Problem Set 2

This problem set is due **in lecture** on **Wednesday, September 25.**

Reading: Akra-Bazzi handout; §5.1-5.3; Chapter 7; §8.1-8.3; Chapter 6

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

Exercise 2-1. Do Exercise 5.3-2 on page 104 of CLRS.

Exercise 2-2. Do Exercise 6.1-2 on page 129 of CLRS.

Exercise 2-3. Do Exercise 6.4-3 on page 136 of CLRS.

Exercise 2-4. Do Exercise 7.2-2 on page 153 of CLRS.

Exercise 2-5. Do Problem 7-3 on page 161 of CLRS.

Problem 2-1. In the past we have assumed that multiplying two integers is a $\Theta(1)$-time operation, but this assumption is unrealistic if the integers grow very large. To multiply two n-bit integers u and v, the traditional algorithm requires $O(n^2)$ bit-multiplications and additions. A divide-and-conquer algorithm splits the numbers into two equal parts, computing the product as

$$uv = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$

(a) Write pseudocode for a procedure that recursively computes the products ac, ad, bc, and bd, and combines them to compute the product uv. Then write a recurrence describing your algorithm's running time, and solve it.

(b) Design and analyze a different divide-and-conquer algorithm that has a better asymptotic running time. (*Hint:* Compute ac and bd, then compute $ad + bc$ using only one additional multiplication along with addition and subtraction.)

Problem 2-2. Consider an unsorted array A consisting of n elements and the following deterministic searching strategy: to find an element x, check each element of A in order and return i as soon as $A[i] = x$, otherwise terminate after examining every element of A.

(a) Give the worst-case running time of this algorithm in the case that there is **one** i such that $A[i] = x$, in the case that there is **no** i such that $A[i] = x$, and in the case that at least **half** of the elements of A are x. Your answers should be as exact as your can make them (not just asymptotic).

(b) Give the average-case running time of this algorithm for the same three cases. Assume that within each case, all permutations of the array are equally likely as inputs.

(c) Lastly, consider a randomized algorithm SCRAMBLE-SEARCH that randomly permutes the input array and then runs the deterministic search from above on the resulting permuted array. Give the expected number of elements (taken over the random choices of the algorithm) examined for each of the three cases.

Problem 2-3. Consider the problem of merging arrays as done in MERGE-SORT. In this problem, you will show that any procedure for merging two sorted arrays, each of length n, must perform at least $2n - 1$ comparisons in the worst case.

Parts **(a)** and **(b)** show a slightly weaker lower bound of $2n - o(n)$ using a decision tree argument. Parts **(c)** and **(d)** show the tighter $2n - 1$ bound.

(a) Given $2n$ unique numbers, how many possible ways are there to divide them into two sorted lists, each with n numbers?

(b) Using a decision tree, show that any algorithm that correctly merges two sorted lists uses at least $2n - o(n)$ comparisons. (See pages 47–48 in CLRS for a discussion of o-notation.)

(c) Show that if two elements are consecutive in the final sorted list and come from different sub-lists, then they must be compared.

(d) Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

Problem 2-4. Louis Reasoner was complaining that in order to sort n numbers in the range $1, \ldots, k$, COUNTING-SORT uses $\Omega(n + k)$ extra storage. "It would be much better if we could sort ***in place***, by using only $O(1)$ auxiliary storage to re-arrange the elements (plus $O(k)$ to maintain the counts)," he was overheard to say. He then offered the following code, to replace lines 9–11 in COUNTING-SORT (see page 168 of CLRS):

```
r ← length[A]
while r ≠ 0
  ▷ Consider element a = A[r] and get its sorted position j
  a ← A[r]
  j ← C[a]
  if j ≥ r
      ▷ Move a to its sorted position, and never touch it again
      C[a] ← C[a] − 1
      while j ≠ r
        ▷ Save element b = A[j] and get its sorted position k
        b ← A[j]
        k ← C[b], C[b] ← C[b] − 1
        A[j] ← a
        ▷ Now b must be moved to its sorted position
        a ← b, j ← k
      A[j] ← a
  r ← r − 1
```

(a) Prove that when the outer loop terminates, A is sorted. (*Hint:* prove that at the start of each iteration, each element $A[j]$ for $j > r$ is either in its final sorted position or needs to move left.)

(b) What is the asymptotic running time of this modified procedure? Justify your answer.

(c) Can this modified procedure be used as a subroutine for RADIX-SORT? Why or why not?

Problem 2-5. Ben Bitdiddle decides to extend the idea of a binary heap to a k-ary heap. Thus, each node in the heap now has k children instead of just two.

(a) If the heap is represented by an array A, describe how to find the parent and the (at most) k children of element $A[i]$.

(b) Ben decides to implement heapsort using his k-ary heaps. He chooses $k = 1$ and argues that for this choice of k, the only operation required in the heapsort algorithm is BUILD-HEAP. Since BUILD-HEAP takes $O(n)$ time, he can actually sort in $O(n)$ time! Find the fallacy in Ben's argument. What familiar sorting algorithm is k-ary heapsort really performing for $k = 1$?

(c) The worst-case number of comparisons $T(n, k)$ performed by HEAPSORT is $\Theta(nhk)$, where $h = \Theta(\log_k n)$ is the height of the heap. Solve for k so that $T(n, k)$ is minimized.