Problem Set 2 Solutions

This problem set is due at 11:59pm on Monday, March 4, 2013.

Problem 2-1. Points on a Circle

In this problem you will design a data structure for operating on a set of points on the unit circle. For simplicity, you may assume that the circle is centered at (0, 0), has radius 1, and that all points are specified by the angle (in radians) that a line connecting them to the origin makes with the x-axis. For example, the point (0, 1) would be represented by π/2, and (-1, 0) would be represented by π. Also assume that all basic arithmetic operations can be performed in O(1) time.

The operations your data structure should support are the following:

- **INSERT(X):** Inserts a point at angle X radians to the x-axis.
- **ROTATE(R):** Rotates all points R radians counter-clockwise.
- **CLEAR-CLOCKWISE(Y):** If there are currently n points in the data structure, removes the first ⌈n/2⌉ points found by starting at Y radians to the x-axis and moving clockwise.

(a) Describe a data structure that implements INSERT and ROTATE that runs in worst-case time O(1).

**Solution:** Store the points in any standard data structure (a linked list works). Additionally, keep track of a variable r, initialized to 0, which stores the total number of radians that the points have been rotated since the beginning of time. Our goal is to maintain the invariant that if a point is stored locally at X radians, that its true location is X + r radians.

To implement ROTATE(R), just set r := r + R. This takes time O(1), as it is a single addition.

To implement INSERT(X), append to the linked list a new node at X − r radians.

It is clear that the invariant is maintained immediately after X is inserted: we store it locally at X − r radians, and X − r + r = X. Any time we rotate the circle by R radians, this is reflected by updating r.

(b) Describe an implementation of CLEAR-CLOCKWISE that runs in worst-case time O(n).

**Solution:** First, for every point, p, we need to find how many radians it takes to get from Y to p going clockwise. This can be done by traversing the linked list and
computing for every stored value $X$, $Y - X - r \pmod{2\pi}$. This takes time $O(1)$ per stored value, which totals to $O(n)$ (if you don’t want to count modding by $2\pi$ as a basic arithmetic operation, you can still implement it in time $O(1)$ using the formula $Y - X - r \pmod{2\pi} = Y - X - r - \lfloor Y - X - r/(2\pi) \rfloor$, which can be computed using just addition and subtraction.)

Now, we have an unordered list of values telling us how long it takes to reach each point from $Y$ moving clockwise. In order to remove the first $\lceil n/2 \rceil$ points, we just need to find the lowest $\lceil n/2 \rceil$ points in this list. To do this, let’s first find the median using divide-and-conquer in time $O(n)$ (using the deterministic algorithm shown in class). Now, we just need to start from $Y$ and remove every point that we reach before the point corresponding to the median of this list. So traverse the linked list, and remove every point whose corresponding value is less than the median. Traversing the list takes time $O(n)$. At each element in the list, we do a single comparison, and possibly a linked-list delete, which both take time $O(1)$.

There are three steps that all take time $O(n)$: computing the number of radians it takes to get to $p$ from $Y$ for all points $p$, finding the median of this list, and removing all points whose value is less than the median. So the total runtime is $O(n)$.

(c) Show that, as long as your implementation of INSERT and ROTATE have worst-case runtime $O(1)$ and your implementation of CLEAR-CLOCKWISE has worst-case runtime $O(n)$, that any sequence of $m$ operations performed by your data structure has worst-case runtime $O(m)$ (and therefore the amortized cost of each operation is $O(1)$).

Solution: We will use the potential method. Let $C_1$ be the worst-case runtime of INSERT, $C_2$ the worst-case runtime of ROTATE, and $C_3n + C_4$ be the worst-case runtime of CLEAR-CLOCKWISE. We define the potential function $\phi(D)$ to be $2C_3n$, where $n$ is the current number of points stored.

Then as ROTATE changes $D$ to $D'$, we get $\phi(D') - \phi(D) = 0$, and therefore the amortized cost of ROTATE is $C_2$.

As INSERT changes $D$ to $D'$, we get $\phi(D') - \phi(D) = 2C_3$, and therefore the amortized cost of INSERT is $C_1 + 2C_3$.

As CLEAR-CLOCKWISE changes $D$ to $D'$, we get $\phi(D') - \phi(D) \leq -C_3n$, and therefore the amortized cost of CLEAR-CLOCKWISE is $\leq C_3n + C_4 - C_3n = C_4$.

Lastly, as the data structure begins empty, $\phi$ is initially 0. As there are always a non-negative number of points stored, $\phi$ is always $\geq 0$. So the potential function is valid, and therefore each operation has amortized cost $O(1)$.

Problem 2-2. Recursive Electoral Colleges

MIT’s new Committee on Committee Recursion is running a two-party election: EE versus CS. Assume that the population size $n = 5^k$ is a power of 5. Create a quinary tree of height $k$, where
each internal node has exactly five children and each leaf corresponds to a unique voter. Define the value of each leaf to be the respective voter’s choice of candidate, and the value of each internal node to be the majority of the its five children’s values. The overall winner is the value of the root.

Consider the following randomized algorithm for determining the winner. At the top level, we call COUNT-VOTES(root). At each recursive level, the procedure COUNT-VOTES(node) randomly selects three out of the five children of node. If those children’s values all agree (which is determined recursively), then the algorithm returns this common preference, because it must be the majority of all five children. Otherwise, the algorithm recursively determines the preference of a fourth random child from the two remaining. If three out of the four children agree, then the algorithm returns this majority, because again it must be the majority of all five children. Otherwise, the algorithm recursively examines the remaining fifth child, and declares the majority preference to be the winner of this subtree. More precisely, the recursion works as follows:

COUNT-VOTES(node)
1 if node is a leaf
2 then return node.vote
3 child₁, child₂, child₃ ← three randomly selected children of node (among the five)
4 choice₁, choice₂, choice₃ ← COUNT-VOTES(child₁), COUNT-VOTES(child₂), COUNT-VOTES(child₃)
5 if choice₁ = choice₂ = choice₃
6 then return choice₁
7 child₄ ← a randomly selected child from the two unpicked children of node
8 choice₄ ← COUNT-VOTES(child₄)
9 if three out of choice₁, choice₂, choice₃, choice₄ agree
10 then return majority of {choice₁, choice₂, choice₃, choice₄}
11 child₅ ← remaining unpicked child of node
12 choice₅ ← COUNT-VOTES(child₅)
13 return majority of {choice₁, choice₂, choice₃, choice₄, choice₅}

(a) Analyze the worst-case expected asymptotic running time of COUNT-VOTES(root) as a function of the population size n.

Solution: This problem is based on Motwani and Raghavan’s Randomized Algorithms Problem 2.3. It is clear that the worst-case happens when each group is split 3-to-2. The recurrence relation that describes the expected running time is

\[ T(k) = \frac{1}{10} 3T(k-1) + \frac{3}{10} 5T(k-1) + \frac{6}{10} \left( \frac{1}{2} 4T(k-1) + \frac{1}{2} 5T(k-1) \right) = 4.5T(k-1), \]

where \( T(1) = O(1) \).

Hence \( T(k) = O(4.5^k) = O(n^{\log_5 4.5}) = O(n^{0.94}) \).
Problem Set 2 Solutions

Now instead of studying the worst case, you will analyze the average case. Suppose that each voter decides independently of the others whether to vote for CS or EE, and that each voter chooses CS (independently) with probability $p$.

(b) What is the asymptotic probability (as $n$ goes to infinity) that COUNT-VOTES(root) will return CS as the winner? How does the answer change depending on $p$?

*Hint:* Write a recurrence to determine the limit probability, and use the following fact about fixed points (no proof necessary). Consider the iteration $x_0, x_1, x_2, \ldots$ where $x_{i+1} = f(x_i)$. If $f : [0, x_0] \to [0, x_0]$ is continuous and has a fixed point at 0 (i.e. $f(0) = 0$), and $f(y) < y$ for all $y \in (0, x_0]$, then the iteration $x_i$ converges to 0.

*Solution:* As you might expect, the asymptotic probability that COUNT-VOTES(root) will return CS as the winner is $\frac{1}{2}$ if $p < \frac{1}{2}$, 1 if $p = \frac{1}{2}$, and $\frac{1}{2}$ if $p > \frac{1}{2}$. The $p = \frac{1}{2}$ case is easy, and can be proved by symmetry: given any instantiation of votes, it is equally likely that all votes are switched, which would yield the opposite result. This induces a one-to-one mapping between equally likely collections of votes that yield CS and votes that yield EE, which therefore means that the probability of returning CS is equal to the probability of returning EE and that they are both $\frac{1}{2}$.

To prove the $p < \frac{1}{2}$ case, let $p_k$ denote the probability that COUNT-VOTES(root) returns CS on a tree of height $k$. Then, because a node at height $-k$ declares CS as winner if and only if at least three of its five children (who are nodes of height $k-1$) declared CS as winner, we have the following recurrence:

$$p_k = \frac{5}{3} p_{k-1}^3 (1 - p_{k-1})^2 + \frac{5}{4} p_{k-1}^4 (1 - p_{k-1}) + \frac{5}{5} p_{k-1}^5$$

After evaluating the binomial terms and rearranging, we obtain

$$p_k = p_{k-1}^3 \left( 10(1 - p_{k-1})^2 + 5p_{k-1}(1 - p_{k-1}) + p_{k-1}^2 \right)$$

We know that $p_k$ has (at most) five fixed points because it is a recurrence of degree 5. We also know that 0 and 1 must be fixed points (because if all the children pick the same thing deterministically, so must the root), and also $\frac{1}{2}$ by the work above. After factoring these out, we can compute the other two roots to be $\frac{5}{4} \pm i \frac{1}{4} \sqrt{\frac{5}{3}}$ using the quadratic formula. Since these roots are imaginary, we will not need to worry about our recursion converging to them.

Now, we observe (by looking at the function $x^3 (10(1-x)^2 + 5x(1-x) + x^2) - x$ that $p_k > p_{k-1}$ when $0 < p < \frac{1}{2}$. Therefore, as $k \rightarrow \infty$, $p_k$ must converge to a fixed point (it cannot oscillate), and the fixed point at the bottom of this range is 0. Therefore, if $p < \frac{1}{2}$, $\lim_{k \rightarrow \infty} p_k = 0$.

A symmetric argument implies that, when $p > \frac{1}{2}$, $\lim_{k \rightarrow \infty} p_k = 1$. 
Analyze the average asymptotic running time of \textsc{Count-Votes}(\textit{root}) as a function the population size \( n \). You should analyze the case \( p = \frac{1}{2} \) separately from the rest, and otherwise should treat \( p \) as an absolute constant (i.e., \( p \) need not appear in the running time).

\textbf{Solution:}

If \( p = \frac{1}{2} \), then by the work in the previous part, each child has a 50-50 chance of choosing CS or EE. Therefore, the probability that we need to inspect only three nodes (that each of the first three inspected agree) is \( \frac{1}{4} \). The probability that we need to inspect exactly four (that the first three disagree, and the fourth agrees with the majority) is exactly \( \frac{3}{8} \). The probability that we need to inspect all five is also \( \frac{3}{8} \). Therefore, the recurrence for the runtime is:

\[
T(k) = 3\frac{1}{4}T(k-1) + 4\frac{3}{8}T(k-1) + 5\frac{3}{8}T(k-1) + O(1)
\]

which solves to:

\[
T(k) = O\left(\frac{33^k}{8}\right) = O(n^{\log_5 \frac{33}{8}}) \approx O(n^{0.88})
\]

When \( p < \frac{1}{2} \), observe first that the recurrence derived in the previous part implies that \( p_k \leq 10p_{k-1}^3 \). In other words, if we take \( k \) sufficiently large so that \( p_{k-1}^3 < \frac{1}{10} \), \( p_k \) approaches 0 extremely fast (faster than exponentially). Let’s call the first value of \( k \) for which \( p_k < \frac{1}{10} \) \( k' \). Now consider a node at height \( k = x + k' \). Our recurrence implies that \( p_k \leq c^{3x} \). Certainly, the node needs only inspect three children if all five children vote for EE, which happens with probability at least \( 1 − 5p_{k'} \). And also, the node needs only inspect five children no matter what. So we get:

\[
T(k) \leq 3(1 − 5c^{3(k-k')})T(k-1) + 25c^{3x}T(k-1) + O(1)
\]

\[
T(k) \leq 3T(k-1) + 20c^{3(k-k')}T(k-1) + O(1)
\]

From here, one can verify using recursion trees that because \( c < 1 \), that this recursion yields the same solution as \( T(k) = 3T(k-1) + O(1) \). So the asymptotic runtime is \( O(3^k) = O(n^{\log_5 3}) \approx O(n^{0.69}) \). The same clearly holds when \( p > \frac{1}{2} \). To summarize, we first argued that up to a constant height (depending on \( p \)), the total amount of work done is a constant. For further heights, the probability that there is any disagreement decreases superexponentially fast, and therefore makes it extremely likely that only three children need to be probed for any node above this height. In fact, the probability of disagreement shrinks so quickly that even as the probability builds up as we move down the recursion tree, it never amounts to anything.
Problem 2-3. Windowing Queries in 1D and 2D

In computer graphics, the problem of \textit{windowing queries} asks to find all visible objects inside a view rectangle. For example, one might want to find all roads on a computerized map inside a rectangular window. In this problem, we will construct static data structures to support windowing queries in 1D and 2D, respectively.

In 1D, both the objects and the query window can be modeled as horizontal segments. Under this model, we are given \( n \) horizontal segments \( S_1 = [a_1, b_1], S_2 = [a_2, b_2], \ldots, S_n = [a_n, b_n] \), and we want to build a static data structure on them that can efficiently output all segments overlapping a given query segment \( Q = [a, b] \).

Inspired by the divide-and-conquer algorithms in this class, you decide to compute the median \( x_{\text{mid}} \) of the multiset of endpoints \( \{a_1, b_1, a_2, b_2, \ldots, a_n, b_n\} \), and divide the segments into three sets:

- \( L \): all segments \( S_i \) lying completely left of \( x_{\text{mid}} \) (i.e., for which \( b_i < x_{\text{mid}} \)).
- \( R \): all segments \( S_i \) lying completely right of \( x_{\text{mid}} \) (i.e., for which \( a_i > x_{\text{mid}} \)).
- \( M \): all segments \( S_i \) that intersect \( x_{\text{mid}} \) (i.e., for which \( a_i \leq x_{\text{mid}} \leq b_i \)).

Then you build a binary tree whose root stores \( x_{\text{mid}} \) and a list of the segments in \( M \), whose left subtree recursively represents \( L \), and whose right subtree recursively represents \( R \).

(a) Modify this data structure to implement 1D window queries in \( O(\lg n + k) \) time, where \( k \) is the number of overlapping (output) segments.

\textit{Hint}: 1. Organize the segments in \( M \) into two lists so that, if \( m \) segments in \( M \) intersect with \( Q \), we can find and report them in \( O(m) \) time. 2. Prove that the height of this binary tree is \( O(\log n) \).

\textbf{Solution}: This data structure is called \textit{interval tree}. We claim that the height of an interval tree on \( n \) segments is \( O(\log n) \). This is because each time through the recursion we split the segments into two subsets \( L \) and \( R \) of sizes at most half the original size. Thus after at most \( \log n \) levels the recursion will bottom out.

We implement 1D window queries as a recursive procedure, \textbf{QUERY}(\( N, Q \)), where \( N \) is a node in the interval tree and \( Q = [a, b] \) is the query segment (so in the beginning, we would call \textbf{QUERY}(\textbf{tree.root}, Q)). We output all the found overlapping segments in a set \( T \). There are three cases:

1. If \( N.x_{\text{mid}} \) is in \([a, b]\), we know that all the segments stored in \( N \) overlap with \( Q \); so we add them to \( T \). Then we call \textbf{QUERY}(\( N.left\_child, Q \)) and \textbf{QUERY}(\( N.right\_child, Q \)).

2. If \( N.x_{\text{mid}} < a \), we know that all the segments in the left subtree of \( N \) do not overlap with \( Q \). On the other hand, we add to \( T \) all the segments in \( N \) whose right end points are no less than \( a \) (i.e., \( b_i \geq a \)). This can be done in linear time by storing these segments’ right end points in descending order (during the construction of the tree) and stopping the search once we encounter a \( b_i \) that is less than \( a \). Then we call \textbf{QUERY}(\( N.right\_child, Q \)).
3. If $N.x_{mid} > b$, we know that all the segments in the right subtree of $N$ do not overlap with $Q$. On the other hand, we add to $T$ all the segments in $N$ whose left end points are no greater than $b$ (i.e., $a_i \leq b$). This can be done in linear time by storing these segments’ left end points in ascending order (during the construction of the tree) and stopping the search once we encounter a $a_i$ that is greater than $b$. Then we call $\text{QUERY}(N.left\_child, Q)$.

Note that at the first time that case 1 happens, we start going down the tree along two separate paths. Call these two paths the boundary paths and the nodes along them the boundary nodes. We claim that on each level of the interval tree, all segments stored in nodes between the two boundary nodes (of the level) must be overlapping with the query segment. This is because otherwise, the query segment cannot be continuous. So the total run-time of $\text{QUERY}$ can be broken into the work done going down the two boundary paths, which is $O(\log n)$ and the work done checking/adding the segments within the boundary paths, which is $O(k)$.

In 2D, for simplicity, we model the objects as either horizontal or vertical segments, and the query window as an axis-aligned rectangle $[x_1, x_2] \times [y_1, y_2]$.

(b) Given $n$ vertical and horizontal segments in the plane, and given a query rectangle, design a static data structure of size $O(n \log n)$ supporting the query of finding all $k$ segments intersecting the query rectangle in $O(\log^2 n + k)$ time.

Hint: Use part (a), range trees, and/or augmentation.

Solution: Consider the problem of finding all (horizontal and vertical) segments that intersect a query rectangle $Q = (X_1, Y_1, X_2, Y_2)$. A solution can be expressed as the set of all horizontal line segments that intersect the vertical line through $X_1$ or $X_2$ and lie between $Y_1$ and $Y_2$ on the vertical axis, and the corresponding set of vertical line segments (replacing “horizontal” with “vertical” and vice versa above). We give the procedure for identifying the intersecting horizontal line segments (for former case); the latter case is precisely symmetric. Consider a horizontal segment $s_i$. It is defined by its left and right endpoints $x_i$ and $x'_i$ and its $y$-coordinate, $y_i$. Our problem is to find all $s_i$ such that $Y_1 \leq y_i \leq Y_2$ and the interval $[x_i, x'_i]$ intersects $[X_1, X_2]$. To accomplish this, we create a data structure made up of a primary balanced BST ordered by $y_i$ (as in range search), and associate with each internal node an interval tree of the $x$-coordinate intervals of the line segments in that node’s subtree. The space required for the primary tree is $O(n)$. It is a balanced tree, so each node has $O(\log n)$ ancestors. Each segment has a representation in each of the interval trees corresponding to its ancestors in the primary search tree, and there are $n$ segments, so the space required for the interval trees is $O(n \log n)$. Once we have built this data structure, we can perform queries for a given query rectangle by walking through the primary tree and querying all the interval trees corresponding to nodes between the
top and bottom coordinates $Y_1$ and $Y_2$ of the query rectangle. As with range search, the size of the set $S$ of interval trees queried is $O(\log n)$. For $s \in S$, define $k_s$ to be the number of result segments found in $s$. The time required to query the interval tree $s$ will be $O(\log n + k_s)$. Then the total time required is

$$\sum_{s \in S} O(\log n + k_s) = \sum_{s \in S} O(\log n) + \sum_{s \in S} O(k_s) = O(\log^2 n + k).$$