15-451/651 Algorithms, Spring 2018

Homework #1 **Due: January 30, 2018**

---

0. **(Exercises: Recurrences and Probability.)**

Solve each recurrence below in Θ notation. As always, prove your answer. For all of these problems $T(1) = 1$.

(a) Solve $T(n) = 3\,T(\lfloor n/2 \rfloor) + n$.

Solution: $T(n) = \Theta(n^{\log_2 3})$. Master theorem.

(b) Now solve $T(n) = 3\,T(\lfloor n/2 \rfloor) + n \lg n$.

Solution: $T(n) = \Theta(n^{\log_2 3})$. For this we can also use the Master theorem. We'll use it to solve two recurrences:

$$T'(n) = 3\,T'(n/2) + n$$

and

$$T''(n) = 3\,T''(n/2) + n^{3/2}$$

By the master theorem, the solution to both of these is $\Theta(n^{\log_2 3})$. Also we have:

$$T'(n) < T(n) < T''(n)$$

From which the result follows

(c) Finally, solve $T(n) = n^{2/3}\,T(\lfloor n^{1/3} \rfloor) + n$.
(E.g., we might get this from a divide-and-conquer procedure that uses linear time to break the problem into $n^{2/3}$ pieces of size $n^{1/3}$ each. Hint: write out the recursion tree.)

Solution: Each level of the recursion tree has a total cost of n, so we only need to figure out the number of levels. If we write n as 2^k we see that as we go down the tree the problem sizes are $2^k, 2^{k/3}, 2^{k/9}, 2^{k/27}, \ldots, 2^{k/(3^{\lceil \log_3 k \rceil})}$. So, the depth of the tree is $\Theta(\log k) = \Theta(\log \log n)$. So, the recurrence solves to $\Theta(n \log \log n)$.

(25 pts) 1. **(A Good Sort)** Consider the following problem.

INPUT: an $n \times n$ matrix M containing n^2 distinct numbers, where the n numbers in each row are in sorted order. (Such a matrix is called a *row-sorted* matrix.)

OUTPUT: a sorted list L of the n^2 numbers in the matrix M.

EXAMPLE: $n = 3$, so $n^2 = 9$. Say the 9 numbers in M are the digits $1, \ldots, 9$. Possible inputs (row-sorted matrices) include:

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

The output for all these would be the sorted list $L = 1, 2, 3, 4, 5, 6, 7, 8, 9$.

It is clear that we can solve this problem using at most $2n^2 \lg n$ comparisons by forgetting about the row-sorted structure of the input matrix M, and sorting the n^2 numbers using, say, MergeSort. (Remember that $\lg n^2 = 2 \lg n$, where $\lg x$ == $\log_2 x$.)

In this problem you will show how to do better, and then give a lower bound. For simplicity, you can assume n is a power of 2.

(a) Show how to solve this problem using at most $n^2 \lg n$ comparisons.

(b) Show that any comparison-based algorithm to solve this problem must use at least $n^2 \lg n - O(n^2)$ comparisons.

Some hints for part (b): Show that if you could solve this problem using fewer than that many comparisons, then you could use this to violate the $\lg(m!)$ lower bound for comparisons needed to sort m elements (which we prove in Lecture #2). You may want to use the fact that $m! > (m/e)^m$. Also, recall that you can merge two sorted arrays of size k using at most $2k - 1$ comparisons.

Solution: For part (a), we will use the fact that merging two sorted lists of size K takes (2K-1) comparisons. Use merge-sort to combine the matrix rows in pairs (first pair the sorted rows and merge them, the merge the resulting lists in pairs, etc.), which takes $(n/2) \times (2n - 1) + (n/4) \times (4n - 1) + ... + 1 \times (2 * n^2/2 - 1) \leq n^2 \lg n - (n - 1)$ comparisons.

For part (b), recall the lower bound for sorting n^2 numbers is

$$\ln(n^2!) \geq n^2 \lg n^2 - 1.4n^2 \geq 2n^2 \lg n - 1.4n^2, \tag{1}$$

using the "information theoretic bound" in Lecture #2.

So suppose there was an algorithm $\mathcal{A}$ that could get from any row-sorted matrix M to a sorted list L using at most $An^2 \lg n - Bn^2$ comparisons. Using $\mathcal{A}$ we will soon show how to sort an *arbitrary* set of n^2 elements using at most

$$(A + 1)n^2 \lg n - Bn^2 \tag{2}$$

comparisons. This must be at least the lower bound (1), which means that $\mathcal{A}$ requires at least $n^2 \lg n - 1.4n^2$ comparisons.

Indeed, start with any set of n^2 elements. Break this into an n groups of n elements each, in an arbitrary way. This requires *zero* comparisons, of course. Now sort each of these n groups, and place them in the n rows of a matrix M. Using MergeSort, for instance, this construction of M takes at most $n \cdot (n \lg n) = n^2 \lg n$ comparisons. Now using this supposed algorithm $\mathcal{A}$, we can get from M to a sorted list in a further $An^2 \lg n - Bn^2$ comparisons. That's a total of $(A + 1)n^2 \lg n - Bn^2$ as claimed in (2).

Remark: Some of you (erroneously) claimed that you could show a lower bound of $n^2 \lg n$. But note that the solution to part (a) sketched above takes strictly fewer than that number of comparisons, so this lower bound must be clearly false. Whenever you prove a lower bound, one useful check is to make sure you cannot do better than the lower bound—if you can, something is clearly broken.

(25 pts) 2. **(Cut and Merge!)**

(a) Give an algorithm that takes as input an unsorted array A with n distinct elements, and k distinct integers $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, outputs the k elements in A having these ranks using $O(n \log k)$ comparisons.[1]

Solution: We know that we can find an element of a given rank in an unsorted list in linear time. Here is the algorithm:

i. Find the element j having rank $i_{k/2}$ in list A.
ii. Split list A into elements larger than j (L) and smaller than j (S).
iii. Recurse with the first $\frac{k}{2}$ indices and the list of smaller elements.
iv. Subtract $i_{k/2}$ from all of the indices larger than $i_{k/2}$ and recurse with those larger indices and the list of larger elements.

Here is the recurrence for the algorithm, where k is the number of ranks to get and n is the size of the list.

$$T(k, n) = T(k/2, |S|) + T(k/2, n - |S|) + O(n)$$
$$T(1, n) = O(n)$$

This recurrence is also balanced, so the total work is the work per level times the number of levels. (Or check that $T(k, n) = O(n \log k)$ using an inductive proof.) This gives us a total work of $O(n \log k)$

(b) In the MERGE problem we have to combine or merge two sorted lists into a single sorted list. Suppose we have two sorted lists of length m and n respectively (without loss of generality, we assume $m \leq n$). Show that any deterministic comparison-based algorithm for MERGE must use $\Omega(m \log(\frac{n+m}{m}))$ comparisons in the worst case.

Solution: We will use the information-theoretic approach from class. Let us denote the output of the MERGE algorithm by a string of length $m + n$ with m a's and n b's, denoting where the elements of the first and second list (respectively) appear in sorted order. So there are $\binom{m+n}{m}$ possible outputs. For each output there is an input for which that output is the only correct output. (E.g., for the output `abbaaab` one such input is `[1 4 5 6]`, `[2 3 7]`.) So using the theorem from Lecture 2, any any

[1] The rank of an element a with respect to an unsorted array A is the number of elements in A that no greater than a. If A has distinct elements, the minimum element has rank 1, and the median has rank $|A|/2$.

deterministic algorithm must take

$$\lg \binom{m+n}{m} \geq \lg \left(\frac{m+n}{m}\right)^m = m \lg \frac{m+n}{m}.$$

(The middle inequality uses $\binom{a}{b} \geq (a/b)^b$ from Quiz 1.)

(25 pts) 3. **(Preprocessing for Query Day.)** You are given as input a set S of n distinct elements. Today you can *preprocess* it using at most $P(n)$ comparisons and store the results. Tomorrow you will be given some element q and you have to answer whether $q \in S$? You can use S and this stored information to compute this answer in $Q(n)$ time. (We are in a comparison-based model for this problem, so just count the number of comparisons.)

(a) *(Do not submit!)* Show that if $P(n) = n \lg n$, then you can achieve $Q(n) = \lg n$, and if $P(n) = 0$, then you can get $Q(n) = n$, both using deterministic algorithms.

(b) Show that if $P(n) = 0$, then any deterministic algorithm must have $Q(n) \geq n$.

Solution: We will use an adversary argument: meaning we design an adversary to answer all the comparisons to make the deterministic algorithm run for a long time.

Say that the elements of S are given by s_i, and assume that we make $Q(n) < n$ many comparisons. Since there are n elements in S, there must be at least one s_i that the algorithm doesn't compare to q.

Consider the following adversarial protocol:

i. When asked to compare q to any s_i, always answer with $q \neq s_i$.

ii. A. If the algorithm halts and outputs that $q \in S$, then claim that all of the remaining s_is are not equal to q.

B. If the algorithm says $q \notin S$, claim that one s_i that was not compared to q is equal to q.

Hence no matter what, the deterministic algorithm that makes less than n comparisons is wrong, because the adversary can always make q in or not in S to contradict the algorithm.

(c) Give a randomized algorithm so that $P(n) = \frac{1}{2} n \lg n + O(n)$, such that the expected query time $Q(n)$ for any fixed query q is at most $\sqrt{n} + \frac{1}{2} \log_2 n + O(1)$.[2]

Solution: Here is a randomized algorithm that solves this problem:

Preprocessing P(n):

i. Pick $2\sqrt{n}$ many random elements from the list.

[2]If you want the bound on the preprocessing can be in expectation, but it does not need to be: the sample solution gives a worst-case bound on the preprocessing time.

ii. Sort the $2\sqrt{n}$ pivots and save them somewhere.

iii. For each remaining element i in the list, run binary search to find the smallest pivot larger than i (if one exists). This creates $2\sqrt{n}+1$ buckets of elements.

Runtime: Step A takes no comparisons, step B takes $2\sqrt{n}\lg(2\sqrt{n})$, and C takes $(n-2\sqrt{n})\lg(2\sqrt{n})$. In total $P(n)$ takes $n\lg(2\sqrt{n}) = \frac{1}{2}n\lg n + n$ comparisons.

Query Q(n): on receiving element q to query,

i. Run binary search on the $2\sqrt{n}$ pivots from preprocessing.

ii. Do a linear search on the elements in the same bucket as the query.

Runtime: Step A takes $\lg(2\sqrt{n}) = \frac{1}{2}\lg n + 1$ comparisons. The expected number of comparisons step B takes is equal to the expected size of the bucket that the query lands in. Below we analyze the expected bucket size.

Dart Lemma (From Piazza): Imagine a circle of unit circumference. You throw k darts at the boundary of the circle, each choosing a position independently and uniformly at random on the circle boundary. For any fixed point p on the boundary, how far (in expectation) along the circle is the closest dart in the clockwise direction, say? The answer is $1/(k+1)$

Proof: Let X be the distance (along the circle) from p to the closest dart position clockwise. For $x \in [0,1]$, note that $X \geq x$ exactly when all the k darts fall in the remaining $1-x$ portion of the circle. So

$$\mathbb{E}[X] = \int_{x=0}^{1} Pr[X \geq x]dx = \int_{x=0}^{1}(1-x)^k dx = -\int_{y=1}^{0} y^k dy = \int_{y=0}^{1} y^k dy = \frac{1}{k+1}.$$

We plan to use the dart lemma, so we want to map the elements in the list to a unit circle, and the pivots we pick to darts being thrown. This mapping causes the first and last buckets of the list to be merged into one (since we loop around), which will give us a larger single bucket. But it will still give a valid upper bound.

i. Assign each element a range of size $\frac{1}{n}$ on this circle, and imagine the range for the query position (initially unknown to the algorithm) to be at 12 o'clock.

ii. Now throw $2\sqrt{n}$ darts onto the circle at random, and the ranges that they fall into will represent the pivots we choose. The size of the bucket containing the query is now the sum of distances from the 12 o'clock position to the closest darts on the left and on the right.

By the dart lemma, the expected distance to the closest dart on the right is $\leq \frac{1}{2\sqrt{n}+1}$, and similarly for the distance on the left. So, the expected length (along the circle) between them is $\frac{2}{2\sqrt{n}+1} \leq \frac{1}{\sqrt{n}}$, and the number of elements represented by the bucket is less than $n \cdot \frac{1}{\sqrt{n}} + 2 = \sqrt{n} + 2$. (The additive two is because going from the continuous case to the discrete case incurs a little error. If you want to save this, you could reprove the dart lemma for the discrete case.)

Thus, step B takes $\sqrt{n} + O(1)$ time in expectation, and $Q(n) \leq \frac{1}{2}\lg n + \sqrt{n} + O(1)$.

Remark: Some of you tried to argue that by symmetry, when you throw in $\sqrt{n}$ darts, the expected size of each bucket is $\sqrt{n}$ (which is true). And "hence the expected size of the bucket containing the query point is $\sqrt{n}$." This latter statement is not correct. It is very close to $2\sqrt{n}$, as the above proof shows (and as you can see by simulation if you like).