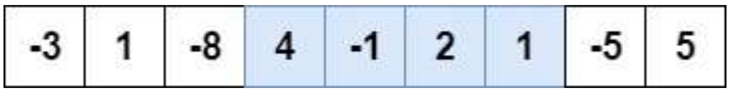
**Divide and conquer** is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The maximum subarray problem is a task to find the series of contiguous elements with the maximum sum in any given array.

For instance, in the below array, the highlighted subarray has the maximum sum(6):



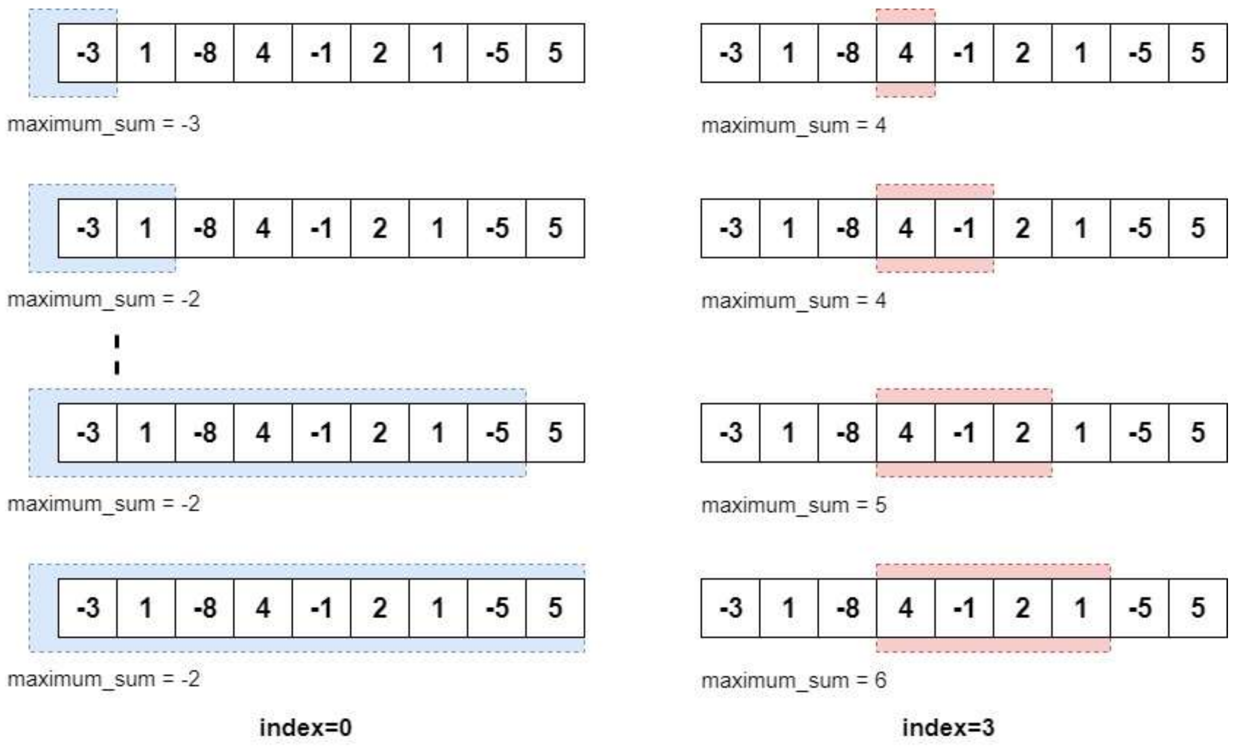
## Brute Force Algorithm

Brute force is an iterative approach to solve a problem. In most cases, the solution requires a number of iterations over a data structure. In the next few sections, we'll apply this approach to solve the maximum subarray problem.

## Approach

Generally speaking, the first solution that comes to mind is to calculate the sum of every possible subarray and return the one with the maximum sum.

To begin with, we'll calculate the sum of every subarray that starts at index 0. And similarly, we'll find all subarrays starting at every index from 0 to n-1 where n is the length of the array:



So we'll start at index 0 and add every element to the running sum in the iteration. We'll also keep track of the maximum sum seen so far. This iteration is shown on the left side of the image above.

On the right side of the image, we can see the iteration that starts at index 3. In the last part of this image, we've got the subarray with the maximum sum between index 3 and 6.

However, our algorithm will continue to find all subarrays starting at indices between 0 and n-1.

## Complexity

Generally speaking the brute force solution iterates over the array many times in order to get every possible solution. This means the time taken by this solution grows quadratically with the number of elements in the array. This may not be a problem for arrays of a small size. **But as the size of the array grows, this solution isn't efficient.**

By inspecting the code, we can also see that there are two nested *for* loops. **Therefore, we can conclude that the time complexity of this algorithm is** *O(n2)*.

## Maximum Subarray Sum using Divide and Conquer algorithm

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1. Divide the given array in two halves
2. Return the maximum of following three
    - Maximum subarray sum in left half (Make a recursive call)
    - Maximum subarray sum in right half (Make a recursive call)
    - Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1. Finally, combine the two and return the maximum among left, right and combination of both.

Below is the implementation of the above approach:

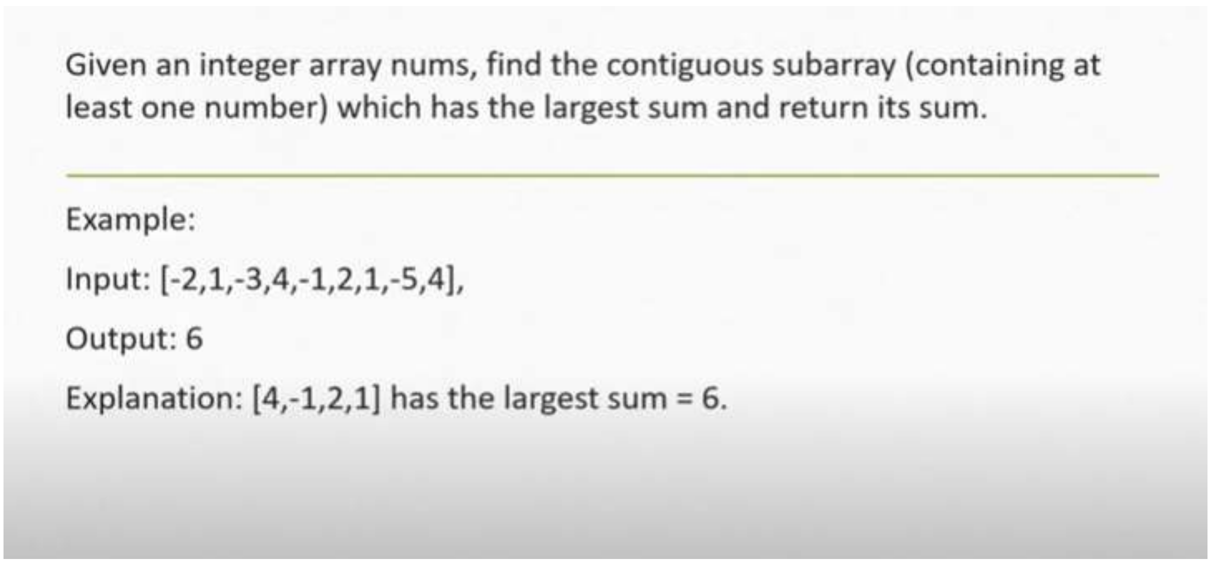
```
// A Divide and Conquer based program for maximum subarray
// sum problem
#include <limits.h>
#include <stdio.h>

// A utility function to find maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// A utility function to find maximum of three integers
int max(int a, int b, int c) { return max(max(a, b), c); }

// Find the maximum possible sum in arr[] auch that arr[m]
// is part of it
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid.
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--) {
        sum = sum + arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m + 1; i <= h; i++) {
        sum = sum + arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    // Return sum of elements on left and right of mid
    // returning only left_sum + right_sum will fail for
    // [-2, 1]
    return max(left_sum + right_sum, left_sum, right_sum);
}

// Returns sum of maximum sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
    // Base Case: Only one element
    if (l == h)
        return arr[l];

    // Find middle point
    int m = (l + h) / 2;

    /* Return maximum of following three possible cases
            a) Maximum subarray sum in left half
            b) Maximum subarray sum in right half
            c) Maximum subarray sum such that the subarray
       crosses the midpoint */
    return max(maxSubArraySum(arr, l, m),
               maxSubArraySum(arr, m + 1, h),
               maxCrossingSum(arr, l, m, h));
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int arr[] = { 2, 3, 4, 5, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int max_sum = maxSubArraySum(arr, 0, n - 1);
    printf("Maximum contiguous sum is %d\n", max_sum);
    getchar();
    return 0;
}
```

Output

```
Maximum contiguous sum is 21n
```

Time Complexity: maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \Theta(n)$



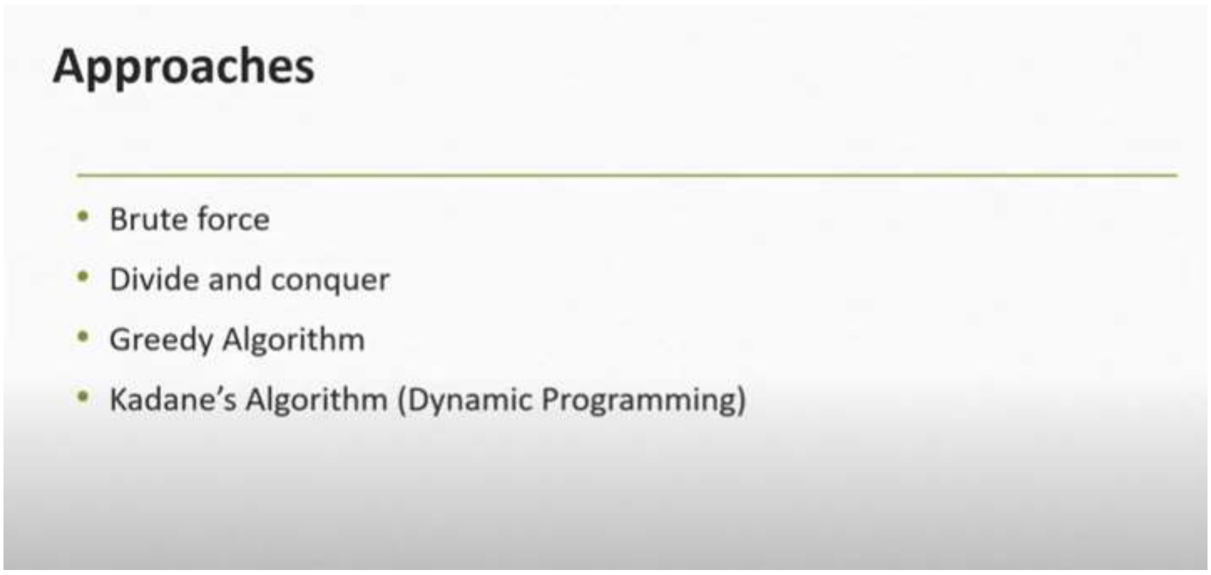
Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,4,-1,2,1,-5,4],

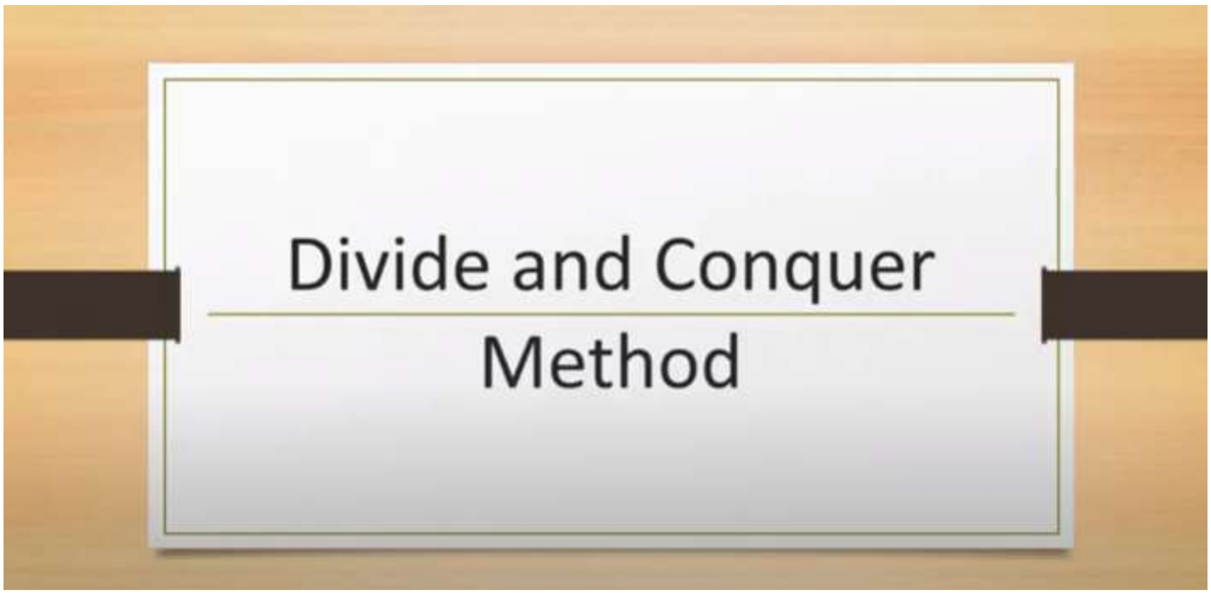
Output: 6

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



## Approaches

- Brute force
- Divide and conquer
- Greedy Algorithm
- Kadane's Algorithm (Dynamic Programming)



## Divide and Conquer Method

## Algorithm

1) Divide the given array in two halves

2) Return the maximum of following three

   a) Recursively calculate the Maximum subarray sum in left half

   b) Recursively calculate the Maximum subarray sum in right half

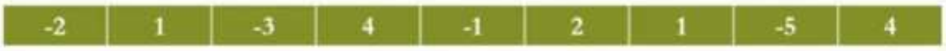
   c) Recursively calculate the Maximum subarray sum such that the subarray crosses the midpoint.

      i. Find the maximum sum starting from mid point and ending at some point on left of mid.

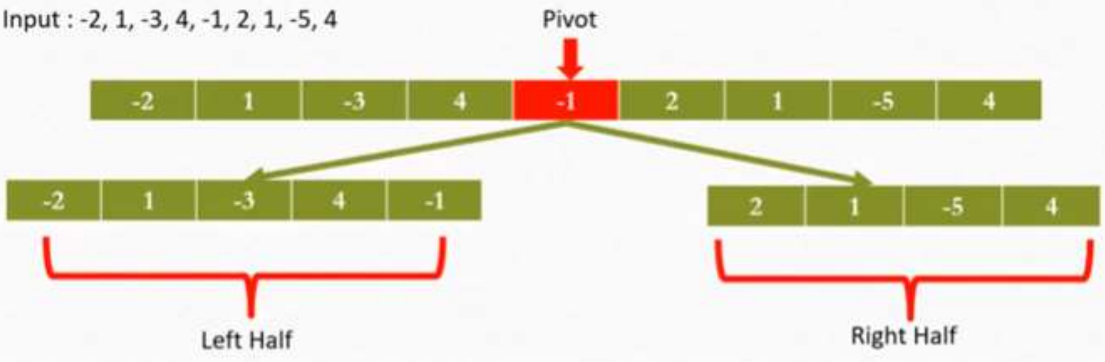
      ii. Find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1.

      iii. Finally, combine the two and return.

Input : -2, 1, -3, 4, -1, 2, 1, -5, 4



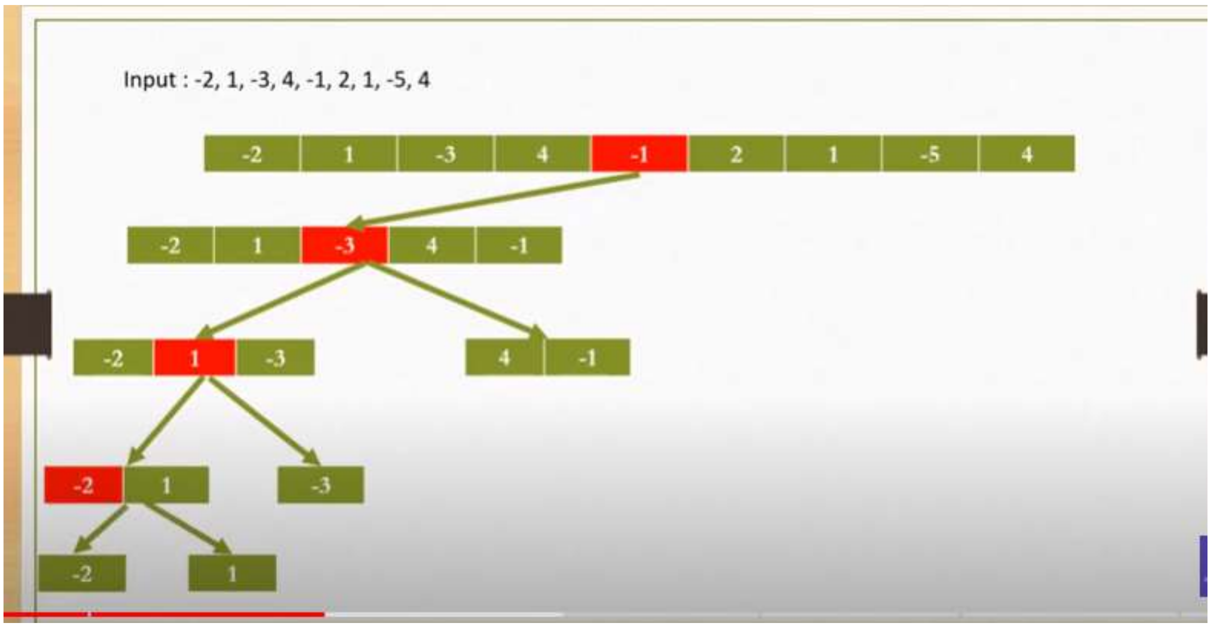
Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

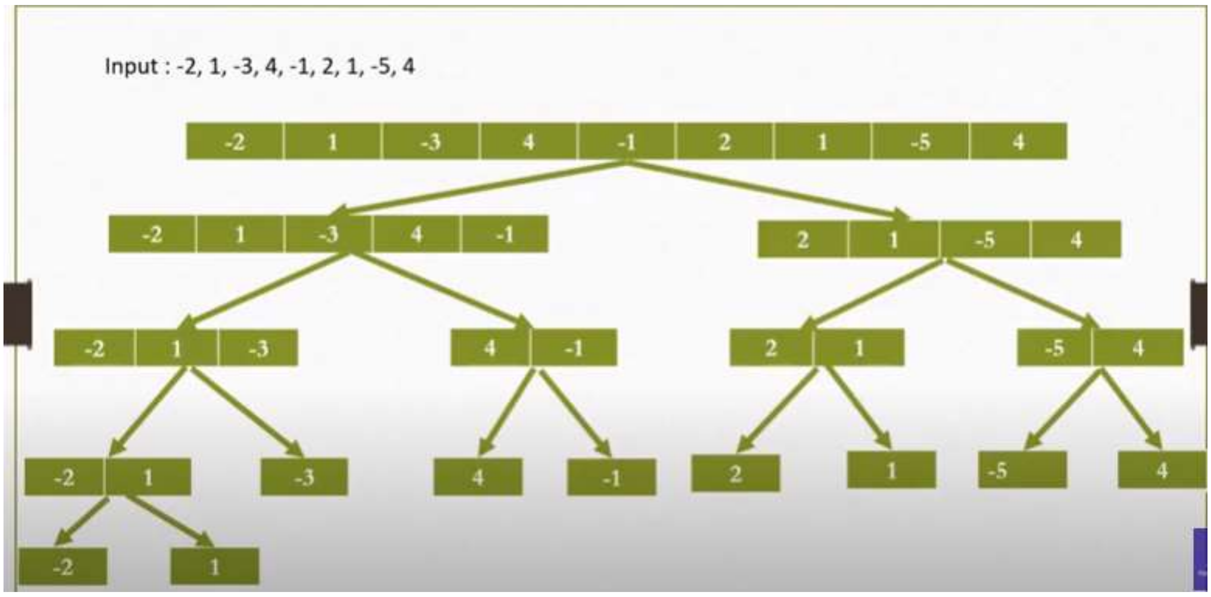


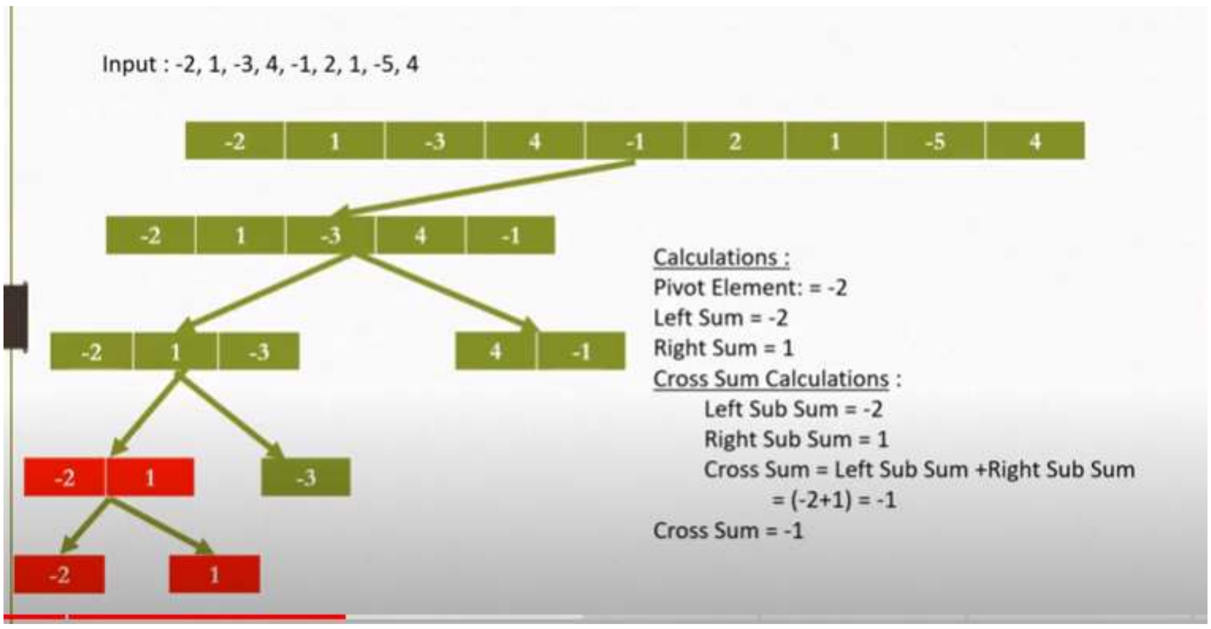
$$\text{Index of Pivot} = \frac{\text{(begin + end)}}{2} = \frac{(0+8)}{2} = 4$$

**Left Half = begin to pivot (inclusive)**

**Right Half = pivot +1 to end**

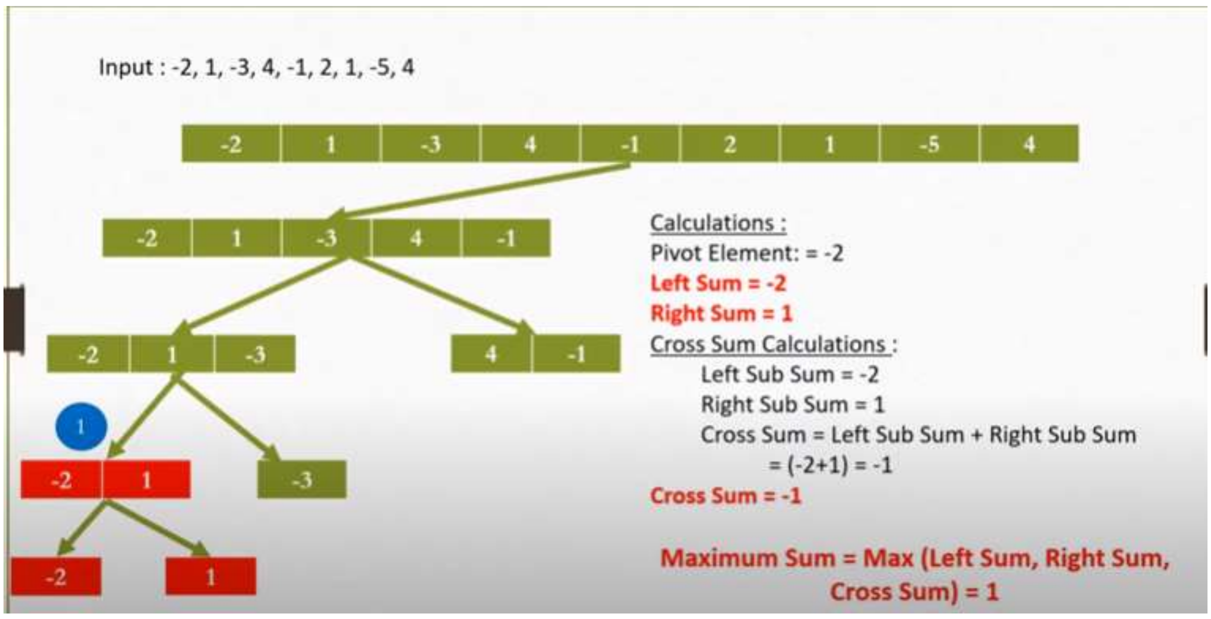

Input : -2, 1, -3, 4, -1, 2, 1, -5, 4


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4



Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

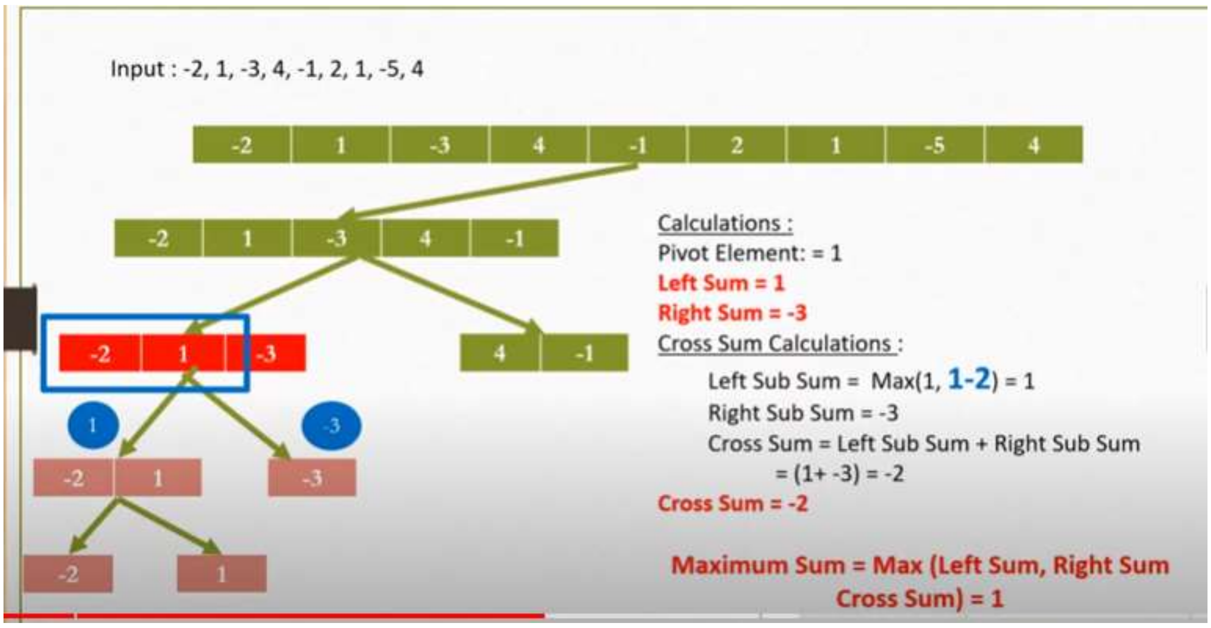
Calculations :
Pivot Element: = -2
Left Sum = -2
Right Sum = 1
Cross Sum Calculations :
Left Sub Sum = -2
Right Sub Sum = 1
Cross Sum = Left Sub Sum +Right Sub Sum
= (-2+1) = -1
Cross Sum = -1



Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

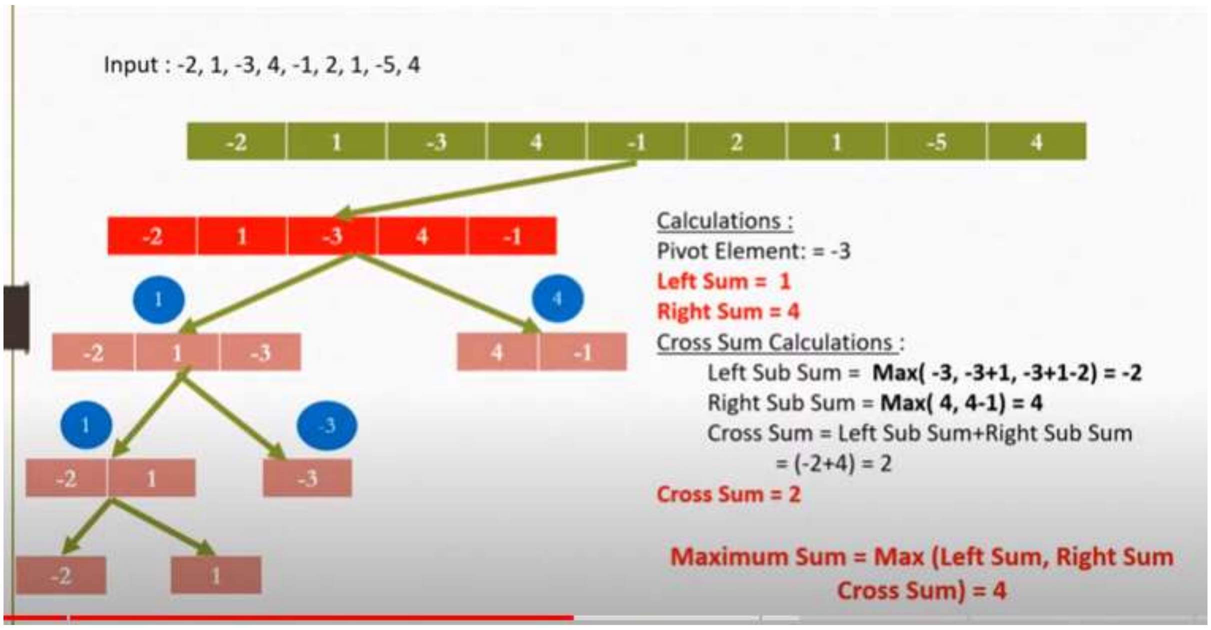
Calculations :
Pivot Element: = -2
**Left Sum = -2**
**Right Sum = 1**
Cross Sum Calculations :
Left Sub Sum = -2
Right Sub Sum = 1
Cross Sum = Left Sub Sum + Right Sub Sum
= (-2+1) = -1
**Cross Sum = -1**

**Maximum Sum = Max (Left Sum, Right Sum, Cross Sum) = 1**


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

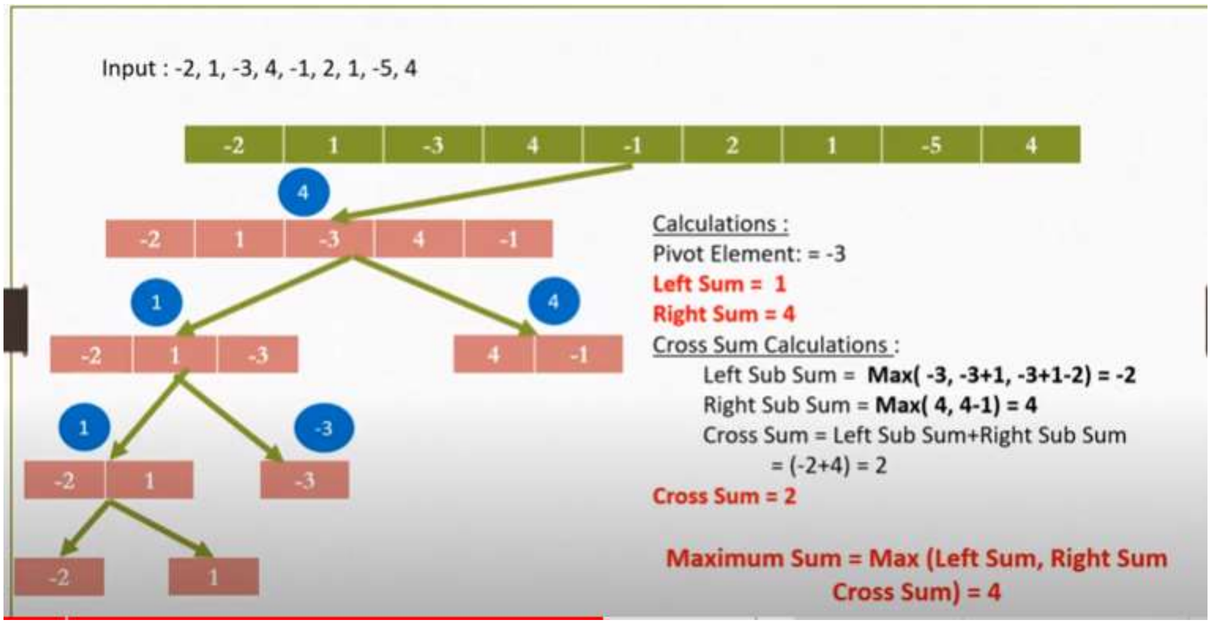
Calculations :
Pivot Element: = 1
**Left Sum = 1**
**Right Sum = -3**
Cross Sum Calculations :
Left Sub Sum = Max(1, **1-2**) = 1
Right Sub Sum = -3
Cross Sum = Left Sub Sum + Right Sub Sum
= (1+ -3) = -2
**Cross Sum = -2**

**Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 1**


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

Calculations :
Pivot Element: = -3
**Left Sum = 1**
**Right Sum = 4**
Cross Sum Calculations :
Left Sub Sum = **Max( -3, -3+1, -3+1-2) = -2**
Right Sub Sum = **Max( 4, 4-1) = 4**
Cross Sum = Left Sub Sum+Right Sub Sum
= (-2+4) = 2
**Cross Sum = 2**

**Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 4**



Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

Calculations :

Pivot Element: = -3

**Left Sum = 1**

**Right Sum = 4**

Cross Sum Calculations :

Left Sub Sum = **Max( -3, -3+1, -3+1-2) = -2**

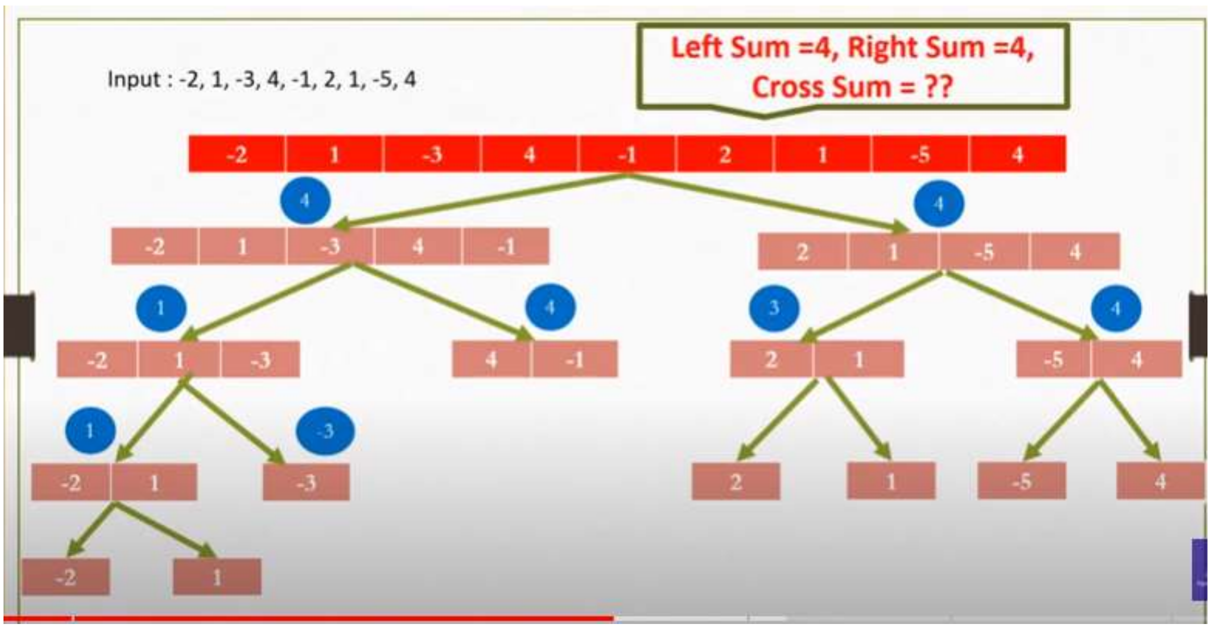
Right Sub Sum = **Max( 4, 4-1) = 4**

Cross Sum = Left Sub Sum+Right Sub Sum

= (-2+4) = 2

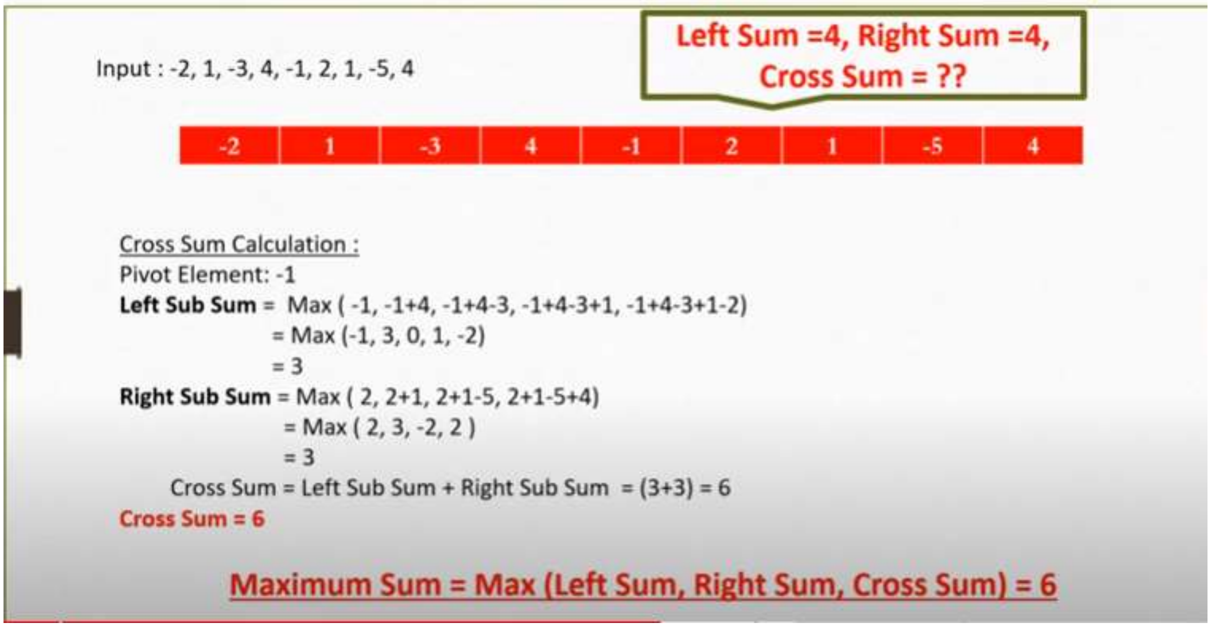
**Cross Sum = 2**

**Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 4**



Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

**Left Sum =4, Right Sum =4, Cross Sum = ??**



Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

**Left Sum =4, Right Sum =4,**
**Cross Sum = ??**

| -2 | 1 | -3 | 4 | -1 | 2 | 1 | -5 | 4 |
|---|---|---|---|---|---|---|---|---|

Cross Sum Calculation :

Pivot Element: -1

**Left Sub Sum** = Max ( -1, -1+4, -1+4-3, -1+4-3+1, -1+4-3+1-2)
= Max (-1, 3, 0, 1, -2)
= 3

**Right Sub Sum** = Max ( 2, 2+1, 2+1-5, 2+1-5+4)
= Max ( 2, 3, -2, 2 )
= 3

Cross Sum = Left Sub Sum + Right Sub Sum = (3+3) = 6

**Cross Sum = 6**

**Maximum Sum = Max (Left Sum, Right Sum, Cross Sum) = 6**