1st PAGE : Question Problem : Longest Subarray with Sum K

(12) Longest subarray with given sum (positive)

longest subarray with sum 'k' (positive case)

eg: arr[] = [1, 2, 3, 1, 1, 1, 1, 9, 2, 3]

sum of subarray means.
with eg. [1, 2] is a subarray its sum '3' but it's length '2'.
[3] is a subarray its sum 3 but it's length '1'.
But [1, 1, 1] is a also subarray its sum is 3 and length '3'
so one of them '3' is the length so this subarray is answer.

a subarray defined as take element from array. contigious element. otherwise it is not array.

1) Brute force approach:
generate the every possible subarray from that array and find the each length which is max and that follow conditions (max == sum) then that subarray is answer.

Q is How do you generate subarray.
By taking '2' pointer:

arr = 1, 2, 3, 1,
= [1], [1, 2], [1, 2, 3] [1 2, 3, 1].
i & j → stay where that is a subarray

these are subarray and it is first iteration, if 'j' will goes till last of array then we move again.

arr → 1 2 3 1
= [2], [2, 3], [2, 3, 1] this all again subarray.

find sum btw (i & j) j+1 for loop

```
l = 0
for(i=0; i<n; i++) {
  for(j=i; j<n; j++) {
    // we are finding sum of generated subarray and calculate
    s=0
    for(k=i; k<j; k++) {
      s = s + a[k];
    if(s==k) {
      return max(l, j-i+1)
```

time complexity is near about $O(n^3)$ because j & k are reducing in next iteration and space is not used so space complexity is $O(1)$

// ~~optimized solution~~:

// reduced brute force method (dist $o(n^2)$).
we don't need to run extra loop that is 'k'
simple we can calculate sum after moving j.

```
l=0
for(i = 0; i<n ; i++)
    s = 0
    for (j = i; j<n; j++)
        s += a[j];
        if (s == k)
            l = max(l, j-i);
if (s == k) return l;
```

length updating

here time Complexity = $o(n^2)$
space = $o(1)$

Better solution: Making | and prefix sum pattern:

(1's learn first about → this Algorithm / pattern

eg a[] = [1, 3, -2, 9, -1, 5]
0 1 2 3 4 5

It is simple and powerfull technique that allows to perform fast calculation on the sum of elements in a given range (called Contiguous Segment of array)

1 3 -2 9 -1 5
↓ ↓ ↓ ↓ ↓ ↓
a[] : [1, 4, 2, 6, 5, 10]. { that is prefix sum stored in same array }
0 1 2 3 4 5
↑

Not taken extra space
But use o(n) times

we sum prefix element and current element from start to end
// we always start iterating loop from next index (not 0, but at 1)

Step wise
0 1 2 3 4 5
a = [1 3 -2 9 -1, 5]
[1 4 -2 9 -1, 5]
[1 4 2 9 -1 5]
[1 4 2 6 -1, 5]
[1, 4, 2, 6 5, 5]
a = [1, 4 2, 6 5 (10)]

we are doing, for(i → 1 → i<n);
D. a[0] = 1, a[1] = a[0]+a
= 1+3 =
a[2] = a[2]+a[1] = 4+(-2) = 2.
so for(i=1; i<n; i++) {
a[i] = a[i] + a[i-1] ✓ o(n)
}
02-06

Application: (class)

If we have array :- A[] = [6, 3, -2, 9, -1, 0, -5]

i =	0	1	2	3	4	5	6
A[] =	6	3	-2	9	-1	0	-5
prefix sum = A[] =	6	9	7	11	10	10	5

It is start from 0 to n
for(i=0; i<n; i++)
a[i] :-

① Calculate the sum between range [0,4] ?
then we can simple take element at '4' index without Calculating again, using prefix sum. So ans is (10)
so if we have ~~take~~ given [0,i] and say find sum btw both range so we can take element at index (j).
we can easily access in index element in O(1) time.

② Calculate the sum btw range [2,6] ?

A[] = [6, 9, 7, 11, 10, 10, 5]

find this. and find this

[0,6] - A[0,1] ⇒ (a[6] - a[1]) 5 - 9 = -4 Ans. (5 - 9 = -4)

formula = $A[i,j] = A[j] - A[i-1]$

generalized form. $[0,j] - [0,i-1]$

complexity O(1), and calculate prefix O(n) so overall O(n) nothing

If we have only positive integer element in array and we want to find subarray of sum k, then if i use hashmap, it will create problem that is unnecessary, because if +ve integer have in an and want to find prefix sum and store it as a key and give it's value index by checking if it is present or not, at allocate key the sum of prefix will be increasing all the time because it have only +ve element. it will used more space not usefull create problem like Redundant check.

So if we have element combination of +ve & -ve both then this hashmap technique work efficiently. we i will write down the pseudo code on next page.

ASUS Vivobook

→ Map<Integer, Integer> pm = new HashMap<>();
int currentSum = 0;
int max = 0;
for (int i=0; i<arr.length; i++) {
currentSum = currentSum + a[i];
if (currentSum == k) {
max = i+1;
}

// If subarray find current sum at first & that is ans.

if (pm.containsKey(currentSum - k)) {
max = Math.max(max, i - pm.get(currentSum - k));
} // if (current sum - k) exist in the map then

if (!pm.containsKey(currentSum)) {
pm.put(currentSum, i);
}
}
return max;

// find sum subarray for +ve & -ve sum = k

optimal approach
time c:→ O(n) traverse the array once, and each operation with the hashmap is O(1)
space comp:
& O(n) in the worst case we store each cumulative sum in the ...

Q
let's dry run
a = [1, -1, 1]
i = 0 1 2
CS = 1 0, k = 1
max = 1 1 3
← (1+2)

CS - k
1-1=0 ✓1
1-0=1 ✓2

2→2
0→1
1→0

(2) find subarray sum [6 3 -2 4 -1, 0, -5]

But if array just contains (+ve) & zeros,
then the optimal approach:

why : Two pointer or greedy approach / Slid window

i = a[0]

a[] = [1, 2, 3, 1, 1, 1, 1, 3, 3].

a[0]
sum = 1 3 6 7 6 7 6

length = 3 3 4 4

K = 6

sum == K
length find.

sum > K
sum = sum - a[i]

while we here

if sum < K
then we add 'j'
into sum (j++)
sum = sum + a[j]

we have to maintain always | sum = 6 | ok.
if increase 'j' and add into sum if it increasing / greater > 6 then we subtract (trim) from the left that is i trimmy from sum

Time complexity
→ O(2N) because j is not inner

when j > length then we will exceed

Algorithm / Pseudo code:

this work only for +ve contain value

```
i = 0, j = 0,
int sum = a[0];
int max = 0, n = a.length;
while(j < n) {
    while (sum > K && i <= j) {
        sum = sum - a[i];
        i++;
    }
    if (sum == K) {
    max = Math.max(max, j-i+1);
    }
    j++;
    if (sum < K && j < n) {
        j++;
    sum = sum + a[j];
    }
    return max;
}
```

what is the time and space complexity.

time complexity:
→ O(2n) because not inner while loop run for each value of 'j', so it will consider as (2n);

and space is not why ~~used here~~ any where so space complexity is O(1)

Quite simple.

OPTIMISEZ APPROACH WITHOUT USING EXTRA SPACE SOLVED FOR +VE ELEMNT ONLY