CSE 2011: Assignment 1

Due Date - Monday, Feb 16, in class!!!

Question 1 **Big-O Analysis** [5 points]

Sort the following functions by increasing order of growth:

$$n \lg n \quad n^{\lg n} \quad (\lg n)^n \quad 2^{\frac{\lg n}{2}} \quad n^2$$

Solution:

$$2^{\frac{\lg n}{2}} = \sqrt{n} \quad n \lg n \quad n^2 \quad n^{\lg n} \quad (\lg n)^n$$

Question 2 **Running Time Analysis** [10 points]

Express the running time of the following program segment using big-θ notation. Show/justify your work!

```
int k=1, sum=0;
for (int i=0; i<n; i++) {
   k=2*k; }
for (int j=k; j>1; j=j/3) {
   sum++; }
```

Solution:

First loop takes θ(n) time, k reaches 2^n.
The second loop takes $\log_3(2^n) = n*\log_3(2) = n*c$, which is also θ(n).
Overall running time: θ(n).

Question 3 **Algorithm Design** [10 points]

Array A of size n contains all integers form 1 to (n+1) but one!

1) Assuming the array is sorted, propose an (best-running time) algorithm that finds/identifies the missing number, and analyze its worst-case running time in terms of big-O notation

2) Assuming the array is NOT sorted, propose an (best-running time) algorithm that finds/identifies the missing number and analyze its worst-case running time in terms of big-O notation.

Solution:

Solution:

1. If the array is sorted, we can use binary search. We are looking for the smallest index i for which $A[i] = i+1$; this will be our missing number. If $A[n/2] = n/2+1$, i is less than or equal to $n/2$, and we can recurse on the first half of A; otherwise it is greater than $n/2$, and we can recurse on the second half of A. In either case we get the recurrence $T(n) = T(n/2)+\Theta(1)$ which gives a worst-case running time of $\Theta(\log n)$.

2. If the array is not sorted, then in the worst case we will have to look at every location in the array (otherwise what we think is the missing element could be in the location we didn't look in). So the best we can hope to do is get an $O(n)$ algorithm. One such algorithm creates an auxiliary array B with indices 1 to $n+1$, initializes all locations to 0, scans through A setting $B[A[i]] = 1$ for each i from 1 to n, and finally scans through B looking for a zero. Each of these three steps takes $O(n)$ time, so we have an $O(n)$ algorithm for this case.

Question 4 **ADTs / Stacks** [10 points]

Using the regular Stack ADT, design an advanced Stack ADT for which getMinimum() method runs in O(1) time. Pushing and popping n elements to this Stack should each run in O(n) (i.e., you are not allowed to perform any sorting (e.g.) as a part of a different method). You are only allowed to use one additional data structures (e.g., another Stack) in your design.

Describe your solution in words and provide a brief pseudocode for the new push() and getMinimum() methods.

Solution:

Take an auxiliary stack that maintains the minimum of all values in the main Stack. For simplicity, let us call the auxiliary stack MinStack.

When we push new element into the main Stack, we also push this element into the MinStack if the element is smaller than the current minimum (i.e., top of MinStack). For an illustration, see the below Figure.

Stack	MinStack
5 -> top	
1	
4	
6	1 - > top
2	2

Now, if at any point we want to get the minimum, then we just need to return the top element from the MinStack.

Regarding the pop – for getMinimum() to work properly, we should only pop from the MinStack when the value we pop from the main Stack is equal to the one on the MinStack.

Example: after popping 5 from the main Stack

Stack	MinStack
1 -> top	
4	
6	1 - > top
2	2

Example: after popping 1 from the main Stack

Stack	MinStack
4 -> top	
6	
2	2 - > top