Kadane's Algorithm : Maximum Subarray Sum in an Array

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] 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.
```

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Practice: Solve Problem

▼ Brute Force Approach

▼ Algorithm / Intuition ›

Intuition:

We will check the sum of every possible subarray and consider the maximum among them. To get every possible subarray sum, we will be using three nested loops. The first loops(say i and j) will iterate over every possible starting index and ending index of a subarray. Basically, in each iteration, the subarray range will be from index i to index j. Using another loop we will get the sum of the elements of the subarray **[i…..j]**. Among all values of the sum calculated, we will consider the maximum one.

Approach:

The steps are as follows:

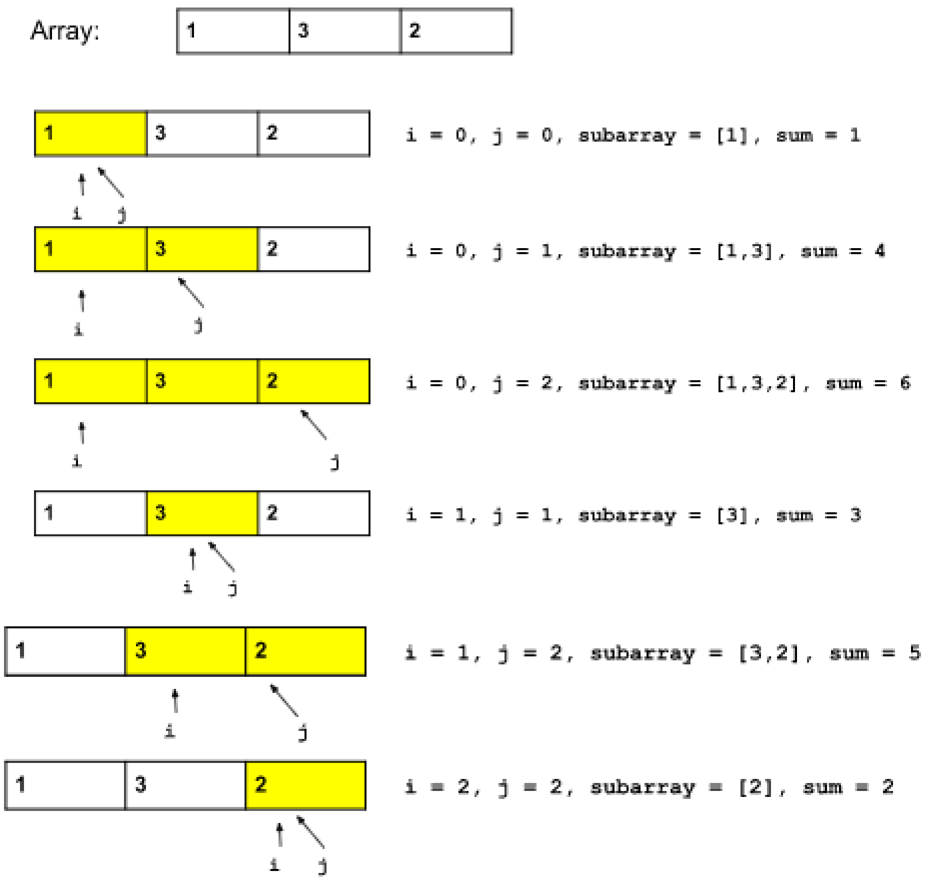
1. First, we will run a loop(say i) that will select every possible starting index of the subarray. The possible starting indices can vary from index 0 to index n-1(n = size of the array).

2. Inside the loop, we will run another loop(say j) that will signify the ending index of the subarray. For every subarray starting from the index i, the possible ending index can vary from index i to n-1(n = size of the array).
3. After that for each subarray starting from index i and ending at index j **(i.e. arr[i….j]),** we will run another loop to calculate the sum of all the elements(of that particular subarray).

Note: We are selecting every possible subarray using two nested loops and for each of them, we add all its elements using another loop.

▼ Dry Run ›

Subarrays are marked with yellow color.



Array:
1
3
2
i
j
i = 0, j = 0, subarray = [1], sum = 1
i = 0, j = 1, subarray = [1,3], sum = 4
i = 0, j = 2, subarray = [1,3,2], sum = 6
i = 1, j = 1, subarray = [3], sum = 3
i = 1, j = 2, subarray = [3,2], sum = 5
i = 2, j = 2, subarray = [2], sum = 2

▼ Code ›

```
import sys

def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1  # maximum sum

    for i in range(n):
        for j in range(i, n):
            # subarray = arr[i.....j]
            summ = 0

            # add all the elements of subarray:
            for k in range(i, j+1):
                summ += arr[k]

            maxi = max(maxi, summ)

    return maxi

if __name__ == "__main__":
    arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
    n = len(arr)
    maxSum = maxSubarraySum(arr, n)
    print("The maximum subarray sum is:", maxSum)
```

Output: The maximum subarray sum is: 6

▼ Complexity Analysis ›

Time Complexity: O(N^3), 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

▼ Algorithm / Intuition ›

Intuition: If we carefully observe, we can notice that to get the sum of the current subarray we just need to add the current element(i.e. **arr[j]**) to the sum of the previous subarray i.e. **arr[i….j-1]**.

Assume previous subarray = **arr[i……j-1]**current subarray = **arr[i…..j]**Sum of **arr[i….j]** = **(sum of arr[i….j-1]) + arr[j]**

This is how we can remove the third loop and while moving j pointer, we can calculate the sum.

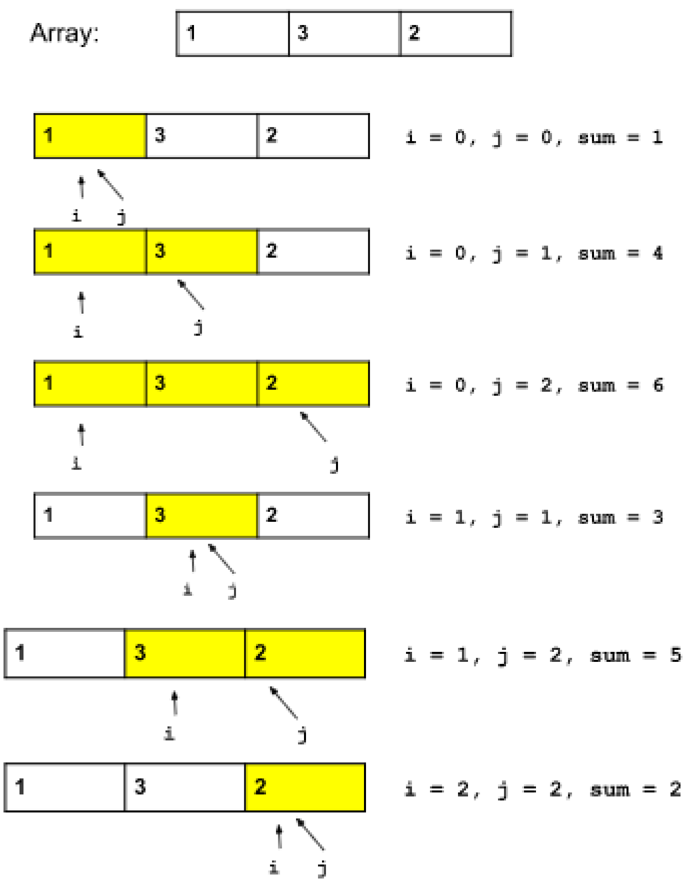
Approach:

The steps are as follows:

1. First, we will run a loop(say i) that will select every possible starting index of the subarray. The possible starting indices can vary from index 0 to index n-1(n = array size).
2. Inside the loop, we will run another loop(say j) that will signify the ending index as well as the current element of the subarray. For every subarray starting from index i, the possible ending index can vary from index i to n-1(n = size of the array).
3. Inside loop j, we will add the current element to the sum of the previous subarray i.e. **sum = sum + arr[j]**. Among all the sums the maximum one will be the answer.

▼ Dry Run ›

Subarrays are marked with yellow color.



▼ Code ›

```
import sys

def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1 # maximum sum

    for i in range(n):
        sum = 0
        for j in range(i, n):
            # current subarray = arr[i.....j]

            #add the current element arr[j]
            # to the sum i.e. sum of arr[i...j-1]
            sum += arr[j]

            maxi = max(maxi, sum) # getting the maximum

    return maxi

arr = [ -2, 1, -3, 4, -1, 2, 1, -5, 4]
n = len(arr)
maxSum = maxSubarraySum(arr, n)
print("The maximum subarray sum is:", maxSum)
```

Output: The maximum subarray sum is: 6

▼ Complexity Analysis ›

Time Complexity: O(N^2), 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

▼ Algorithm / Intuition ›

Intuition:

The intuition of the algorithm is not to consider the subarray as a part of the answer if its sum is less than 0. ***A subarray with a sum less than 0 will always reduce our answer and so this type of subarray cannot be a part of the subarray with maximum sum.***

Here, we will iterate the given array with a single loop and while iterating we will add the elements in a sum variable. Now, if at any point the sum becomes less than 0, we will set the sum as 0 as we are not going to consider any subarray with a negative sum. Among all the sums calculated, we will consider the maximum one.

Thus we can solve this problem with a single loop.

Approach:

The steps are as follows:

1. We will run a loop(say i) to iterate the given array.
2. Now, while iterating we will add the elements to the sum variable and consider the maximum one.
3. 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 say to consider the sum of the empty subarray while solving this problem. So, in these cases, before returning the answer we will compare the maximum subarray sum calculated with 0(i.e. The sum of an empty subarray is 0). And after that, we will return the maximum one.
For e.g. if the given array is {-1, -4, -5}, the answer will be 0 instead of -1 in this case.

This is applicable to all the approaches discussed above.

But if this case is not explicitly mentioned we will not consider this case.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

▼ Code ›

```
import sys

def maxSubarraySum(arr, n):
    maxi = -sys.maxsize-1 # maximum sum
    sum = 0

    for i in range(n):
        sum += arr[i]

        if sum > maxi:
            maxi = sum

        # If sum < 0: discard the sum calculated
        if sum < 0:
            sum = 0

    # To consider the sum of the empty subarray
    # uncomment the following check:

    #if maxi < 0: maxi = 0

    return maxi

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
n = len(arr)
maxSum = maxSubarraySum(arr, n)
print("The maximum subarray sum is:", maxSum)
```

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

▼ Algorithm / Intuition ›

There might be more than one subarray with the maximum sum. We need to print any of them.

Intuition: Our approach is to store the starting index and the ending index of the subarray. Thus we can easily get the subarray afterward without actually storing the subarray elements.

If we carefully observe our algorithm, we can notice that the subarray always starts at the particular index where the sum variable is equal to 0, and at the ending index, the sum always crosses the previous maximum sum(i.e. **maxi**).

- 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**.

The rest of the approach will be the same as Kadane's Algorithm.

Note: *For a better understanding of intuition, please watch the video at the bottom of the page.*

▼ Code ›

```python
import sys

def maxSubarraySum(arr, n):
    maxi = -sys.maxsize - 1  # maximum sum
    sum = 0

    start = 0
    ansStart, ansEnd = -1, -1
    for i in range(n):

        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:
    print("The subarray is: [", end="")
    for i in range(ansStart, ansEnd + 1):
        print(arr[i], end=" ")
    print("]")

    # To consider the sum of the empty subarray
    # uncomment the following check:

    # if maxi < 0:
    #     maxi = 0

    return maxi

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
n = len(arr)
maxSum = maxSubarraySum(arr, n)
print("The maximum subarray sum is:", maxSum)
```

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.