**Kadane's Algorithm : Maximum Subarray Sum in an Array**

**Theory:**

**Subarray:** contiguous subset of an array.

Ex: arr = [-2,1,-3,4,-1,2,1,-5,4]

[-3,4,-1] is a valid sub array but [-3,4,2] is not a valid sub array since it is not in contiguous manner.

**Subsequence:** subset of an array, i.e. elements may not be in contiguous form.

Ex: arr = [-2,1,-3,4,-1,2,1,-5,4]

[-3,2,4] is a subsequence.

**Note:** A single element can be considered as subarray and sub sequence both.

**Problem Statement**: Given an integer array arr, find the contiguous subarray (containing at least one number) which  
has the largest sum and returns its sum and prints the subarray.

**Examples**

**Example 1:Input:** arr = [-2,1,-3,4,-1,2,1,-5,4]

**Output:** 6

**Explanation:** [4,-1,2,1] subarray has the largest sum = 6.

**Examples 2:Input:** arr = [1]

**Output:** 1

**Explanation:** Array has only one element and which is giving positive sum of 1.

**Brute Force Approach**

**Intuition:**

Checking the sum of each and every possible sub array,i.e. trying out all of the possible sub array and whichever subarray gives maximum sum is the answer.

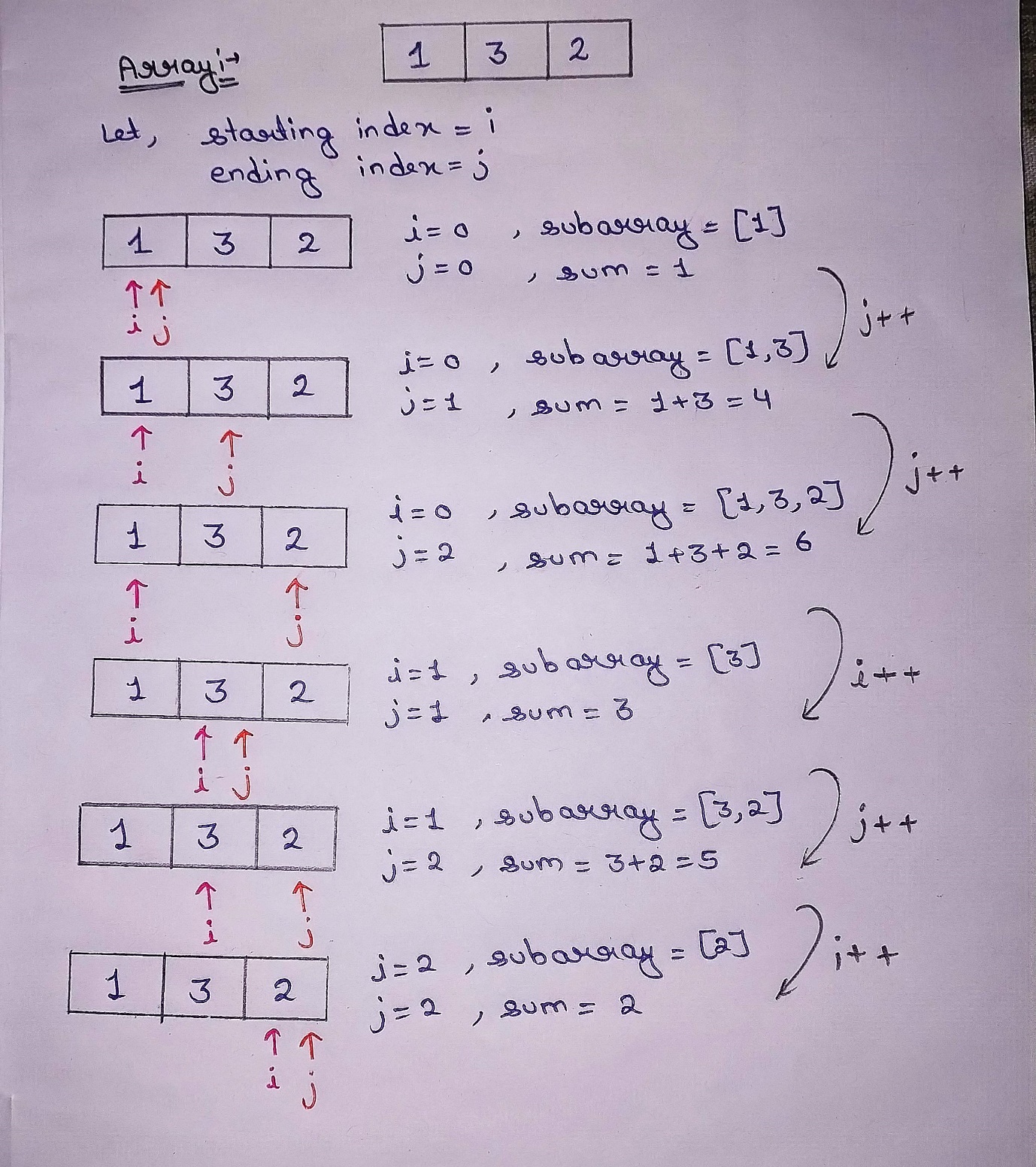
**Steps to be followed:**

* LOOP 1: First we will run a loop from index 0 to n{size-1} to take sub arrays starting from each element of array. To select every possible starting index of the subarray
* LOOP 2: Inside the above loop we will run another loop to get the all possible ending indexes for the starting index of loop 1.
* LOOP 3: Now, we will run another loop inside loop 2 to get the sum of subarray which has a starting index coming from loop 1 and ending index coming from loop2. This way we can get the sum of all possible sub arrays.

**Dry Run:**

Subarrays are marked with yellow color.

First calculating the sum of all possible subarrays starting from 1 and then every possible subarrays from 3 and at the end from 2.



**Code:**

#include <bits/stdc++.h>

using namespace std;

int maxSubarraySum(int arr[], int n)

{

int maxi = INT\_MIN; // maximum sum is stored that;s why we initialized with INT\_MIN which is the lowest possible value.

for (int i = 0; i < n; i++)

{

for (int j = i; j < n; j++)

{

// subarray = arr[i.....j]

int sum = 0;

//adding all the elements of subarray arr[i…j]

for (int k = i; k <= j; k++) {

sum += arr[k];//sum of arr[i…j]

}

maxi = max(maxi, sum);//if sum is greater than previos subarray then maxi is updated.

}

}

return maxi;

}

int main()

{

int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};

int n = sizeof(arr) / sizeof(arr[0]);

int maxSum = maxSubarraySum(arr, n);

cout << "The maximum subarray sum is: " << maxSum << endl;

return 0;

}

**Output**: The maximum subarray sum is: 6

**Complexity Analysis:**

**Time Complexity:**O(N3), where N = size of the array.  
**Reason:**We are using three nested loops, each running approximately N times.

**Space Complexity:**O(1) as we are not using any extra space

**Better Approach:**

**Intuition:**

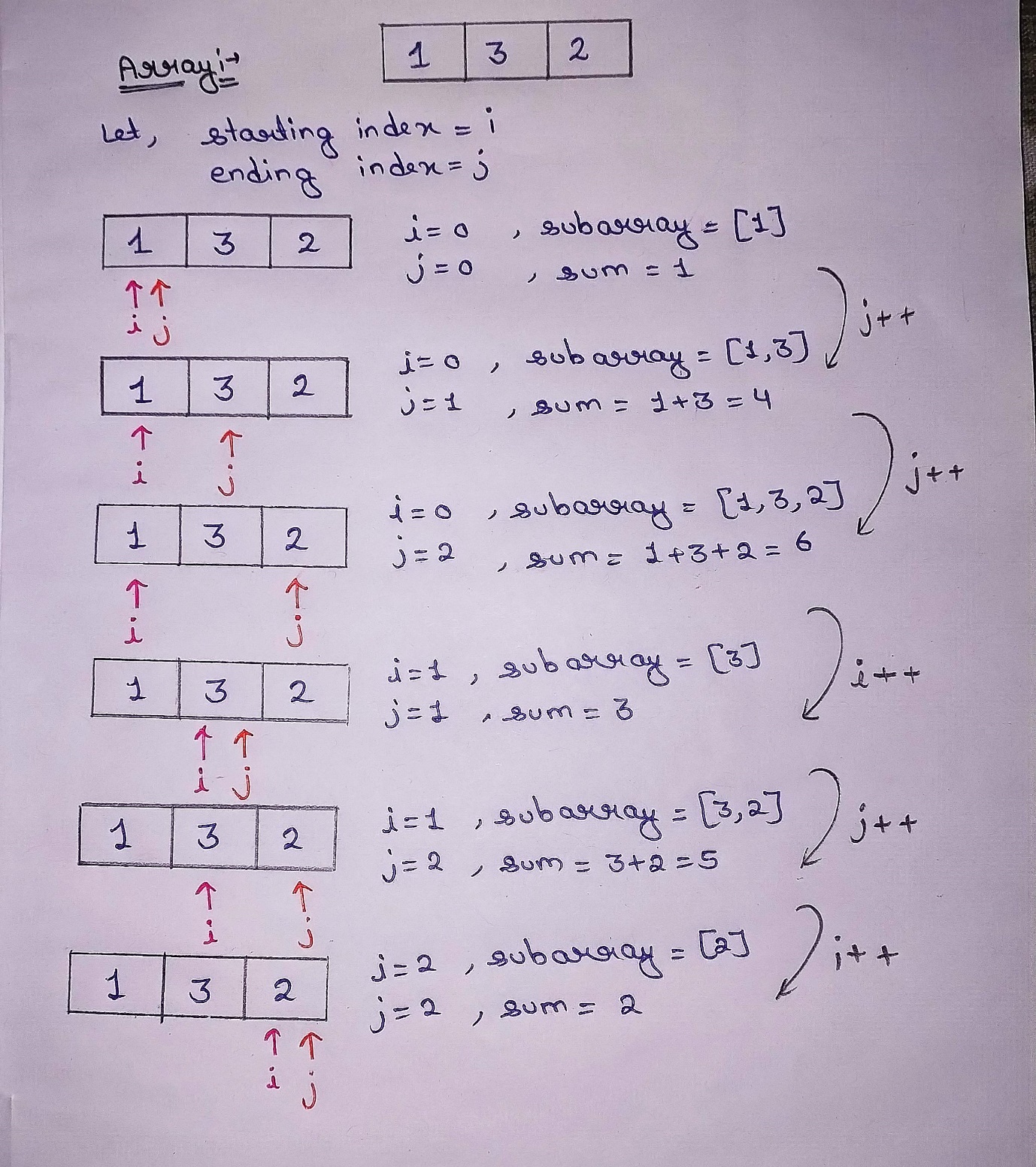
We can observe that in previous solution we used loop 3 to calculate the sum of each subarray, instead we can add new element to the previous subarray to get the sum of new subarray. For ex: arr[]={1,4,2} sum of subarray{1,4}=5 then sum of subarray{1,4,2}=5+2=7, instead of doing 1+4+2=7.

**Steps to be followed:**

* LOOP 1: First we will run a loop from index 0 to n{size-1} to take sub arrays starting from each element of array. To select every possible starting index of the subarray
* LOOP 2: Inside the above loop we will run another loop to get the all possible ending indexes for the starting index of loop 1 as well as the current element of the subarray. Now in this loop only, we will add the current element to the sum of the previous subarray i.e. **sum = sum + arr[j]**.

**Dry Run:**

Subarrays are marked with yellow colour.



**Code:**

#include <bits/stdc++.h>

using namespace std;

int maxSubarraySum(int arr[], int n)

{

int maxi = INT\_MIN;

for (int i = 0; i < n; i++)

{

int sum = 0;

for (int j = i; j < n; j++)

{

// current subarray = arr[i.....j]

//adding the current element arr[j]

// to the sum i.e. sum of arr[i...j-1]

sum += arr[j];

maxi = max(maxi, sum);

}

}

return maxi;

}

int main()

{

int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};

int n = sizeof(arr) / sizeof(arr[0]);

int maxSum = maxSubarraySum(arr, n);

cout << "The maximum subarray sum is: " << maxSum << endl;

return 0;

}

**Output**: The maximum subarray sum is: 6

**Complexity Analysis:**

**Time Complexity:**O(N2), where N = size of the array.  
**Reason:**We are using two nested loops, each running approximately N times.

**Space Complexity:**O(1) as we are not using any extra space.

**Optimal Approach (KADANE’S ALGORITHM):**

### ****Intuition:****

### Since we want maximum sum so if a subarray with a sum less than 0 then it will always reduce our answer by subtracting and so this type of subarray cannot be a part of the subarray with maximum sum.

### ****Approach:****

### we will iterate the given array with a single loop

### Now, while iterating we will add the elements to the sum variable and consider the maximum one. If at any point the sum becomes negative we will set the sum to 0 as we are not going to consider it as a part of our answer.

**Note:**In some cases, the question might ASK to consider the sum of the empty subarray while solving this problem.

Since, if the array contains all negative elements, then in no case it will be equal to or greater than zero. In this case, zero is considered as sum of empty subarray{only if empty subarray is allowed/asked}.  
For e.g. if the given array is {-1, -4, -5}, the answer will be 0 instead of -1 in this case.

**Code:**

#include <bits/stdc++.h>

using namespace std;

long long maxSubarraySum(int arr[], int n) {

long long maxi = LONG\_MIN;

long long sum = 0;

for (int i = 0; i < n; i++) {

sum += arr[i];

if (sum > maxi) {

maxi = sum;

}

// If sum < 0: discard the sum calculated and return 0.

if (sum < 0) {

sum = 0;

}

}

// To consider the sum of the empty subarray

// uncomment the following check:

//if (maxi < 0) maxi = 0;

return maxi;

}

int main()

{

int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};

int n = sizeof(arr) / sizeof(arr[0]);

long long maxSum = maxSubarraySum(arr, n);

cout << "The maximum subarray sum is: " << maxSum << endl;

return 0;

}

**Output**: The maximum subarray sum is: 6

**Complexity Analysis:**

**Time Complexity:**O(N), where N = size of the array.  
**Reason:**We are using a single loop running N times.

**Space Complexity:**O(1) as we are not using any extra space.

**Follow-up question:**

**Intuition:**

We will store the starting index in ansStart and ending index in ansEnd while calculating the maximum sum of subarray, and then we will use these indexes to print the subarray.

* So, we will keep a track of the starting index inside the loop using a **start** variable.
* We will take two variables **ansStart** and **ansEnd** initialized with -1. And when the sum crosses the maximum sum, we will set **ansStart** to the **start** variable and **ansEnd** to the **current index i.e. i**.

**Code:**

#include <bits/stdc++.h>

using namespace std;

long long maxSubarraySum(int arr[], int n) {

long long maxi = LONG\_MIN;

long long sum = 0;

int start = 0;

int ansStart = -1, ansEnd = -1;

for (int i = 0; i < n; i++) {

if (sum == 0) start = i; // starting index

sum += arr[i];

if (sum > maxi) {

maxi = sum;

ansStart = start;

ansEnd = i;

}

// If sum < 0: discard the sum calculated

if (sum < 0) {

sum = 0;

}

}

//printing the subarray from ansStart to ansEnd:

cout << "The subarray is: [";

for (int i = ansStart; i <= ansEnd; i++) {

cout << arr[i] << " ";

}

cout << "]n";

// To consider the sum of the empty subarray

// uncomment the following check:

//if (maxi < 0) maxi = 0;

return maxi;

}

int main()

{

int arr[] = { -2, 1, -3, 4, -1, 2, 1, -5, 4};

int n = sizeof(arr) / sizeof(arr[0]);

long long maxSum = maxSubarraySum(arr, n);

cout << "The maximum subarray sum is: " << maxSum << endl;

return 0;

}

**Output:**The subarray is: [4 -1 2 1 ]  
The maximum subarray sum is: 6

**Complexity Analysis:**

**Time Complexity:**O(N), where N = size of the array.  
**Reason:**We are using a single loop running N times.

**Space Complexity:**O(1) as we are not using any extra space.