CSE 373 Analysis of Algorithms March 3, 2015

Solutions to Midterm 1

Problem 1

[1] The functions from lowest to highest order:

$$(\tfrac{1}{3})^n \mid 6 \mid \log\log n \mid \log n, \ln n \mid (\log n)^2 \mid n^{1/3} + \log n \mid \sqrt{n} \mid$$
$$\tfrac{n}{\log n} \mid n \mid n\log n \mid n^2, n^2 + \log n \mid n^3 \mid n - n^3 + 7n^5 \mid (\tfrac{3}{2})^n \mid 2^n \mid n!$$

[2] $f_1(n) = O(g_1(n)) \Leftrightarrow \exists n_1, c_1$, such that $\forall n \geq n_1, f_1(n) \leq c_1 g_1(n)$.

$f_2(n) = O(g_2(n)) \Leftrightarrow \exists n_2, c_2$, such that $\forall n \geq n_2, f_2(n) \leq c_2 g_2(n)$.

Thus, simply pick $n_3 = max(n_1, n_2)$ (you must include this step!), $c_3 = c_1 \cdot c_2$, and we have:

$\forall n \geq n_3, f_1(n) \cdot f_2(n) \leq c_1 g_1(n) c_2 g_2(n) \leq c_3 g_1(n) g_2(n)$.

Thus: $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$

Problem 2

The data structure we use is:

```
struct stack_element
{
    element x;
    element minx;
}
```

The basic idea is, for each element x, we also store the minimum element in the interval from this element to the bottom of the stack. For each operation:

- Push(x): Set current minimum $curmin$ to be $min(x, S.Top().minx)$, and push $(x, curmin)$ onto the stack.
- Pop(): The same as a regular stack: remove the top element.
- Top(): The same as a regular stack: return the top element.
- getMin(): Return $S.Top().minx$.

Apparently each opearation takes $O(1)$.

A wrong algorithm is to use only one pointer/value to record the minimum element in the stack. Suppose the element we pop is the minimum element, then this algorithm fails to find the new minimum element.

Problem 3

Check if the *maximum* value in the left subtree is smaller than current node and the *minimum value* in right subtree is larger than current node. Repeat this for each node. This takes $O(n)$ time because we check each node for only once.

Aonther easy solution is to do an in-order traversal of the tree and check if it is in ascending order. This takes $O(n)$ time.

Note: Simply check if left child is smaller and right child is larger is incorrect. See the example below:

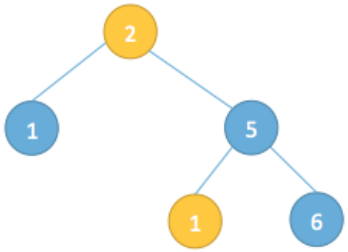

Figure 1. Not a binary search tree

Problem 4

Two steps:

[1] Sort the last $\sqrt{n}$ element using any sorting algorithm. Let's say we are using bubble sort, and this step takes $O(n)$.

[2] Merge the first $n-\sqrt{n}$ elements with the last $\sqrt{n}$ elements using the same method as merge sort. This step also takes $O(n)$. In total the time complexity is $O(n) = o(nlogn)$.

Problem 5

We maintain a minimum heap (priority queue) for current piles we have. For each step, we merge the two piles with minimum number of stones, until there's only one pile left. A single merge operation consists of four steps:

(1) Get the smallest element a in the heap and delete it, which takes $O(logn)$
(2) Get the current smallest element b and delete it, which takes $O(logn)$
(3) Insert an element $a+b$ in the end of the list, which takes $O(logn)$.
(4) Add $a+b$ to the total amount of work we have done, which takes $O(1)$.

Each step takes $O(logn)$, and we have $n-1$ steps in total. Thus, the time complexity is $O(nlogn)$.

A slower algorithm is to use an array or linked list to maintain the piles, which takes $O(n^2)$. A wrong algorithm is to "pair" the original piles, and merge each pair. This method fails to deal with the merged piles, thus cannot guarantee that we are always merging two smallest piles.