1. Given the following integer elements: 85, 25, 45, 11, 3, 30, 1, 6

a. Draw the tree representation of the heap that results when all of the above elements are added (in the given order) to an initially empty minimum binary heap. Circle the final tree that results from performing the additions.

b. After adding all the elements, perform 3 removes on the heap. Circle the tree that results after the two elements are removed.

Please show your work. You do not need to show the array representation of the heap. You do not have to draw an entirely new tree after each element is added or removed, but since the final answer depends on every add/remove being done correctly, you may wish to show the tree at various important stages to help earn partial credit in case of an error.

2. The array below is to be sorted in ascending order.

17, 53, 71, 62, 36, 46, 41, 23, 12

a. After the initial partition step of the version of quicksort discussed in lecture, with 36 as the pivot, how would the array be ordered?

b. After the initial iteration of bubble sort, how would the array be ordered?

3. Sorting variable-length items

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time

4. You wish to store a set of n numbers in either a max-heap or a sorted array. For each application below, state which data structure is better, or if it does not matter. Explain your answers.

a. Want to find the maximum element quickly.

b. Want to be able to delete an element quickly.

c. Want to be able to form the structure quickly.

d. Want to find the minimum element quickly

5. The operation `HEAP-DELETE(A,i)` deletes the item in node `i` from heap `A`. Give an implementation of `HEAP-DELETE` that runs in $O(\log(n))$ time for an n-element max-heap.

6. Suppose we have descending sorted array A with n elements, and always pick first element of A as pivot and sort A once in descending order and once in ascending order using quick sort. what is time complexity in each case?

7. Suppose you are given a sorted list of n elements followed by $f(n)$ randomly ordered elements. How would you sort the entire list if:

a. $f(n) = O(1)$

b. $f(n) = O(\log(n))$

c. $f(n) = O(\sqrt[2]{n})$

d. How large can $f(n)$ be for the entire list still to be sortable in $O(n)$ time?

8. You have to split an unsorted integer array, W_n, of size n into the two subarrays, U_k and V_{n-k}, of sizes k and $n-k$, respectively, such that all the values in the subarray U_k are smaller than the values in the subarray V_{n-k}. You may use the following two algorithms:

a. Sort the array W_n with `QuickSort` and then simply fetch the smallest k items from the positions $i = 0, 1, \dots, k-1$ of the sorted array to form U_k (you should ignore time of data fetching comparing to the sorting time).

b. Directly select k smallest items with `QuickSelect` from the unsorted array (that is, use `QuickSelect` k times to find items that might be placed to the positions $i = 0, 1, \dots, k-1$ of the sorted array). Let processing time of `QuickSort` and `QuickSelect` be $T_{sort}(n) = c_{sort}.nlog_2 n$ and $T_{select}(n) = c_{select}.n$, respectively. Choose the algorithm with the smallest processing time for any particular n and k. Assume that in the previous case `QuickSort` spends 10.24 $microseconds$ to sort $n = 1024$ unsorted items and `QuickSelect` spends 1.024 $microseconds$ to select a single i-smallest item from $n = 1024$ unsorted items. What algorithm, a or b, must be chosen if $n = 2^{20}$ and $k = 2^9 = 512$?

9. Consider the following recursive algorithm for sorting a subarray `A[i..n]` of unique items. It is initially invoked as `Sort(A, 1, n)` to sort the entire array. It returns a truth value that is True if the array is sorted, False otherwise; this value is only used in the algorithm for deciding whether to continue sorting.

```
Sort (A, i, n)
1 if i = n
2       for j = 2 to n
3               if A[j-1] > A[j]
4                       return False
5       return True
6 else
7       for j = i to n
8               exchange(A[i], A[j])
9               if Sort(A, i+1, n)
10                      return True
11              exchange(A[i], A[j])
12      return False
```

Note that the value of `i` recursively increases from `1` to `n` (see line `10`). The recursion stops when `i` reaches `n`; at this point, the array is checked to see if it is in order. If it isn't, False is returned and the algorithm continues. If it is, True is returned and all the recursion "unwinds" and stops, leaving the array in sorted order. The when `i` isn't equal to `n`, the algorithm tries exchanging every element in `A[i..n]` with `A[i]`, seeing if the resulting array is (recursively) sorted (and if so, returning with True), then restoring the array to its original order to try again. This just generates every possible permutation of `A[1..n]`, testing each one to see if it is sorted.

Give upper and lower bounds on the number of times line `4` is executed. Your bounds should be within a factor of n of each other. You may assume that, on average, half of the permutations must be checked before the sorted one is found.