601.433/633 Introduction to Algorithms **Fall 2025**
Homework #2 **Due:** September 29, 2025, 11:59pm

Remember: you may work in groups of up to three people, but must write up your solution entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. You may use the internet to look up formulas, definitions, etc., but may not simply look up the answers online.

Please include proofs with all of your answers, unless stated otherwise.

1 Jug Matching (33 points)

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do all the blue ones. Moreover, for every red jug there is a blue jug that holds exactly the same amount of water (and vice versa).

Your job is to find a matching between red jugs and blue jugs that hold the same amount of water. To do this, you are only allowed to use the following operation: pick a red jug and a blue jug, fill the red jug, and pour it into the blue jug. This will tell you whether the volume of the red jug is less than, equal to, or greater than the volume of the blue jug. In other words, you can compare any red jug and any blue jug. But you *cannot* compare two red jugs or two blue jugs.

Give a randomized algorithm that uses $O(n \log n)$ comparisons in expectation. Prove that your algorithm is correct and that it uses $O(n \log n)$ comparisons in expectation.

1. If n is 0, return

2. If $n = 1$, return the matching which matches the unique pitcher in R to the unique pitcher in B.

3. Pick a pitcher p_R from R uniformly at random to be the pivot. Compare each pitcher in B to p_R. Let L_B be the pitchers of B that are less than p_R, let p_B be the pitchers of B with volume equal to p_R, and let G_B be the pitchers of B that are greater than p_R.

4. Now compare each pitcher in R to p_B. Let L_R be the pitchers of R that are less than p_B, and let G_R be the pitchers of R that are greater than p_B.

5. Match p_R and p_B. Then recurse once on L_R and L_B, and recurse a second time on G_R and G_B.

6. (9 points) Prove that your algorithm is correct (i.e., that it always returns the correct matching).

Solution. We will use an inductive argument to prove that if our algorithm is called on two sets of pitchers of size n for which a matching does exist, then our algorithm will find it. This clearly implies correctness, since the initial call on (R, B) is to two sets of pitchers for which a matching does exist.

For the base cases, first suppose that $n = 0$. Then there ar eno pitchers so we correctly return the empty matching. Now suppose that $n = 1$. Then the algorithm simply returns the trivial matching, which is correct as long as there exists a matching between the two sets.

Now suppose that the inductive hypothesis is true for all $n' < n$, and consider the case of being called on R, B which have size n. The algorithm matches p_R and p_B correctly, since it directly compares them and finds out that they're equal. It is then recursively called on (L_R, L_B) and on (G_R, G_B). Clearly nothing in L_R can be matched to anything in G_B (since everything in L_R has size less than p_R, and everything in G_B has weight larger than p_B and hence larger than p_R). Similarly, nothing in G_R and be matched to anything in L_B. Thus if there is a matching between R and B, there must be a matching between L_R and L_B and a matching between G_R and G_B.

Thus by the inductive hypothesis we know that if there is a matching between R and B, our recursive calls will find a matching between L_R and L_B and between G_R and G_B (since they all have size less than n). Since we correctly match p_R and p_B, this implies that we have correctly matched R and B if such a matching exists.

7. (9 points) Prove that your algorithm uses $O(n \log n)$ comparisons in expectation.

Solution. To analyze the expected number of comparisons, note that before the recursive calls there are $2n - 1$ comparisons: n comparisons of blue pitchers to p_R, and then $n - 1$ comparison of red pitchers (other than p_R) to p_B. Let R_i be the red pitcher that has i-th smallest size. Then if R_i is p_A, the two recursive calls are on sets of is $i - 1$ and sets of size $n - i$. So we have the following recurrence relation for the number of comparisons:

$$\begin{aligned} T(n) &= n + (n-1) + \sum_{i=1}^{n} \Pr[R_i \text{ is pivot}] \cdot \mathbb{E}[\#\text{comparisons} \mid R_i \text{ is pivot}] \\ &= 2n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \\ &= 2n - 1 + \frac{2}{n} \sum_{i=1}^{n} T(i) \end{aligned}$$

We claim that $T(n) \leq 4n \ln n + 1$. We will prove this by induction.

The base case is $n = 1$, where $T(n) = 1 \leq 4n \ln n + 1$. For general n, assume $T(i) \leq 4i \ln i + 1$

for all $i < n$, we have

$$\begin{aligned}
T(n) =& 2n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} T(i) \\
\leq& 2n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}(4i \ln i + 1) \\
=& 2n - 1 + \frac{2(n-1)}{n} + \frac{2}{n}\sum_{i=1}^{n-1} 4i \ln i \\
\leq& 2n + 1 - \frac{2}{n} + \frac{8}{n}\int_1^n (x \ln x) dx \\
=& 2n + 1 - \frac{2}{n} + \frac{8}{n}\left(\frac{1}{2}x^2 \ln x - \frac{1}{4}x^2\right)_{x=1}^{n} \\
=& 2n + 1 - \frac{2}{n} + \frac{8}{n}\left(\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 + \frac{1}{4}\right) \\
\leq& 2n + 1 - \frac{2}{n} + 4n \ln n - 2n + \frac{2}{n} \\
=& 4n \ln n + 1
\end{aligned}$$

Therefore the induction claim holds, which shows $T(n) = O(n \log n)$.

2 Costly Median (34 points)

Suppose that you are given n distinct numbers $x_1, x_2, \ldots, x_n \in \mathbb{R}^+$, each of which also has a *cost* $c_i \in \mathbb{R}^+$ so that $\sum_{i=1}^n c_i = 1$. The *costly median* is defined to be the number x_k such that

$$\sum_{i:x_i<x_k} c_i < \frac{1}{2} \qquad \text{and} \qquad \sum_{i:x_i>x_k} c_i \leq \frac{1}{2}.$$

Give a deterministic algorithm which finds the costly median and has $O(n)$ worst-cast running time (and prove correctness and running time).

Solution

Let us assume that the n elements are stored in an array A. We will use the deterministic algorithm discussed in class (Section 4.4) that returns the median of an array in $O(n)$ time. We will call this algorithm applied to array A **MEDIAN(A)**, and will use it for our costly median algorithm. The following algorithm computes the median (in $O(n)$ time) and recurses on the half of the input that contains the costly median:

COSTLY-MEDIAN(A, L):

1. $n =$ **LENGTH**(A).
2. $m =$ **MEDIAN**(A).
3. $B = \emptyset, G = \emptyset$. $//B = \{i : A[i] < m\}, G = \{i : A[i] \geq m\}$

4. $C(B) = 0$. //total cost of B

5. **if LENGTH**(A)= 1: **return** $A[1]$

6. **for** $i = 1$ to n:

   if $(A[i] < m)$: Set $C(B) = C(B) + c_i$, and append $A[i]$ to array B
   else: Append $A[i]$ to array G

7. **if** $(L + C(B) > 1/2)$:

   COSTLY-MEDIAN(B, L) //costly median is in B

8. **else**:

   COSTLY-MEDIAN$(G, L + C(B))$

Initially the algorithm calls **COSTLY-MEDIAN(A, 0)**, where A is the original array. In all recursive calls L keeps track of the total cost of elements of the *initial* input array that are less than all the elements of A of the recursive call. Array B contains all elements less than median m, and G contains all the elements greater than or equal to the median m, and $C(B)$ is the total cost of elements in B.

Correctness. We first show that the following conditions hold for every recursive call:

- The costly median y of the initial A is always is always present in array for which the recursive call is made.
- L is the total cost of elements x_i less than all the elements in A.

These conditions clearly hold for the initial call. We can show by induction that they stay true for all recursive calls. Assume that initially these conditions holds for A_i which is the array on which the i-th recursive call is made. By induction hypothesis the costly median y is in A_i, and so we know that y is either in B or in G.

First let us consider the case $L + C(B) > 1/2$. The total cost of all elements less than any element in G is greater than $1/2$ (since $L + C(B) > 1/2$), and thus by definition the costly median cannot be in G, and so it must be in B. Also, since we have not discarded any element less than all elements in B, the condition for L still holds in this case.

If $L + C(B) \leq 1/2$, then by definition y must be in G. All elements of G are greater than all elements in B. Thus the total cost of elements x_i in the original array less than all the elements in G is $L + C(B)$, and the desired condition is also satisfied for this case. This algorithm terminates because size of A always decreases, and since the costly median is always in the array in which the recursive call is made, the algorithm returns y when it terminates.

Running time. In every call to the algorithm, only $O(n)$ time is taken directly (not in recursive calls): computing the length takes $O(n)$ time, computing the median takes $O(n)$ time (since we are using the algorithm from class), and creating B and G (and keeping track of C(B)) takes only $O(n)$ time. When we recurse, we have recursed on either B or G, and since we split around the actual median m, both B and G have size at most $\lceil n/2 \rceil$. Hence the total running time is at most $T(n) \leq T(\lceil n/2 \rceil) + O(n)$, which we can solve to get $T(n) = O(n)$.

3 More Lower Bounds (33 points)

Consider the following two-dimensional sorting problem: we are given an arbitrary array of n^2 numbers (unsorted), and have to output an $n \times n$ matrix of the inputs in which all rows and columns are sorted.

As an example, suppose $n = 3$ so $n^2 = 9$. Suppose the 9 numbers are just the integers $\{1, 2, \ldots, 9\}$. Then possible outputs include (but are not limited to)

```
1 4 7     1 3 5     1 2 6
2 5 8     2 4 6     3 4 8
3 6 9     7 8 9     5 7 9
```

It is obvious that we can solve this in $O(n^2 \log n)$ time by sorting the numbers and then using the first n as the first row, the next n as the second row, etc. For this question, you should prove that this is tight. Formally, you should prove that in the comparison model, any algorithm that solves this problem requires $\Omega(n^2 \log n)$ time. For simplicity, you can (as always) assume that n is a power of 2.

Hints: instead of reasoning directly about the decision tree, show that if there exists an algorithm making a number of comparisons that is not $\Omega(n^2 \log n)$ then we could break the sorting lower bound (we could sort an array of size n using fewer than $\log(n!)$ comparisons). Useful facts to keep in mind are that $n! > (n/e)^n$ and that we can merge two sorted arrays of length n using $2n - 1$ comparisons. You might need to be careful with constants.

Solution. We first prove the following Claim:

Claim 1 *Given an $2^k \times 2^k$ matrix which all rows and columns are sorted, we can sort all the items in the matrix within $2^{2k} \cdot k$ comparisons.*

Proof: The algorithm is as follows: First partition the rows to 2^{k-1} groups, where each group has 2 rows. Merge the two rows in each group into a sorted array. Then partition the arrays to 2^{k-2} groups, merge the two rows in each group again. Keep doing this until only one array is left. This array must be sorted and it contains all the items in the matrix.

In the i-th step, there are 2^{k-i} groups, and each array in the group has 2^{k+i-1} items. Thus the number of comparisons used in each step is $2^{k-i} \times (2 \cdot 2^{k+i-1} - 1) < 2^{2k}$. There are k steps in total, so the total number of comparisons is at most $2^{2k} \cdot k$. ■

Let $n = 2^k$ and assume that there is an algorithm to sort an $n \times n$ matrix within $o(n^2 \log n) = o(2^{2k} \cdot k)$ comparisons. Then we can sort all items in an arbitrary $n \times n$ matrix within $2^{2k} \cdot k + o(2^{2k} \cdot k)$ comparisons using Claim 1. However, the input for sorting n^2 items has $(n^2)!$ possibilities (all the permutations), so the number of comparisons to sort n^2 items is at least

$$\log((n^2)!) > \log\left(\left(\frac{n^2}{e}\right)^{n^2}\right) = 2n^2 \log n - n^2 \log e = 2^{2k+1} \cdot k - 2^{2k} \cdot \log e = 2^{2k+1} \cdot k - O(2^{2k}),$$

and this is larger than $2^{2k} \cdot k + o(2^{2k} \cdot k)$ when k is large enough, contradicted.

Therefore, there is no algorithm that can sort an $n \times n$ matrix within $o(n^2 \log n)$ comparisons.