Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld

Thursday, October 20, 2016
Problem Set 3

Problem Set 3

All parts are due Tuesday, November 1, 2016 at 11:59PM.

Name: Milo Knowles

Collaborators: Magnus Johnson, Ayush Sharma, Gillian Belton

Part A

Problem 3-1.

(a) To sort the buildings by number, I would use Radix Sort. From Lecture 7 on linear time sorting, we saw that Radix sort takes $O(nd)$ time where n is the number of integer elements to sort, and d is the maximum number of digits in the integers. This can be applied to sorting the buildings at MIT because each building will have a number of d digits.

(b) Because we now have an array of the buildings sorted by their number, we can run a binary search on the array to find a particular building. To find a building with a d digit number in an array of n sorted buildings will take $O(nd)$ time, because a binary search requires $O(lgn)$ comparisons, and each comparison takes $O(d)$ time. An artery can have, at most, 10 buildings, because there are 10 discrete locations along each axis (positions 0-9). To traverse an artery, we simply run a binary search on each potential building address in the artery, starting at 0 and ending at 9. This will take 10 $O(nd)$ searches to complete, which is overall and $O(nd)$ algorithm.

Problem 3-2.

(a)

- Given two sorted arrays, L_1 and L_2, each of size $\frac{n}{2}$, we can find the kth largest element by alternating binary searches on the two sorted arrays.
- First, check whether one array is bigger than all of the elements in the other. If $L_1[0] > L_2[\frac{n}{2}]$, then all of the elements in L_1 are greater than L_2, so we can just return the kth largest element in L_1, which is $L_1[\frac{n}{2} - k]$. If $L_2[0] > L_1[\frac{n}{2}]$, then return $L_2[\frac{n}{2} - k]$.
- In general, we are trying to reduce our search space (which begins with size n) to size k, because in this case we can just return the kth element.

- We start by looking at the middle element of L_1, which is at index $i_1 = \frac{n}{4}$. We run a binary search on L_2 to find the index i_2 where this element would fit into L_2 in $\theta(\log_2 \frac{n}{2})$ time. The expression $s = n - i_1 - i_2$ gives us the current size of our search space. We must consider 3 cases:
  - $s = k \pm 1$. This means that we have found our kth largest element. If $s = k$, then all we need to do is compare $L_1[i_1]$ and $L_2[i_2]$ and take the minimum of the two. If $s = k + 1$ then we need to take the second smallest element amongst $L_1[i_1]$, $L_1[i_1 + 1]$, $L_2[i_2]$, and $L_2[i_2 + 1]$. If $s = k - 1$ then we return $min(L_1[i_1 - 1], L_2[i_2 - 1])$.
  - $s > k$. This means that our kth largest element has to be greater than both $L_1[i_1]$ and $L_2[i_2]$. Therefore, it cannot be in the lower half of our search space in L_1. So we can narrow down our search space in L_1 to $L_1[i_1 : \frac{n}{2}]$.
  - $s < k$. This means that our kth largest element is not greater than $L_1[i_1]$ or $L_2[i_2]$. Therefore, it cannot be in the upper half of our search space in L_1. So we can narrow down our search space in L_1 to $L_1[0 : i_1]$.
- In any of these cases, we have either returned the answer or reduced the size of L_1 by a factor of 2. Now, we repeat the above process on L_2: letting i_2 be the midpoint of i_2, finding the index i_1 in L_1 where $L_2[i_2]$, and considering the three cases above. As we keep running through this process, we are alternating between halving L_1 and L_2. Eventually, we will hit the case where $s = k \pm 1$, and return the kth element. It will take $O(\log_2 n)$ iterations to reduce the search size of L_1 and L_2 down until we have $s = k \pm 1$. Each iteration requires an $O(\log_2 n)$ binary search in the opposing list. Therefore, the run time is $O(\log^2 n)$.

(b)

- Augment the data structure for a node by adding a property that stores the size of the subtree rooted at that node. When we call AVL-Insert, we add a new node as a leaf. We can move up the AVL-Tree on a path from leaf to root by recursively moving to the parent of each node. At each node, we update the subtree size as:

  node.size = Lchild.size + Rchild.size

- One small caveat is that when we add a new leaf, we say that is has size 1, so that its parent size will increase by 1.
- Because this is an AVL-Tree, the maximum length of a path from leaf to root is $O(\log_2 n)$. Therefore, recursively moving up the tree to update weights will take $O(\log_2 n)$ time.

(c)

- Once again, we have a search space of size n that we are trying to reduce to size k so that we can easily find the kth largest element. Let the two AVL-Trees be called T_1 and T_2, and assume that they were built using modified AVL-insert such that each node stores the size of the subtree rooted at it.

- First, we look at the root node of T_1. We can find where it ranks in the order of items in the tree in $O(1)$ time through the formula:

  R1=root.Rchild.size + 1

  This formula will always hold because all nodes to the right of a node in an AVL-Tree are greater than it.
- Then, we run an AVL-search for root.key in T_2, which takes $\theta(\log_2 n)$ time. Although keys in both AVL-Trees are unique, AVL-search will still return the place where root.key should be in T_2. We also want to know how many elements are larger than root.key in T_2. We can determine this during the AVL-search for root.key by keeping track of all of the nodes at which we go left. Whenever we go left during this search, this means that our element is smaller than current node and all of the nodes in the right subtree of the current node. Therefore, during our AVL-search we update the rank of our search element whenever we go left by the formula:

  R2 += node.Rchild.size + 1

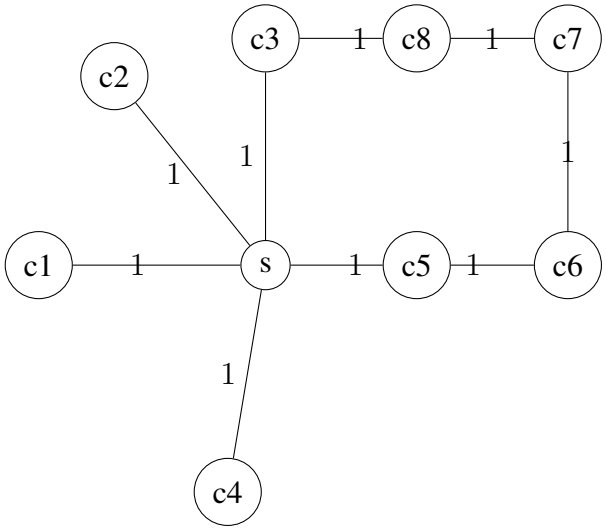
- Our AVL-search still takes $O(\log_2 n)$ time. By adding up the number of nodes $R = R_1 + R_2$ greater than root.key, we can compare this to k and determine how to narrow down our search space in T_1, identical to the method used in Problem 2a.
- If $R > k$, we can reduce T_1 to the subtree rooted at root.Rchild by setting root.Rchild to the new root node.
- If $R < k$, we can reduce T_1 to the subtree rooted at root.Lchild by setting root.Lchild to the new root node.
- If $R = k \pm 1$, then we can solve for the kth largest value. If $R = k$, then we return the value at the root of T_1. If $R = k + 1$, then we find the successor of root node in T_1 and in T_2, and return the minimum of the two. If $R = k - 1$, then we find the predecessor of the root node in T_1 and T_2, and return the maximum of the two.
- Just like in Problem 2a, we alternate between T_1 and T_2, reducing the size of the tree by a factor of 2 with every iteration of the algorithm. Therefore, we are guaranteed to reach the case $R = k \pm 1$ after $O(\log_2 n)$ iterations. Each iteration requires an $O(\log_2 n)$ AVL-search, so our runtime is $O(\log^2 n)$.

Problem 3-3.

(a)
- Our fields connected by dirt paths of equal length is analogous to an unweighted, undirected graph.

- Run a breadth-first-search on this graph in $O(V + E)$ time. Because our graph is unweighted and BFS considers every possible path to the search frontier as it propagates, we know the shortest path distance to a chicken as soon as it is found.
- We also know the vertex (field) that a chicken is located in. Whenever a chicken is found during BFS, we append our (c, v, d) tuple to the list L of chickens in $O(1)$ time.
- Because there are m chickens, it takes $\theta(m)$ time to append all chickens to list L.
- Therefore, the overall runtime is $O(V + E + m)$.

(b) This is False. Consider the following counterexample:



The 5 closest nodes in this case are c_1, c_2, c_3, c_4, c_5, because they are all distance 1 away from s. A path that visits all of these nodes and returns to s would have length 10. However, it we create a path $s, c_5, c_6, c_7, c_8, c_3, s$, we can visit 5 chickens with a path of length 6.

(c) We have a directed, acyclic graph. Therefore, we can find a topological sorting of the graph in $O(V + E)$ time, using the depth-first-search method. Now, if we visit the fields in order of our topological sort (using our helicopter to travel in $O(1)$ time, we will eventually have visited every field and pushed all of the chickens to our fenced area at s.

Correctness:

- By visiting fields in the order of our topological sort, each new field we visit will be downhill from the ones we visited previously. Therefore, we will always be "pushing" the chickens downhill. This also guarantees that we will not arrive at a field downhill from a chicken and cut off its path to s.
- Because chickens cannot go to fields we have visited, and we visit every field, no chickens can be left in the fields. Because there is always a downhill path that a

chicken can follow to s and we never visit a field downhill from a chicken, this means that the only place chickens can end up at is s.

(d) Find a strictly downhill path to s with a chicken at some node on that path.Visit the node that is adjacent and downhill from the chicken, then visit the node with the chicken. The chicken will either be forced to travel uphill to get away from you, or move downhill to a node that you have already visited. In either case, the behavioral rules of the chicken are violated.

Part B

Problem 3-4.

(a) *Submit your implementation on alg.csail.mit.edu*

(b) *Submit your implementation on alg.csail.mit.edu*

(c)

- Given a pattern of length k that we would like to search, consider all of the ways the pattern could appear in the corrupted document. Each of the k digits could be replaced with a ?, giving 2^k possible variations. Store all of all the hash values of these 2^k variations in a list in $\theta(2^k)$ time. Note the ? should be treated as a distinct character with a value mapped to it in our hash function.
- The CorruptedSearch algorithm extends the rolling hash search function we implemented in Problem 4b. However, whenever a ? is added on to the rolling hash, the algorithm recognizes that the rolling hash will contain a ? for the next k roll forwards. For the next k roll forwards, the CorruptedSearch algorithm will compare all 2^k possible pattern hash values to the hash value of the rolling hash.
- We know that the runtime of a rolling hash search is $O(n)$, because we must roll the hash forward $n - k$ times. However, in the case that our rolling hash window contains a ?, we will have to check all 2^k hash variations of our pattern. This means that the CorruptedSearch runs in $O(2^k + n)$ time.
- If $m = \theta(n)$, then the load factor is $\frac{n}{m}$ which is $O(1)$. We can expect to have a small number of collisions between with out hashing function in this case, meaning that occasionally the algorithm will run into a substring of the document with the same hash value as our search, but is not a valid match. For each collision, it takes $O(k)$ time to compare our k length search to a k length substring in the document, so resolving collisions takes $O(k)$ time. Because we only expect to have $O(1)$ collisions, having an $m = \theta(n)$ will cost us $O(k)$ time, and therefore the algorithm's runtime is still $O(2^k + n)$.
- This algorithm is guaranteed to find a match in the document (if one exists), because it considers every possible way the search string could appear in the document. Therefore, no matter how the document is corrupted, the search will

still work. Whenever the rolling hash finds a potential match, the match is verified by checking every character.

- For patterns of size k that are relatively small to the size n of the document, the $O(2^k + n)$ runtime of CorruptedSearch will be faster than the $O(nk)$ runtime of the naive search. This is because 2^k will be small compared to nk. For example, searching for a 4 character pattern in a 100 character document will take $O(16 + 100) = O(116)$ in with CorruptedSearch, but will take $O(4 * 100) = O(400)$ with the naive method.