April 2, 2018

Sean Powell | 77611A

Data Structures and Algorithms

ICS 1018

Table of Contents

[Question 1: 2](#_Toc510389683)

[Question 2: 2](#_Toc510389684)

[Question 3: 3](#_Toc510389685)

[Optimization: 3](#_Toc510389686)

[Question 4: 4](#_Toc510389687)

[Question 5: 5](#_Toc510389688)

[Question 6: 5](#_Toc510389689)

[Question 7: 5](#_Toc510389690)

[Question 8: 6](#_Toc510389691)

[Question 9: 7](#_Toc510389692)

[Run Class: 7](#_Toc510389693)

Should there be any issue accessing the code that accompanied this document you can find a copy of the code on https://github.com/Sean-Powell/DataStructureAndAlgorithmsAssignment

# Question 1:

In question one a list from 1 to 1024 was taken. First using the properties of the multiplication matrix and the fact that after the leading diagonal the numbers are repeated. So, the number of numbers can be reduced from 1024^2 which is 1,048,576 to the summation of 1 to 1023 which has a value of 523,776. This simple optimization manages to remove 524,800 numbers which is over half the search space.

An object called a multinumber was defined this object stores the 2 factors of the number and the product of the number. A list is created using the above defined optimization. This list is then sorted using merge sort (n log n) using the product as the factor to be sorted by. This groups all the products that are the same together, thus allowing us to only search this block of numbers to find the different factors that can make up that product. This allows us to search a vastly smaller search space than if we had to search the entire list every time.

This can be achieved by after obtaining the sorted tree first we find the index that the block of new products starts at, then find the index where the last number of that block is found. The space in between these two indexes are then searched for the different factors that can be used to get to that product. If a match is found the result is outputted to the console. When the block is being searched the search is only on the indexes in the block that are greater than itself as if the elements index is less than the element being checked currently then this comparison has already been made when the block with the lower index was being checked.

# Question 2:

Question two first prompts the user to enter an equation in RPN format. The user inputted string the elements in the string are separated via a space between each element. If the element is a number, then it is added to the stack. If the element is a +, -, \* or / then the last 2 elements on the stack are popped off. If the operation is a subtraction or a division then the elements are switched. The respective operation is then preformed on the two element the result is then added back to the stack. This is repeated until the end of the user inputted expression is reached. If at any point there are not enough elements on the stack to preform an operation or there is more than one number on the stack at the end of user inputted equation being process, then the equation is said to be invalid. Otherwise at the end of the expression the answer is displayed, and the equation is said to be valid. At every step of the process the state of the stack is outputted to the console.

# Question 3:

The fact that if a number is a factor of the number that we are checking to be prime will be equal to or less than the square root allows us to only check up to and including the square root to further optimize this function to find a prime number.

First the user is prompted to enter either 1 or 2. If the user enters 1 then the method that checks if a number is prime or not. It does this by first calculating the square root of the number then rounds the number up if it Is not an integer. After this it looks through all the numbers from 2 up to the square root checking if the numbers are a factor of the number being checked. After it gets to the number 11 it uses the optimization defined below to further improve the efficiency of the function. If it finds that none of the number up to and including the square root is a factor, then is returns true. If it does locate a factor, then it returns false.

## Optimization:

After it gets to the number 11 its starts alternating in adding 2,3,2 to the number. This is because after the number 11, all the even numbers and the multiples of 5 can be skipped. This is achieved by using a size 3 array holding 1,2,1. The code that looped through this is shown below, the j is incremented outside the method itself. When j == 3 it is reset back to 0 to stop an array out of bounds exception.

|  |
| --- |
| **private int[] additionAmount = {1, 2, 1};**  **private int addAmount(){**  **if(j == 3){**  **j = 0;**  **}**  **return additionAmount[j];**  **}** |

If the user inputs a 2 then the program executes the sieve of Eratosthenes. This method first creates a scanner object reading from System.in it then prompts the user on what number they want to run the sieve up to. This will loop up until the user enters a number that is greater than 0. The square root is then calculated and rounded up in the case that it is not an integer. The method then populates a list of numbers from 2 all the way up until n. This also uses the addAmount method as outlined above to reduce the numbers in the list. The method then loops through from 2 through till the number it retrieves from the list is greater than the square root. If it is less then it goes through the list checking if that number is a factor, of any number in the list. If it is a factor, then the number that it is a factor of it will be removed. After it has gone through all the numbers less than the square root it then prints the list with 4 numbers on every line. This printing is in a try catch block so that if there are not 4 numbers in the list left to be printed it won't crash and will only print the amount that is left and will create a new line at the end.

There is also the check that if the current index is >= to the length of the list then it will exit the loop. This stops the program from suffering from an out of bounds exception in the case that the user inputted the upper limit of 2 for the sieve. Since it the list starts at 0 the last element in the list will be 1. However, the square root of 2 is still 2 so this would cause the function to try access index 2 of the array, thus causing the exception.

# Question 4:

The user is first prompted to enter either 1 to add numbers to the Binary Search Tree(BST) or 2 to search the BST to see if the user inputted number can be found within the tree.

If 1 is selected the program prompts the user to input a number, the first number the user inputs will serve as the root node for the BST. After this each time the user enters a new number, the BST will be traversed, and a new node is created for the new number and inputted in the correct location. If the number is greater than equal to the number currently stored in the node it will traverse to the right otherwise it will traverse to the left. This will loop until the user enters a none integer value to exit the input loop.

If the user inputs 2 the program will prompt the user to input an integer and will then traverse through the tree checking each node if the data stored within it is a match to the data value being searched for. This is managed in a recursive manner with the following function.

|  |
| --- |
| **private boolean findNode(int \_input, Node \_nodeBeingChecked){**  **if(\_nodeBeingChecked == null){//checks if node being checked is equal to null**  **return false;**  **}else if(\_input == \_nodeBeingChecked.getData()){**  **//checks if the nodes data is equal to the search**  **return true;**  **}else{**  **//the traversals also check if the node that is being traversed to are equal to null.**  **//If they are the data is not in the tree.**  **if(\_nodeBeingChecked.getData() >= \_input) { //traverse to the right**  **return \_nodeBeingChecked.getRightNode() != null &&**  **findNode(\_input, \_nodeBeingChecked.getRightNode());**  **}else{//otherwise it will traverse to the left**  **return \_nodeBeingChecked.getLeftNode() != null &&**  **findNode(\_input, \_nodeBeingChecked.getLeftNode());**  **}**  **}**  **}** |

The node is defined as follows, A Node rightNode and a Node leftNode both storing the respective object for both the left and right node this is set to null if there is no node. There is also the Data element that holds the integer value of that node in the BST.

When the user finally enters -1 to exit the question the tree is displayed using the output:

“Data: X Left Node: Y Right Node Z”

With X being the data from the current node, Y being the data from the left node and Z being the data from the right node. This method uses recursion to display the whole tree. Y and Z being set within the method if there is a node in that location if not the method will output null instead of a data value.

# Question 5:

This uses the Babylonian method for computing the square root of a number. The user is prompted to input the number that they want the square root to be approximated for. They are then further prompted to input the degrees of accuracy they want to achieve. The Babylonian method is then started with this data, first the square root is guessed to be half the original number. The same line is then looped to the number of iterations as required by the user this uses the following code.

|  |
| --- |
| **for(int i = 0; i < \_iterations; i++){**  **ans = (ans + (\_number / ans)) \* 0.5;**  **}** |

After this loop is completed the final answer is returned and then it is outputted to the user.

# Question 6:

This function takes a list of integers to be checked for duplicate numbers. This is achieved by first hashing the integer using SHA-256 this is then checked to see if there is a collision in the list of hashes. If there Is a collision the integer that was hashed to obtain this hash is added to the list of duplicate numbers otherwise the hash is added to the list of hashes. Finally, the list of duplicate numbers is outputted.

Hashing is used since the list of integers does not change then in this case the hashing function can be considered O(1). With the overall efficiency of this method being O(nk) with n size being the size of the list of integers to be inputted and k being the number of unique numbers in the list.

# Question 7:

This function first takes an input ArrayList of integers into the function then recursively finds the largest number. It does this by first checking the length of the ArrayList that is passed to the method, if the list is of size 1 then then number is returned. If it is not then the first number is stored in num1 then it is removed from the list, the new list is then passed recursively to the method. This repeats until the above base case is reached. Once it is reached the number returned from the recursive method is stored in num2. Num1 and num2 are then compared and the largest integer out of them is returned. In the case that the numbers are the same then num2 is returned. Eventually when all the recursive calls have been handled the original method that was called will return the largest number in the list that was passed to it.

# Question 8:

This method requires the user to input the degree of the angle in radians, then the user is requested to input the degree of accuracy they would like to calculate. Finally, the user is prompted on if they want to run the cosine or sine expansion. The program then calculates either cosine or sine respectively with the following code \_radians being the angle and \_degrees being the degrees of accuracy required. \_degrees being the degrees of accuracy requested by the user. The answer in the form of a double is returned to the user.

|  |
| --- |
| double cosExpansion(double \_radians, int \_degrees){  double ans = 0;  for(int i = 0; i < \_degrees; i++){  if(i == 0){  ans = 1;  }else {  ans += (Math.pow(-1, i) \* Math.pow(\_radians, 2 \* i)) / factIt(2 \* i);  }  }  return ans;  }  double sinExpansion(double \_radians, int \_degrees){  double ans = 0;  for(int i = 0; i < \_degrees; i++){  if(i == 0){  ans = \_radians;  }else{  ans += (Math.pow(-1, i) \* Math.pow(\_radians, (2 \* i) + 1) / factIt((2 \* i) + 1));  }  }  return ans;  } |

factIt is an iterative method for calculating the factorial of a number as shown below.

|  |
| --- |
| private long factIt(int x){  long ans = 1;  while(x >= 1){  ans = ans \* x;  x--;  }  return ans;  } |

# Question 9:

This question prompts the user to input the degree that they wish to calculate up to. The method is done using both the BigInteger and an iterative approach to increase the degrees the program can calculate too. The BigInteger stops the overflow error from happening and the iterative approach to the method makes it so the method does not use too much memory during the calculation.

The method works such that both the current Fibonacci number and the previous one is set to 1 at the start. The current one is set equal to a temp variable and then the old one is added to the new one and the old one is set to the value store in the temp variable. This loops for n-2 times with n being the amount of degrees of accuracy required with as the first two degrees are 1. After the method has calculated the number the number is then outputted to the user.

# Run Class:

The run class that is run when the program is first called it has a while loop that runs until the user inputs -1, otherwise it will prompt the user which of the above questions they want to run prompting them to input from 1 to 9. If the user inputs anything other than this the program will notify the user that they have inputted an invalid input. This class then manages some user input and the calling of the different classes to run the respective questions. The class also contains the method responsible for allowing string input.