COMP3121 ASSIGNMNET 1

Jingjie Jin

Z5085901

Question 1:

(a).

Sort the array S using Merge-Sort. Sorting takes time O(nlogn). For each $y \in S$ separately use binary search to check if integer $(x - y)$ exists in S. Binary search takes time O(logn) and is executed n times. Thus, the total time is O(nlogn).

Merge-Sort(S)

Get the sorted array S.

Input: array S sorted in ascending order; required sum integer x

Output: return True if there exist two numbers in S whose sum is exactly x.

return False otherwise.

Function:

```
i =0, j = n-1
while i <= j:
    if S[i] + S[j] == x, then:
        return True;
    else if S[i] + s[j] > x, then:
        j--;
    else:
        i++;
return False
```

(b).

Use HashSet to catch the certain pair of two elements in S whose sum is x. Then the problem can be solved by scanning the list S in one time. Note that the size of S is n, so the algorithm use time O(n).

Input: array S; required sum integer x.

Output: return True if there exist two numbers in S whose sum is exactly x.
return False otherwise.

Function:

```
define HashSet set;
for i in S:
    diff = x – S[i];
    if set.contains(S[i]), then:
        return True
    end if;
    set.add(diff);
return False;
```

Question 2:

Sort the array S using Merge-Sort. Sorting takes time O(nlogn). Then use AVL tree to find L and R. Each query takes time O(logn) and is executed n times. Thus, the total time is O(nlogn).

Merge-Sort(S)

Get the sorted array S in ascending order

Input: array S sorted, integer L, integer R. (Assume that L < R)

Output: n numbers of elements of the array have value between L and R.

Function findLeftBoundary:

```
if (start == end) return end;
mid = (start + end) / 2;
if S[mid] < L:
    return findLeftBoundary(mid+1, end, L);
else:
    return findLeftBoundary(start, mid, L);
```

Function findRightBoundary:

```
if (start == end) return end;
mid = (start + end + 1) / 2;
if S[mid] > R:
    return findRightBoundary(start, mid-1, R);
else:
    return findRightBoundary(mid, end, R);
```

Function ResultCount:

```
int[] result;
for (int i = 0; i < n; i++):
    leftBoundary = findLeftBoundary(0, n-1, L);
    rightBoundary = findLeftBoundary(0, n-1, R);
    count = rightBoundary – leftBoundary;
    result.add(count);
```

Question 3:

As Cantor`s diagonal argument proves, I can support the teams in a suitable subset that differs from each other friend.

Thus, I just need ask friends for their support one by one.

"Does friend i support team i ?"

Function:

(F: set of Friends T: set of Teams)

for i in N:

if F[i] supports T[i]:

I do not support T[i];

else:

I do support T[i].

For example:

F0 = (**0**,0,0,0,0,0,0,...)

F1 = (1,**1**,1,1,1,1,1,...)

F2 = (0,1,**0**,1,0,1,0,...)

F3 = (1,0,1,**0**,1,0,1,...)

F4 = (1,1,0,1,**0**,1,1,...)

F5 = (0,0,1,1,0,**1**,1,...)

F6 = (1,0,0,0,1,0,**0**,...)...

ME =(**1**,**0**,**1**,**1**,**1**,**0**,**1**,...)

Question 4:

Divide interval [0,1] into n equal buckets:

[0, 1/n], [1/n, 2/n], ... [(n-1)/n, 1]

Then put each xi to a bucket (could be the same bucket).

Now prove that it satisfies that $\sum_{i=2}^{n} |y_i - y_{i-1}| < 2$

The result could be splitted into two parts:

1. y_i and y_{i-1} are in the same bucket.

   In this condition, the largest value of $\sum_{i=2}^{n} |y_i - y_{i-1}|$ not greater than the average size of each bucket, which is 1/n. Assume that there are n-1 times $|y_i - y_{i-1}|$ meet the largest value. Thus, $\sum_{i=2}^{n} |y_i - y_{i-1}|$ = (n-1)/n < 1.

2. y_i and y_{i-1} are in the different buckets.

   In this condition, the largest value of $\sum_{i=2}^{n} |y_i - y_{i-1}|$ is the sum from adjacent intervals in [0,1].

Finally, we can obtain the result with the sum of $\sum_{i=2}^{n} |y_i - y_{i-1}|$ is smaller than 1+1=2.

Question 5:

(a).

Ask a person A if he/she knows another person B.

Function Knows(A,B):

1. If A knows B, then A cannot be celebrity. Discard A, and B may be celebrity.
2. If A does not know B, then B cannot be celebrity. Discard B, and A may be celebrity.

Repeat above two steps regarded as function Knows(A, B) till we left with only one person.

Ensure the remained person is celebrity.

We can use stack to verity celebrity.

1. Push all the persons into a stack.
2. Pop off top two persons from the stack, discard one person based on return status of Knows(A, B).
3. Push the remained person onto stack.
4. Repeat step 2 and until only one person remains in the stack.
5. Check the remained person in stack does not know anyone else.

The elimination stage requires exactly n−1 questions, since each question reduces the size on the list by 1.

Next, in the verification phase, we ask the remained person n-1 questions, and we ask the other n-1 people one question. This phase requires at most 2(n-1) questions. Thus, the total times is 3(n-1).

(b).

It is possible to save an additional ⌊log2 n⌋ questions in the verification phase by not repeating we have already asked in the elimination phase. By maintaining the elements in a queue, the celebrity is involved (i.e. either asked or asked about) at least ⌊log2 n⌋ questions during the elimination phase.

Question 6:

(a).

We can use the idea of undirected graph to calculate the degrees of each vertice (student). Every student' name must be written at least once representing the voter themselves. Thus, after getting the degrees, we minus 1 on the each number.

For example:

There are 5 students in the class and the five voting pieces writing with:

(AB), (BA), (AC), (AD), (BE)

Degrees:

A: 4; B: 3; C: 1; D: 1; E: 1

Minus 1 to get the counts of votes each student received:

A: 3; B: 2; C: 0; D: 0; E: 0;

(b).

We can check the votes of students whose name has been appeared only once. Like the example given in problem(a). C, D, E are the students we need. Then we look over the pieces their names written, and another names in the pieces are they voted for.

For example:

C voted for A; D voted for A; E voted for B

(c).

Consider that the idea of directed graph. Each vertice has only one directed edge to another vertice or itself (Each student can vote for only once). This means that there are n out-degrees in this directed graph.

In a directed graph,

$$|E| = \sum_{v \in V} deg^-(v) = \sum_{v \in V} deg^+(v).$$

Thus, number of edges = in-degrees = n

Note that each student received at least one vote, and there are n students, so the maximum possible number of votes received by any student is also n/n = 1.

(d).

$O(n^2)$:

First, we know that each student has received only one vote from above problems. This means that each name of students was written totally twice in the pieces of paper, so we can distinguish their identities (voter /candidate) by elimination.

Create an empty list, and traverse through each vote. In each traverse, we choose one of the names in each vote and check whether it occurs in the list or not. After checking and assignment of identities, the pairs of votes are all added in the list, and could match up the rules that problem gives.

Pseudo Code:

```
Create a large list named votes containing the pairs of votes.
Create an empty list.
For vote in votes:
    If list.has(name1):
        If name1 = voter in list:
            name1 = candidate in current vote;
            name2 = voter in current vote;
            list.add( (name2, name1) ).
        Else:
            name1 = voter in current vote;
            name2 = candidate in current vote;
            list.add( (name1, name2) ).
    Else:
        name1 = voter in current vote;
        name2 = candidate in current vote;
        list.add( (name1, name2) ).
```

Querying the name1 in list each time takes (n-1) times, and the traversing takes n times.

Thus, the total time is $0+1+2+\cdots+n-1 = n(n-1)/2 = O(n^2)$.

A slightly improvement to get **O(n)**:

Modify the query function **list.has()** by using **hashtable**.
Through (key, value) in hashtable, we can target the existing pair and required name quickly taking time O(1).
Thus, the total time can be reduced to O(n).

END