CMPSCI 611 - Advanced Algorithms
Homework 1

Jennie Steshenko (Collaboration: Emma Strubell)

Question 1

Part A

The order of growth of functions is as follows
$g_7 = (n^{1/\log_2 n})^3 = (n^{\log_2 n/\log_n n})^3 = (2^{\log_n n/\log_n n})^3 = 2^3 = 8$,
$g_{10} = (\log_2 n)^2$,
$g_1 = 2^{\sqrt{\log_2 n}}$,
$g_3 = n(\log_2 n)^3$,
$g_9 = n^{1+1/\log_2 \log_2 n} = n \times n^{1+1/\log_2 \log_2 n}$,
$g_4 = n^{4/3} = n^{1+1/3} = n \times n^{1/3}$,
$g_{11} = (\log_2 n)^{\log_2 n}$,
$g_5 = n^{\log_2 n}$,
$g_6 = 2^n$,
$g_{12} = \pi^n = 3.14^n$,
$g_8 = n! \approx n^n$,
$g_2 = 2^{2^n}$

Explanations for the ordering

$g_{10} = O(g_1)$:
$\lim_{n\to\infty} \frac{\log^2 n}{2^{\sqrt{\log n}}} = \lim_{n\to\infty} \frac{2\log n}{2^{\sqrt{\log n}}} = 2\lim_{n\to\infty} \frac{\log\log n}{\log 2^{\sqrt{\log n}}} = 2\lim_{n\to\infty} \frac{\log\log n}{\sqrt{\log n}} =$
$2\lim_{n\to\infty} \frac{\log\log\log n}{\log\log^{1/2} n} = 2\lim_{n\to\infty} \frac{\log\log\log n}{1/2\log\log n} = 4\lim_{n\to\infty} \frac{\log\log\log n}{1\log\log n} = 0$
$\log\log\log n$ grows slower than $\log\log n$, hence the ratio test gives 0

$g1 = O(g3)$:
$\lim_{n\to\infty} \frac{g_1}{g_3} = \lim_{n\to\infty} \frac{2^{\sqrt{\log_2 n}}}{n\log_2^3 n} = \lim_{n\to\infty} \frac{\sqrt{\log_2 n}}{\log_2(n\log^3 n)} = \lim_{n\to\infty} \frac{\sqrt{\log_2 n}}{\log_2 n + \log_2\log^3 n} =$
$\lim_{n\to\infty} \frac{\sqrt{\log_2 n}}{\log_2 n + 3\log_2\log n} = 0$
$\log n$ grows faster than $\sqrt{\log n}$, hence the ratio test gives 0, and $2^{\sqrt{\log n}} \in o(n\log^3 n)$

$g_3 = O(g_9)$:
$n^{\frac{1}{\log_2\log_2 n}} \,?\, (\log_2 n)^3$
$\frac{1}{\log_2\log_2 n} \,?\, \log_n(\log_2 n)^3 = \frac{\log_2(\log_2 n)^2}{\log_2 n} = \frac{3\log_2(\log_2 n)}{\log_2 n}$
$\frac{1}{\log_2\log_2 n} \,?\, \frac{3\log_2\log_2 n}{\log_2 n}$
$\log_2 n \,?\, 3(\log_2\log_2 n)^2$

$\sqrt{\log_2 n}\,?\,\sqrt{3}\sqrt{(\log_2 \log_2 n)^2}$
$\sqrt{\log_2 n}\,?\,\sqrt{3}\log_2 \log_2 n;\ \{\log_2 n = x\}$
$\lim_{x\to\infty} \frac{\sqrt{3}\log_2 x}{\sqrt{x}} = 0$

$g_9 = O(g_4)$:
$\lim_{x\to\infty} \frac{1}{\log_2 \log_2 n} = 0$
g_4 will be bigger than $g_9\ \forall n > n_0.(\frac{1}{\log_2 \log_2\ n_0} = 1/3)$

$g4 = O(g11)$:
$\lim_{n\to\infty} \frac{n^{4/3}}{\log n^{\log n}} = \lim_{n\to\infty} \frac{\log n^{4/3}}{\log\log n^{\log n}} = \lim_{n\to\infty} \frac{4/3\log n}{\log^2 n} = \lim_{n\to\infty} \frac{4/3}{\log n} = 0$

$g_{11} = O(g_5)$:
$\log n^{\log n} = \overbrace{\log n \times \log n \times ... \times \log n}^{k=\log n\ \text{times}}$
$n^{\log n} = \overbrace{n \times n \times ... \times n}^{k=\log n\ \text{times}}$ Since $\log n \in O(n)$, clearly $\log n^{\log n} \in O(n^{\log n})$

$g_5 = O(g_6)$:
$\lim_{n\to\infty} \frac{n^{\log n}}{2^n} = \lim_{n\to\infty} \frac{\log n^{\log n}}{n} = \lim_{n\to\infty} \frac{\log_n n^{\log n}/\log_n 2}{n} = \lim_{n\to\infty} \frac{\log n}{\log_n 2\cdot n} = 0$
Because $\lg n \in O(n)$, the ratio test gives 0 and $n^{\lg n} \in o(2^n)$.
$g_{12} = O(g_8)$:
$\pi^n = \overbrace{\pi \times \pi \times ... \times \pi}^{n\times}$
$n! = 1 \times 2 \times 3 \times 4 \times ... \times n$
g8 = O(g2) (i think the change-of-base stuff is correct in this one, have to go back and re-do the others):
We know by Stirling's approximation that $n! = O(n^n)$. So, it is sufficient to show that $n^n \in O(2^{2^n})$
$\lim_{n\to\infty} \frac{n^n}{2^{2^n}} = \lim_{n\to\infty} \frac{\lg n^n}{2^n} = \lim_{n\to\infty} \frac{\log_n n^n/\log_n 2}{2^n} = \lim_{n\to\infty} \frac{n}{\log_n 2\cdot 2^n} = 0$
Because $\log_n 2$ is a constant and $n \in O(2^n)$, the ratio test gives zero and $n^n \in o(2^{2^n})$, which implies that $n! \in O(2^{2^n})$
We know by Stirling's approximation that $n! = O(n^n)$. So, it is sufficient to show that $n^n \in O(2^{2^n})$. $\lim_{n\to\infty} \frac{n^n}{2^{2^n}} = \lim_{n\to\infty} \frac{\lg n^n}{2^n} = \lim_{n\to\infty} \frac{\log_n n^n/\log_n 2}{2^n} = \lim_{n\to\infty} \frac{n}{\log_n 2\cdot 2^n} = 0$;
Because $\log_n 2$ is a constant and $n \in O(2^n)$, the ratio test gives zero and $n^n \in o(2^{2^n})$, which implies that $n! \in O(2^{2^n})$

Part B

1. $logf(n)\ is\ O(logg(n))$ - **True**
   $O(log(g(n))) = log(cf(n)) = logc + logf(n)$
   $\lim_{x\to\infty} \frac{logc+logf(n)}{logf(n)} = \lim_{x\to\infty}(\frac{logc}{logf(n)} + \frac{logf(n)}{logf(n)}) = 0 + 1 = 1$

2. $2^{f(n)}$ *is* $O(2^{g(n)})$ - **False**
   Let us take $f(n) = x$ and $g(n) = 2x$ then:
   $\lim_{x\to\infty} \frac{2x}{x} = lim_{x\to\infty} 2\frac{x}{x} = 2$
   $\lim_{x\to\infty} \frac{2^{2x}}{2^x} = \lim_{x\to\infty} \frac{2^x * 2^x}{2^x} = \lim_{x\to\infty} 2^x = \infty$
3. $(f(n))^2$ *is* $O((g(n))^2)$ - **True**
   $O((g(n))^2) = (cf(n))^2 = c^2(f(n))^2$
   $\lim_{x\to\infty} \frac{c^2(f(n))^2}{(f(n))^2} = c^2$

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 intermintently, 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 $O(nlogn)$
  (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 $indexOf(half, completeArray) < (sizeOf(completeArray))/2$ 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 $O(nlog(n))$
  - Use binarySearch in the matchFromArray to find the value $x - Item$
  - If binarySearch returns NOT NULL
    Return TRUE and break
  - Return FALSE

The highest comlexity is encountered when the sorting of the complete set is executed. The sum matching process will run $mlog(k)$ 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

This is a simple proof by induction. For n = 1: There is only one edge, hence it's possible to construct only one minimal spanning tree
For n = k - 1: Assume that (k-1) edges were chosen by now, and they resulted only in one minimal spanning tree
For n = k: An additional edge needs to be chosen at this point. Kruskal's algorithm is a greedy algorithm and it always chooses the edge with the minimal weight available. It is known that there are no to edges with the same weight, hence when coming to choose, the group of edges with minimal weight is of size 1, and only that edge will be added to the minimal spanning tree. This will repeat for each k, $0 < k < n$ according to the problem statement.