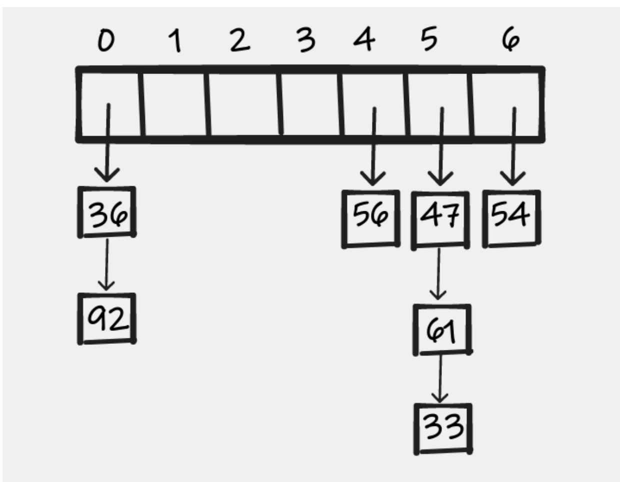
Problem Set 3

Name: Your Name

Collaborators: Name1, Name2

Problem 3-1.

(a) 

.

(b) 13 as a brute force solution.

Problem 3-2.

(a) they should choose their IDs such that $|(ak_1 + b) - (ak_2 - b)| = cn$ where c is any positive integer numbere which implies that $k_1 \equiv k_2 \mod n$,so it's guaranted.

(b) the privious approach will not work as we devide by n, but this division helps to make adjacent k_1 and k_2 collide, so it's guaranted if $k_1 - k_2 < 2$.

(c) the probability of collisions in this case is $1/n$, then if $n > 1$ it's not guaranted.

Problem 3-3.

(a) Let's assume that each string is stored sequentially in a contiguous chunk of memory, in an encoding such that the numerical representation of each charac- ter is bounded above by some constant number k (e.g. 127 in case of ascii code), where the numerical representation of one character c_i is smaller than that of another character c_j if c_i comes before c_j in the English alphabet. Then each string can be thought of as an integer between 0 and u = $k^{64\log_4 n} = O(n^{64\log_4 k}) = O(n^{O(1)})$,stored in a constant number of machine words, so can be sorted using radix sort in worst-case $O(n + n\log_n n^{O(1)}) = O(n)$ time.

(b) as the number of years is bounded by some range we can use radix sort to sort it in $O(n)$ time.

(c) multiply all numbers by n^3, then we have integers in range $[0, 4n^3]$, use radix sort to sort them in $O(n)$ time.

(d) use *merge sort* to sort them in $O(nlogn)$ as we can only compare every pair to determine which one is older.

Problem 3-4.

(a) if the largest number of reams in a box is known (in $O(n)$) we can use radix sort to sort them (saving their original indicies in (b_i, i)) in $O(n)$ time then we can use *two-finger algorithm* as follows

- if $b_i + b_j < r$ increment the left pointer.
- if $b_i + b_j > r$ increment the right pointer.
- if $b_i + b_j = r$ check if $|i - j| < \frac{n}{10}$.
  - if $|i - j| = \frac{n}{10}$ then there is a close pair and algorithm terminates.
  - if $|i - j| \neq \frac{n}{10}$ there is no close pair, since which pointer moves $b_i + b_j \neq r$ as the array is sorted and **no two boxes contain the same number of reams** ans algorithm terminates.

if the largest number of reams is unknown we can hash them in $O(n)$ time saving their indicies like (b_i, i) and loop over the boxes and search for $(r - b_i)$ in the hash table in $O(1)$ and this takes $O(n)_{exp}$.

(b) the first approach in part **(a)** fits well in this case with a modification that evey box has number of reams $> n$ do not consider it in the new data structure $[(b_1, 1), (b_2, 2), \cdots]$.

Problem 3-5.

(a) we aim to structure a hash table by hashing evey k-length substring in A so we can search for it in $O(1)$, since A and K is given we can calcutale (the weight) of substring of length K as follows:

1. use **ASCII** code as the weight of a letter.
2. take the first k letter K and sum thier weights.
3. create a hash table by hashing the sum and insert it in form (S, i) where i is a counter to determine number of anagrams of this weight.
4. erase the first element in K and append the next letter in A to K and hash its sum of weights.
5. repeat step **(4)** till the end of A.

given string B with length k we can calcutale its sum in $O(K)$, search for it in the k hash table and return i in $O(1)_exp$.

(b) we can hash every (weight) of string S_i in $\boldsymbol{S}$ in $O(nk)$, calcutale the weight of $\boldsymbol{T}$ in $O(\boldsymbol{T})$ and search for it in the hash table in $O(1)$.

(c)
```python
def freq(str):
    arr = [0] * 26
    for c in str:
        arr[ord(c) - ord('a')] += 1

    return arr


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 = [0] * len(S)
    frq = [0] * 26
    hsh = {}
    k = len(S[0])
    for i in range(len(T)):
        frq[ord(T[i]) - ord('a')] += 1;
        if i > k - 1:
            frq[ord(T[i - k]) - ord('a')] -= 1;
        if i >= k - 1:
            key = tuple(frq) # tuple is a hashable type
            if key in hsh:
                hsh[key] += 1
            else:
                hsh[key] = 1



    for i in range(len(S)):
        arr = freq(S[i])
        key = tuple(arr)
        if key in hsh:
            A[i] = hsh[key]





    return tuple(A)
```