CS 161 (Stanford, Fall 2025) Section 3

1 Expectation

1. *True or False:* Expected runtime averages the runtime over the outcomes of random events within the algorithm for a random input.
2. I have an algorithm that takes positive integers (n, i) where $1 \leq i \leq n$. The algorithm rolls a n-sided die repeatedly until the die returns any value $\leq i$. What is the expected runtime in n? Worst-case runtime? Rigorous proof not necessary :)

Solution.

1. **False**. Expected runtime is calculated over random events in the algorithm, but an adversary is still allowed to choose a worst-case input! Worst-case runtime differs by allowing an adversary to also choose the outcomes of random events. The key thing to take away here is that, although we are not computing "worst-case" runtime, we are still partially performing worst-case analysis.
2. The worst possible input for i is $i = 1$, as that minimizes the probability that the die is $\leq i$. In that case, the algorithm returns only when the die rolls a 1. In expectation, this takes n rolls, so this algorithm has an expected runtime of $\boxed{O(n)}$. In the worst-case analysis, we can fix the die to always turn up $2 > 1$, gasp! The algorithm won't ever terminate; its worst-case runtime is $\boxed{\infty}$.

2 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 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.

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 prove your algorithm's correctness and expected runtime.

Solution.

Consider the following procedure for matching lightbulbs with their corresponding sockets. If

the cardinalities of L and S are equal to 1, then we know that $\ell \in L$ matches $s \in S$, so we can match and return. Otherwise, we run the following recursive procedure, $\texttt{Match}(L, S)$:

- 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 $s^\star$).
- For every other lightbulb ℓ': test whether ℓ' is too big or too small to fit into $s^\star$.
- 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 $s^\star$ respectively.
- $\texttt{Match}(L_{\texttt{big}}, S_{\texttt{big}})$
- $\texttt{Match}(L_{\texttt{small}}, S_{\texttt{small}})$

Correctness. Consider a given call to `Match`. 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, $s^\star$. Note that if a bulb is too big to fit in $s^\star$, then it must fit in a socket that was too big for ℓ; likewise, if a bulb is too small to fit in $s^\star$, 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. Thus, our recursive calls will correctly match the remaining bulbs to their corresponding sockets.

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/sockes 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 $\Omega(n^2)$ time.)

3 Batch Statistics

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 $r_0, ..., r_k$, which are integers in the range $\{1, ..., n\}$. (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 $1, ..., k$. 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.

Example:

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

Hint: we are looking for an $O(nlogk)$ runtime 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 $O(nlogn)$ 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 lightbulb matching problem from a previous section, 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 algorithm:

Find the median rank r_m using the 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.

Notes 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 $O(nlog(k))$. 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 $O(n)$.

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.
- A_{low} = all ranks in A less than a_m.
- A_{high} = all ranks in A greater than a_m.
- Return Concat-Lists($[a_m]$, BATCH-SELECT(A_{high}, R_{high}), BATCH-SELECT(A_{low}, R_{low}))

4 Sorting with Low Adaptivity

Sometimes, the steps of an algorithm don't depend on one another - this happens frequently in real world settings. When we have such an "independence" between steps, we can use parallelization to speed up the algorithm! (Outside this question we won't spend much time discussing this issue in this class.)

1. We say that a comparison-based sorting algorithm is *non-adaptive* if it commits in advance to the pairs of elements that it will compare. In other words, when choosing a comparison at step k, the algorithm does NOT rely on information from previous comparisons at steps $(1, ..., k-1)$. Prove that any non-adaptive sorting algorithm requires $\Omega(n^2)$ comparisons.

2. Now we define a “stage" of operations as an arbitrary-sized set of operations that all occur simultaneously. We say that a comparison-based sorting algorithm has *adaptivity* t if it can be executed in $t+1$ stages, where the pairs to be compared in the i-th stage only depend on outcome of comparisons in stages $1, \ldots, i-1$ (but not on other comparisons in the i-th stage). For example, non-adaptive algorithms have 0 adaptivity.

   What is the adaptivity of the MergeSort algorithm?

3. Give a (possibly randomized) algorithm with worst-case adaptivity 1 (aka 2 stages) and expected number of comparisons $n^{3/2}$.

   Hint - you may find the following derivations useful:

   (1) Suppose $k + \ell \leq n$. If I sample two sets of size k, ℓ at random (it's enough that one of them is random!) out of n elements, the probability that they do not intersect is given by:

$$\begin{aligned}
\Pr[\text{sets don't intersect}] &= \frac{\#\text{ of ways of picking } k \text{ out of } n-\ell}{\#\text{ of ways of picking } k \text{ out of } n} \\
&= \frac{\binom{n-\ell}{k}}{\binom{n}{k}} \\
&= \frac{(n-\ell)!(n-k)!}{n!(n-k-\ell)!} \\
&\leq \left(\frac{n-\ell}{n}\right)^k \\
&\leq e^{-k\ell/n}.
\end{aligned}$$

   (2) If I sum the above expression for many k's, I have:

$$\sum_{k=1}^{k\leq b} e^{-k\ell/n} < \sum_{k=0}^{\infty} e^{-k\ell/n} = \frac{1}{1-e^{-\ell/n}} = O(n/\ell).$$

Solution.

1. **Deterministic algorithms:**

   Assume by **contradiction** that we have a non-adaptive algorithm that uses less than $\binom{n}{2} = \Theta(n^2)$ comparisons. Then there must be a pair of elements i, j that it never compares. If i, j are consecutive in the sorted array (say the smallest two elements), it is impossible to know based on comparisons to other elements whether $i > j$ or $j > i$.

   Randomized algorithms:

For any fixed draw of the randomness with $< \binom{n}{2}$ comparisons, there is still a pair i, j that is never compared.

2. When merging two sub-arrays of length k, the algorithm is completely sequential, i.e. the result of each comparison determines the next two elements to be compared. Thus the Merge subroutine has adaptivity $\Theta(k)$.

   MergeSort has $\log(n)$ stages; in the ℓ-th stage we run many independent merges of size 2^ℓ-subproblems. Hence in total the adaptivity of MergeSort is given by:

$$\sum_{\ell=0}^{\log(n)} \Theta(2^\ell) = \Theta(2^{\log(n)}) = \Theta(n).$$

3. **Algorithm description:**

   Stage 1: Select $\sqrt{n}$ random pivots and compare them to each other and to all other elements. If the pivots are $p_1 < p_2 < \cdots < p_{\sqrt{n}}$, we can partition all other elements into $\sqrt{n}+1$ sub-problems, where the elements in the i-th sub-problem are between p_{i-1} and p_i (we let $p_0 = -\infty$ and $p_{\sqrt{n}+1} = +\infty$).

   Stage 2: In each sub-problem, compare every pair of elements (and then use the answers to run any sorting algorithm that we saw in lecture).

   Analysis of # comparisons:

   The algorithm clearly has adaptivity 1 (two stages), and Stage 1 takes $O(\sqrt{n} \times n) = O(n^{3/2})$ comparisons. It remains to analyze Stage 2; specifically we want to analyze, for each element j, the expected number X_j of elements that are greater than j-th element in its sub-problem. The total number of comparisons is then $\sum_j X_j$.

   We now bound $\mathbb{E}[X_j]$. To analyze the probability that the $(j+k)$-th element is in the same sub-problem, notice that we pick $\sqrt{n}$ out of n possible pivots, and if any of the $k+1$ elements in ranks $j, j+1, \ldots, j+k$ are picked, the $(j+k)$-th element will not be in the same sub-problem as j. So this is like the probability that random subsets of sizes $k+1$ and $\sqrt{n}$ don't intersect. By the hint (1), this is at most $e^{-k\sqrt{n}/n}$. Summing over all k, we have by hint (2) that:

$$\begin{aligned}\mathbb{E}[X_j] &= \sum_{k=0}^{n-j} \Pr[(j+k)\text{-th element in the same problem as } j] \\ &\leq \sum_{k=0}^{n-j} e^{-k\sqrt{n}/n} \\ &= O(n/\sqrt{n}) = O(\sqrt{n}).\end{aligned}$$

Therefore the total expected number of comparisons is:

$$\mathbb{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbb{E}[X_j] = \sum_{j=1}^{n} O(\sqrt{n}) = O(n^{3/2}).$$