Exercise 1
Suppose you’re given \( n \) distinct ordered pairs of integers \( (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \), 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. 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.

a. 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. Since we haven’t covered hashing yet, your algorithm should not use any form of hash table.

b. 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 \( O(n) \). (Hint: Your running time may also be expressed as \( O(k^2 n) \).) Prove the correctness and runtime of your algorithms.

c. Is your algorithm from part b guaranteed to terminate?

Solution 1

a. Consider the following procedure:

- For all pairs of points, compute their slope and intercept \((m, b)\)
- Sort these pairs lexicographically
- 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.

b. Consider repeating the following procedure until success:

- Sample two points uniformly at random
- Compute their $(m, b)$
- Count ← 2
- For all other points $(x_i, y_i)$:
  - if $y_i = mx_i + b$: count ← 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 line $y = mx + b$. 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 at least

$$\frac{\binom{n/k}{2}}{\binom{n}{2}} = \frac{n/k}{n} \cdot \frac{n/k - 1}{n - 1}$$

$$= \frac{1}{k} \cdot \frac{n - k}{n - 1}$$

$$= \frac{1}{k^2} \frac{n - k}{n - 1}$$

$$= \frac{1}{k^2} \left(1 - \frac{k - 1}{n - 1}\right)$$

$$= \Omega(k^{-2}). \text{ [for constant } k]\right)$$

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 $1/p = O(k^2)$. Thus, our overall expected runtime is $O(k^2 n) = O(n)$ because $k$ is a fixed constant.

c. The algorithm is not guaranteed to terminate but the probability of failure decreases exponentially.

**Exercise 2**

In lecture, we saw the quicksort algorithm, which is an example of a Las Vegas algorithm: a randomized algorithm that is always correct, but that its run time is a random variable. In this question, we consider another family of randomized algorithms: Monte Carlo algorithms.
A Monte Carlo algorithm can return a incorrect answer with some probability. In this question we will see how to amplify the success probability of such algorithms. We assume that the problem we are trying to solve is a decision problem, that is, the two possible outputs are Yes or No.

a. Assume that you are given access to an algorithm $A$ that has one-sided error: if the correct answer is No, $A$ always returns No, and if the correct answer is Yes, $A$ returns Yes with probability $1/2$ and No with probability $1/2$. For a given $p < 1/2$, design a randomized algorithm that fails (that is, reports the wrong answer) with probability at most $p$. What is the run time of your algorithm?

b. Assume that you are given access to an algorithm with two-sided error: given any input, it returns the correct answer with probability $2/3$ and the wrong answer with probability $1/3$. For a given $p < 1/3$, design a randomized algorithm that fails with probability at most $p$. What is the run time of your algorithm?

Solution 2

a. Consider the following algorithm:

(a) Perform the following $\lceil \log(1/p) \rceil$ times:

Run $A$ independently on the input. If the result is Yes, return Yes.

(b) If all the $\lceil \log(1/p) \rceil$ independent runs returned No, return No.

Correctness: Consider two cases. If the correct answer is No, all the runs of $A$ return No, and our algorithm eventually returns No. That is, on these inputs, the failure probability is 0.

If the correct answer is Yes, each run of $A$ returns Yes with probability $1/2$. Our algorithm fails, that is, we return No, only if all the runs of $A$ return No. That happens with probability

$$\left(\frac{1}{2}\right)^{\lceil \log(1/p) \rceil} \leq \left(\frac{1}{2}\right)^{\log(1/p)} = \frac{1}{2^{\log(1/p)}} = p.$$  

Run time: Assume that the run time of algorithm $A$ is $f(n)$ (on inputs of size $n$). We perform $\lceil \log(1/p) \rceil$ runs of $A$, so the total run time is $O(f(n) \log(1/p))$.

b. Consider the following algorithm:

(a) Run $A \lceil 48 \ln(1/p) \rceil$ times independently.

(b) If the majority of runs returned Yes, return Yes. Otherwise, return No. (Break ties arbitrarily.)

Correctness: Assume that the number of independent runs of algorithm $A$ is $k$. We show that picking any $k \geq 48 \ln(1/p)$ will lead to failure probability at most $p$. For $i = 1, \ldots, k$, let $X_i$ be an indicator random variable that is 1 when the $i$-th run of $A$ returned the correct answer (happens with probability $2/3$) and 0 otherwise. These $X_i$ are independent. Our algorithm returns the same answer has the majority of the runs. Rewriting this condition using the $X_i$, our algorithm fails if and only if

$$\sum_{i=1}^{k} X_i \leq \frac{1}{2} k.$$  

Note that $E[X_i] = 2/3$ and $E\left[\sum_{i=1}^{k} X_i\right] = \frac{2}{3} k$. We can use Chernoff bound to get an upper
bound on the failure probability:

\[
\Pr \left[ \sum_{i=1}^{k} X_i \leq \frac{1}{2} k \right] = \Pr \left[ \sum_{i=1}^{k} X_i \leq \left( 1 - \frac{1}{4} \right) \frac{2}{3} k \right] \\
\leq e^{-\frac{1}{12} \frac{2}{3} k} \\
= e^{-\frac{1}{48} k}.
\]

Our goal is to get the failure probability to be at most \( p \). If we pick \( k \) such that \( e^{-\frac{1}{48} k} \leq p \), the failure probability is going to be at most \( p \). Specifically, for any \( k \geq 48 \ln (1/p) \), we get that

\[
\Pr \left[ \sum_{i=1}^{k} X_i \leq \frac{1}{2} k \right] \leq e^{-\frac{1}{48} k} \leq p.
\]

Note: Chernoff bounds are an advanced topic (out of scope for this class). You are not expected to be familiar with them. The goal of this problem is introduce the really cool (and useful) idea that the success probability of randomized algorithms can be amplified!

**Run time:** Assume that the run time of algorithm \( A \) is \( f(n) \) (on inputs of size \( n \)). We perform \([48 \ln(1/p)]\) runs of \( A \), so the total run time is \( O(f(n) \log (1/p)) \).

**Exercise 3**

You are given a collection of \( n \) differently-sized light bulbs that have to be fit into \( n \) flashlights in a dark room. You are guaranteed that there is exactly one appropriately-sized light bulb for each flashlight and vice versa; however, there is no way to compare two bulbs together or two flashlights together as you are in the dark and can barely see! (You are, however, able to see where the flashlights and light bulbs are.) You can try to fit a light bulb into a chosen flashlight, from which you can determine whether the light bulb’s base is too large, too small, or is an exact fit for the flashlight. 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/flashlights to other bulbs/flashlights.)

Suggest a (possibly randomized) algorithm to match each light bulb to its matching flashlight. 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 3**

Consider the following procedure for matching light bulbs with their corresponding flashlights. If the cardinalities of \( L \) and \( F \) are equal to 1, then we know that \( \ell \in L \) matches \( f \in F \), so we can match and return. Otherwise, we run the following recursive procedure:

\textbf{Match}(\( L, F \)):

- Select a lightbulb \( \ell \in L \) uniformly at random
- For every flashlight \( f \in F \): test whether \( \ell \) is too small, too big, or just right. Call the matching flashlight \( f^* \).
- For every other lightbulb \( \ell' \): test whether \( \ell' \) is too big or too small to fit into \( f^* \)
• $F_{\text{big}}, F_{\text{small}} \leftarrow$ flashlights too big and too small for $\ell$, respectively
• $L_{\text{big}}, L_{\text{small}} \leftarrow$ lightbulbs too big and too small for $f^*$, respectively
• Match($L_{\text{big}}, F_{\text{big}}$)
• Match($L_{\text{small}}, F_{\text{small}}$)

**Correctness:** Consider a given call to Match. For the lightbulb we pick at random, $\ell$, we go through all the flashlights in $F$, so we are guaranteed to find its unique matching flashlight, $f^*$. Note that if a bulb is too big to fit in $f^*$, then it must fit in a flashlight that was too big for $\ell$; likewise, if a bulb is too small to fit in $f^*$, then it must fit in a flashlight that was too small for $\ell$. Thus, we can partition the bulbs and flashlights simultaneously, such that we only have to compare “small” bulbs to “small” flashlights and “big” bulbs to “big” flashlights. Thus, our recursive calls will correctly match the remaining bulbs to their corresponding flashlights.

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

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

for some constant $c$. Note that because we pick $\ell$ uniformly at random, and the bulbs/flashlights are distinct sizes, this recurrence is exactly the same as the Quicksort recurrence. Thus, our algorithm has expected $O(n \log n)$ runtime and worst-case $O(n^2)$ runtime. (Note: Using randomness allows us to get an expected runtime of $O(n \log n)$ on EVERY input. Otherwise, there might be “bad” inputs which run in $\Omega(n^2)$ time.)