z5135897
Mingkai Ma

1.
(a)

1) Sort the array using merge sort
2) Set two pointers: left and right, left = 0, right = length of array – 1
3) While left < right:
    a. If(array[left] + array[right] == sum) then return left and right
    b. Else if(array[left] + array[right] > sum) then right = right – 1
    c. Else left = left + 1 (In this case array[left] + array[right] < sum)

4) if left == right:
    not found

Merge Sort(described in the lecture slides):

```
Merge_Sort(A, p, r):
    if(p < r):
        Then q = floor( (p + r) / 2)
        Merge_Sort(A, p, q);
        Merge_Sort(A, q + 1, r)
        Merge(A, p, q, r)
```

Analysis: The time complexity of merge sort is O(n log n), after sort, the while loop takes O(n),
So the total time complexity is O(n log n).

(b)
In order to accomplish the same task in O(n), we need auxiliary space.
We use a hash table to store the index information of the elements in array.

1) initialize a hash table, here using C++ map: unordered_map<int, int> hash_table
   the key in hash table is the value of the element, the value in hash table is the index of the element

2) for i = 0 to length of array – 1 do:
    calculate the other addend, a1 = sum – array[i]
    if hash_table contains a1 then return a1 and array[i]
    else add hash_table: hash_table[array[i]] = i

3) if not return after the end of the for loop, then not found.

Analysis:
Time complexity: O(n), because we traverse the array only once. The look up operation of hash table takes O(1) time.

Space complexity: O (n), We need extra space for the hash table.

2.
1)Sort the array using merge sort
2)Get two indices i1 and i2, using two modified binary search according to L and R, then return i2 – i1 + 1

Merge Sort(described in the lecture slides):

```
Merge_Sort(A, p, r):
        if(p < r):
                Then q = floor( (p + r) / 2)
                Merge_Sort(A, p, q);
                Merge_Sort(A, q + 1, r)
                Merge(A, p, q, r)
```

Binary Search for low bound:

1) set left = 0, right = length of array – 1;
2) while left <= right:
    a. set mid = (left + right) / 2
    b. if(array[mid] >= L) then right = mid – 1
    c. else then left = mid + 1
3) return left

Binary Search for upper bound:

1) set left = 0 , right = length of array – 1;
2) while(left <= right)
    a. set mid = (left + right) / 2
    b. if(array[mid] <= R) then left = mid + 1
    c. else then right = mid – 1
3) return right

Analysis: the time complexity of merge sort is O(n log n), the time complexity for the two binary search is O(log n), so the total complexity is O(n log n)

3.

Ask the nth friend if he or she supports nth team, then do the opposite thing, which means if n-th friend supports the n-th team, then we will not support this team, while if n-th friend doesn't support the n-th team, we will support this team, by asking N questions like this, we can get the subsets of teams that we support.

4.

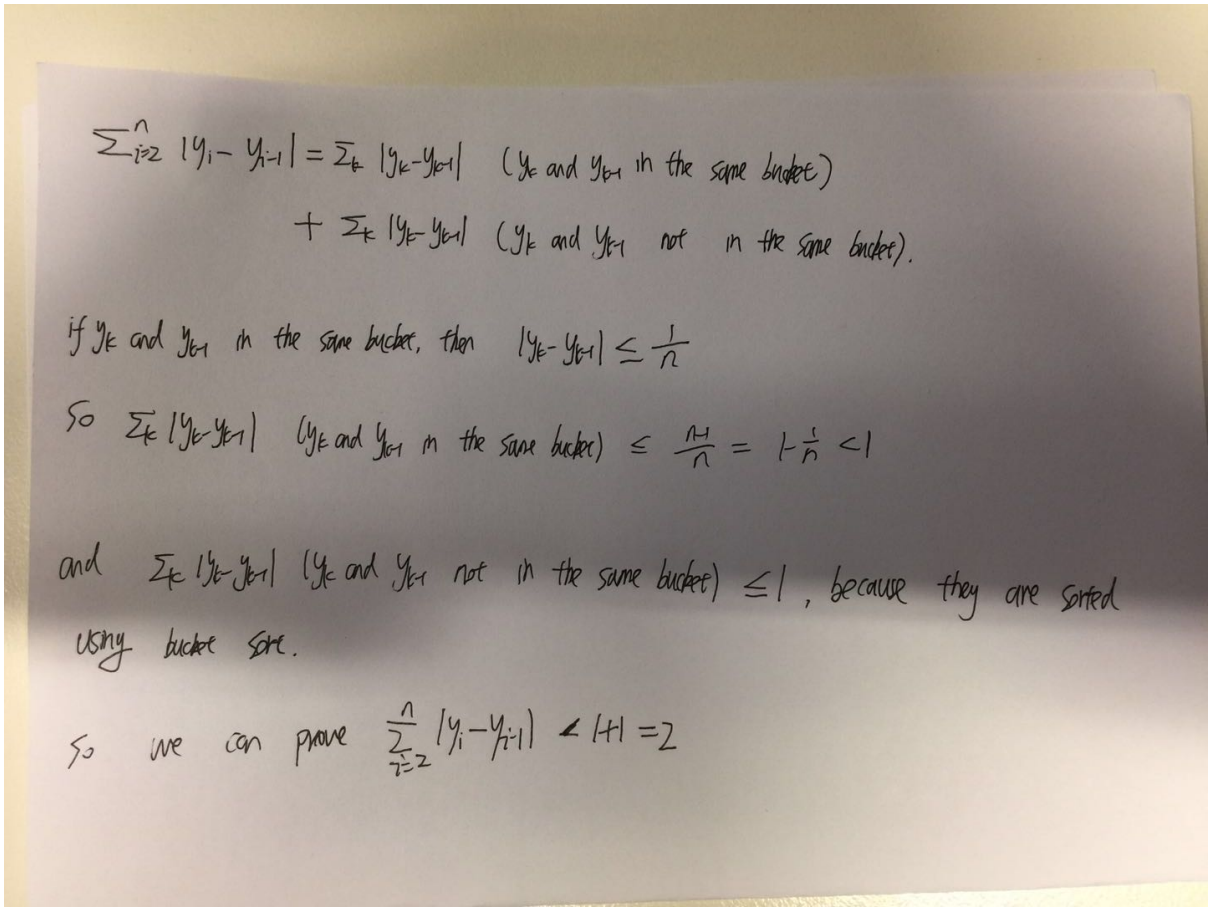
Split the interval [0,1] into n equal buckets.
Set pair like (xi, B(xi))

B(xi) = floor(n * xi)

B(xi) is a bucket, different xi may have same B(xi), which means different xi may belong to same bucket.

We get these pairs and then sort these pairs according to B(xi) using counting sort, which takes O(n) time.

In each bucket, we change the sequence of numbers in the bucket, find the minimum element and put the minimum element as the first element, and find the maximum element and put the element as the last element. Find minimum element and maximum element only takes O(n) time.
So



By using modified bucket sort, we can finish the task.
Summary:
Using bucket sort:

1) Split the interval [0,1] into n buckets.
2) Sort these buckets using counting sort, which takes O(n) time.
3) In each bucket, put the minimum element at the first position, put the max element at the last position, these takes O(n) time.

5.

(a)

We find the celebrity based on the following rules:

1) If X knows Y, then X can't be the celebrity, we eliminate and discard X.
2) If X doesn't know Y, then Y can't be the celebrity, we eliminate and discard Y.
3) Repeat step 1 and 2 until there is only one left.
4) Check the only left one knows or known by others.

We use a data structure : stack, to perform the task.

The function Ask(X, Y) is the question we are going to ask.

1) push all people into the stack
2) pop two persons from the stack, X and Y, Ask(X, Y). Based on the rule we describe above, for example, if X knows Y, we eliminate and discard X, and push back Y into the stack.
3) Repeat step 1 and 2 until the size of stack is one, which means there is only one person left in the stack.

4) Set the left person as Z.
5) for i = 1 to n - 1:
   Ask(Z, i) and Ask(i, Z)

Now let's count the Ask times in the worst cases:
In the step 2,we need to ask n – 1 times, in the step 5, we need to ask 2 * (n - 1). Therefore in worst cases, the total time we need to ask is (n - 1) + 2 * (n - 1) = 3 * (n - 1).

(b)
If we use a queue not a stack, the queue is first in first out. We can save $\lfloor \log_2 n \rfloor$ times to ask questions. In step 5 above, we don't need to repeat any questions we already asked in step 2. By doing so, we can make a little improvement from 3 * (n - 1) to $3n - \lfloor \log_2 n \rfloor - 2$.

6.
(a) Count how many time a name appears in all votes, get the result, R, then the student whose name is this received R – 1 votes.
For example: there are 3 student, A, B, C. All votes are: AB, BA, CB. A appears 2 time, so student A received 2 – 1 = 1 votes, B appears 3 times, so student B received 3 – 1 = 2 votes, same reason, student C received 1 – 0 = 0 votes.

(b) If there is a name that only appears once in all votes, then we can know that this student did not receive any votes. For example, the student C above. After we know C didn't receive any votes, we can easily know he or she voted for B, because C only appears in one vote: CB, we know that C voted for B.

(c)If every student received at least one vote, then the maximum possible number of votes received by any student is just 1.
Assume there is one student, received m votes(m > 1), so we can get the total votes are : totalVotes = n – 1 + m
Because m > 1, so n – 1 + m > n, but we know that n students can have exactly n total votes, contradiction.
Therefore, the maximum possible number of votes in this condition is 1.

(d)
see next page:
1) firstly, we count number of appearance of each name into a hash table, the key is the name, the value is the number of appearance – 1
2) If we find that each value in the hash table of each key is 1, then this means everyone received one vote. In this case, loop through all votes, every time we get a vote, construct X voted for Y, X and Y are the names in the vote paper.
3) If we find there are some values in the hash table of some keys is 0, then start from these zero values, we loop through all votes, if name X(hash_table[X] = 0) in vote, then construct X voted for Y.
4) Now deal with those votes that have not been constructed. Choose randomly one votes, constructed X voted for Y , X and Y in this vote. Then among the left votes, find the vote which contains name Y, construct Y voted Z, Y and Z in this vote, repeat until no votes left.

Example:
AB, BC, CD, DB, EB.

1) firstly we construct the hash table M described above. M[A] = 0, M[E] =0; M[B] = 3, M[C] = 1, M[D] = 1;
2) Because M[A] and M[E] equals to 0, so find the votes contains A and E, construct A voted for B and E voted for B;

3) Left votes: BC, CD, DB. We randomly choose CD, construct C voted for D, then find vote DB, because it contains D, construct D voted for B, then there is only one left BC, we construct B voted for C.