Name: Hrishikesh Rajan
Email: hrishikeshrajan3@gmail.com

Assignment 1

Q.2)You are a product manager and currently leading a team to develop a new product.Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad. Suppose you have a version and you want to find out the first bad one,which causes all the following ones to be bad. Also, talk about the time complexity of your code

Test Cases:
Input: [0,0,0,1,1,1,1,1,1]
Output: 3

Explanation: 0 indicates a good version and 1 indicates a bad version. So, the index of the first 1 is at 3. Thus, the output is 3
Ans.2)

Case : 1 (Iterative Approach)

```
function badVersion(arr){
   for (let i = 0; i < arr.length; i++) {
      if(arr[i]===1){
         return i;
      }

   }
}

const arr = [0,0,0,1,1,1,1,1,1];

const result = badVersion(arr,0,arr.length-1)
console.log(result)
```

Time Complexity :

O(n)

Explanation :
Since each value inside the array is compared with the given value, it goes up to n-1, where n is the array's length

Case : 2 (Optimised Approach)

```
function badVersion(arr,low,high){

    while(low <= high){

        let mid =  Math.floor(low + ((high - low)/2));

        if(arr[mid] == 0){
            low = mid + 1
        }
        if(arr[mid] <= 1){
            high = mid + 1
        }
        if((arr[mid] === 1 )&& (arr[mid -1]) === 0){
            return mid
        }

    }

    return -1

}

const arr = [0,0,0,1,1,1,1,1,1];

const result = badVersion(arr,0,arr.length-1)
console.log(result)
```

Test Cases:

Input: [0,0,0,1,1,1,1,1,1]
Output: 3

Time Complexity :
O(log(n))

Explanation :
Our goal here is to split the array into two parts by finding the middle value. After It comparing the middle value with the best versions “0” if yes, that means the left section of the array is full of successful versions .So the condition is to find the bad versions “1”, we don’t need to search in left section for that we reduce the search space in the left section by left = mid + 1. Since middle value is already compared so it should be mid+1 to get the next value.Next condition is to check the value approximate equal or exact equal to bad version “1” , If we get true “(arr[mid]<=1)” which means we found the bad version (“still the index is unknown”). So here again we could reduce the search space in the right section my high = mid -1 . The last we check is the middle is equal to the bad version and the left value to the good version. That helps to ensure that the current bad version is the first bad version “ (arr[mid] === 1) && (arr[mid -1]) === 0)” . If not the iteration continues until it reaches the first bad version then in returns the index “mid” value.So here we are not traversing through the array in a brute force approach but instead reducing the search space by a factor so the time complexity become O(log(n))

Q.3)Given a positive integer num, write a program that returns True if num is a perfect square else return False. Do not use built-in functions like sqrt. Also, talk about the time complexity of your code.

Test Cases:
Input: 16
Output: True

Input: 14
Output: False

Case 1 (Iterative Approach)

```
function isPerfectSquare(num){

for (let i = 1; i <= num; i++) {

   if( ((i * i)) === num){
      return (true)
   }
}
return false
}

const result = isPerfectSquare(14)
console.log(result)
```

Test Cases:
Input: 16
Output: True

Input: 14
Output: False

Time Complexity :
O(n)
Explanation:
Even though the difference between input and output is huge, the value can relate to order of n time complexity up to a certain point

Case 2 (Optimised Approach)

```
function isPerfectSquare(num){
let low = 0;
let high = num;

while(low <= high){

    let mid = Math.floor(low + ((high -low)/2))

    if((mid * mid) === num){
        return true;

    }

    if((mid * mid) > num){
        high = mid - 1;
    }

    if( (mid * mid) < num){
        low = mid + 1
    }
}
    return false
    }

const result = isPerfectSquare(14)
 console.log(result)
```

Test Cases:
Input: 16
Output: True
Input: 14
Output: False

Time Complexity :
O(log(n))

Explanation :
Here instead of calculating each value from the beginning we divide the two that we could get a mid value. Then we multiply the middle value by itself where it matches the given input. If it does not match then it checks if the multiplied value is greater than the given input , if yes that means the current mid is higher so we need a smaller mid value by adjusting “high = mid -1” . Suppose if the multiplied value at the beginning is smaller, then we need to increase the mid value to get a higher mid number . So we do “low = mid + 1”, This process continues until the multiplied value is equal to given input. If matched return “true” else return “false” .So here we reducing the search space the multiplication that we need to only calculate less number to reach the output.So we could say it’s an order of log(n).That is O(log n)

Q.1)Compute and return the square root of x, where x is guaranteed to be a non-negative integer. And since the return type is an integer, the decimal digits are truncated and only the integer part of the result is returned. Also, talk about the time complexity of your Code.

Test Cases:
Input: 4
Output: 2
Input: 8
Output: 2

Explanation: The square root of 8 is 2.8284…., the decimal part is truncated and 2 is
returned

Ans)

Case 1 (Iterative Approach)

```
function squaring(num){

   let x =num;
   let i =1;
   while(i<=20){
      x = (x + (num/x))/2
      i++
   }

   return Math.floor(x)
}

const result = squaring(36)
console.log(result)
```

Time Complexity :
O(n)
Explanation:
Here the root of the given input is calculated using the mathematical formal called Newton raphson method.But the problem with this approach is that it can reach the order of n if the number is that to get the precise floating point values. Hence it can be considered as On(n)

Case 2 (Optimised Approach)

```
function square(num, low, high){
   let mid = (low + high) / 2;
   let power = mid * mid;
   if ((power === num) || (Math.abs(power - num) < 0.00001)){
      return mid;
   }else if (power < num){
      return square(num, mid, high);
   }else{
      return square(num, low, mid);
   }
}

function getSquareRoot(num){
   let i = 1;
   while (true){
      if (i * i === num){
         return i;
      }else if (i * i > num){
         let res = square(num, i - 1, i);
         return Math.floor(res);
      };
      i++;
   }
}
console.log(getSquareRoot(8));
```

Time Complexity :
O(nlog(n))
Ans :

Explanation :
Here first we go through a normal brute force approach that will take order of n time complexity.But in the cases for non perfect squares it goes through another algorithm which works based on order of log(n). So here the order of log (n) become nested to the order of n
Hence we could say that total time complexity is O(nlog(n))