1. (15 points) Quick sort.
   a. (5 points) Study the pseudocode of the Partition algorithm in lecture9.ppt. Using Slide #37 as a model, illustrate the operation of Partition on array A = [14 1 16 11 12 20 4 15 3 19]. Importantly, indicate where is the pivot element when the algorithm terminates.
   b. (5 points) Using Slide #38 as a model, illustrate the operation of Quick Sort on array A above.
   c. (5 points) In the pseudocode for Quicksort on Slide #31, Quicksort was recursively applied to two subarrays: one subarray from index p to q − 1, and the other from index q + 1 to r. This is correct because element A[q] is already in the correct position. Now consider the following modified pseudocode for Quicksort:

   ```
   QUICKSORT(A, p, r)
   if (p < r)
       then q <- PARTITION(A, p, r)
       QUICKSORT(A, p, q-1)
       QUICKSORT(A, q, r)
   ```

   The only difference between the pseudocode above and the original one is the last line. In the above pseudocode, the second subarray starts from q instead of q + 1. Besides having to sort one extra element, this modified algorithm will sometimes fail to terminate. Give an example array for which the modified algorithm will fail to terminate.

2. (10 points) Indicate whether the following statement is true or false. Briefly justify your answers.
   a. Quick sort runs in linear time for an already sorted array.
   b. Quick sort can run in Θ(n^2) time in the worst case.
   c. Randomized quick sort is guaranteed to run in Θ(n log n) time.
   d. The average running time on all possible inputs is Θ(n log n) for both quick sort and randomized quick sort.

3. (20 points) Analysis of Randomized Algorithm.
   During lecture, we analyzed Randomized Quick Sort and proved its expected running time using the substitution method. Here we practice the analysis with a simple randomized algorithm.

   ```
   RandAlg(A[1..n])
   if (n <= 1) return;
   Func(A[1..n]); // Func has a running time of Θ(n).
   Let r be a random number between 0 and 1;
   if (r < 0.5) // with probability 0.5
       RandAlg(A[1..n/2]); // RandAlg is applied to a subarray of size n/2
       RandAlg(A[n/2+1..n]); // RandAlg is applied to a subarray of size n/2
   else // with probability 1 - 0.5
       RandAlg(A[1..n/3]); // RandAlg is applied to a subarray of size n/3
       RandAlg(A[n/3+1..2n/3]); // RandAlg is applied to a subarray of size n/3
   ```
a. (5 points) What is the best case running time of RandAlg as a function of \( n \)?

b. (5 points) What is the worst case running time of RandAlg as a function of \( n \)?

c. (3 points) Using the analysis on Slide #57 as a template, write down a recurrence for the expected running time of RandAlg as a function of \( n \).

d. (7 points) Using substitution method to prove that the expected running time is \( \Theta(n) \).

e. (Extra credit: 5 points) If the line \( \text{if } (r < 0.5) \) is changed to \( \text{if } (r < 0.9) \), what is the expected running time now?

4. (30 points) Heap and heap sort.

a. (5 points) Is the sequence \([23, 17, 14, 6, 13, 10, 1, 5, 7, 12]\) a max-heap?

b. (5 points) What are the minimum and maximum numbers of elements in a heap of height \( h \)? Briefly justify your answer.

c. (5 points) Study the pseudocode of Heapify and BuildHeap on Slide #16 and #29 in lecture10.ppt. Using Slides #30 – 42 as a model, illustrate the operation of BuildHeap on array \( A = [14, 15, 10, 5, 6, 16, 11, 3, 12, 20] \). Figure 1 on page 3 is provided for your convenience. You only need to show the content of the tree after each call to heapify(). Make sure your final tree is indeed a heap.

d. (5 points) Starting from the heap shown in Figure 2 (page 4), show the content of the new heap after each heap operation.

e. (5 points) Why do we want the loop index \( i \) in algorithm BuildHeap to decrease from \( \lceil \text{length}(A)/2 \rceil \) to 1 rather than increase from 1 to \( \lceil \text{length}(A)/2 \rceil \)?

f. (5 points) What is the running time of heapsort on an array \( A \) of length \( n \) that is already sorted in increasing order? What about decreasing order? (Hint: first think about the cost for building heap in these two cases, then think about the cost for the actual sorting part. You can use examples \([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]\) or \([9, 8, 7, 6, 5, 4, 3, 2, 1, 0]\) to help you think.)

5. (15 points) Building a heap using insertion.

The procedure BuildHeap can also be implemented by repeatedly using HeapInsert to insert the elements into the heap. Consider the following implementation:

```
BuildHeap2(A)
    heapsize(A) = 1;
    for (i = 2 to length(A))
        HeapInsert(A, A[i]);
```

a. (5 points) Compare the procedure BuildHeap2 with the procedure BuildHeap on Slide #29 in lecture10.ppt. Do the two procedures always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. (5 points) What is the worst-case time complexity of BuildHeap2? Briefly justify your answer.

c. (5 points) How does the running time of the two procedures compare?

6. (Extra credit: 20 points) Nuts and bolts. You are given a collection of \( n \) bolts of different widths and \( n \) corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no
way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an efficient algorithm for this problem with average-case efficiency in $\theta(n \log n)$.

7. **Bonus** (5 points) How much time did you spend on this homework? Who did you discuss with and what was the discussion about? What do you think about the difficulty level of the homeworks/exams in general? Do you have any comments/suggestions about the lecture, recitation, homework, and exam?
Figure 1: Build heap
Figure 2: Heap operations