CS 161 (Stanford, Winter 2025) 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.

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?

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**]**

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 $1/2$.

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?