CS 161 (Stanford, Winter 2026) Section 3

1 The power of randomness

Imagine you have a list of n restaurants in your area that you've never been to. Half of them are great restaurants that you'd want to return to over and over, and half of them won't be worth the cost in your opinion. The restaurants can be in an arbitrary order on this list, and you have no way of learning more about them besides actually going to eat at one.

(a) Design a randomized algorithm to find a restaurant you like in an expected $O(1)$ number of visits.

(b) Explain why any non-randomized algorithm will require $O(n)$ visits before finding a restaurant you like in the worst case.

Solution

(a) Select a restaurant at random from the list. There is a $1/2$ probability that it is good. Repeat until you find a good restaurant. The expected number of visits until you find a good restaurant is $2 = O(1)$.

(b) Any deterministic algorithm must decide which restaurant to visit next purely based on the restaurants you've already been to. Since the restaurants can be in any order, an adversary who knows your algorithm can organize the list so that there's a bad restaurant at the index your algorithm will select after having seen k bad restaurants for $k = 1, \ldots, n/2 - 1$. This forces you to see all $n/2 = O(n)$ bad restaurants before finding a good one.

2 Expected runtime

Consider an algorithm that takes positive integers (n, i) where $1 \leq i \leq n$. The algorithm rolls an n-sided die until the die returns any value $\leq i$.

(a) What is the expected runtime of this algorithm, in terms of n?

(b) What is the worst-case runtime of this algorithm, in terms of n?

(c) What is the average runtime of this algorithm in terms of n if i is drawn uniformly at random from $1, \ldots, n$?

Note: While this is a useful way to measure runtime in some cases, we won't use it in this class.

(d) Probability exercise: Going back to an arbitrary (not random) input (n, i), give an upper bound on the worst-case probability that the algorithm fails to terminate in $O(n^2)$ steps.

Hint: You may use the approximation $(1 + \frac{x}{n})^n \approx e^x$.

(e) How long do I need to let the algorithm run in order to have at least a $1 - c$ probability of success in the worst case, for some constant $c > 0$?

(f) To consider (take-home): Imagine various scenarios where you might be deciding whether to implement an algorithm with the runtime properties we've just calculated. Perhaps it's a safety-critical component of commercial aviation software, or an algorithm that's selecting a recipe you'll make for dinner each day. What are some questions or considerations you might have that could be answered by each of the calculations above?

Solution

(a) The worst possible input to this algorithm is $(n, 1)$, since this leaves only a $1/n$ probability of success on each roll. In this case, the expected number of rolls until a success is n (this is a **geometric random variable**), so the expected runtime is $O(n)$.

(b) In the worst case, every single dice roll fails, so the worst-case runtime is ∞.

(c) Similar to part (a), the expected runtime on an input (n, i) is n/i. Taking the average over all n possible inputs, the average runtime is

$$\frac{1}{n}\sum_{i=1}^{n}\frac{n}{i} = \sum_{i=1}^{n}\frac{1}{i} = O(\log n).$$

Why is this sum $O(\log n)$? (Actually, it's even $\Theta(\log n)$.) Here are two ways to prove it.

- Method 1: $\sum_{i=1}^{n} 1/i$ is roughly the area under the curve $f(x) = 1/x$ from $x = 1$ to $x = n$ (approximated with rectangles of width 1), and $\int_1^x 1/x\,dx = \log n - \log 1 = \log n$.
- Method 2: Matching up terms, we can see that

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots \leq \underbrace{1} + \underbrace{\frac{1}{2} + \frac{1}{2}} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}} + \ldots + \underbrace{2^k \cdot \frac{1}{2^k}} + \ldots.$$

On the right hand side, each group of fractions adds to 1, and $2^k - 1$ fractions make up k groups, for a sum of k. Therefore $2^k - 1$ terms on the left add up to no more than k. Substituting $n = 2^k - 1$ and rounding, $\sum_{i=1}^{n} 1/i \leq \lceil \log n \rceil + 1$, which shows that $\sum_{i=1}^{n} 1/i = O(n)$.
The other direction of inequality (to show $\sum_{i=1}^{n} 1/i = \Omega(\log n)$) is left as an exercise. Hint: get a lower bound by dividing the right hand side above by 2.

(d) The worst-case probability that a single roll fails is $1 - \frac{1}{n}$. The probability that $O(n^2)$ independent rolls fail in a row is therefore

$$\left(1 - \frac{1}{n}\right)^{O(n^2)} = \left((1 - \frac{1}{n})^n\right)^{O(n)} \approx \left(e^{-1}\right)^{O(n)} = e^{-O(n)}.$$

Note: this is not the same as $O(e^{-n})$. (Why not?) We can interpret $e^{-O(n)}$ as e^{-cn} for some unspecified constant c.
This is an exponentially small probability that the algorithm fails to terminate in $O(n^2)$ steps.

(e) Similar to above, the worst-case probability that t rolls in a row fail is $(1-1/n)^t$. We want to find the smallest value of t that makes this quantity smaller than c. Using the approximation from the hint,

$$\left(1-\frac{1}{n}\right)^t = \left((1-\frac{1}{n})^n\right)^{t/n} \approx e^{-t/n}.$$

Setting this equal to c and solving for t, we find

$$t = n\log(1/c).$$

This means we only need a constant multiple of n units of time to succeed with high probability.

(f) There are many possible answers; here are some general ideas:

- You might need part (a) if you have no control over the inputs of your algorithm, but you'll run it a bunch of times and just need to know how fast it will be on average.
- You might need part (b) if you have no prior knowledge about the inputs and need your algorithm to work fast every single time. (So the algorithm above probably shouldn't go in commercial aviation software...)
- You might need part (c) if you have a good idea of the distribution of inputs your algorithm will encounter, and just need to know how fast it will be in the long run.
- You might need part (d) if you need to make sure your algorithm succeeds nearly every single time, and want to know how much time to budget for this to happen.
- You might need part (e) if you have a very strict time budget and want to know how often you can expect your algorithm to succeed. E.g. maybe you get some data from the internet every second, and have only one second to make a prediction before the next batch comes.

3 Adaptive Algorithms

In practice, when the steps of a given algorithm don't depend on one another, we can often execute them in parallel to gain efficiency.

Take multiplying a vector by a scalar as an example. Each multiplication of the scalar with an element of the given vector is naturally independent of the other multiplications. So, these operations can be done in parallel.

In this question, we will be analyzing *comparison-based* sorting algorithms. A *comparison-based* sorting algorithm is one which determines the sorted order of an array by comparing pairs of its elements. Observe that MergeSort and QuickSort are both comparison-based sorting algorithms.

We say that a comparison-based sorting algorithm has *adaptivity* t if it runs in $t+1$ sequential iterations, where the pairs to be compared in the i-th iteration depend on the outcomes of comparisons in previous iterations $1, \ldots, i-1$. In other words, the algorithm cannot proceed to iteration i until all comparisons in previous rounds are complete.

Example: Consider a function $f(x)$ which inserts element x into sorted array A using a linear scan. First, we compare x with $A[0]$. If $x > A[0]$, we compare it with $A[1]$. Then if $x > A[1]$, we compare it with $A[2]$, and so on. Since we do not compare x with $A[i+1]$ until we know the result of the comparison with $A[i]$, each comparison is in its own sequential iteration, and thus the adaptivity of this function is $\Theta(n)$.

3.1 Adaptivity of MergeSort

What is the adaptivity of the MergeSort algorithm?

[We are expecting: Adaptivity in terms of big-Θ notation, and a clear explanation supporting this answer**]**

Solution

The adaptivity is $\Theta(n)$. 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 i-th stage we run many independent merges of size 2^i-subproblems. Hence in total the adaptivity of MergeSort is given by:

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

3.2 Adaptivity of QuickSort

What is the expected adaptivity of QuickSort? Note that there are many ways of solving this problem. One way is to consider "comparable pairs" i.e. pairs of elements that might still be compared by the algorithm, and analyze how the expected total number of comparable pairs changes over iterations of the algorithm.

[We are expecting: Adaptivity in terms of big-Θ notation, and a clear explanation supporting this answer**]**

Hint: Consider any sub-problem of QuickSort of size k at a current stage of $i-1$, and let X_{i-1} denote the number of comparisons done at this stage. How can we get $E[X_i]$, the

expected number of comparisons done at stage i? Assume that choosing a "good pivot" (pivot within the first to third quartile of the array) occurs with probability 1/2.

Solution

The adaptivity is $\Theta(\log(n))$.
First, notice that QuickSort's recursive tree has $\Omega(\log(n))$ levels, and each level depends on the previous one, so adaptivity is at least $\Omega(\log(n))$.
Now notice that all the comparisons in each level of the QuickSort recursive tree can run in parallel. Essentially, in order to partition an array around a pivot, we know (before the start of the partition) that we will compare all the elements in the array against the pivot. Within a single call to partition, whether or not an element will be compared with the pivot has nothing to do with the result of comparing another element with the pivot. If there are multiple subproblems on *a single tree level*, each of those subproblems are independent as well (i.e. we know ahead of time that we won't compare elements across different subproblems). Thus, we can run all these comparisons in parallel, or in one stage. Since each level of the recursive tree is one stage, what we need to do is to find the expected number of levels.
Consider any sub-problem of QuickSort of size k, and let the current stage be $i-1$. With probability 1/2, the pivot is chosen from middle of the array ($\{s_{k/4}, \ldots, s_{3k/4}\}$, where $s_1, s_2, \ldots, s_k$ denotes the sorted array). Let X_i denote the number of comparisons to be done at i^{th} stage. In the current $(i-1)^{th}$ stage, we performed $k-1$ comparisons to partition the array. In the next stage (i^{th}), we need to perform (1) at most $3k/4$ comparisons (since the choice of a good pivot would create a subproblem with at most $3k/4$ elements) with probability of 1/2, or (2) at most k comparisons (bad pivot) with probability of 1/2. Hence, we reduce the number of comparisons in expectation by at least $(1/8)k$. We can see this from the following:

$$E[X_i] \leq (1/2)(3k/4) + (1/2)(k) = (7/8)k = (7/8)X_{i-1}$$
$$E[X_i] \leq (7/8)E[X_{i-1}]$$

Using the above recurrence, we get that $E[X_i] \leq (7/8)^i E[X_0] = (7/8)^i \times O(n)$. For $i = c\log_{8/7}(n)$, we get that $E[X_i] = O(1)$. Hence, after $O(logn)$ stages, we get that the number of comparisons needed to be done is $O(1)$. Thus, adaptivity is $\Theta(\log n)$.

4 All on the same line

Suppose you're given n distinct ordered pairs of integers $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, where for all i, j, $x_i \neq x_j$ and $y_i \neq y_j$. Recall that two points uniquely define a line, $y = mx + b$, with slope m and intercept b. (Note that choosing m and b also uniquely defines a line.) We say that a set of points S is *collinear* if they all fall on the same line; that is, for all $(x_i, y_i) \in S$, $y_i = mx_i + b$ for fixed m and b. In this question, we want to find the maximum cardinality subset of the given points which are collinear – in less jargon, we're looking for the

maximum integer N such that we can find N of the given points the same line. Assume that given two points, you can compute the corresponding m and b for the line passing through them in constant time, and you can compare two slopes or two intercepts in constant time.

This is a challenging problem – so we're only going to pseudocode at a high level!

1. Design an algorithm to find a maximum cardinality set of collinear points in $O(n^2 \log n)$ time. If there are several maximal sets, your algorithm can output any such set. *Some hints:*

   - $O(n^2 \log n) = O(n^2 \log n^2)$, which looks like sorting n^2 items.
   - Start small; how would we verify that 3 points are on the same line?

2. It is not known whether we can solve the collinear points problem in better than $O(n^2)$ time. But suppose we know that our maximum cardinality set of collinear points consists of exactly n/k points for some constant k. Design a randomized algorithm that reports the points in some maximum cardinality set in expected time $O(n)$. Prove the correctness and runtime of your algorithms.

   Some hints:

   - Your expected running time may also be expressed as $O(k^2 n)$.
   - You algorithm might not terminate!

For your own reflection: Imagine that you, an algorithm designer, had to pick one of the algorithms in part (a) or (b) to implement in the autopilot of an airplane, as part of the route-planning of a self driving car, or in any other scenario in which human lives are at stake. Given what you know about the performance and worst-case scenario of each of the algorithms, which algorithm would you choose and why?

Solution

1. Consider the following procedure:
   - For all pairs of points, compute their slope and intercept (m, b)
   - Sort these pairs lexicographically (that is, in increasing order of m, and then in increasing order of b as a tiebreaker.)
   - Iterate through the pairs, and note where the longest run of identical (m, b) pairs occurs
   - Return a list of the points in this run of pairs

   We claim this procedure finds the maximum cardinality set of collinear points in $O(n^2 \log n)$ time.

   Correctness: We know that a line is defined uniquely by its slope and intercept. Thus, if two pairs of points give the same slope and intercept, all of the points in the pairs are collinear. If we sort pairs by their resulting (m, b), we know that all pairs of points with identical (m, b) values will be adjacent. Each set of k collinear points will have $\binom{k}{2}$ adjacent (m, b) pairs, so the largest set will correspond to

the maximum cardinality set of collinear points.

Runtime: We know there are $\binom{n}{2} = O(n^2)$ pairs of points. For a given pair of points, we can compute the slope and intercept in $O(1)$ time. Moreover, because we can compare (m, b) pairs in $O(1)$ time, we can run any comparison-based sorting algorithm to sort the (m, b) pairs in $O(n^2 \log n^2) = O(n^2 \log n)$ time.

2. Consider repeating the following procedure until success:
   - Sample two points uniformly at random
   - Compute their (m, b)
   - count $\leftarrow 2$
   - For all other points (x_i, y_i), pair it with one of the first two and calculate its (m_i, b_i). If $(m_i, b_i) = (m, b)$: count $\leftarrow$ count+1
   - if count $= n/k$: SUCCESS

   We claim this procedure will find the maximum cardinality set of collinear points with constant probability, so the expected number of repetitions needed is also a constant.

   Correctness: We are guaranteed the maximum cardinality set has n/k collinear points. Each iteration, we sample two points and find the corresponding linear $y = mx + b$. Then, we check for every other point, if the point is on the line. We repeat this process until we find a set of points with n/k collinear points, so we will repeat this procedure until we find the maximum cardinality set of collinear points.

   Runtime: In each iteration, we sample two points in constant time and then go through every other point. Thus, each iteration will take $O(n)$ time. The question that we must ask is how many iterations do we need in expectation until we find the right line. We know that n/k points are on the line with the maximum number of points. Thus, if we sample two points from all the points uniformly at random, the probability of selecting two points on the line is

$$\begin{aligned}
\frac{n/k}{n} \cdot \frac{n/k - 1}{n - 1} &= \frac{1}{k} \cdot \frac{\frac{n-k}{k}}{n-1} \\
&= \frac{1}{k^2} \cdot \frac{n-k}{n-1} \\
&\geq \frac{1}{2k^2} && \text{for } n \geq 2k - 1 \\
&= \Omega(k^{-2}).
\end{aligned}$$

   We can view the number of iterations our algorithm takes as a geometric random variable - flipping a coin with probability $p = \Omega(k^{-2})$ until we get a success. The expected value of a geometric random variable is $\frac{1}{p} = O(k^2)$. Thus, our overall expected runtime is $O(k^2 n) = O(n)$ because k is a fixed constant.

   The worst-case runtime is ∞; if we have control over the sampling, we can always choose a pair of points that don't lie on a line containing n/k points. As

in problem 2, you can consider the probability that the algorithm doesn't return after T iterations; you'll find that it exponentially decreases with T.