Divide and Conquer Examples

Instructor: Dr. Tarunpreet Bhatia

Assistant Professor, CSED

Thapar Institute of Engineering and Technology

Topics

- Maximum sub-array sum
- Mergesort
- Quicksort

Divide and Conquer Approach

Divide: the main problem into a number of subproblems that are smaller instances of the same problem.

Conquer: solve the subproblems recursively. If the size of subproblems is small enough, solve them directly.

Combine: the solutions to the subproblems into the solution for the original problem.

Maximum Subarray Sum

- 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.
- Some properties of this problem are:
  - If the array contains all non-negative numbers, the maximum subarray is the entire array.
  - Several different sub-arrays may have the same maximum sum.
- For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7

Possible Solutions

- **Brute force approach I : Using 3 nested loops**
- **Brute force approach II : Using 2 nested loops**
- **Divide and Conquer approach : Similar to merge sort**
- Dynamic Programming Approach I : Using an auxiliary array
- Dynamic Programming Approach II : Kadanes's Algorithm

Brute Force

- We can use brute-force to calculate sum of all possible subarray, and then get the maximum sum.
- Time complexity of brute-force solution will be O(n^3).
- The brute-force solution can be improved to O(n^2) time complexity by using a variable to store the running sum at all possible positions.

Example: Brute Force Approach I

Brute Force Approach I

```
int maxSubarraySum ( int A [] , int n)
{
    int ans = INT_MIN;
    for(int sub_array_size = 1; sub_array_size <=n; sub_array_size ++)
    {
       for(int start_index = 0; start_index < n; start_index++)
       {
          if(start_index+sub_array_size >n)
              break;
          int sum = 0;
          for(int i = start_index; i < (start_index+sub_array_size); i++)
             sum = sum + A[i];
          if(sum > ans)
             ans = sum;
       }
    }
    return ans;
}
```

Example: Brute Force Approach II

Brute Force Approach II

```
int maxSubarraySum ( int A [] , int n)
{
    int ans = INT_MIN;
    for(int start_index = 0; start_index < n; start_index++)
    {
       int sum =0;
       for(int sub_array_size = 1; sub_array_size <=n; sub_array_size ++)
       {
          if(start_index+sub_array_size >n)
              break;
          sum = sum + A[start_index + sub_array_size - 1]; //Last element
          if(sum > ans)
             ans = sum;
       }
    }
    return ans;
}
```

Divide and Conquer

- You could divide the array into two equal parts and then recursively find the maximum subarray sum of the left part and the right part. But what if the actual subarray with maximum sum is formed of some elements from the left and some elements from the right?
- The sub-array we're looking for can be in only one of three places:
  - On the left part of the array (between 0 and the mid)
  - On the right part of the array (between the mid + 1 and the end)
  - Somewhere crossing the midpoint.

Steps

1. Divide the array into two equal parts
2. Recursively calculate the maximum sum for left and right subarray
3. To find cross_sum:
    a. Iterate from mid to the starting part of the left subarray and at every point, check the maximum possible sum till that point and store in the parameter left_sum.
    b. Iterate from mid+1 to the ending point of right subarray and at every point, check the maximum possible sum till that point and store in the parameter right_sum.
    c. Add left_sum and right_sum to get the cross_sum

4. Return the maximum among (left_sum, right_sum, cross_sum)

Example of Cross-sum

Example: DAC

Divide and Conquer Approach

```
int max_sum_subarray(int A[], int p, int r)
{
  if (p == r)
  {
    return A[p];
  }

  int mid = (r+p)/2;
  int left_sum = max_sum_subarray(A, p, mid);
  int right_sum = max_sum_subarray(A, mid+1, r);
  int cross_sum = max_cross_sum(A, p, mid, r);
  return maximum(left_sum, right_sum, cross_sum);
}
```

```
int max_cross_sum(int A[], int p, int mid, int r)
{
  int left_sum = INT_MIN;
  int sum = 0;
  for (int i=mid; i>=p; i--)
  {
    sum = sum+A[i];
    if (sum>left_sum)
      left_sum = sum;
  }
  int right_sum = INT_MIN;
  sum = 0;

  for (int i=mid+1; i<=r; i++)
  {
    sum=sum+A[i];
    if (sum>right_sum)
      right_sum = sum;
  }
  return (left_sum+right_sum);
}
```

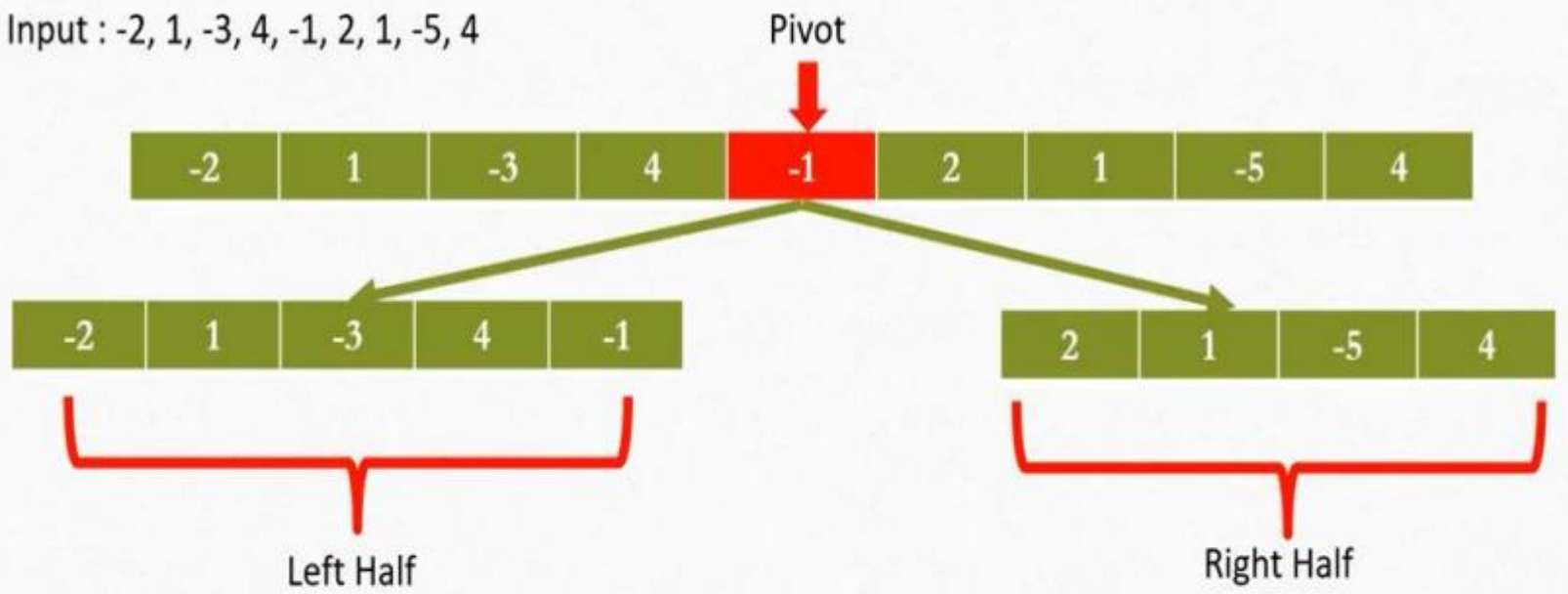
Example 1

-2	-5	6	-2	-3	1	5	-6

Recurrence Relation

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Example 2

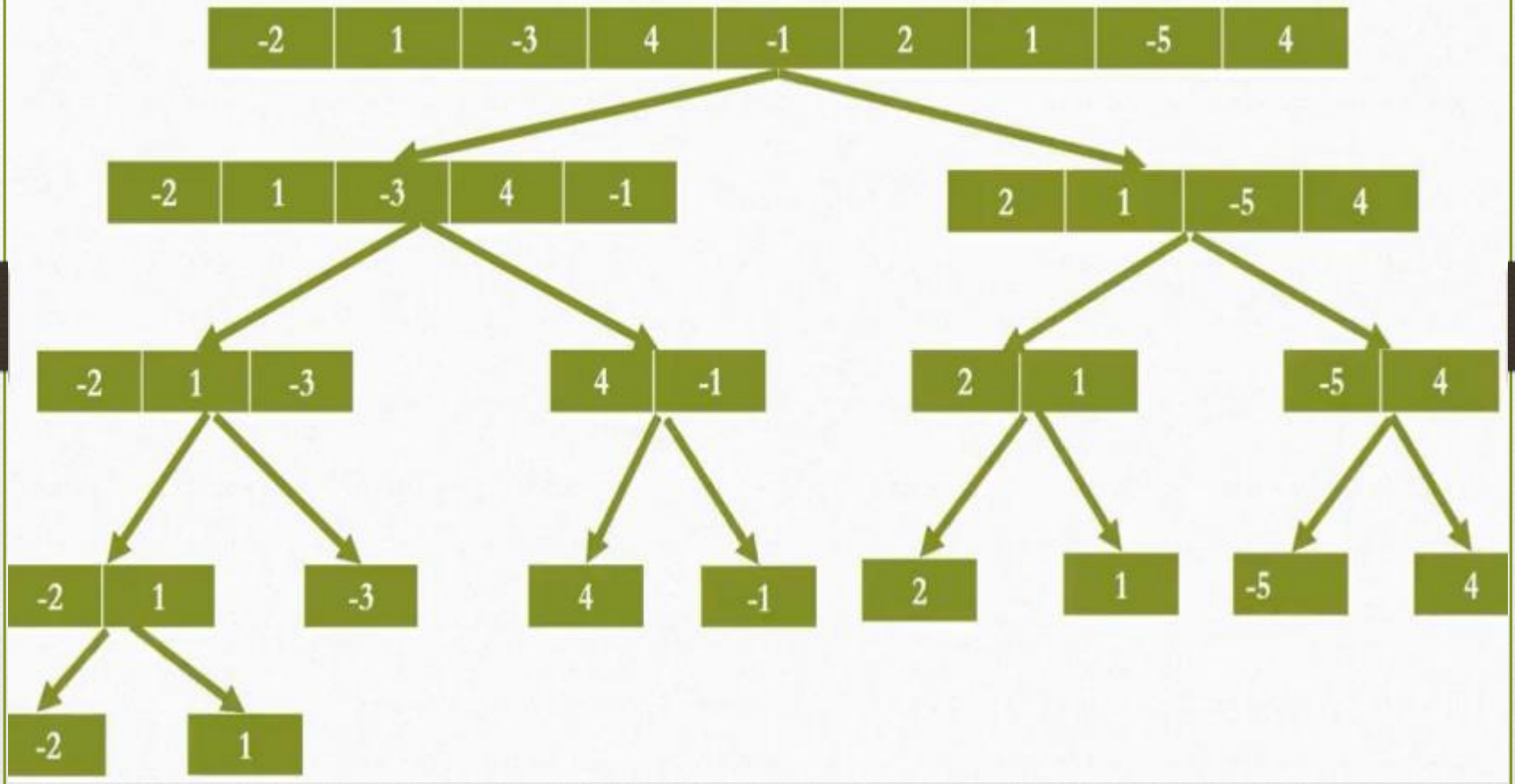


$$\text{Index of Pivot} = \frac{(\text{begin} + \text{end})}{2} = \frac{(0+8)}{2} = 4$$

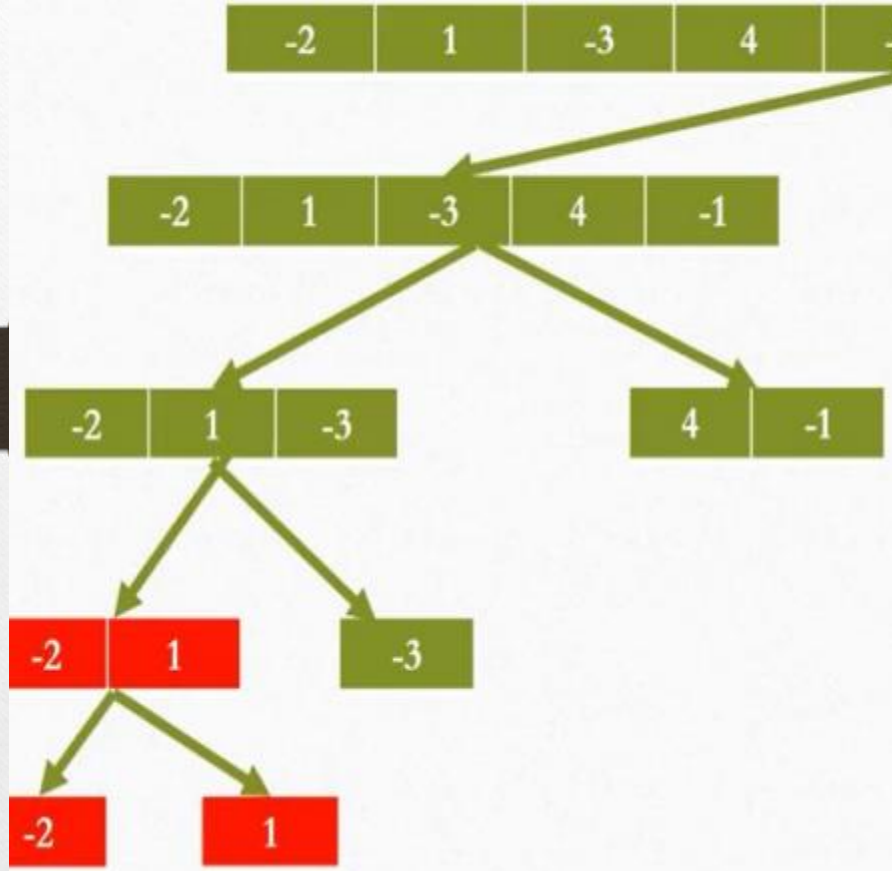
Left Half = begin to pivot (inclusive)

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



Calculations :

Pivot Element: = -2

Left Sum = -2

Right Sum = 1

Cross Sum Calculations :

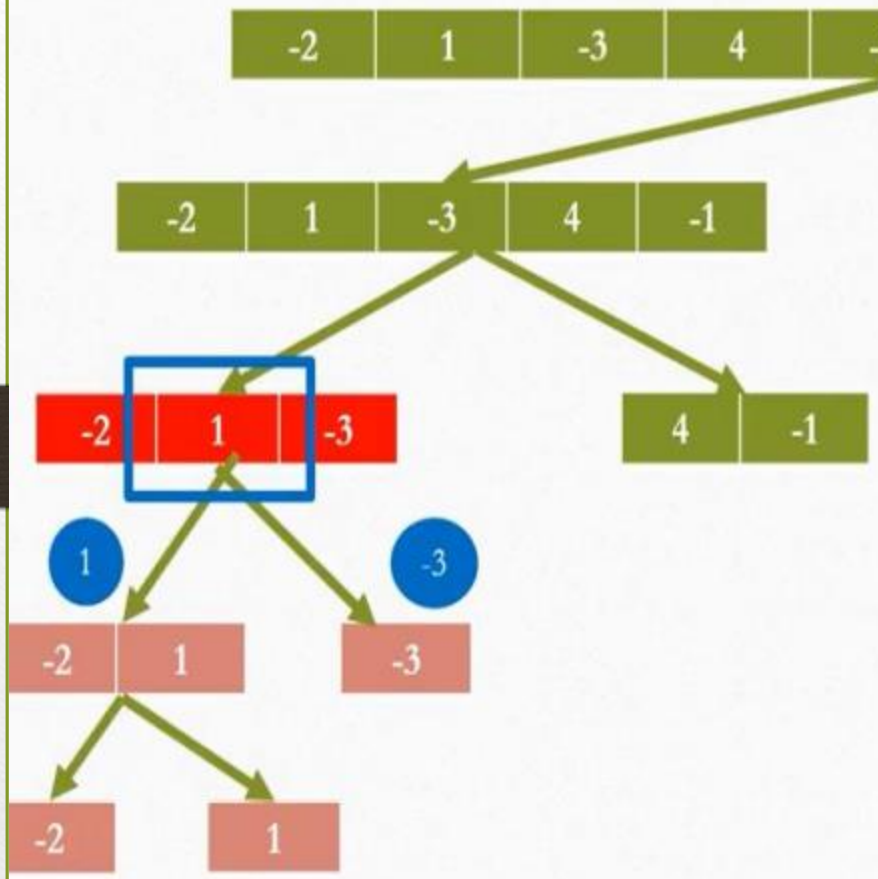
Left Sub Sum = -2

Right Sub Sum = 1

Cross Sum = Left Sub Sum +Right Sub Sum

= (-2+1) = -1

Cross Sum = -1



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

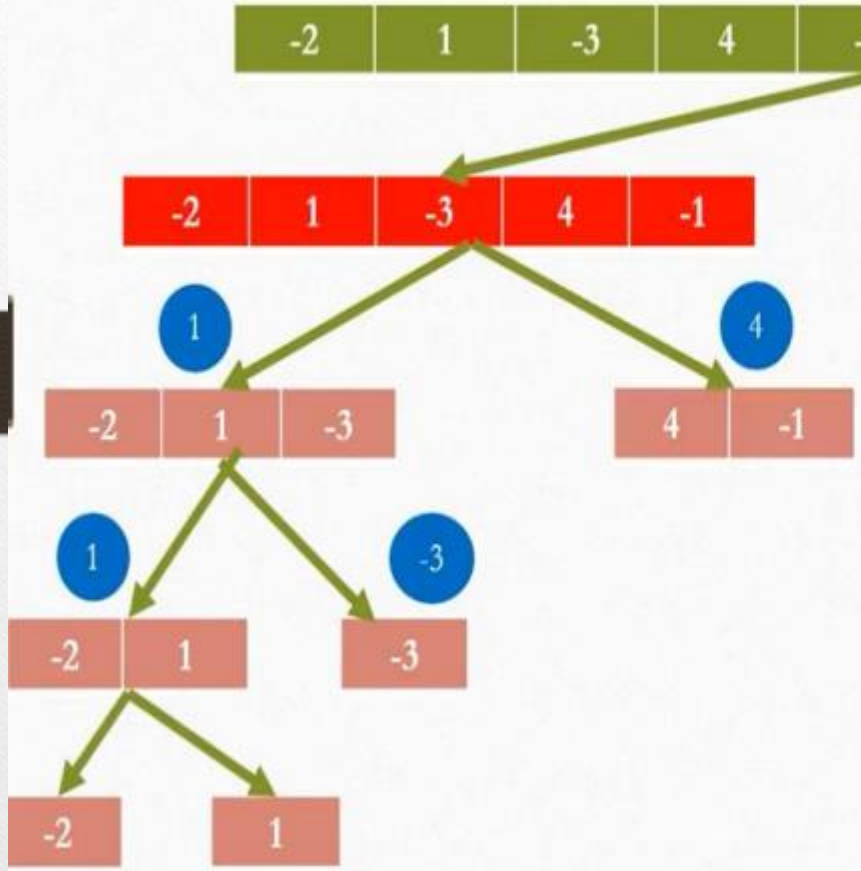
Cross Sum = Left Sub Sum + Right Sub Sum

= (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**

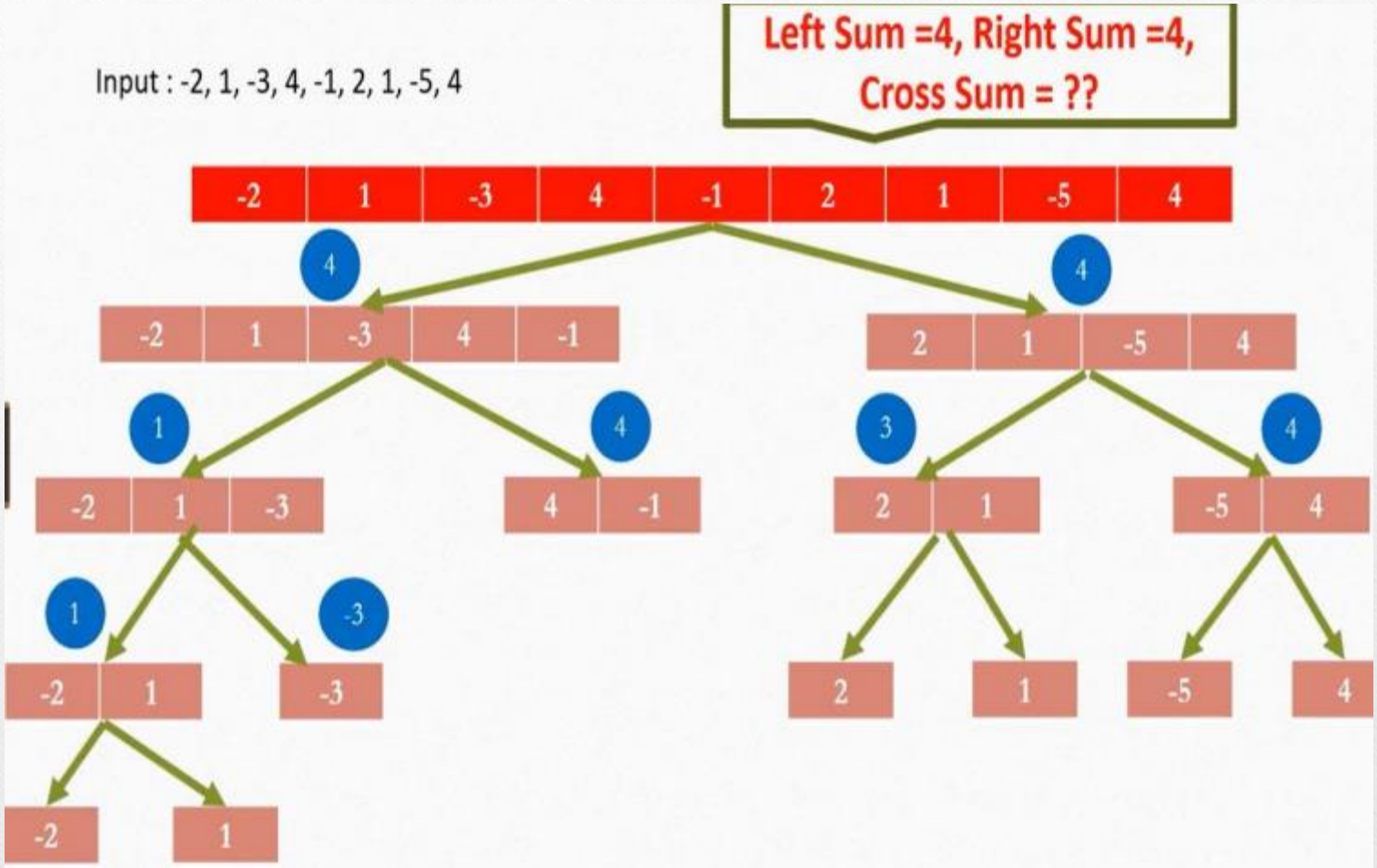
Right Sub Sum = **Max(4, 4-1) = 4**

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
Left Sum =4, Right Sum =4,
Cross Sum = ??
-2 1 -3 4 -1 2 1 -5 4
4
4
-2 1 -3 4 -1
2 1 -5 4
1
4
3
4
-2 1 -3
4 -1
2 1
-5 4
1
-3
-2 1
-3
2
1
-5
4
-2
1

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

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

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

Try this!

Apply this algorithm to the below examples to find maximum sub-array sum:

1. Input array =[-6,-2,8,3,4,-2]

2. Input array =[13,-3,-25,20,-3,-16,-23,18,20,-7,12,-5,-22,15,-4,7]

Fake Coin

- You are given n coins. They all look identical. They should all be the same weight, too -- but one is a fake, made of a lighter metal. Your neighbor has an old-fashioned balance scale that enables you to compare any two sets of coins. If it tips either to the left or to the right, you will know that the one of the sets is heavier than the other. Sadly, you aren't on speaking terms with the neighbor, so he charges you each time you weigh anything. Your task is to design an algorithm to find the fake coin in the fewest number of weighings. How many times must you use the scale?

Approach

- Suppose we divide the coins into *three* piles, where at least two of them contain the same number of coins. After weighing the equal-sized piles, we can eliminate ~2/3 of the coins! To design an algorithm, we need to be more precise.
  - If $n \bmod 3 = 0$, we can divide the coins into three piles of exactly $n/3$ apiece.
  - If $n \bmod 3 = 1$, then $n = 3k + 1$ for some k. We can divide the coins into three piles of k, k, and $k+1$. It will simplify our algorithm, though, if we split them into three piles of $k+1$, $k+1$, and $k-1$.
  - If $n \bmod 3 = 2$, then $n = 3k + 2$ for some k. We can divide the coins into three piles of $k+1$, $k+1$, and k.

This approach and binary search can be classified more generally as **decrease-by-constant-factor** algorithms.

Algorithm

```
if n = 1 then
    the coin is fake
else
    divide the coins into piles of A = ceiling(n/3), B = ceiling(n/3),
        and C = n-2*ceiling(n/3)
    weigh A and B
    if the scale balances then
      iterate with C
    else
      iterate with the lighter of A and B
```

- How many weighings does this require? Approximately $\log_3 n$.

Comparison 2-piles and 3-piles

$$k = \log_2 n \quad \rightarrow \quad n = 2^k$$

$$m = \log_3 n \quad \rightarrow \quad n = 3^m$$

$$2^k = 3^m$$

$$\log_2 2^k = \log_2 3^m$$

$$k * \log_2 2 = m * \log_2 3$$

$$k = m * \log_2 3$$

$$\frac{k}{m} = \log_2 3$$

So:

$$\frac{\log_2 n}{\log_3 n} = \log_2 3 = 1.584963...$$

Maximum single-sell profit problem

- Problem Statement: Suppose we are given an array of n integers representing stock prices on a single day. We want to find **a pair** (buyDay, sellDay), with buyDay ≤ sellDay, such that if we bought the stock on buyDay and sold it on sellDay, we would maximize our profit.
- Example: [1, 5, 3, 2] → 4; [9, 4, 2, 1] → 0

Approaches

- Brute force: $O(n^2)$ time, $O(1)$ space.
- Divide-and-Conquer: $O(n \lg n)$ time, $O(\lg n)$ space.
- Optimized Divide-and-Conquer: $O(n)$ time, $O(\lg n)$ space.

Divide and Conquer Approach

- If we have a single day, the best option is to buy on that day and then sell it back on the same day for no profit.
- Otherwise, split the array into two halves. If we think about what the optimal answer might be, it must be in one of three places:
  - The correct buy/sell pair occurs completely within the first half.
  - The correct buy/sell pair occurs completely within the second half.
  - The correct buy/sell pair occurs across both halves - we buy in the first half, then sell in the second half.