**Chapter XVIII**

**Algorithms**

**Chapter XVIII Topics**

18.1 Introduction

18.2 The List Case Study

18.3 Improving Input and Output

18.4 The Linear Search

18.5 The Bubble Sort

18.6 The Selection Sort

18.7 The Insertion Sort

18.8 The Binary Search

18.9 The Merge Sort

18.10 Testing Algorithm Efficiency with *TimeTest*

18.11 Informal Algorithmic Analysis

18.12 Exact Calculation of Execution Counts

18.13 Summary

**18.1 Introduction**

We need to take a Java break with this chapter. There will be no significant new Java language features introduced. Do keep remembering that you are learning introductory computer science concepts and the language Java is used to teach these concepts. At most high schools BASIC, Pascal, C and C++ were used in computer science before switching to Java. In college you will see a bigger variety in introductory computer science classes. Java is certainly popular, but so are Scheme, Visual BASIC, C++ and other languages, and the choice of the introductory computer science programming language changes quite frequently.

This chapter, however, focuses on an area that does not have many changes: the algorithms used in computer programming. The algorithms that are presented in this chapter are essentially unchanged from earlier chapters in BASIC, Pascal and C++ books. Sure the language syntax is different, but the essence of the algorithm is unchanged. Do you remember what an algorithm is?

|  |
| --- |
| **Algorithm Definition** |
| An algorithm is a step-by-step solution to a problem. |

|  |
| --- |
| **Niklaus Wirth’s Programming Language Definition** |
| Niklaus Wirth, the creator of the programming language Pascal, made the following equation about data structures and algorithms.  *Data Structures + Algorithms = Programs* |

You have done quite a variety of program assignments at this stage and each program required the creation of an algorithm. There were times when you repeated the same types of algorithms for different assignments. The aim of this chapter is to look at a group of practical algorithms that are commonly used in computer science. In particular we want to look at computer program algorithms that are used to process array information.

This chapter also emphasizes the recurring theme of computer science, not to *reinvent the wheel*. If practical algorithms have been created for certain situations, use them, store them, and reuse them as some later date. Time is too precious to start from scratch with every program when useful tools have already been created. By the end of this chapter I hope that your programming toolbox will be considerably expanded.

**18.2 The List Case Study**

Back in Chapter XII, you saw that occasionally material is presented in the form of a *Case Study*. There was one section on the *Jack O’Lantern Case Study* and another section on the *Train Case Study*. These single section cases studies are what I would call *small* *case studies*. By comparison, the *GridWorld Case Study*, would be a *big case study* which encompasses many chapters.

We will now start the *List Case Study* which I would classify as *medium size*. The point is to show that Java static arrays can be a member of an object. Essentially, arrays can be used for *composition*. This very concept was introduced right after the case studies in Chapter XII. The creation of a **List** class allows storage of array elements along with the actions or *methods* that process the array. This OOP approach creates a neatly *encapsulated* package for list processing.

Well we now have arrived at our official *Algorithm* chapter. In this chapter you will learn some of the common algorithms used in computer science. These algorithms are independent of a programming language. The syntax implementation in sample programs may be specific to Java, but the algorithmic considerations carry across all program languages.

The **List** class will grow with each new algorithm. Program **Java1801.java**, shown in figure 18.1, it the first stage of *List Case Study*. The **main** method of the program tests the **List** class. Each stage of the case study has a number. The **List** class has four methods right now. There are two *constructors*, an **assign** method and a **display** method.

The first constructor has one parameter to indicate the number of array elements. The second constructor has two parameters. The first parameter is for the array size, the second parameter indicates the initial value for each array element.

The **assign** method assigns new values to each array element, using random values in the **[1000..9999]** range. The **display** method shows the value for each element of a **List** object, called **array1** and **array2** in this program.

This program and a few later programs will include the entire **List** class with the program. As the case study advances and grows with each additional method, only the **main** method and the latest addition of the **List** class will be shown by itself. Some methods, which are not used in future program examples, will be deleted from the **List** class. This approach will create greater program clarity and allows more focus on the newest methods.

## Figure 18.1

|  |
| --- |
| // Java1801.java  // List case study #1  // The first stage of the List case study.  import java.util.Random;    public class Java1801  {  public static void main(String args[])  {  List1 array1 = new List1(10);  array1.display();  List1 array2 = new List1(10,999);  array2.display();  array2.assign();  array2.display();  System.out.println();  }  }  class List1  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array    public List1(int s)  {  System.out.println("\nCONSTRUCTING NEW LIST OBJECT WITH DEFAULT VALUES");  size = s;  intArray = new int[size];  }  public List1(int s, int n)  {  System.out.println("\nCONSTRUCTING NEW LIST OBJECT WITH SPECIFIED VALUES");  size = s;  intArray = new int[size];  for (int k = 0; k < size; k++)  intArray[k] = n;  }  public void assign()  {  System.out.println("\nASSIGNING RANDOM VALUES TO LIST OBJECT");  Random rndInt = new Random(12345);  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(1000);  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }    } |

**Figure 18.1 Continued**

|  |
| --- |
| **Java1801.java Output**  CONSTRUCTING NEW LIST OBJECT WITH DEFAULT VALUES  DISPLAYING ARRAY ELEMENTS  0 0 0 0 0 0 0 0 0 0  CONSTRUCTING NEW LIST OBJECT WITH SPECIFIED VALUES  DISPLAYING ARRAY ELEMENTS  999 999 999 999 999 999 999 999 999 999  ASSIGNING RANDOM VALUES TO LIST OBJECT  DISPLAYING ARRAY ELEMENTS  5745 5123 7462 1959 4979 8216 3078 7518 3355 7268 |

**List Case Study #2, Adding a Third Constructor**

The **assign** method provides a set of random values for a **List** object. These values are always in the **[1000..9999]** range. Our next step is to create a third constructor that can control the kind of random values that will be assigned to a new **List** object.

Three parameters are necessary for this new constructor. The first parameter specifies the size of the new **List** object. The second parameter indicates the smallest possible random integer value. The third parameter passes the value of the largest upper bound value of the random integer range. Program **Java1802.java**, in figure 18.2, demonstrates the use of this new constructor. Four objects are instantiated with four different random integer ranges. The range of the integers is displayed by the constructor method.

## Figure 18.2

|  |
| --- |
| // Java1802.java  import java.util.Random;      public class Java1802  {  public static void main(String args[])  {  List2 array1 = new List2(15,0,100);  array1.display();  List2 array2 = new List2(15,100,999);  array2.display();  List2 array3 = new List2(15,0,1);  array3.display();  List2 array4 = new List2(15,500,505);  array4.display();  System.out.println();  }  }  class List2  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    **public List2(int s, int min, int max)**  **{**  **Random rndInt = new Random(12345);**  **minInt = min;**  **maxInt = max;**  **size = s;**  **System.out.println("\nCONSTRUCT LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");**  **intArray = new int[size];**  **int range = maxInt - minInt + 1;**  **for (int k = 0; k < size; k++)**  **intArray[k] = rndInt.nextInt(range) + minInt;**  **}**  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  } |

**Figure 18.2 Continued**

|  |
| --- |
| **Java1802.java Output**  CONSTRUCTING LIST WITH VALUES in [0..100] range  DISPLAYING ARRAY ELEMENTS  98 39 24 89 4 68 81 81 80 19 61 20 63 46 46  CONSTRUCTING LIST WITH VALUES in [100..999] range  DISPLAYING ARRAY ELEMENTS  594 854 659 335 731 643 985 651 462 423 774 887 519 710 648  CONSTRUCTING LIST WITH VALUES in [0..1] range  DISPLAYING ARRAY ELEMENTS  0 1 0 1 0 0 1 1 0 1 1 1 0 1 0  CONSTRUCTING LIST WITH VALUES in [500..505] range  DISPLAYING ARRAY ELEMENTS  501 504 504 503 503 500 500 501 501 501 504 503 501 502 503 |

**18.3 Improving Input and Output**

Before we go too much further with some new algorithms, it is time to make the **List** class more user-friendly in the *input/output* department. Output of any sizeable list will fly by on the monitor without allowing proper viewing of the intermediate values. Input right now does not exist. The size of the **List** objects and the range of the values are hard coded in the programs. This is not a satisfactory arrangement. The first issue of too much output is handled by program **Java1803.java**, in figure 18.3. This program adds a **pause** method to the class. We are not looking at a *breaking news* method, but a small, practical method, which stops program execution until the <Enter> key is pressed. The programmer can select where to call this method to control program output. In the case of the current program example, **pause** is called after the display of each list. The result is that the elements of each separate list can be viewed before displaying the next list. Please remember that only the **main** method is shown completely and the partial **List** class is shown with the currently introduced method, like **pause**.

## Figure 18.3

|  |
| --- |
| // Java1803.java  // List case study #3  // This program adds the <pause> method, which freezes output display until  // the <Enter> key is pressed. This new method allows output viewing on the  // monitor when the display becomes too large.  import java.util.\*;      public class Java1803  {  public static void main(String args[])  {  List3 array1 = new List3(60,100,200);  array1.display();  array1.pause();  List3 array2 = new List3(100,100,999);  array2.display();  array2.pause();  List3 array3 = new List3(200,10,19);  array3.display();  array3.pause();  List3 array4 = new List3(40,500,505);  array4.display();  array4.pause();  System.out.println();  }  }  class List3  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List3(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }    **public void pause()**  **{**  **Scanner input = new Scanner(System.in);**  **String dummy;**  **System.out.print("\nPress <Enter> to continue ===>> ");**  **dummy = input.nextLine();**  **}**  } |

**Figure 18.3 Continued**

|  |
| --- |
| **Java1803.java Output**  CONSTRUCTING LIST WITH VALUES in [100..200] range  DISPLAYING ARRAY ELEMENTS  132 100 158 192 165 105 141 110 151 102 179 191 111 184 138 121  172 191 133 133 100 195 190 100 193 158 134 132 116 141 124 118  160 146 128 133 111 101 188 162 136 129 144 193 109 156 126 113  110 116 112 116 169 122 170 146 131 191 146 149  Press <Enter> to continue ===>>  CONSTRUCTING LIST WITH VALUES in [100..999] range  DISPLAYING ARRAY ELEMENTS  844 515 920 963 593 216 910 865 377 950 725 391 425 751 669 564  313 370 157 604 226 639 636 934 988 719 582 900 690 938 823 398  612 577 587 430 376 327 197 262 599 902 204 427 373 165 738 559  919 389 418 299 376 955 958 807 440 268 347 731 352 239 340 920  361 792 171 158 589 404 344 464 942 712 961 895 364 894 464 943  877 428 921 942 191 842 655 750 771 761 349 793 206 567 947 530  725 296 884 818  Press <Enter> to continue ===>>  CONSTRUCTING LIST WITH VALUES in [10..19] range  DISPLAYING ARRAY ELEMENTS  10 14 10 19 11 17 19 10 19 18 19 10 16 15 16 16 17 11 19 16  13 14 18 15 19 19 13 15 18 18 17 15 14 17 11 16 15 12 15 10  15 15 16 15 12 14 18 18 11 17 11 14 10 19 11 13 10 19 10 15  19 17 12 10 14 14 17 14 16 14 15 11 12 11 11 11 11 16 15 17  11 16 15 12 19 13 19 10 17 19 13 18 13 17 12 12 10 15 11 18  16 19 11 15 12 15 19 13 12 11 13 15 17 15 11 12 16 14 10 10  16 11 13 14 10 12 19 14 11 13 13 19 11 12 19 13 13 19 18 13  15 16 19 11 13 10 11 17 19 17 18 18 17 11 12 13 19 11 16 12  15 12 18 16 16 11 17 17 13 12 19 12 16 12 10 16 19 17 10 16  14 19 16 11 18 17 17 15 18 18 10 11 16 19 16 12 14 13 19 13  Press <Enter> to continue ===>>  CONSTRUCTING LIST WITH VALUES in [500..505] range  DISPLAYING ARRAY ELEMENTS  504 505 500 505 505 500 500 503 505 503 504 504 502 502 502 505  505 500 501 501 502 502 500 501 500 500 505 504 503 505 502 504  501 500 503 501 500 500 501 501  Press <Enter> to continue ===>> |

**List Case Study #4, Keyboard Input**

This stage is quite significant. The program user now can take control of the program execution process. The three **List** constructor parameters of **listSize**, **listMin** and **listMax** will be prompted and entered at the keyboard. The input process will be handled by the **main** method and the input values will then be used to construct the requested **List** objects. Program **Java1804.java**, in figure 18.4, executes the program and then prompts for user input.

## Figure 18.4

|  |
| --- |
| // Java1804.java  // List case study #4  // This program allows all list information to be entered at the keyboard  // before a list object is constructed.  import java.util.\*;    public class Java1804  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List4 array = new List4(listSize,listMin,listMax);  array.display();  array.pause();  System.out.println();  }  }  class List4  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List4(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  } |

**Figure 18.4 Continued**

|  |
| --- |
| **Java1804.java Output**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>> |

**18.4 The Linear Search**

Now let us look at what happens in the real world. In a moment of weakness your mother or father gives you a credit card to do shopping at the local mall. You are excited and tightly clutch the piece of plastic in your hands. At each one of the stores you hand the credit card for payment. Is your new purchase now yours? No it is not. First you have to wait while the store clerk determines if your credit card can handle the purchase. Is the card valid? Does the card have sufficient credit left in its balance? These questions can only be answered by finding the information record associated with your parents’ credit card. In other words, a computer needs to perform a search to find the proper credit card record.

The credit card is but one example. In an auto parts store, a clerk punches in some part number and the computer searches to see if the part exists in inventory. And now in the modern Internet world searching takes on a new meaning when special programs, called *search engines*, look for requested topics on the ever- expanding World Wide Web. In other words, searching is a major big deal and in this chapter we start to explore various ways that you can search for requested elements in an array.

There is not a simpler search than the *Inefficient Linear Search*. In this search you start at the beginning of a list and traverse to the end, comparing every element along the way. A Boolean variable is set to true if a match is found. The **linearSearch** method is shown first, by itself in figure 18.5. Then program **Java1805.java**, in figure 18.6, demonstrates this first searching approach. In just a moment I will explain why this search is called inefficient. This algorithm is called **linear** or sometimes **sequential** because it starts at one end and goes in a linear or sequential progression to the end. There are two program outputs, one for an existing number and a second output for a non-existing number.

**Figure 18.5**

|  |
| --- |
| **public boolean linearSearch(int sn)**  **{**  **boolean found = false;**  **for (int k = 0; k < size; k++)**  **if (intArray[k] == sn)**  **found = true;**  **return found;**  **}** |

## Figure 18.6

|  |
| --- |
| // Java1805.java  // List case study #5  // This program introduces the "inefficient" Linear Search algorithm.  import java.util.\*;  public class Java1805  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();      List5 array = new List5(listSize,listMin,listMax);  array.display();  array.pause();  System.out.print("\nEnter search number ===>> ");  int searchNumber = input.nextInt();  if (array.linearSearch(searchNumber))  System.out.println(searchNumber + " is in the list");  else  System.out.println(searchNumber + " is not in the list");  System.out.println();  }  }  class List5  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List5(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  **public boolean linearSearch(int sn)**  **{**  **boolean found = false;**  **for (int k = 0; k < size; k++)**  **if (intArray[k] == sn)**  **found = true;**  **return found;**  **}**  } |

## Figure 18.6 Continued

|  |
| --- |
| **Java1805.java Output #1**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>>  Enter search number ===>> 4141  4141 is in the list  Press any key to continue... |

|  |
| --- |
| **Java1805.java Output #2**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>>  Enter search number ===>> 5000  5000 is not in the list  Press any key to continue... |

Did you understand why this *Linear Search* algorithm is called inefficient? What happens when the search number is the very first number in the list? The loop still continues and compares every element in the array until the end. This is flat silly. Imagine that you are at a car repair shop and after your file is found, the clerk continues to check every file. The next search algorithm is a more civilized *Linear Search*and uses a Boolean variable both to signal that the requested element is found as well as a loop condition to terminate the search. Figure 18.7 compares the previous *Linear Search* with the improved algorithm and program.

**Figure 18.7**

|  |  |
| --- | --- |
| **Inefficient Linear Search** | **Efficient Linear Search** |
| **public boolean linearSearch(int SN)**  **{**  **boolean found = false;**  **for (int k = 0; k < Size; k++)**  **if (intArray[k] == sn)**  **found = true;**  **return found;**  **}** | **public boolean linearSearch(int sn)**  **{**  **boolean found = false;**  **int k = 0;**  **while (k < size && !found)**  **{**  **if (intArray[k] == sn)**  **found = true;**  **else**  **k++;**  **}**  **return found;**  **}** |

**Java1806.java**, in figure 18.8, demonstrates the altered algorithm. Do not expect to see any difference in execution speed. With today's computers, you will need to search a rather substantial list size to make any measurable comparisons.

## Figure 18.8

|  |
| --- |
| // Java1806.java  // List case study #6  // The inefficient linear search is replaced with a conditional loop, which stops  // the repetition once the searchNumber is found.  import java.util.\*;    public class Java1806  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();    List6 array = new List6(listSize,listMin,listMax);  array.display();  array.pause();  System.out.print("\nEnter search number ===>> ");  int searchNumber = input.nextInt();  if (array.linearSearch(searchNumber))  System.out.println(searchNumber + " is in the list");  else  System.out.println(searchNumber + " is not in the list");  System.out.println();  }  }  class List6  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List6(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  **public boolean linearSearch(int sn)**  **{**  **boolean found = false;**  **int k = 0;**  **while (k < size && !found)**  **{**  **if (intArray[k] == sn)**  **found = true;**  **else**  **k++;**  **}**  **return found;**  **}**  } |

## Figure 18.8 Continued

|  |
| --- |
| **Java1806.java Output**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>>  Enter search number ===>> 4141  4141 is in the list  Press any key to continue... |

You might be satisfied with the improved *Linear Search*, but I am not. The actual Java implementation of the search is really not very practical. You are told that some requested number exists or it does not exist. In real life searches normally provide information about the location of a successful search. Imagine a very large warehouse with stocked inventory. Your trusty computer has told you that *Inventory part #548321-A* is stocked. Well you are so pleased since there are only 200,000 parts stored and finding *548321-A* should be a breeze. The next stage, program **Java1807.java**, in figure 18.9, of the case study uses the efficient *Linear Search* algorithm and returns the index value of the array element if the item is found. An index value of **-1** is returned if the search item is not in the list.

## Figure 18.9

|  |
| --- |
| // Java1807.java  // List case study #7  // This program makes the Linear Search algorithm more practical  // by returning the index of the SearchNumber or -1 if not found.  import java.util.\*;  public class Java1807  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();    List7 array = new List7(listSize,listMin,listMax);  array.display();  array.pause();  System.out.print("\nEnter search number ===>> ");  int searchNumber = input.nextInt();  int index = array.linearSearch(searchNumber);  if (index == -1)  System.out.println(searchNumber + " is not in the list");  else  System.out.println(searchNumber + " is found at index " + index);  System.out.println();  }  }  class List7  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List7(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  **public int linearSearch(int sn)**  **{**  **boolean found = false;**  **int k = 0;**  **while (k < size && !found)**  **{**  **if (intArray[k] == sn)**  **found = true;**  **else**  **k++;**  **}**  **if (found)**  **return k;**  **else**  **return -1;**  **}**  } |

**Figure 18.9 Continued**

|  |
| --- |
| **Java1807.java Output #1**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>>  Enter search number ===>> 4141  4141 is found at index 95 |

|  |
| --- |
| **Java1807.java Output #2**  Enter list size ===>> 100  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175 3801 5351 2792 7316 7428 6781 1425 8943 2871 3439 4729 8397  7501 5825 9903 3555 8952 1831 7010 5108 1396 5582 7099 7758 9318 4580 3412 1691  4350 8728 4774 3644 1054 1508 7380 5599 1158 3655 2594 2497 4294 4087 7553 8318  2583 1499 6370 6773 4990 2519 8076 8121 2918 6056 3188 3826 7849 3992 9552 6875  1003 5900 9832 3364 8275 1544 3409 5055 7398 9116 8687 1439 6953 3367 9551 4141  7715 5466 1174 6677  Press <Enter> to continue ===>>  Enter search number ===>> 5000  5000 is not in the list |

**18.5 The Bubble Sort**

We will return to more searching in a later section. Right now we need to leave searching behind to learn some other algorithms. Any additional improvements on searching algorithms will require that the data are sorted. I believe that nobody is really interested in sorting. Do not get me wrong, sorting is extremely important, but only because we desire searching. File cabinets have files neatly alphabetized or organized according to account numbers or some other order. These files are organized according to a sorting scheme for the purpose of finding the files easily. It is the same with library books that are organized in categories and sorted according to some special library number.

|  |
| --- |
| **Why Do We Sort?** |
| Sorting does not exist in a vacuum.  The reason for sorting is to allow more efficient searching. |

The first sort in this chapter is the *Bubble Sort*. This humble sort gets a lot of bad press. The poor *Bubble Sort* is banned from many textbooks and not allowed to be uttered in certain computer science environments. Why? It is by far the most inefficient sort in a large arsenal of sorts. However, it also happens to be the easiest sort to explain to students first introduced to sorting algorithms.

There is a secondary motivation. This chapter does not have as its primary motivation to study algorithmic efficiency. You are only getting started. This does not prevent a gentle introduction, and a small taste, of certain efficiency considerations. Starting with an inefficient algorithm like the *Bubble Sort* helps to point out certain efficiency problems and how they can be solved.

The *Bubble Sort* gets its name because data *bubbles* to the top, one item at a time. Consider the following small array of five numbers, shown in figure 18.10. It will be used to demonstrate the logic of the Bubble Sort step-by-step. At every stage, adjacent numbers are compared, and if two adjacent numbers are not in the correct place, they are swapped. Each pass through the number list places the largest number at the top. It has *bubbled* to the surface. The illustrations show how numbers will be sorted from smallest to largest. The smallest number will end up in the left-most array location, and the largest number will end up in the right-most location.

**Figure 18.10**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 45 | 32 | 28 | 57 | 38 |

45 is greater than 32; the two numbers need to be swapped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 32 | 28 | 45 | 57 | 38 |

45 is greater than 28; the two numbers need to be swapped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 32 | 28 | 45 | 57 | 38 |

45 is not greater than 57; the numbers are left alone.

57 is greater than 38; the two numbers need to be swapped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 32 | 28 | 45 | 38 | **57** |

One pass is now complete. The largest number, **57**, is in the correct place.

A second pass will start from the beginning with the same logic.

32 is greater than 28; the two numbers need to be swapped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | 45 | 38 | **57** |

32 is not greater than 45; the numbers are left alone.

45 is greater than 38; the two numbers need to be swapped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | 38 | **45** | **57** |

We can see that the list is now sorted. Our current algorithm does not realize this.

It is not necessary to compare 45 and 57.

The second pass is complete, and **45** is in the correct place.

The third pass will start.

28 is not greater than 32; the numbers are left alone.

32 is not greater than 38; the numbers are left alone.

**Figure 18.10 Continued**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | **38** | **45** | **57** |

The third pass is complete, and **38** is “known” to be in the correct place.

The fourth - and final - pass will start.

28 is not greater than 32; the numbers are left alone.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | **32** | **38** | **45** | **57** |

The fourth pass is complete, and **32** is “known” to be in the correct place.

A fifth pass is not necessary. 28 is the only number left.

With **5** numbers there will be **4** comparison passes.

With **N** numbers there will be **N-1** comparison passes.

|  |
| --- |
| **Bubble Sort Logic** |
| Compare adjacent array elements.  Swap the elements if they are not ordered correctly.  Continue this process until the largest element is in  the last element of the array.  Repeat the comparison process in the same manner.  During the second pass make one less comparison,  and place the second-largest number in the second-to-last  element of the array.  Repeat these comparison passes with **N** elements,  **N-1** times. Each pass makes one less comparison. |

You are armed with the logic of a *Bubble Sort* routine. Now how do you translate such an algorithm into Java program code? How about in gentle stages? Program **Java1808.java**, in figure 18.11, adds the **partialSort** method. It is an incomplete *Bubble Sort* that only places one element, the largest, in the correct location. If we can understand this step, then it will be easier to see how the entire sort works.

## Figure 18.11

|  |
| --- |
| // Java1808.java  // List case study #8  // This program introduces a "partial-sort" algorithm. Only the largest number is places at the end of the list.  import java.util.\*;    public class Java1808  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List8 array = new List8(listSize,listMin,listMax);  array.display();  array.pause();  array.partialSort();  array.display();  array.pause();  System.out.println();  }  }  class List8  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List8(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }    **public void partialSort()**  **{**  **int temp;**  **for (int q = 0; q < size-1; q++)**  **if (intArray[q] > intArray[q+1])**  **{**  **temp = intArray[q];**  **intArray[q] = intArray[q+1];**  **intArray[q+1] = temp;**  **}**  **}**  } |

**Figure 18.11 Continued**

|  |
| --- |
| **Java1808.java Output**  Enter list size ===>> 20  Enter minimum value ===>> 1000  Enter maximum value ===>> 9999  CONSTRUCTING LIST WITH VALUES in [1000..9999] range  DISPLAYING ARRAY ELEMENTS  6251 6080 9241 1828 4055 2084 2375 9802 2501 5389 2517 1942 5390 3806 3012 2384  8787 5303 8532 6175  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  6080 6251 1828 4055 2084 2375 9241 2501 5389 2517 1942 5390 3806 3012 2384 8787  5303 8532 6175 **9802**  Press <Enter> to continue ===>> |

The **partialSort** method shows the required code for placing one array element in the correct place. It represent one so-called *comparison pass*. How many of these passes need to be made? You are correct. There are **n-1** comparison passes, with **n** equal to the number of elements in the list. If we take the previous **PartialSort** method and package that inside another loop structure, which executes **n-1** times, we might just be in business. Program **Java1809.java**, in figure 7.12, demonstrates the completed **bubbleSort**method.

## Figure 18.12

|  |
| --- |
| // Java1809.java  // List case study #9  // This program sorts in ascending order using the BubbleSort.  import java.util.\*;    public class Java1809  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();    List9 array = new List9(listSize,listMin,listMax);  array.display();  array.pause();  array.bubbleSort();  array.display();  array.pause();  System.out.println();  }  }  class List9  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List9(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }    **public void bubbleSort()**  **{**  **int temp;**  **for (int p = 1; p < size; p++)**  **for (int q = 0; q < size-1; q++)**  **if (intArray[q] > intArray[q+1])**  **{**  **temp = intArray[q];**  **intArray[q] = intArray[q+1];**  **intArray [q+1] = temp;**  **}**  **}**  } |

**Figure 18.12 Continued**

|  |
| --- |
| **Java1809.java Output**  Enter list size ===>> 100  Enter minimum value ===>> 10000  Enter maximum value ===>> 99999  CONSTRUCTING LIST WITH VALUES in [10000..99999] range  DISPLAYING ARRAY ELEMENTS  96251 69080 18241 28828 40055 92084 11375 72802 83501 23389  29517 82942 59390 30806 93012 20384 44787 68303 44532 15175  39801 14351 92792 43316 88428 78781 55425 35943 83871 57439  49729 17397 16501 50825 90903 66555 35952 55831 43010 68108  37396 32582 34099 88758 45318 67580 21412 64691 85350 80728  49774 21644 46054 37508 34380 32599 19158 57655 83594 38497  67294 76087 16553 17318 20583 73499 42370 51773 13990 11519  89076 71121 92918 69056 30188 84826 52849 39992 63552 60875  19003 68900 81832 21364 53275 10544 84409 32055 43398 72116  44687 28439 96953 93367 54551 22141 25715 50466 73174 78677  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  10544 11375 11519 13990 14351 15175 16501 16553 17318 17397  18241 19003 19158 20384 20583 21364 21412 21644 22141 23389  25715 28439 28828 29517 30188 30806 32055 32582 32599 34099  34380 35943 35952 37396 37508 38497 39801 39992 40055 42370  43010 43316 43398 44532 44687 44787 45318 46054 49729 49774  50466 50825 51773 52849 53275 54551 55425 55831 57439 57655  59390 60875 63552 64691 66555 67294 67580 68108 68303 68900  69056 69080 71121 72116 72802 73174 73499 76087 78677 78781  80728 81832 82942 83501 83594 83871 84409 84826 85350 88428  88758 89076 90903 92084 92792 92918 93012 93367 96251 96953  Press <Enter> to continue ===>> |

At the start of the *Bubble Sort* section I mentioned that it is good to start with an inefficient sort. This helps to appreciate efficiency and *thinking about efficiency* as improvements are made. So can you think of any improvements? I bet there are some clever students who realize that each comparison pass makes the same number of comparisons. Now, that is not very clever, because after each pass there is another number in the correct location. Some mechanism needs to be used to reduce comparisons for each pass.

There is also the issue of readability. Readability is an odd type of efficiency. When a program is made more readable it does not execute faster, but it debugs and updates faster. The ability to fix or update a program is a different, but also very important efficiency consideration.

Our **bubbleSort** method includes some lines of code that swap the adjacent list items. We can gain readability by creating a **swap** method. Furthermore, we should make this method **private**, because it is only used within the **List** class by the member **bubbleSort** method. Program **Java1810.ava**, in figure 18.13, addresses both improvements. Do you understand why the comparison passes are reduced with each iteration?

## Figure 18.13

|  |
| --- |
| // Java1810.java  // List case study #10  // This program introduces the private <swap> method that is used by the  // <bubbleSort> and other methods. It also improves the bubbleSort by  // reducing the number of comparison made on each pass.  import java.util.\*;    public class Java1810  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List10 array = new List10(listSize,listMin,listMax);  array.display();  array.pause();  array.bubbleSort();  array.display();  array.pause();  System.out.println();  }  }  class List10  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List10(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  **private void swap(int x, int y)**  **{**  **int temp = intArray[x];**  **intArray[x] = intArray[y];**  **intArray[y] = temp;**  **}**  public void bubbleSort()  {  for (int p = 1; p < size; p++)  for (int q = 0; q < size-p; q++)  if (intArray[q] > intArray[q+1])  **swap(q,q+1);**  }  } |

## Figure 18.13 Continued

|  |
| --- |
| **Java1810.java Output**  Enter list size ===>> 40  Enter minimum value ===>> 10  Enter maximum value ===>> 99  CONSTRUCTING LIST WITH VALUES in [10..99] range  DISPLAYING ARRAY ELEMENTS  41 50 61 28 95 14 35 82 71 79 87 52 80 26 42 44  57 83 72 55 21 41 92 26 48 31 75 33 81 19 49 27  31 65 93 45 42 31 80 68  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  14 19 21 26 26 27 28 31 31 31 33 35 41 41 42 42  44 45 48 49 50 52 55 57 61 65 68 71 72 75 79 80  80 81 82 83 87 92 93 95  Press <Enter> to continue ===>> |

**The Smart Bubble Sort**

Maybe you are bothered by a sort routine that keeps right on *sorting* when a list is already sorted. There has to be some clever way to identify if a list is sorted. Perhaps we have improved the **BubbleSort** method with several clever features, but as it stands our current algorithm is still clueless that the list is sorted before all the comparison passes are finished. Can this be improved? You bet and for starters the outer **for** loop has to go. The **for** loop is fixed and forces **N-1** passes in a *Dumb Bubble Sort*, regardless of the sort status of its elements. What is needed is a conditional loop. So the next question is how do you know if a list is sorted? Well consider this. How many swaps are made in a list that is already sorted? None, exactly! So here is the trick. Assume that a list is sorted and sets a Boolean variable, **Sorted**, to true. If a comparison determines that a swap needs to be made, **Sorted** becomes **false**. Continue this process until an entire pass is made without any calls to the swap method.

Program **Java1811.java**, in figure 18.14, proudly replaces the previous *dumb Bubble Sort* with the new *smart Bubble Sort*. You may be surprised that variable **P** (used previously for the outer loop) is used here as well. In the *dumb* *Bubble Sort* the outer loop control variable, **P**,is also used to decrease the number of comparisons of the inner loop with **Size-P**. Now that the outer loop is gone, there is still the need to decrease the number of comparisons. That is why **P** is included and incremented inside the conditional loop. Note that the variable **Sorted** is reset to **true** each time inside the outer loop.

## Figure 18.14

|  |
| --- |
| // Java1811.java  // List case study #11  // This program makes the BubbleSort "smart" by adding a conditional loop structure to see if the list is sorted.    import java.util.\*;    public class Java1811  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();    List11 array = new List11(listSize,listMin,listMax);  array.display();  array.pause();  array.bubbleSort();  array.display();  array.pause();  System.out.println();  }  }  class List11  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List11(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  private void swap(int x, int y)  {  int temp = intArray[x];  intArray[x] = intArray[y];  intArray[y] = temp;  }  **public void bubbleSort()**  **{**  **boolean sorted;**  **int p = 1;**  **do**  **{**  **sorted = true;**  **for (int q = 0; q < size-p; q++)**  **if (intArray[q] > intArray[q+1])**  **{**  **swap(q,q+1);**  **sorted = false;**  **}**  **p++;**  **}**  **while (!sorted);**  **}**  } |

## Figure 18.14 Continued

|  |
| --- |
| **Java1811.java Output**  Enter list size ===>> 40  Enter minimum value ===>> 100  Enter maximum value ===>> 200  CONSTRUCTING LIST WITH VALUES in [100..200] range  DISPLAYING ARRAY ELEMENTS  116 109 134 139 189 174 178 182 144 169 152 195 168 102 168 158  191 105 170 169 147 196 109 112 117 188 187 186 148 165 180 111  147 180 196 112 181 177 171 193  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  102 105 109 109 111 112 112 116 117 134 139 144 147 147 148 152  158 165 168 168 169 169 170 171 174 177 178 180 180 181 182 186  187 188 189 191 193 195 196 196  Press <Enter> to continue ===>> |

**18.6 The Selection Sort**

One improvement in our sorting introduction has been made. So is it possible to create any other improvements? Consider one part of the *Bubble Sort* that is very time consuming. Every time that a set of adjacent numbers is not in order, a swap needs to be made. Swapping results in executing three program statements. Three statements do not appear to be a big deal, but consider the following arithmetic in a list of 10,000 numbers. Consider the worst scenario where all 10,000 numbers in a list are in reverse order. This means that every number will need to be swapped in every location. You start with 9,999 swaps on the first pass and end up with 1 swap on the last pass. This will be an average of 5000 swaps for almost 10,000 passes for a total of 50,000,000 swaps. Since there are three statements in a swap routine that means that 150,000,000 program statements will need to be executed in this case. Even on a very efficient computer, the execution of 150,000,000 statements will take a considerable time penalty. This penalty becomes worse if the same algorithm is used frequently during the execution of a program. So what is the point? The point is that we can avoid all this excessive swapping business and improve execution time. Let us take another look at that list of five numbers, shown in figure 18.15, used to demonstrate the Bubble sort and follow a different set of rules to sort those numbers.

**Figure 18.15**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 45 | 32 | 28 | 57 | 38 |

We start by picking the first number, 45, as the smallest number.

Compare 45 with 32; 32 is smaller; make 32 the smallest number.

Compare 32 with 28; 28 is smaller; make 28 the smallest number.

Compare 28 with 57; 28 is smaller; keep 28 as the smallest number.

Compare 28 with 38; 28 is smaller; keep 28 as the smallest number.

Swap the smallest number, 28, with the first number, 45.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | 45 | 57 | 38 |

Repeat the comparison process for a second pass; start with the second number.

Pick 32 as the smallest number.

Compare 32 with 45; 32 is smaller; keep 32 as the smallest number.

Compare 32 with 57; 32 is smaller; keep 32 as the smallest number.

Compare 32 with 38; 32 is smaller; keep 32 as the smallest number.

No swapping is required on this pass. The smallest number is in the correct place.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | 45 | 57 | 38 |

Repeat the comparison process a third time; start with the third number.

Pick 45 as the smallest number.

Compare 45 with 57; 45 is smaller; keep 45 as the smallest number.

Compare 45 with 38; 38 is smaller; make 38 as the smallest number.

Swap the smallest number, 38, with the starting number, 45.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 28 | 32 | 38 | 57 | 45 |

Repeat the comparison process a fourth time; start with the fourth number.

Pick 57 as the smallest number.

Compare 57 with 45; 45 is smaller; make 45 the smallest number.

Swap the smallest number, 45, with the starting number, 57.

This is the fourth and final pass. The numbers are sorted.

|  |
| --- |
| **Selection Sort Logic** |
| Set the first number as the smallest number.  Compare the smallest number to each number in the list.  If any number is smaller, it becomes the smallest number.  After every number is compared, swap the smallest  number with the first number.  The smallest number is now in the correct location.  Repeat the comparison process in the same manner.  During the second pass, start with the second number  and make it the smallest number. At the conclusion  of the comparison pass swap the smallest number  with the second number.  Repeat these comparison passes with **N** elements,  **N-1** times. Each pass makes one less comparison. |

When you look at the source code of the **selectionSort** method added in program **Java1812.java**, shown by figure 18.16, you may find that variable, **smallest** does not store the smallest value. It stores the value of the index of the smallest number. This approach is simpler and takes less code. The logic of the sort is precisely as it was just described. Once again the outer loop variable, **p** is used to decrease the number of comparisons of the inner loop, as we did for the **bubbleSort** method.

## Figure 18.16

|  |
| --- |
| // Java1812.java  // List case study #12  // This stage replaces the Bubble Sort with the Selection Sort.  import java.util.\*;    public class Java1812  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List12 array = new List12(listSize,listMin,listMax);  array.display();  array.pause();  array.selectionSort();  array.display();  array.pause();  System.out.println();  }  }  class List12  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List12(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }    public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  private void swap(int x, int y)  {  int temp = intArray[x];  intArray[x] = intArray[y];  intArray[y] = temp;  }  **public void selectionSort()**  **{**  **int p,q;**  **int smallest;**  **for (p = 0; p < size-1; p++)**  **{**  **smallest = p;**  **for (q = p+1; q < size; q++)**  **if (intArray[q] < intArray[smallest])**  **smallest = q;**  **if (intArray[p] != intArray[smallest])**  **swap(p,smallest);**  **}**  **}**  } |

**Figure 18.16 Continued**

|  |
| --- |
| **Java1812.java Output**  Enter list size ===>> 80  Enter minimum value ===>> 10  Enter maximum value ===>> 99  CONSTRUCTING LIST WITH VALUES in [10..99] range  DISPLAYING ARRAY ELEMENTS  41 50 61 28 95 14 35 82 71 79 87 52 80 26 42 44 57 83 72 55  21 41 92 26 48 31 75 33 81 19 49 27 31 65 93 45 42 31 80 68  46 92 79 18 48 80 82 71 30 88 94 44 64 68 90 19 78 55 74 67  64 37 83 38 63 59 70 23 40 89 66 21 38 26 38 46 19 32 12 35  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  12 14 18 19 19 19 21 21 23 26 26 26 27 28 30 31 31 31 32 33  35 35 37 38 38 38 40 41 41 42 42 44 44 45 46 46 48 48 49 50  52 55 55 57 59 61 63 64 64 65 66 67 68 68 70 71 71 72 74 75  78 79 79 80 80 80 81 82 82 83 83 87 88 89 90 92 92 93 94 95  Press <Enter> to continue ===>> |

**18.7 The Insertion Sort**

You have a couple sorts now. First you learned the *Bubble Sort* with its variations and then you learned an improvement on the *Bubble Sort* with the *Selection Sort*. Now it is time to learn the *Insertion Sort*. The *Insertion Sort* is a considerable improvement depending on how it is used, but more on that later. Right now you need to learn the logic of the *Insertion Sort*. Consider the array of integers in figure 18.17. Now imagine that you have eight or more index cards with numbers on them. They need to be sorted in ascending order. How would you sort them? I suspect that you and many other people would take each index card and insert it into a growing pile of index cards that is sorted. Each index card is taken, compared to the stack that is already sorted and then inserted at the correct location.

**Figure 18.17**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |

Figure 18.18 starts the process by taking the first number and placing it in a very small, one-element, sorted array.

**Figure 18.18**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |
| 45 |

Now continue with number **83**. This number is larger than **45**. It will be inserted behind number **45**, as shown in figure 18.19.

**Figure 18.19**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |
| 45 | 83 |

The third number **19**, needs to be inserted at the very front. Figure 18.20 shows the slowly growing, sorted array.

**Figure 18.20**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |
| 19 | 45 | 83 |

Many of you will see a distinct pattern emerging. With each new number a comparison is made with the numbers that are already sorted. Figure 18.21 shows that the fourth number **98**, needs to be inserted at the very back.

**Figure 18.21**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |
| 19 | 45 | 83 | 98 |

This process is continued until you have the sorted array shown in figure 18.22.

**Figure 18.22**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 45 | 83 | 19 | 98 | 85 | 32 | 50 | 73 |
| 19 | 32 | 45 | 50 | 73 | 83 | 85 | 98 |

The program code for the *Insertion Sort* is very different from the *Bubble Sort* and the *Selection Sort*. There are no swapping-array-elements in sight. You do have to use three steps to achieve the insertion process.

|  |
| --- |
| **Insertion Sort Steps** |
| **1. Use a search routine to find the proper insertion**  **location.**  **2. Move all array elements, starting with the insertion**  **index, to the next array location.**  **3. Insert the new array element in the "empty" location.** |

Program **Java1813.java**, in figure 18.23, replaces the *Selection Sort* with the *Insertion Sort*. You will see three methods for this process. The **insertionSort** method provides a loop that visits each element of the random array. With each array element, method **linearSearch** is called to find the proper insertion index location. Once the index location is found, method **insertItem** follows by moving array elements to the next array location. This creates room to insert the new value at the proper index location.

**Figure 18.23**

|  |
| --- |
| // Java1813.java  // List case study #13  // This stage replaces the Selection Sort with the Insertion Sort.  import java.util.\*;    public class Java1813  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List13 array = new List13(listSize,listMin,listMax);  array.display();  array.pause();  array.insertionSort();  array.display();  array.pause();  System.out.println();  }  }  class List13  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List13(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;  }  public void display()  {  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(intArray[k] + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }    **private int linearSearch(int searchNumber, int numElements)**  **{**  **int index = 0;**  **while (index < numElements && searchNumber > intArray[index])**  **index++;**  **return index;**  **}**  **private void insertItem(int searchNumber, int numElements, int index)**  **{**  **for (int k = numElements-1; k > index; k--)**  **intArray[k] = intArray[k-1];**  **intArray[index] = searchNumber;**  **}**  **public void insertionSort()**  **{**  **for (int k = 0; k < size; k++)**  **{**  **int numElements = k + 1;**  **int index = linearSearch(intArray[k],numElements);**  **insertItem(intArray[k],numElements,index);**  **}**  **}**  } |

**Figure 18.23 Continued**

|  |
| --- |
| **Java1813.java Output**  Enter list size ===>> 100  Enter minimum value ===>> 100  Enter maximum value ===>> 999  CONSTRUCTING LIST WITH VALUES in [100..999] range  DISPLAYING ARRAY ELEMENTS  851 680 241 928 455 284 575 802 701 889 717 142 890 206 312 584  687 803 432 775 201 851 992 116 228 481 525 843 171 739 229 297  301 425 903 855 852 931 710 608 496 182 799 558 318 980 712 791  750 628 274 944 154 608 180 199 258 955 794 697 694 487 353 218  783 599 970 473 490 719 876 921 218 656 488 226 649 392 552 575  103 500 832 664 175 644 709 555 198 116 587 539 653 667 551 541  515 966 274 377  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  103 116 116 142 154 171 175 180 182 198 199 201 206 218 218 226  228 229 241 258 274 274 284 297 301 312 318 353 377 392 425 432  455 473 481 487 488 490 496 500 515 525 539 541 551 552 555 558  575 575 584 587 599 608 608 628 644 649 653 656 664 667 680 687  694 697 701 709 710 712 717 719 739 750 775 783 791 794 799 802  803 832 843 851 851 852 855 876 889 890 903 921 928 931 944 955  966 970 980 992  Press <Enter> to continue ===> |

**18.8 The Binary Search**

We are now ready to return to the all-important business of searching. It was mentioned that searching can benefit from sorting, which is why the whole sorting issue became a big deal in the first place. We now have sorted a variety of ways and this means you are anxious to see what you have gained with all this newly found knowledge.

Imagine a thick telephone book. It has 2000 pages. Do you perform a sequential search when you look for a phone number? I sure hope not. Such an approach would be horrendous. You approximate the location of the number and open the book. Now there are three possibilities. You have the correct page; the number is on an earlier page; the number is on a later page. You repeat the process of guessing until you find the correct page.

Now imagine that you have a really bad guessing ability. Who knows, maybe you are alphabetically challenged. All you know how to do is find mid point pages by splitting pages in two. So consider the following arithmetic.

Start with a 2000 page telephone book.

Split in two, and ignore 1000 pages and search in the remaining 1000 pages.

Split in two, and ignore 500 pages and search in the remaining 500 pages.

Split in two, and ignore 250 pages and search in the remaining 250 pages.

Split in two, and ignore 125 pages and search in the remaining 125 pages.

Split in two, and ignore 62 pages and search in the remaining 62 pages.

Split in two, and ignore 31 pages and search in the remaining 31 pages.

Split in two, and ignore 15 pages and search in the remaining 15 pages.

Split in two, and ignore 7 pages and search in the remaining 7 pages.

Split in two, and ignore 3 pages and search in the remaining 3 pages.

Split in two, and ignore 1 page and search in the remaining 1 page.

This splitting in half is telling us that even with a bad guessing techniques, and splitting each section in half, it will at most require looking at 11 pages for a book with 2000 pages. This is a worst case scenario. With a sequential search starting at page 1, it will take looking at 2000 pages in a worst case scenario. This is quite a difference, and this difference is the logic used by the *Binary Search*.

Let us apply this logic to a list of 12 numbers and see what happens when we search for a given element in figure 18.24.

**Figure 18.24**

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 |

Suppose we wish to find element **55**.

We take the first and last element index and divide by 2. (0 + 14)/2 = 7

We check list[7].

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 |

We see that list[7] = 45 and realize that the List[0]..List[7] can now be ignored.

We continue the search with List[8]..List[14].

We take the first and last element index and divide by 2. (8 + 14)/2 = 11

We check List[11].

We see that List[11] = 65 and realize that the List[11]..List[14] can be ignored.

We continue the search with List[8]..List[10].

We take the first and last element index and divide by 2. (8 + 10)/2 = 9

We check List[9].

We see that List[9] = 55. We have found the search item.

|  |
| --- |
| **Binary Search Logic** |
| The Binary Search only works with sorted lists.  Make the smallest index **small** and the largest index **large**.  Find the midpoint index with **(small + large) / 2**  Compare the midpoint value with the search item.  If the value is found you are done.  Otherwise re-assign **small** or **large**.  If the **searchItem** is greater you have a new **small**.  If the **searchItem** is lesser you have a new **large**.  Repeat the same process.  Continue the process until the **searchItem** is found or  **large** becomes less than **small**. |

## The *Binary Search* is more complicated than the *Linear Search*, but the extra work is worth it. A list of one million elements is quite large and even such a large list requires only at most 24 comparisons with a *Binary Search*. The *Linear Search* may take up to 1,000,000 comparisons in a worst case scenario. Program Java1814.java, in figure 18.25, replaces the linearSearch method with the binarySearch method.

## Figure 18.25

|  |
| --- |
| // Java1814.java  // List case study #14  // This program introduces the "Binary Search" which searches a sorted list  // far more efficiently than a "Linear Search" can.  import java.util.\*;  import java.text.DecimalFormat;    public class Java1814  {  public static void main(String args[])  {  Scanner input = new Scanner(System.in);  System.out.print("\nEnter list size ===>> ");  int listSize = input.nextInt();  System.out.print("Enter minimum value ===>> ");  int listMin = input.nextInt();  System.out.print("Enter maximum value ===>> ");  int listMax = input.nextInt();  List14 array = new List14(listSize,listMin,listMax);  array.display();  array.pause();  array.selectionSort();  array.display();  array.pause();  System.out.print("\nEnter search number ===>> ");  int searchNumber = input.nextInt();  int index = array.binarySearch(searchNumber);  if (index == -1)  System.out.println(searchNumber + " is not in the list");  else  System.out.println(searchNumber + " is found at index " + index);  }  }  class List14  {  private int intArray[]; // stores array elements  private int size; // number of elements in the array  private int minInt; // smallest random integer  private int maxInt; // largest random integer    public List14(int s, int min, int max)  {  Random rndInt = new Random(12345);  minInt = min;  maxInt = max;  size = s;  System.out.println("\nCONSTRUCTING LIST WITH VALUES in [" + minInt + ".." + maxInt + "] range");  intArray = new int[size];  int range = maxInt - minInt + 1;  for (int k = 0; k < size; k++)  intArray[k] = rndInt.nextInt(range) + minInt;    }  public void display()  {  DecimalFormat output = new DecimalFormat("0000");  System.out.println("\nDISPLAYING ARRAY ELEMENTS");  for (int k = 0; k < size; k++)  System.out.print(output.format(intArray[k]) + " ");  System.out.println();  }  public void pause()  {  Scanner input = new Scanner(System.in);  String dummy;  System.out.print("\nPress <Enter> to continue ===>> ");  dummy = input.nextLine();  }  private void swap(int x, int y)  {  int temp = intArray[x]; intArray[x] = intArray[y]; intArray[y] = temp;  }  public void selectionSort()  {  int p,q;  int smallest;  for (p = 0; p < size-1; p++)  {  smallest = p;  for (q = p+1; q < size; q++)  if (intArray[q] < intArray[smallest])  smallest = q;  if (intArray[p] != intArray[smallest])  swap(p,smallest);  }  }    **public int binarySearch(int sn)**  **{**  **boolean found = false;**  **int lo = 0;**  **int hi = size-1;**  **int mid = 0;**  **while (lo <= hi && !found)**  **{**  **mid = (lo + hi) / 2;**  **if (intArray[mid] == sn)**  **found = true;**  **else**  **{**  **if (sn > intArray[mid])**  **lo = mid + 1;**  **else**  **hi = mid - 1;**  **}**  **}**  **if (found)**  **return mid;**  **else**  **return -1;**  **}**    } |

**Figure 18.25 Continued**

|  |
| --- |
| **Java1814.java Output #1**  Enter list size ===>> 20  Enter minimum value ===>> 400  Enter maximum value ===>> 1600  CONSTRUCTING LIST WITH VALUES in [400..1600] range  DISPLAYING ARRAY ELEMENTS  1319 1021 1175 0554 1222 0722 1211 1086 0766 1006 0471 1029 0992 1176 0744 0898  0594 0640 0499 1538  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  0471 0499 0554 0594 0640 0722 0744 0766 0898 0992 1006 1021 1029 1086 1175 1176  1211 1222 1319 1538  Press <Enter> to continue ===>>  Enter search number ===>> 1086  1086 is found at index 13 |

**Figure 18.25 Continued**

|  |
| --- |
| **Java1814.java Output #2**  Enter list size ===>> 20  Enter minimum value ===>> 400  Enter maximum value ===>> 1600  CONSTRUCTING LIST WITH VALUES in [400..1600] range  DISPLAYING ARRAY ELEMENTS  1319 1021 1175 0554 1222 0722 1211 1086 0766 1006 0471 1029 0992 1176 0744 0898  0594 0640 0499 1538  Press <Enter> to continue ===>>  DISPLAYING ARRAY ELEMENTS  0471 0499 0554 0594 0640 0722 0744 0766 0898 0992 1006 1021 1029 1086 1175 1176  1211 1222 1319 1538  Press <Enter> to continue ===>>  Enter search number ===>> 500  500 is not in the list |

**18.9 The Merge Sort**

The *Bubble Sort*, *Selection Sort* and *Insertion Sort* have been presented in detail, meaning that each sort was presented logically and then followed by a working implementation program. You will now learn about one more sort. It is a sort that will only be presented in a logical sense. The presentation is language independent and there will be no program code to follow the sort explanation. This sort will be revisited by computer science majors during their second semester in college, where you will learn the details of the Java implementation.

All three of the earlier sorts are adequate sorts. They are adequate in the sense that any list of data that is reasonable in size will be sorted in a time period that is acceptable. What is a reasonable size? That is hard to tell. Several decades ago it took an early Apple computer 45 minutes to sort 1000 numbers using the BASIC computer language to implement a *Bubble Sort*. I believe that may well be the origin for the bad press that is given to the *Bubble Sort*. Today, you can sort thousands of elements with the *Bubble Sort* and not observe any time delay. However, be assured that when data lists grow, the first three sorts presented in this chapter will become very slow indeed.

There is a group of sorts, called the *Merge Sort*, *Heap Sort*, *Tree Sort* and *Quick Sort*, which have remarkable speed. At the end of this chapter you will have an opportunity to observe a comparison in speed to see how much of a difference a certain algorithm can make.

One of the intentions of the AP Computer Science course is to appreciate that algorithms have different execution behaviors. In this course, and in this chapter you will get an introduction to look at algorithmic behavior in an informal manner. The *Merge Sort* is used as an excellent example of an algorithm that performs far better than many other sorting algorithms. More will come on that later. Right now you need to understand how a *Merge Sort* managestosort data.

Imagine that you have two lists of data. Each list is sorted in ascending order. If it is necessary to combine the two lists into one larger, and sorted list, would you take advantage of the fact that the two smaller lists are already sorted? I am sure you would. With two sorted stacks of index cards, you can create one large, sorted stack by merging the two stacks.

Figure 18.26 shows two sorted lists. Imagine that you have to arrange these cards with their numbers into one sorted lists, as you see in figure 18.27. This would not be a very tough process. There is one very significant problem. Merging two lists in this manner only works if you have two smaller lists, which are already sorted. In most cases you will have one list of random data.

**Figure 18.26**

**List L1**

92

90

75

72

50

39

35

34

22

85

82

61

46

29

18

17

**List L2**

**Figure 18.27**

75

72

50

35

22

92

90

39

34

82

61

85

46

29

18

17

The drawing above shows **List1** in the higher position and **List2** in the lower position. Figure 18.28 shows the completed merging process.

**Figure 18.28**

34

75

72

35

22

92

90

39

50

82

61

85

46

29

18

17

The first introduction into the *Merge Sort* sounds great, but the reality is that you need to know how to use a sort with data that is totally random. You rarely find a situation where sets of sorted lists are handed to you for further sorting process. We will start with a list of eight, unsorted, numbers, shown in figure 18.29, and then proceed to manipulate these numbers in figures 18.30 through 18.35 by some logical fashion until the list is sorted. If we can discover a method for eight numbers there may be a good chance that it will work for larger arrays as well.

**Figure 18.29**

**500**

**751**

**809**

**179**

**342**

**678**

**143**

**456**

First we need to split the array into two parts and check to see if each half of the array can be merged into one larger array. We know that sorting is possible if we have access to two lists that are sorted already.

Figure 18.30shows how the array splits into two lovely halves. We have a merge process available, but correct merging requires that the lists to be merged are already sorted. All we have right now are two smaller arrays of four elements, and each smaller array is still unsorted. Merging at this stage will only rearrange the array in some useless, unsorted, fashion. How about splitting each one of the smaller arrays? Perhaps that will help.

**Figure 18.30**

**500**

**751**

**809**

**179**

**342**

**678**

**143**

**456**

Well this is just terrific. Figure 18.31has four smaller arrays, and surprise, each one of the arrays is as unsorted as when we started. We do not have one big problem now, we have four little problems. We are doing very little merging, but we are sure splitting very well. Hang on, and just for fun humor me and split one more time. Maybe, just maybe, something useful will happen.

**Figure 18.31**

**500**

**751**

**809**

**179**

**342**

**678**

**143**

**456**

Figure 18.32probably does not create tremendous excitement. We have now managed to split the original array so far that no more splitting can be done. This brings up an interesting question. Is a list with one element sorted? The question may seem peculiar, but the idea is very significant. Keep in mind that we have the capability of merging lists, provided they are sorted. We have not managed to do any merging, because we have not had any sorted lists to work with. A list with one element is very small, and it is also sorted. This mean that now we can merge, and let us do this with eight lists and merge them into four sorted lists.

**Figure 18.32**

**500**

**751**

**809**

**179**

**342**

**678**

**143**

**456**

Now we are getting somewhere. Four little merges have been performed, and each one of the merges created a small, but sorted array of two elements. We are now seeing something really useful because the four small lists in figure 18.33can be merged into two sorted lists of four elements.

**Figure 18.33**

**751**

**500**

**809**

**179**

**678**

**342**

**456**

**143**

Some serious celebration can start right about now. It appears that figure 18.34shows two sorted lists. We are only one merge process away from having the whole works sorted. What you see here is custom ordered for merging. We have two lists, and they are both sorted.

**Figure 18.34**

**809**

**751**

**500**

**179**

**678**

**456**

**342**

**143**

Success comes in figure 18.35. We have performed three merge passes and the whole list is sorted. Does this process seem faster than the types of data processing you saw with earlier sorts? Maybe you are not convinced, but you will be with some actual testing that will provide clear proof about the sorting speed of the sorts introduced in this chapter.

**Figure 18.35**

**809**

**751**

**678**

**500**

**456**

**342**

**179**

**143**

**18.10 Testing Algorithm Efficiency with**

**the TimeTest class**

Execution efficiency is concerned with the time required to process a certain amount of data. The remainder of this chapter will make use of a small, user-created, class called **TimeTest**. The functionality of the new class is best illustrated by first executing a small program. Program **Java1815.java**, in figure 18.36, asks the user to press the <Enter> key a first time, followed by pressing the <Enter> key a second time. After the second press the elapsed time between the two *presses* is displayed. The elapsed time is accurate to one/billionth or 1/1,000,000,000 of a second.

**Figure 18.36**

|  |
| --- |
| // Java1815.java  // This program uses a <TimeTest> class, which displays the elapsed time  // in hh:mm:ss.fractions for executing some program code.  import java.util.\*;  import java.text.DecimalFormat;  public class Java1815  {  public static void main(String args[])  {  System.out.println("\nJava1815.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Press <Enter> to start the clock ===>> ");  input.nextLine();  //////////  time.startClock();  System.out.println();  System.out.print("Press <Enter> to stop the clock ===>> ");  input.nextLine();  time.stopClock();  //////////    System.out.println();  System.out.println(time);  System.out.println();  }  }    class TimeTest  {    private long startNanos; // tick count at the start of the test  private long endNanos; // tick count at the end of the test  private long nanos; // elapsed number of nano seconds  private long hours; // elapsed hours  private long minutes; // elapsed minutes  private long seconds; // elapsed seconds  private long fractions; // elapsed fractions of a second    public TimeTest()  {  startNanos = 0;  endNanos = 0;  nanos = 0;  hours = 0;  minutes = 0;  seconds = 0;  fractions = 0;  }    public void startClock()  {  startNanos = System.nanoTime();  }  public void stopClock()  {  endNanos = System.nanoTime();  computeTime();  }    public void delay(long n)  {  n = n \* 1000000;  long startDelay = System.nanoTime();  long endDelay = 0;  while (endDelay - startDelay < n)  endDelay = System.nanoTime();  }    private void computeTime()  {  nanos = endNanos - startNanos;  hours = nanos / 3600000000000L;  long leftOver = nanos % 3600000000000L;  minutes = leftOver / 60000000000L;  leftOver = leftOver % 60000000000L;  seconds = leftOver / 1000000000L;  fractions = leftOver % 1000000000L;  }    public String toString()  {  DecimalFormat twos = new DecimalFormat("00");  DecimalFormat nines = new DecimalFormat("000000000");  String temp = twos.format(hours) + ":";  temp = temp + twos.format(minutes) + ":";  temp = temp + twos.format(seconds) + ".";  temp = temp + nines.format(fractions);  return temp;  }  } |

**Figure 18.36 Continued**

|  |
| --- |
| **Java1815.java Output**  Java1815.java  Press <Enter> to start the clock ===>>  Press <Enter> to stop the clock ===>>  00:00:06.198884725 |

The **TimeTest** class is not a standard Java class. I have created this class to simplify the testing of various algorithms. You start by instantiating an object of the **TimeTest** class, such as **time**. A call to **time.startClock()** accesses the computer's internal clock, which is ticking away every one billionth of a second. The number of ticks is stored and compared to a second access by a call to **time.stopClock()**. The difference between the ticks is then displayed. You are not really expected to know exactly how this **TimeTest** class works. All you need to understand is how an object of this class is capable of displaying elapsed time.

The **TimeTest** class offers a second feature, which is an accurate **delay** method. There are times when an intentional execution delay is desirable. One of the simplest types of delay algorithms is to create some loop structure and make the computer busily count to itself. This approach works, but such an algorithm creates different time delays on computers with different processing speeds. A superior **delay** method will delay the same length of time, regardless of the computer's processing speed. Program **Java1816.java**, in figure 18.37, demonstrates such a method. The parameter of the **delay** method indicates the number of milli-seconds that needs to be delayed. It is appropriate to add the **delay** method to the **TimeTest** class, because it uses the same **nanoTime** method, which accesses the computer's internal clock.

**Figure 18.37**

|  |
| --- |
| // Java1816.java  // This program demonstrates the <delay> method, which delays program execution  // an indicated number of milli seconds amount of time.  import java.util.\*;  import java.text.DecimalFormat;  public class Java1816  {  public static void main(String args[])  {  System.out.println("\nJava1816.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter time delay in milli seconds ===>> ");  int delayTime = input.nextInt();    //////////  time.startClock();  time.delay(delayTime);  time.stopClock();  //////////    System.out.println();  System.out.println(time);  System.out.println();  }  }    class TimeTest  {  private long startNanos; // tick count at the start of the test  private long endNanos; // tick count at the end of the test  private long nanos; // elapsed number of nano seconds  private long hours; // elapsed hours  private long minutes; // elapsed minutes  private long seconds; // elapsed seconds  private long fractions; // elapsed fractions of a second    public TimeTest()  {  startNanos = 0;  endNanos = 0;  nanos = 0;  hours = 0;  minutes = 0;  seconds = 0;  fractions = 0;  }    public void startClock()  {  startNanos = System.nanoTime();  }  public void stopClock()  {  endNanos = System.nanoTime();  computeTime();  }    **public void delay(long n)**  **{**  **n = n \* 1000000;**  **long startDelay = System.nanoTime();**  **long endDelay = 0;**  **while (endDelay - startDelay < n)**  **endDelay = System.nanoTime();**  **}**    private void computeTime()  {  nanos = endNanos - startNanos;  hours = nanos / 3600000000000L;  long leftOver = nanos % 3600000000000L;  minutes = leftOver / 60000000000L;  leftOver = leftOver % 60000000000L;  seconds = leftOver / 1000000000L;  fractions = leftOver % 1000000000L;  }    public String toString()  {  DecimalFormat twos = new DecimalFormat("00");  DecimalFormat nines = new DecimalFormat("000000000");  String temp = twos.format(hours) + ":";  temp = temp + twos.format(minutes) + ":";  temp = temp + twos.format(seconds) + ".";  temp = temp + nines.format(fractions);  return temp;  }  } |

**Figure 18.37 Continued**

|  |
| --- |
| **Java1816.java Output**  Java1816.java  Enter time delay in milli seconds ===>> 2345  00:00:02.345055308 |

When you run program **Java1816.java** you will notice that the time delay follows the requested delay very closely, but not precisely. This slight discrepancy is caused by the overhead time involved in making method calls. Nevertheless, the **delay** method is very practical and can be used whenever a delay is practical. This happens especially in the creation of a video game. The processing speed of computers is constantly changing. A game, such as *Tetris* written on a ten-year old computer without any delay method, will run entirely to fast on a current computer.

**18.11 Informal Algorithmic Analysis**

A class is a toolbox for computer programming. The methods in the class are the tools that process the appropriate materials or in the case of computer science the appropriate data. Plumbers, carpenters, electricians, roofers and many other building professionals not only need to know how to use tools, they also need to know which tool to use.

You think that a hammer is just a hammer? Think again. There are large, heavy hammers for carpentry. There is a totally different hammer for roofing. There are hammers for metal working, hammers for laying brick, hammers for attaching heels to shoes. There are many different types of hammers and the hammer designed for carpentry is a poor choice to attach a heel to a shoe.

In this chapter you have seen many different types of algorithms. An algorithm is a tool and may bring up the question about the difference between an algorithm and a method. Do you understand the difference?

|  |
| --- |
| **Algorithms and Methods** |
| An *algorithm* is a step-by-step sequence of instructions to complete a desired process. An *algorithm* can be written and explained in English. An *algorithm* is not connected to any programming language.  A *method* is one implementation of an *algorithm*.  A *method* is written in a programming language. |

Writing programs requires efficiency. Just like builders use tool kits with tools that are well established in the building trade, so do programmers use algorithms that are well known for certain programming requirements. Programs frequently require sorting data. An experienced programmer does not write a *Merge Sort* from scratch. Algorithms, like different types of sorts, will likely be stored in a library of classes that the programmer has found useful from previous programming projects. This means that it is not as important to know how to write a certain algorithm into a method. All well-known algorithms have already been written in every conceivable programming language. The real issue is that you know which method to use for which purpose.

Selecting the correct programming tool for a program requires knowledge of algorithms, which includes more than knowing that method **mergeSort** sorts data. You need to know the behavior of an algorithm. Algorithmic analysis is a sophisticated topic that is very mathematical in nature. The formal analysis of algorithms - meaning that you use a bunch of math stuff – is something computer science majors learn during their second semester in college. During this first, introductory course, you will only get a brief, informal approach to algorithmic analysis. Many observations can be made without doing precise, complicated mathematical analysis. People can conclude that a large boulder dropping from a cliff on their car will do more damage than the droppings of birds flying over their car. It is not necessary to perform the Newtonian physic formulas that will compute the mass of the boulder and the bird droppings to conclude that the boulder will do more damage. It a sort of *common sense type of analysis*. This is precisely what you will be doing in the next bunch of programs. You will observe the execution behavior of a variety of algorithm with the help of the **TimeTest** class and then draw various conclusions.

Program **Java1817.java**, in figure 18.38, starts the analysis process. This program measures the length of time that it takes to count a set of consecutive integers. With each program example, you need to first look at the program execution and then an explanation will be given after you have done some observations.

**Figure 18.38**

|  |
| --- |
| // Java1817.java  // This program measures the time necessary to count integers.  import java.util.\*;  public class Java1817  {  public static void main(String args[])  {  System.out.println("\nJava1817.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter an integer ===>> ");  int count = input.nextInt();  //////////  time.startClock();  counter(count);  time.stopClock();  //////////  System.out.println();  System.out.println(time);  }  public static void counter (int n)  {  for (int k = 1; k <= n; k++);    }  } |

|  |
| --- |
| **Java1817.java Output #1**  Java1817.java  Enter an integer ===>> 1000000  00:00:00.001898894 |
| **Java1817.java Output #2**  Java1817.java  Enter an integer ===>> 2000000  00:00:00.003350455 |

Each program in this section usually shows one program execution. The program in figure 18.38 shows two executions, but the next set of programs will only demonstrate one execution. Make sure that you execute the program multiple times with different quantities of data. In particular, double the data amount several times. I will display my observations in a matrix after each program. It is possible that you will get different results. Keep in mind that the execution efficiency of an algorithm depends on the processing speed of a computer. Each algorithm will be tested to the nearest nano-second.

Figure 18.39 show the behavior of the **counter** method. The data quantity is intentionally doubled twice. Does it appear that the process time also roughly doubles each time that the data quantity doubles?

**Figure 18.39**

|  |  |  |  |
| --- | --- | --- | --- |
| **Counter Algorithm** | | | |
| **Data Quantity** | **1,000,000** | **2,000,000** | **4,000,000** |
| **Process Time** | **0.001905056** | **0.003302208** | **0.0062202662** |

The next algorithm, program **Java1818.java** in figure 18.40, is our good friend the *Bubble Sort*. The implementation of a *Bubble Sort* involves a set of nested loops. When data processing involves nested loops, the behavior will be different from a straight forward single loop. Execute this program and once again use different quantities of data in some pattern. Be sure that the data is large enough. Comparing the efficiency measurements of small data sets is not very accurate.

**Figure 18.40**

|  |
| --- |
| // Java1818.java  // This program tests the efficiency of a nested loop algorithm  // like the Bubble Sort.  import java.util.\*;  public class Java1818  {  public static void main(String args[])  {  System.out.println("\nJava1818.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array size ===>> ");  int size = input.nextInt();  int[] list = new int[size];  createList(list,size);  System.out.println();  // displayList(list);    //////////  time.startClock();  bubbleSort(list);  time.stopClock();  //////////    // displayList(list);  System.out.println();  System.out.println(time);  }    public static void createList (int[] list, int size)  {  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  list[k] = rndInt;  }  }  public static void displayList(int[] list)  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  System.out.println("\n\n");  }    public static void bubbleSort(int[] list)  {  for (int p = 1; p < list.length; p++)  for ( int q = 0; q < list.length-p; q++)  if (list[q] > list[q+1])  {  int temp = list[q];  list[q] = list[q+1];  list[q+1] = temp;  }  }  } |

**Figure 18.40 Continued**

|  |
| --- |
| **Java1818.java Output**  Java1818.java  Enter array size ===>> 10000  00:00:00.517366233 |

When you only consider executing a *Bubble Sort* with a small data list, like 5000 data elements or less, you cannot tell that this is a slow sort. When you look at the matrix in figure 18.41, you may appreciate why the *Bubble Sort* is not real popular. Each time that the data list doubles, the process time shoots up. It appears to quadruple each time that the data quantity doubles. At 10,000 the time is about half a second. Waiting half a second is not so bad, but at 40,000 the time is already up to more than 8 seconds. At this rate a large data list can take more than an hour. You can certainly conclude that this sort is not the choice for a very large list.

**Figure 18.41**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubble Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.517366233** | **2.052005328** | **8.208783794** |

One reason why the *Selection Sort* is preferred to the *Bubble Sort*, is the more efficient swapping process. In a *Bubble Sort*, during each comparison pass any elements that are not in order are swapped. The *Selection Sort* still needs to make comparisons, but does not swap immediately when data is unsorted. At the conclusion of the comparison pass there is only one swap made with the largest array element. Program **Java1819.java**, in figure 18.42, tests the efficiency of the *Selection Sort*. When you run the program, make sure to use the same test data so that you can make an accurate comparison.

**Figure 18.42**

|  |
| --- |
| // Java1819.java  // This program tests the efficiency of a nested loop algorithm like the Selection Sort.  import java.util.\*;  public class Java1819  {  public static void main(String args[])  {  System.out.println("\nJava1819.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array size ===>> ");  int size = input.nextInt();  int[] list = new int[size];  createList(list,size);  System.out.println();  // displayList(list);    //////////  time.startClock();  selectionSort(list);  time.stopClock();  //////////    // displayList(list);  System.out.println();  System.out.println(time);  }    public static void createList (int[] list, int size)  {  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  list[k] = rndInt;  }  }  public static void displayList(int[] list)  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  System.out.println("\n\n");  }    public static void selectionSort(int[] list)  {  int smallest = 0;  for (int p = 0; p < list.length; p++)  {  smallest = p;  for (int q = p+1; q < list.length; q++)  if (list[q] < list[smallest])  smallest = q;  if (list[p] != list[smallest])  {  int temp = list[p];  list[p] = list[smallest];  list[smallest] = temp;  }  }  }  } |

**Figure 18.42 Continued**

|  |
| --- |
| **Java1819.java Output**  Java1819.java  Enter array size ===>> 10000  00:00:00.185082756 |

Is the *Selection Sort* faster than the *Bubble Sort*? The first execution certainly indicates that it is faster and as you test other array sizes you will conclude that there is a significant difference. It appears that the single swap for each comparison pass does improve matters. The matrix in figure 18.43 intentionally shows the results of the two sorting routines that have been tested so far.

**Figure 18.43**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubble Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.517366233** | **2.052005328** | **8.208783794** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Selection Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.185082756** | **0.756219155** | **3.034551756** |

The matrix shows some interesting results. The *Selection Sort* manages to sort 40,000 numbers faster than the *Bubble Sort* can sort 10,000 numbers, but there is something curious going on if you look at the matrix closely. What happens to the *Selection Sort* when the data list doubles and doubles again? You will see that the process time quadruples each time. In fact, in this area both the *Bubble Sort* and the *Selection Sort* have similar behavior. Undeniably, the *Selection Sort* is faster, but this business of quadrupling every time the list doubles means that once again a large data list will be unacceptable, even for the improved *Selection Sort*.

What do you think will happen when we test the *Insertion Sort*? Will it be the same, better or worse than the previous two sorts? You may conclude that it has to be faster than the *Bubble Sort*, because it has been mentioned that the *Bubble Sort* is the slowest sort. Program **Java1820.java**, in figure 18.44, tests the *Insertion Sort*. You will note that the *Insertion Sort* is a method of the **IntList** class. The previous programs have tested algorithms as static methods in the same class as the **main** method. This is not a requirement.

**Figure 18.44**

|  |
| --- |
| // Java1820.java  // This program demonstrates the efficiency of the Insertion Sort.  import java.util.\*;  public class Java1820  {  public static void main(String args[])  {  System.out.println("\nJava1820.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array size ===>> ");  int size = input.nextInt();  IntList x = new IntList(size);  System.out.println();  // x.displayList();    //////////  time.startClock();  x.insertionSort();  time.stopClock();  //////////    // x.displayList();  System.out.println();  System.out.println(time);  }  }  class IntList  {    private int[] intArray;  private int size;    public IntList(int s)  {  size = s;  intArray = new int[size];  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  intArray[k] = rndInt;  }  }  public void displayList()  {  System.out.println();  for (int k = 0; k < intArray.length; k++)  System.out.print(intArray[k] + " ");  System.out.println("\n\n");  }    private int linearSearch(int searchNumber, int numElements)  {  int index = 0;  while (index < numElements && searchNumber > intArray[index])  index++;  return index;  }  private void insertItem(int searchNumber, int numElements, int index)  {  for (int k = numElements-1; k > index; k--)  intArray[k] = intArray[k-1];  intArray[index] = searchNumber;  }  public void insertionSort()  {  for (int k = 0; k < size; k++)  {  int numElements = k + 1;  int index = linearSearch(intArray[k],numElements);  insertItem(intArray[k],numElements,index);  }  }  } |

**Figure 18.44 Continued**

|  |
| --- |
| **Java1820.java Output**  Java1820.java  Enter array size ===>> 10000  00:00:00.181686235 |

The first impression is that the *Insertion Sort* behaves somewhere in the same area as the *Selection Sort*. This is true, but after we look at the comparison matrix, in figure 18.45, we need to come back and do some more thinking. It is all about, making informed decisions about algorithm selection.

**Figure 18.45**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bubble Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.517366233** | **2.052005328** | **8.208783794** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Selection Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.185082756** | **0.756219155** | **3.034551756** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertion Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.181686235** | **0.711653836** | **2.832258840** |

So what do you think? Is the small improvement of the *Insertion Sort* significant? Would you use the longer code of the *Insertion Sort* to say no to the simpler code of the *Selection Sort*? Actually, there is another issue.

The *Bubble Sort* and the *Selection Sort* both sort an entire set of data. If this involves an array of patient records then the addition of new patient records does involve sorting the entire set of records with the new records added. Does that seem odd? Forget computers right now. If you work in an office with 5000 files that are nicely sorted according to customer account numbers, and several new files are added, will you then start from the beginning and sort all the files? That will probably seem absurd. It makes more sense to take the few new files and insert them in the proper location.

In other words, the *Insertion Sort* is not much better than the *Selection Sort* if both sorts are used to sort 10,000 records. However, if there is the situation where you have 10,000 records that are already sorted and now 10 additional records are added, you may find the *Insertion Sort* is superior by far.

One thing is sure about the first three sorting algorithms, they all quadruple in process time when the data list doubles. This quadrupling business really kills a sorting algorithm for a large list. So is there hope for doing any type of sorting when a list becomes large, or do we simply go out for a long lunch while the data list gets sorted?

The time has come to test the *Merge Sort*. This nifty sort was introduced as an advanced sort that would not get a whole lot of explanation in the implementation department. This philosophy has not changed. You will see the code of the *Merge Sort* with program **Java1821.java**, in figure 18.46, and you will realize one thing for sure. The *Merge Sort* is more complicated than the *Bubble Sort*. How the *Merge Sort* is coded will be a topic for the next course, but right now observe the execution speed of this sort. It really is quite amazing. We are going to look at the performance of some large lists. If you did not realize earlier why the method calls to **displayList** were commented out, you will see the need now.

**Figure 18.46**

|  |
| --- |
| // Java1821.java  // This program demonstrates the amazing efficiency of the Merge Sort.  import java.util.\*;  public class Java1821  {  public static void main(String args[])  {  System.out.println("\nJava1821.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array size ===>> ");  int size = input.nextInt();  IntList x = new IntList(size);  //x.displayList();  System.out.println();  //////////  time.startClock();  x.mergeSort(0,size-1);  time.stopClock();  //////////  //x.displayList();  System.out.println();  System.out.println(time);  }  }  class IntList  {  private int[] intArray;  private int[] tempArray;    public IntList(int size)  {  intArray = new int[size];  tempArray = new int[size];  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  intArray[k] = rndInt;  }  }  public void displayList()  {  System.out.println();  for (int k = 0; k < intArray.length; k++)  System.out.print(intArray[k] + " ");  System.out.println("\n\n");  }    public void merge(int first, int mid, int last)  {  int p = first;  int q = mid+1;  int k = first;  while (p <= mid && q <= last)  {  if (intArray[p] <= intArray[q])  {  tempArray[k] = intArray[p];  p++;  }  else  {  tempArray[k] = intArray[q];  q++;  }  k++;  }  while (p <= mid)  {  tempArray[k] = intArray[p];  p++;  k++;  }  while (q <= last)  {  tempArray[k] = intArray[q];  q++;  k++;  }  for (int h = first; h <= last; h++)  intArray[h] = tempArray[h];  }      public void mergeSort(int first, int last)  {  if (first < last)  {  int mid = (first + last) / 2;  mergeSort(first,mid);  mergeSort(mid+1,last);  merge(first,mid,last);  }  }    } |

**Figure 18.46 Continued**

|  |
| --- |
| **Java1821.java Output #1**  Java1821.java  Enter array size ===>> 10000  00:00:00.005125010 |

**Figure 18.46 Continued**

|  |
| --- |
| **Java1821.java Output #2**  Java1821.java  Enter array size ===>> 1000000  00:00:00.428386483 |

The first observation you will make is that the *Merge* *Sort* is much faster than the previous sorts. The true difference is not realized until large data sets are tested. Look at the second output where 100,000 are tested. The *Merge Sort* actually sorts 100,000 numbers faster than the *Bubble Sort* manages with 10,000 numbers. Look at the efficiency matrix, in figure 18.47, and compare the *Merge Sort* with the previous fastest sort, the *Insertion* Sort. I do believe we have a sort for large data sets now.

**Figure 18.47**

|  |  |  |  |
| --- | --- | --- | --- |
| **Insertion Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.181686235** | **0.711653836** | **2.832258840** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge Sort Algorithm** | | | |
| **Data Quantity** | **10,000** | **20,000** | **40,000** |
| **Process time** | **0.005188043** | **0.008757089** | **0.18028978** |

You can tell that the *Merge Sort* does not have this quadruple problem of the other three sorts. The process time seems to roughly double as the data list double. What exactly happens with the efficiency of the *Merge Sort* will be explained in the future. Right now there is no question that it is blazingly fast. Consider that the **0.181686235** time for sorting **10,000** numbers with the *Insertion Sort* is roughly the same time that the *Merge Sort* uses to sort **40,000** numbers. If you actually continue and compare the earlier sorts with the *Merge Sort* for ever larger numbers, you will find that the *Merge Sort* will look better and better by comparison.

There is more to algorithmic life than sorting. After all, was it not mentioned that the primary purpose for sorting data is so that it can be searched more efficiently? Earlier in this chapter the *Binary Search* was shown as a marvel of efficiency over the rather blah *Linear Search*. Now there exists a slight problem. Searching requires much less processing than sorting. This is logical. If you need to find one file in a set of 10,000 files, you will process less than if you must sort every one of the 10,000 files. This fast processing speed requires that an intentional delay is used to slow the algorithms down. There still is a pretty accurate comparison, because both searches will get the same delay. Each time that the loop is iterated, the same time delay is used.

It is a little like comparing the acceleration of two trucks. You can place 1000 pounds in the bed of each truck. This added weight will slow down the acceleration, but you can still compare the two trucks accurately. Program **Java1822.java**, in figure 18.48, starts with the *Linear Search*

**Figure 18.48**

|  |
| --- |
| // Java1822.java  // This program measures the time necessary to find an element  // in an array of randomly arranged integers with a Linear Search algorithm.  import java.util.\*;  public class Java1822  {  public static void main(String args[])  {  System.out.println("\nJava1822.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array Size ===>> ");  int size = input.nextInt();  int[] list = new int[size];  createList(list,size);  System.out.println();  //displayList(list);  System.out.print("Enter search item ===>> ");  int searchItem = input.nextInt();      //////////  time.startClock();  int index = linearSearch(list,searchItem);  time.stopClock();  //////////    displayIndex(index);  System.out.println();  System.out.println(time);  }      public static void createList (int[] list, int size)  {  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  list[k] = rndInt;  }  }      public static void displayList(int[] list)  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  System.out.println("\n\n");  }  public static int linearSearch(int[] list, int searchItem)  {  boolean found = false;  int k = 0;  while (k < list.length && !found)  {  if (list[k] == searchItem)  found = true;  else  k++;  }  if (found)  return k;  else  return -1;  }  public static void displayIndex(int index)  {  if (index == -1)  System.out.println("The SearchItem is not in the integer list ");  else  System.out.println("The SearchItem was found at index " + index);  }      } |

**Figure 18.48 Continued**

|  |
| --- |
| **Java1822.java Output**  Java1822.java  Enter array Size ===>> 1000000  Enter search item ===>> 100  The SearchItem is not in the integer list  00:00:00.004082603 |

The efficiency matrix, in figure 18.49 of the *Linear Search* shows that the process time doubles as the data size double. This should be an expected result. With small data sizes that result may not be quite as consistent. The larger data sets are also used to give a better comparison with the next *Binary Search*. The search algorithms are intentionally tested by looking for a number that does not exist. This is the worst-case scenario. Using the worst-case scenario makes for an accurate comparison. It is possible that the requested item is the first data element in the list. If the data element does not exist, searching will take the longest time.

**Figure 18.49**

|  |  |  |  |
| --- | --- | --- | --- |
| **Linear Search Algorithm** | | | |
| **Data Quantity** | **1,000,000** | **2,000,000** | **4,000,000** |
| **Process time** | **0.004082603** | **0.007305898** | **0.013774055** |

Now consider the *Binary Search*, shown by **Java1823.java** in figure 18.50. You have learned earlier in this chapter that it has a remarkable efficiency. Now you have the **TimeTest** class to see if this is true.

**Figure 18.50**

|  |
| --- |
| // Java1823.java  // This program measures the time necessary to find an element  // in a sorted array with a Binary Search algorithm.  import java.util.\*;  public class Java1823  {  public static void main(String args[])  {  System.out.println("\nJava1823.java\n");  Scanner input = new Scanner(System.in);  TimeTest time = new TimeTest();  System.out.print("Enter array Size ===>> ");  int size = input.nextInt();  int[] list = new int[size];  createList(list,size);  System.out.println();  // displayList(list);  System.out.print("Enter search item ===>> ");  int searchItem = input.nextInt();  //////////  time.startClock();  int index = binarySearch(list,searchItem);  time.stopClock();  //////////  displayIndex(index);  System.out.println();  System.out.println(time);  }      public static void createList (int[] list, int size)  {  Random rand = new Random(1234);  for (int k = 0; k < size; k++)  {  int rndInt = rand.nextInt(9000) + 1000;  list[k] = rndInt;  }  }    public static void displayList(int[] list)  {  for (int k = 0; k < list.length; k++)  System.out.print(list[k] + " ");  System.out.println("\n\n");  }  public static int binarySearch( int[] list, int searchItem)  {  int lo = 0;  int hi = list.length-1;  int mid = 0;  boolean found = false;  while (lo <= hi && !found)  {  mid = (lo + hi) / 2;  if (list[mid] == searchItem)  found = true;  else  if (list[mid] > searchItem)  hi = mid - 1;  else  lo = mid + 1;  }  if (found)  return mid;  else  return -1;  }  public static void displayIndex(int index)  {  if (index == -1)  System.out.println("The SearchItem is not in the integer list ");  else  System.out.println("The SearchItem was found at index " + index);  }  } |

**Figure 18.50 Continued**

|  |
| --- |
| **Java1823.java Output**  Java1823.java  Enter array Size ===>> 1000000  Enter search item ===>> 111111111  The SearchItem is not in the integer list  00:00:00.000018189 |

The matrix, in figure 18.51, proves that the *Binary Search* is considerably faster than the *Linear Search*. Now keep in mind that this was tested with a still relatively small data list. With a very large data list the *Linear Search* will fall farther and farther behind its blazing big brother, the *Binary Search*.

**Figure 18.51**

|  |  |  |  |
| --- | --- | --- | --- |
| **Linear Search Algorithm** | | | |
| **Data Quantity** | **1,000,000** | **2,000,000** | **4,000,000** |
| **Process time** | **0.004082603** | **0.007305898** | **0.013774055** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Algorithm** | | | |
| **Data Quantity** | **1,000,000** | **2,000,000** | **4,000,000** |
| **Process time** | **0.000018189** | **0.000018487** | **0.000018985** |

**18.12 Exact Calculations of Executions**

You have been doing computer science for close to two semesters, or perhaps even longer. When you get to this chapter you should be realizing that computer science frequently lays the foundation for an important topic long before the topic is discussed in detail. This does mean that the teacher or author feels a little uncomfortable. Too much information may provide a complete picture of the topic, but it will likely confuse the majority of students. A modest introduction to the topic may be exactly the level of complexity that the class can handle, but now it is not really clear why this topic exists.

I do hope that you saw some benefit in our brief discussion about algorithm analysis in the previous section. This topic will continue in the next computer science course. Algorithmic analysis also requires that you can analyze how frequently program statements are executed. Now this is precise. It is not a matter of "well it roughly doubles or it roughly quadruples". No, can you exactly state how frequently certain program, statements will execute in a method?

Program **Java1824.java**, in figure 18.52, calls five different methods. Each method executes the statement **count++** a different number of times. Using an actual count will make it easy for the computer to verify our calculations. The program exists for proof primarily. Go past the program and we will analyze each one of the five methods.

**Figure 18.52**

|  |
| --- |
| // Java1824.java  // This program demonstrates the exact calculation counts of five different  // methods based on the entered value of n.  import java.util.\*;  public class Java1824  {  public static void main(String args[])  {  System.out.println("\nJava1824.java\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter the value of n ===>> ");  int n = input.nextInt();  System.out.println("\n\n");  method1(n);  method2(n);  method3(n);  method4(n);  method5(n);  System.out.println("\n\n");  }      public static void method1(int n)  {  int count = 0;  for (int k = 1; k <= n; k++)  count++;  System.out.println("Method 1: " + count);  System.out.println();  }    public static void method2(int n)  {  int count = 0;  for (int k = 1; k <= n/2; k++)  count++;  System.out.println("Method 2: " + count);  System.out.println();  }    public static void method3(int n)  {  int count = 0;  for (int p = 1; p <= n; p++)  for (int q = 1; q <= n; q++)  count++;  System.out.println("Method 3: " + count);  System.out.println();  }    public static void method4(int n)  {  int count = 0;  for (int p = 1; p < n; p++)  for (int q = 1; q < n; q++)  count++;  System.out.println("Method 4: " + count);  System.out.println();  }    public static void method5(int n)  {  int count = 0;  for (int p = 0; p < n; p++)  for (int q = 1; q <= n-p; q++)  count++;  System.out.println("Method 5: " + count);  System.out.println();  }  } |

**Figure 18.52 Continued**

|  |
| --- |
| **Java1824.java Output**  Java1824.java  Enter the value of n ===>> 100  Method 1: 100  Method 2: 50  Method 3: 10000  Method 4: 9801  Method 5: 5050 |

Each method will be shown in turn. Our mission is to look at the entered value for the variable **n** and then calculate the number of times that **count++** will be executed. It is always possible to calculate the execution count for a specific value of **n**, but it is better if you can make a general conclusion.

**method1**, in figure 18.53, starts nice and easy. This is a loop that starts with loop counter value **1** and continues until the loop counter equals the value of **n**. In this case the exact number of execution will always be identical to the value of **n**.

**Figure 18.53**

|  |
| --- |
| **public static void method1(int n)**  **{**  **int count = 0;**  **for (int k = 1; k <= n; k++)**  **count++;**  **System.out.println("Method 1: " + count);**  **System.out.println();**  **}** |

**method2**, in figure 18.54,is only a little tougher than the previous method. In this case the loop continues to repeat as long as the value has not reached **n/2**. This means that the execution count will be half the value of **n**.

**Figure 18.54**

|  |
| --- |
| **public static void method2(int n)**  **{**  **int count = 0;**  **for (int k = 1; k <= n/2; k++)**  **count++;**  **System.out.println("Method 2: " + count);**  **System.out.println();**  **}** |

**method3**, in figure 18.55, uses two nested loops. Both loops execute **n** times. This means that **count++** will be executed **n \* n** or **n2** times.

**Figure 18.55**

|  |
| --- |
| **public static void method3(int n)**  **{**  **int count = 0;**  **for (int p = 1; p <= n; p++)**  **for (int q = 1; q <= n; q++)**  **count++;**  **System.out.println("Method 3: " + count);**  **System.out.println();**  **}** |

**method4**, in figure 18.56, becomes a little tougher. Once again there is a set of nested loops, but now each loop executes **n - 1** times. This means that the inner statement executes **(n - 1) \* (n - 1)** or **n2 - 2n + 1** times. In the sample execution the value for **n** is **100**. So let us check it out.

Using the formula **n2 - 2n + 1**, we get **1002 - 2\*100 + 1** or **10000 - 200 + 1** and that equals **9,801**, which is what you see in the sample execution.

**Figure 18.56**

|  |
| --- |
| **public static void method4(int n)**  **{**  **int count = 0;**  **for (int p = 1; p < n; p++)**  **for (int q = 1; q < n; q++)**  **count++;**  **System.out.println("Method 4: " + count);**  **System.out.println();**  **}** |

The last method, in figure 18.57, is pretty tough. The outer loop executes **n - 1** times, which is not too difficult. The inner loop executes differently each time. It executes **n - p**, but **p** changes with each repetition of the outer loop. So how do we calculate this?

With a value of **100** for **n**, the outer loop calculates **100** times. Now the first time in the inner loop it executes **n - 0** or **100** times, the next time it is **99** times, then **98** times and so on until the last execution the inner loop executes **1** time. Perhaps there is a pattern. The first inner-execution plus the last one is **100 + 1** or **101**. The second execution plus the second-to-last one is **99 + 2** or **101**. If you continue this pattern you will see that there are **50** sums of **101** for a total execution count of **5050**. In general you get **(n2 + n) / 2**.

**Figure 18.57**

|  |
| --- |
| **public static void method5(int n)**  **{**  **int count = 0;**  **for (int p = 0; p < n; p++)**  **for (int q = 1; q <= n-p; q++)**  **count++;**  **System.out.println("Method 5: " + count);**  **System.out.println();**  **}** |

**18.13 Summary**

This chapter is an excellent example of a *computer science* chapter, rather than a *Java* chapter. The chapter is titled *Algorithms I*, because this topic will be continued in a later chapter.

An algorithm is a logical sequence of unambiguous instruction to accomplish a specified goal. An algorithm is program language independent. Algorithms can be explained, analyzed and implemented in any program language.

In this chapter the earlier **List** case study of the array chapter is used to show a variety of algorithms that operate on a list of data. For most of the program examples the **List** object contains random integers.

The first algorithm is the *Linear Search*. This algorithm searches a list of data sequentially from start to end to find the requested data. The *Linear Search* can find data in a list regardless of its sorting order.

Sorting algorithms are both common and important in computer science. Arranging data in a desired order makes it easier and more efficient to find data.

The *Bubble Sort* is the easiest sorting algorithm. It is also the most inefficient way to sort data. With today's computers this is not a concern for small lists of data of 1000 or less elements. The *Bubble Sort* compares adjacent elements and swaps them if they are not in the proper sequence.

The *Selection Sort* improves one problem area for the *Bubble Sort*, by reducing the many swaps made during each comparison pass. The *Selection Sort* algorithm stores the smallest array element until all elements have been compared. At the conclusion of the entire comparison pass only one swap is made.

The *Insertion Sort* uses a sorting process that may well be the natural way that people sort cards, files or any data. Each new element is added, or inserted, in the correct location.

The first three sorts have no difficulty with data lists that are not too large. For lists up to 1000 elements, there will not be any noticeable delay. For large lists, the delay is in many seconds and very large lists will mean that sorting may take minutes or hours. The *Merge Sort* uses a clever merging process of existing sorted lists to sort lists very rapidly.

Processing a list with records requires special considerations. First, there is the awareness of instantiation all the required objects properly. A record in Java is implemented with a class, which results in an array of objects. The array itself is an object, which is part of an even larger List object. Another consideration with algorithms that process records is that comparisons and searches need to be made according to a specified field in the record.

The **TimeTest** class was introduced to test the execution efficiency of the algorithms in this chapter. The analysis of the algorithms is considered informal. You learned that the *Selection Sort* and the *Insertion Sort* both were superior to the *Bubble Sort****,*** but all three of the sorts become unacceptable for large lists. These sorts show that the process time quadruples each time that the data list doubles in size.

The *Merge Sort* tested at a very remarkable speed, even for very large data lists. The process time seems to increase roughly at the same rate as the data list.

The execution efficiency test also showed that the *Binary Search* is considerably more efficient than the *Linear Search*.