Problem Set 3

All parts are due Thursday, March 20 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. 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.

Your Name: Eric Klinkhammer

Collaborators: Name1, Name2

Part A

Problem 3-1.

(a) Data Structure Description
A hash table has to be used if the user wants constant time access to a random element. A hash table is a data structure that maps every object it stores (and can store) into an integer, and ultimately into an index in an array.

In this way, the hash table knows where to put every object (neglected collisions for the moment) before it begins. This is what saves time on the tasks.

With regards to collisions, or when different objects are mapped to the same index, this can handled in different ways. The two most obvious are through either an additional data structure (a list, for example, if insert is the desired fast operation) or through chaining, where the object is put into another slot in the hash table through some process (for example just putting it in the next open slot).

(b) Algorithm Description
The creation of a hash table from an unsorted array involves just iterating through the array once and mapping each element into the hash table. To reduce collisions, I would create a table with size on the order of but slightly bigger than n, the number of elements in the array. I would then hash them all individually. Each hash has an amoritized cost of $O(1)$, so the entire array can be converted in $O(n)$ time.

To get an unsorted array from the hash table, you would iterate through the array of the table and return the results. This would take $O(m)$ time, where m is the size of the table, but since m was defined to be order n (and it should be, to achieve the right mix of collisions and space saving), the run time of this is also $O(n)$.

(c) Set Membership
To test for set membership, I would take the key of the object I am searching for and look up the location associated with the key. If the location was empty, I would be done and return false. If it had the object we were looking for, I would be equally done. The other case, if it has a different object, would require looking wherever or however the hash table chooses to resolve its collisions.

This test for membership on average takes $O(1)$, but in the worst case, assuming all of the data hashes to the same spot (and then the collision resolution works identically for all of them), it will be $O(n)$ time.

(d) Insertion and Delete
Insert and delete work in a similar fashion as the test for set membership, except, in the case of insert, when the empty slot is found the object is put into that slot, and, in the case of delete, when the object is found, it is deleted. It is important that the delete makes the slot blank (so it won't confuse future collisions). Accordingly, they have a similar run time: $O(1)$ on average, but $O(n)$ in the worse case.

(e) Set Intersection
The intersection of two sets is all of their common elements. Therefore, to build an intersection set, one only has to traverse the members of one set, check the membership of that elment in the other set, and, if found, insert it in the resulting set.

The unsorted array of elements can be taken from the first hash table in $O(n)$ time, and then there will be $O(n)$ lookups and at most $O(n)$ inserts. This means that making the new set will also be an $O(n)$.

(f) Set Union
The union of two sets is all of their elements, without duplicates. Since the insert method in a set already handles duplicates, constructing the union of two sets involves only inserting every element from both into a new set.

It would be $O(n)$ to get the elements in each table, as before, and then $O(n)$ again to insert them into the new set. Again, this entire operation is $O(n)$.

(g) Set Difference
The set difference between sets A and B (A - B) would be all of the elements in A not

also found in B.

This can be created by finding the intersection of A and B, and then iterating through all of the intersection points, deleting them from set A. In practice, you could also copy set A and delete them from the copy, so as to preserve the original sets. Either way, the intersection would be $O(n)$ as above, and the iterating and deleting would be $O(n)$ operations each taking $O(1)$ on average. If you were creating a copy, that could be done in $O(n)$ time, so, regardless, the set difference operation can be done in $O(n)$ time.

Probabilistic Analysis of Hashing With Chaining

(h) Expected Time of Search
Given that those the worst possible case for hash table insertion happens, and all keys are hashed to the same value, the expected value for search is $\frac{(n+1)(n+2)}{2n}$.

All values hashed are equally likely to be chosen by search, and the time for each of those values ranges from 1 to n. Addionally, it is possible that the value is not in the array (I define key space as potential keys, not keys in the set. If this is a mistake, then the sum would only range to n). Expected value is defined as:

$$\begin{aligned} E_t &= \sum_{i=1}^{n+1} p_t t_i \\ &= \frac{1}{n}\sum_{i=1}^{n+1} i \\ &= \frac{(n+1)(n+2)}{2n} \end{aligned}$$

You will notice this is essentially $\frac{n}{2}$, which makes sense, given that its a uniform distribution of between 1 and n time.

(i) Expected Time for Successful Search
The only difference between this and the previous part, as far as I can tell, is that it will never take $O(n+1)$ time when it doesn't find an element. The expected value is calculated essentially the same, with the final answer being:$\frac{(n)(n+1)}{2n}$.

$$\begin{aligned} E_t &= \sum_{i=1}^{n} p_t t_i \\ &= \frac{1}{n}\sum_{i=1}^{n} i \\ &= \frac{(n)(n+1)}{2n} \end{aligned}$$

(j) Expected Time for Search with normal inserts
To determine the expected time for search, we first have to determine the average number of collisions for a given key value.

We define a new random variable CH_i as the number of keys hashed to position i. It has a probability distribution as below. It is important to recognize that such a random variable is a Binomial distribution (with success in the Bernoulli trial being defined as a collision).

$$CH_i = \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1 - \left(\frac{1}{m}\right)\right)^{n-i}$$

Because it is a binomial, we know that the expected value is $\frac{n}{m}$. Search is a constant time operation unless it runs into a collision. Since the expected number of collisions is $\frac{n}{m}$, and the search is normally $O(1)$, the expected run time (not the worse case) of a successful search is $O(1) + \frac{n}{m}$.

Meandering Hashes

(k) Naive Brute Force
Assuming only one orientation is acceptable, you would have to perform up to k^2 checks at each of the potential starting positions of the smaller array, of which there are $(n-k)^2$. The running time is then $O((n-k)^2k^2)$, which, since n is much larger than k, simplifies to $O((nk)^2)$.

(l) Algorithm Description
A rolling hash could be used here in a similar way as the 1-D case. The entire target matrix could be hashed and the value stored. Then, a hash could be computed for the inital k by k block. Then, the hash would roll across the row, each time computing the value of the window that contains k different elements (a new column). Since hash values are integers, and relatively easy to store compared to the matrices being operated on, it makes more sense to store the hash values for the row. Now the window is rolled down one row (k new elements in the roll), but then is rolled across the row, one new element at a time. However, the rolling hash still has to make k updates (k-1 from the top, 1 at the bottom). .

The algorithm's code below assumes that the hash function works in such a way that elments can be rolled off the top and onto the bottom.

(m) Pseudocode

```
int 2D compare( S, T )
        hash = hash(T);
        rolling_hash = hash(T sized matrix with left corner at S(0,0)
        for rows in range(n-k-1):
                rolling_hash.remove(top_row) # Roll off top k values
                rolling_hash.add(new_row) # Roll on k values
                for cols in range(n-k -1):
                        rolling_hash = rolling_hash.remove(old_column)
                        rolling_hash = rolling_hash.add(new_column)
                        check rolling_hash with T:
                                if match, compare element by element
                rolling_hash = the value it was before going across
```

(n) Time Complexity
This algorithm will run in $O(kn^2 + k)$. It must do the initial calculation and the final one (hopefully just one) that both take $O(k)$. The algorithm then does $(n - k)^2$ rolls, but each roll involves k steps. Therefore, the hash rolling and comparison will take $O(kn^2)$. Finally, the constant time to check if the equality is a collision will bring up the running time to $O(kn^2 + k)$.

(o) Space Complexity
This algorithm should use $O(1)$ space, seeing as it only has to store a few integers (hash(T), the value of the one on the left most column (to go down), and the current rolling hash value.