundo Novo. São Paulo: Abril Cultural, 1982.

INSERT-SORT WITH ROMANIAN FOLK DANCE. Criado por: Sapientia University, Tirgu Mures (Marosvásárhely), Romania. Dirigido por: Kátai Zoltán and Tóth László. 4:03 min. Disponível em: <https://www.youtube.com/watch?time_continue=35&v=ROalU379l3U>. Acesso em: 14/09/2019.

LE COADIC, Y. F. **A Ciência da Informação**. Brasília: Briquet de Lemos, 1994.

LIMA, N. C. A.; RICARTE, J. V. G.; SOUZA, J. É. G. Algoritmos de ordenação: um estudo comparativo. **Anais do Encontro de Computação do Oeste Potiguar**. Pau dos Ferros/RN, v. 1, p. 166-173, jun. 2017. ECOP Encontro de Computação do Oeste Potiguar/UFERSA. Universidade Federal Rural do Semiárido, 2017. Disponível em: < https://periodicos.ufersa.edu.br/index.php/ecop>. Acesso em: 14/-9/2019.

MENOTTI, D. **Algoritmos e estrutura de dados**. Departamento de Informática, Universidade Federal do Paraná, s d. 26 slides. Disponível em: <https://web.inf.ufpr.br/menotti/ci056-2015-2-1/slides/aulaORDSimples.pdf>. Acesso em: 14/09/2019.

ROCHA, C. E. A. L. **Desenvolvimento de sistema de informação para apoio à gestão de projetos em sintonia com o marco legal da ciência, tecnologia e inovação**. (Tese). Paraná: Programa de Pós-Graduação em Engenharia Elétrica e Informática. Universidade Federal do Paraná, 2018.