Problem Set 3

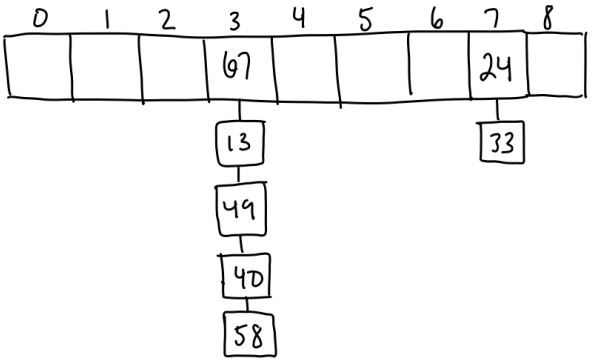
All parts are due September 27, 2019 at 6PM.

Name: Your Name

Collaborators: Terryn Brunelle, Julia Moseyko, David Magrefty

Problem 3-1.

(a) $h(67) = (11 \cdot 67 + 4) \mod 9 = 3$
$h(13) = (11 \cdot 13 + 4) \mod 9 = 3$
$h(49) = (11 \cdot 49 + 4) \mod 9 = 3$
$h(24) = (11 \cdot 24 + 4) \mod 9 = 7$
$h(40) = (11 \cdot 40 + 4) \mod 9 = 3$
$h(33) = (11 \cdot 33 + 4) \mod 9 = 7$
$h(58) = (11 \cdot 58 + 4) \mod 9 = 3$



(b) The maximum chain length is 5.

(c) c=13
Based on universal hashing, c has to be a prime number greater than u (the number of possible keys), which is 9 in this case.

Problem 3-2. `build(A)` for a set runs in O(n) time. Also, we want to initialize a variable `offset` that represents the difference between index 0 and the value of the minimum key of the first value that appears in the set. This variable will be useful for future operations and takes O(1) time to initialize.

`set_at` is done by using the `find(k)` function to find the item at k, where k is equal to the index we've inputted to `set_at` plus `offset`. We then change the value of the item at k to the value of x,

the item that we're trying to insert (i.e. `find(k).value = x`). `get_at` is done by using `find(k)`, where k is the key and is equal to the index that we are trying to get plus the offset value. `find(k)` takes expected O(1) time to complete and changing the value of an element also takes constant time, so the overall running-time is O(1) expected.

`insert_at` is done by shifting all of the items with keys greater than or equal to the key of the item that we're trying to insert (which equals the index plus the offset) up one, ie by increasing their key values by 1. Then, we use `insert(x)` to insert x (as in `set_at`), where x.key is the desired index plus the `offset` value. `delete_at` is the same as `insert_at` except reversed: use `delete(k)` to delete the item with key k, where k equals the desired index plus the `value`. Then, we shift all of the items with keys greater than or equal to the key of the item we're trying to insert down one by decreasing their key values by 1. `insert(x)` and `delete(k)` both take O(1) time, and shifting the keys of the affected items takes worst-case O(n) time because we might have to shift at most n items, and each update of `offset` takes O(1). Therefore, the overall running-time for `delete_at` and `insert_at` is O(n).

`insert_first(x)` is done by using `insert(x)`, where x.key is equal to the `offset` value minus one. Then, we subtract one from `offset` because the new first element is now one slot before. `delete_first` is the opposite of `insert_first(x)`: use `delete(k)` where k is the key that equals the `offset` value, then increase `offset` by 1. `insert_last` is done by using `insert(x)`, where the key of this value would be equal to `offset` plus the size of the set. `delete_last` is the opposite of `insert_last`: use `delete(k)` with the key of the item we're deleting equal to the size of the set minus one plus `offset`. `insert(x)` and `delete(k)` both take amortized expected constant time, and increasing/decreasing the `offset` also takes constant time, and since these are the only operations we are performing, we get an overall running-time of amortized expected O(1) for `insert_first(x)` and `insert_last`.

Problem 3-3.

(a) First, find the minimum value of the IDs; if this minimum value is negative, add its absolute value to all of the IDs so that everything becomes positive and the IDs are shifted to a range of 0 to 2n. If the minimum value is positive or 0, then nothing needs to be done. Now, with all of the keys as positive integers, use radix sort on the keys to sort by the IDs. Finding the minimum takes constant time, adding the minimum value to every ID takes worst-case O(n) time, and radix sort takes O(2n), so overall the algorithm has constant running time O(n).

(b) For each name, hash it by converting each of its letters into the equivalent base-26 number (since there are 26 letters in the alphabet. for example, if the name is "abc", then its number equivalents would be $1 \cdot 26^0, 2 \cdot 26^1, 3 \cdot 26^3$) and then adding all of the letters' numbers together. This sum will be the key for the name, and then we use direct access array sort to sort the names by placing them in the spot corresponding to their key. Getting the hash key for each name takes O(n) time because it takes a constant amount of time to convert each letter and add all the numbers, and there are n names total. Then, placing the names in their correct spot in the array takes O(n) time because it takes constant time to insert each item in its slot based on its key. Overall, the running time would be O(n).

(c) Since all of the numbers are positive, we can just directly do a radix sort using the number of fights as the keys. Each number is converted into a tuple and then sorted from least significant to most significant key. Since the number of fights are positive integers less than n^2, the running time of the algorithm would be less than O($n + n \log_n n^2$) = O(3n) which is a linear running time of O(n).

(d) First, multiply each fraction by the product of all the f_i's and then sort so that we can compare the fractions without dealing with floats. Then, since there is no upper bound on the possible number of fights, use merge sort to sort the critters by their number of wins. Merge sort is faster than something like radix sort when the number of fights is very large because O($n + n \log_n u$) also becomes very large, and merge sort's running time is O($n \log n$). Therefore, the overall running time would be O($n \log n$).

Problem 3-4.

(a) First, initialize an empty hash table. Then, iterate through S; for each side length in S, subtract it from h, and look to see if $h - s_i$ exists in the hash table ($h - s_i$ would be the value that we use as the lookup key). If it exists, then s_i and $h - s_i$ is a pair that sums to exactly h; if it doesn't exist, add s_i to the hash table (with s_i as the key) and repeat this process of subtracting from h and checking the hash table, iterating through S until a pair is found. If we iterate through all of S without finding a pair, then the algorithm returns that such a pair doesn't exist. Computer $h - s_i$, looking up a value in a hash table, and inserting a value in a hash table all take constant time. These operations are done for the n items in S, so the overall running-time is O(n).

(b) First, use radix sort on S so that it is sorted from least to greatest. Initialize a variable called `difference`, which stores the current smallest difference found between a pair of numbers in S and starts off with the value of h, a left index variable that starts at 0, and a right index variable that starts at n-1. Then, iterate through the values in the sorted array by updating the left and right indices until left index is no longer less than right index. For each iteration, if the absolute value of the difference between the value at left index + value at right index and h ($abs(S[left] + S[right] - h)$) is less than the current `difference`, we update `difference` to be this new difference because we have now found two values who sum to a total that is closer to h. We also store the indices of the left and right index that we're currently using into two variables (for example, `final_left, final_right`) that we will reference later. If the sum of the value at the left index plus the value at the right index is greater than h, we decrease the right index by one because the sum is too high. Otherwise, we know that the sum of the two values is too low and not close enough to h compared to other pairs we've looked at, so we have to increase the left index by one. When the left index is no longer less than the right index, we know that we are finished iterating through sorted S because otherwise we would start repeating pairs, and we return the values associated with the `final_left` and `final_right`.

Radix sort takes O($n + n \log_n u$) time, where $u = 600n^6$ because h is the largest possible length that a side length can have (can't have a side that is greater than the height), which means that the radix sort part will take O($n + n \log_n(600n^6)$) worst-case, which is about O($n + 6n$), which is a linear running time. Then, as we iterate through the sorted S, updating the variables within each iteration takes constant time, and we only iterate through each element in S once because we make sure that we don't repeat pairs in the algorithm, so the running-time of this part is O(n). Therefore, the overall running-time is worst-case O(n) for this algorithm.

Problem 3-5.

(a) Instead of starting the sum with the most significant digit, we can start with the least significant digit. To eliminate the repetitiveness of having to multiply each s_i by 128 so many times as in the given R(S) equation, we can keep track of how many 128's have been multiplied already in a variable called `128_num` and just multiply each s_i by this variable and then update it to reflect the new number of 128's that are being multiplied. We also store a variable called `total_sum` to keep track of the sum. For instance, we start by storing 128^0 as the initial value in `128_num` and then multiply the most significant digit $s_{|S|-1}$ by `128_num`. We then add this product to `total_sum` and update `128_num` by multiplying it by 128. Then, we go to the next most significant digit $s_{|S|-2}$ and multiply it by `128_num`, which is now 128^1, as it should be based on the original given equation for R(S), add this product to `total_sum`, and then update `128_num` by multiplying it by 128. We repeat this process of multiplying each digit by the current `128_num`, adding this product to `total_sum`, and updating `128_num` by multiplying it by 128 for each multiplication round in descending order from most significant digit to least significant digit. Initializing `128_num` and `total_sum` take constant time, and for each digit, the multiplication then updating `total_sum` and `128_num` operations each also take O(1) for a total of $O(3|S|)$ for all of the digits. Thus, the overall running-time is $O(|S|)$.

(b) First, we compute $R(D_i)$ using the algorithm from part a). Using this, we can compute $R(D_{i+1})$ by shifting the frame of reference down 1 by subtracting off the value of the first (most significant) digit in D_i and then adding on the value of the next digit of D. More formally, this looks like computing $(R(D_i) - \frac{\texttt{ord(s\_i)} \cdot f}{128}) \cdot 128 + \texttt{ord(s\_\{|Q|+i})$. The $\frac{\texttt{ord(s\_i)} \cdot f}{128}$ part represents the digit that we are removing because the equation for the value of the digit s_i in base 128 states that it is equal to its integer value times $128^{|Q|-1} = \frac{128^{|Q|}}{128} = \frac{f}{128}$ (since it is the most significant digit, it gets multiplied by $128^{|Q|-1}$). Next, this difference gets multiplied by 128 because we have to shift the frame over by one to the next group of $|Q|$ letters, so each digit increases by an order of significance and what used to be the second most significant digit is now the most significant digit for the new substring. Finally, we add $\texttt{ord(s\_\{|Q|+i})$, which represents the integer value of the new digit (the last digit in D_{i+1}). Each of these mathematical operations takes constant time, and since we are only doing this once to compute D_{i+1} from D_i, the overall running time is O(1).

(c) First, we compute the value of R(Q) to know the numerical value of the query string. Computing R(Q) takes $O(|Q|)$, as we found in part a). Next, we go to D and compute $R(D_i)$ for $i \in \{0, ...|D| - |Q| - 1\}$: we get the numerical value for all possible contiguous sub-strings of D that are length $|Q|$, starting from the sub-string of D that begins with index 0 to the one that begins with index $|D|-|Q|-1$ (the last index of D is $|D|-1$ and the substring has length $|Q|$). For each computation of $R(D_i)$, we compare its value to the value of R(Q) and return True if the two values are equal. When we find one that returns True, we can stop iterating, but worst-case we will end up

iterating through all of the $R(D_i)$'s. This comparison takes O(1) for each computation of $R(D_i)$ for a total of $O(|D| - |Q| - 1)$ time to complete all the comparisons. After we compute $R(D_0)$ in $O(|Q|)$, we can use what we found in part b) to compute the next $R(D_i)$ in O(1) time. We just initialize a variable `current_R` at the beginning that keeps track of the value of the current $R(D_i)$, which will start as $R(D_0)$: each time we shift our frame to the next substring, we update the value of `current_R` with the steps from part b). `current_R` is actually what we are comparing to R(Q) at each new sub-string (after comparing, we update `current_R` to reflect $R(D_i)$), which takes O(1) for each (worst-case $O(|D| - |Q| - 1)$ total). The total running-time for computing the $R(D_i)$'s would be $O(|D| - |Q| - 1)$ in the worst-case that we have to go through all of the substrings. Thus, the overall running-time would be at most $O(|D|)$, since $|D| > |Q|$.

(d) We are given the value of $R'(D_i)$, which represents the value of the i^{th} substring of length $|Q|$ in D. Similar to part b), we want to essentially remove the value of the most significant digit in the substring (the first letter), shift the frame, then add on the next letter in D. From $R'(D_i)$, we subtract $(\texttt{ord(s\_i)} \cdot 128^{|Q|-1}) \mod p$, which is equal to $\texttt{ord(s\_i)} \cdot \frac{f}{128} \mod p = \texttt{ord(s\_i)} \cdot \frac{f'}{128}$. Next, we multiply this difference by 128 to represent the shift of significance; each digit increases by an order of significance as we shift to the next substring. Finally, we add the value of the new letter, which equals $\texttt{ord(s\_\{|Q|+i)} \mod p$, and then take the modulus of the entire expression. Overall, the calculation looks like $(R'(D_i) - \frac{\texttt{ord(s\_i)} \cdot f'}{128}) \cdot 128 + (\texttt{ord(s\_\{|Q|+i)} \mod p) \mod p$. Every operation done in this expression takes constant time since they are just mathematical operations, so the overall running-time is O(1).

(e) First, compute $R'(Q)$ to find the value that represents Q, the query string. This takes O(1). Then, we compute $R'(D_{i=0})$ to get the value of the first substring of length $|Q|$, which also takes O(1). Once we get this value, we store it in a variable called `current_R'` that keeps track of the current value of $R'(D_i)$ which will be used later to compute the value of other substrings, and then we compare it to $R'(Q)$ to see if they are equivalent. This comparison takes O(1). If they are not equivalent, then we can move on and compute $R(D_{i+1})$. If they are equivalent, then we must check if the two strings are actually the same or if there was a false match; we compare Q and D_i character by character, which takes $O(|Q|)$. If the two strings are exactly the same, then we return True. If they are not the same, then we move on and compute $R(D_{i+1})$. Using the value stored in `current_R'`, we can calculate $R(D_{i+1})$ in O(1) time from the value of $R'(D_i)$. We update `current_R'` to reflect this new value $R(D_{i+1})$, which takes O(1), and then perform the same checks as above to see if it is equivalent to Q. This checking for equivalence and then shifting the frame process will have to be repeated for a total of $|D|-|Q|-1$ times worst-case because that is the first index of the last substring of length $|Q|$ in D, and if none of the comparisons return True (no match is found), then we will have to go through all of the possible contiguous substrings of length $|Q|$ in D. This means we go through this process $O(|D| - |Q| - 1) = O(|D|)$ times, and each time we make a comparison that takes $O(|Q|)$, so we get

$O(|D||Q|)$ total. However, the likelihood of a fase match occurring is $1/|Q|$, so the overall running-time of this algorithm is $O(|D|)$.

(f) Submit your implementation to `alg.mit.edu`.