Problem Set 3

Name: Thomas Ren

Collaborators: None

Problem 3-1.

(a) For this part, I have written the code to implement this hash table. And the results shows as follows:

```
['(36)<->(92)', '', '', '', '(56)', '(47)<->(61)<->(33)', '(52)']
```

The specific code implemented are as follows:

```python
# Problem Session 3 -> Problem 3-1 Hash It Out
def Hash_Out(A, a, b, k):
    '''
    h(key) = (a*key+b) mod k
    :param A: a list of integer keys
    :param a: a in hash function h(key)
    :param b: b in hash function h(key)
    :param k: k in hash function h(key)
    :return: a hash table chaining with sequence i.e. list in Python
    '''
    HASH = [[] for i in range(k)]
    for item in A:
        new_item = (a*item+b)%k
        HASH[new_item].append(item)

    return HASH


# Problem Set 3 -> Problem 3-1
def Hash_Chaining_linked_list(A,a,b,n):
    '''
    :param A: a list of integer keys
    :param a: a in hash function h(key)
    :param b: b in hash function h(key)
    :param n: n in hash function h(key)
    :return: a hash table chaining with doubly linked list
    '''

    Hash_table = Hash_Out(A,a,b,n)
```

```
    chaining_hash_table = []
    for slot in Hash_table:
        # if slot is not None:
        chaining = Doubly_Linked_List_Seq()
        chaining.build(slot)
        chaining_hash_table.append(chaining.__str__())

    return chaining_hash_table
```

(b) If we consider hashing by hash function $h(k) = (10k + 4)\ mod\ c$, we would get c slots. Since the list A has **length 7**, we cannot fit all the items without collision for any $c < 7$.
Thereby, we will try to enumerate c starting from 7.
Here, we give the implementation as follows:

```
def Hash_Out_double_modulo(A, a, b, c, n):
    '''
    h(key) = (a*key+b) mod n
    :param A: a list of integer keys
    :param a: a in hash function h(key)
    :param b: b in hash function h(key)
    :param c: c in hash function h(key)
    :param n: n in hash function h(key)
    :return: a hash table chaining with sequence i.e. list in Python
    '''
    HASH = [[] for i in range(n)]
    for item in A:
        new_item = (a*item+b)%c%n
        HASH[new_item].append(item)

    return HASH


def collision(A,a,b,c,n):
    '''
    h(key) = (a*key+b) mod c mod n
    :param A: a list of integer keys
    :param a: a in hash function h(key)
    :param b: b in hash function h(key)
    :param c: c in hash function h(key)
    :param n: n in hash function h(key)
    :return: a boolean value tells whether a collision exists in our hash table
    '''
    Hash_table = Hash_Out_double_modulo(A,a,b,c,n)
    for slot in Hash_table:
        if len(slot)>1:
            print('collision exists for slot',slot)
            return False

```

```
    return True


def find_min_c(A,a,b,n):
    '''
    h(key) = (a*key+b) mod c mod n
    :param A: a list of integer keys
    :param a: a in hash function h(key)
    :param b: b in hash function h(key)
    :param n: n in hash function h(key)
    :return: c_min: the minimum value satisfying the non-collision situation
    '''
    c_min = n
    while True:
        if collision(A,a,b,c_min,n):
            print('The minimum c without causing collision is:',c_min)
            return c_min
        else:
            print('collision exists for c =',c_min)
            c_min += 1
```

The results for the case given in the problem is as follows:

```
collision exists for slot [36, 92]
collision exists for c = 7
collision exists for slot [36, 52, 56, 92]
collision exists for c = 8
collision exists for slot [61, 52]
collision exists for c = 9
collision exists for slot [47, 61, 36, 52, 56, 33, 92]
collision exists for c = 10
collision exists for slot [52, 92]
collision exists for c = 11
collision exists for slot [56, 92]
collision exists for c = 12
The minimum c without causing collision is: 13

Process finished with exit code 0
```

Problem 3-2. For simplicity, we assign r_1 to be the room ID of k_1 and r_2 to be the room ID of k_2.

(a) This could work if $k_1 \equiv k_2 \; mod \; n$.
Then, we would have $r_1 = (ak_1 + b) \; mod \; n$.
Since $k_1 \equiv k_2 \; mod \; n$, we can write $k_2 = sn + k_1$, for some constant s.
Then, we could write $r_2 = ak_2 + b \; mod \; n = a(sn + k_1) + b \; mod \; n = asn + ak_1 + b \; mod \; n = ak_1 + b \; mod \; n = r_1$

(b) Since u is a really large number, so if k changes for a small value relative to u, the results won't change.
For example, if $k_2 = k_1 + 1$, then $r_1 = r_2$, except for the threshold.
So, if we choose k_1 to be 0 and k_2 to be 1. Then $r_1 = r_2 = 0$.
Thereby, they will be assigned to the same room.

(c) This hash function is known to be the **universal hash family**. According to the lecture, the probability of any two different values being assigned to the same hash slot is at most $\frac{1}{n}$.
Thereby, their highest probability of being roommates is $\frac{1}{n}$.

Problem 3-3.

(a) For each each ice core, we get a **unique** identifier.
The first thought comes to our mind should be using **Direct Access Array Sort**, since there is no collision exists. However, **Direct Access Array Sort** takes $O(u)$ time to run, here, $u = 128^{16\lceil \log_4(\sqrt{n}) \rceil}$ for which we assign a unique value for an ASCII character. Then, we would deduce that $O(u) = O(n^c)$, for some constant c. This is clearly worse than comparison model e.g. merge sort which only takes $O(n \log n)$ time.
Since we get that $u = O(n^c)$, for some constant c, we would know immediately that we can achieve linear complexity by using **Radix Sort**. The running time is $\Theta(n + n \log_n u) = \Theta(n + cn) = \Theta(n)$

(b) Since the here is a bound for u that $u = O(800,000)$, we don't need **Radix Sort** to achieve linear running time.
Here, we will use **Counting Sort** because there is no promise for no collision. Still, we will get a sorted array in $\Theta(800,000 + n) = \Theta(n)$.

(c) Multiply every key of thickness by n^3 to get all keys to be integers i.e. m which ranging from 0 to $4n^3$.
Then use **Radix Sort** to sort integer keys m takes $\Theta(n + n \log_n 4n^3) = \Theta(n)$.

(d) This part of problem limit our algorithm to **Comparison Models**, which gives best case $\Theta(n \log n)$ running time.
To be more specific, we will use **Merge Sort**, which has "two-finger algorithm" for $merge$ operation.

Problem 3-4.

(a) Since this part requires **expected** $O(n)$-time, we build a **Hash Table** in **linear time** so that we can achieve certain item by its key in **constant time**.
The algorithm is as follows:

```
for b_i in B:
    b_j = find(r-b_i)
    if find(r-b_i) is None:
        return False
    else:
        if |j-i| <= n/10:
            return True
        else:
            pass
```

(b) Since $r < n^2 \Rightarrow r = O(n^2)$, we can use **Radix Sort** to first sort the B in **linear time**. Then we will use the two-finger algorithm to approach our best pair sum t to target r, after that, determine whether they are **close pair**, if so, return the pair, else, continue moving our two fingers.
The algorithm is as follows:

```
# delete all the items that are larger than r
for i in range(len(S)):
    if S[i] >= h:
        del S[i:]
        break
# two-finger algorithm
i,j = 0,len(B)-1
target_i, target_j = i,j
while j - i > 1:
    pair_sum = S[i] + S[j]
    if pair_sum > h:
        j -= 1
    elif pair_sum < h:
        i += 1
    else:
        if abs(s[i].key-s[j].key)<len(B)/10:
            return s[i].key,s[j].key
        else:
            pass
```

Problem 3-5.

(a) Since we need to return the anagram substring count of B in A in $O(k)$-time, we cannot traverse the whole set of A, which is $O(n)$-time, we should, instead, give each substring a key, which is the same for anagrams, so that we can determine whether B exists in A and how many or not.
We solve this by giving the key of each substring a frequency table of alphabets, which is implemented in Python using **tuple** data structure. This frequency table should be hashable as well, by which we mean immutable.
Then we collect all the frequency table to a hash table, where the key is the frequency table and the item is the number of this frequency table, we implemented this hash table using **dictionary** in Python.
Then, we take $O(|A|)$-time to build the **dictionary** of substring table by shift one space and synchronize the frequency table in $O(1)$-time. For each B, we build the frequency table of B in $O(k)$-time, then we can find the number of anagram of B in $O(1)$-time in the hash table.
The Python code is implemented as follows:

```
def single_count_anagram_substring(A,B):
    '''
    :param A: a string
    :param B: a substring with length less than A
    :return: the number of anagram substring B in A
    '''
    k = len(B)
    substring_table = {}
    for i in range(len(A)-k+1):  # O(|A|+k) = O(|A|)
        if i == 0:
            substring = A[:k]
        else:
            substring = substring[1:]+A[i+k-1]
        frequency_table = build_frequency_table(substring)
        if frequency_table in substring_table:
            substring_table[frequency_table] += 1
        else:
            substring_table[frequency_table] = 1

    B_frequency_table = build_frequency_table(B)
    if B_frequency_table not in substring_table:
        print('There is no anagram substring of B in A')
        return 0
    else:
        return substring_table[B_frequency_table]




def build_frequency_table(str):
```

```python
    '''
    :param str: a string
    :return: a tuple of frequency table of alphabets
    '''
    alphabet = [0 for i in range(26)]
    for letter in str:
        index = ord(letter) - 97
        alphabet[index] += 1

    return tuple(alphabet)
```

(b) We can first build the substring hash table in $O(|T|)$-time and for each string in S, we do the aforementioned step to find the number of anagram of this string, which takes $O(k)$-time. For traversing the whole S, takes $O(n)$-time, thus the whole process takes $O(|T| + nk)$-time.
The Python implementation code is as follows:

```python
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 = []
    ##################
    # YOUR CODE HERE #
    ##################
    k = len(S[0])
    n = len(S)
    A = [0 for i in range(n)]
    substring_table = {}
    for i in range(len(T) - k + 1):  # O(|A|+k) = O(|A|)
        if i == 0:
            substring = T[:k]
        else:
            substring = substring[1:] + T[i + k - 1]
        frequency_table = build_frequency_table(substring)
        if frequency_table in substring_table:
            substring_table[frequency_table] += 1
        else:
            substring_table[frequency_table] = 1

    for index,s in enumerate(S):
        s_frequency_table = build_frequency_table(s)
        if s_frequency_table in substring_table:
            A[index] = substring_table[s_frequency_table]
        else:
            A[index] = 0
```

```
    return tuple(A)
```

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