**Institute of Technology Tralee**

**Computing Department**

**Structured Programming 2**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 13 – Array Algorithms**

In the last lab sheet we looked at some Java API methods that allowed us to quickly sort and search any array we wanted, through the **Arrays** class. In order to give you a proper appreciation of what these API methods do, and just how much time and effort they save us as developers, we will now do it the “hard way” from first principles and examine one particular **sorting** **algorithm**, followed by an examination of the **binary-search** **algorithm**. Hopefully a decent knowledge of the operation of these algorithms will also improve your problem-solving skills and better prepare you for the **Algorithms & Data Structures** module in year 2. **Take your time** reading through this lab sheet and try to digest the material as well as you can, asking questions if you get stuck anywhere.

**The Selection Sort Algorithm**

As mentioned in the last lab sheet, sorting information is a hugely important task in computing and is one of the most intensely researched areas within the field. If you have a list of jobs to do on a particular day, sometimes you will have to prioritise these jobs by sorting them numerically. These everyday occurrences all rely on sorting in some shape or form.

In terms of sorting arrays, it is possible to do so in either **ascending** or **descending** order. Many sorting algorithms exist e.g. “bubble sort”, “quick sort”, “merge sort” and each one varies in terms of its complexity and efficiency. One of the more straightforward sorting algorithms is called “**selection sort**” but it is important to realise that it is certainly not the most efficient one (more complicated usually implies more efficient!!). However, it does what it is meant to do and, for the programs we shall use it with, it will serve its purpose adequately.

**Understanding the Operation of the Selection Sort Algorithm**

For the sake of demonstration, we shall assume we are sorting the list into ***ascending*** order. Our integer array is being declared to have a size of 10.

You can take it that the following values have been input to the array:

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 23 | 12 | 6 | 64 | 5 | 13 | 1 | 84 | 3 | 79 |

subscript number: 0 1 2 3 4 5 6 7 8 9

The algorithm works as follows:

**1.** First look at the array as a whole and find the **smallest** of all the elements in the array i.e. find the smallest of nums[0], nums[1], nums[2]........nums[9] in the example above. In this case the smallest element would be nums[6] and its value is 1.

**2.** **Swap** this element with nums[0] (the first slot in the entire array) so that in this case nums[0] is now 1 and nums[6] will be 23.

So after this step the array looks as follows:

Subarray with 9 values

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 12 | 6 | 64 | 5 | 13 | 23 | 84 | 3 | 79 |

subscript number: 0 1 2 3 4 5 6 7 8 9

**3.** For all intents and purposes, the first block (element) of the array is now **sorted** in ascending order, so we now look to find the smallest value in the **remainder** of the array i.e. the smaller “**subarray**” nums[1], nums[2], nums[3],.....nums[9]. Here the smallest value encountered will be nums[8] which is 3.

**4.** Swap this element with nums[1] (the first slot of the smaller subarray) so that in this case nums[1] is now 3 and nums[8] will be 12.

So after this step the array looks as follows:

Subarray with 8 values

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 6 | 64 | 5 | 13 | 23 | 84 | 12 | 79 |

subscript number: 0 1 2 3 4 5 6 7 8 9

**5.** Continue **applying the same process** to each “subarray” until we are left with a subarray which contains just 1 element.

**6.** Then the array is completely sorted in ascending order and we end up with the following:

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 6 | 12 | 13 | 23 | 64 | 79 | 84 |

subscript number: 0 1 2 3 4 5 6 7 8 9

So, conceptually at least, the algorithm is reasonably straightforward to make sense of. The next program aims to show how it can be implemented in Java.

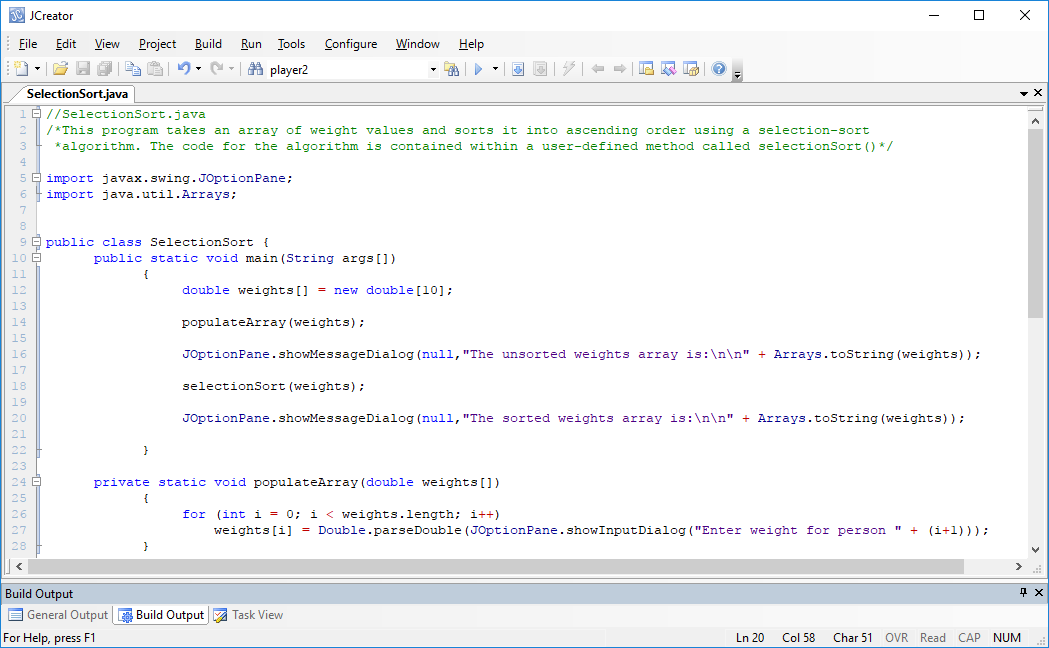
For those who like a more visual view of things, please see the following link in your browser which shows an interactive animation of the selection-sort algorithm

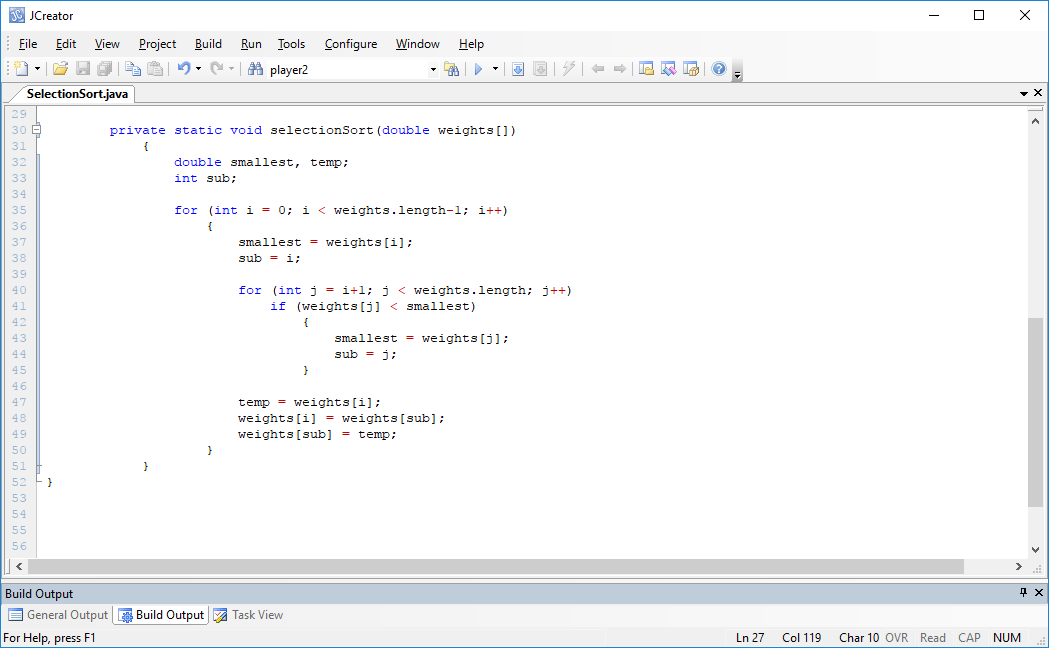
<http://www.cs.armstrong.edu/liang/animation/web/SelectionSort.html>

**Using the Selection-Sort Algorithm**

**Aim:** This program shows the selection-sort algorithm being used to sort an array of weight values into ascending order through a user-defined method called **selectionSort**()

**Java Code:**





**Program Analysis:**

● The program declares a double array of size 10 to store 10 weight values. weights is the object reference that “points to” this array. Then a populateArray() method is called and the object reference weights is passed into it. This method allows the user to populate the array with a set of values, via input dialogs. The code within the method

**weights[i] = Double.parseDouble(JOptionPane.showInputDialog("Enter weight for person " + (i+1)));**

shows that it is possible to combine the prompting, reading in and conversion into a single line, as you may have seen elsewhere such as in text-books or on the Web. You are, of course, welcome to break this process up into 2 separate steps as I have always done in my sample programs and solutions.

You will also see that, on this occasion, I have used exactly the same name **weights** for the method parameter name as I did for the array object reference back in main(). There is **no conflict** at all here because both identifiers are local to the methods they appear in and cannot be seen outside of these methods.

• When the array has been populated with values, it is displayed on a message dialog using the Arrays class **toString**() method. The eagle-eyed among you may have noticed that I used a slightly different version of the **showMessageDialog**() method here. This abbreviated version takes just **2 arguments**, compared to the 4 I normally use. You will see that with this version the title-bar receives a default value of “Message” and an information-message icon is used. There are yet other versions of this method also within the API – having several methods defined within a class that all share the same name but have different parameter lists is called **method overloading** and is one of the OO concepts you will cover in year 2.

• Once the unsorted array has been displayed, it is then passed (or at least a reference to it is passed) to the **selectionSort**() method.

The code

**double temp, smallest;**

**int sub;**

declares two double variables and a single int variable.

The job of the variable **temp** is to temporarily store the value of the first element within the subarray under consideration. It plays a crucial role in the **swapping phase** of the selection-sort algorithm, the swapping couldn’t take place without the existence of this so-called **temp**orary or “dummy” variable. This will be discussed shortly on page 7.

The variable **smallest** is there to store the value of the smallest element encountered so far for the subarray under consideration. Once the subarray has been fully examined, this variable will store the value of the smallest weight found within that subarray.

The variable **sub** is there to store the subscript number of the smallest element encountered so far for the subarray under consideration. Once the subarray has been fully examined, and the smallest element has been found within that subarray, its subscript number is stored within this variable and used later when the swapping occurs.

• the code

**for (int i = 0; i < weights.length-1; i++)**

**{**

**smallest = weights[i];**

**sub = i;**

**for (int j = i+1; j < weights.length; j++)**

**if (weights[j] < smallest)**

**{**

**smallest = weights[j];**

**sub = j;**

**}**

**temp = weights[i];**

**weights[i] = weights[sub];**

**weights[sub] = temp;**

**}**

takes the weights array and puts it into ascending order. You will notice that there are 2 for loops being used here. The **outer** for loop has the following form:

**for (int i = 0; i < weights.length-1; i++)**

**{**

**//loop to find the smallest of weights[i], .....,weights[9] and note its subscript**

**//swap weights[i] with weights[sub] (the smallest weight value)**

**}**

As you can see, it will iterate a total of 9 times in this case. The reason we don’t need the outer loop to iterate 10 times is that we know once the loop has iterated 9 times the array will be sorted (on the 9th iteration only a subarray with 2 elements in it remains and so after sorting these 2, both elements must be in their correct location in the list). However, it is worth noting that if we had a 10th iteration it would not affect the program in a negative way, except that it **would not be as efficient** as it might be.

• The code

**smallest = weights[i];**

**sub = i;**

appears at the beginning of the **outer** for loop. These lines of code **initialise** the values of smallest and sub to (respectively) the value of the first element of each successive subarray and the corresponding subscript number of each subarray’s first element, which are likely to change, especially when the subarray is large, since we are likely to find a smaller weight value as we examine the remaining elements of the subarray in question.

The **internal** for loop has the following form

**for (int j = i+1; j < weights.length; j++)**

**if (weights[j] < smallest)**

**{**

**smallest = weights[j];**

**sub = j;**

**}**

This loop iterates a different number of times, depending on the current value of the variable i (which comes from the **outer** for loop). So, when the outer loop runs initially, the value of i will be 0 and hence the internal for loop will read as follows:

**for (int j = 1; j < 10; j++)**

**if (weights[j] < smallest)**

**{**

**smallest = weights[j];**

**sub = j;**

**}**

So we see that the inner loop will iterate 9 times in this case. Each time the loop iterates, a test is performed where the weight of each element in the array in question (weights[j]) is compared with the value of the variable smallest (which stores the value of the smallest element encountered to this point). If it turns out that the value of weights[j] is less than that of smallest element encountered so far, then smallest is reset with the value of weights[j] and the subscript number of this element (j) is noted and stored in variable sub. Otherwise, nothing happens and the loop continues to the next iteration, and continues to iterate until j reaches the value 10, at which point the loop test condition becomes false.

Obviously, as the outer for loop continues to iterate, the number of iterations of the internal for loop decreases and decreases until eventually the outer for loop terminates, at which stage the entire array is sorted in ascending order.

• the code

**temp = weights[i];**

**weights[i] = weights[sub];**

**weights[sub] = temp;**

actually performs the task of **swapping** the values of the smallest element found in the subarray under consideration i.e. weights[sub] with the first element of that subarray, weights[i]. In order to perform this swap, a **temporary variable** needs to be introduced to temporarily store one of the values – to appreciate this concept better, consider the case where we have two integer variables a and b where a is 2 and b is 5. If we now wish to swap their values we could try doing the following:

**a=b;**

**b=a;**

we might think this would swap the values so a is now 5 and b is now 2 but let’s just look at this a bit closer

**a=b;**

will certainly set the value of a to 5 as required

but

**b=a;**

will just leave b as it is, since a is now 5. So both variables end up with the value 5.

So we see why it is necessary to introduce a temporary variable (it’s usually called *temp*) to perform the swap correctly. The idea is that the temporary variable will hold on to the current value of one of the variables involved in the swapping process. As a thought experiment now, and using the example of a and b above, introduce the extra variable temp and prove to yourself that the successful swapping of the values of a and b can be achieved.

**Organising your Work**

You should have a folder under X: called SP2Stuff created. This time, create a folder called **Lab13** within SP2Stuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **SelectionSort.java** in your Lab13 folder. Now, for practice, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and test it fully. With proper testing you should see that this program is **not validated.**

**Exercise 1**

Write a program called Exercise1.java that **initializes** two parallel arrays with the following values which store the names and corresponding GPAs of a set of students:

**names**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| James Maye | Joe Bloggs | Jane Doe | Teresa Coughlan | Sam Stewart |

**GPAs**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3.56 | 2.47 | 3.12 | 2.55 | 2.78 |

Once initialized, the program should then proceed to sort the GPAs array in ascending order, using the selection-sort algorithm. Note here that you need to **modify** the selection-sort method/algorithm used in the sample program as you need to also make sure the corresponding parallel array of names ends up sorted in the same order. In the first instance, try to code the method from scratch to see how much of the algorithm you can recall.

Once sorted, your program should then display the contents of the parallel arrays in **descending order** on a text-area as indicated below (so the student with the highest GPA appears first) neatly aligned, using formatting - create a method called **displayResults**() for this part. Display the GPAs to **2 decimal places**.

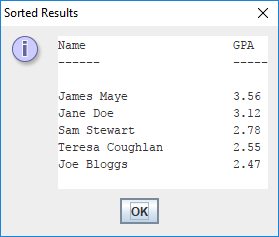
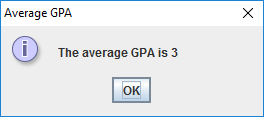
Next, your program should include an additional user-defined method called **averageGPA**() to determine and return the (accurate) average GPA achieved by the set of students - this should then be displayed in main() to **the nearest whole number**.

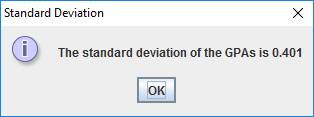
Finally, there should be a method called **standardDeviationGPA()** to determine and return the standard deviation of the GPAs - and then display it in main() to **3 decimal places**. Note that the standard deviation, σ is given by the formula:



Where N is the number of GPA values under consideration (5 in this case), µ represents the average GPA, xi represents each individual GPA value in the set and

 means the sum of the squares of the differences between the individual GPAs and the average GPA (try to **reuse** the **averageGPA**() method when writing this one, by **calling it** within the method).



**The Binary Search Algorithm**

You have already seen a **linear search** algorithm being used in the previous lab sheet. It is relatively straightforward to understand – it’s just a case of beginning at the very first element of the array and then proceeding to look at each subsequent element, until the “key” value we seek has been found (although it may not exist within the array at all). In terms of **efficiency**, we would say that this linear search algorithm is not the best because, on average, for values that do exist, we must look at half the elements within the array every time we do a search. Sometimes we will be lucky, and find the value we seek very quickly (it might be within the first few slots of the array), yet other times we will have to look through almost every slot in the array, and sometimes it may even be the last value in the array. If we consider cases where the value may not even exist within the array, the efficiency of the algorithm is even worse.

The binary search algorithm is used to **improve the efficiency** of our searching. We mentioned this algorithm in the last lab sheet and used the one provided with the Arrays class of the Java API. It was stated that this algorithm “chops” the range of values to be searched in 2 each time it iterates, hence the term “binary” search. The key to the operation of the algorithm is that the values in the array have **already been sorted** into either ascending or descending order.

**Understanding the Operation of the Binary Search Algorithm**

For the sake of demonstration, we shall assume we are dealing with an array that has already been **sorted** **into ascending order**. We will use the same sorted array we finished with at the end of the selection-sort discussion earlier and we will imagine that we are searching for the value 79 in the array. This integer array is of size 10 and has the following values following the sort:

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 6 | 12 | 13 | 23 | 64 | 79 | 84 |

subscript number: 0 1 2 3 4 5 6 7 8 9

The algorithm works as follows:

**1.** First, taking the array as a whole, find the **middle element** in the array. In this case there are an even number of elements in the array so we could argue that the middle element lies at either nums[4] or nums[5]. The algorithm is normally written to take the **integer average** of the lowest and highest subscript numbers under consideration so, in this case, it would be the average of 0 and 9, which is 4.5 but, as an integer in Java, this becomes 4 i.e. the 5th element in the array.

**2.** We now compare the value stored at this element of the array with the value we seek, 79. We see that the value stored at nums[4] is 12 in this case. As the value we seek exceeds this value, and we know the array is sorted into ascending order, we can deduce that the value we seek cannot be in the first half of the array we are looking through. So we have effectively halved our search range as a result of just one test! We can now say that, if the value 79 is in the array, it must be in the second half of it, so the next time we do our search, we can limit it to this part of the array. We are basically **resetting our lower limit of the search range to 1 beyond the current middle element**.

The range of values of interest

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 6 | 12 | 13 | 23 | 64 | 79 | 84 |

subscript number: 0 1 2 3 4 5 6 7 8 9

**3.** In a similar way, we now determine which value is stored at the middle element of this smaller array. In this case the average of the lower and upper subscript numbers is (5+9)/2 = 7 so the middle element lies at nums[7]. We then compare the value we seek, 79, we the value stored at nums[7], which is 64. It turns out 79 exceeds 64 so we can now eliminate all the elements in the first half of this smaller array from being examined in the future, since the array is sorted in ascending order. So again we are **resetting our lower limit of the search range to 1 beyond the current middle element**

The range of values of interest

Array name: nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 5 | 6 | 12 | 13 | 23 | 64 | 79 | 84 |

subscript number: 0 1 2 3 4 5 6 7 8 9

**4.** At this point we have chopped our search range in half again and are just left with 2 possible slots. In this case the average of the lower and upper subscript numbers is (8+9)/2 = 8.5 which becomes 8 as a Java integer. So the value we seek, 79, is compared with the value stored at nums[8], which is also 79. As the values match, we have found the value we were looking for and the algorithm can stop immediately.

**5.** Imagine now that we were looking for the value 89 instead. As you can see, this value does not exist within the array. In this instance, the algorithm would have worked just as before, and then the value 89 would be compared to 79. As 89 exceeds 79 then the search range would be chopped in half to leave just one remaining element. The value stored at nums[9], which is 84, would be compared to 89 and, as there is no match and because 89 exceeds 84, the next step would be to reset the lower limit of the search range to 1 beyond the current middle element – this would make the lower limit 10, which **exceeds the upper limit value** of 9, so the algorithm would stop and, similar to the linear search we looked at before, we might indicate this by returning a subscript value such as -1 to say “the value you were looking for could not be found” or else maybe just return the boolean value false.

Again, conceptually at least, the binary search algorithm is reasonably straightforward to understand. The next program aims to show how it can be implemented in Java.

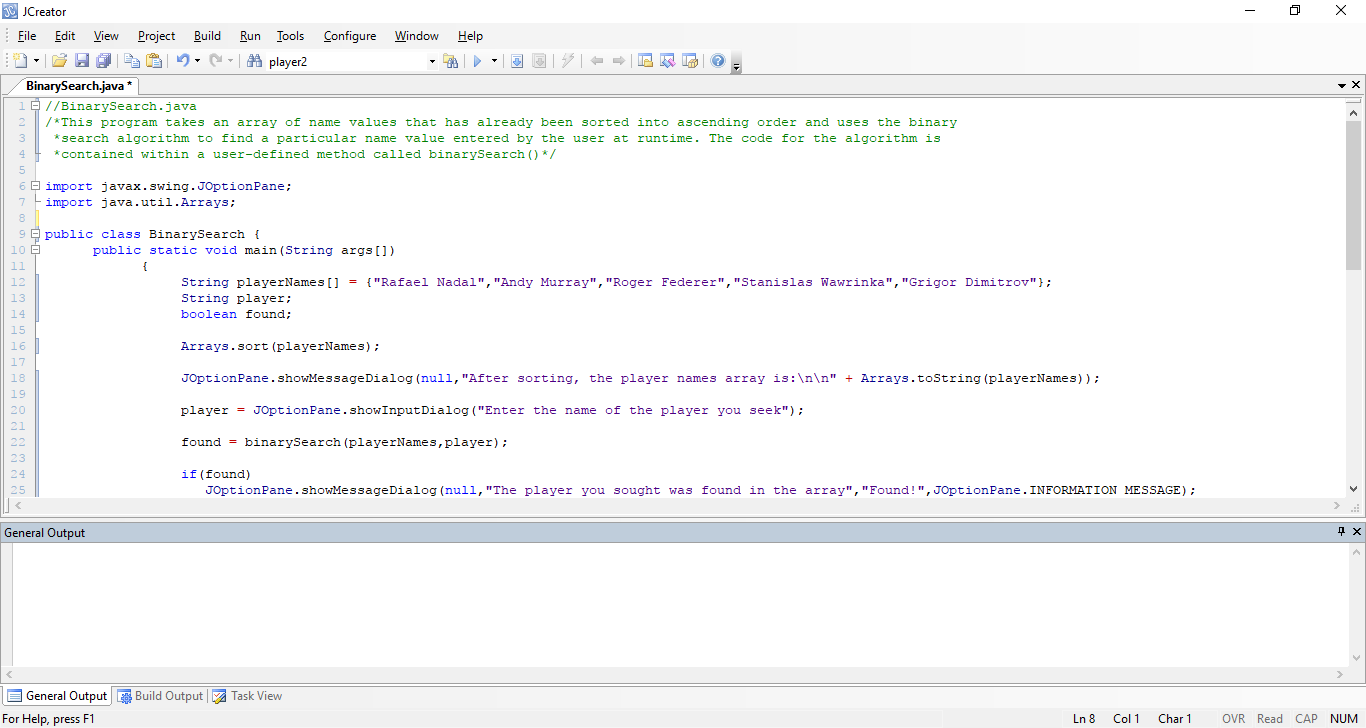
For those who like a more visual view of things, please see the following link in your browser which shows an interactive animation of the binary search algorithm

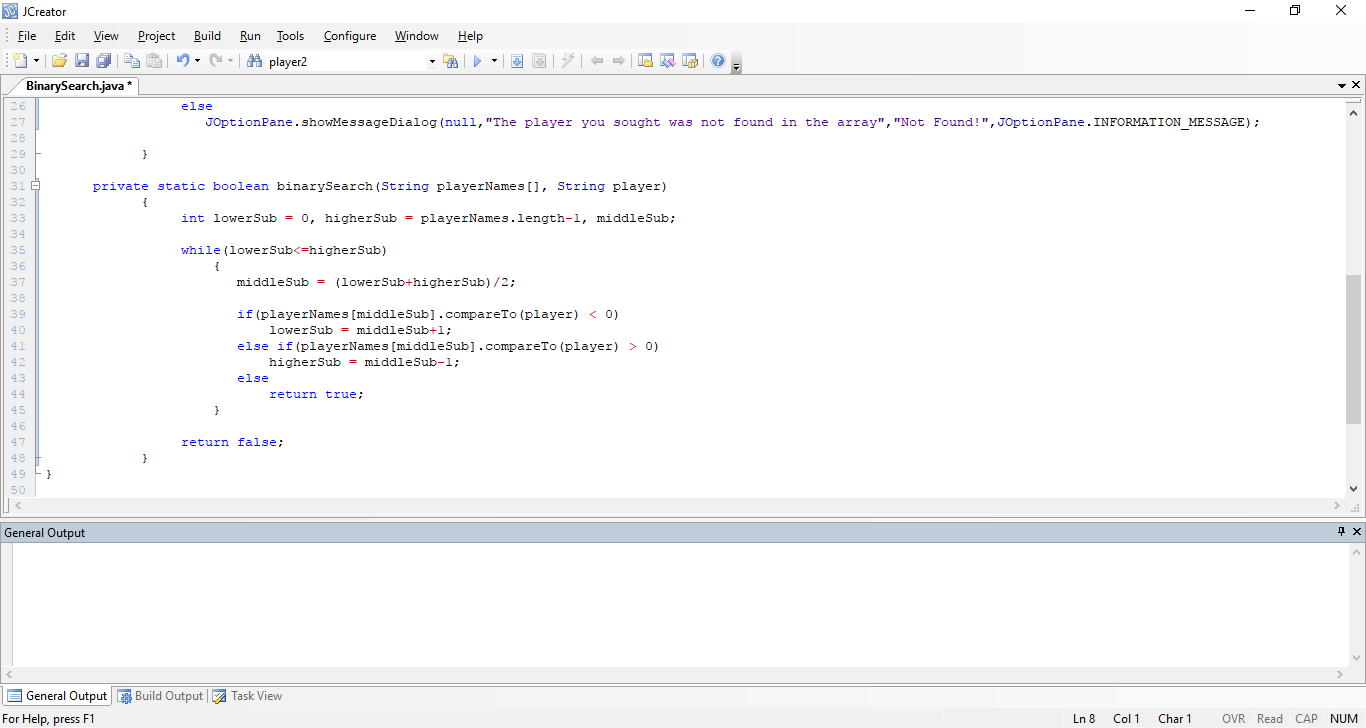
<http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html>

**Using the Binary Search Algorithm**

**Aim:** This program shows the binary search algorithm being used on an array of name values (that has already been sorted into ascending order) through a user-defined method called **binarySearch**()

**Java Code:**





**Program Analysis:**

● The program initializes a String array with the names of 5 tennis players. playerNames is the object reference that “points to” this array. Then the array is sorted into alphabetical order, using the **sort()** method of the **Arrays** class, and displayed using the **toString()** method from the same class. So at this point the output appears as follows, proving the sort was successful:



● Next the user is asked to enter the name of the player they wish to find in the array. Once the value is entered it is passed, along with the playerNames array, to the **binarySearch**() method.

● The binarySearch() method begins by defining 3 integer variables. **lowerSub** will store the lowest subscript number of the array under consideration. To begin with, this will always be zero as we are looking at the entire array at that point. **higherSub** will store the highest subscript number of the array under consideration. To begin with, this will always be **playerNames.length – 1**, so in this case it really holds the value 4 to begin with. The variable **middleSub** will be instrumental to the operation of the algorithm but this does not need to be set to begin with.

● The algorithm uses a **task-controlled** while loop that potentially iterates until the value of lowerSub exceeds the value of higherSub. At this point the loop test condition becomes false and the loop stops, returning the boolean value false to indicate that the value sought by the user could not be found within the array.

● Within the loop body lies the real “meat” of the algorithm. Firstly, the value of middleSub is determined. As outlined earlier, this is just the **integer average** of the lower and higher subscript numbers for the array under consideration. So, in this case, when the while loop iterates for the very first time, lowerSub will be zero, higherSub will be 4, therefore middleSub will be (0+4)/2 = 2, which is the subscript number associated with the middle element of the array – this element of the sorted array has the value “Rafael Nadal”

● Next up is a **nested else-if** structure that is used to compare the value the user is looking for (stored in the variable player) with the value stored at the middle element of the array under consideration i.e. playerNames[middleSub], which is “Rafael Nadal” for the first iteration of the while loop. You can see here that the method used to compare String values alphabetically in Java is called **compareTo**().

This method compares the String values letter by letter and the code

**playerNames[middleSub].compareTo(player) < 0**

will compare the String “Rafael Nadal” to the search String entered by the user, the first time the loop iterates. If it turns out that the String “Rafael Nadal” is “**alphabetically less than**” the String entered by the user, then the above expression evaluates to true and it would mean that the String entered by the user must (assuming only letters were entered) begin with something in the range R-Z uppercase or a-z lowercase e.g. “Raga Muffin” would be alphabetically greater than “Rafael Nadal” because the g in “Raga” comes after the f in “Rafael” in the alphabet. The compareTo() method is based on the **ASCII code** values of characters, as you might expect. Recall that the ASCII codes of digits are less than those for the uppercase letters which, in turn, are less than those of lowercase letters.

So, if it turns out that the string at the middle position in the array under consideration is alphabetically less than the string entered by the user, then it means **we** **can effectively ignore all other slots in the array** **below that point** and thereby halve our search range for the next loop iteration. This is achieved with the code

**lowerSub** = **middleSub+1**;

● In a similar manner, if it turns out that the expression

**playerNames[middleSub].compareTo(player) > 0**

evaluates to true, then it must be the case that the String at the middle position in the array under consideration is “**alphabetically greater than**” the String entered by the user. In this case it would mean that any strings in the slots of the array **above** the middle slot cannot possibly match the search string and so the search range can be halved for the next loop iteration through the code

**higherSub = middleSub-1;**

● Of course, if neither of the previous expressions evaluated to true then it must be the case that the String entered by the user **exactly matches** the String located at playerNames[middleSub], which, on the first iteration of the while loop, would mean the user entered the String “Rafael Nadal”. In this case, the method would return the boolean value true immediately to indicate an exact match.

● As mentioned earlier, in the absence of a match, the while loop will continue to iterate, halving the search range each time until eventually the value of lowerSub exceeds higherSub. When that occurs, the loop stops and the boolean value false is returned from the method to indicate **no match** was found.

● Once the method returns its boolean result to main(), the value returned is used within an **if-else** structure to decide the appropriate display message for the user.

**There is a lot of food for thought in this sample program – you should now take the sorted playerNames String array discussed above, pick some random String value of your own choosing and try to trace through the binarySearch() method using pen and paper to see how it would operate for your chosen value. Doing this will help you to get to grips with the mechanics of the algorithm much more quickly.**

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **BinarySearch.java** in your Lab13 folder. Now, for practice, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and **test** it fully.

**Exercise 2**

Write a program called Exercise2.java that begins by creating 2 parallel arrays that will respectively store the t-numbers and course names of exactly 10 ITT students. When the arrays have been populated with data, through a single user-defined method called **populateArrays**(), the content of the arrays should be displayed just by using the **Arrays.toString**() method. Then the arrays should be **sorted** into ascending order by t-number using a **modified** version of the **selectionSort**() algorithm covered earlier in the lab sheet – remember here that when you are sorting the t-numbers array, you’ll want to also make sure the corresponding slots in the course names array also move to the correct positions in that array. Also be aware that, since you are trying to keep track of the “smallest” String here, that you will need to use **compareTo**() in this modified version of the method.

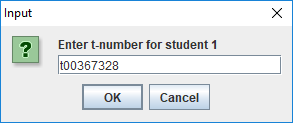
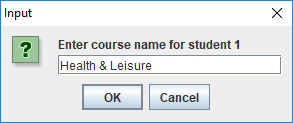
Once the arrays are sorted, your program should then display the content of the arrays again, to verify the sort has been successful. Next, the user should be prompted for the t-number of a student and then you should try to write the **binarySearch**() method from scratch and use it to search the t-numbers array for the t-number entered, with the user getting issued with an appropriate message back in main() depending on whether or not the t-number was found. If the t-number was found, the message should also display the corresponding course name for the student - to get this information, ideally you will **modify** the binarySearch() method so that, rather than giving back a simple boolean true or false result as it does now, it will instead give back the subscript number of the slot in the array that contains the search value (if it is found) or -1 otherwise (to indicate it was not found). Once that subscript value is known, it can be used to immediately access the corresponding slot in the sorted course names array.

Finally, the program should determine the following information:

* The number of students whose t-number begins with “t003”
* The number of students whose course name is either “Health & Leisure” or “Chemistry”
* The percentage of students taking the “Computing” course whose t-number is from “t00036” onwards (use **compareTo**() here)
* A neatly formatted list of all the t-numbers, and their corresponding course name, for all students taking the “Mechtronics”, “Early Childcare” or “Computing” courses

All the statistical information in the last part here should be displayed on the same text-area.

Your program will run similar to the following (take values from 1st message dialog below when testing)

  …. rest of input omitted for brevity … 