Understanding the problem statement

Given an array arr=[5,4,6,7,9,8,3,1,2] and integer X, we should find the continuous sub array such that sum of all elements in that sub array should be equal to X. If X= 21, sub array is [9,8,3,1]

ans = [9,8,3,1]

Approach 1

Find all continuous sub arrays and check whether their sum is equal to X or not. With first number 5 there are (n) continuous sub arrays possible like [5],[5,4],[5,4,6].....[5,4,6,7,9,8,3,1,2] With second number 4 there are (n-1) continuous sub arrays possible like [4],[4,6],[4,67]... So total sub arrays will be O(n^2) So for every i ranging from 0 to n-1 we take j ranging from i+1 to n; then arr[i:j] will be sub array. No extra space is used so space is O(1)

Time Complexity : O(n^2)

Space Complexity : O(1)

Implementation

In [12]:
```
def subArraySum(arr,size,reqsum):
    for i in range(size):
        curr_sum = 0
        for j in range(i,size):
            curr_sum = curr_sum+arr[j]
            if curr_sum == reqsum:
                print("subarray is index is from {0} to {1}".format(i,j))
                return 1
    return 0
```

In [13]:
```
arr = [5,4,6,7,9,8,3,1,2]
reqsum = 21
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 4 to 7
# 1
```

```
subarray is index is from 4 to 7
```

Out[13]:
```
1
```

In [14]:
```
arr = [5,4,6,7,9,8,3,1,2]
reqsum = 15
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 0 to 2
# 1
```

```
subarray is index is from 0 to 2
```

Out[14]:
```
1
```

In [15]:
```
arr = [5,4,6,7,9,8,3,1,2]
reqsum = 45
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 0 to 8
# 1
```

```
subarray is index is from 0 to 8
```

Out[15]:
```
1
```

Approach 2

1. Use two pointers
2. Let first pointer point to index 0 and second pointer point to index 1.
3. curr_sum = sum(arr[firstpointer : secondpointer-1])
4. If curr_sum is greater than reqsum, we have to remove elements. If we decrement secondpointer and remove element then that sub array we have already see, so we increment leftpointer and then remove element
5. If curr_sum is less than reqsum, we have to add elements. We increment right pointer and add elements.
6. If curr_sum is equal to reqsum, then arr[firstpointer : secondpointer-1] is the required sub array.

At max we scan each element twice, so complexity is 2n i.e O(n). And we use two pointers extra. This is independent of size of input. So space complexity is O(1)

Time Complexity : O(n)

Space Complexity : O(1)

Let us see how our no of sub arrays to test got reduced with this logic. Now in the above array arr=[5,4,6,7,9,8,3,1,2], using first approach we tested all sub arrays like [5],[5,4],[5,4,6],[5,4,6,7],[5,4,6,7,9],[5,4,6,7,9,8].....[5,4,6,7,9,8,3,1,2]. But using second approach after testing [5],[5,4],[5,4,6],[5,4,6,7].. out curr_sum becomes 22 which is greater than reqsum 21. So we increment left pointer and then start testing [4],[4,6],[4,67]..... we skipped testing [5,4,6,7,9],[5,4,6,7,9,8].....[5,4,6,7,9,8,3,1,2]. Thus our testing sub arrays got reduces and our time complexity got reduced.

Implementation

In [16]:
```python
def subArraySum(arr,size,reqsum):
    lp = 0
    curr_sum = arr[0]
    for rp in range(1,size+1):
        while(curr_sum > reqsum and lp < rp-1 ):
            curr_sum -= arr[lp]
            lp += 1
        if curr_sum == reqsum:
            print("subarray is index is from {0} to {1}".format(lp,rp-1))
            return 1
        if rp < size:
            curr_sum += arr[rp]
    print("No subarray found")
    return 0
```

In [17]:
```python
arr = [5,4,6,7,9,8,3,1,2]
reqsum = 21
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 4 to 7
# 1
```

```
subarray is index is from 4 to 7
```

Out[17]:
```
1
```

In [18]:
```python
arr = [5,4,6,7,9,8,3,1,2]
```

```
reqsum = 15
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 0 to 2
# 1
```

```
subarray is index is from 0 to 2
```

Out[18]:
```
1
```

In [19]:
```
arr = [5,4,6,7,9,8,3,1,2]
reqsum = 45
subArraySum(arr,9,reqsum)
# ans
# subarray is index is from 0 to 8
# 1
```

```
subarray is index is from 0 to 8
```

Out[19]:
```
1
```

Approach 3

In approach 2, we are incrementing left pointer to reduce sum and incrementing right pointer to increase sum. This approach does not work if we have negative elements in the array. For example, if we have -5,6,7,8 subarray,curr_sum is 16 [-5+6+7+8] and lp is at 0 i.e -5. To reduce sum if we increment lp to 1 the sum becomes 6+7+8 = 21. So this logic does not work if we have negative elements in array. We should use below approach in case of negative elements array.

1. Given an array [7,6,-3,4,8] and X= 12. We should store all sums of sub_arrays to hash. 2. first subarray is [7], sum is 7 store sum to hash H= [7] 3. next subarray is [7,6]. sum is 13. Now subtract X from curr_sum. 13-12 = 1. check whether 1 is present in H or not. If it is present then we got subarray.or else repeat process till end of array. H becomes [7,13] [.....hash_sum_index.....curr_sum_index] if curr_sum - req_sum is in hash at hash_index that means [hash_sum_index+1 ... curr_sum_index] is our required sub_array. 4. next subarray is [7,6,-3]. sum is 13-3=10 . 10-12 = -2 not in H. H becomes [7,13,10] 5. next subarray is [7,6,-3,4]. sum is 14. 14-12 = 2 not in H. H becomes [7,13,10,14] 6. next subarray is [7,6,-3,4,8]. sum is 22. 22-12 = 10. 10 is in H at index 2. so our subarray should be from index 3 to current_index 4 i.e [4,8] # ans [4,8]. we got this because we are substracting whole array sum till current_index - required_sum, and checking if we have already got this sum somewhere till some index. If we already got it , then from that index onwards till current index gives our required_sum. current_index_sum - required_sum = old_sum means current_index_sum - old_sum = required_sum that is why answer is from old_sum_index+1 to current_index_sum.

For find the sub array with given sum X=0 also we can use approach 3.

In [ ]:
```

```