Problem Set 3

Name: Vivek Verma

Collaborators: None

Problem 3-1.

(a)

$$\begin{aligned}
&0| \rightarrow 36 \rightarrow 92\\
&1| \rightarrow\\
&2| \rightarrow\\
&3| \rightarrow\\
&4| \rightarrow 56\\
&5| \rightarrow 47 \rightarrow 61 \rightarrow 33\\
&6| \rightarrow 52
\end{aligned}$$

(b) Initialize c = 7, because if c is less than 7, then there cannot be no-collision according to pigeonhole principle.
Manually building the hash table for c =7, 8, 9, 11, 12 leads to collision.
c = 13 is the smallest value of c for which there is no collision.
Hash table for c = 13 is given by ,

$$\begin{aligned}
&0| \rightarrow 36\\
&1| \rightarrow 92\\
&2| \rightarrow 33\\
&3| \rightarrow 61\\
&4| \rightarrow 52\\
&5| \rightarrow 56\\
&6| \rightarrow 47
\end{aligned}$$

Problem 3-2. If k_1 is the ID of Rory and k_2 is the ID of Tiri and chosen hash function is h, then they will be assigned same rooms if and only if

$$h(k_1) = h(k_2)$$

(a)

$$\begin{aligned}
&h_{ab}(k_1) = h_{ab}(k_2) \\
\implies\ &(ak_1 + b)\ mod\ n = (ak_2 + b)\ mod\ n \\
\implies\ &(ak_1 + b) \equiv (ak_2 + b)\ mod\ n \qquad [\because x\ mod\ n = y\ mod\ n \iff x \equiv y\ mod\ n] \\
\implies\ &k_1 \equiv k_2\ mod\ n
\end{aligned}$$

i.e Rory and Tiri are guranteed to be roommates if the difference between their IDs is divisible by n.

(b)

$$\begin{aligned}
&h_a(k_1) = h_a(k_2) \\
\implies\ &\left(\left\lfloor \frac{k_1 n}{u} \right\rfloor + a\right)\ mod\ n = \left(\left\lfloor \frac{k_2 n}{u} \right\rfloor + a\right)\ mod\ n \\
\implies\ &\left(\left\lfloor \frac{k_1 n}{u} \right\rfloor + a\right) \equiv \left(\left\lfloor \frac{k_2 n}{u} \right\rfloor + a\right)\ mod\ n \qquad [\because x\ mod\ n = y\ mod\ n \iff x \equiv y\ mod\ n] \\
\implies\ &\left\lfloor \frac{k_1 n}{u} \right\rfloor \equiv \left\lfloor \frac{k_2 n}{u} \right\rfloor\ mod\ n
\end{aligned}$$

Since $u \gg 2n$, the ratio $\frac{u}{n}$ will be less than 1 and almost close to 0.
If k_1 and k_2 are chosen to be as close as possible, then because of the floor function, the expressions $\lfloor \frac{k_1 n}{u} \rfloor$ and $\lfloor \frac{k_1 n}{u} \rfloor$ will both be zero.

For example, Consider n = 10, u = 1000 and a = 42,
If Rory choose $k_1 = 57$ and Tiri choose $k_2 = 58$, then

$$h_a(k_1) = \left(\left\lfloor \frac{57 * 10}{1000} \right\rfloor + 42\right)\ mod\ n = (\lfloor 0.57 \rfloor + 42)\ mod\ n = 42\ mod\ n$$

$$h_a(k_2) = \left(\left\lfloor \frac{58 * 10}{1000} \right\rfloor + 42\right)\ mod\ n = (\lfloor 0.58 \rfloor + 42)\ mod\ n = 42\ mod\ n$$

Rory and Tiri are guaranteed to be roommates if they choose the closest possible values k_1 and k_2.

(c) Since the given hash function is universal, the probability of collision for any two keys picked from the possible values is atmost $\frac{1}{n}$.

$\therefore$ there is no such choice of k_1 and k_2 which can guarantee that Rory and Tiri will be roommates.
And the highest probability they could possibly achieve of being roommates is $\frac{1}{n}$

Problem 3-3.

(a) — *After looking at solution* —
Each string is $16 \lceil log_4(\sqrt{n}) \rceil$ characters long.
And 1 character is 8 bits long.
Maximum value of integer representation of string of size n will be $2^{16\lceil log_4 \sqrt{n} \rceil * 8} = O(n^{33})$.
Hence the range of integer values to sort is [0, n^{33}].
Radix sort is efficient here with time complexity $O(n + 33n)$

(b) Here $u = 800,000$.
If we choose $n = 10$ and $c = 6$, then $u < n^c$.
Radix sort will be efficient with time complexity $O(n + cnlog_n n) = O(n + cn)$.

(c) Given, $thickness = \frac{m}{n^3}$
with range [0, 4].
If thickness is multiplied by n^3, then range will be $[0, 4n^3]$.
And since $u = 4n^3 < n^c forc = 4$, Radix sort will be efficient sorting algorithm with time complexity $O(n + cnlog_n 4n) = O(n + cn(log_n 4 + 1))$.

(d) Since comparisons are allowed in $O(1)$ time and the most efficient sorting algorithm in Comparison model is Merge sort.
$\therefore$ Merge sort will be efficient with time complexity $O(nlgn)$.

Problem 3-4.

(a)

1. Build a hash table H with keys as $b_i \in B$ and values as $i \in [0, n-1]$.
2. Run a loop over B with loop variable $i = [0, n-1]$.
3. find $r - b_i$ in H:
   - if found, return $|i - H[r - b_i]| < \frac{n}{10}$.
   - else, return false.

Building a hash table takes $O(n)$ expected time.
Looping over B takes $O(n)$ time.
$find(r - b_i)$ takes O(1) expected time.

$\therefore$ The above algorithm requires $O(n)$ expected time.

(b) Since $r < n^2$. And $b_i >= r$ are useless.
Discard all the $b_i \in B\ such\ that\ b_i > r\ \forall\ i \in [0, n-1]$, and for remaining, replace $b_i\ with\ (b_i, i)$ to keep track of original indices.

Consider the new range of B to be $[0, k-1]$.
Using Radix sort to sort B. $[\because k < r < n^2]$.

Using two-finger algorithm:
Initialize two variables $s = 0$ and $t = k - 1$.
There are three cases:

1. $b_s + b_t == r$
   If $|i - j| < \frac{n}{10}$, return true, where i and j are corresponding second tuple entries of b_s and b_t respectively.
2. $b_s + b_t < r$
   Increment s by 1 because there cannot be any lower value of b_s which satisfies $b_s + b_t = r$, since b_t is maximum.
3. $b_s + b_t > r$
   Decrement t by 1 because there can be any lower value of b_t which satisfies $b_s + b_t = r$, since b_s is minimum.

if $s == t$, return false.

Radix sort requires $O(n + 2n)$ worst-case time for sorting.
Running two finger algorithm over B requires $O(n)$ worst-case time.

$\therefore$ The above algorithm requires $O(n)$ worst-case time.

Problem 3-5.

(a) Number of contiguous substrings of A = $|A| - k + 1$

For all the contiguous substrings of A, there can be a frequency table.
Then, a hash table can be built,
with $<$ **key** $= frequency\, table,$ **value** $= Number\, of\, substrings\, with\, same\, frequency\, table >$.

To build the frequency table for first substring of A, a loop can be used to build the frequency table in $O(k)$ time.
Then to build subsequent frequency tables for other substrings of A, we can keep track of the last frequency table and just decrement value of alphabet at index $i - 1$ and increment value of alphabet $i + k - 1$ in previous frequency table in $O(1)$ time.
where, i is the starting index of current substring and $i + k - 1$ is the ending index of current substring.

After building each frequency table, check if it exists in the hash table. If it does, then increment its value by 1, otherwise add the frequency table to the hash table as key and corresponding value 1.

Performing the operation for given B of size k:
Create a frequency table of B in $O(k)$ time. and check whether the frequency table exists in hash table in $O(1)$ expected time. If it exists, return its value.

To build all the frequency tables and storing their counts in hash table takes $O(|A|)$ expected time.
The operation to count the anagram substring count of B takes $O(k)$ expected time.

(b) Build the data structure named H for string T in $O(|T|)$ time.
Run a loop over S and for each $s_i : i \in [0, n)$, find a_i in $O(k)$ expected time.
Running the loop n times implies that building array $A = (a_0, ..., a_{n-1})$ will require $O(nk)$ expected time.

$\therefore$ Total time to run the algorithm will be $O(|T| + nk)$.

(c)
```
def count_anagram_substrings(T, S):
    '''
    Input:  T | String
            S | Tuple of strings S_i of equal length k < |T|
    Output: A | Tuple of integers a_i:
              | the anagram substring count of S_i in T
    '''
    A = []

    k = len(S[0])
    get_idx = lambda alpha: ord(alpha) - ord('a')

    # initialize hash table to store frequency tables and their counts
    H = {}

    # variable to hold the last frequency table
    prev_ft = None
    for i in range(len(T) - k + 1):
        # build the first frequency table by looping over substring
        # in O(k) time
        if i == 0:
            ft = [0] * 26
            for j in T[i:i+k]:
                ft[get_idx(j)] += 1
        # build other substrings using previous frequency table
        # in O(1) expected time
        else:
            ft = list(prev_ft)
            ft[get_idx(T[i-1])] -= 1
            ft[get_idx(T[i+k-1])] += 1

        # convert list to tuple to make it hashable
        ft = tuple(ft)

        # incrment the count of frequency table in the hash table
        # or add it if doesn't exist
        if ft in H:
            H[ft] += 1
        else:
            H[ft] = 1

        # set current frequency table as previous frequency table
        # for next substring
        prev_ft = ft

    for s_i in S:
        # build frequency table for s_i
        ft = [0] * 26
        for j in s_i:
            ft[get_idx(j)] += 1

        ft = tuple(ft)

        # find its value in data structure and append to A
        A.append(H[ft])

    return tuple(A)
```