CS 161 Winter 2020 Section 3

January 23-24, 2020

1 Light Bulbs and Sockets

You are given a collection of n differently sized light bulbs that have to be fit into n sockets in a dark room. You are guaranteed that there is exactly one appropriately-sized socket for each light bulb and vice versa; however, there is no way to compare two bulbs together or two sockets together as you are in the dark and can barely see! (You are, however, able to see where the sockets and light bulbs are.) You can try and fit a light bulb into a chosen socket, from which you can determine whether the light bulb's base is too large, too small, or is an exact fit for the socket. If the bulb fits exactly, it will flash once, in which case you have a correct match. (Note that the flashing light does not allow you to visually compare bulbs/sockets to other bulbs/sockets.)

Suggest a (possibly randomized) algorithm to match each light bulb to its matching socket. Your algorithm should run strictly faster than quadratic time in expectation. Give an upper bound on the worst-case runtime, then justify your algorithm's correctness and expected runtime.

Initial Ideas and Hints Notice that the brute force matching solution (trying all possible pairs) takes $O(n^2)$ time, so we might shoot for a runtime of O(nlogn), a time which may remind you of many divide and conquer algorithms. Lets try using divide and conquer to recursively match lightbulbs and sockets. How can we break lightbulbs/sockets into smaller groups? One of the constraints to pay attention to is that you can't compare bulbs with each other or sockets with each other (without this constraint, we could just sort the bulbs and sort the sockets in O(nlogn) time total and match them all in order). But is there a way to 'indirectly' compare two bulbs with each other (and 'indirectly' compare two sockets with each other? Notice that by comparing every socket to a single lightbulb, ℓ, you can partition the sockets into those that are too large for ℓ and those that are too small for ℓ, and this effectively compares all sockets to each other, in an indirect way. Can we doing something similar with the light bulbs?

English Description We recursively partition the lightbulbs and sockets into associated, smaller groups that are easier to match. To do so, we use a random lightbulb, ℓ^* to partition the sockets into a group that are too large for ℓ^* and a group that are too small for ℓ^*, and in the process find a match for ℓ^*, which we call s^*. Now we use s^* to partition the remaining lightbulbs into a group that is too large for s* and a group that is too small for s*. We recursively match (i) the associated "too large" light bulbs and "too large" sockets, and (ii) the associated "too small" light bulbs and "too small" sockets.

Rough Pseudocode

MATCH(L, S):

- Base Case: If $|L| == 0$, stop matching and return
- Select a lightbulb $\ell^* \in L$ uniformly at random

- For every socket $s \in S$: test whether ℓ^* is too small, too big, or just right (call the matching socket s^* and match ℓ^* with s^*)
- For every other lightbulb ℓ: test whether ℓ is too big or too small to fit into s^*
- S_{big}, $S_{\text{small}} \leftarrow$ sockets too big and too small for ℓ^*, respectively
- L_{big}, $L_{\text{small}} \leftarrow$ lightbulbs too big and too small for s^*, respectively
- MATCH(L_{big}, S_{big})
- MATCH(L_{small}, S_{small})

Correctness: We can prove correctness using strong induction.

- **Inductive Hypothesis.** MATCH correctly matches n lightbulbs and n sockets if each lightbulb has a match in the socket set and each socket has a match in the lightbulb set.
- **Base Case.** Let $n = 0$. With no lightbulbs or sockets everything is trivially matched up, and our algorithm does nothing, as required!
- **Inductive Step.** Let $k > 0$ be an integer. We assume that our inductive hypothesis holds for $n < k$, i.e. we assume that MATCH correctly matches n lightbulbs and n sockets for $n < k$. We want to show that MATCH correctly matches k lightbulbs and sockets. For the lightbulb we pick at random, ℓ^*, we go through all the sockets in S, so we are guaranteed to find its unique matching socket, s^*. Note that if a bulb is too big to fit in s^*, then it must fit in a socket that was too big for ℓ^*; likewise, if a bulb is too small to fit in s^*, then it must fit in a socket that was too small for ℓ^*. Thus, we can partition the bulbs and sockets simultaneously, such that we only have to compare "small" bulbs to "small" sockets and "big" bulbs to "big" sockets. Since $|L_{big}|$ and $|L_{small}| < k$, then by our inductive hypothesis, we know our recursive calls will correctly match the remaining bulbs to their corresponding sockets. Ultimately, MATCH will match all k lightbulbs and sockets correctly.
- **Conclusion.** By strong induction, we know that MATCH correctly matches n lightbulbs and n sockets for all natural numbers n.

Runtime: Note that at each level, we perform a linear amount of work: we go through each socket and each bulb and then partition the bulbs and sockets accordingly. Then, we recurse on the big and small groups. Thus, our runtime will be

$$T(n) \leq T(|L_{\text{big}}|) + T(|L_{\text{small}}|) + cn$$

Note that because we pick ℓ uniformly at random, and the bulbs/sockets are distinct sizes, this recurrence is exactly the same as the Quicksort recurrence. Thus, our algorithm has expected $O(n \log n)$ runtime. (Note: Using randomness allows us to get a runtime which has expected run time of $O(n \log n)$ on EVERY input. Otherwise, there might be "bad" inputs which run in $O(n^2)$ time.)

2 Batch Statistics

(Note: this problem has appeared on past midterms)

Design an algorithm which takes as input array A consisting of n possibly very large integers as well as an array R that contains k ranks $r_0, \ldots, r_k$, which are integers in the range $\{1, \ldots, n\}$. (You may assume that $k < n$.) The algorithm should output an array B which contains the r_j-th smallest of the n integers, for every j in $\{1, \ldots, k\}$. So if an $r_j = 3$ in input array R, then we want to return the 3rd smallest element in the input array A as part of the output.

Input: A which is an unsorted array of n unbounded distinct integers; R which is an unsorted array of k distinct ranks.
Output: B which is an array containing the r-th (smallest) integer of A for every r in R.

Example:

- Input: A = [11, 19, 13, 14, 16, 18, 17, 12, 15]; R = [3, 7]
- Output: [17,13]
- Explanation: 17 is the 7-th smallest element of A and 13 is the 3-rd smallest of A. [13,17] is also an acceptable output

Hint: we are looking for an $O(n \log(k))$-time algorithm.

Initial Ideas and Hints Let's first enumerate some 'naive' solutions, which is always recommended when approaching algorithm design questions. We could sort the array A in $O(n \log n)$ time, using something like MergeSort, and then we could just index into that sorted array at the positions/ranks indicated by R. Another 'naive' solution is to run the linear time SELECT algorithm on the array A for k times (once for each rank specified in R). For the example given, we would run SELECT$(A, 3)$ and SELECT$(A, 7)$ and return those values. This solution would be $O(nk)$, since each call to SELECT takes $O(n)$ and we run it k times.

Keep these solutions in mind, and think about how we can save on them and why they might be 'overkill'. The first solution feels like overkill because we're sorting the entire array A, which feels really wasteful if R is small (e.g. if R only cared about one rank, like the minimum). The second solution might feel wasteful because once we find out what one rank is (e.g. what the 3rd smallest element is), it feels like we could somehow use that information to save us trouble in the next rounds.

Notice that this problem has a similar structure to the previous one where each element in R is "matched" to an element in A (the r[i]th element of A), just as each light-bulb was matched to a socket. We might guess that a similar divide and conquer strategy might work if only we can figure out how to partition the A array and the R array into associated "smaller" and "larger" portions, and recurse. To partition A, you can't just use a rank from R, but instead need an element in A. Another interesting thing to be aware of is that our suggested runtime has the $\log(k)$ term.

English description of the algorithm

Find the median rank r_m using Select algorithm.*
Run Select algorithm to find a_m, the r_m-th smallest integer in A.
Recurse separately on (i) the ranks and integers greater than r_m and a_m (respectively); and (ii) the

ones smaller than r_m and a_m.

*Note that using the median of R makes the algorithm deterministic. You could have picked a random pivot in R instead (analogous to quicksort) and achieved expected runtime O(nlog(k)), but then the worst-case runtime would be O(nk), if we always pick min(R), for instance. When this question was used on an exam in the past we accepted both variations for full credit.

Rough Pseudocode
BATCH-SELECT(A, R)

- $n = |A|$ and $k = |R|$
- if $k == 0$, return []
- r_m = SELECT($R, \frac{k}{2}$)
- a_m = SELECT(A, r_m)
- R_{low} = all ranks in R less than r_m
- R_{high} = all ranks in R greater than r_m. From each we subtract r_m. See note.**
- A_{low} = all elements in A less than a_m
- A_{high} = all elements in A greater than a_m
- Return Concatenate-Lists([a_m], BATCH-SELECT(A_{high}, R_{high}), BATCH-SELECT(A_{low}, R_{low}))

**Note that we need to subtract r_m from each of the high ranks since the rank of each element in A_{high} was reduced when we removed the smaller elements.

Note about Proof of Correctness:
We omit the proof of correctness here, but the proof would use strong induction, and would be very similar in structure to the lightbulbs and sockets proof.

Runtime Explanation:
At each iteration we halve the size of R, and thus the recursion tree has a depth of $\log(k)$. At each level of the recursion, each element in R and A participates in one call to select, and is compared to one median. Thus the total work in each level is $O(n + k) = O(n)$, since n > k. Hence the total running time is $O(n \log(k))$. Note that it's possible for A_{low} and A_{high} to be very unevenly sized, but this won't impact the fact that the total work per level is $O(n)$ (please draw out the recursion tree or walk through the work involved at each level carefully to see this).