## Runtime Expectations

1. *True or False:* Expected runtime averages the runtime over the outcomes of random events within the algorithm and make no assumption about the input.

**SOLUTION: False**. Expected runtime is calculated over random events in the algorithm, but an adversary is still allowed to choose an 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. 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?

**SOLUTION:** 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 $O(n)$.

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 $\infty$.

## More Sorting!

We are given an unsorted array A with $n$ numbers between 1 and M, where M is a large but constant position integer. We want to find if there exist two elements of the array that are within $T$ of one another.

1. Design a simple algorithm that solves this in $O(n^2)$.
2. Design a simple algoirthm that solves this in $O(n \log n)$.
3. How could you solve this in $O(n)$? (Hint: modify counting sort.)

**SOLUTION:**

1. Compare all pairs of numbers to see if any are within T of each other.
2. Sort the array, then compare only adjacent elements to see if they are within T of each other.

3. Because we know that the elements are from 1 to M, if they are integers, we can simply use counting sort to sort the elements and check subsequent elements to see if they are T apart. If we cannot create M buckets due to memory constraints, or if the array elements are real numbers, we could split the items up into buckets of size T. If any bucket has 2 or more items, then those elements are within T of each other. Otherwise, each bucket holds at most 1 element (indeed, the elements are sorted) and we only need to check pairs of elements in adjacent buckets, and there at most $n - 1$ such pairs.

   If creating M/T buckets would take up too much memory, you can create a hierarchy and only split up the non-empty cells. For example, a hierarchy of depth 2 could split the range of 1 to M into buckets of size $T^2$, then take only the non-empty buckets and split those into size T.

**Problem Solving Notes:**

1. *Read and Interpret:* The runtimes for each algorithm in this problem are provided for you. How can we use this information to guide us towards approaches and algorithms we have seen in class?
2. *Information Needed:* What information do you need in order to use counting sort? It's important to consider memory constraints, and the information we glean from the number of each items in each bucket when we face those constraints. Why do we mention using either buckets of size $M$ or size $M/T$?
3. *Solution Plan:* Once you've decided on the number of buckets to use in counting sort, devise a strategy for answering the question posed above.

## 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?

**SOLUTION:** 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 paris 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.

**Problem Solving Notes:**

(a) *Read and Interpret:* We are working with a unique input for this problem: sets of points! We can use geometry fundamentals to calculate their properties (slope and intercept) and sort them accordingly.

(b) *Information Needed:* What information do you need in order to determine if a set of points—consisting of pairs—is collinear? What's the easiest way for us capture collinear groups when creating our algorithm?

(c) *Solution Plan:* Once you've determined the best way to represent and sort each possible pair of points, find the largest set of collinear points.

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!

**SOLUTION:** Consider repeating the following procedure until success:

- Sample two points uniformly at random
- Compute their $(m, b)$
- count $\leftarrow 2$
- For all other points $(x_i, y_i)$: if $y_i = mx_i + b$: 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

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

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

The worst-case runtime is $\infty$; 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. On your own time, consider the probability that the algorithm doesn't return after $T$ iterations; you'll find that it exponentially decreases with $T$.

**Problem Solving Notes:**

(a) *Read and Interpret:* The problem statement recommends using a randomized algorithm to obtain the expected time $O(n)$. What data do we need to randomly select to begin the process of finding the largest set of collinear points?

(b) *Information Needed:* Once we make our random selection, our following steps must be completed in $O(n)$ time.

(c) *Solution Plan:* Once we have our strategy for our randomized algorithm, what checks do we have to put in place to meet the recommended runtime?

*For your own reflection:* Imagine that you, an algorithm designer, had to pick one of the algorithms in part (1) or (2) 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?

## 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 7-th 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 $O(n \log k)$ runtime algorithm.*

**SOLUTION:** 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 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:**

1. Find the median rank $r_m$ using the SELECT algorithm.
2. Run the SELECT algorithm to find $a_m$, the $r_m$-th smallest integer in A.
3. 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 $O(n \log(k))$, 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.

**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 from last section.

**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 $O(n)$.

**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$(A_{low}, R_{low})$)

**Problem Solving Notes:**

1. *Read and Interpret:* The runtime suggested—$O(n \log k)$—is smaller than "naive" solutions we have seen in previous homeworks, such as applying MergeSort and SELECT. What information or steps do those algorithms rely on that we could possibly eliminate in our approach here?
2. *Information Needed:* The $\log k$ term in the suggested runtime suggests to us that we will need to recursively divide-and-conquer this problem. What properties of the information provided to us would allow us to do so?
3. *Solution Plan:* Once you've decided how to divide the problem at each step, determine what calls you need to make at each step of the tree and calculate the algorithm's runtime.