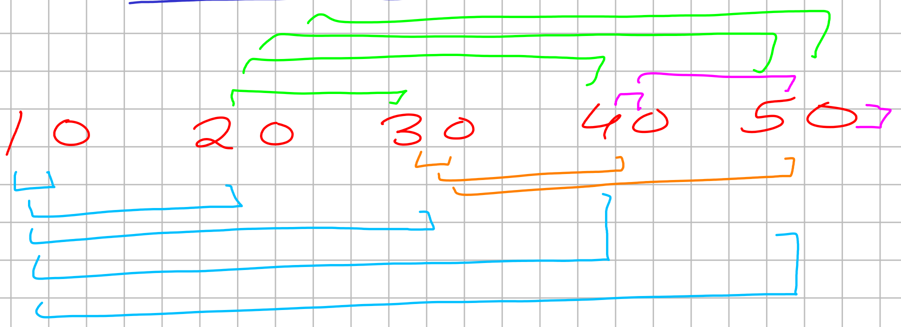
Generate Subarray.

Theory :- Subarray of an array is a contiguous part of an given array.

eg :-

0	1	2	3	4
10	20	30	40	50



all subarrays.

let i be start of the subarray and j be the end of the subarray.

We can say,

$\boxed{0 \le i \le n-1}$ $\boxed{i \le j \le n-1}$

no of possible subarrays for n size

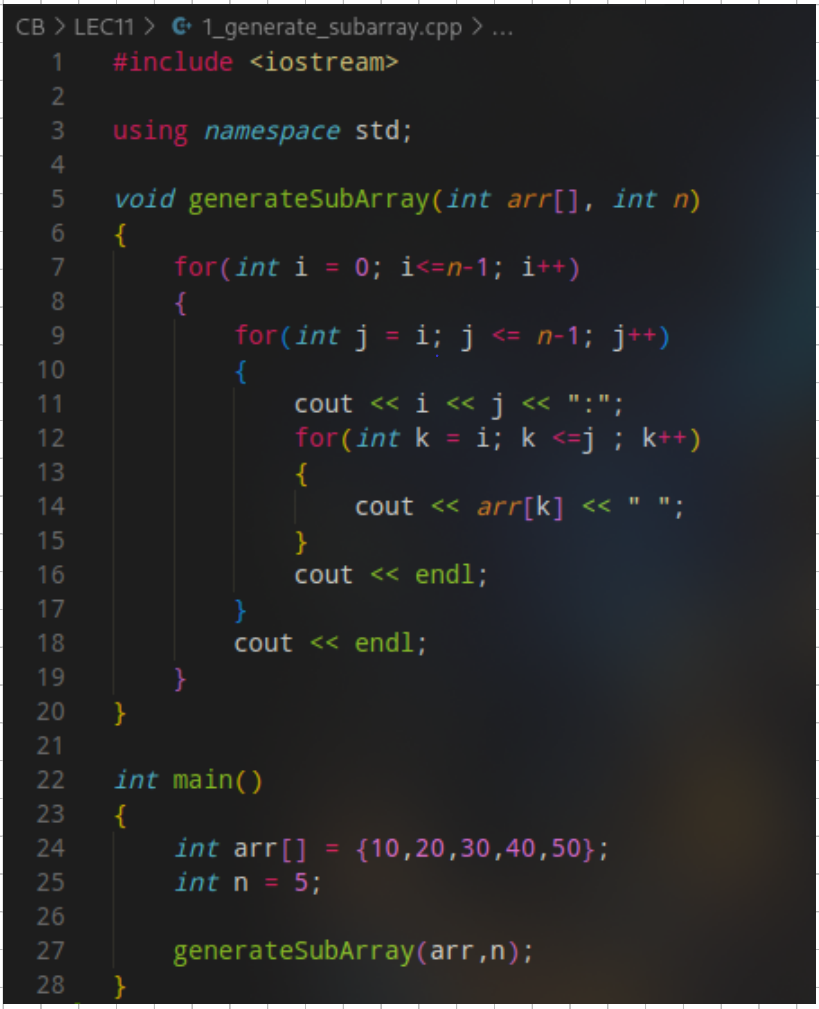
i	j	# Sub-arrays
0	0 to n-1	n
1	1 to n-1	n-1
2	2 to n-1	n-2
⋮	⋮	n-3
		⋮
n-1	n-1 to n-1	1

$\Sigma = \frac{n(n+1)}{2}$

$\approx n^2$

$\boxed{\frac{n(n+1)}{2}}$ → no of subarray

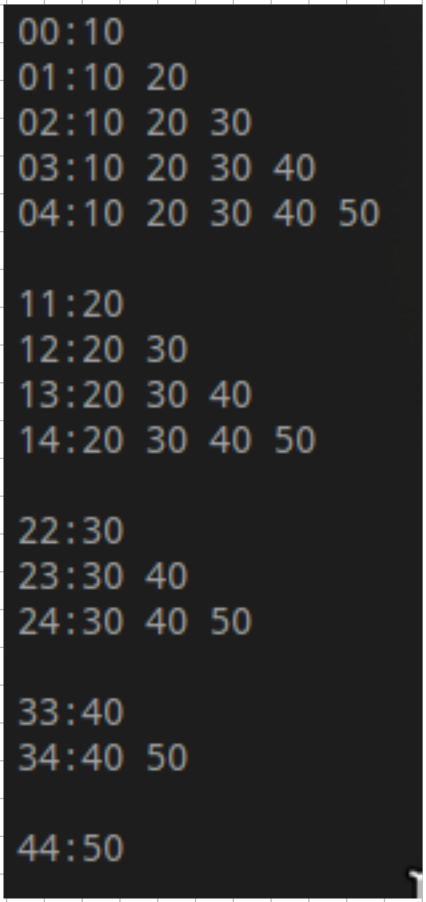
Code:-



CB > LEC11 > 1_generate_subarray.cpp > ...

```cpp
#include <iostream>

using namespace std;

void generateSubArray(int arr[], int n)
{
    for(int i = 0; i<=n-1; i++)
    {
        for(int j = i; j <= n-1; j++)
        {
            cout << i << j << ":";
            for(int k = i; k <=j ; k++)
            {
                cout << arr[k] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }
}

int main()
{
    int arr[] = {10,20,30,40,50};
    int n = 5;

    generateSubArray(arr,n);
}
```

Output



```
00:10
01:10 20
02:10 20 30
03:10 20 30 40
04:10 20 30 40 50

11:20
12:20 30
13:20 30 40
14:20 30 40 50

22:30
23:30 40
24:30 40 50

33:40
34:40 50

44:50
```

Maximum Subarray Sum

↳ To find the maximum subarray sum

Theory:-

Commulative Sum / Prefix Sum

↳ eg:- 10 20 30 40 50

10 → 10

10 20 → 30

10 20 30 → 60

10 20 30 40 → 100

10 20 30 40 50 → 150

prefix sum

Method - I

Brute force method :-

Can be implemented from the above code.

Time complexity :- $O(n^3)$

Space complexity :- $O(1)$

Method - II

Optimizing the above method. by computing the prefix sum of the array.

eg :-

arr

0	1	2	3	4	5	6	7	8
-2	1	-3	4	-1	2	1	-5	4

S_{16}

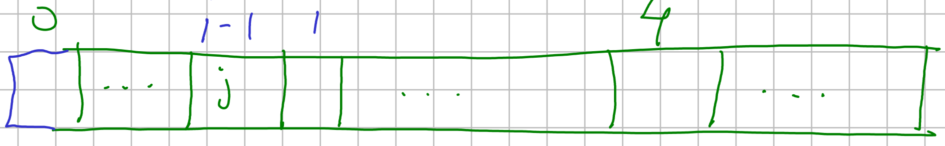
prefix sum arr.

0	1	2	3	4	5	6	7	8	9
	-2	-1	-4	0	-1	1	2	-3	1

(1) $] \rightarrow S_{16}$

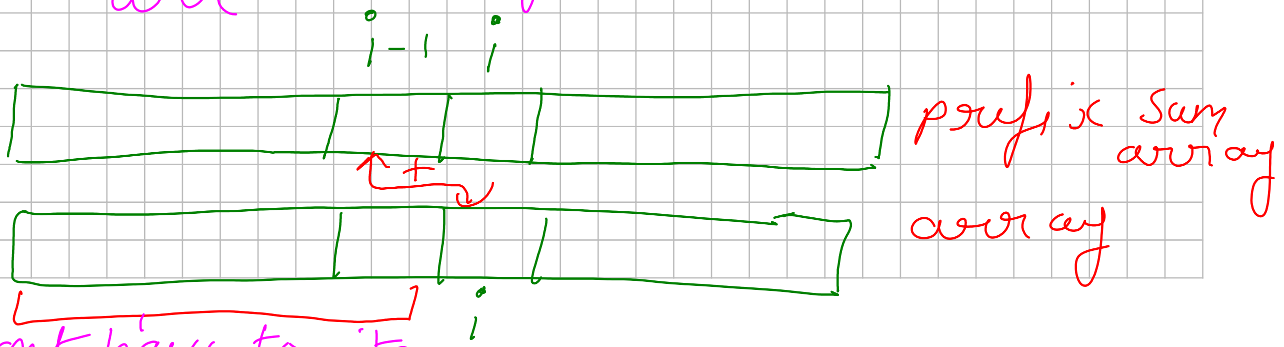
formula :- $\boxed{psum[j+1] - pSum[i]} = S_{ij}$

↳ for finding the subarray sum



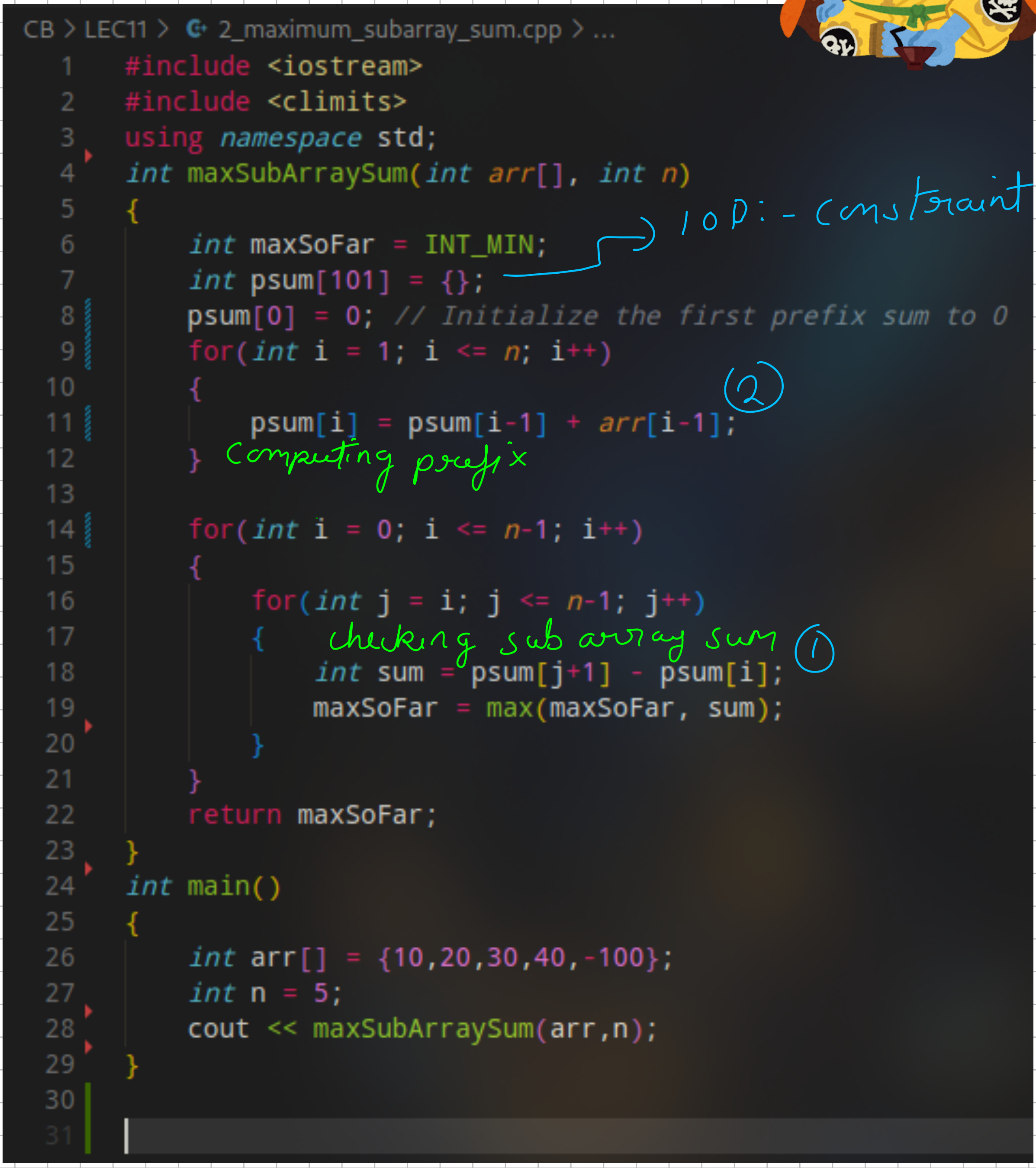
(2) $\boxed{pSum[i] = pSum[i-1] + arr[i-1]}$

↳ for computing the prefix sum arr



dont have to iter over these values

Code :-



CB > LEC11 > 2_maximum_subarray_sum.cpp > ...

```cpp
#include <iostream>
#include <climits>
using namespace std;
int maxSubArraySum(int arr[], int n)
{
    int maxSoFar = INT_MIN;
    int psum[101] = {};
    psum[0] = 0; // Initialize the first prefix sum to 0
    for(int i = 1; i <= n; i++)
    {
        psum[i] = psum[i-1] + arr[i-1];
    }

    for(int i = 0; i <= n-1; i++)
    {
        for(int j = i; j <= n-1; j++)
        {
            int sum = psum[j+1] - psum[i];
            maxSoFar = max(maxSoFar, sum);
        }
    }
    return maxSoFar;
}
int main()
{
    int arr[] = {10,20,30,40,-100};
    int n = 5;
    cout << maxSubArraySum(arr,n);
}

```

10D:- constraint

(2)

Computing prefix

checking sub array sum (1)

Output :- 100

Time Complexity :- $O(n^2)$

Space Complexity :- $O(n)$

Method - III (Kadane Method)

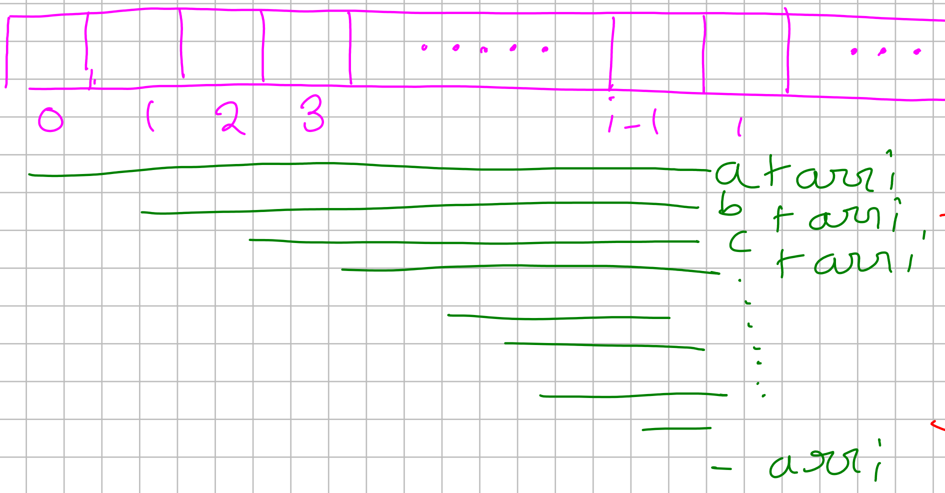
Best Method for finding subarray sum.

Theory

Assume :- arr,

0	1	2	3	4	
-3	2	-1	4	-5	arr
-3	2	1	5	0	x
x_0	x_1	x_2	x_3	x_4	

$x_i \longrightarrow$ Max (sum of elements which end at i) (contiguous of arr)



* dont have to check as they are < a + arr_i

∴ if we have, x_{i-1} and arr_i

we can find x_i by

$$\boxed{x_i \longrightarrow max(x_i + arr_i, arr_i)}$$

Calculating values of x,

$x_1 = max(x_0 + arr_0, arr_0)$

$= max(-3 + 2, 2)$

$= max(-1, 2) = 2$

$x_2 = max(x_1 + arr_1, arr_1)$

$= max(2 + -1, -1)$

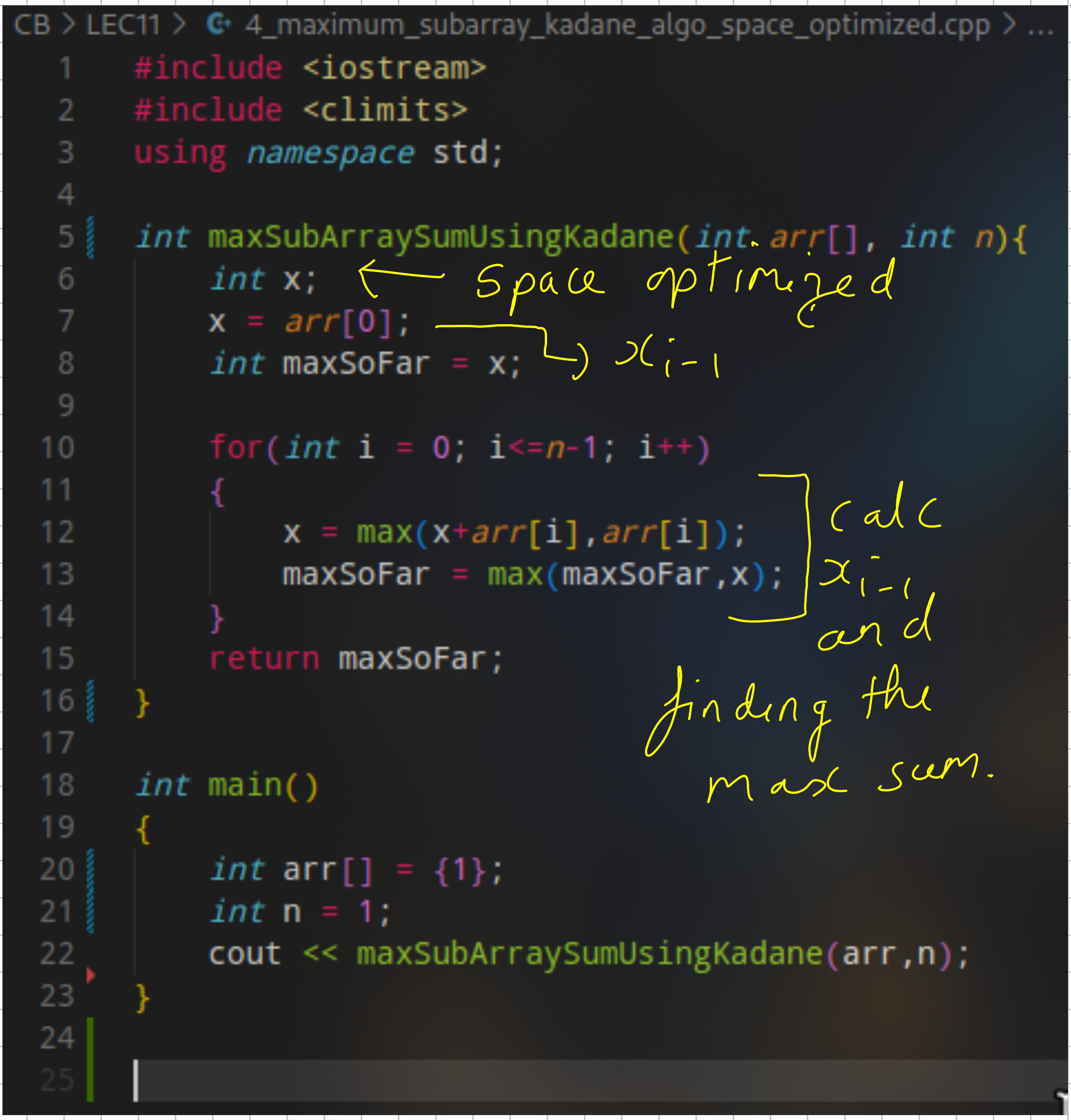
$= max(1, -1) = 1$

$x_3 = \max(x_2 + arr_2 , arr_2)$
$= \max(1+4, 4) = 5$

$x_4 = \max(x_3 + arr_3 , arr_3)$
$= \max(5 + -5, -5)$
$= \max(0, -5) = 0$ //

x_0 is $arr_0 = \underline{\underline{-3}}$

Code:-



CB > LEC11 > 4_maximum_subarray_kadane_algo_space_optimized.cpp > ...

```
#include <iostream>
#include <climits>
using namespace std;

int maxSubArraySumUsingKadane(int arr[], int n){
    int x;
    x = arr[0];
    int maxSoFar = x;

    for(int i = 0; i<=n-1; i++)
    {
        x = max(x+arr[i],arr[i]);
        maxSoFar = max(maxSoFar,x);
    }
    return maxSoFar;
}

int main()
{
    int arr[] = {1};
    int n = 1;
    cout << maxSubArraySumUsingKadane(arr,n);
}

```

← space optimized

→ x_{i-1}

calc x_{i-1} and finding the max sum.

Time Complexity → O(n)

Space Complexity → O(1), if arr of x is maintained then, O(n)