

# CMPSCI 611 - Advanced Algorithms

## Homework 1

Jennie Steshenko (Collaboration: Emma Strubell)

### Question 1

#### Part A

The order of growth of functions is as follows

$$g_7 = (n^{1/\log_2 n})^3 = 8,$$

$$g_{10} = (\log n)^2,$$

$$g_1 = 2\sqrt{\log_2 n},$$

$$g_3 = n(\log_2 n)^3,$$

$$g_9 = n^{1+1/\log \log n} \approx n^{1.2},$$

$$g_4 = n^{4/3},$$

$$g_{11} = (\log n)^{\log n},$$

$$g_5 = n^{\log_2 n},$$

$$g_6 = 2^n,$$

$$g_{12} = \pi^n = 3.14^n,$$

$$g_8 = n!,$$

$$g_2 = 2^{2^n}$$

#### Part B

1.  $\log f(n)$  is  $O(\log g(n))$  - **True**

$$O(\log(g(n))) = \log(cf(n)) = \log c + \log f(n)$$

$$\lim_{x \rightarrow \infty} \frac{\log c + \log f(n)}{\log f(n)} = \lim_{x \rightarrow \infty} \left( \frac{\log c}{\log f(n)} + \frac{\log f(n)}{\log f(n)} \right) = 0 + 1 = 1$$

2.  $2^{f(n)}$  is  $O(2^{g(n)})$  - **False**

Let us take  $f(n) = x$  and  $g(n) = 2x$  then:

$$\lim_{x \rightarrow \infty} \frac{2^{2x}}{2^x} = \lim_{x \rightarrow \infty} \frac{2^x \cdot 2^x}{2^x} = \lim_{x \rightarrow \infty} 2^x = \infty$$

3.  $(f(n))^2$  is  $O((g(n))^2)$  - **True**

$$O((g(n))^2) = (cf(n))^2 = c^2(f(n))^2$$

$$\lim_{x \rightarrow \infty} \frac{c^2(f(n))^2}{(f(n))^2} = c^2$$

## Question 2

### Proposed Algorithm

The assumptions and notations used in this algorithm:

- Arrays: A, B
- Array sizes: sizeA, sizeB (e.g. the index of the last cell in array A that should be checked)

The Algorithm: The general idea is to look for the  $K^{th}$  smallest item by scanning the arrays interminently, in a manner similar to a binary search.

### Question 3

The algorithm to solve this problem is very similar to the MergeSort algorithm, with one additional variable to count the number of inverted pairs, and one additional line in the algorithm to track the value of the variable.

*The suggested algorithm is a modification to an array based algorithm of the MergeSort algorithm on Wikipedia.org*

```
function CountInvertedPairs(intArray intArr, int start, int end)
// if list size is 1 consider it sorted and return it
if sizeof(intArr) <= 1
    return intArr
// else array size is > 1, so split the array into two subarrays
// The counter of inverted pairs, as global
var integer invertedCount = 0
// The start and end of the left of the subarray
var int leftStart, leftEnd
// The start and end of the right of the subarray
var int rightStart, rightEnd
var integer middle = floor((end - start) / 2)
// recursively call CountInvertedPairs() to further split each
// sublist until subarray size is 1
left = CountInvertedPairs(intArr, start, middle)
right = CountInvertedPairs(intArr, middle + 1, end)
// merge the sublists returned from prior calls to
// CountInvertedPairs() and return the resulting merged sublist
return merge(left right invertedCount)

function merge(intArray left, intArray right, invertedCount)
var intArray result
// Indexes to each of the arrays - left, right and result
var int leftIndex = 0, rightIndex = 0, resIndex = 0
while sizeof(left) > 0 or sizeof(right) > 0
    if sizeof(left) > 0 and sizeof(right) > 0
        if left[leftIndex] <= right[rightIndex]
            result[resIndex] = left[leftIndex]
            leftIndex++
            resIndex++
        else
            result[resIndex] = right[rightindex]
            rightIndex++
            resIndex++
            invertedCount = rightIndex - resIndex
    else if sizeof(left) > 0
        result[resIndex] = left[leftIndex]
        leftIndex++
```

```
        resIndex++
    else if sizeof(right) > 0
        result[resIndex] = right[rightindex]
        rightIndex++
        resIndex++
        invertedCount = rightIndex - resIndex
    end while
    return result
```

## Question 4

Assumption:

- The set  $S$  is stored in an array
- All the numbers in the given set are less than  $x$ . If not, a simple modification to the MergeSort algorithm can exclude all irrelevant numbers.
- None of the numbers in the set appears more than one. If not, a simple modification to the MergeSort algorithm can exclude all irrelevant numbers.

The algorithm:

- Run `arrayMergeSort(set S)` - complexity of  $O(n \log n)$   
(The same algorithm as presented in the solution to Question 3, without the counter and operations for counting inverted pairs)
- Use a binary search to find *half* - all members of the array to the left of this number are smaller or equal to  $x/2$ , and the members to the right of the array are bigger or equal to  $x/2$  - complexity of  $O(\log(n))$
- If  $\text{indexOf}(\text{half}, \text{completeArray}) < (\text{sizeof}(\text{completeArray}))/2$  then
  - `matchToArr[start] = 0`
  - `matchToArr[end] = indexOf(half, completeArray)`
  - `matchFromArr[start] = indexOf(half, completeArray) + 1`
  - `matchFromArr[end] = sizeof(completeArray)`Else
  - `matchToArr[start] = indexOf(half, completeArray) + 1`
  - `matchToArr[end] = sizeof(completeArray)`
  - `matchFromArr[start] = 0`
  - `matchFromArr[end] = indexOf(half, completeArray)`
- For Each Item in `matchToArray` - complexity of  $O(n \log(n))$ 
  - Use `binarySearch` in the `matchFromArray` to find the value  $x - \text{Item}$
  - If `binarySearch` returns NOT NULL  
Return TRUE and break
  - Return FALSE

The highest complexity is encountered when the sorting of the complete set is executed. The sum matching process will run  $m \log(k)$  times, where at most  $m = k = n/2$ , and in most cases  $m < n/2$ , hence the total matching process will run faster than  $O((n/2) \log(n/2))$

### Question 5

The algorithm used to build spanning trees with minimal weight is Kruskal's algorithm, which is greedy. At each iteration, the algorithm chooses the next best candidate, in this case, the next edge with minimal weight that is available for choosing.

In order to have two different trees with the same minimal weight, the algorithm must be able to choose 2 different subsets of edges that have the same sum of weights. Since the algorithm is greedy, that can happen only in the case when there are 2 edges with the same weight, that lead to different nodes, thus allowing the existence of different final sets when the algorithm has finished running.

However, it is indicated that all edges in the given set are of different weight, thus every time the algorithm will run it will make exactly the same choices, thus each time the exact same minimal spanning tree will be built.