Question 2

Divyansh Mathur, 210357

*Simplifying the question:* Given an array of n integers we have to find the minimum value of j-i where 0<=i<=j<n and A[i]+A[i+1]….A[j] >=P

The cost will be C x ans from above question.

**a.)** **To Design an algorithm in O(nlog(n)) time complexity for determining the minimum cost room allocation**

*Designing Algorithm:* Since the maximum value which can be achieved is n-1 and the minimum value that can be achieved is 0, we can apply binary search on this range to see for which j-i the room allocation is possible.

*Checking whether room allocation is possible for given* j-i: check for all sums:

A [0] +A [1] + ….. A[j-i]

A [1] +A[ 2]+ …..A[j-i+1]

… and so on.

If any of this sum is greater than or equal to P than this j-i can be ans and hence we return true.

We will set a low pointer to 0 and a high pointer to n-1 and check for mid element.

If this mid element satisfies the condition then record this ans and move to the left half of the segment i.e. make high=mid-1

Else if this element does not satisfy the condition then move to the right segment in search of ans i.e. make low=mid+1

*Checking the condition in O(n) time:*

Precompute the prefix sum of the Array such that S[i]= A[0]+A[1]+…..A[i]

Therefore sum A[i]+A[i+1]+…A[j]= S[j]-S[i]+A[i]

For each i from 0 to n-mid-1 find the maximum value of S[i+mid]-S[i]+A[i]. If it is greater than P return true. This takes O(n) time

*Overall Time Complexity:*

O(nlogn)

*Pseudocode*

Create\_prefix\_Sum(S[n])

low=0, high=n-1

while(low<=high){

    mid=(low+high)/2

    if(check(mid)==true) {

        ans=mid

        high=mid-1

    }else {

        low=mid+1

    }

}

Create\_prefix\_Sum(S[n]){

    S[0]=A[0]

    for i from 1 to n-1{

        S[i]=S[i-1]+A[i]

    }

}

check( mid){

    if(S[i+mid]-S[i]+A[i]>=P){

        return true

    }else {

        return false

    }

}

**b.)** **To Design an algorithm in O(n) time complexity for determining the minimum cost room allocation**

*Designing Algorithm:* We want the smallest length subarray such that its sum is greater than P. Since the capacity of a room cannot be negative, **if we take a sum of subarray, it will always be greater than if we left some front or rear elements**.

We first set our pointers low and high to 0. So, we first find an index high such that sum of all elements from A[low] to A[high] >= P. Once we find the high, we move the low forward and reduce the sum until this sum value becomes less than P. At each iteration we check if the condition is valid, and we check if this (high-low) is the minimum of all plausible cases. Once sum is less than P we again increase high until sum exceeds or equals P

*Overall Time Complexity:* We are approximately visiting every element almost 2 times. T(n)= 2n

Therefore O(n) time

*Pseudocode:*

low =0, high=0, sum=0, ans=infinity {a very high value, maybe INT\_MAX}

while(high<n){

    while(sum<P && high<n){

        sum+=A[high]

        high++

    }

    while(sum>=P){

C[low]=high-low

        ans= min(ans,high-low)

        sum-=A[low]

        low++

    }

}

**c.)** **To Design an algorithm in O(nlog(n)) time complexity for determining the maximum number of contiguous rooms they can get which satisfy the beauty constraints**

*Simplifying the Question:*  We are given an array of n positive integers, we need to find length of maximum contiguous subsequence such that GCD of that subsequence is >=k.

*Designing Algorithm:* If we are able to find GCD for a given sequence A(i,j) in constant time then we can apply binary search on answer.

*Binary Search:* So the maximum length of such subsequence can be n, and the minimum length can be 1 (assuming at least one integer is >=k) , so we need to apply binary search between 1 to n, this takes O(log n ) time.

*To check whether a length is valid length or not:* If them given length is non 0, then we check the every possible index whether it is the starting point of such contiguous subsequence. Now to suit to our overall time complexity this should take maximum of O(n) time. So, we need to find GCD of a subsequence (contiguous) in O(1) time.

*Find GCD of a subsequence (contiguous) in O(1) time:* To do this we can use concept of Range minima Query and range minima data structure. We make a n x log (n) matrix M, such that M[i][j] indicates the GCD of all the elements from i to i+2j  (maximum value of j will be log n, concepts of range minima).

*Preprocessing:* To make the M matrix we need extra space of n log n and extra time of O(nlogn) but it is a onetime thing and our overall time complexity will still remain O(nlogn). To preprocess in O(nlogn) time we need to use the fact that

M[i][j] = GCD of ( M[i][j-1] and M[i+2j-1][j-1] )

Such that none of these terms goes out of bounds.

**\*we are using the fact that GCD of an array from i to j = GCD of GCD of array from i to k and k+1 to j**

*Pseudocode for preprocessing:*

GCD (no1, no2){

    //blackbox function which returns GCD of 2 input

}

POW[logn]{// creating array such that POW[j]=2^j

    POW[1]=2

    for i from 2 to logn:

        POW[i]=2\*POW[i-1]

}

LOG[n+1]// LOG[m]: such that the greatest integer 𝒌 such that 𝟐^𝒌  ≤ 𝒎+1

for i from 0 to n-1

{

    M[i][0]=A[i] // since GCD of a no. is the no. itself

}

for j from 1 to logn

{

    for( int i=0; i+ POW[j] -1 < n; i++){

        M[i][j]= GCD( M[i][j-1], M[i+POW[j-1]][j-1])

    }

}

*Binary Search:*

We set low =1 and high =n, for a mid we check is it possible or not, if it is then low=mid+1 else high=mid-1. Checking involves comparing GCD of every contiguous subsequence of that length with k

*Pseudocode:*

low =1, high=n, ans

while(low<=high){

    mid=(low+high)/2

    if(check(mid)==true) {

        ans=mid

        low=mid+1}

    else high=mid-1

}

check(mid){

    for i from 0 to n-mid

        if( findgcd (i,mid) >= k) return true

        else return false

}

findgcd(i, mid){

    j=i+mid-1

    t=POW[mid];

    h=LOG[mid];

    if (t == mid)  return  M[i][h];

    else  return GCD(M[i][h], M[j-t][h]);

}

Print(ans)

**c.) To Give proof of correctness and time complexity analysis of approach for part (b)**

**Theorem:** C[i] gives the length of the smallest subarray such that its sum >=P if possible otherwise gives no value

**Proof:**  We increment j until we encounter sum>=P for the first time. Once we encounter sum>=P. We get to know the smallest length starting from i. Point to note is that

Sum\_Subarray [i+1,j] < Sum\_Subarray [i ,j],

because Sum\_Subarray [i ,j]-Arr[i]= Sum\_Subarray [i +1,j] and Arr[i] >0

Now if Sum\_Subarray [i+1][j] is also >=P then again j-(i+1) gives the length of smallest subarray.

Before the initial loop iteration, 'i,' 'j,' and 'sum' are set to 0, while 'ans' is initialized with a high value symbolizing infinity. Since the subarray 'arr[0:0]' is empty, 'sum' correctly represents the sum (0), and 'ans' is suitably set for minimum cost.

Assuming the loop's validity before each iteration, we consider two scenarios:

1. If 'sum' < 'P,' we increment 'j' and update 'sum' with 'arr[j].' The loop's validity remains intact as 'sum' stays the subarray sum, 'j' advances, and 'i' remains unchanged.
2. When 'sum' >= 'P,' we enter an inner loop. It repeatedly increases 'i' and decreases 'sum' until 'sum' < 'P.' The loop's validity persists as 'sum' adjusts by subtracting elements, 'i' increments, and 'j' stays constant.

The loop concludes when 'j' exceeds 'n' due to incremental 'j' and finite 'n.' Post-termination, the loop's validity still holds. Thus, 'ans' indeed stores the minimum cost for a subarray sum of at least 'P.'

Hence Proved.

