Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 20, 2016
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld — Problem Set 3

Problem Set 3

All parts are due on November 1, 2016 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 3-1. [20 points] **The Future Campus of MIT**

The year is 26100, and humans have achieved (1) individual flight and (2) access to unlimited dimensions of space. MIT buildings now form a grid in d-dimensional space, and are numbered with d-digit decimal numbers, with each digit representing location along one dimension. There could be at most one building with a given number, but not all numbers are occupied by buildings.

An i-***artery*** is a maximal set of buildings that differ in the ith digit, but share all $d - 1$ other digits. In other words, an artery is the set of buildings lying on a line parallel to a coordinate axis. For example, building 25628 is between building 25627 and building 25629 along a 0-artery (where digit 0 is the "ones" digit) in 5-dimensional MIT.

The MIT students of the future are still smart and refer to the buildings by their numbers, but for some reason the building directory given to you has the buildings sorted by name (an irrelevant feature of buildings).

(a) [10 points] Give an efficient algorithm to sort the buildings by number. The algorithm should run in $O(nd)$ time for n buildings.

Solution: Radix sort the building by their building numbers, using counting sort to sort each digit. For n buildings and 10 values of digits, counting sort takes $O(n)$ time, and this must occur once for each digit, for $O(nd)$ time total. We can use radix sort on the building numbers because they are integers.

(b) [10 points] You want to traverse a particular i-artery of MIT in its entirety, given by coordinates $x_0, x_1, \ldots, x_{i-1}, x_{i+1}, x_{i+2}, \ldots, x_{d-1}$, in increasing order by ith digit. Given the sorted-by-building directory from part (a), give an $O(d \log n)$-time algorithm to efficiently extract an in-order list of buildings you will pass through.

Solution: Binary search the in-order list of buildings for each of the 10 possible building numbers along on i-artery. There's a constant number of these numbers, and binary search on these numbers takes $O(d \log n)$ time; $O(d)$ is the time necessary to make each comparison, since each comparison involves going through the d digits of the number and comparing against the number from the given i-artery digit-by-digit.

Problem 3-2. [25 points] **Finding a Place in the Search Engine Space!**

Elexa, the internet ranking site, has gotten word of the up-and-coming 6006LE, so they decided to publish a new dataset: the amount of internet traffic received by each of the top billion sites in the past year.

Unfortunately (or fortunately), Estoy, the Elexa engineer responsible for the data, recently fell in love with binary search trees, after encountering them in 6.006. He randomly partitions the data into two disjoint sets, and stores each set in a BST. In each BST, a node's key stores the amount of traffic for one site.

As an algorithmically efficient traffic analyst, your goal is to be able to quickly find the kth largest site among these two binary search trees.

(a) [10 points] To start, imagine Estoy gave you two sorted arrays, instead of two BSTs. Give an $O(\log^2 n)$-time algorithm to determine the kth largest number among these two sorted arrays. Justify your time complexity.

Solution: Denote the two arrays A and B. Use binary search to find the zero-indexed location i in array A where $b_m = B[\lfloor \frac{n}{2} \rfloor]$ would be inserted to preserve A sorted array. By construction, $k_t = i + \lfloor \frac{n}{2} \rfloor$ elements are smaller than b_m. If $k = k_t + 1$, then the k-th rank element is b_m. Otherwise, if $k > k_t + 1$, the k-th rank element must be in the right half of array B or the right part of A, right of index i: update $k \to k - k_t$ and recurse with two smaller arrays $A[i : n]$ and $B[\lfloor \frac{n}{2} \rfloor : n]$. For the final case, if $k < k_t + 1$, the k-th rank element must be on the left part of the arrays, so recurse on subproblems $A[0 : i]$ and $B[0 : \lfloor \frac{n}{2} \rfloor]$.

Every recursive step we perform an $O(\lg n)$ binary search in A, and the size of B halves. When $|B| = 1$, the $k{-}th$ rank element is the k-th element in the sorted array A, after the one element in B has been inserted into A. We terminate when $|B| = 1$. Since $|B|$ halves, the divide-and-conquer recurrence becomes $T(n) = T(n/2) + O(\lg n)$. This solves to $O(\log^2 n)$.

(b) [5 points] Next, describe how to modify the AVL-insert algorithm to maintain, at each node x, the number of nodes in the subtree rooted at x. (And suppose that Elexa implements these modifications.)

Solution: When traversing downward from the root during insertion, increment the sub-tree size count of each node (in the path from the root to the inserted location). If rotations are necessary, we only need to adjust a constant number of sub-tree size counts. We consider a left-rotate and right-rotate seperately. Using the notation and diagram from page six of Lecture 6 notes, during a left rotate, the size count stored at x changes from $|A| + |B| + |C| + 2$ to $|A| + |B| + 1$. The size count stored at y changes from $|B| + |C| + 1$ to $|A| + |B| + |C| + 2$. During a right rotate, the subtree-size of z changes from $|B| + |C| + |D| + 2$ to $|C| + |D| + 1$. The size count of y changes from

$|B| + |C| + 1$ to $|B| + |C| + |D| + 2$.

The time complexity of AVL-insert remains $O(\lg n)$, because every rotation updates a constant number of sizes, and updating sizes along the path from the root to the insertion location is $O(\lg n)$

(c) [10 points] Finally, give an $O(\log^2 n)$-time algorithm to find the kth-largest data point across these two size-augmented AVL trees.

Solution: We adapt our solution from part (a). Instead of querying the middle element of array B in array A, find the insertion location of the root of B (r_b) in tree A. Everything in r_b's left sub-tree must be smaller than r_b. Denote the size of this subtree as p. To determine the number of elements smaller than r_b in tree A (q), we keep a running count as we do our downward traversal during insert: if insertion chooses the right sub-tree to traverse, add the size of the left subtree to our running count q. The total number of elements smaller than r_b across the two trees is $p + q$.

Like before, we break into different cases based on how k compares $p+q$. If $k > p+q$, we know the k-th largest element is not in the r_b's left subtree of size m, so we remove this left sub-tree. After removing the left sub-tree of B, reinsert the the root into the right subtree, and repeat. The two other two cases ($k = p + q$ and $k < p + q$) are analogous to that in part (a), and uses analogous operations to the sub-tree removal just described.

The size of the tree B decreases by half every iteration, and we do a $O(\lg n)$ psuedo-insertion into A every operation. The recurrence for this is $T(N) = T(\frac{n}{2}) + O(\lg N)$, which solves to the desired runtime.

Problem 3-3. [25 points] **Chicken Farming**

Following the success of 6006LE, you decide to follow your dreams and open a carbon-neutral, cruelty-free, cage-free, fence-free chicken farm. You order m chickens to be delivered to your house in the hilly countryside, and go to work (to solve Problem 3-2 on your recent 6.006 problem set). The problem takes longer than you expected, and by the time you get back home that evening the chickens should have already been delivered ... but they're nowhere to be seen!

As you realize that a "fence-free chicken farm" may have been a bad idea, you wander up to a nearby hill and realize where all the chickens have gone. The m chickens (lovingly named $c_1, c_2, \ldots, c_m$) are all roosting in the $|V|$ fields near your house, given by the set V. There are dirt paths between some pairs of the fields, given by the set E, and every dirt path is the same length. There can be multiple chickens (or none) nesting in a single field $v \in V$. You are at the starting field $s \in V$ which you know is connected to every other field, and you can't see from here where each chicken's settled... so it looks like you'll have to go searching.

(a) [10 points] You decide you'd like to map out where the chickens have settled, and how far you need to walk to collect the eggs from each chicken's nest. Assume you have access to a helicopter and can move between fields you've previously visited (and know the location of) in $O(1)$. Give an algorithm for constructing a list of entries $L = \{(c, v, d) \mid c \in c_1, c_2, \ldots, c_m\}$ denoting that chicken c is located at field v, and that the shortest path from s to v is of length d. Your algorithm should run in $O(V+E+m)$ time.

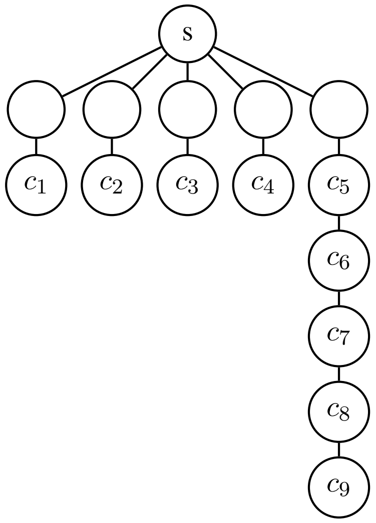
Solution: We can treat the network of fields as a graph, where each field is a vertex and each path between fields is an edge. We know that all the fields are connected and the edges undirected, and we're looking for the single-source shortest path to all the nodes from the "start field" node in an unweighted graph. This seems like a job for Breath-First Search!
Run a BFS starting at field s, and your constructed chicken directory $L = \{\}$. For each node encountered in the algorithm, store a "distance" field, which for s is 0. Each time you visit a node, its distance is set to the distance of its parents plus 1. When you visit a node v, for each chicken c_i you find in that node, add an entry to L corresponding to $(c_i, v, \text{distance})$. When your BFS queue is empty (and you have therefore finished visiting all the nodes), return L.

Running the BFS takes $O(|V| + |E|)$, and for each of the m chickens we add an entry to our list exactly once, so the total runtime is $O(|V| + |E| + m)$.

(b) [5 points] Now that you've catalogued every chicken, you'd like to start farming by collecting eggs from 5 different chickens. You can assume that you can carry infinitely many eggs at the same time. Define a "k-collection path" to be a path starting from s, passing through some number of fields, collecting eggs from the k chickens you pass, and finally returning to s. You pick 5 chickens from your catalogue that have the shortest distance d, and you claim that the shortest possible 5-collection path contains these 5 chickens. Are you correct? If yes, provide a proof. If not, provide a counterexample.

Solution: This claim is incorrect. The nodes which are the shortest distance to the origin are by no means always the nodes that are the shortest distance to each other.

For example, imagine the following graph:



The closest 5 nodes to s according to the directory L from part (a) are c_1, c_2, c_3, c_4, c_5. However, the 5-collection path going through these nodes requires us to go to each chicken (2 path-lengths) and then go back (also 2 path-lengths). In total, the length of this path is 20. On the other hand, the 5-collection path through c_5, c_6, c_7, c_8, c_9 requires just 12 path-lengths (equivalent to the distance to c_9 and back. This is a clear counterexample to the original statement.

(c) [10 points] After a while, you get tired of getting up every morning to wander through the fields gathering eggs. You decide to give up on your dreams and finally build a fenced enclosure for the chickens in field s. Now you need to bring the chickens back to the enclosure, but every time you go to a field with chickens in it, all the chickens run away from you. Luckily:

- Chickens run only on the existing dirt paths E between the fields.
- Every path $e \in E$ is sloped, and the chickens only run downhill on each path (meaning that every path is unidirectional for chickens).
- Field s is reachable via some directed (chicken-traversable) path from every field.
- No chicken will run to any field that you have visited before (as you leave your scent behind).
- You have your helicopter again, so assume that you instantaneously transport yourself between any two fields.

Give an algorithm to compute a sequence of fields in V that you should visit to get all the chickens to run into your fenced enclosure s. Your algorithm should run in $O(V + E)$ time.

(Note: there are no cycles in the chickens' potential paths because, as in reality, there are no downhill cycles.)

Solution: We can now model our network of fields as a DAG (directed acyclic graph) because each of our edges is directed (the paths are sloped) and we have no cycles (because there is no such thing as a "downhill cycle" - if you walk in a circle and end up where you started, you must always have gone downhill as often as uphill). We also know that field s is reachable by a chicken-traversable path from every field.

We must now consider how to move chickens from their current locations to the remaining fields. The danger is that we may visit a field too early, which would restrict chickens from running to that field, and possibly cut off that chicken's only path to s. We will plan a visitation pattern that will ensure this doesn't happen. On a high-level, we want to iteratively go to the farthest fields from s (whether they contain chickens or not) and successively push chickens out of the farthest fields. If we do this repeatedly, we will make the set of fields chickens can have ended up in shrink smaller and smaller until it includes only s. We can define any field as a "farthest" current field if it contains no incoming edges from unvisited fields. We wish to find an ordering of fields such that each time we visit a field, it is has no incoming edges that have not yet been visited.

This ordering is precisely the output of a topological sort! Topological sort runs on a DAG and outputs a sequence of nodes such that, when visited in order, no node has any unvisited parent nodes. We run a topological sort on our graph starting at s. We then iterate through our fields in the order returned by the topological sort (minus the last element in the returned sequence, s). This ensures that all of our chickens are pushed back to the field by our house, and we can run our chicken farm in peace. For now.

Running topological sort is the same time complexity as a DFS, and requires $O(|V| + |E|)$ time. Visiting the fields themselves takes $O(|V|)$ time. Therefore, the overall runtime for the algorithm is $O(|V| + |E|)$.

An alternative approach was to do a BFS from s on a graph G' which was the same as G, but with all the edges reversed. Then a valid ordering for visiting the fields was in reverse order of the "distance" generated by this modified BFS. This approach also takes $O(|V| + |E|)$ as that is the runtime of a BFS on the modified graph.

(d) [1 points] (Extra Credit) Can you visit the fields in such an order that you would force a chicken to violate its own behavioral rules? (Of course, as this is a cruelty-free chicken farm, you would never actually do such a thing.)

Solution: If we know there is a chicken in a field v, and then visit all of v's child fields and then visit v, the chicken will have no valid behavioral options and will seg fault.

Part B

Problem 3-4. [30 points] **Rolling in the Deep with Wildcard Strings**

6006LE believes in giving its interns high impact projects. For your first project, you've been tasked with working on the 6006LE core search functionality. Specifically, you need to implement various forms of a FIND(*pattern*, *document*) function, that searches for a given string *pattern* inside of a larger *document* string. Both the *pattern* and the *document* strings are written in an alphabet A with a letters.

Your colleague suggested that you should use rolling hashing, and gave you a helper library (`rolling_hash.py`) that you can import and use. Specifically, you will be using a rolling hash function H that maps strings to the range $[0, 1, \dots, m-1]$, and is given by:

$$H(c_1c_2c_3\cdots c_k) = \left(c_1a^{k-1} + c_2a^{k-2} + c_3a^{k-3} + \cdots + c_ka^0\right) \mod m.$$

Here, $c_i \in A$ for $i = 1, 2, \dots, k$ are characters from the alphabet A. If you have computed the hash of a string $c_1c_2c_3\cdots c_k$, and you want to compute the hash of $c_2c_3\cdots c_kc_{k+1}$ from some $c_{k+1} \in A$, you can "roll the hash forward" as follows:

$$\begin{aligned}H(c_2c_3\cdots c_kc_{k+1}) &= c_2a^{k-1} + c_3a^{k-2} + \cdots + c_ka^1 + c_{k+1}a^0 \mod m\\ &= \left(c_1a^{k-1} + c_2a^{k-2} + c_3a^{k-3} + \cdots + c_ka^0 - c_1a^{k-1}\right)\cdot a + c_{k+1} \mod m\\ &= \left(H\left(c_1c_2c_3\cdots c_k\right) - c_1a^{k-1}\right)\cdot a + c_{k+1} \mod m.\end{aligned}$$

(a) [5 points] Implement the function ROLLFORWARD that rolls the hash forward: skipping a character from the start of the rolling hash, followed by appending a new character to the end of the rolling hash. Look at the `rolling_hash` library that has been given to you; you will use and modify some of the fields inside the `rolling_hash` instance that is passed to ROLLFORWARD. Your implementation should run in $O(1)$ time.

Solution:

```
def roll_forward(rolling_hash_obj, next_letter):
    """
    "Roll the hash forward" by discarding the oldest input character
         and
    appending next_letter to the input. Return the new hash, and
        save it in rolling_hash_obj.hash_val as well

    Parameters
    ----------
    rolling_hash_obj : rolling_hash
        Instance of rolling_hash
    next_letter : str
        New letter to append to input.

    Returns
    -------
    hsh : int
        Hash of updated input.
    """
    self = rolling_hash_obj

    # Pop a letter from the left.
    old_letter = rolling_hash_obj.sliding_window.popleft()
    old_val    = self.alphabet_map[old_letter]

    # Push a letter to the right.
    self.sliding_window.append(next_letter)
    # return self.init_hash("".join(self.sliding_window))
    new_val = self.alphabet_map[next_letter]

    # Roll the hash_val.
    self.hash_val = ((self.hash_val - self.a_to_k_minus_1 * old_val)
        * self.a + new_val) % self.m

    # Return.
    return self.hash_val
```

(b) [10 points] Now, complete the implementation of EXACTSEARCH, which determines if the *document* string contains the specified *pattern* string. Your implementation should run in $O(n)$ time.

(*Hint:* You can use part (a).) **Solution:**

```
    # Store parameters. Set default alphabet.
    # DO NOT MODIFY
    rolling_hash_obj.set_roll_forward_fn(roll_forward)
    ## END OF DO NOT MODIFY ##

    # may be helpful for you
    n = len(document)
    k = len(pattern)

    # If n < k, cannot possibly have a match.
    if n < k: return None

    # hash pattern
    pattern_hash = rolling_hash(rolling_hash_obj.k, rolling_hash_obj
       .alphabet).init_hash(pattern)

    # hash substrings of document
    frame_hash = rolling_hash_obj.init_hash(document[:k])
    if frame_hash == pattern_hash: return 0
    for inclusive_end_idx in range(k, n):
        next_hash = rolling_hash_obj.roll_forward(document[
            inclusive_end_idx])
        if next_hash == pattern_hash:
            if "".join(rolling_hash_obj.sliding_window) == pattern:
                return inclusive_end_idx - k + 1

    # No match! Return None explicitly.
    return None
```

(c) [15 points] After finishing your intern project, you realize that you still have plenty of time left until the end your internship! 6006LE is worried that hackers might be corrupting documents by changing some of the document characters. They know which characters could have been changed, but they're not sure what those characters are supposed to be. However, 6006LE still wants to implement a document search.

To approach this, you'd like to write a function called CORRUPTEDSEARCH($pattern$, $document$). In this case, any character of the document could be '?', which means that the character could have been corrupted, and it can match with any character from the alphabet. For example, the $pattern$ `abc` matches the document `acc?bcc?`, if the first '?' is an 'a.' Notice that the alphabet A does not contain the character '?', which is only used to indicate a corrupted character.

Assume that the document has length n, and the pattern has length k. Using the idea of a rolling hash, design an algorithm to solve CORRUPTEDSEARCH in $O(2^k + n)$ time. As usual, you should justify the correctness and runtime of your algorithm. In addition, discuss *approximately* under what conditions this algorithm performs better than the naive algorithm of iterating through every length-k substring, and checking if any of them match the pattern string.

You do **not** need to implement this function; it suffices to include your answer in your written portion. Note that your implementation must work if the range of the hash function m is a prime number that satisfies $m = \Theta(n)$.

Solution: For a given pattern, there are 2^k possible patterns, considering all possibilities of which indices are '?'s. For example, the pattern "ab" can match with "ab", "?b", "a?", or "??" in the document. We place these hashes and place them in an array indexed by the *wildcard index*, whose binary encoding indicates where the '?'s are in the pattern. For example, if the pattern is "ab", we have `arr[00] = hash("??")`, `arr[01] = hash("?b")`, `arr[10] = hash("a?")`, `arr[11] = hash("ab")`.

Note that computing all pattern strings, then computing their hashes from scratch, runs in $O(k \cdot 2^k)$. For an $O(2^k)$ algorithm, we can implement a recursive approach as follows:

```
# Store an array of size 2^k for the hash values
hashes = [0]*(2**k)

# Precompute all the powers of a mod m
```

```
powers = []
cur = 1
for i in range(k):
    powers.append(cur)
    cur = (cur * rolling_hash.a) % rolling_hash.m

# Recursively generate all our hashes
def gen_hashes(hash_val, index, i):
    # We've reached the end, add it to our hashes array
    if i == len(pattern):
        hashes[index] = hash_val
        return

    # This is the case where we have a non-corrupted character. Add
        the hash term corresponding with
    # the character at this index. In addition, set the
        corresponding bit in the index to 1.
    new_hash = (hash_val + powers[k - 1 - i]*rolling_hash.
        alphabet_map[pattern[i]]) % rolling_hash.m
    new_index = index + 2**(k - 1 - i)
    gen_hashes(new_hash, new_index, i + 1)

    # This is the case where we have a corrupted character. Add the
        hash term corresponding with
    # the ? character. In addition, don't add anything to the index
        (which sets the bit as 0).
    new_hash = (hash_val + powers[k - 1 - i]*rolling_hash.
        alphabet_map['?']) % rolling_hash.m
    new_index = index
    gen_hashes(new_hash, new_index, i + 1)

# Call this to actually populate the hashes
gen_hashes(0, 0, 0)
```

Now, we perform a rolling hash on the document. However, we also keep track of the *wildcard index*, which stores where the corrupted characters are in the document.This way, we know exactly which hash to compare with for each iteration. For example, if the document is `dd??cb` and our pattern is `abc`, the length-3 document substrings are `dd?, d??, ??c, ?cb`. It follows that our wildcard indices are $110, 100, 001, 011$ – indexing this into our hashes array, this means we only need to compare these substrings with the hash values for `ab?, a??, ??c, ?bc`, respectively. Note that we can roll the wildcard index forward in $O(1)$ just like our rolling hash; the difference is that it's a binary number, and we're simply checking whether each new character is a '?' or not.

In summary, at any iteration, we roll forward our rolling hash and wildcard index. Then, we index into our hashes array and compare with the corresponding hash – if the hash values are equal, then we check whether the substring matches the pattern. Now, the probability of a collision is $1/m$, just like the normal Karp Rabin algorithm, and our rolling hash runs in $\Theta(n)$.

Altogether, precomputing the hashes runs in $O(2^k)$, and the rolling hash runs in $O(n)$, so we have a total runtime of $O(2^k + n)$.

Note that the naive algorithm runs in $O(nk)$, so this algorithm performs better *approximately* when $k = \log n$. In practice, the rolling hash algorithm would not be very inefficient.

Note: A common approach was to place all 2^k possible values in a set, and check if every substring hash matches any pattern hash in the set. If the hash value is in the set, then check the entire substring to see if it matches the pattern string, which takes $O(k)$. However, the probability this happens is around $2^k/m$, which is $O(2^k/n)$. If checking the entire substring takes $O(k)$, and we're looping through all n indices, we have an expected cost of $O(2^k/n \cdot n \cdot k) = O(2^k \cdot k)$ – which doesn't fit within the runtime constraints.

In addition, many students simply claimed that you can precompute all the hashes in $O(2^k)$, without realizing this is nontrivial.