*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology — March 3, 2016
Instructors: Nancy Lynch, Aleksander Madry, and Bruce Tidor — Problem Set 3

# Problem Set 3

**All parts are due March 17, 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.** [25 points] **AVL trees**

An AVL tree, as presented in Lecture 6, is a certain kind of balanced Binary Search Tree. Between operations, an AVL tree satisfies the invariant that, for every internal node $u$ in the tree, the heights of $u$'s subtrees differ by at most $1$. In particular, if we use the common convention that the height of an empty tree is $-1$, then if node $u$ has only one subtree, that subtree must have height $0$, that is, it must consist of just a single node.

The AVL tree maintains this invariant during an Insert operation by inserting the new entry at a new leaf node, then moving up the tree from the new leaf node, rebalancing as long as the difference in heights between a node's subtrees is greater than $1$. A key fact is that, when a subtree rooted at a node $u$ is rebalanced during an Insert operation, the height of the subtree after rebalancing is either the same as it was just before the rebalancing, or is reduced by exactly $1$. This is important to ensure that rebalancing at one node cannot cause changes that could make it impossible to rebalance at higher nodes.

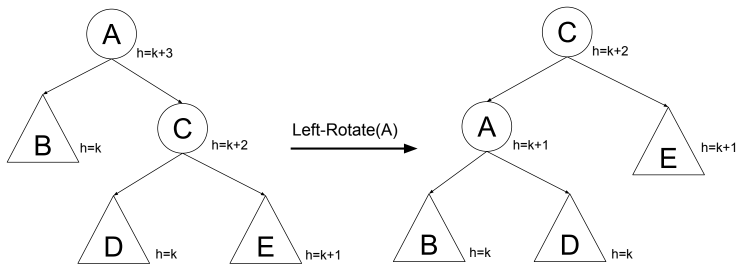

**Figure 1**: After rebalancing, the subtree rooted at $A$ is now rooted at $C$, and the height is reduced by $1$ from $k+3$ to $k+2$.

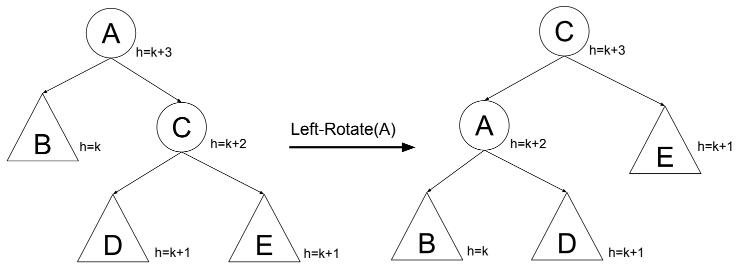

**Figure 2**: After rebalancing, the subtree rooted at $A$ is now rooted at $C$, and the height remains the same, $k + 3$.

One might wonder whether AVL trees do too much work by rebalancing every time the tree gets even a little out of balance. In this problem, we will consider allowing a little more leeway, maintaining the invariant that the heights of each node's subtrees differ by at most 2. (In particular, if $u$ has only one subtree, it must have height at most 1.) Let's call the new type of tree an *AVL-2 tree*.

**(a)** [4 points] Describe in words an algorithm based on rebalancing to implement the Insert operation on an AVL-2 tree.

**Solution:** We insert the new node into the tree. This may cause imbalances. In an AVL-2 tree, an imbalance happens when the left and right subtrees' heights differ by 3 (instead of 2 as in a normal AVL tree). As in an AVL tree, imbalances may occur on a segment of the branch from the new node toward the root. We start rebalancing at the lowest unbalanced node on the branch and move upward until the imbalance is resolved. Specifically, we can stop when we reach a node where the height is restored to what it was before the Insert, or when we reach the root of the tree.

For rebalancing at a node, the same three cases work as for AVL Insert. The only difference is that we invoke the cases when the difference between the heights is greater than 2, instead of greater than 1. For some more detail on the cases, see the solution to part (c) below.

**(b)** [6 points] Show that, when a subtree is rebalanced during your AVL-2 Insert algorithm, the height of the subtree after rebalancing is either the same as it was just before the rebalancing, or is reduced by exactly 1.

**Solution:** This requires a detailed analysis of the cases that arise in rebalancing at a node. Let's establish some notation. Let $A$ be the lowest node that violates the AVL-2 invariant, and let $B$ and $C$ denote its left and right children, respectively. Suppose the height of the subtree rooted at $A$, $height(A)$, is $k + 4$. Without loss of generality we can assume $height(B) < height(C)$; then because of the imbalance at $A$, it must be that $height(B) = k$ and $height(C) = k + 3$.

By assumption, the subtrees rooted at $B$ and $C$ are both balanced (i.e., within $2$). Since the subtree rooted at $C$ is balanced and $height(C) = k + 3$, one of its children has height $k + 2$, and the other must have height $k + 2$, $k + 1$, or $k$. Let $D$ and $E$ denote the left and right children of $C$, respectively.

Then we consider cases. Here we are combining the first two cases from the lecture into one, because they are both handled in the same way.

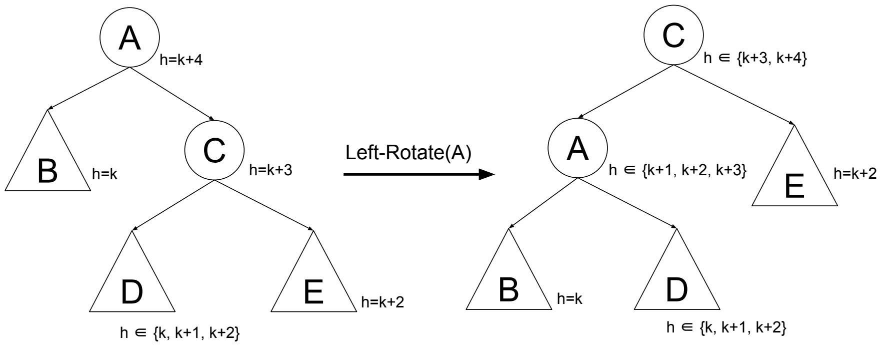

**Figure 3**: Case 1 of rebalancing. The height of the subtree remains the same or decreases by one.

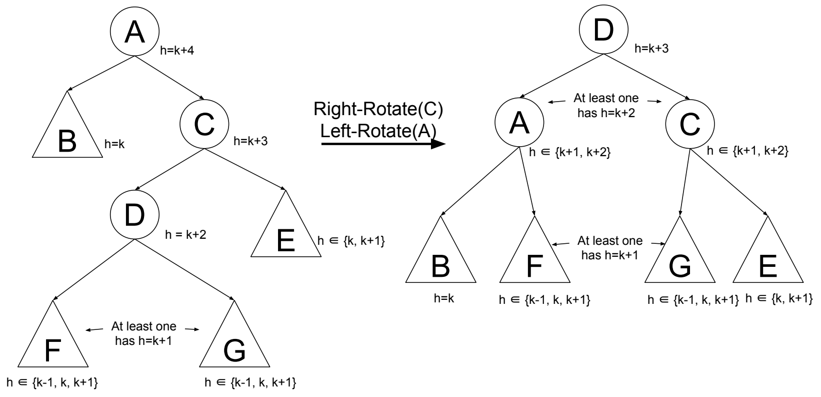

**Figure 4**: Case 2 of rebalancing. The height of the subtree decreases by one.

*Case 1:* $height(E) = k + 2$
Since the subtree at $C$ is balanced (within 2) and $height(C) = k + 3$, it must be that $height(D) \in \{k + 2, k + 1, k\}$. We do a left-rotation at $A$, making $C$ the new root. $A$ becomes the left child of $C$, and $D$ becomes the right child of $A$. ($E$ remains the right child of $C$, and $B$ remains the left child of $A$.) Then because $A$ is now $D$'s parent, the new height of $A$ is in $\{k+3, k+2, k+1\}$. So the new height of $C$ is in $\{k+4, k+3\}$, as needed.

*Case 2:* $height(D) = k + 2$ and $height(E) \neq k + 2$
This is the more complicated case. It must be that $height(E) \in \{k + 1, k\}$. Let $F$ and $G$ be the left and right children of $D$, respectively. Since $height(D) = k + 2$, we know that one of $F$ and $G$ must have height exactly $k + 1$ and since $C$'s subtree is balanced, the height of the other is in $\{k + 1, k, k - 1\}$.

The double-rotation causes $D$ to become the new root, with left child $A$ and right child $C$. Also, $A$ keeps left child $B$ and acquires right child $F$. And $C$ acquires left child $G$ and keeps right child $E$. Now, because $A$'s children are now $B$ and $F$, $A$'s new height must be in $\{k + 2, k + 1\}$. Similarly, since $C$'s children are now $G$ and $E$, $C$'s new height must be in $\{k + 2, k + 1\}$. Moreover, since one of $F$ and $G$ has height $k + 1$, we know that one of $A$ and $C$ has height equal to $k + 2$. Therefore, the new height of $B$ must be exactly $k + 3$, as needed.

**(c)** [5 points] Give an example sequence of insertions, starting from an empty tree, for which an AVL tree rebalances at least twice but an AVL-2 tree does not rebalance at all.

**Solution:** Consider the sequence $4, 5, 6, 1, 2, 3$. It causes the AVL tree to rebalance upon insertion of 6, 2, and 3. Some of these are double insertions. On the other hand, the AVL-2 tree does not rebalance at all.

**(d)** [5 points] Prove that AVL-2 trees always have height $O(\log n)$, where $n$ is the number of nodes in the tree.

**Solution:** Let $N_h$ be the minimum number of nodes in an AVL-2 tree of depth $h$. Then one can write a recurrence for these heights. In the worst case the heights of the two top-level subtrees of an AVL-2 tree differ by 2; thus

$$N_h \geq N_{h-1} + N_{h-3} + 1 \geq 2N_{h-3}.$$

Solving the recurrence gives:

$$N_h \geq 2^{h/3}.$$

Therefore,

$$h \leq 3 \log (N_h).$$

It follows that $h$ is $O(\log n)$.

**(e)** [5 points] We can generalize the construction of AVL-2 trees to AVL-$c$ trees, where $c$ is any positive integer. Namely, define an *AVL-c tree* to be the same as an AVL tree except that the invariant is that the heights of each node's subtrees differ by at most $c$. (In particular, if $u$ has only one subtree, it must have height at most $c - 1$.) Prove that, for any positive integer $c$, AVL-$c$ trees always have height $O(\log n)$, where $n$ is the number of nodes in the tree.

**Solution:** As before, let $N_h$ be the minimum number of nodes in an AVL-$c$ tree of depth $h$. Now our recurrence looks like:

$$N_h \geq N_{h-1} + N_{h-1-c} + 1 \geq 2N_{h-1-c}$$

Solving the recurrence gives:

$$N_h \geq 2^{h/(c+1)}$$

Therefore,

$$h \leq (c+1) \log N_h.$$

It follows that $h$ is $O(c \log n)$, which is $O(\log n)$ for constant $c$.

**Problem 3-2.** [15 points] **Lower Bound on Sorting**

Suppose that we are given a sequence $A$ of $n = mk$ elements to sort, with no duplicate elements. But now, the input sequence is not completely arbitrary: It consists of $m$ successive subsequences, $A_1, A_2, \ldots, A_m$, each containing exactly $k$ elements. Each of the subsequences is unsorted. However, for every $i$ and $j$, $i \neq j$, either all elements of $A_i$ are less than all the elements of $A_j$, or vice versa.

For example, $A$ might be the sequence $[10, 12, 11, 5, 4, 6, 9, 7, 8, 2, 3, 1]$, with $n = 12$, $m = 4$, $k = 3$, $A_1 = [10, 12, 11]$, $A_2 = [5, 4, 6]$, $A_3 = [9, 7, 8]$, and $A_4 = [2, 3, 1]$.

**(a)** [3 points] Explain briefly why the given constraints imply that there is some way of sorting $A$ in which all the elements of each $A_i$ appear consecutively.

**Solution:** We can define a strict total ordering on the subsequences: Define $A_i < A_j$ exactly if all elements of $A_i$ are less than or equal to all elements of $A_j$. This satisfies the usual conditions for a strict total ordering (orders every pair, transitive, antisymmetric). If we arrange the subsequences according to this total order, and within each subsequence, arrange the elements in increasing order, then the result is a complete sort of $A$. Moreover, all the elements of each $A_i$ appear consecutively because they were sorted initially according to the total ordering of the subsequences.

**(b)** [3 points] Considering all possible inputs $A$ satisfying the given constraints, exactly how many different possible permutations of the indices in $A$ can occur in the sorted outputs?

**Solution:** There are $m!$ possible orders for the set of subsequences. For each possible order for the subsequences $A_1, ..., A_m$, we can choose the orders of the elements within the individual subsequences. Within each subsequence, there are $k!$ possible orders for the elements. Since any combination of orders is allowed, the total number of allowed orders of all the subsequences is $(k!)^m$. Therefore, the overall count of possible permutations is $m! \times (k!)^m$.

**(c)** [5 points] Prove an asymptotic lower bound on the number of comparisons needed (in the worst case) to sort all arrays $A$ satisfying the given constraints.

**Solution:** For any algorithm to be successful, it must be able to produce any possible valid permutation, that is, any permutation as described in part (b). Thus, the number of leaves in a decision tree must be at least $m! \times (k!)^m$, one for each valid permutation. But comparing two distinct elements results in only two possible options. This leads to the following bounds on the number $l$ of leaves:

$$m! \times (k!)^m \leq l \leq 2^h,$$

where $h$ is the height of the tree. Therefore:

$$h \geq \log(m!) + m \log(k!) = \Omega(m \log m + mk \log k),$$

where the last expression was obtained using Stirling's approximation.

**(d)** [4 points] Describe (in words) an algorithm that sorts an input sequence $A$ that is known to satisfy the given constraints. The values $n$, $m$, and $k$ are parameters of the algorithm, so the algorithm can use knowledge of those. Analyze the time complexity of your algorithm, and make your runtime as tight as possible, in particular, try to make it match the lower bound on comparisons that you proved in part (c).

**Solution:** For each of the $k$ different subsequences, extract the first element (or in other words, every $k^{th}$ element of $A$), and sort those elements. These $m$ sorted elements will give us the ordering of the subsequences. This takes $O(m \log m)$ time using (for example) Merge Sort. Then sort each of the $m$ subsequences. This takes $O(k \log k)$ time for each subsequence. Thus, the total time to sort all of the subsequences separately is $O(mk \log k)$. Finally, putting the sequences one after the other after they were sorted gives the required entire sorted sequence, leading to a grand total of $O(m \log m + mk \log k)$ time. Since it matches the lower bound on comparisons from part (c), we can conclude that $T(n) = \Theta(m \log m + mk \log k)$

**Problem 3-3.** [20 points] **Alphabetizing the Phone Book**

Design an algorithm to alphabetize an unsorted list of names and phone numbers. Specifically, assume that you are given an arbitrarily long list $L$ of entries that are triples of the form (last name, first name, phone number). Last names and first names consist of letters of the alphabet, and the distinction between upper and lower case is not significant. The list contains no duplicate triples, but it may contain the same full name with different phone numbers. Your program should output a new list containing the same entries, but the names should be alphabetized in the usual way, according to last name and for the same last name, according to first name. Moreover, all the triples containing the same full name (and different phone numbers) should appear in the same order in the output list as in the input list.

**(a)** [10 points] Describe carefully, in words, an algorithm that, given a list $L$ of $n$ triples of the form (last name, first name, phone number) having no duplicate triples, sorts the list based on last names. Moreover, all the triples containing the same last name should appear in the same order in the output list as in the input list. Your algorithm should run in time $O(k_l n)$, where $k_l$ is an upper bound on the lengths of any last name in the list.

**Solution:** We first transform all last names to be of the same length. We do this by scanning the list to find the longest length of any last name, $k_l$. Then, we pad the end of all last names with "null" characters until they are all of length $k_l$. We define the "null" to precede all other characters in the total ordering of chracters.

Now, we sort $L$ using radix sort. In particular, to sort on the last names while keeping everything else in order, we sort (just) on the indices that are part of the last name, in order from the low-order digit (highest numerical index) to the high-order digit (lowest index). These should be indices $k_l - 1, k_1 - 2, ..., 0$. Once we have this sorted list, we remove all "null" characters from all last names.

Pseudocode:

Here, $i$ is the index into each entry tuple in $L$. For example, when $i$ = 0, we are referring to last names. Similarly, if $i$ = 1, we are referring to the first names.

$l$ is an index into the list $L$.

$j$ is the position of the character in the last/first name we are sorting on. For example, $j$ = 0, $i$ = 0, and $l$ = 0 refers to the first character of the last name of the first entry in $L$.

$k$ is the upper bound on the lengths of either last or first names in $L$.

$null$ = "0"
$C$ = [$null$, "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]

```
FIND-K($L, i$)
1  $k$ = 0
2  for $l \leftarrow 0$ to length($L$)-1
3        $k$ = max($k$, length($L[l][i]$))
4  return $k$

ADD-NULLS($L, i, k$)
1  for $l \leftarrow 0$ to length($L$)-1
2        $L[l][i]$.append($null$ * ($k$ - length($L[l][i]$)))

REMOVE-NULLS($L, i$)
1  for $l \leftarrow 0$ to length($L$)-1
2        $L[l][i]$.replace($null$, “”)

STABLE($C, L, k, i, j$)
 1  $characters$ = {}
 2  for $l \leftarrow 0$ to length($L$)-1
 3        $currentChar$ = $L[l][i][j]$
 4        if $currentChar$ not in $characters$
 5              $characters[currentChar]$ = []
 6        $characters[currentChar]$.append($L[l]$)
 7  $results$ = []
 8  for $c \leftarrow 0$ to length($C$)-1
 9        for $a \leftarrow 0$ to length($characters[C[c]]$)-1
10              $results$.append($characters[C[c]][a]$)
11  return $results$

RADIX-SORT($L, i$)
1  $k$ = Find-K($L$, $i$)
2  Add-Nulls($L$, $i$, $k$)
3  for $j \leftarrow k - 1$ to 0
4        $L$ = Stable($C$, $L$, $k$, $i$, $j$)
5  Remove-Nulls($L$, $i$)

Radix-Sort($L$, 0)
```

Calling Radix-Sort($L$, 0) will $L$ sort by last names.

**(b)** [6 points] Now describe carefully, in words, an algorithm for the main problem: given a list $L$ of $n$ triples of the form (last name, first name, phone number) having no duplicate triples, it should sort the list based on last names, and for the same last name, according to first name. All the triples containing the same full name (both first and last name) should appear in the same order in the output list as in the input list.

Your algorithm should run in time $O(kn)$, where $k$ is an upper bound on the lengths of any last names and any first names in the list.

**Solution:** We first run our algorithm in part **(a)** on first names rather than last names. We then run our algorithm in part **(a)** on last names. The key idea behind this is that our sorting algorithm preserves the order of triples in the original list on the event of a tie. Thus, if we first sort on first names and then sort on last names, in the event of a ties in last names, our prior order (which was sorted on first names) would be preserved.

Pseudocode:

```
Radix-Sort($L$, 1)
Radix-Sort($L$, 0)
```

Other than these two lines, we have the same pseudocode as in part **(a)**. We first call Radix-Sort($L$, 1) to sort $L$ by first names and then we call Radix-Sort($L$, 0) to sort $L$ by last names.

**(c)** [4 points] Analyze the running times of your algorithms from parts (a) and (b).

**Solution:** Scanning the list to find the longest length of any last name, $k_l$ can be done in $O(n)$ time. This is because finding the length of a string is an $O(1)$ operation and we do this $n$ times, one for each triple in $L$.

Padding the end of all last names with "null" characters until they are all of length $k_l$ can be done in $O(k_l n)$ time. This is because appending up to $k_l$ characters to string can be done in $O(k_l)$ time and we do this $n$ times.

Similarly, removing all "null" characters from all last names can also be done in $O(k_l n)$ time. This is because removing the last up to $k_l$ characters from a string can be done in $O(k_l)$ time and we do this $n$ times.

Each run of sorting can be done in $O(n)$ time. Each run involves adding $n$ triples to a constant number of buckets which is an $O(1)$ operation for each triple. After adding all the triples to the buckets, we extract the triples from the buckets in order which also requires $O(n)$ time. In total, this requires $O(n)$ time. Since we sort $k_l$ times and each sort requires $O(n)$ time, actually sorting the terms after preprocessing requires $O(k_l n)$ time.

In total, the runtime of **(a)** is the sum of scanning $L$ to find $k_l$, padding the end of all last names with "null" characters, sorting $L$, and then removing all the padding. This is $O(k_l n)$ time in total.

Since **(b)** requires running our algorithm from **(a)** twice, once for first names and another for last names, the runtime of **(b)** is $O(k_f n + k_l n)$. Here, $k_f$ is is an upper bound on the lengths of any first name in $L$. However, since $k = max(k_f, k_l)$ as $k$ is an upper bound on the lengths of any first or last name in $L$, the runtime of our algorithm is also $O(kn)$.

**Problem 3-4.** [20 points] **Analysis of Hashing With Chaining**

Analysis of hashing with chaining usually assumes a "simple uniform hashing" property of the operation sequence: regardless of what has happened in the past, the key that appears in the next operation in the sequence is equally likely to be one that hashes to any location in the table. This assumption makes analysis easier, and is approximately achieved by real hash functions.

This problem considers some issues involved in analyzing the behavior of hash tables with chaining, under the simple uniform hashing assumption and some variations. In all cases, we assume that the hash table has $m$ locations numbered $0, \ldots, m-1$. We are considering scenarios in which $n$ Insert operations occur for successive, distinct keys $k_1, k_2, \ldots, k_n$, followed by a single Search operation.

**(a)** [3 points] Suppose that the $n$ Inserts are all for keys that hash to table location $0$. This is unlikely according to the simple uniform hashing assumption, but it could happen. Then the Search operation requests a key randomly, in such a way that the resulting hash value is equally likely to be any location in the hash table. Give a good upper bound on the expected time for the Search operation to complete, as a function of $n$ and $m$.

**Solution:** The given Search operation is equally likely to reach any of the $m$ slots. Thus, the probability it reaches slot $0$ is $\frac{1}{m}$ and the probability it reaches any of the remaining ones is $1 - \frac{1}{m} = \frac{m-1}{m}$.
Formally, let $H_0$ be the event that the search reaches location $0$ and $H_{other}$ be the event that the search reaches anywhere else. Then $Pr(H_0) = \frac{1}{m}$ and $Pr(H_{other}) = \frac{m-1}{m}$. Define the random variable $T$ to be the amount of time the search operation takes to complete. Then we want $E[T]$, the expected amount of time for search to complete. Using the law of total probability (or iterated expectation or tower rule) we get:

$$E[T] = E[T|H_0]Pr(H_0) + E[T|H_{other}]Pr(H_{other}).$$

If we go to location 0 then the expected time to search that chain, $E[T|H_0]$, is at most $n + c$ for some constant $c$. If we go anywhere else, the expected time to search the (empty) chain, $E[T|H_{other}]$, is at most a constant $c$. Therefore:

$$E[T] \leq (n+c) \cdot \frac{1}{m} + c(1 - \frac{1}{m}) = \frac{n}{m} + c,$$

which implies that $E[T]$ is bounded by $\frac{n}{m} + O(1)$.

**(b)** [2 points] Now consider a different pattern of Insert locations. Suppose that, out of the $n$ Inserts, half are for keys that hash to location $0$ and half for keys that hash to location $1$. Again, the Search operation requests a key randomly, so that the resulting hash value is equally likely to be any location in the hash table. Again, give a good upper bound on the expected time for the Search operation to complete, as a function of $n$ and $m$.

**Solution:** Let $H_i$ be the event that the search reaches location $i$, for $i \in \{1, 2\}$, and let $H_{other}$ be the event that the search reaches anything else. Then $Pr(H_i) = \frac{1}{m}$ for $i \in \{1, 2\}$ and $Pr(H_{other}) = \frac{m-2}{m}$. Define the random variable $T$ as in part (a)—the amount of time the search operation takes to complete. We want $E[T]$, the expected amount of time for search to complete.

Using the law of total probability as before, we get:

$$E[T] = E[T|H_0]Pr(H_0) + E[T|H_1]Pr(H_1) + E[T|H_{other}]Pr(H_{other}).$$

Now $E[T|H_i] \leq \frac{n}{2} + c$ for $i \in \{0, 1\}$. So:

$$E[T] \leq 2(\frac{n}{2} + c)(\frac{1}{m}) + c(\frac{m-2}{m}) = \frac{n}{m} + c.$$

Therefore, the bound is the same as before:

$$E[T] = \frac{n}{m} + O(1).$$

**(c)** [3 points] Generalize the result and your analysis for parts (a) and (b) to any other possible pattern of Insert locations.

**Solution:** Suppose that $n_i$ is the number of inserts that hash to location $i$. Then $\sum_{i=0}^{m-1} n_i = n$, since the total number of elements in the hash table is exactly $n$. Let $H_i$ be the event that the search reaches location $i$; then $Pr(H_i) = \frac{1}{m}$ for every $i \in \{0, \ldots, m-1\}$. Then we have:

$$E[T] = \sum_{i=0}^{m-1} E[T \mid H_i]Pr(H_i)$$

$$\leq \sum_{i=0}^{m=1} (n_i + c)(\frac{1}{m})$$

$$= (n + cm)(\frac{1}{m}) = \frac{n}{m} + c.$$

Then: $E[T]$ is bounded by $\frac{n}{m} + O(1)$, as before.

However, the situation becomes different if we consider only successful Search operations:

**(d)** [4 points] As in part (a), suppose that the $n$ Inserts are all for keys that hash to location $0$. But now suppose that we know that the Search is successful, that is, it requests a key that is actually in the table. Suppose that it is equally likely to be any key in the table. Give a good asymptotic upper bound on the expected time for the Search operation to complete.

**Solution:** Now we are conditioning on the search being successful. Thus, the searched-for key must be in the single long chain of $n$ entries. Then the expected time is no more than the time to scan the chain which is at most $n + O(1)$

**(e)** [8 points] But bad tables as in part (d) are not so likely, under the simple uniform hashing assumption. So now let's consider tables that arise as a result of random choices of hash locations for the $n$ Insert operations. Namely, assume that, for each Insert operation, regardless of what has happened in the past, the key that appears in the operation is equally likely to hash to any location in the table.

Suppose again, as in part (d), that we know that the Search is successful, that is, it requests a key that is actually in the table, and suppose that it is equally likely to be any key in the table. Now prove an upper bound of $\frac{n}{m} + O(1)$ on the expected time for the Search operation to complete.
*Hint:* For each inserted key $k_i$, define a random variable $CH_i$ representing the number of inserted keys that hash to the same location as $k_i$, counting $k_i$ itself, that is, the final length of the chain in the location that $k_i$ hashes to. Analyze the expected value of $CH_i$.

**Solution:** We first bound the expected value of $CH_i$. For any $i$ and $j$, $1 \leq i, j \leq n$, define indicator random variable $X_{i,j}$ to be $1$ if $k_i$ and $k_j$ hash to the same location in the table, and $0$ otherwise. As in the hint, define a random variable $CH_i$ to be the number of inserted keys that hash to the same location as key $k_i$, including $k_i$ itself. $CH_i$ gives the final length of the chain in the location that key $k_i$ hashes to.
We can compute the length of the chain for the location of key $k_i$ as:

$$CH_i = \sum_{j \neq i} X_{i,j} + 1,$$

i.e. we are counting the total number of elements $k_j$ that hash to the same location as $k_i$ plus 1 for $k_i$ itself.
We want the expected length of a chain $E[CH_i]$:

$$E[CH_i] = E[\sum_{j \neq i} X_{i,j} + 1].$$

By linearity of expectation we have:

$$E[CH_i] = \sum_{j \neq i} E[X_{i,j}] + 1$$

Using the fact that the expected value of an indicator random variable is the same as its probability of being $1$, and the simple uniform hashing assumption, we get:

$$E[CH_i] = \frac{n-1}{m} + 1 \leq \frac{n}{m} + 1.$$

Now we use the bound on the expected chain lengths to obtain a bound on the expected search time. We assumed that, no matter how the keys get distributed in the hash table, the Search is equally likely to look for any key in the table. Equivalently, the Search will look for each $k_i$ with probability exactly $\frac{1}{n}$.
The expected time to search for any particular key $k_i$ is bounded by $E[CH_i] + c$ for some constant $c$. Therefore, the expected time for the Search operation is

$$E[T] \leq \sum_{i=1}^{n} (E[CH_i] + c)\frac{1}{n},$$

which is just a weighted average of the expected lengths of chains, weighted by the probability of searching for each key. This inequality again follows from the law of total expectation. Since, for every $i$, $E[CH_i] \leq \frac{n}{m} + 1$, we have:

$$E[T] \leq \sum_{i=1}^{n} (\frac{n}{m} + 1 + c)(\frac{1}{n}) = \frac{n}{m} + 1 + c,$$

which results in an expected time bound of $\frac{n}{m} + O(1)$.

**Problem 3-5.** [20 points] **Ancient Scrolls**

At the *Centre for the Study of Ancient Documents*, researchers examine ancient scrolls written in ancient languages and try to determine who wrote them, and what they mean. When you arrive for your summer internship at CSAD, you are given a large pile of long scrolls written in an unknown language over a fairly small, unfamiliar character alphabet $A$. Your assigned job is to try to figure out the author of each scroll. Actually, you are supposed to write a program that will allow CSAD researchers to efficiently determine the authorship of such scrolls in the future.

Fortunately, the researchers at CSAD have already assembled a list of $m$ possible authors. Each author has a "signature", which is a string of symbols from $A$ that appears in each of his or her documents. For some unknown cultural reason, or coincidentally, all signatures are of exactly the same length $k$. Unfortunately, a signature string may appear anywhere in a scroll.

**(a)** [12 points] Describe (in words and pseudocode) an efficient algorithm that searches through a scroll to find the first match of any of the signatures, if any appear in the scroll. If there is no such occurrence, then your algorithm should return "none". Your searching algorithm should run in time $O(n + k)$, normally, where $n$ is the length (number of characters) in the scroll and $k$ is the (fixed, as described above) length of a signature. This time does not include any time for preprocessing the list of signatures, but you should try to keep the preprocessing time to $O(mk)$.

**Solution:** We first preprocess by creating a dictionary with hashed signatures as keys and complete signatures as values. Specifically, we perform a rolling hash on each author's signature and store a list containing the author's signature as the value together with its hash as the key. Multiple signatures could hash to the same value, so we maintain a list of signatures for each key. When multiple signatures have the same hash, the value is a list of all those signatures.

Next, we go through the scroll, finding the rolling hash of every $k$ characters and checking if each hash is in the dictionary. More precisely, we first find the rolling hash of the first $k$ items characters and check if it is in the dictionary. If so, we check if the current $k$ characters are in the list of values for that key in the dictionary. If so, we return the matching signature. We then calculate the hash of appending the next character $c$ and removing the first character of our current hash and repeat our previous action. If we reach the end of the scroll without ever matching a key, we return "none".

Let $p$ be some prime number, let $A$ be the character alphabet, and let $k$ be the length of each signature. We assume we have some function $\text{charToInt}$ which converts each character of the alphabet to an unique integer.

Pseudocode:

```
APPEND-CHAR($hash, c$)
1  return (($hash$*length($A$) )+charToInt($c$)) % $p$
```

```
REMOVE-FIRST($hash, c, L$)
1  return ($hash$ - charToInt($c$) * (length($A$)$^{L-1}$ % $p$)) % $p$

HASH($sig$)
1  $hash$ = 0
2  for $i \leftarrow 0$ to length($sig$)-1
       $hash$ = Append-Char($hash$, $sig[i]$)
4  return $hash$

FIND-AUTHOR($scroll, signatures$)
 1  if length($scroll$) $< k$
 2      return none
 3
 4  $authorDict$ = {}
 5  for $i \leftarrow 0$ to length($signatures$)-1
 6      $authorHash$ = Hash($signatures[i]$)
 7      if $authorHash$ not in $authorDict$
 8          $authorDict[authorHash]$ = []
 9      $authorDict[authorHash]$.append($signatures[i]$)
10
11  $hash$ = Hash($scroll$[0: $k$-1])
12  if $hash$ in $authorDict$
13      if $scroll$[0: $k$-1] in $authorDict[hash]$
14          return $scroll$[0: $k$-1]
15
16  for $i \leftarrow k$ to length($scroll$)-1
17      $hash$ = Append-Char ($hash$, $scroll$[i])
18      $hash$ = Remove-First ($hash$, $scroll$[i-k], $k$ + 1)
19      if $hash$ in $authorDict$
20          if $scroll[i$- $k$+1: $i]$ in $authorDict[hash]$
21              return $scroll[i$- $k$+1: $i]$
22  return none
```

**(b)** [4 points] Analyze the time complexity of your preprocessing algorithm for the list of signatures.

**Solution:** This part takes time $O(km)$, because we process the $m$ signatures one at a time and each takes time $O(k)$ to process. To find the rolling hash of a signature, we must spend $O(k)$ time. Note that it is possible for two signatures to have the same rolling hash. That is OK for correctness—we just store both signatures in the dictionary associated with the common rolling hash value. We will ignore this case for the complexity analysis, however—both in this part and the next.]

**(c)** [4 points] Analyze the time complexity of your main algorithm, which searches the string looking for matches.

**Solution:** The time for this (ignoring the extra time to process multiple signatures associated with the same rolling hash, and also the extra time for checking spurious matches), is $O(n + k)$ (which is $O(n)$).

The analysis for this is similar to that for ordinary rolling hash, for one pattern: Computing the rolling-hash of the first substring takes $O(k)$. For each successive substring, it takes time $O(1)$ to look up the rolling-hash in the dictionary, and $O(1)$ to move on to the next substring (this involves calculating the next rolling hash value). When we find a match, it takes $O(k)$ time to determine if it is "real". We are ignoring the time used to check for matches that turn out not to be correct.

## Part B

There is no Part B this time!