MATH 416, PROBLEM SET 2

Comments about homework.

- Solutions to homework should be written clearly, with justification, in complete sentences. Your solution should resemble something you'd write to teach another student in the class how to solve the problem.
- You are encouraged to work with other 416 students on the homework, but solutions must be written independently. Include a list of your collaborators at the top of your homework.
- You should submit your homework on Gradescope, indicating to Gradescope where the various pieces of your solutions are. The easiest (and recommended) way to do this is to start a new page for each problem.
- Attempting and struggling with problems is **critical** to learning mathematics. Do not search for published solutions to problems. I don't have to tell you that doing so constitutes academic dishonesty; it's also a terrible way to get better at math.

  If you get stuck, ask someone else for a hint. Better yet, go for a walk.

WARNING. These are not necessarily *model solutions*; they are meant to help you understand the problems you didn't totally solve and maybe to give you alternative solutions. Sometimes I will give less or more detail here than I would expect from you.

Problem 1. In class we described a binary-search algorithm that searches a sorted array $L[1,\dots,n]$ for a given element x in time $O(\log n)$. Show that this is optimal, in the following sense: any algorithm that access the array only via comparisons must take $\Omega(\log n)$ steps.

Solution. The proof follows our proof of the lower bound for comparison sorting. The decision tree for a comparison-based search is a binary tree with (at least) n leaves, one for each of n possible outcomes. A binary tree with $\geq n$ leaves must have height $\geq \log_2 n$. The height of the decision tree is the worst-case number of comparisons, which is therefore $\Omega(\log n)$. $\square$

Problem 2. Consider the Bubblesort algorithm, given in pseudocode below.

Algorithm 1: Bubblesort

```
   Data: $A = (A[1], \dots, A[n])$ a list of $n$ numbers
1  for $i = 1$ to $i = n$ do
2      for $j = n$ downto $i + 1$ do
3          if $A[j] < A[j-1]$ then
4              swap $A[j-1]$ and $A[j]$
5  return $A$
```

(a) Let A' denote the output of Bubblesort on input A. To prove that Bubblesort is correct, we need to prove at least that A' is correctly sorted. What else must be proved?

(b) Identify and prove the validity of a loop invariant for the **for** loop in lines 1–4.

(c) Identify and prove the validity of a loop invariant for the **for** loop in lines 2–4.

(d) What is the worst-case running time of Bubblesort? How does it compare to the running time of Insertion Sort?

Solution. (a) You must also show that it contains the same elements as A does, but this is clear because the only changes made to the array during the course of the algorithm are swaps, so (by induction) the contents never change.

(b) At the start of the i^{th} iteration of the outer **for** loop, $A[1], \dots, A[i-1]$ will be the $i-1$ smallest elements of A in the correct order. At the beginning, this says nothing: the first 0 elements are sorted.

Maintenance: Suppose that at the start of the i^{th} iteration the first $i-1$ elements are in positions $A[1], \dots, A[i-1]$, in order. In the next part we will show that after finishing the iteration of the inner loop the smallest element of $\{A[i], \dots, A[n]\}$ is $A[i]$. The smallest $i-1$ elements of the array are already stored in positions 1 through $i-1$, so $A[i]$ must therefore be the i^{th}-smallest entry in the array. Therefore $A[1], \dots, A[i]$ are the i smallest elements, in order, as desired.

At the end of the algorithm, the smallest $n-1$ elements will be sorted into positions $A[1], \dots, A[n-1]$, which leaves as the only possibility that the maximum element is $A[n]$.

(c) At the start of each iteration, the least element of $\{A[i], \dots, A[n]\}$ occurs among $A[i], \dots, A[j]$. This is certainly true at the start.

Maintenance: Suppose that $j = k$ and that the least element of $\{A[i], \dots, A[n]\}$ occurs among $A[i], \dots, A[k]$. The algorithm compares $A[k]$ and $A[k-1]$.

Case 1: If $A[k] < A[k-1]$, then $A[k-1]$ cannot be the least element of $\{A[i], \dots, A[n]\}$. After we swap $A[k]$ and $A[k-1]$, the least element of $\{A[i], \dots, A[n]\}$ now must be one of $A[i], \dots, A[k-1]$, so we have maintained the invariant.

Case 2: If $A[k] = A[k-1]$, and $A[k]$ is the least element of $\{A[i], \dots, A[n]\}$, then so is $A[k-1]$, so it occurs among $A[i], \dots, A[k-1]$ (after we do nothing). If neither $A[k]$ nor $A[k-1]$ is the least element of $\{A[i], \dots, A[n]\}$, then the least element must occur among $A[i], \dots, A[k-2]$.

Case 3: If $A[k] > A[k-1]$, then we don't perform a swap. In this case, $A[k]$ cannot be the least element of $\{A[i], \dots, A[n]\}$, so the least element must be among $A[i], \dots, A[k-1]$ already.

When the loop terminates, the smallest element of $\{A[i], \dots, A[n]\}$ is $A[i]$.

(d) The i^{th} iteration of the outer **for** loop will cause $n-i$ iterations of the inner loop. Assuming (as we may) that the inner loop has constant time-cost (one comparison and possibly one swap), we conclude that the number of times the algorithm visits line 4, say, should be

$$\sum_{i=1}^{n}(n-i) = \sum_{k=0}^{n-1} = n(n-1)/2,$$

which is $O(n^2)$. This analysis does not depend on the input, so this is also the *best*-case running time of Bubblesort.

It is worth mentioning that Insertion Sort, which also has worst-case running time $O(n^2)$, has linear best-case running time.

Problem 3. Prove (carefully!) by induction that a binary tree of height $\leq h$ has at most 2^h leaves.

Solution. Let's write $l(T)$ for the number of leaves of a tree T and $h(T)$ for its height.

We prove by induction on the number n of vertices of T that every binary tree T with n vertices, $l(T) \leq 2^{h(T)}$.

This is certainly true for a tree with only $n = 1$ vertex.

Suppose inductively that every binary tree T with $< n$ vertices, $l(T) \leq 2^{h(T)}$.[1] Suppose that T has n vertices. Let T_L be the left subtree of the root, and let T_R be the right subtree (each without the root of T). Every leaf of T is a leaf either of T_L or T_R, so $l(T) = l(T_L) + l(T_R)$. And a path of maximal length from the root in T goes through either T_L or T_R, so $h(T) = \max(h(T_L), h(T_R)) + 1$. (It is possible that T_L or T_R is empty, so we should define $h(\emptyset) = -1$.) Each of T_L and T_R has fewer than n vertices, so our induction hypothesis implies that $l(T_L) \leq 2^{h(T_L)}$ and $l(T_R) \leq 2^{h(T_R)}$. Now we have

$$\begin{aligned} l(T) &= l(T_L) + l(T_R) \\ &\leq 2^{h(T_L)} + 2^{h(T_R)} \\ &\leq 2^{\max(h(T_L),h(T_R))} + 2^{\max(h(T_L),h(T_R))} \\ &= 2^{\max(h(T_L),h(T_R))+1} \\ &= 2^{h(T)}, \end{aligned}$$

which is what we set out to prove. □

[1] We are using so-called *strong induction*.

Problem 4. Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \log_2(n)$.

Solution. When $n = 2$ we have $T(2) = 2\log_2(2) = 2$.

The inductive hypothesis is that when $n = 2^k$ then $T(2^k) = 2^k \log_2(2^k) = nk$. Now for $n = 2^{k+1}$, by definition of the recurrence

$$T(2^{k+1}) = 2T(2^{k+1}/2) + 2^{k+1}.$$

By induction, we have

$$\begin{aligned} 2T(2^{k+1}/2) + 2^{k+1} &= 2T(2^k) + 2^{k+1} \\ &= 2(2^k \log_2(2^k)) + 2^{k+1} \\ &= 2^{k+1}(\log_2(2^k) + 1) \\ &= 2^{k+1}(\log_2(2^{k+1})) \end{aligned}$$

which completes the proof by induction. □

Problem 5. We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \dots n]$, we recursively sort $A[1 \dots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \dots n-1]$. Write an recurrence for the worst-case running time of this recursive version of insertion sort.

Solution. To insertion sort a list of length n, we first insertion sort a list of length $n-1$. This takes $T(n-1)$. We then must place $A[n]$ into the sorted list $A[1 \dots n-1]$. This can take up to $n-1$ swaps. So we have that the worst case run time satisfies the recurrence:

$$T(n) = T(n-1) + (n-1).$$

Problem 6. Let $A[1 \dots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A.

(a) List the five inversions of the array $\langle 2, 3, 8, 6, 1\rangle$.
(b) Without proving it. What array with elements from the set $\{1, 2, \dots, n\}$ has the most inversions? (and how many are there?).
(c) What is the relationship between the running time of insertion sort and the number of inversions in the input array? Give a brief justification.
(d) Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \log(n))$ worst-case time. (Hint: Modify merge sort.) You do not need to prove that your algorithm runs in $\Theta(n \log(n))$, but you should give a brief explanation.

Solution. (a) The inversions of $\langle 2, 3, 8, 6, 1\rangle$ are $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, and $(4, 5)$.
(b) The array with the most inversions is $\langle n, n-1, \dots, 2, 1\rangle$. There are $\binom{n}{2}$ inversions.
(c) In insertion sort we sort the first i elements in the list and then decide where the $i + 1$th element goes. The number of swaps we perform with the $i + 1$th element is the same as the number of array entries before $A[i + 1]$ that are greater than $A[i + 1]$ in other words they correspond exactly to the j such that $j < i + 1$ BUT $A[j] > A[i + 1]$. Therefore, the number of inversions in an array is the same as the number of swaps performed by insertion sort.
(d) The key is to modify how we merge two ordered lists. Suppose A_1 and A_2 are ordered. Giving an array $A = A_1 + A_2$ (where we append A_2 to the end of A_1), recall that Merge(A_1, A_2) is given as follows:

Algorithm 2: Merge

```
INPUT: $A_1$ and $A_2$ ordered lists of length $n$
OUTPUT: $A_1 + A_2$ merged as an ordered list. LOCVARS: $P_1$, $P_2$
  (pointers), and $A_3$ (array) INITIALVALS: $P_1 = 1$, $P_2=1$, and $A_3 = []$
while $P_1 \leq n$ and $P_2 \leq n$ do
    if $A_1[P_1] < A_2[P_2]$ then
        Append $A_1[P_1]$ to $A_3$
        Increase $P_1$ by 1
        if $P_1 = n + 1$ then
            Append the list $A_2[P_2 \dots n]$ to $A_3$.
    else
        Append $A_2[P_2]$ to $A_3$
        Increase $P_2$ by 1
        if $P_2 = n + 1$ then
            Append the list $A_1[P_1 \dots n]$ to $A_3$.
```

The observation is that we should also keep track of inversions as we go, with a counter. The point is that if we take A_1 and append A_2 at the end. The number of total inversions if we sort $A_1 + A_2$ is given by counting the inversions in A_1, the inversions in A_2, and then once A_1 and A_2 are sorted we must also add an inversion for every pair of elements $(a_1, a_2) \in A_1 \times A_2$ with $a_1 > a_2$. This can be tracked during the Merge process.

Algorithm 3: MergeAndTrackInversions

```
1  INPUT: $A_1$ and $A_2$ (ordered lists of lengths $n_1$ and $n_2$ respectively)
     and $i_1$, $i_2$ (total inversions so far for $A_1$ and $A_2$ respectively).
2  OUTPUT: $A_1 + A_2$ merged as an ordered list and $i$ the total inversions.
3  LOCVARS: $P_1$, $P_2$ (pointers), $A_3$ (array), and $i$ (inversion counter)
4  INITIALVALS: $P_1 = 1$, $P_2$=1, $A_3 = []$, $i = i_1 + i_2$.
5  while $P_1 \leq n_1$ and $P_2 \leq n_2$ do
6      if $A_1[P_1] < A_2[P_2]$ then
7          Append $A_1[P_1]$ to $A_3$
8          Increase $P_1$ by 1
9          if $P_1 = n + 1$ then
10             Append the list $A_2[P_2 \ldots n]$ to $A_3$.
11     else
12         Append $A_2[P_2]$ to $A_3$
13         Increase $P_2$ by 1
14         Increase $i$ by $n_1 - P_1 + 1$. (THIS IS THE MAIN CHANGE)
15         if $P_2 = n + 1$ then
16             Append the list $A_1[P_1 \ldots n]$ to $A_3$.
```

Using this algorithm in place of Merge in MergeSort counts the total inversions. The run time is the same as MergeAndTrackInversions also runs in $O(n)$.

Problem 7. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a flip – insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over. Describe an algorithm to sort an arbitrary stack of pancakes using $O(n)$ flips.

Solution. Here is one possible algorithm. For pancakes stacks of height 1 there is nothing to do. Now for a pancake stack of height n, first perform flips on the top $n-1$ pancakes so that they are sorted. This takes $T(n-1)$ flips. Now we need to put the last pancake (name it P) in the correct spot.

(1) Flip all of the pancakes (so that the P is now on top).
(2) Make a flip between the two pancakes where the P belongs.
(3) Make another flip directly above P
(4) Flip all pancakes.

Thus, to order all the pancakes requires $T(n-1)+4$ flips. So we see that $T(n)$ is $O(n)$.