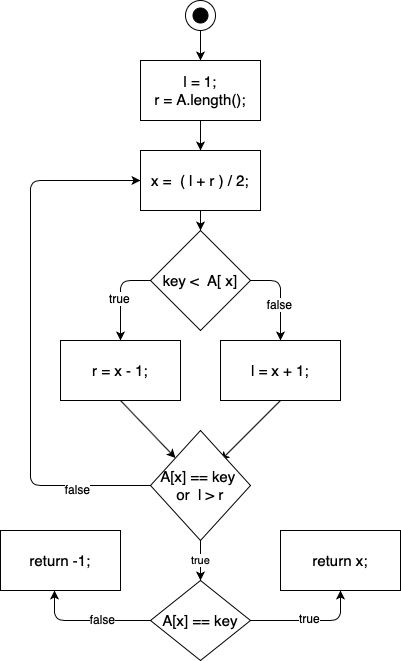
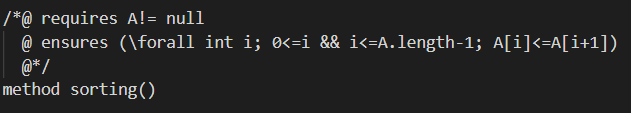
**Question 1**

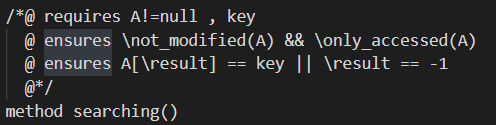
****

**Question 2**

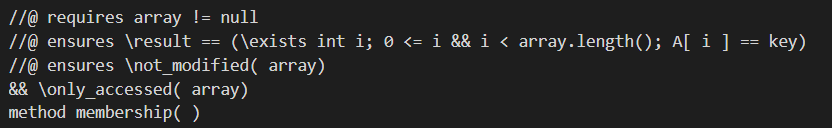
**Sorting**

****

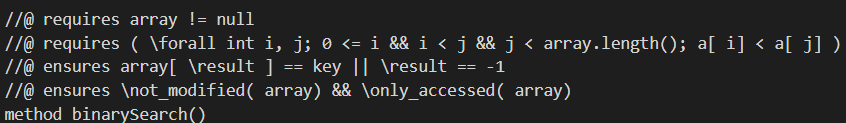
**Searching**

****

**Membership**

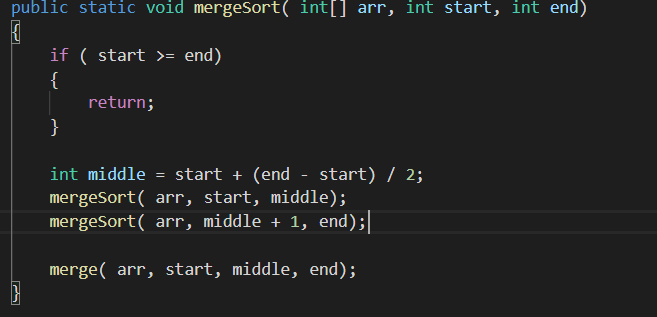
****

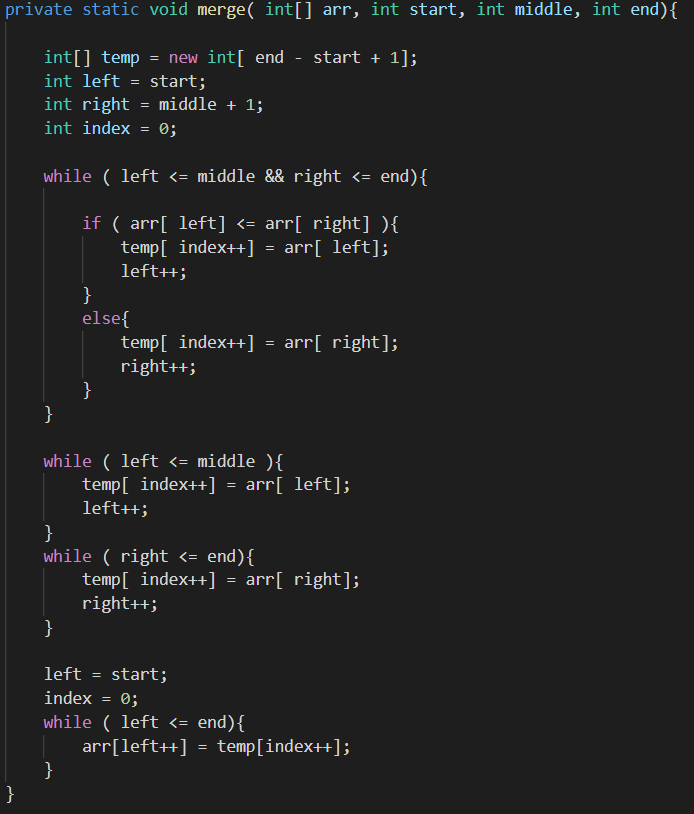
**Binary Search**

****

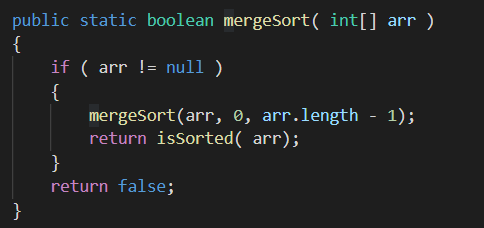
**Question 3**

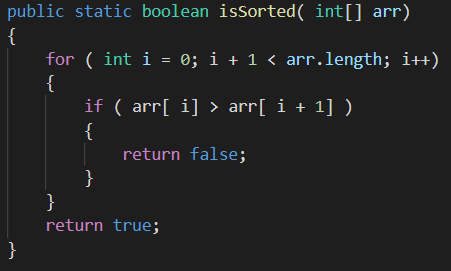
**Program (i)** shows the implementation of merge sort which is used to sort integer arrays of arbitrary length.



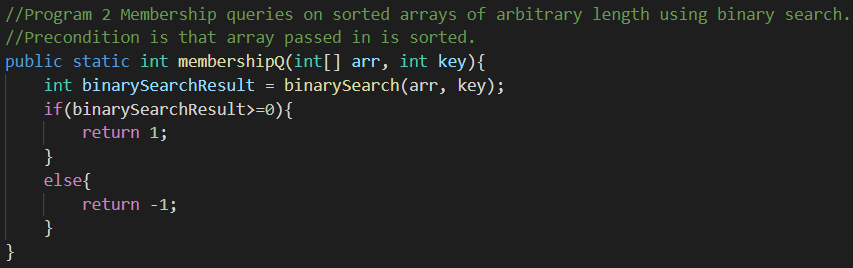


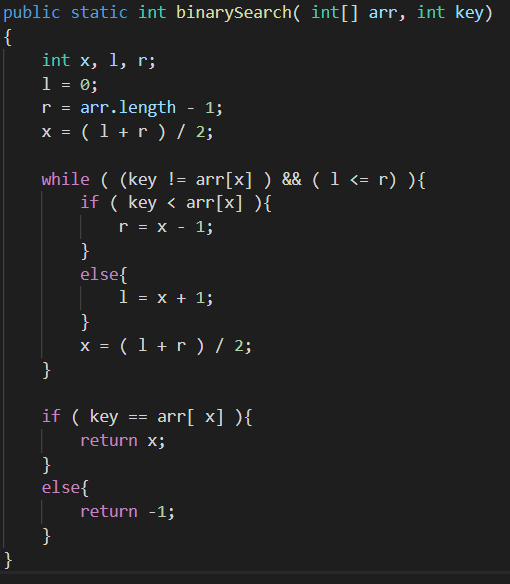
In addition, we have also overloaded merge sort with a Boolean function which checks if the array is indeed sorted.



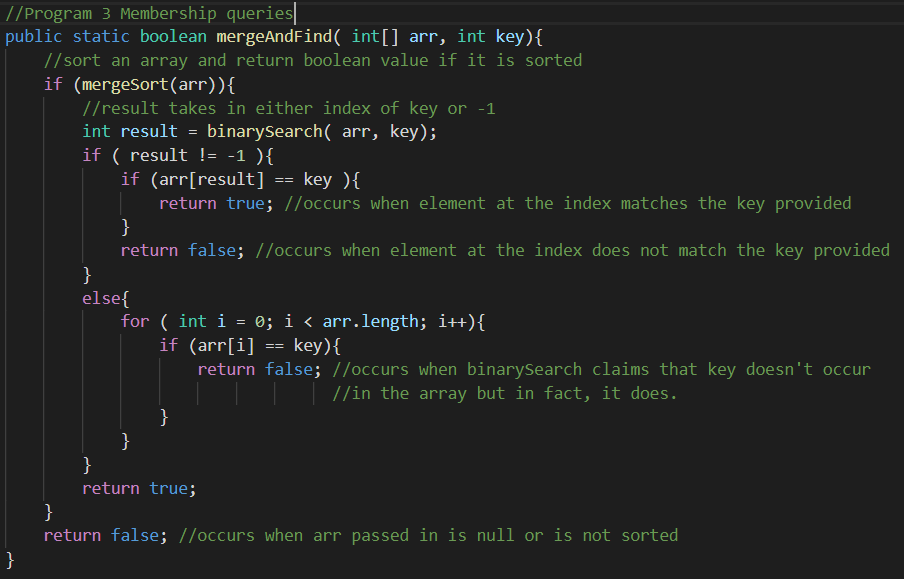


**Program(ii)** shows the implementation of membership queries on sorted array of arbitrary length using binary search.





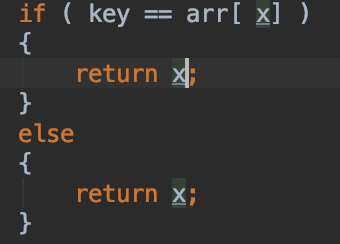
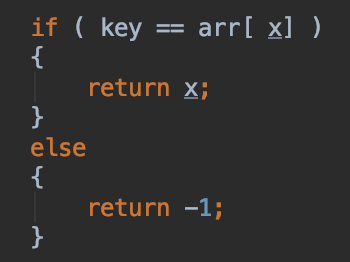
**Program(iii)** shows the implementation of membership queries on unsorted array of arbitrary length using binary search. It combines aspects of program(i) and (ii)



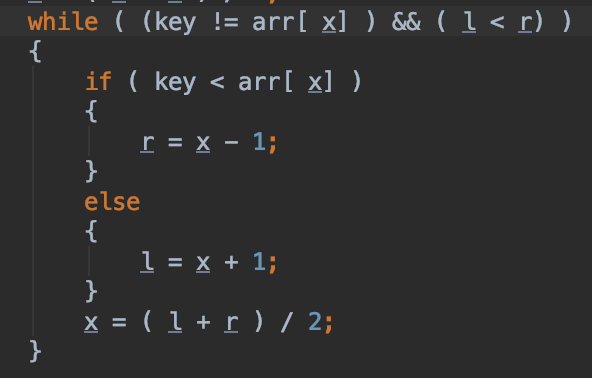
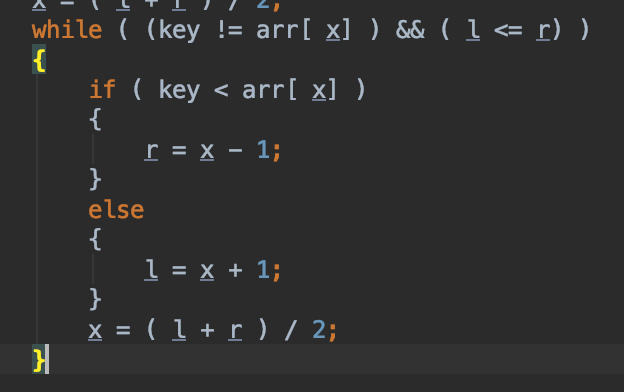
**Question 4**

|  |  |  |
| --- | --- | --- |
| Length = 10 | Random Testing | Pair Wise Testing |
| Mutation 1: returned x in binarySearch for all case | 1 | 6 |
| Mutation 2: while ( (key != arr[ x] ) && ( l < r) ) | 39421 | 7 |
| Mutation 3: if ( key <= arr[ x] ) | 812 | 3 |
| Mutation 4: while ( (key != arr[ x] ) && ( l > r) ) | 173 | 1 |
| Mutation 5: while ( (key == arr[ x] ) && ( l <= r) ) | 722 | 1 |
| Mutation 6: while ( left < end)  {  arr[left++] = temp[index++];  } | 1 | 1 |

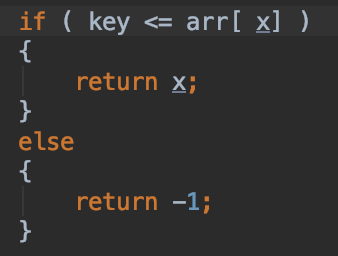
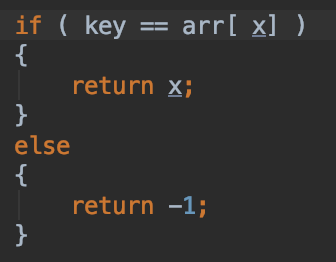
* **Mutation 1**

In the binary search, at the end either an index is returned if key is found or -1, indicating key is not found. Here, we have introduced a mutation by returning the last found index even though key is not found. Therefore, binary search function always returned the index that was found last by the function.

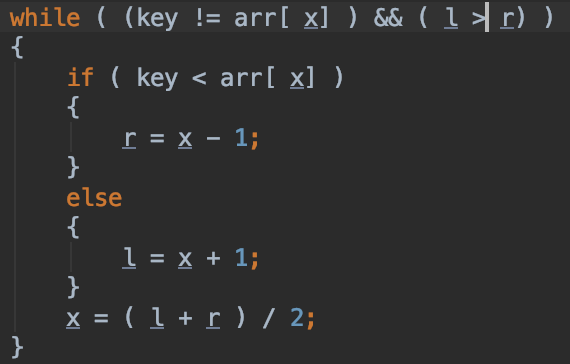
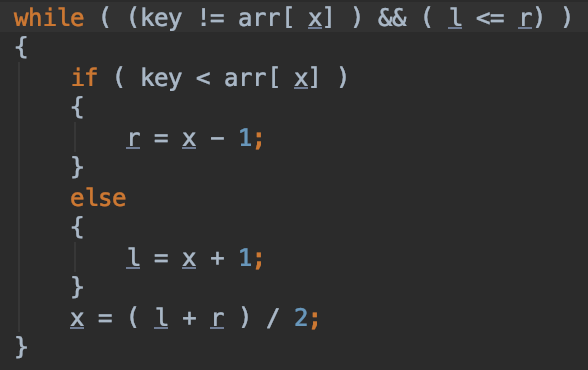
* **Mutation 2**

In the binary search function, the logic is changed to introduce a mutation. When the key is searched in the loop, when the left is equal or lower than the right index, this has been altered to while left index is lower than the right index. Therefore, while the key is searched, left index cannot be equal to the right index in the binary search method.

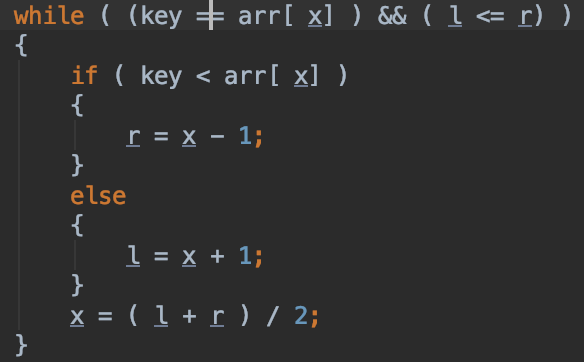
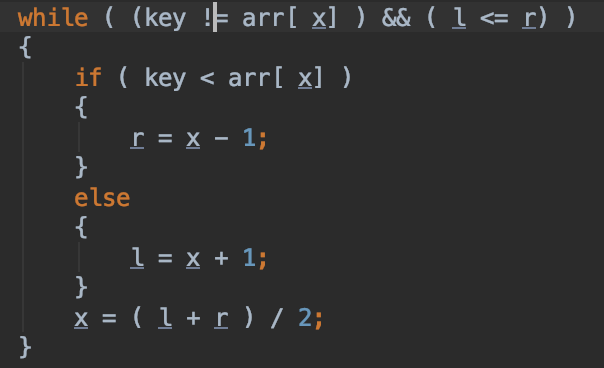
* **Mutation 3**

In the end of the binary search, the value of the index in the array is compared with the key. If there are equal, the index is returned, else -1. A mutation is introduced by making this comparison if key is less or equal to the found index in the array.

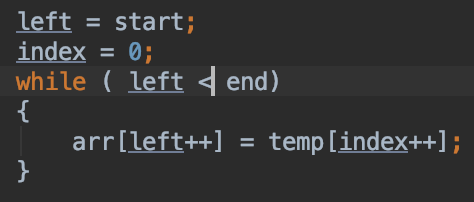
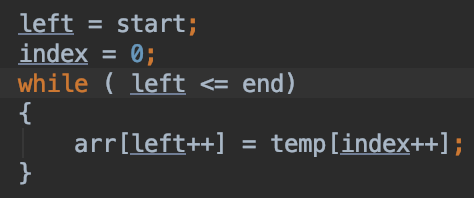
* **Mutation 4**

The logic in the comparison of left and right index while searching for the key in the binary search is mutated again. This time, while the left index is greater than the right index a valid search could be made.

* **Mutation 5**

The loop in binary search continues while a key is not found. To introduce another mutation, this logic is changed to while the key equals to the value in the middle of the array that is sliced with left and right margin.

* **Mutation 6**

This time, since all the mutations were in binary search, the logic in merge sort is mutated. In the end of the merging part, the temporary array is being copied back to the original range in the argument array. The mutation is done by not allowing the left index to reach the end of the range. This way, the temporarily sorted array’s last element is not copied back to the original array.

**Test Effort Comparison**

When the table is investigated for array length 10, it can be declared that random testing takes more attempts to find these mutations than pair-wise testing. There are some cases where random testing is better in terms of finding the mutation in lesser number of attempts. But overall, pair-wise testing is consistent with small variance. However, random testing has a big variance in the number of test cases needed to find the mutation. This is because the randomness factor is not helpful with certain boolean logic cases in binary search as depicted in the table. While there were 100,000 test cases prepared for random testing, pair-wise testing had only 57 test cases. This shows that pair-wise testing may cover mutations/bugs with less number of test cases. Consequently, it is more efficient than random testing.

|  |  |  |
| --- | --- | --- |
| Length = 100 | Random Testing | Pair Wise Testing |
| Mutation 1: returned x in binarySearch for all case | 1 | 102 |
| Mutation 2: while ( (key != arr[ x] ) && ( l < r) ) | unable to find | unable to find |
| Mutation 3: if ( key <= arr[ x] ) | 154 | 2071 |
| Mutation 4: while ( (key != arr[ x] ) && ( l > r) ) | 160 | 1 |
| Mutation 5: while ( (key == arr[ x] ) && ( l <= r) ) | 624 | 1 |
| Mutation 6: while ( left < end)  {  arr[left++] = temp[index++];  } | 1 | 1 |

When the length of the array is increased, it seems as the array length is so large, although random testing had tried 100,000 test cases, both testing methods failed to find the bug in mutation 2. Although this time random testing performed much better than pair-wise testing in mutation 1 and 3, overall, pair-wise testing dominates random testing with finding the bug on the first test case with mutation 4 and 5. Therefore, as the length of the input has increased, the performance of pair-wise testing seems to be diminished. While the length is increased, certain bugs became even harder to find, such as the index dependency in mutation 2. Therefore, it can be claimed that pair-wise testing is better in general than random testing because of its performance in small-size inputs and its almost equal performance in large-inputs in this testing scenario.