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:
   1. If(array[left] + array[right] == sum) then return left and right
   2. Else if(array[left] + array[right] > sum) then right = right – 1
   3. Else left = left + 1 (In this case array[left] + array[right] < sum)

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: 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

1. for i = 0 to length of array – 1 do:

calculate the the other addend, a1 = sum – array[i]

if hash\_table contains a1 then return a1 and array[i]

else update hash\_table: hash\_table[array[i]] = i

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:
   1. set mid = (left + right) / 2
   2. if(array[mid] >= x) then right = mid – 1
   3. 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)
   1. set mid = (left + right) / 2
   2. if(array[mid] <= y) then left = mid + 1
   3. 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.

Because it is independent for everyone about their own subset, in order to get our own special subset, we need to know every other people’s subset.

Firstly we need to know every other people’s subset, and the total of them is set S.

Then we get all subset of the N teams, let’s say this set is R.

Then we can pick any subset from R \ S.

Sulo2:

1. push N teams into an array, create a hash table, M, the key is people, the value is their support subset
2. for i = 0 to N

ask everyone if they support array[i]:

1. if none of them support array[i]:

return array[i];

1. if all of them support array[i]:

return a team or some teams exclude array[i];

1. if some of them support and some of them don’t support:

store information into the hash table M;

continue;

1. from step2- c), we can know if there is someone who support all teams, if there is nobody support all teams, we choose all teams as our subset then return;

If there is someone support all teams, we check whether there is someone who support 0 teams. If there is nobody support 0 teams, we choose empty subset for our subset then return;

1. Set S = {N teams}, then we loop through the hash table

For key = key1 to keyN in M:

If S \ M[key] not in M:

return S \ M[key];

4.

Split the interval [0,1] into n equal buckets.

Set pair like (xi, B(xi))

B(xi) = floor( n \* xi)

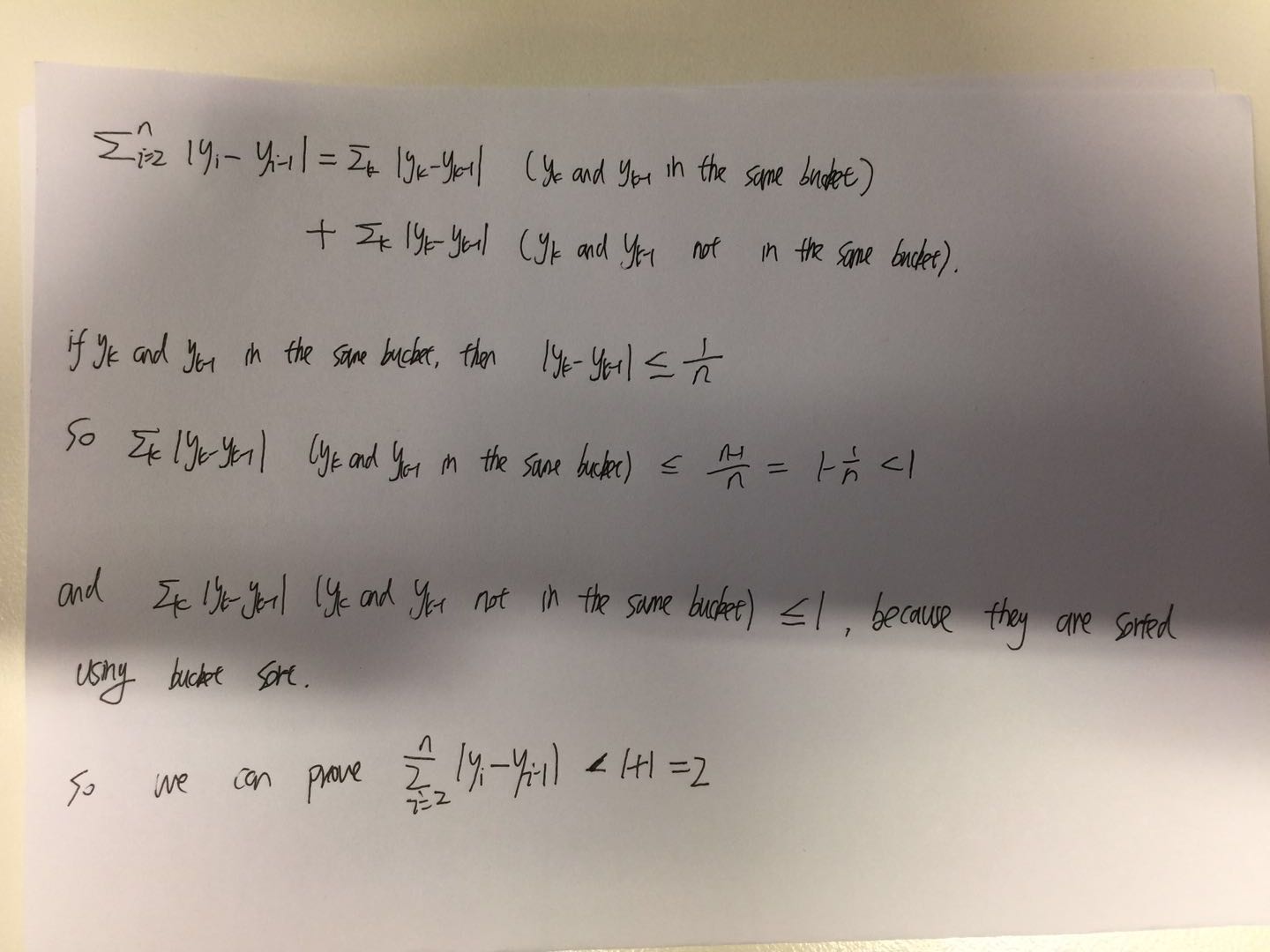
B(xi) is a bucket, different xi may have same B(xi), which means different xi may belong to same bucket.

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.