1. [15 points] We have described a randomized algorithm for Selection() in class that runs in linear time. It turns out that there is a deterministic algorithm that has the same asymptotic performance. More precisely, there exists an algorithm $\text{Det-Selection}(A, p, q, r)$ such that, given an array $A$, it returns the $r$-th smallest element in the subarray $A[p, \ldots, q]$ in time $O(q - p + 1)$. For example, $\text{Det-Selection}(A, p, q, 1)$ returns the smallest element in the subarray $A[p, \ldots, q]$, while $\text{Det-Selection}(A, p, q, 2)$ returns the second smallest element in the subarray $A[p, \ldots, q]$.

Using this deterministic version of Selection procedure, give a deterministic version of the QuickSort algorithm that can sort an array in $O(n \log n)$ worst case running time. [Hint: Essentially, you need to choose the pivoting element in your Partition() procedure more carefully. ]

You can assume that all elements in the input array have distinct values. 

More specifically, you would need to provide pseudo-code for procedure $\text{Det-QuickSort}(A, p, q)$ which sorts the subarray $A[p, \ldots, q]$ by a QuickSort like procedure (i.e, partition then recursive calls). You can use $\text{Det-Selection}(A, p, q, r)$ as a sub-routine (i.e, you don’t need to write its pseudo-code). To sort an array $A$, you will then call $\text{Det-QuickSort}(A, 1, n)$.

You also need to provide the worst case time complexity analysis of your algorithm.

2. [20 points] Given a set of $n$ distinct numbers, we wish to find the $k$ largest in sorted order using a comparison-based algorithm. Give an algorithm that implements each of the following methods, and analyze the running times of the algorithms in terms of $n$ and $k$ – you should provide an upper bound for the worst-case time complexity if your algorithm is deterministic, and expected running time if your algorithm is randomized. You should make your algorithm as fast (asymptotic) as possible and your bound tight. Slower algorithms /analysis receive fewer point. (What we describe below are not full algorithms. You need to give pseudo-code to implement each idea as an algorithm, and output the $k$ largest numbers in sorted order. In your pseudo-code, you can use any algorithm we described in class as subroutine: such as MergeSort(), Heap-ExtractMax(), etc. )

2.a Sort the numbers, and then list the $k$ largest.

2.b Build a max-priority queue from the numbers, and then extract the largest $k$ elements.

2.c Use a selection algorithm to find the $k$-th largest number, and the retrieve the largest $k$ numbers.

3. [ 5 points] Draw the binary tree produced by inserting the following elements in a max-heap in the given order:

$$4, 8, 3, 0, 10, 15, 12$$

(Show your work by showing the partial trees after each insertion. )

4. [15 points] Consider the following max heap in Figure 1:

![Max Heap Diagram]

Figure 1.

4.a Apply the operation Insert(55) to the heap in Figure 1, and draw the resulting binary tree. (Show your work by showing the key intermediate steps.)
Program5($A, n$)
/* $A$ is an array of $n$ elements */
1 P.Initialize();
2 for $i \leftarrow 1$ to $n$ do
3   for $j \leftarrow 1$ to $n$ do
4     P.Insert($A[i] \ast A[j]$);
5   end
6 while P.Size() $\neq 0$ do
7     $x \leftarrow$ P.ExtractMax();
8     Print $x$;
9 end
10 Print $x$;
11 end

4.b Apply the operation ExtractMax() operation to the heap in Figure 1 (without applying the operation in (4.a)). Show your work.

5. [10 points] Consider the following program:
P.Initialize() initializes the data structures.
P.Insert($x$) inserts elements $x$ in $P$.
P.Maximum() returns the maximum element of $P$ (without deleting it from $P$).
P.Size() returns the number of elements in $P$.

Analyze carefully the (asymptotic) running time of Program5 assuming that $P$ is implemented as a Max-Heap. Note that the time for P.Insert() is dependent on the number of elements in $P$ which changes over the running time of the algorithm. Operations P.Initialize() takes constant time.

Show your work of how you obtained the time complexity.

6. [10 points] Consider the following program:

Program6($A, n$)
/* $A$ is an array of $n$ elements */
1 P.Initialize();
2 for $i \leftarrow 1$ to $n$ do
3   $j \leftarrow 1$;
4   while $j \leq n$ do
5     P.Insert($A[i] \ast A[j]$);
6     $j \leftarrow j \ast 2$;
7   end
8 while P.Size() $\neq 0$ do
9     $x \leftarrow$ P.ExtractMax();
10    Print $x$;
11 end

P.Initialize() initializes the data structures.
P.Insert($x$) inserts elements $x$ in $P$.
P.ExtractMax() returns the maximum element of $P$ and deletes it from $P$.
P.Size() returns the number of elements in $P$.

Analyze carefully the running time of Program6 assuming that $P$ is implemented as a Max-Heap. Note that the time for P.Insert() and P.ExtractMax() is dependent on the number of elements in $P$ which changes over the running time of the algorithm. Operations P.Initialize() and P.Size() take constant time.

Show your work.