Online Test: Part B, Question 1

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 1

This question is about sorting. 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.

0	3	3	6	1	5	7	5	4

Implement the Insertion Sort algorithm, by hand, on this nine-digit array so that the smallest value is in the leftmost element, and the largest value is in the rightmost element. Show how the array changes in the steps in the algorithm and how choices are made to change the values in the array. 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:

To start, we pass the length of the array to a variable and set a variable i to zero. Then we run our first loop while index i is less than the length of the array. We then set var j to index i for our second loop. The second (nested) loop will run while j is less than zero and j-1 is greater than j. The second loop will swap j with j-1 and decrement j. Starting with value 3 at index 1, it is greater than zero so i will move to index 2. This is also value 3 which is not greater than 3 at the previous index so no swap will happen. Then at index 3 with value 6, no swap will occur. At index 4 with value 1, it will compare it with the values of index 3,2,1 and swap 3 times until it stops at index 1 because index 0 contains value 0 which is not greater than value 1. At index 5 with value 5, it will swap once with the value at index 4. At this point, the values of our array are in the order [0,1,3,3,5,6,7,5,4]. Then at index 6 with value 7 a swap will occur as it is greater than the value at the previous index. At index 7 with value 5, two swaps will occur with the values at index 6 and 5. It will not swap further as the value at index 5 is the same (not greater technically speaking)as the value we are comparing with. At the final index (8), the value 4 will be swapped 4 times running through indexes 7,6,5,4 where it will stop to take up residence at index 4 as it is greater than the values of lower index values. Here our insertion sort will end with the final array = [0,1,3,3,4,5,5,6,7].

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

2. Consider the following piece of JavaScript describing a sorting algorithm:

```
function swap(array,i,j) {
  var store = array[i];
  array[i] = array[j];
  array[j] = store;
  return array;
}

function sort(array) {
  var n = array.length;
  for (var i = 1; i < n; i++) {
    var count = 0;
    for (var j = n - 1; j > i - 1; j--) {
      if (array[j] < array[j-1]){
        swap(array,j,j-1);
        count++;
      }
    }
    if (count === 0) {
      return array;
    }
  }
  return array;
}
```

(a) Which of the following is the worst-case time complexity in n of the sorting algorithm for an array of length n? Click on the box next to your answer (to the right). [2 marks]

$O(n)$ □ $O(n \log n)$ □ $O(n^2)$ ☑ None of the other options □

(b) Explain your answer to part 2a by making reference to the worst-case input array. Give your answer in the text-box below. [7 marks]

This algorithm works largely the same as the algorithm for insertion sort except that it starts by grabbing values from the far right and working backwards instead of starting at index 1 of the array. As such I believe that the worst-case time complexity of this algorithm is O(n^2). This algorithm has two loops with one of those two loops being nested. Assuming the worst case Assuming the worst input where every element of the array given to the function will be out of order and need to be swapped at least once if not more than once, the nested loop will have to compare every element of the whole array with every element of i the outer loop is going through. And, if statements have a constant worst-case time complexity of O(1). So the combined worst case time complexity comes to O(n^2).

3. Go to the file called question1.js and you will find the following incomplete JavaScript code:

```javascript
function swap(array,i,j) {
  var store = array[i];
  array[i] = array[j];
  array[j] = store;
  return array;
}

function recSort(array, r) {
  if (r === array.length - 1) {
    return array;
  }
  for (var j = array.length - 1; j > r - 1; j--) {
    if (array[j] < array[j-1]){
      swap(array,j,j-1);
    }
  }
  return MISSING;
}
```

This code is meant to recursively implement the same algorithm as in part 2 of this question, such that the function call recSort(array, 0) will sort array in ascending order.

(a) In the JavaScript file, replace MISSING with a recursive function call of recSort to correctly implement the sorting algorithm. [2 marks]

(b) Also in the JavaScript file, introduce a variable count to reduce the typical number of recursive function calls required to sort the argument array. [3 marks]

(c) A colleague has claimed that it is better to implement an iterative version of Insertion Sort, rather than the recursive function recSort. Explain whether you agree or disagree with this claim, and why you agree or disagree. Put your answer in the text box below. [8 marks]

I disagree with my colleagues claim. The better implementation of insertion sort is as a recursive function. Recursive functions hold many advantages over traditional iterative algorithms. The recursion allows the sorting code here to be half the size of the iterative version which in turn makes the code simpler. A shorter code is quicker to write and debug, not to mention also quicker to understand by a different person when there is physically less code to look over. Recursion here reduces the time complexity of the function as it no longer needs to run a nested loop, even if it is at the expense of extra memory. Not to mention some recursive functions are more universal as some algorithms are recursion based. Having functions written in recursion will save you the time and headache of adapting code for different languages that are recursion-based.