Tristan Oa Galea

B.s.c. in computer science

Data Structures & Algorithms Documentation

ICT 1018

Table of Contents

Task 1 pg. 3

Task 2 pg. 14

Task 3 pg. 19

Task 4 pg. 27

Task 5 pg. 33

Task 6 pg. 37

Task 7 pg. 41

Task 8 pg. 46

Task 9 pg. 50

Task 10 pg. 57

Task 11 pg. 62

Task 12 pg. 66

Foreward

This assignment was programmed using the Python programming language. The documentation for this assignment is individually split up based on each task.

Each task explanation is divided into 3 sections: (i) Task explanation, (ii) Actual source code and (iii) Code testing. For each task, (i) a brief overview of what has been achieved in the task performed is discussed. Following this, (ii) the actual python code is shown. Finally, (iii) the testing of this code is explained with the help of screenshots.

Each task explanation gives a brief overview of what has been achieved in the task performed. A more detailed explanation of sections of code can be found in the comments included in the pasted source code.

Task 1 Explanation:

In this first task, 2 empty lists were created. The user was then asked to enter a value for the desired length of the first list. A function called ‘user\_input()’ was created to accept this user input and validate it using try and except statements in the following ways:

* The function ensured that only integer values could be stored. This was done so as to make sure that the user enters valid data. If the user enters any other data type such as a character, he/she is informed and prompted to re-enter a length for the list without causing the program to crash.
* A further validation performed was that of assuring that the user enters a length greater than or equal to 256, as specified.

After the user successfully enters a value which satisfies both validation techniques, another function ‘populate\_array()’ is called which populates the list randomly with integers from a range of 0-1024 up to the length entered by the user. The randomly generated list is then displayed to the user.

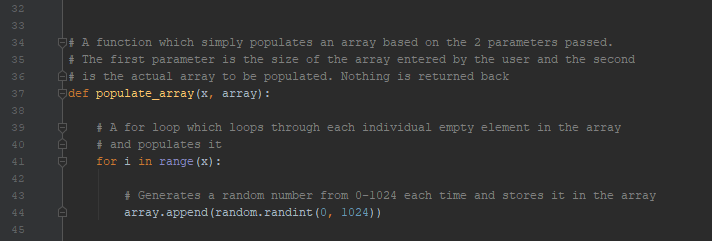
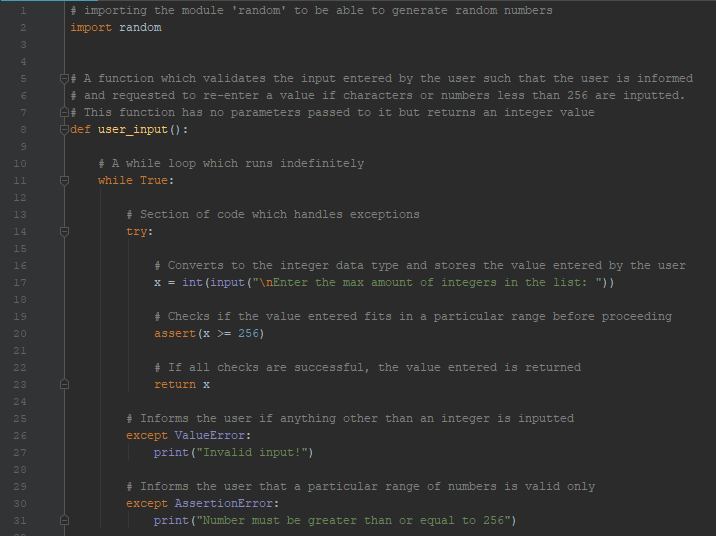
This same procedure is repeated for array B, this time, with an extra validation method involved. This consists of a while loop which was implemented to ensure that the length of the second list, upon passing both previous validation tests, is not equal to that of the first. This was done to satisfy the question requirements.

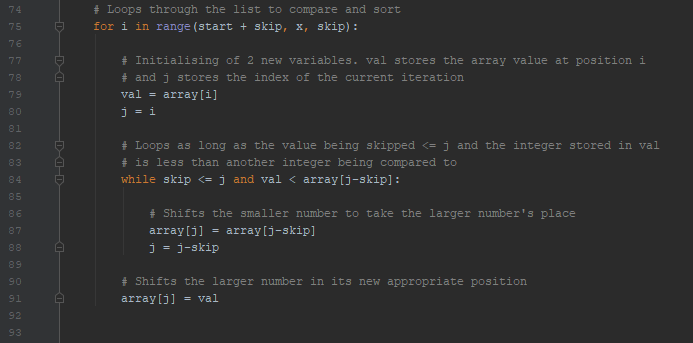
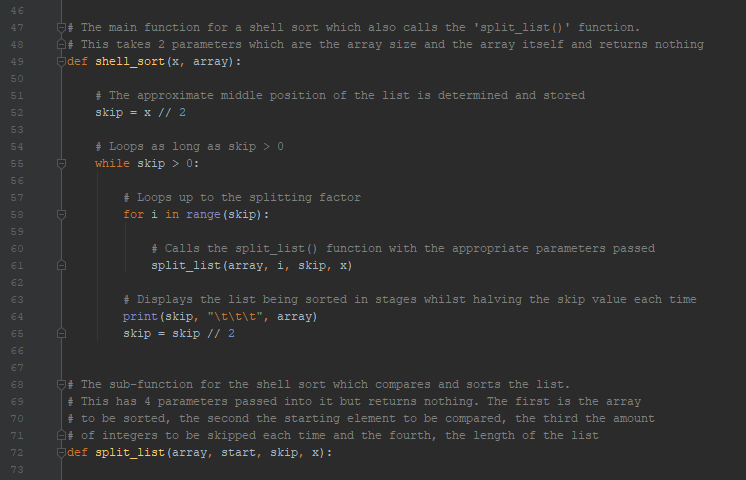
After having successfully created, populated and printed out the arrays, the 2 lists are then passed through 2 separate functions to be sorted. The ‘shell\_sort()’ function is called to sort ‘array\_a’ and the user is informed of the gap size and an updated version of the list based on the sorting of that gap size. The final version of the list, having a gap size of 1 is displayed as the last iteration.

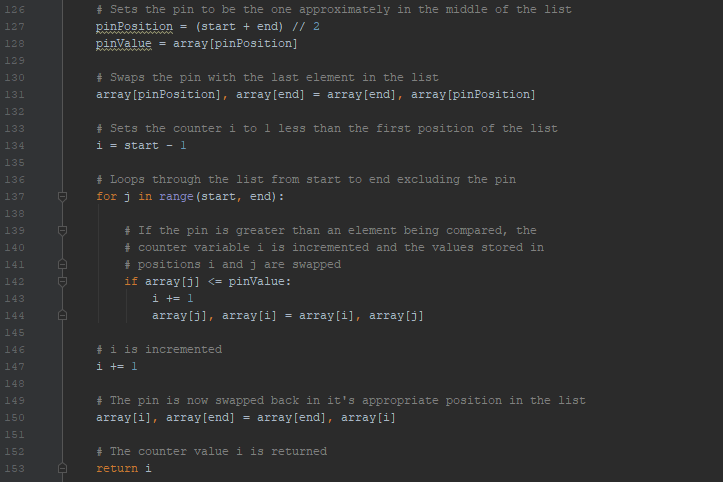
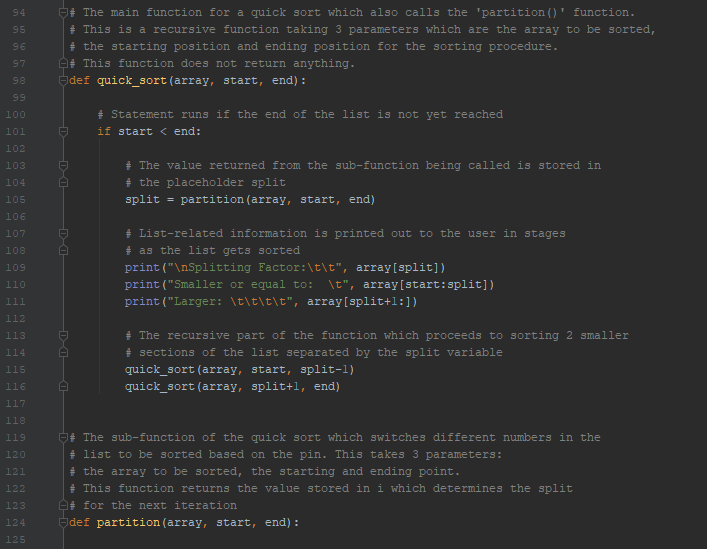
Moving on, the ‘quick\_sort()’ function is called to sort ‘array\_b’. With every iteration, the user is informed of the splitting factor which is set to be approximately the element in the middle of the list. The lists smaller and greater than the number are also displayed to show the user the new lists on which the ‘quick\_sort()’ function is going to be called again. These 3 qualities are displayed to the user with each iteration in a random manner. This means that the user might encounter the splitting of a particular list at a far later stage in the sorting algorithm (ie a newly split list might be split again and displayed to the user after a couple of different iterations and not exactly after). Although this may be confusing, nothing could be done to ensure that the lists split up one following the other. Finally, the sorted ‘array\_b’ is printed out to the user.

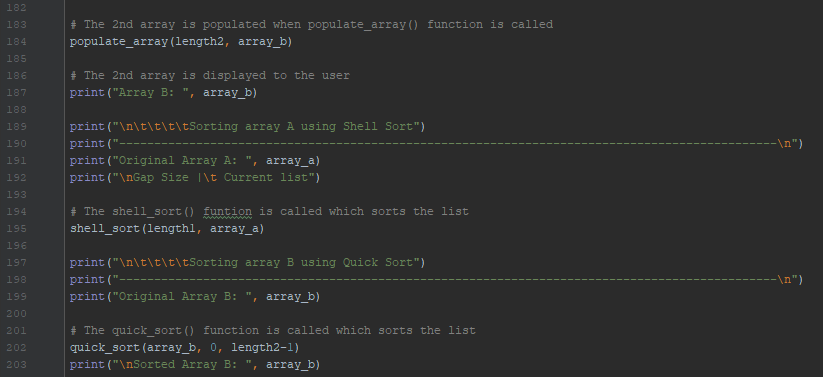
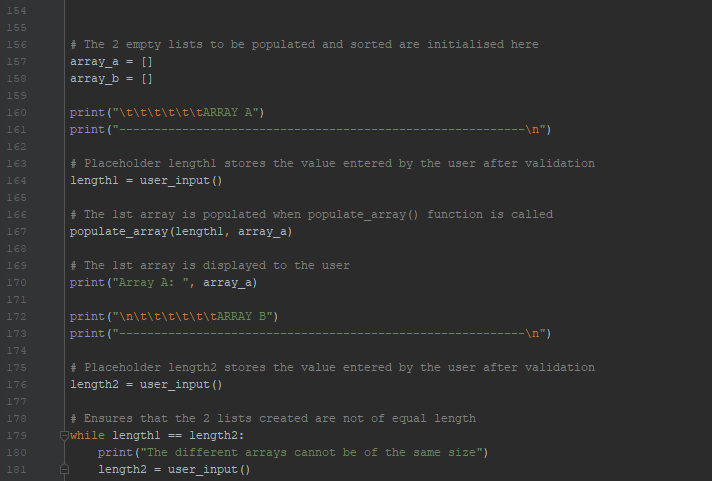
Task 1 was successfully implemented and fit all the specified requirements without having any bugs.

Task 1 Code:



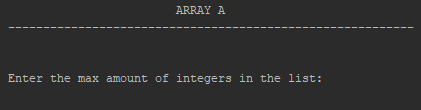






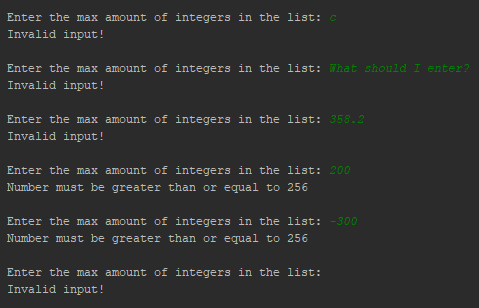
Task 1 Testing:

Upon running the program, the user is asked to enter the maximum amount of integers in the first list:



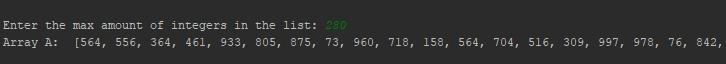
Assuming the user misunderstands what is required of him and instead enters:

1. A character
2. Multiple characters
3. A float
4. A number less than 256
5. A negative number
6. ‘Enter’ key

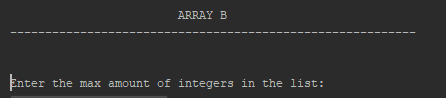


For each of the above incorrect instances, the program handles the exception correctly and keeps waiting for an appropriate integer value before proceeding.

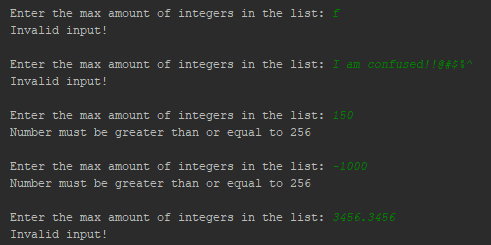
Considering now that the user understands what is expected of him, the value entered is successfully stored and a list of size as specified by user is generated. The value the user enters is 280 and a list of 280 elements is randomly generated. Below, a fraction of the list created is displayed.



Now that list A has been successfully populated, list B is now left to be created. Like before, the user is asked once again to enter a length for this list.



Assuming the user still does not understand what is expected of him and enters garbage data, the program continues persisting for the correct data as shown.



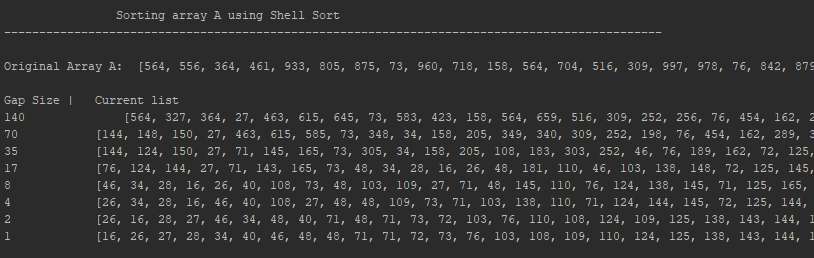
Given that the user now enters a value in the range specified (ie a value not less than 256) but equal to the length of the previous list, the program prints on the screen the following:



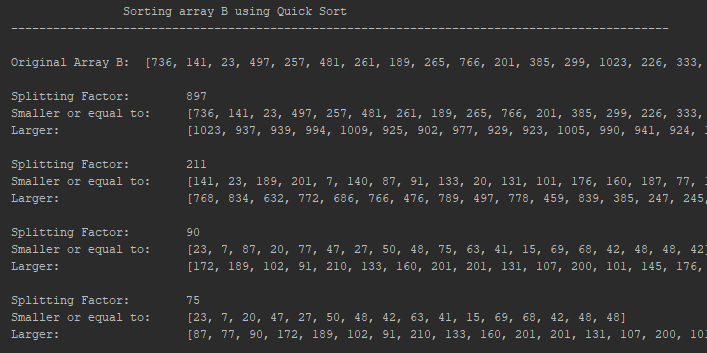
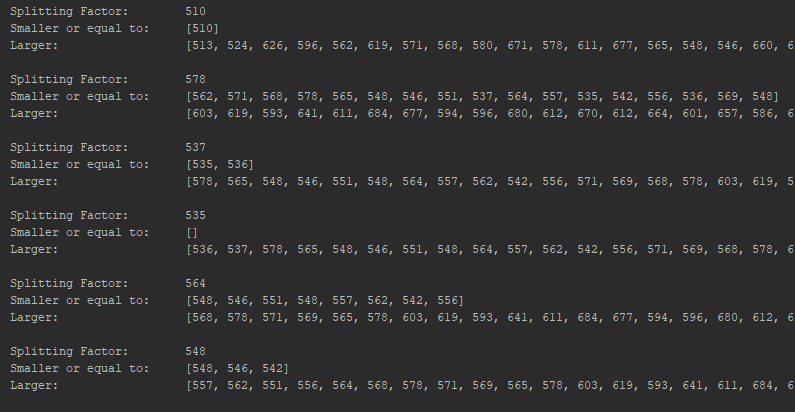
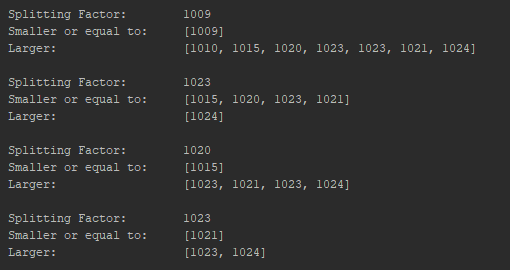
The user finally understands and this time enters the value 300. This data is accepted and the program generates 300 random elements which are stored in list B. This list is then displayed to the user.



From this point onwards, the program sorts list A using Shell Sort and list B using Quick Sort as can be observed. Due to the large amount of elements in the list, only a fraction of each list being sorted is shown in the screenshots.



The screenshot shown above displays all the gap sizes required for the sorting of the list A with the final sorted list as the last line. In the screenshots below we can see parts of the sorting process for list B. Quick Sort, being much more demanding than Shell Sort, requires more steps for the list to be sorted. For this reason, only 3 random samples of this process are shown in the next 3 screenshots:

Finally, the sorted list B is displayed upon completion:



Task 2 Explanation:

After completing and testing task 1, it was time to move on to task 2. In this task, all the contents of task 1 were imported at the very start so that they could be included rather than having to code everything again. Therefore when running the program, 2 empty arrays are initialized, populated and sorted from the start.

In this task, only one function was declared. This is called the ‘merge()’ function which initializes an ‘array\_c’ with exactly the length of ‘array\_a’ and ‘array\_b’. For this to be done, all the elements in list C has to be set to ‘NONE’ initially.

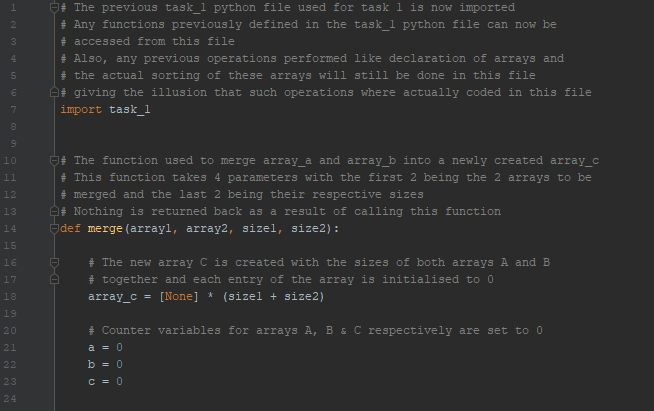
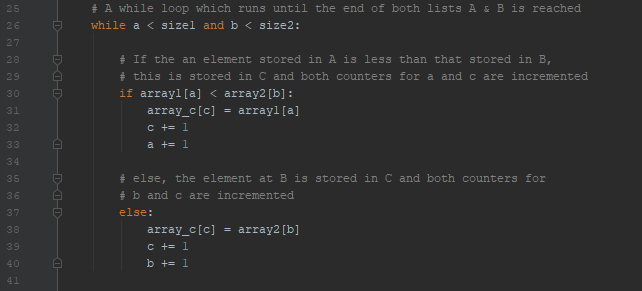
Moving on, lists A and B would be compared when passed as the 1st and 2nd parameters to the ‘merge()’ function. The smaller element out of the elements being compared would be added to position ‘c’ in the ‘array\_c’. The counters of list C and of the list whose value has been added to list C are then incremented by 1 to point to the next position in both lists. This process would repeat until the end of both lists A and B is reached.

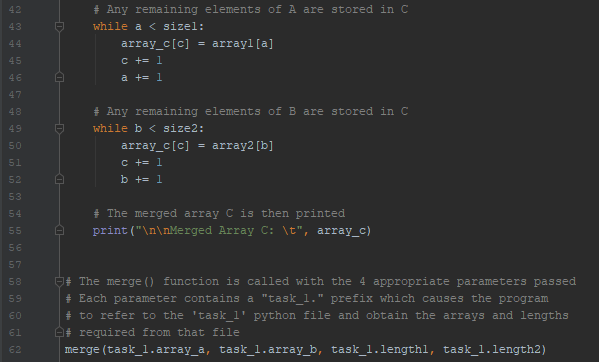
This function was originally coded as is and the list was printed out at the end of the merging. However, when inspecting the new merged list C, it was evident that the last few elements were still set as ‘None’. This implied that not all the elements from either list A or B had been added. To fix this, 2 separate while loops, one for A and another for B were added to append any remaining elements in the previous lists to the new one. ‘array\_c’ would then be printed out after these 2 loops as the last line in this function.

When calling the ‘merge()’ function, the 2 lists passed and their respective lengths had to include a ‘task\_1.’ prefix to indicate that the lists and variables passed were stored in the external ‘task\_1.py’ file.

Task 2 was successfully implemented and fit all the specified requirements without having any bugs.

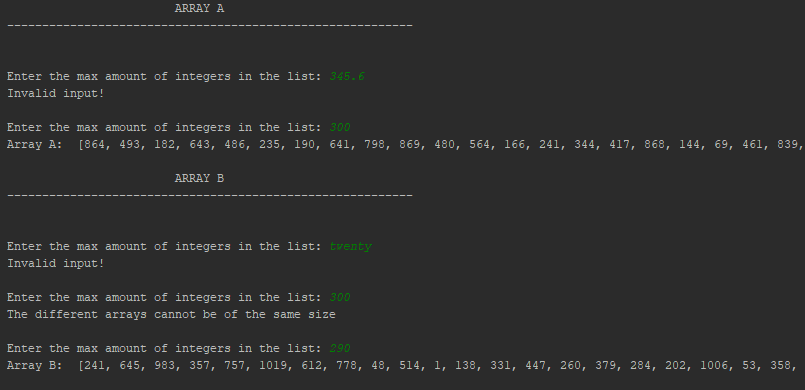
Task 2 Code:

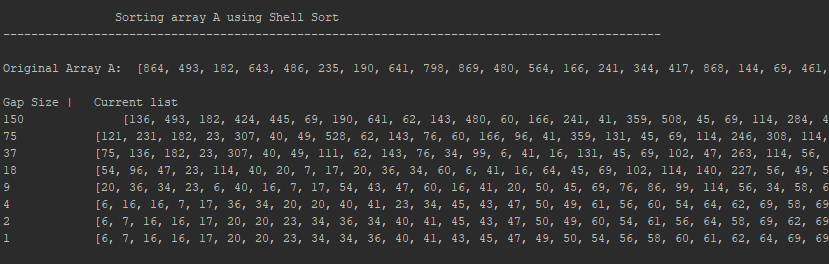


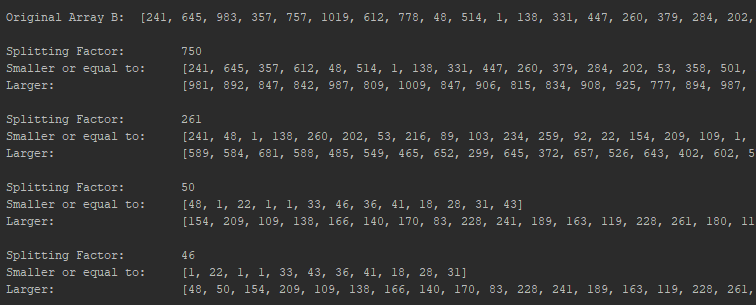
Task 2 Testing:

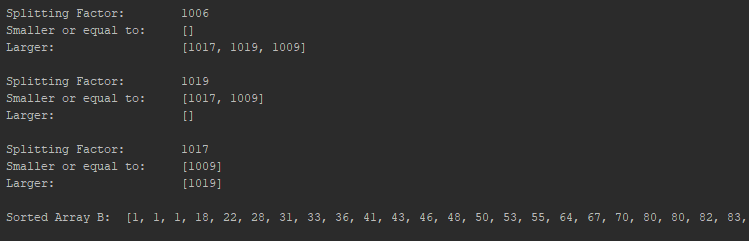
When ‘task\_2’ is run, the contents of ‘task\_1’ are evaluated first due to the ‘import’ preprocessor command. Thus the user has to enter valid values for the lengths of lists A and B respectively. The user input entered below shows incorrect followed by valid input:

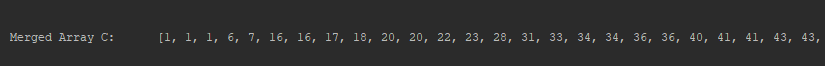


‘array\_a’ was assigned a length of 300 elements whilst that of ‘array\_b’ was 290. Parts of the 2 randomly generated lists can also be seen in the above screen dump. These lists were then sorted as before and the final merged ‘array\_c’ is displayed to the user:









Task 3 Explanation:

In this task, it was required to create a function which identifies the extreme points of a given array.

To start off, an empty ‘arrayA’ was initialized. The user was then asked to enter the length the list would have as well as the min and max values that could be present when calling the ‘user\_input()’ function. This function consisted of a modified version of the one previously defined in task\_1. It consisted of a try-except statement which would ensure that the user enters only integer values. If not, a warning message is displayed allowing the user to enter the value again. Upon successfully entering the first value, a counter ‘i’ initially set to 1 would be incremented and the if statement which runs only when ‘I’ is 2 is run. The user is now asked to enter the min value present in the list. When appropriate data is entered, ‘i’ is incremented once again and the max value present in the list is requested from the user.

These values are then returned back and stored in placeholders to be passed as parameters (including the actual ‘arrayA’ as the 4th argument) to another function which populates the list with randomly generated integers. This is the ‘generate\_list()’ function which works on a similar concept to that of task\_1 with the only difference of having variables entered by the user determine the min and max values in the list.

After the list is generated, it is displayed to the user. Following this, the newly created ‘extreme()’ function (with the array to be tested and its length as parameters) is called which simply determines the extreme points of the list.

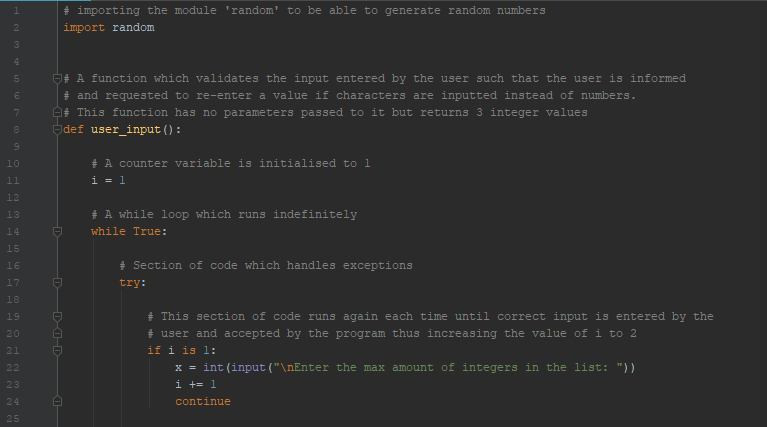
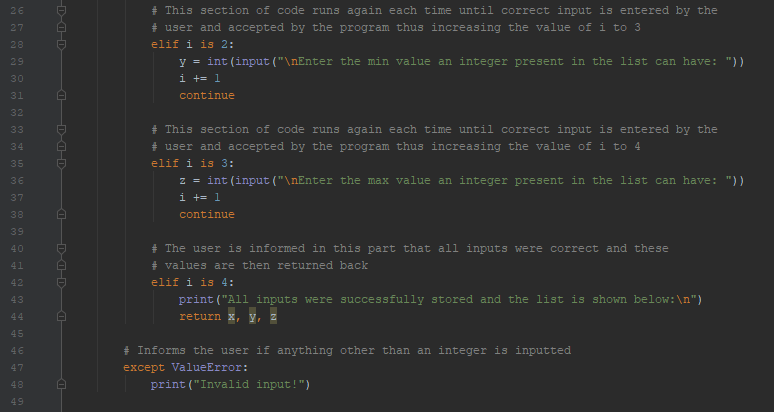
In this function, an array ‘points’, to store the extreme points of ‘arrayA’ is defined. Then a for loop, which loops through all the possible extreme points (excluding the first and the last elements) checks for these points and appends them to the ‘points’ list. Following this is another if statement, which lets the user know that a list is sorted if no extreme points are found. If not, the extreme points found are displayed to the user.

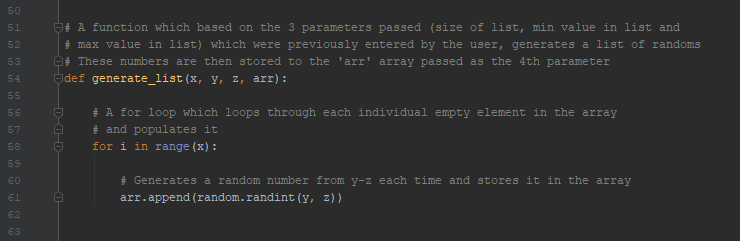
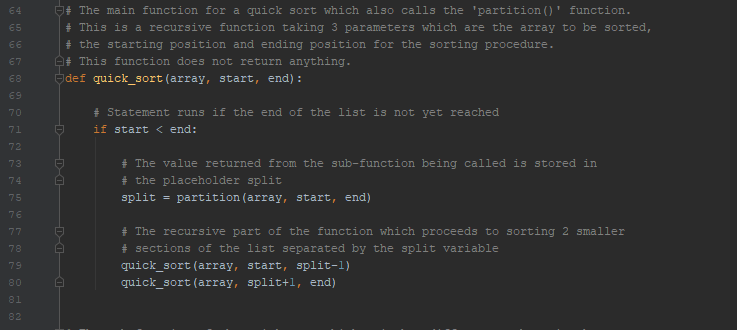
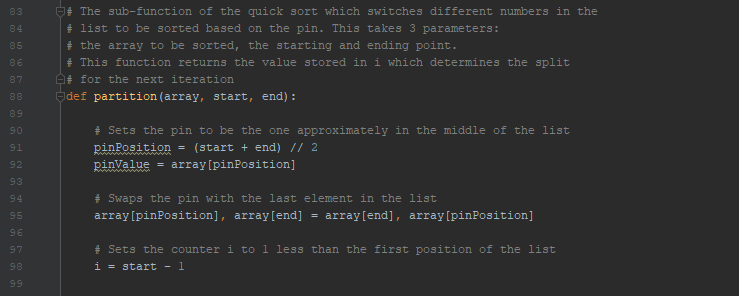
To test if a sorted array has any extreme points, a ‘quick\_sort()’ function was created, identical to the one in task 1. The ‘arrayA’ list was then sorted using this function and displayed to the user in ascending order. This newly sorted array was then passed through the ‘extreme()’ function and as expected, the array had no extreme points. This was because the elements were now one after the other in ascending order and neither

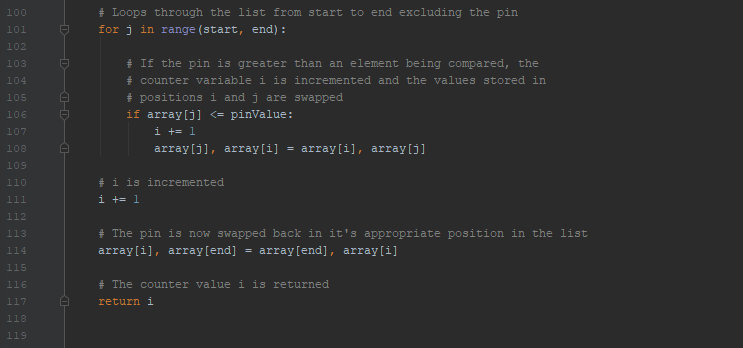
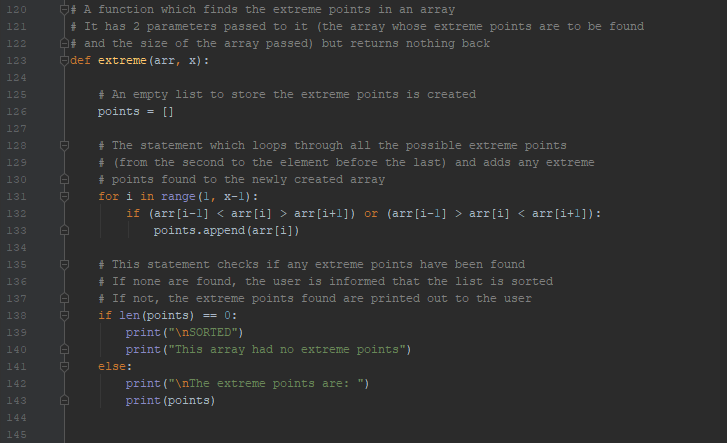
arr[i-1] < arr[i] > arr[i+1] nor arr[i-1] > arr[i] < arr[i+1]

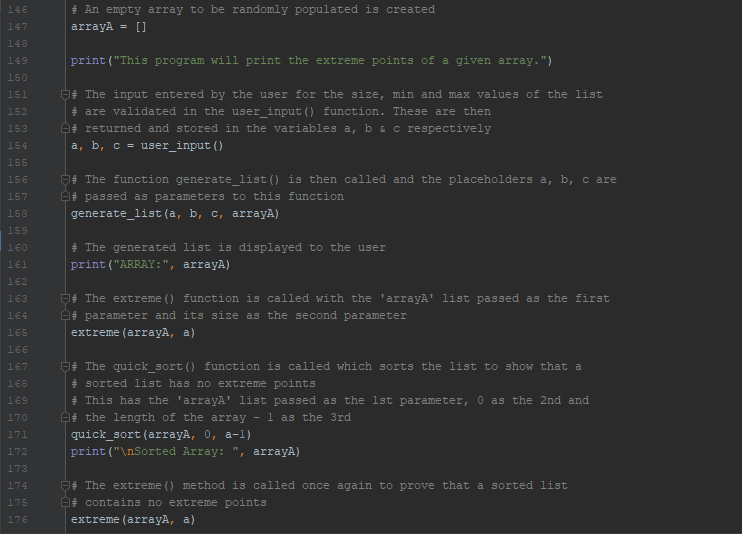
Task 3 was successfully implemented and fit all the specified requirements without having any bugs.

Task 3 Code:

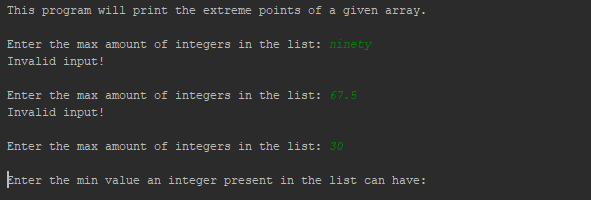
  

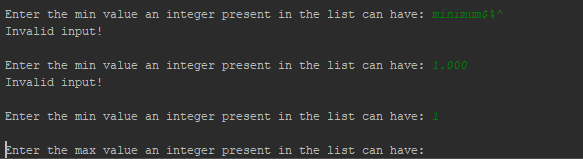


Task 3 Testing:

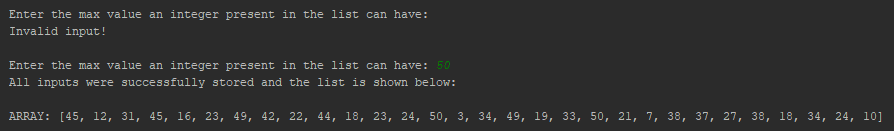
Upon running the program, the user is requested to enter the length of the list he/she wants to create. Consider 2 invalid input attempts are entered before a correct value is entered and stored.



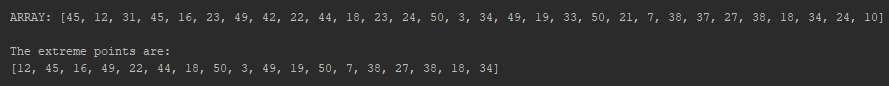
When a valid value is entered, this is stored and the program asks for the minimum value possible to be present in the list to be generated. Consider the user enters a further 2 invalid inputs before entering a 3rd successful one.



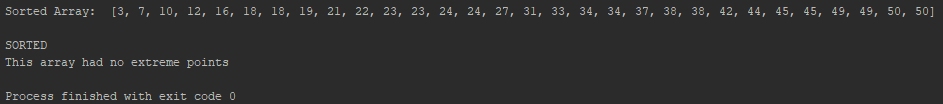
Moving on, the value ‘1’ is stored as the minimum possible value for the list to be generated. Now, the user is asked one last time to enter a max value to be present in the list. Consider the user hits ‘Enter’ without entering anything on the 1st attempt followed by a 2nd successful attempt. This leads to the list being generated based on the input fed by the user as shown:



From now on, the program runs till the end without requiring any further user input. Following the printed generated array is another array which displays the extreme points found.



As explained before, to fully understand if a sorted array has any extreme points or not, the array was then passed through a quick\_sort()’ function and the sorted result was displayed. This newly sorted array was then passed through the ‘extreme()’ function once again and as expected, no extreme points were found.



Task 4 Explanation:

In this task, a program was written to identify all 2-pairs of integers that have the same product given that the 4 multiplied are all different.

Initially, an empty ‘numbers’ list was instantiated. This would be populated based on a list size entered by the user. The user input for the list size was validated using the ‘user\_input()’ function. Just like task 1, this consisted of a try-except statement which would ensure that the user enters an integer value only. The only difference from the one implemented in task 1 is that the user has to enter a value greater or equal to 4. This was done to ensure that the list consisted of at least 4 elements, which was the minimum amount of elements that could be compared.

Moving on, the ‘user\_input()’ function returns the valid data entered and passes it as a parameter to the ‘populate\_array()’ function, along with the list to be populated. Once again, this function is identical to the one implemented in task 1 with the only difference that numbers are generated from 1-1024 instead of 0-1024. Then, the list is generated and printed out to the user.

The final function implemented in this task, which also happens to be the function which identifies all the 2-pairs of integers having the same product, is then called with the list size as the 1st parameter and the list as the 2nd.

In this function, a counter variable h is instantiated to 0 at the start. This would increment each time a 2-pair of integers is found. This was added because whenever an array passed had 0 pairs, nothing would be printed out on the screen. In this manner, the user would be informed at the end the amount of 2-pairs found, even when none are found.

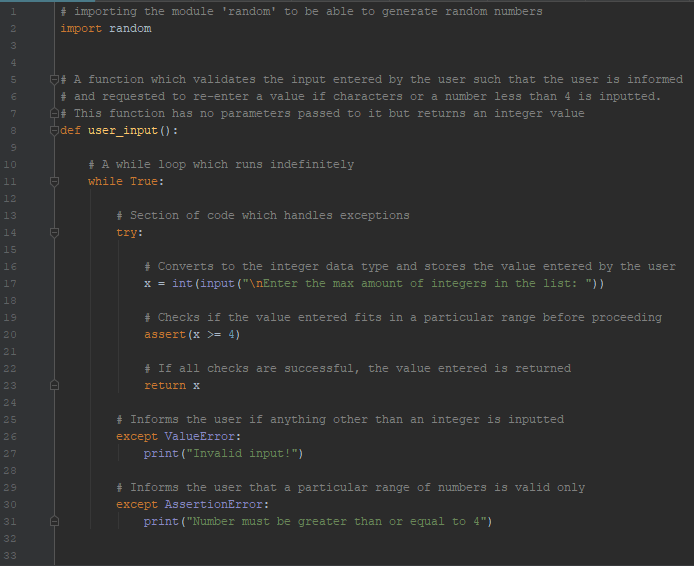
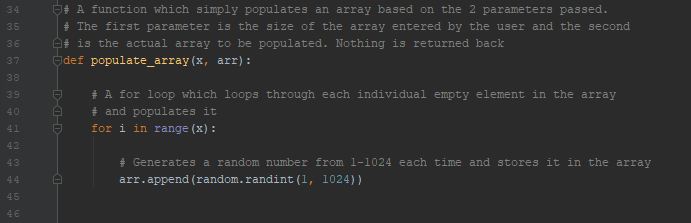
Implemented also in this function are 4 nested for loops which loop through each individual element in the list. At the core of these for loops is an if statement which actually identifies the 2-pairs of integers while at the same time making sure that arr[i] ≠ arr[j] ≠ arr[k] ≠ arr[l]. If any are found, these are printed out and the counter variable is incremented by 1 each time.

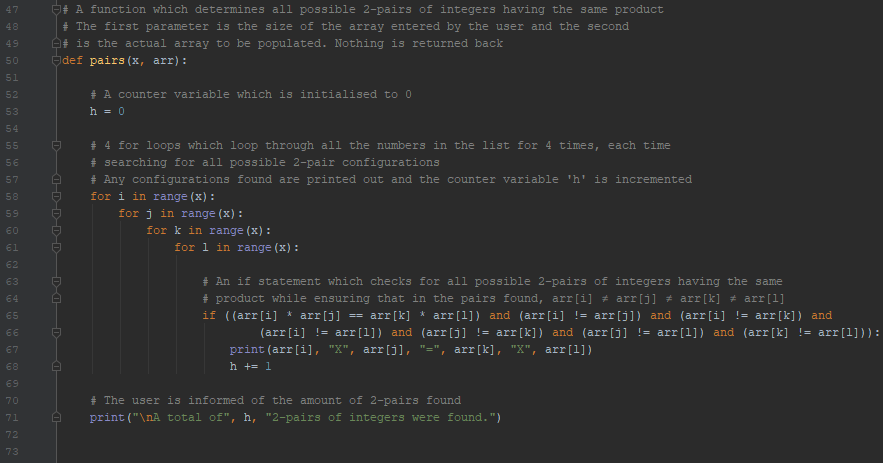
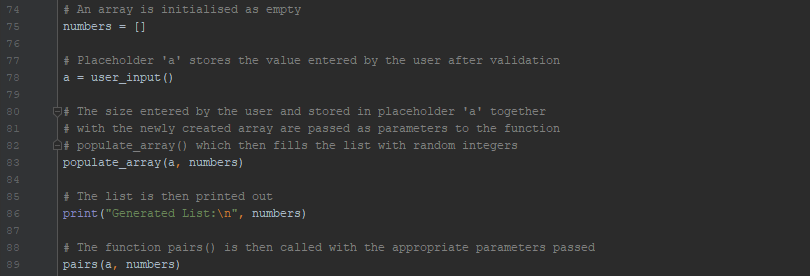
Finally, the user is informed of the total repeated pairs found at the end of the function and the program finishes execution.

One evident bug that showed up when testing this program was the fact that a 2-pair found is usually printed out 8 times. Once in the ‘a X b = c X d’ formation and another time in a swapped commutative form ‘a X b = d X c’. This process is then repeated for a and b aswell. Then, the program usually considers another case in which ‘a’ and ‘b’ are swapped with ‘c’ and ‘d’. This eventually leads to another 4 cases due to the commutative swapping already explained. This meant that the total amount of 2-pairs found was actually 8 times the actual amount, if these irrelevant pairs were omitted. So for 3 real 2-pairs found, the program would register 24 instead, by swapping the first and last 2 elements and also by swapping elements ‘a’ and ‘b’ with ‘c’ and ‘d’. Given the time allotted for completing this assignment, this was considered as a minor bug and was not further investigated to be fixed.

Apart from that, task 4 was successfully implemented and fit all the other specified requirements.

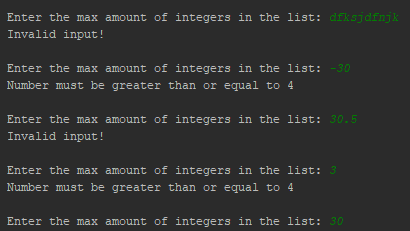
Task 4 Code:

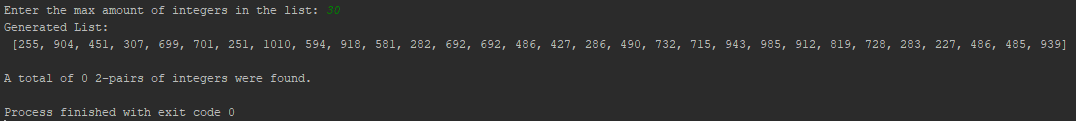
 

Task 4 Testing:

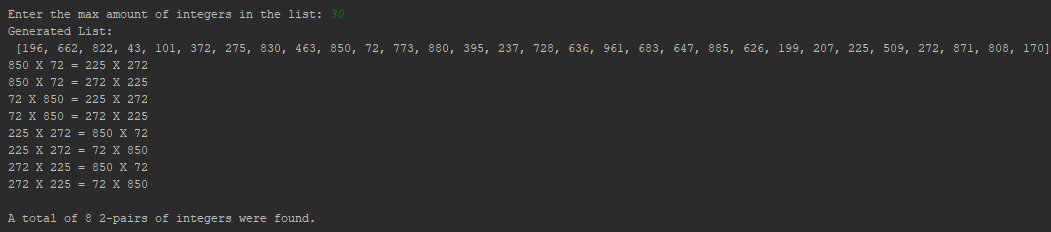
The user is first asked to enter a length for the list. Suppose the user enters 4 incorrect values followed by a valid one.



Upon entering 30, the list created is populated with 30 random elements from 1 to 1024. This list is then displayed to the user and checks for all possible 2-pairs of integers are performed. The result of the checks performed are shown below:



In this case, 0 2-pairs of integers were found in this randomly generated list containing 30 elements. Consider now generating another list of 30 random elements.



This time round, the program claims to have found 8 2-pairs of integers having the same product. However, as already discussed, the actual amount of distinct 2 pairs found is 1 because of a bug. From the printed 2-pairs, one can notice that the 7 other copies are the same as the first case.

Task 5 Explanation:

In task 5, a program that uses an ADT Stack to evaluate arithmetic expressions in RPN format was created. It contains the following arithmetic operations: +, -, x, and /.

A stack class was created which defined basic stack operations. This included the actual list being implemented as a stack defined in the constructor function, a push function, a pop function and a function which returns the string representation of the elements in the stack.

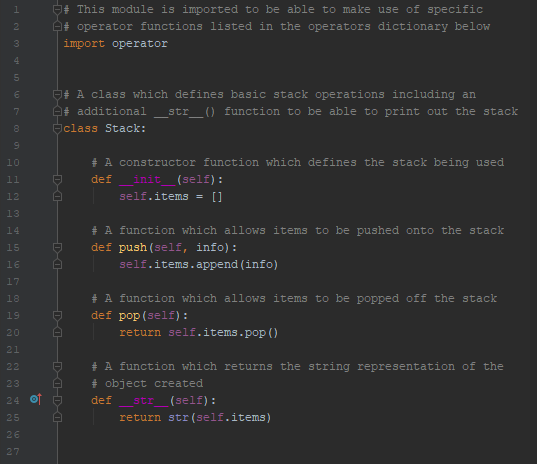
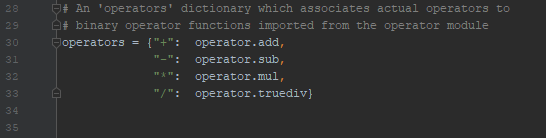
Next, the ‘operator’ module was imported to be able to use operator functions (operator.add, operator.sub, operator.mul & operator.truediv). These were placed in a dictionary and the operator symbols were assigned to them.

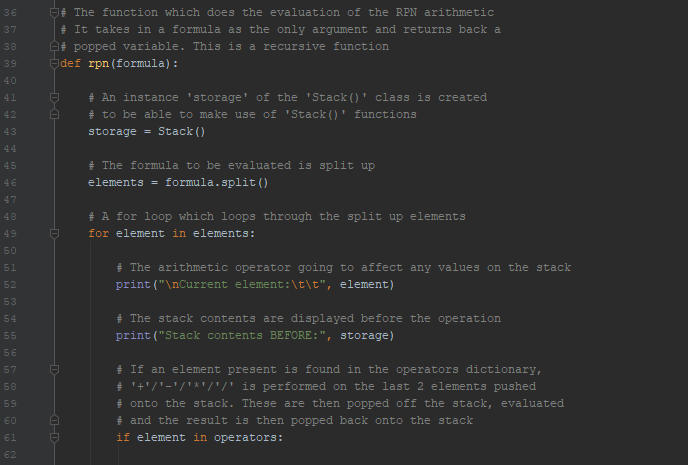
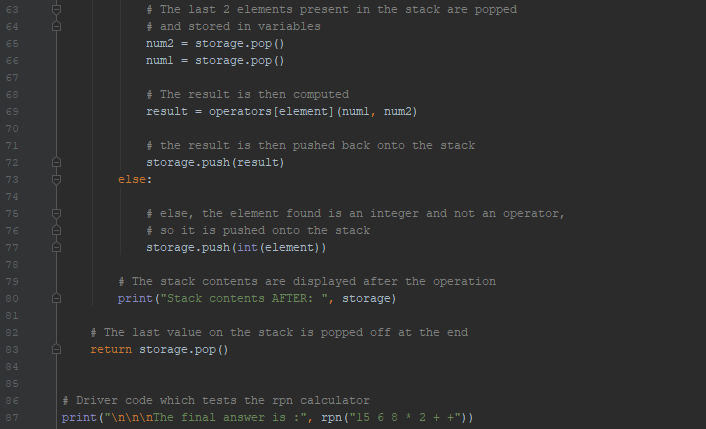
Moving on, an ‘rpn()’ function was defined to be able compute mathematical functions in rpn notation. In this function, a stack object called ‘storage’ was created. An array elements was initialized to store the separate formula elements. These were then traversed one after the other, pushing the integers onto the stack each time one is encountered. If an operator is encountered, the 2 last pushed elements are popped off and the expression is evaluated, pushing the answer back onto the stack.

The user is informed at every step of the process of what is going on. When all the expression is evaluated, the answer is displayed to the user.

Task 5 was successfully implemented and fit all the specified requirements without having any bugs.

Task 5 Code:

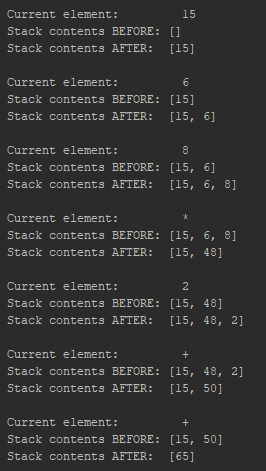
Task 5 Testing:

The following expression was to be evaluated: (6 x 8) + (15 + 2)

This was converted into RPN form and included in the driver code of task 5.



When the user runs the program, the expression is evaluated and onscreen updates are given to the user to inform him of the calculation’s current status.



The final answer is displayed in the end:



Task 6 Explanation:

In this task, a program was written which check if a given number is prime. Following this, the Sieve of Eratosthenes Algorithm was implemented.

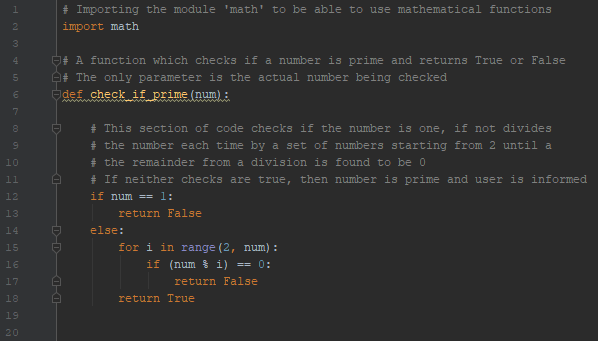
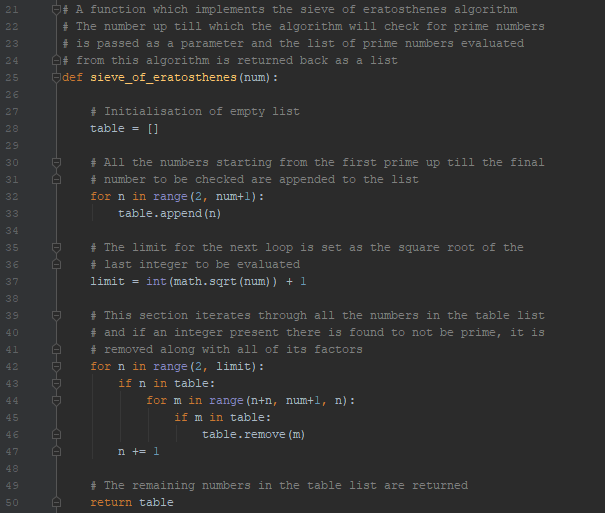
A function was created to determine if an entered number was prime. In this function, the number being tested was passed as the only parameter and an if statement was implemented to check if the number was 1. If so, False is returned as 1 is not prime. If not, the else branch contains a for loop which loops from 2 up to the number, checking if the remainder of anything divided by the number gives a 0. If so a direct divisor and the number is not prime, thus returning false once again. But if all the numbers are traversed and not a single direct divisor is found, then True is returned because the number is prime.

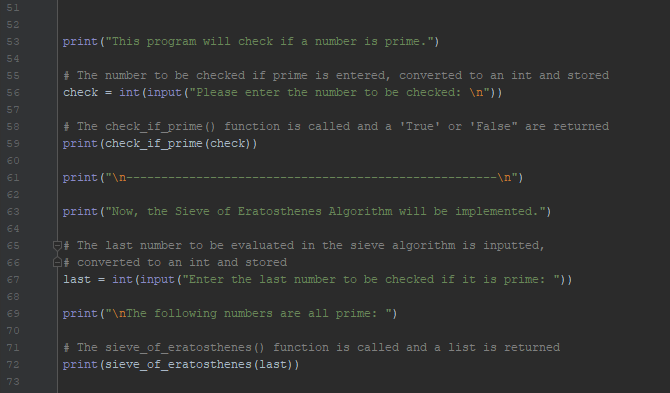
The Sieve of Eratosthenes Algorithm was implemented next. This function consisted of an empty list which would be filled with numbers from 2 up till the number being entered by the user to be tested as the last element. Then, a for loop which traverses from 2 up till the square root of the number entered, loops through all the numbers while eliminating any non-prime numbers and their factors from the list. The remaining numbers in the list (ie prime factors) are then shown to the user.

An optimization made to the Sieve of Eratosthenes function was that when checking for non-prime numbers to be removed from the list, the list was traversed from 2 up till the square root of the number entered rather than looping up till the number entered. This was done because if a number found is not prime, then all it’s factors are not prime by default. Thus, the loop went on till the square root of the number entered and to compensate for this change, all remaining factors of numbers removed from the list were also removed rather than having to traverse all the list and simply removing a tested number.

Task 6 was successfully implemented and fit all the specified requirements without having any bugs.

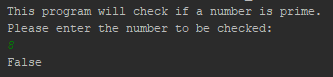
Task 6 Code:

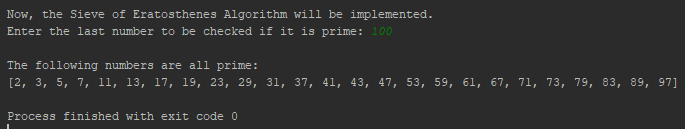


Task 6 Testing:

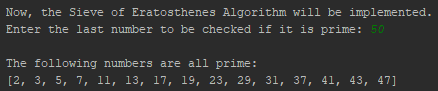
When running task 6, the user is asked to enter a number to be tested if it is prime. Consider testing 2 non-prime numbers and 2 prime numbers

Moving on, the user is now asked to enter the last number to be evaluated in the Sieve of Eratosthenes Algorithm. If 100 is entered, the following output is given:



If 50 is now entered:



Task 7 Explanation:

In this task, a Binary Search Tree was implemented. This was done with the help of 2 classes, a ‘Node’ class and a ‘Tree’ class.

In the ‘Node’ class, a constructor function was initialized which specifies all the possible nodes a binary search tree can have. These are the root node, left node and right node.

Moving on, the next function is the ‘inserting()’ function which is a function called from the Tree class to be able to insert an integer in the correct position in the BST. This function checks if the value being entered is less than the root and if so proceeds to checking if a left node exists, if so this function is called recursively but if not, the value being tested is set as the left child. However, if the value entered is greater than the root, another check is performed to check if a right node exists. If so, the function is called recursively, but if not, the value is set as the right child. If the value being tested is neither greater nor smaller than the value being compared too, then the user is informed that the value being entered is already present in the BST.

The last function implemented in this class simply prints out the elements stored in the BST using inorder traversal. This method was chosen over the pre and post order traversals as it is the easiest to understand and follow of the three.

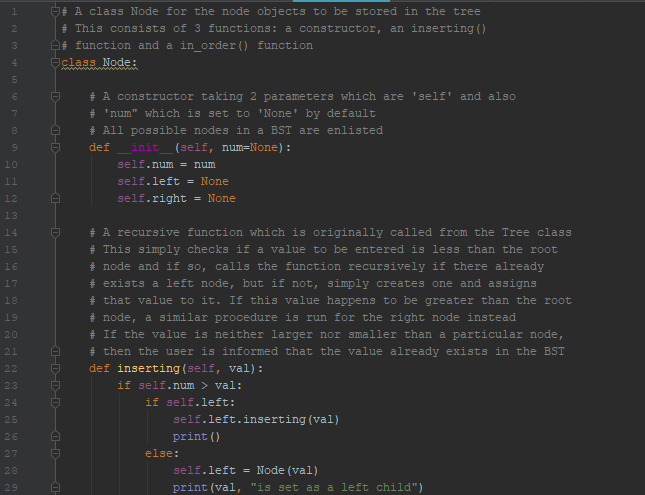
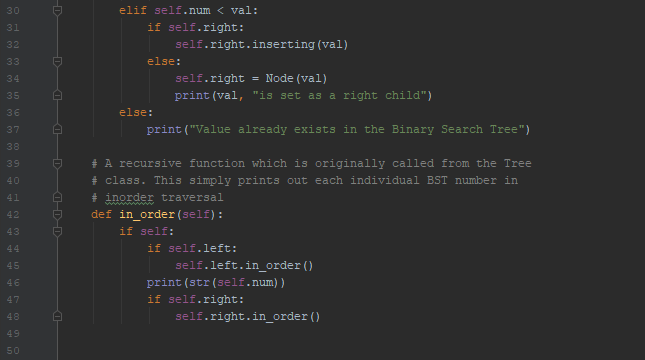
In the Tree class, a constructor function is initialized which sets the root node to None at the start.

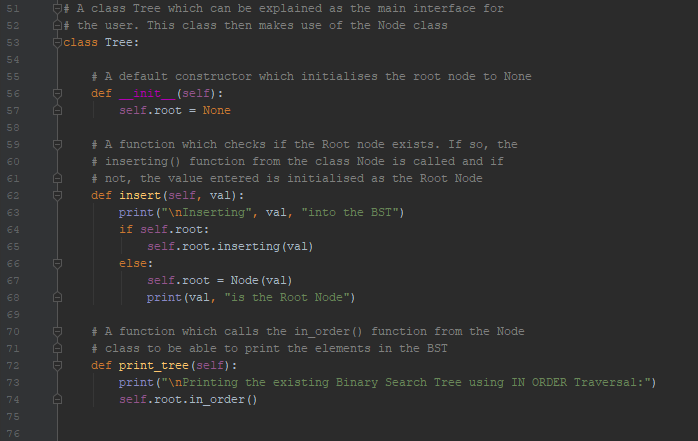
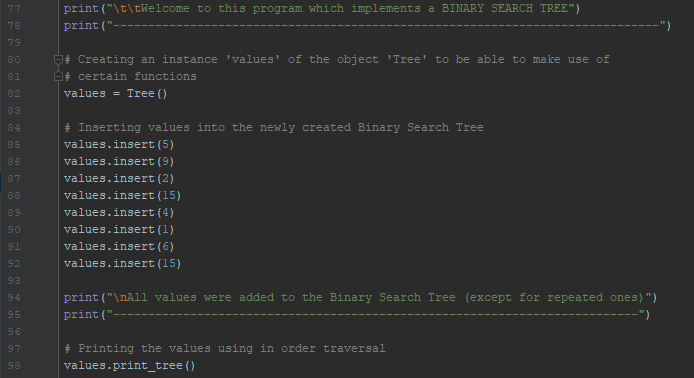
The next function called ‘insert()’ checks if the root node exists. If so, the ‘inserting()’ function from the Node class is called to be able to insert this node in the appropriate position. If not, then it means that there is no root node and so, the value entered is set as the root node.

The final function in the Tree class is the ‘print\_tree()’ function. This simply calls the ‘in\_order()’ function from the Node class.

Task 7 was successfully implemented and fit all the specified requirements without having any bugs.

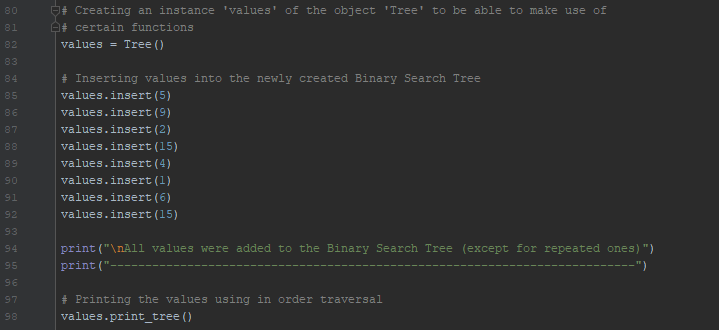
Task 7 Code:

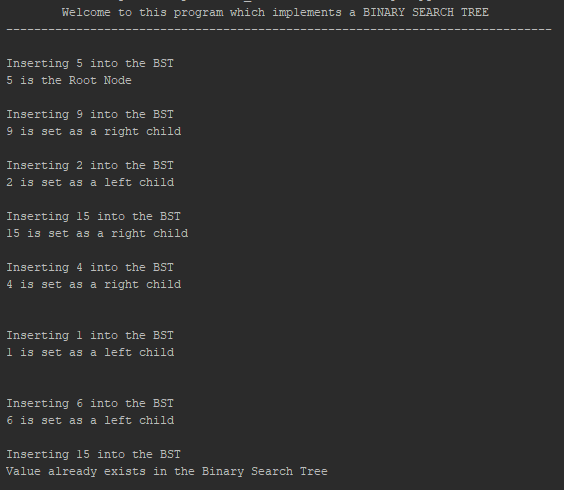
 

Task 7 Testing:

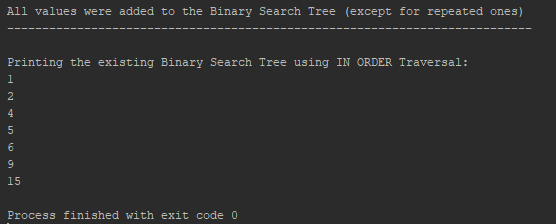
To test this program, the following code was inserted:



First, an instance of the object Tree was created to be able to make use of certain functions available in this class. Next, some values were inserted into the binary search tree. Note that the number ‘15’ was inserted twice to test the BST. Finally, the ‘print\_tree()’ function was called to print out the tree using the in order traversal method. The program was then compiled and the following output was displayed to inform the user on what is going on:



As expected, the unique values were added to the tree. When trying to add 15 a second time, the user is informed that it already exists in the tree and so this value is ignored. Next, a message is printed out to inform the user that the values have been successfully added and finally, the contents of the tree are printed out.



Task 8 Explanation:

In task 8, a program that finds an approximation to the square root of a given number n using the Newton-Raphson method was written.

In this task, a ‘user\_input()’ function was written to be able to validate and store input entered by the user. It was decided that the user had to enter 3 different values to be able to work out a square root approximation. These were:

1. The number n whose square root was to be found
2. An approximate answer that the user thinks is correct
3. The number of approximations required to reach the desired answer.

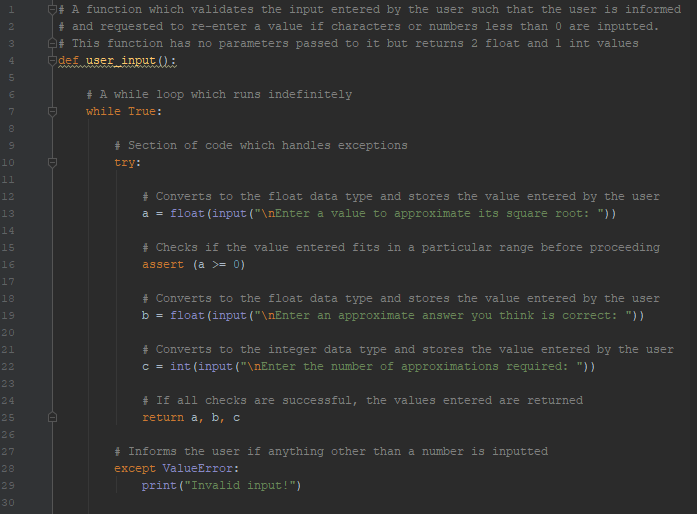
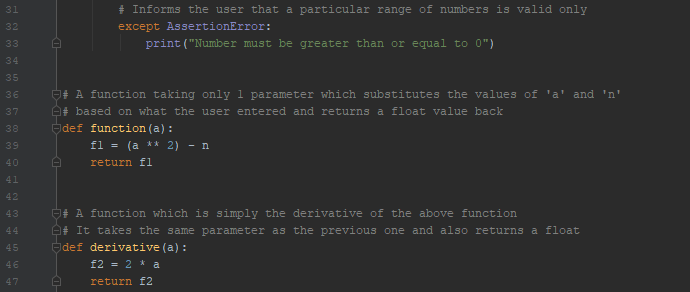
These were then inserted in a try-except statement in this function which made sure that values entered for 1 and 2 were float and the value entered for 3 was an integer. It also made sure that the value entered for 1 was greater or equal to 0.

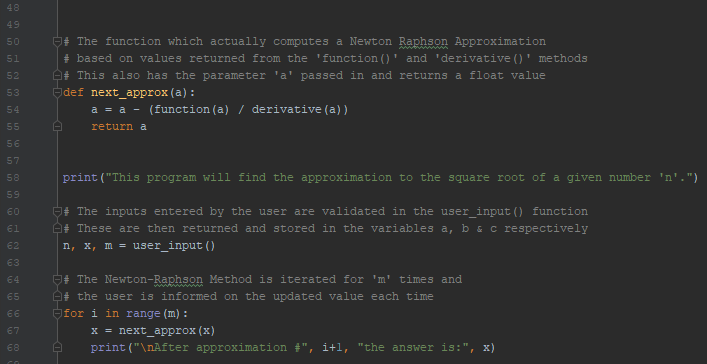
Moving on, a function for f(x) and another for its derivative were written. These were then called from the next function called ‘next\_approx()’ which calculated the next approximation for the square root of the number n. This function consisted of the formula used for the Newton Raphson method.

Finally, a for loop was implemented at the end to call the ‘next\_approx()’ function for so many times based on the previous user input, each time obtaining a more accurate approximation of the square root function.

Task 8 was successfully implemented and fit all the specified requirements without having any bugs.

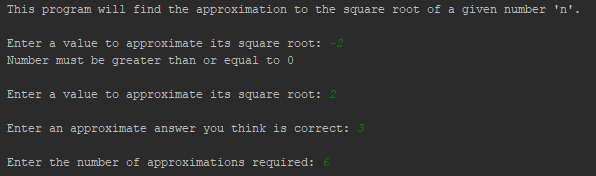
Task 8 Code:

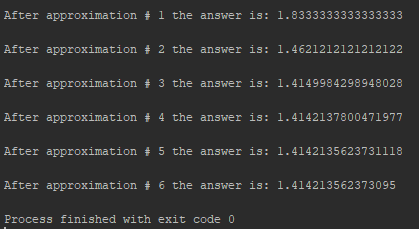


Task 8 Testing:

When running this task, the user was asked to enter the 3 values discussed. Consider the user entered an invalid number on the first attempt followed by 3 valid ones. The output is shown below:



The program then computes the answers based on the values entered and the user is informed at every step of the way as shown:



These values were compared to the values on a calculator and were found to be more accurate as the program computes more decimal points.

Task 9 Explanation:

In this task, a program was written to identify all the numbers repeated more than once in a given array of integers. This was done in the fastest, most practical and memory efficient way as possible.

To start off, a ‘user\_input()’ function identical to the one in task 3 was coded. This asked the user to enter the maximum amount of integers he/she wanted in a list to be created, followed by the possible min and max values present in the list. These values were validated in this function to ensure that the user enters correct information.

The next function to be created was the ‘generate\_list()’ function which is also similar to that created in task 3. It simply populates an empty list initialized at the start of the program based on what the user entered. The only difference is that these numbers generated are returned back and eventually printed out to the user.

The overall most efficient sorting algorithm out of all the sorting algorithms existing is the quick sort algorithm. This was implemented just like in task 1 and the generated numbers were then sorted through this function.

Finally a ‘repeat()’ function was created which identified the repeated numbers and printed them out to the user. This was done by first creating an array ‘repeated’ which consisted of only 1 element at the start ( “REPEATED NUMBERS: “). This array was created to store any repeated elements found in the tested list. The reason why this string of text was inserted in the ‘repeated’ list will be explained shortly.

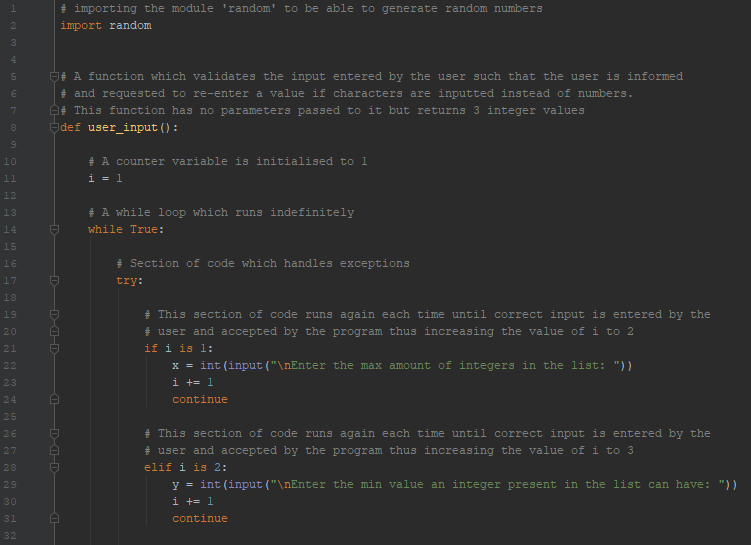
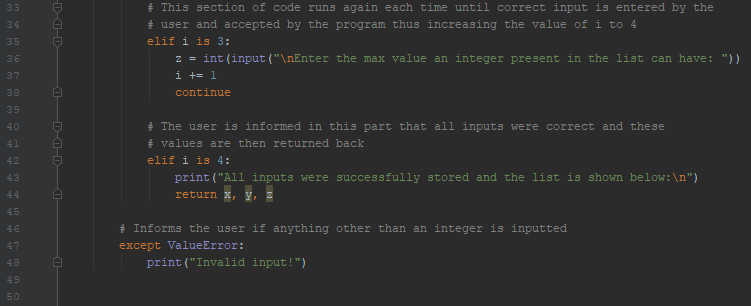
Next, a for loop was implemented which appended any repeated numbers to the list. To avoid having more than one instance of the same number show up in the list ‘repeated’, the code was optimized in the following manner:

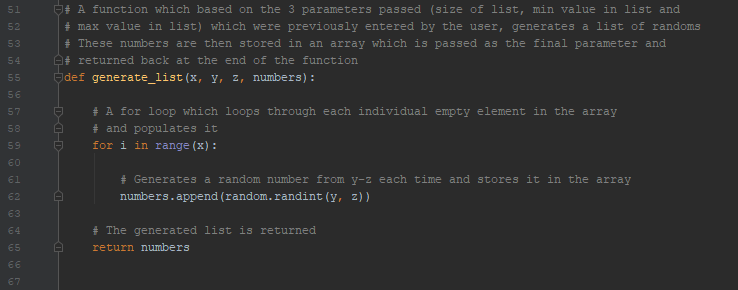
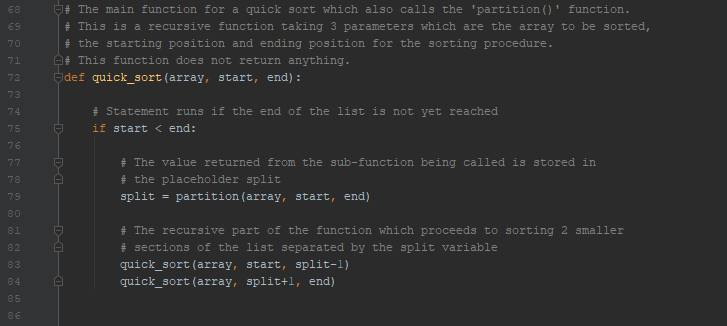
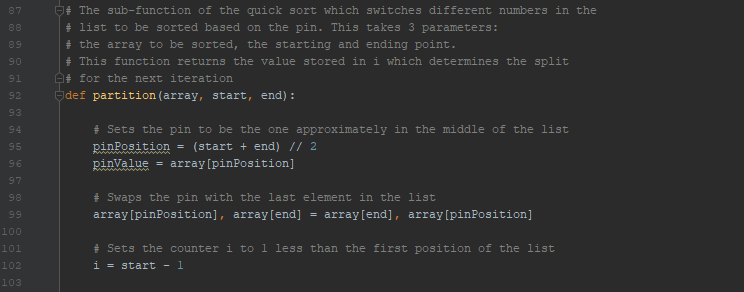
A number in numbers[i] would be added to the list if and only if the number in numbers[i+1] is the same & the number to be added is not equal to the last number added to the ‘repeated’ list. In this manner, only one instance of a repeated number would be present in the ‘repeated’ list.

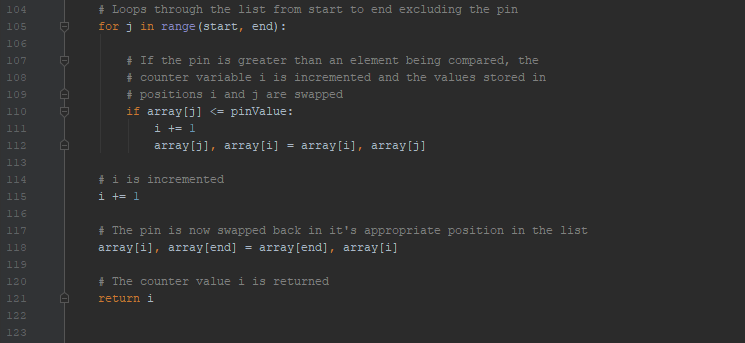
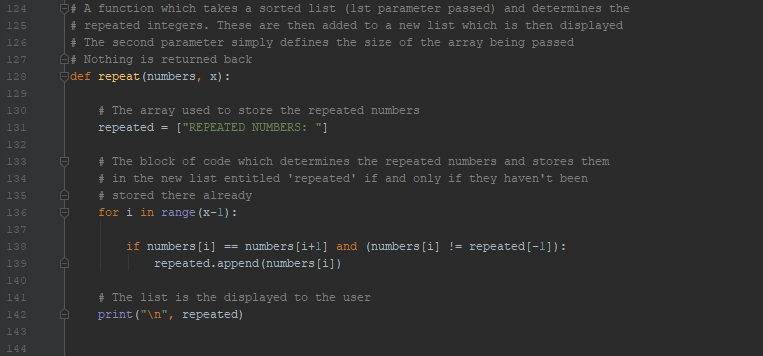
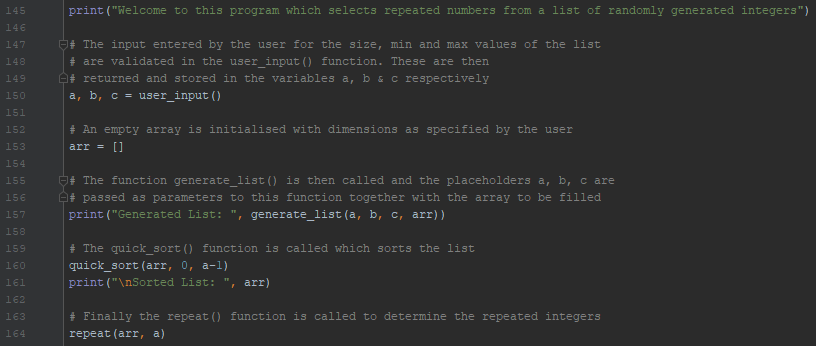
The reason why strings were added at the start of the function to the ‘repeated’ list was so that the if statement could work at the very start, when there were still no numbers in the ‘repeated’ list. When checking for repeated numbers, the if statement also ensured that no numbers identical to the one going to be added were available in ‘repeated[-1]’. This statement always worked when the list was not empty as last element in the ‘repeated’ list could be accessed. However, if the list was initialized to empty, the program would give an error and not run because there would be nothing else to compare to. Thus, adding this string of characters made the if statement valid and the program work.

Task 9 was successfully implemented and fit all the specified requirements without having any bugs.

Task 9 Code:

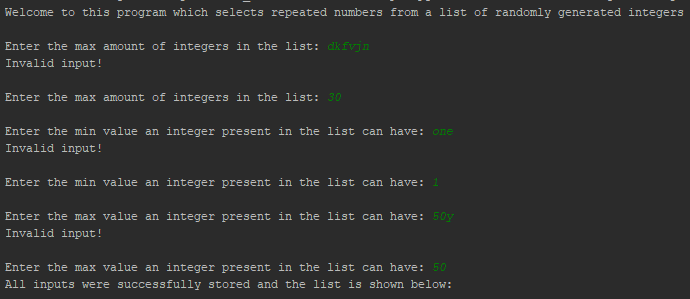
 

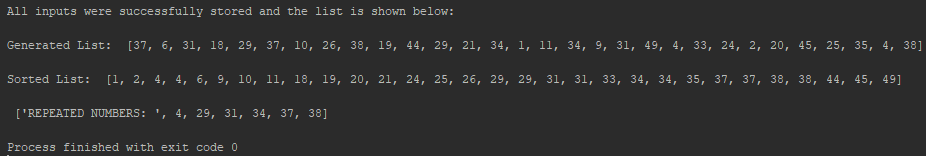
  

Task 9 Testing:

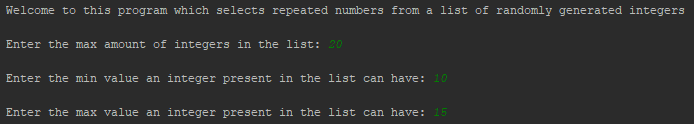
Upon hitting run, the user is asked to enter 3 values for the list. Consider the user entered an invalid followed by a valid input each time.



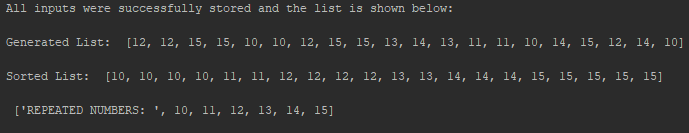
The program then creates the list, sorts it and displays the repeated values to the user as shown.



In the case above, it can be seen that the values: 4, 29, 31, 34, 37 and 38 were repeated twice. Consider now a case in which numbers appear repeated more than just twice. The user input for this case is shown below:



This meant that a list of length 20 with values between 10-15 could be found in the generated list. This forced the creation of a list with a lot of repeated values. In fact, all the numbers from 10-15 were appended to the ‘repeated’ list and displayed to the user as shown:



As seen, 10 was present 4 times in the list but was only appended once to the ‘repeated’ list due to the optimized if statement.

Task 10 Explanation:

In this task, a function was written to recursively call itself to find the largest number in a given list of integers.

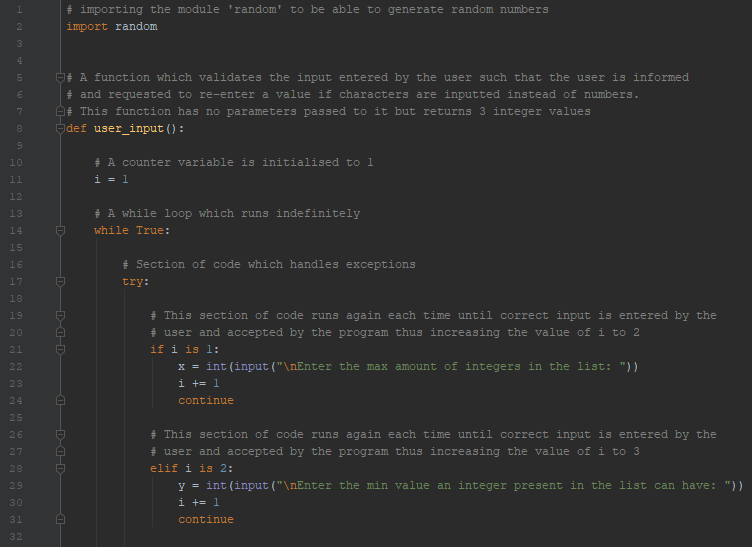
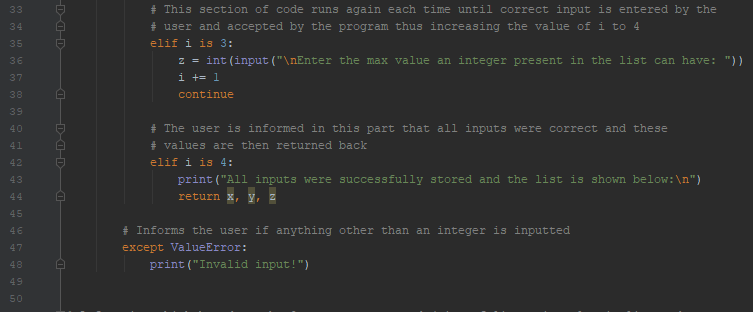
Starting off, a ‘user\_input()’ function was written, identical to that of task 9, were a user is asked to enter the length of a list to be generated, followed by the min and max values possibly present in the list. This functions aim is to validate this input and ensure that the user enters integer values.

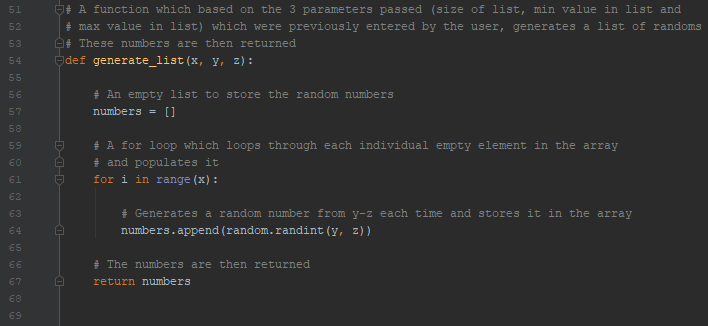
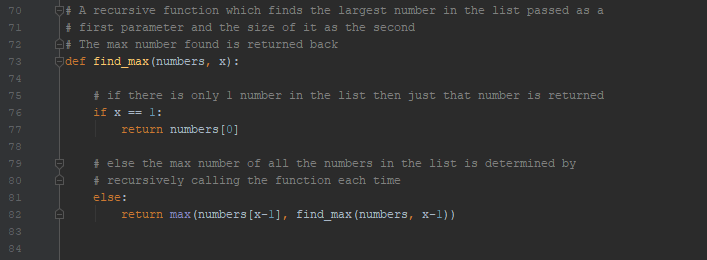
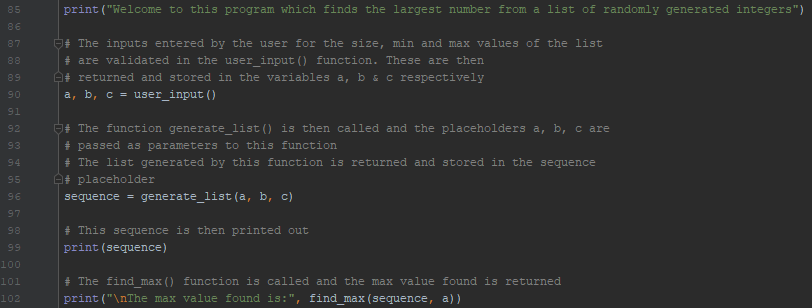
Next, based on the entered values, a function ‘generate\_list()’ is called and based on the values entered, a created list is populated. This function is once again identical to that created for the previous task. The numbers are then returned at the end of the function.

The last function present is the ‘find\_max()’ function itself. This function consists of a for loop which checks if there is only 1 number in the list and if so, returns that number as the max. If not, the max of a number present in the list and a recursive call to the same function is found.

Task 10 was successfully implemented and fit all the specified requirements without having any bugs.

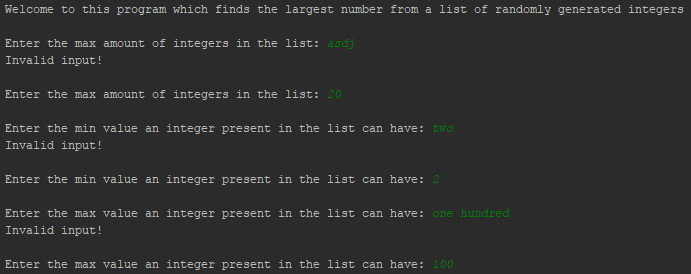
Task 10 Code:

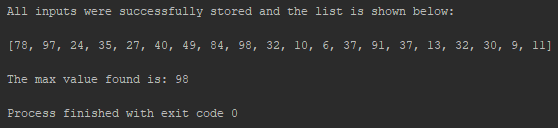
  

Task 10 Testing:

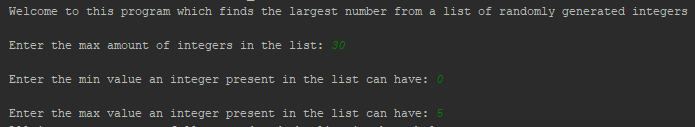
When running this program, the user is prompted to enter 3 values at the start. Consider the user enters an invalid value followed by a valid one each time.



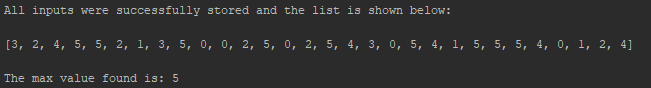
Next, the list is generated and displayed to the user. Following this, the max value in the list is found and displayed.



As can be seen above, the function worked as expected and 98 was printed out as the max value found. Consider now the case in which a list having repeated elements is traversed to find its max value. The parameters entered for this list are the following:



Given a length of 30 in which numbers from 0 to 5 can be inputted into the list, the numbers generated are forced to be repeated. But as expected, no errors were given and the program still worked as expected returning 5 as the max number present in the list.



Task 11 Explanation:

In this program, functions for the sine and cosine series expansions were written and tested out.

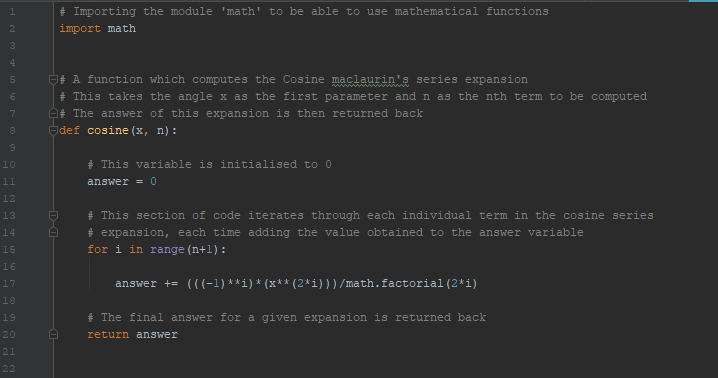
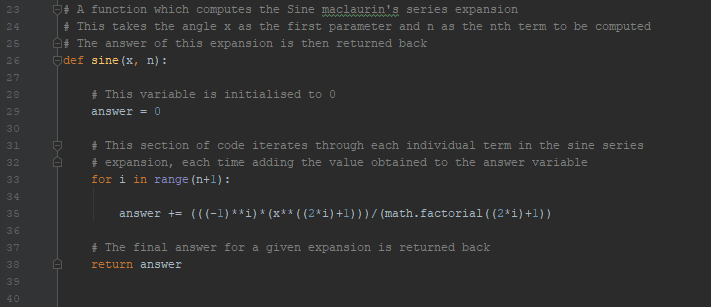
To start out, 2 variables were created to store input from the user. The first one stored the value given to the angle and the second, the position of the last term of the expansion the user wants to compute.

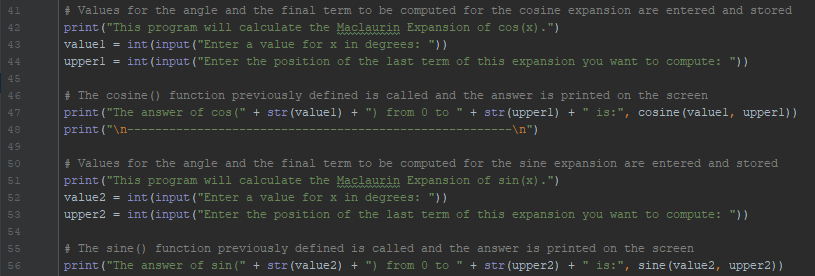
Next a cosine function was created which has these 2 variables passed as parameters. In this function, an answer variable is declared to 0 at the start. Following this, a for loop iterates based on the user input and the mathematical representation of the cosine function is evaluated, each time storing the result in the answer variable. When looping finishes, the answer is returned back to the user and displayed.

Two other variables dedicated to a sine function are also created to store a value for the angle and the position of the last term in the sequence to be computed. Once again, another function for sine is created which has the same layout and functionality as the cosine function, with the only difference being the formula involved to compute the result. In this case, the result is also returned and displayed to the user.

Task 11 was successfully implemented and fit all the specified requirements without having any bugs.

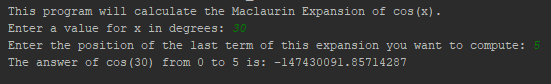
Task 11 Code:

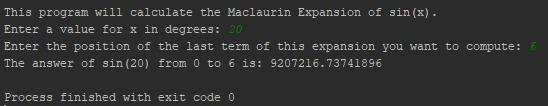


Task 11 Testing:

When running the program, the user is asked for 2 input values. Consider the case in which the user enters 30 followed by 5. The result of computation for the cosine series expansion is displayed:



Next 2 values for the sine series expansion are required. Consider the user enters 20 followed by 6. The result of computation for the sine series expansion is displayed:



Both results were compared to results on a calculator and were found to be correct.

Task 12 Explanation:

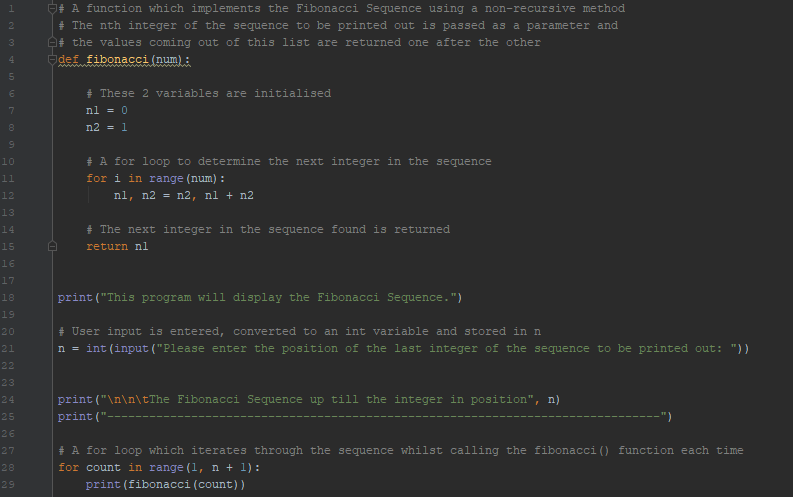
In task 12, the Fibonacci sequence of numbers was implemented iteratively.

The user was asked to input the position of the last element in the sequence he/she wanted printed out. This value was then inserted in a for loop which iterated from 1 up till the number entered. This for loop was constantly printing out the result of the ‘fibonacci()’ function created.

The ‘fibonacci()’ function consisted of 2 variables initialized to 0 and 1 respectively at the start. These variables were then inserted in a for loop which set the value of n1 to the present value of n2 and the new value of n2 to the added value of both variables. Nonetheless, this setting of variables was performed in a single line of code. The final value of n1 was finally returned back and printed to the user.

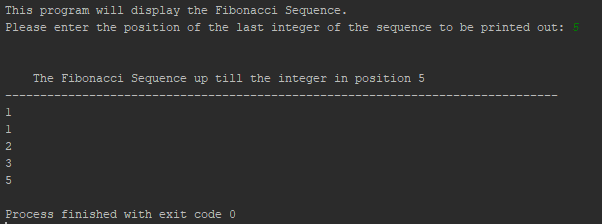
Task 12 was successfully implemented and fit all the specified requirements without having any bugs.

Task 12 Code:

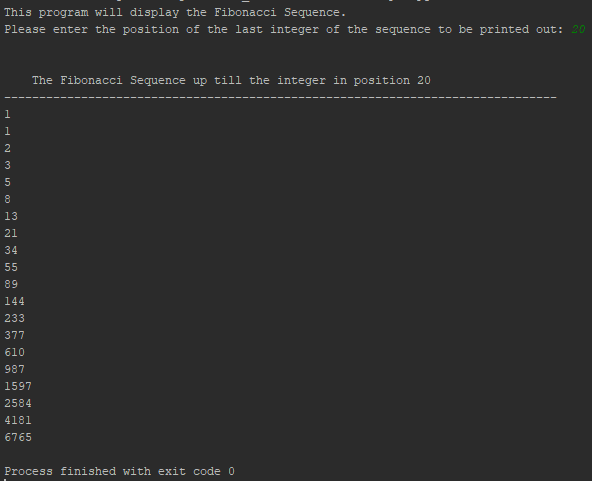


Task 12 Testing:

The program was run and the user was prompted to enter the position of the last element in the sequence he wanted to compute. Consider the case in which the user enters 5:



Consider now the case in which the user enters 20:



---------- END OF ASSIGNMENT ----------