CS 31: Homework 2

Thomas Monfre

January 24, 2019

Problem 2-1

Exercise 8.3-4: Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

For each of the n integers in the range 0 to $n^3 - 1$, let us convert each to base n. Since we are given n integers in the range $n^3 - 1$, we therefore know each of the n integers can be represented with at most 3 digits in base n. We know this to be the case because $[0, n-1]$ can be represented with 1 digit in base n, $[n, n^2 - 1]$ can be represented with 2 digits in base n, and $[n^2, n^3 - 1]$ can be represented with 3 digits in base n.

Then, for each given number x, in order to convert x to base n, let us perform the following:

The first (rightmost) digit can be computed as $x \mod n$. The second (middle) digit can be computed as $\lfloor \frac{x}{n} \rfloor \mod n$. The third digit (leftmost) can be computed as $\lfloor \frac{x}{n^2} \rfloor \mod n$. Generally speaking, the ith digit (from right to left) can be computed as $\lfloor \frac{x}{n^{(i-1)}} \rfloor \mod n$. Each of these conversions can be computed in constant time. Therefore, each of the n integers can be represented by at most three digits (handling potential leading zeroes) in base n. Since we have a bound of 3 on the number of digits required to represent each value, we therefore can convert each value to base n in constant time.

Lemma 8.3 from CLRS states n d-digit numbers in which each digit can take on up to k possible values can be correctly sorted by Radix-Sort in $\Theta(d(n+k))$ time, given the stable sort it uses takes $\Theta(n+k)$ time.

We have n numbers, each of which is at most 3 digits. Since we converted each of our integers to base n, each digit can take on up to n values.

Using this, let us then perform Radix-Sort on a collection of our n integers. We will use counting sort as our stable sort algorithm as it can sort n elements in the range 0 to n in time $\Theta(n)$ as per Section 8.2 of CLRS.

We therefore have $\Theta(3(n+n)) = \Theta(n)$. Therefore, we can correctly sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

Problem 2-2

Exercise 9.1-1: Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

We can compute the second smallest of n elements with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case by comparing elements in pairs in the style of a single-elimination tournament. For each element, we not only store its value, but also the elements it has been compared against. Then, for each comparison (game played), we advance the smaller of the two elements (the winner) and add the larger of the two elements (the loser) to that element's list of compared values. We continue this operation until one value is left. If at any point there are an odd number of contenders, we compare the first $\lfloor \frac{n}{2} \rfloor$ pairs only. We then advance the last element, but do not add any elements to its list of compared items. This is similar to a bye system in a tournament.

When one value is left after making a series of comparisons, we have the smallest element in the list and all elements it was compared against. Since we advance smaller elements and ignore larger elements, the second smallest element at some point was compared against the smallest. We know this to be the case because for each round of the tournament (level of comparisons), we advance only those elements found to be smallest in their single comparison (game). In a single-elimination tournament, every team loses exactly once, except for the winner. Therefore, at some point the second smallest element must have been compared against the smallest in order to no longer be in contention for being the smallest element. If it had not lost to the smallest element, then the second smallest element would be the winner, not the smallest. Therefore, the second smallest element exists in the list of elements that the smallest value was compared against.

There are $\lceil \lg n \rceil$ levels of comparisons made since each pair contains exactly one winner and one loser. Therefore, the number of contenders left in the game is cut in half on each round (level). Since there are $\lceil \lg n \rceil$ levels, we know the smallest element was compared against exactly $\lceil \lg n \rceil$ elements since it played a game at every level. We can therefore make $\lceil \lg n \rceil - 1$ comparisons to find the smallest element among those compared with the true smallest element. The smallest of these elements is the second smallest element in the collection.

To compute this, we first define a helper function PLAY-GAME that takes two objects a and b that each have a *value* attribute and a *victories* attribute. PLAY-GAME determines a winner among a and b and adds the loser to the winner's array of played games (*victories*).

```
PLAY-GAME(a, b)
1  if a.value < b.value
2       winner = a
3       loser = b
4  else
5       winner = b
6       loser = a
7
8  winner.victories.append(loser)
9  return winner
```

We can now define our recursive function FIND-SECOND-SMALLEST that takes an array A of two or more objects with a *value* attribute and a *victories* attribute. This function recursively performs rounds of the tournament to find a winner. If a winner is found, it iterates through the winner's *victories* array to find the second smallest value, then returns. We constrain A to size two or greater in order to ensure a second smallest element exists to be found.

```
FIND-SECOND-SMALLEST(A)
 1  if A.length == 1  // we found a winner
 2       contenders = A[0].victories
 3       smallest = contenders[0]
 4       for i = 1 to contenders.length − 1
 5            if contenders[i] < smallest
 6                 smallest = contenders[i]
 7       return smallest
 8
 9  else
10       winners = [ ]
11       for each pair (a, b) in A
12            x =Play-Game(a,b)
13            winners.append(x)
14       if A is of odd length
15            add the last element in A to winners
16       return FIND-SECOND-SMALLEST(winners)
```

This algorithm works as follows. At line 1, we check to see if A is of size 1. In this case, we were only given one element. If we are only given one element, we know we have hit the base case of our recursive algorithm and have therefore found a winner. In this case, we revert to the intuition described above: the second smallest element at some point was compared with the smallest element. We then grab the array of values the smallest element was compared with on line 2 and set it to a variable called *contenders*. *contenders* is of length $\lceil \lg n \rceil$ since the winner played in $\lceil \lg n \rceil$ games and therefore was compared with $\lceil \lg n \rceil$ elements. On lines 3-6, we make $\lceil \lg n \rceil - 1$ comparisons to find the smallest element among contenders. We know we make $\lceil \lg n \rceil - 1$ comparisons because we set *smallest* to the first item in *contenders* on line 3, then iterate over the $\lceil \lg n \rceil - 1$ elements left in the list and make comparisons. Therefore, after breaking out of the loop at line 7, *smallest* is the second smallest element in the collection. So, we return that value at line 7.

If A is of size greater than 1, then we still have comparisons to make. We then revert to line 10 where we create an empty array of winners. This array holds the winners of the next round of the tournament. Then on lines 11-13, we call our helper method PLAY-GAME to determine the smaller of each pair. PLAY-GAME handles adding the larger element to the *victories* array of the smaller. Then on lines 14 and 15 we handle any necessary byes by promoting the last element in A as a winner only if A is of odd length. This ensures we don't miss an element. Then, we have our collection of winners and proceed on to the next round by recursively calling FIND-SECOND-SMALLEST at line 16 and returning its returned value.

Let us now prove this algorithm makes at worst $n + \lceil \lg n \rceil - 2$ comparisons. First, let us show that

$(n-1)$ comparisons are made between pairs of elements during the tournament. This can be proved using our intuition of the tournament. In any single-elimination tournament, there are exactly $(n-1)$ games played because every team loses exactly once except for the winner. Taking this to the nature of comparing elements, we know we make $(n-1)$ comparisons in order to find the smallest element since every element is determined larger than some other element than the smallest. Therefore, the process of comparing each pair of elements requires $(n-1)$ comparisons. Then, the only other comparisons made are in determining the second smallest element after hitting our base case. It was proved above that the smallest element is compared with $\lceil \lg n \rceil$ elements. Using the intuition above that finding the smallest element in a collection requires $(n-1)$ comparisons, we then need $\lceil \lg n \rceil - 1$ comparisons in order to find the smallest element among those that were compared with the true smallest. We also handle this by setting *smallest* to the first element in contenders at line 3. We then make $\lceil \lg n \rceil - 1$ comparisons after. After making these comparisons, we return.

Therefore, this algorithm makes at worst $n - 1 + \lceil \lg n \rceil - 1$ comparisons which equals $n - \lceil \lg n \rceil - 2$. Hence, we can find the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Problem 2-3

Exercise 9.3-9: Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Let us first abstract this problem from the oil wells, and consider the mathematical context: we are fitting a line to a set of points. Since the problem specifically states the pipeline runs east to west, we know our line will have a slope of 0. Therefore, our goal is to find an optimal y-intercept for the line, such that the total length of the spurs (i.e. the total distance of each point from the line) is minimized. Let us show that the optimal y-intercept (location) for the line is the median of the y-values of each point.

First, let us consider the case in which we have an odd number of points. Then, the median point is the exact middle value (the value such that half the observations fall to the left, and half the observations fall to the right). Since the y-intercept of our line is the median, we therefore know the line will intersect the median point. Since we have an odd number of points, we therefore know an equal number of points will occur above and below the line. If we were to then move the line up, we then have two cases. In the first, we move the line up, but the line never crosses over a point. Then, the distance of each point above the line to the line would decrease. Since the same number of points exist below the line, however, all saved distance will be lost in the increased distance of each point below the line. This puts us back to the same total distance traveled as when the line crossed through the median value. However, since the line has moved, the median point now has a distance greater than 0 to travel. Therefore, the total distance of each point to the line has increased in moving the line off the median. This construct holds if we were to move the line down as well. In the second case, we move the line up and it passes a point. In this case, we do not have an equal number of points above the line as below. However, since we've crossed a point, all distance saved above the line is lost below, and we have added distance from the median as well as the point we crossed. This generalizes if we were to move the line down as well. Therefore, the location of the line in which total distance is minimized occurs when the median value has a distance of 0 to travel. This occurs when the line intersects the median, indicating total distance travelled is minimized when the y-intercept of the line is the median y-value of all points.

Now, let us consider the case in which we have an even number of points. Then, we compute the median point as any value in between the lower and upper medians, inclusive. Since we have an even number of total points, we therefore know there exists an even number of points above the upper median and an even number of points below the lower median. Therefore, if we move the line to a new y-intercept that is within the range of the lower and upper medians, inclusive, any distance saved above the line will be lost below the line and vice versa (depending on what direction the line was moved). The total distance of the lower median to the line plus the upper median to the line will not change because any distance saved for the lower median will be lost by the upper median and vice versa (depending on what direction the line was moved).

In order to show any value in this range achieves a minimum total distance, let us define the upper median minus the lower median as n. This represents the length of the range in which valid y-intercepts can be chosen. If we were to move the line above the upper median, any distance saved for the points above the upper median would be lost by the identical number of points below the median. Furthermore, the total distance between the lower median and the line and the upper median and the line would be greater than n. We know this because the distance of the lower median to the line must be at least size n since the line is above the upper median. We then also must add the distance of the upper median to the line as well. This argument generalizes to the act of moving the line below the lower median. In this case, the total distance of the points to the line has increased. Therefore, the location of the line in which total distance is minimized occurs when the distance of the lower median to the line plus the distance of the upper median to the line is n. This occurs when the median is any value in between the lower and upper bounds, inclusive.

For simplicity of argument, let us then define the median when we have an even number of values as the lower median. In this case, the line intersects the lower median. The distance saved by the number of points below the lower median is lost by the identical number of points above the upper median. Furthermore, the upper median has to travel a distance of n to get to the line and the lower median has to travel a distance of 0 to get to the line. Therefore, our minimum distance is maintained.

Since we have proven the optimal location for the pipeline occurs at the median y-coordinate of all the points, let us now show this value can be computed in linear time. We can compute the median value in linear time using SELECT from CLRS Chapter 9.3. Specifically, we will call SELECT($\lfloor \frac{n}{2} \rfloor$) in order to get the y-coordinate of the lower median, as discussed above. SELECT finds the desired element in $O(n)$ time by recursively partitioning the input array of points. We therefore will have $\Theta(n)$ for putting all y-coordinates into an array and $O(n)$ for computing the median. $\Theta(n) + O(n) = \Theta(n)$. Therefore, we can compute an optimal location for the main pipeline in linear time.

Problem 2-4

Exercise 14.1-5: Given an element x in an n-node order-statistic tree and a natural number i, how can we determine the ith successor of x in the linear order of the tree in $O(\lg n)$ time?

We can compute the ith successor of x in $O(\lg n)$ time by using OS-Rank and OS-Select from CLRS Chapter 14.1. The algorithm is simple: compute the rank r of x using OS-Rank then make a call to OS-Select to retrieve the $(i + r)$th smallest key in the order-statistic tree T. This will be the ith successor of x.

Get-Successor(x, i)
1 $r =$ OS-Rank(T, x)
2 **return** OS-Select($T.root$, $i + r$)

We can prove this algorithm returns the the ith successor of x in a linear order walk of the tree by examining the following properties. For all nodes y such that OS-Rank$(T, y) >$ OS-Rank(T, x), y is a successor of x. Similarly, for all nodes z such that OS-Rank$(T, z) <$ OS-Rank(T, x), z is **not** a successor of x.

Therefore, the ith successor of x is simply the node of the order-statistic tree T with a rank from the root that is i greater than the rank of x. At line 1 of Get-Successor we compute the rank of x as r. Therefore, the ith successor of x in the order-statistic tree T is the node that is of rank $r + i$.

We know from CLRS Chapter 14.1 that OS-Select($T.root$, j) returns a pointer to the node containing the jth smallest key in the order-statistic tree T. Therefore, our call to OS-Select at line 2 returns the node of rank $i + r$ in the order-statistic tree T. Since we have shown the ith successor of x is the node of rank $i + r$, we have proven Get-Successor returns the ith successor of x in the linear order of the tree.

We will now prove this algorithm runs in $O(\lg n)$ time. The analysis here is simple. We know from CLRS Chapter 14.1 that both OS-Rank and OS-Select run in time $O(\lg n)$ for an n-node order-statistic tree. Therefore, we have $O(\lg n) + O(\lg n) = O(\lg n)$.

Problem 2-5

Exercise 14.1-6: Observe that whenever we reference the *size* attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank, Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

We will first discuss insertion. As noted in CLRS Chapter 14, insertion in a red-black tree consists of two phases. The first goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and performing up to two rotations.

In the first phase, as we traverse down the tree, let us increment the *rank* of a node if we take its left path. Essentially, on each node we visit, if it is a left child, increment the *rank* of the parent by 1. We perform this action because traversing down a path indicates we will be inserting a node down that path. Going left indicates the left part of the subtree rooted at the parent is increasing in size. Therefore, the *rank* of the parent must increase. We **do not** increment the parent if we travel down its right child, because the rank of the parent will not be affected by inserting our node.

In the second phase, as we perform at most two rotations, only one node has its *rank* attribute invalidated on a given rotation. On a left rotation, this is the parent node that is incident on the link to which we are rotating. On a right rotation, this is the child node that is incident on the link to which we are rotating. We will define this node as y and the node it is incident to on the rotated link as x. This naming convention follows that of CLRS Chapter 14.1 (see the top of page 344 in the third edition for clarification on naming).

In order to correct the *rank* attribute on rotation, we simply change $y.rank$ in reference to $x.rank$. If we are performing a left rotation, we set: $y.rank = y.rank + x.rank$. Essentially we take y's rank before rotation and add it to x's rank. If we are performing a right rotation, we set: $y.rank = y.rank - x.rank$. This handles the changing ranks from rotations. If we've rotated left, $y.rank$ needs to increase by $x.rank$ because we've added $x.rank$ nodes to the left of y. If we've rotated right, $y.rank$ needs to decrease by $x.rank$ because we've removed $x.rank$ nodes to the left of y. Therefore, these two steps maintain the *rank* attribute on insertion.

We will now discuss deletion. As noted in CLRS Chapter 14, deletion in a red-black tree also consists of two phases. The first removes a node, then traverses up the tree, moving up to two other nodes. The second phase performs at most three rotations.

In the first phase, let us traverse a simple path from the original position of the lowest node that moves up to the root. On each node we visit, if it is a left-child, decrement the *rank* attribute of its parent by 1. The logic is similar to that of above. If we are at a left-child, we've removed a node from the subtree rooted at that node's parent. Therefore, its rank will decrease by 1. We do not decrement the parent if we are at its right child, because the parent will not have been affected by the deletion. During this phase, we may move up to two other nodes. On each move, we propogate our changes to *rank* up the tree, decrementing a parent if we are at its left child.

In the second phase, we handle the maximum three rotations similar to that above. We use the same naming conventions of y and x. On a left rotation we set: $y.rank = y.rank + x.rank$, and on

a right rotation we set: $y.rank = y.rank - x.rank$. The reason this corrects the ranks is the same as above. On a left rotation, $y.rank$ needs to increase by $x.rank$ because we've added $x.rank$ nodes to the left of y. On a right rotation, $y.rank$ needs to decrease by $x.rank$ because we've removed $x.rank$ nodes to the left of y. These steps handle adjusting rank on deletion.

Let us now observe that all of these operations take at most $O(\lg n)$ time. When incrementing or decrementing ranks, we perform one action for each level of the tree we visit. Since our underlying data structure is a red-black tree, we know its height is at most $\lg n$. Therefore, incrementing and decrementing ranks takes time $O(\lg n)$. When handling rotations, updating $y.rank$ takes constant time. Since we have perform at most two rotations on insertion and at most three rotations on deletion, handling rotations takes time $O(1)$.

Therefore, we can maintain the *rank* attribute of a tree during insertion and deletion in time $O(\lg n)$.

Problem 2-6

Exercise 14.1-7: Show how to use an order-statistic tree to count the number of inversions (see CLRS Problem 2-4) in an array of size n in time $O(n \lg n)$.

For an array A of size n, the pair of indices (i, j) is an inversion of A if $i < j$ and $A[i] > A[j]$. An order-statistic tree is an augmented red-black tree, that holds a *size* attribute of each node and supports the operations OS-SELECT and OS-RANK.

In order to determine the number of inversions in an array A of size n, let us iteratively construct an order-statistic tree T and sum up the number of inversions that a given node is a member of. Then, for each index j in the list, add node x with key $A[j]$ to T. This can be performed in time $O(\lg n)$ as described in CLRS Chapters 13 and 14.

Since we are iteratively adding nodes to the tree, we then know there exist $j+1$ nodes in the tree on each iteration. We have $j+1$ nodes because our indices start from 0. In other words, $j+1$ elements exist at or before j in A. Since an order-statistic tree orders elements by key, we then know the number of elements to the right of x in T create inversions since they all exist at a smaller index in A (hence their presence in the tree) and they all have greater values since they are positioned to the right of x in T. Since we have an order-statistic tree, we can compute the number of these inversions for x with the following logic. There are $j+1$ elements in T and OS-RANK(T, x) returns the ordered position of x in T. Therefore, the total number of inversions that j contributes towards is the value: $(j+1) -$ OS-RANK(T, x).

Therefore, by summing up all inversions that each element contributes towards, we will have counted the total number of inversions in A.

```
COMPUTE-INVERSIONS(A)
1  T = new OrderStatisticTree()
2  total = 0
3  for i = 0 to n
4        z = new node with key A[i]
5        RB-INSERT(T,z)
6        sum = (i + 1) − OS-RANK(T, z)
7        total = total + sum
8  return total
```

Let us now show this algorithm runs in time $O(\lg n)$. The call on line 1 to create an order-statistic tree T takes constant time since the tree is empty. The for loop on line 3 contributes $\Theta(n)$ to the run-time since we are performing a set of actions on all n items. Line 4 runs in constant time since we are simply constructing a singular object. The call to RB-INSERT on line 5 runs in time $O(\lg n)$ as per CLRS Chapter 13. The call to OS-RANK on line 6 also runs in time $O(\lg n)$ as per CLRS Chapter 14. Lines 7 and 8 run in constant time.

Therefore, for n items in the array A we perform $O(1) + O(\lg n) + O(\lg n) + O(1)$ operations. We therefore have $O(n \lg n)$. Hence, we can count the number of inversions in an array of size n in time $O(n \lg n)$ using an order-statistic tree.

Problem 2-7

Problem 14-2, part b only: We define the ***Josephus problem*** as follows. Suppose that n people form a circle and that we are given a positive integer $m \leq n$. Beginning with a designated first person, we proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. The order in which the people are removed from the circle defines the ***(n,m)-Josephus permutation*** of the integers $1, 2, ..., n$. For example, the (7,3)-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

Suppose that m is not a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n,m)-Josephus permutation.

In order to compute the (n,m)-Josephus permutation of an array A of objects with an associated value in $O(n \lg n)$ time, let us use an order-statistic tree.

```
GET-JOSEPHUS-PERMUTATION(A, m)
 1  n = A.length
 2  T = new OrderStatisticTree()
 3  for j = 1 to n
 4       z = new node with attributes key = j and value = A[j]
 5       RB-INSERT(T,z)
 6
 7  output = [ ]
 8  prev_rank = 1
 9  num_values_in_tree = n
10
11  while num_values_in_tree > 1
12       rank = ((m − 2 + prev_rank)  mod num_values_in_tree) + 1
13       x = OS-SELECT(T.root, rank)
14       add x.value to output
15       RB-DELETE(T, x)
16       num_values_in_tree = num_values_in_tree − 1
17       prev_rank = rank  mod num_values_in_tree
18
19  add OS-SELECT(T.root, 1).value to output
20  return output
```

This algorithm first constructs an order-statistic tree T by iterating over each value in the array A and adding a node with that value to the tree. The key of each node in the tree is its position in A. We store the key as the index in order to maintain the same order in the tree as we found in the array. If we based our keys on the value in the array, unless the values in the array are non-decreasing, our tree would not be structured with the same order as the array. We then hold onto an attribute *value* that holds the value of each element in the array. *value* will be used when determining the order in which the elements are removed. The process of constructing the tree takes time $\Theta(n \lg n)$ since we call RB-INSERT at line 5, which takes $O(\lg n)$, for each of our n values.

After constructing the tree, we then define a series of helper variables used to determine the order in which items should be deleted. *output* is an array that will hold values in A in the order in which they are deleted. It is initialized to an empty array since we haven't yet removed any values from T. $prev\_rank$ is the rank of the last element we deleted in the tree. This is helpful for calculating the next element to delete from the tree. On each deletion, we advance one rank in the tree since there is one less element present. Therefore, we initialize $prev\_rank$ to 1. $num\_values\_in\_tree$ is the number of nodes in the tree. It is initialized to n since we added n nodes to T in lines 3-5. Each of these operations takes constant time.

We then construct a while loop that iterates until $num\_values\_in\_tree > 1$. On each iteration, we compute the rank of the element to delete. We then grab that element, store its value in *output*, remove it from the tree, then update our helper variables.

Let us first describe the process of computing the rank of the element to delete. We can prove that $((m - 2 + prev\_rank) \mod num\_values\_in\_tree) + 1$ is the rank of the correct item to delete with the following logic. $prev\_rank$ always holds the rank in T of the element x we last deleted from the tree. We therefore need to move m spaces ahead in order to find the next element to delete. This yields rank $m + prev\_rank$. Since we previously deleted x, however, the rank of each element located to the right of x has decreased by 1. In other words, the deletion caused us to advance 1 position since we have taken the same rank, but all elements to the right of x have shifted over. So, we subtract 1 from $(m + prev\_rank)$ to handle this. This value moves us ahead the desired number of positions. However, it does not account for the possibility that we looped around the array and have started at the beginning. So, we take the modulo of this number by $num\_values\_in\_tree$ in order to allow for the circular structure of deletions.

This seems mostly ideal, however, for some number q, the range of possible values of any number mod q is $[0, (q-1)]$. This means the range of possible ranks we've calculated is $[0, num\_values\_in\_tree - 1]$. Since no element in the tree is of rank 0 and there exists some element in the tree of rank $num\_values\_in\_tree$ that we are not considering, we must shift the range of ranks up by 1. To do this, we subtract another value from $(m - 1 + prev\_rank)$ to get $(m - 2 + prev\_rank)$. This means that any value that previously would have computed rank 0, is now rank $num\_values\_in\_tree - 1$ and any value that previously would have computed rank $num\_values\_in\_tree - 1$ is now rank $num\_values\_in\_tree - 2$. This has solved our problem of computing 0 instead of $num\_values\_in\_tree$. Now all we have to do is shift this distribution of values up by 1. Therefore, after computing the mod, we add 1. This yields $((m - 2 + prev\_rank) \mod num\_values\_in\_tree) + 1$.

Therefore, we can prove $((m - 2 + prev\_rank) \mod num\_values\_in\_tree) + 1$ is the rank of the correct item to delete by observing that we need to advance m ranks from $prev\_rank$, but must subtract 1 to handle the step we took on deleting the previous element. We take the mod to create a circular structure of ranks, then subtract 1 within the mod and add 1 outside of it to shift our distribution to the range $[1, num\_values\_in\_tree]$ which is the range of valid ranks in the tree. Therefore, on each iteration of the loop, $((m - 2 + prev\_rank) \mod num\_values\_in\_tree) + 1$ is the rank of the correct item to delete. This calculation takes constant time.

After computing $rank$, we then grab a reference to the item in T at that rank using OS-Select. This call takes time $O(\lg n)$. Once we have x, we then grab its value and add that value to *output* on line 14 in constant time. Since we have now grabbed the value we were intending on removing,

we need to remove that value from the tree. We therefore call RB-DELETE at line 15 to remove x from T. This call takes time $O(\lg n)$. Next we update $num\_values\_in\_tree$ to accurately reflect how many items we have taken out of the tree. Essentially, we decrement it by 1 since we removed an element from the tree. Lastly, we update $prev\_rank$ to be the rank in the new tree of the item we removed from the last tree. To do this we take $rank \mod num\_values\_in\_tree$. Since we've decremented $num\_values\_in\_tree$, we are therefore finding the equivalent of $rank$ in the new tree. So, if we had a tree of 5 elements and removed the element at rank 5, we would be at rank 1 since $5 \mod 4 = 1$. If we had a tree of 4 elements and removed the element at rank 3, we would be at rank 0 since $3 \mod 3 = 0$. This outcome is handled by the computation of $rank$ above.

We then repeat these steps until only one value is left in T. At that point we break out of the loop, add the singular value in the tree to *output* and return.

We can prove this loop only operates on $n - 1$ values using the following loop invariant: At the start of each iteration of the while loop, $num\_values\_in\_tree$ is the number of nodes in the order-statistic tree T. We seek to perform the actions of the while loop only when the size of T is greater than 1.

Initialization: Prior to the first iteration, line 9 sets $num\_values\_in\_tree$ to n. At this point, T contains n elements since we added n nodes to T at lines 3-5. Therefore, we will enter the loop so long as $n > 1$. In this case, there is more than one value in the tree, hence our desire to enter the loop.

Maintenance: At the end of each iteration of the while loop, we decrement $num\_values\_in\_tree$ by 1 since our call to RB-DELETE at line 15 removed a single node from the tree.

Termination: The loop terminates when $num\_values\_in\_tree = 1$. In this case, there is only one node in the tree. This indicates we do not need to calculate the rank or perform any deletions since there only exists one element to remove. We therefore break from the loop and grab that value.

Therefore, this loop only operates on $n-1$ values. We know it removes the correct element on each iteration since we proved above that $((m-2+prev\_rank) \mod num\_values\_in\_tree)+1$ is the rank of the correct item to delete from the tree. Therefore, this algorithm correctly computes the Josephus-permutation.

The run-time of this algorithm can be computed as follows: $\Theta(n \lg n)$ for constructing the tree + $O(1)$ for defining helper variables + $\Theta((n-1)\lg n)$ for the while loop + $\Theta(\lg n)$ for the final call to OS-SELECT.

We therefore have $\Theta(n \lg n) + \Theta(1) + \Theta((n-1)\lg n) + \Theta(\lg n) = \Theta(n \lg n) + \Theta(1) + \Theta(n \lg n) = \Theta(n \lg n)$. Hence, we can compute the Josephus-permutation of an array A in time $\Theta(n \lg n)$.

Who I Worked With

Problem 2-2: Kunal Rathi and I worked together on this problem. We collaborated on determining how to hold onto only $\lg n$ contenders for the second smallest item.

Problem 2-3: I worked with Kunal Rathi on coming up with a proof for the median when there are an even number of points.

Problem 2-6: Emma Rafkin and I worked together on this problem. We worked on determining the proper number of subtractions to make in the tree.

Problem 2-7: Emma Rafkin, Kunal Rathi, and I worked together on this problem. We all worked on constructing the proper algorithm and proving its correctness.