University of Toronto
ECE-345: Algorithms and Data Structures
Solutions to Midterm Examination (Fall 2007)

1. (a) $A(n) = \Theta(n^{\log_5(3)})$

   (b) We can draw a recursion tree like we did in class. At one part of the tree goes $n/5$-th of the work while on the other part it goes $4n/5$ of the work for the recurrence. Clearly, at each level we do $\Theta(n)$ work while the number of tree levels is $log_{5/4}n = \Theta(\log n)$. Therefore, $B(n) = \Theta(n \log n)$

2. (a) Function `foo` is similar to quicksort, except that it recurs in only one of the two subarrays formed by `partition`. The call `foo(a,1,N,k)` will return the k-th smallest number in the array. Namely, after each partitioning, element $a[i]$ is in its final place. If $k = i$, we are done. Otherwise, the k-th smallest number must be in $a[l], a[l+1], \cdots, a[i-1]$ if $k < i$, and in $a[i+1], a[i+2], \cdots, a[r]$ if $k > i$, so that we can recursively solve the problem by searching in the proper subarray.

   (b) The worst-case time complexity occurs when $k = 1$ and the array is already sorted. In this case, all the possible branches in a recurrence tree will be visited. Terefore the worst-cast time complexity is $\Theta(N + (N-1) + \cdots + 1) = \Theta(N^2)$.

3. **Base Case**: $n = 1$ can be represented as 2^0.

   Inductive hypothesis: Suppose $n = k$, can be represented as

   (a) $n = 2^i + .. + 2^1$ when k is even.

   (b) $n = 2^i + .. + 2^0$ when k is odd.

   Induction Step:

   (a) When $n = k+1$, 1 can be represented as 2^0 and from hypothesis k can be represented as $2^i + .. + 2^1$, hence $n = k + 1$ can be represented as $n = 2^i + .. + 2^1 + 2^0$

   (b) When $n = k+1$, 1 can be represented as 2^0 and from hypothesis k can be represented as $2^i + .. + 2^0$. Adding 2^0 to this k, causes lowest missing power of 2 to increment by one and all other lower power terms of 2 to vanish e.g., consider adding 2^0 to $2^1 + 2^0$: lowest missing power of 2 is 2^2 so the sum would be 2^2 and all lower powers i.e., 2^1 and 2^0 vanish.

4. Let us call the original heap H. The idea is to generate a separate heap called F_i of the "frontiers" of H, which at the i'th step of our algorithm will always have all the candidates for the i'th largest number of H. The algorithm functions recursively as follows: to create F_i we extract the maximum number from F_{i-1} and insert its two children from H into the resulting heap (we can think of the nodes of F as pointers to nodes of H in order to make this computation efficient). The number extracted from F_{i-1} would be the $i-1$'th largest number in H.

   For example, F_1 would trivially be the root of H. When we extract it and look at its children in H, we get the candidates for the second largest number of H. Similarly, once we extract the largest number of F_2 and look at its children in H, we have a heap of all possible candidates for the third largest number of H. Altogether, computing F_i from F_{i-1} requires 1 `extract-max` and 2 `insert` operations and some constant overhead. We are `insert`ing at most $2k$ elements, bounding the size of of F as $O(k)$. Since both of these heap operations require $O(\log k)$ time on heaps of size $O(k)$, we have an overall running time of $O(k \log k)$.

5. (a) $OPT(i, v)$ is the maximum prize that can be collected with a connected subtree rooted at v if at most i nodes can be selected.

   (b) If no nodes can be selected, then the maximum prize is always 0 (first case). If at least one node can be selected, then a leaf can be selected, so we can get p_v from it (second case). If $i + 1 \geq 1$ nodes can be

selected from a subtree rooted at an internal node v, then this must include the node v itself, to maintain connectivity. The remaining i nodes can be distributed in any way between the two subtrees. The optimum solution distributes them in the best possible way, hence it gets the maximum over all legal distributions.

(c) We simply output $OPT(k, r)$, the maximum prize that can be selected with any connected subtree rooted at r selecting at most k nodes.

(d) We fill in a table of size $O(nk)$. Each update goes over at most $O(k)$ different divisions, so a bottom up implementation takes $O(nk^2)$.