Online Test: Part B, Question 2

Problem Solving for Computer Science

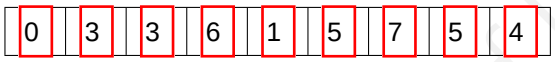
REMEMBER TO ONLY ATTEMPT TWO OF THE THREE LONG QUESTIONS IN PART B

This question carries a total of 30 marks and consists of sub-questions.

Question 2

This question is about searching, and determining how often a value appears in an array. For this question you will need your nine-digit student number beginning with "03", *not* your campus username. You can find your nine-digit student number on your Goldsmiths ID card, or through the MyGoldsmiths app.

1. Once you have your student number, consider the following empty array. In order, put each digit of your student number into this array so that each element stores one digit.



Implement the Binary Search algorithm, by hand, on this nine-element array to search for the value 11. This number will not appear in the array above, but you need to demonstrate how the algorithm works to show this. To implement it you first need to sort the array, which you can do by inspection without implementing an algorithm. You can write your implementation in the text box below. Alternatively, if you prefer, take a photo of your implementation, put the name of the image file in the box below, and upload the image with the rest of your files. [8 marks]

Answer:

For starters, the sorted array will be [0,1,3,3,4,5,5,6,7]. To start binary search we will need to set the variable 'left' to index zero and a variable 'right' to the index array.length-1. Also, if right is equal to zero, then we return false. Then we will loop over the values in the array until we find the element we are looking for or as long as our left variable is smaller or equal to the right variable. We will then pick a mid point of the array by adding the left and right variables, and then rounding down the division of the sum by 2. We will assign the result to a 'mid' variable. Using the mid variable (which will be 4 at index 4 here), we will check if the element at the index of the mid variable is the same as the element we are searching for. If not, we will see if the element we are searching for is greater or larger than the element stored at the mid variables index. In this case it will be larger so, the left variable will be set to mid+1. We will find a new mid point using the same method and assign it to the mid variable and repeat the same process as before. Element 5 at index 6 will be the mid point that is smaller than element 11 so, the new left variable will be element 6 at index 7. After comparison the left will be set to index 7, the same as the right variable. The mid point will be set to the same index. After comparing again, the left will move beyond index 8 at which point left will have exceeded right and the loop will break. False will be returned at the end, indicating the element searched for in the array does not exist.

If you made an image, put name of the image file here: binSearch

2. Consider the following piece of incomplete JavaScript that can be found in the JavaScript file `question2.js`:

```
function countTimes(array, item) {
  n = array.length;
  var count = 0;
  for (var i = 0; i < n; i++) {
    if (array[i] === item) {
      MISSING;
    }
  }
  return count;
}
```

When completed, this function should return the number of times that a value `item` appears in the argument `array`. It is based on a modified version of the Linear Search algorithm.

(a) In the JavaScript file, replace `MISSING` to complete this function. [2 marks]

(b) The worst-case time complexity of the algorithm implemented above is $O(n)$ for an array of length n. Explain why in the text box below. [5 marks]

It has O(n) worst-case time complexity, same as linear search. That's because in the worst-case where the element being searched for is at the very end of the array or not in it at all, the algorithm has a loop that must compare every value in the array that is passed to the function. So, for an array of size n, the algorithm must loop over O(n) for n elements.

3. Consider the following piece of JavaScript that can be found in the JavaScript file `question2.js`:

```
function newCountTimes(array, item) {
  var l = 0;
  var r = array.length - 1;
  var m;
  var count = 0;
  var i;
  while (l <= r) {
    m = Math.ceil(l + ((r - l)/2));
    if (array[m] === item) {
      count = 0;
      i = 0;
      while (array[m + i] === item) {
        count++;
        i++;
      }
      return count;
    } else if (array[m] > item) {
      r = m - 1;
    } else {
      l = m + 1;
    }
  }
  return count;
}
```

This function `newCountTimes` is a proposed method to return the number of times that a value `item` appears in a sorted array. However, this method does not work.

(a) In the text box below briefly explain why this method will not correctly determine the number of times a value `item` appears in an array. In your explanation, include an example `array` and argument `item` of when this method does not work. [7 marks]

Everything is done correctly except for line 8, which is the reason why this method does not work. Because of how it calculates its mid point. It rounds up fractional values instead of rounding them rounding down. This causes it to skip a single value while searching, because a binary search algorithm only compares the searched for element with the value of the mid point in the array. But when the sub array becomes too small (like 2 elements small), the midpoint becomes equal to the right pointer instead of becoming equal to the left pointer first. This way the mid pointer never gets to check the value stored at the left pointer because the mid pointer is set equal to the right pointer because of the rounding up, then the left pointer just moves to meet the right pointer without the mid pointer having come. This leaves the value stored at the mid pointer unchecked. Such a case can be seen if you run the command console.log (newCountTimes(arrayS,5)); on the array, [0,1,3,3,4,5,5,6,7]. It will only count 5 once.

(b) In the text box below describe a new method to return the number of times a value appears in a sorted array. This new method should have a better worst-case time complexity than the completed function `countTimes`. You should explain the method, not just give code. You should also explain why it has better time complexity. [8 marks]

countTimes was a linear search algorithm with worst-case time complexity of O(n). newCountTimes is a binary search algorithm and we know that the worst-case time complexities of binary search algorithms is O(log n). If we were to simply fix newCountTimes by changing line 8 to m = Math.floor((l+r)/2), so that it no longer skips over values. This would give us an algorithm that returns the number of times a value appears in a stored array and with a better worst-case time complexity than countTimes. Binary search is quicker than linear search at searching for an item, the only drawback being that the array given to a binary search algorithm must be sorted before hand whereas it makes no difference to a linear search algorithm. The only other type of algorithm with a better worst-case time complexity than a linear search algorithm would be a ternary search algorithm. However, it is not faster than binary search. They both have the same worst-case time complexity.