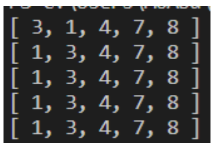
Q1,

A:

i) [3,7,1,4,8], [3,1 4,7,8], [3,1 4,7,8], [1,3 4,7,8] , 

```
[ 3, 1, 4, 7, 8 ]
[ 1, 3, 4, 7, 8 ]
[ 1, 3, 4, 7, 8 ]
[ 1, 3, 4, 7, 8 ]
[ 1, 3, 4, 7, 8 ]
```

largest value bubbles up to the top of the array.

ii) N^2

iii) The worst-case time complexity for both the insertion and bubble sort are the same so it won't matter which one I picked as they both have a worst case time complexity of O(n^2).

B:

i) N^2

ii) The linear search algorithm has a worst-case time complexity of O(n) so it's better to use that on the array instead of sorting it before hand with the bubble sort algorithm with a time complexity of O(n^2).

iii) If the Array is already sorted then it's best to go with the first method since the worst case time complexity will be O(log n)

C:

i) This ternary search has two pivot points and searches the array using those 2 pivot points for the element unlike the bubble sort which only has one pivot point.

ii) The worst time complexity for the ternary search is 0(log3N) but that the constant doesn't matter so it's actually 0(logN) which isn't any different from the worst-case time complexity for the binary search with 0(logN) so both algorithms have the same worst case time complexity.

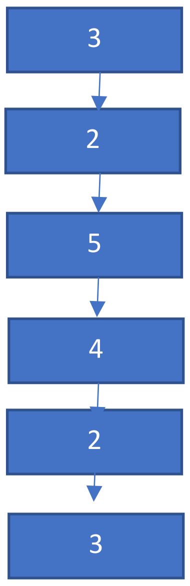
Q2,

A:

i) [3,2,5,4,2,3], this is a palindrome since the array is the same backwards.

ii) An array containing one and nothing are palindromes since it satisfies the definition of palindromes as it's the same backwards and forwards.

E:

i) 

3 would be at the top of the stack

F:

i) The peek would return 3.

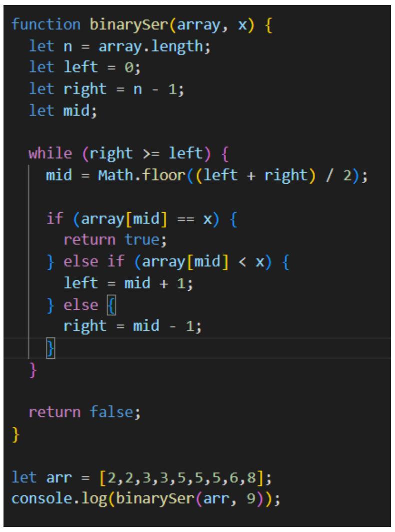
Q3,

A:

i) There is an infinite recursion caused by the last line off the function return n * factorial(n); this is an issue since we are calling n each time within the function so the base case will never be reached. The solution is to write return n * factorial(n-1); the function call will be one less each time and eventually reach the base case.

C:

1. The sorted array is [2,2,3,3,5,5,5,6,8]
2. Array is [2,2,3,3,5,5,5,6,8], L pointer = 0, Right pointer = 8 and mid pointer = 4. We will then exclude values less than 5. The L pointer = 5, , Right pointer = 8 and mid pointer = 6. We will again exclude all the values less than 5 since 9 is greater than 5. The L pointer = 7, Right pointer = 8 and mid pointer = 7. Once the left pointer passes the right pointer, we will know that the item isn't in the array so we can set that as a condition.



```
function binarySer(array, x) {
  let n = array.length;
  let left = 0;
  let right = n - 1;
  let mid;

  while (right >= left) {
    mid = Math.floor((left + right) / 2);

    if (array[mid] == x) {
      return true;
    } else if (array[mid] < x) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }

  return false;
}

let arr = [2,2,3,3,5,5,5,6,8];
console.log(binarySer(arr, 9));
```

D:

1. Decrease and conquer takes a problem and turns it into a smaller problem to be solved. Binary search takes the input and chops it into a smaller piece in order to solve the problem more efficiently.
2. Divide and conquer takes a problem and divides into two or more smaller problems that are smaller versions of the same problem like for example quick sort. Quick sort partitions parts of an Array then sorts those partitions one by one then adds up the results at the end.

E:

1. The worst case time complexity Binary Search is 0(Logn) since you are chopping up the input each time before calling another function to recurse into.