Design and Analysis of Algorithms

- ***Course Code***: BCSE304L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 1

Module:1	Design Paradigms: Greedy, Divide and Conquer Techniques	6 hours

Overview and Importance of Algorithms - Stages of algorithm development: Describing the problem, Identifying a suitable technique, Design of an algorithm, Derive Time Complexity, Proof of Correctness of the algorithm, Illustration of Design Stages -
Greedy techniques: Fractional Knapsack Problem, and Huffman coding
Divide and Conquer: Maximum Subarray, Karatsuba faster integer multiplication algorithm.

Maximum Subarray

Definition:

- The Maximum Subarray problem involves finding the contiguous subarray with the largest sum within a given array of numbers.
- The problem of maximum subarray sum is basically finding the part of an array whose elements has the largest sum.

Example:

- Let's consider an array arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4].
- The Maximum Subarray is [4, -1, 2, 1] with a sum of 6.

Maximum Subarray

Example:

array arr = [3,5,1,7,9].

The Maximum Subarray is [[3,5,1,7,9] with a max sum of 3+5+1+7+9=25.

Example:

array arr = [3,-1,-1,10,-3,-2,4].

The Maximum Subarray is [3,-1,-1,10] with a max sum of 3-1-1+10=11.

Maximum Subarray

Time complexity in brute force approach:

A brute force approach would involve checking all possible subarrays and their sums, leading to a time complexity of O(n^2), where n is the length of the array.

Time complexity in Dive and Conquer approach:

The Divide and Conquer approach has a time complexity of O(n log n), where n is the length of the array.

Recurrence relation: $T(n)=2T(\mathrm{n}/2)+O(n)$

Maximum Subarray: brute force approach

```
Algorithm MaximumSubarrayBruteforce(arr):
  maxsum = MIN_INT          // Initialize maxsum to the smallest possible integer value
  n = length of arr         // Get the length of the array
    // Outer loop: iterate through all possible starting indices
  for i = 0 to n-1 do:
    sum = 0                 // Initialize the sum for the current starting index
    // Inner loop: iterate through subarrays starting from index i
    for j = i to n-1 do:
      sum = sum + arr[j]    // Add the current element to the running sum
      // Update maxsum if the current sum is greater
      if sum >= maxsum then:
        maxsum = sum
  return maxsum             // Return the overall maximum subarray sum
```

A brute force approach would involve checking all possible subarrays and their sums, leading to a time complexity of O(n^2), where n is the length of the array.

Maximum Subarray

Time complexity in Dive and Conquer approach:

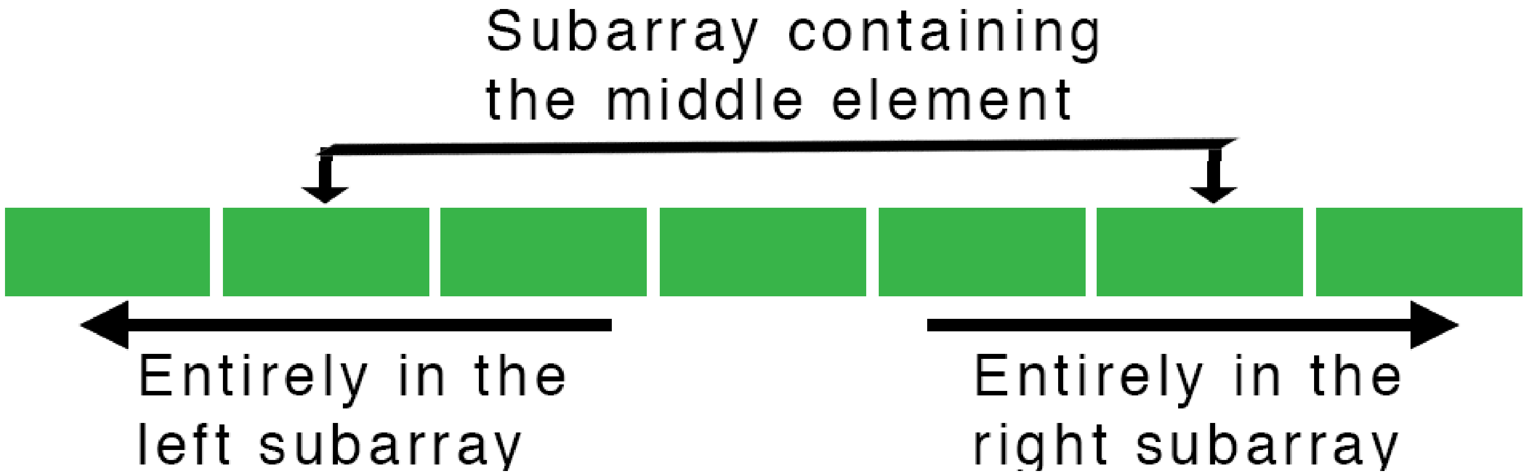
The Divide and Conquer approach has a time complexity of O(n log n), where n is the length of the array.

Recurrence relation: $T(n)=2T(n/2)+O(n)$

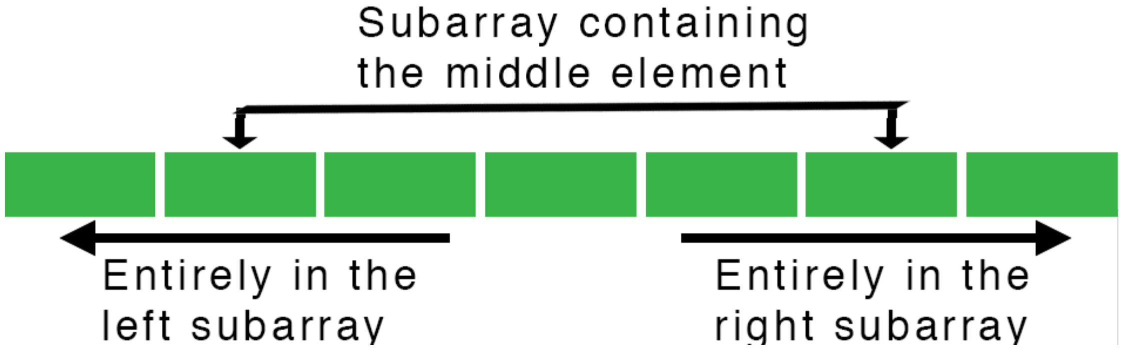
Example:

array arr = [3,-1,-1,10,-3,-2,4].

The Maximum Subarray is [3,-1,-1,10] with a max sum of 3-1-1+10=11.



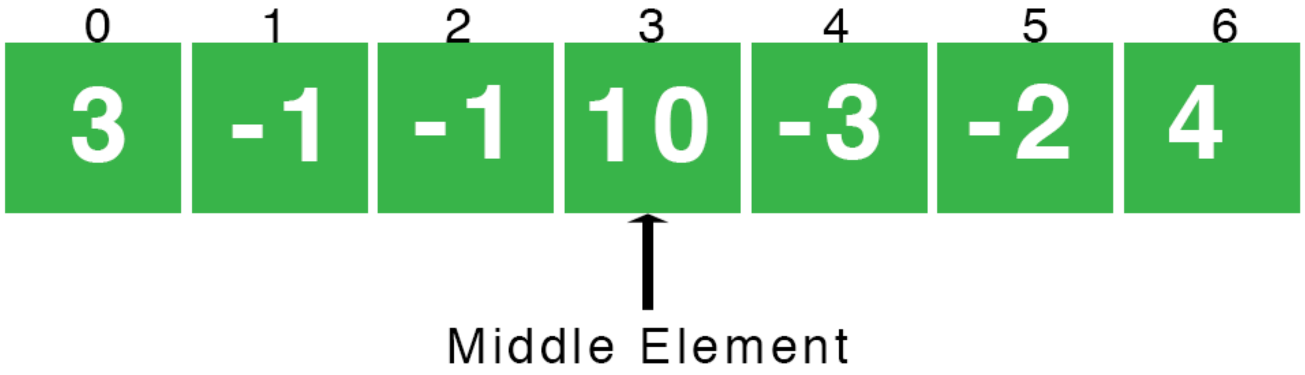
Maximum Subarray



Example:

array arr = [3,-1,-1,10,-3,-2,4].

The Maximum Subarray is [3,-1,-1,10] with a max sum of 3-1-1+10=11**.**



Left subarray
sum = 0
Start from index 3:
10 > 0 => sum = 10
10-1=9<sum(10)
9-1=8<sum(10)
8+3=11>sum => sum = 11

Right subarray
sum = 0
Start from index 4:
-3 < sum(0), sum is 0
-3-2=-5<sum(0), sum is 0
-5+4=-1<sum(0), sum is 0

Final crossing sum = 11+0=0 from index 0 to 3

Maximum Subarray

```
function find_maximum_subarray(arr, low, high):
    if low == high:
        return low, high, arr[low]

    mid = (low + high) / 2

    left_low, left_high, left_sum = find_maximum_subarray(arr, low, mid)
    right_low, right_high, right_sum = find_maximum_subarray(arr, mid + 1, high)
    cross_low, cross_high, cross_sum = find_max_crossing_subarray(arr, low, mid, high)

    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    else:
        return cross_low, cross_high, cross_sum
```

Maximum Subarray

```
function find_max_crossing_subarray(arr, low, mid, high):
    left_sum = negative infinity
    sum = 0
    max_left = 0

    for i from mid downto low:
        sum += arr[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i

    right_sum = negative infinity
    sum = 0
    max_right = 0

    for j from mid + 1 to high:
        sum += arr[j]
        if sum > right_sum:
            right_sum = sum
            max_right = j

    return max_left, max_right, left_sum + right_sum
```

Maximum Subarray

```
function find_max_crossing_subarray(arr, low, mid, high):
    left_sum = negative infinity
    sum = 0
    max_left = 0

    for i from mid downto low:
        sum += arr[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i

    right_sum = negative infinity
    sum = 0
    max_right = 0

    for j from mid + 1 to high:
        sum += arr[j]
        if sum > right_sum:
            right_sum = sum
            max_right = j

    return max_left, max_right, left_sum + right_sum
```

```
function find_maximum_subarray(arr, low, high):
    if low == high:
        return low, high, arr[low]

    mid = (low + high) / 2

    left_low, left_high, left_sum = find_maximum_subarray(arr, low, mid)
    right_low, right_high, right_sum = find_maximum_subarray(arr, mid + 1, high)
    cross_low, cross_high, cross_sum = find_max_crossing_subarray(arr, low, mid, high)

    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    elif right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    else:
        return cross_low, cross_high, cross_sum
```

```
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

low, high, max_sum = find_maximum_subarray(arr, 0, length(arr) - 1)

max_subarray = arr[low:high + 1]

print("Maximum Subarray:", max_subarray, "with sum", max_sum)
```

Maximum Subarray

Recurrence relation: $T(n)=2T(n/2)+O(n)$

- This recurrence relation signifies that the problem is divided into two subproblems of size n/2, and the merging step (combining solutions) takes O(n) time.
- In the context of the Maximum Subarray problem, the recursive algorithm involves finding the maximum subarray in the left half, the right half, and a maximum subarray crossing the midpoint. The maximum of these three will be the solution for the entire array.

This can be represented as:

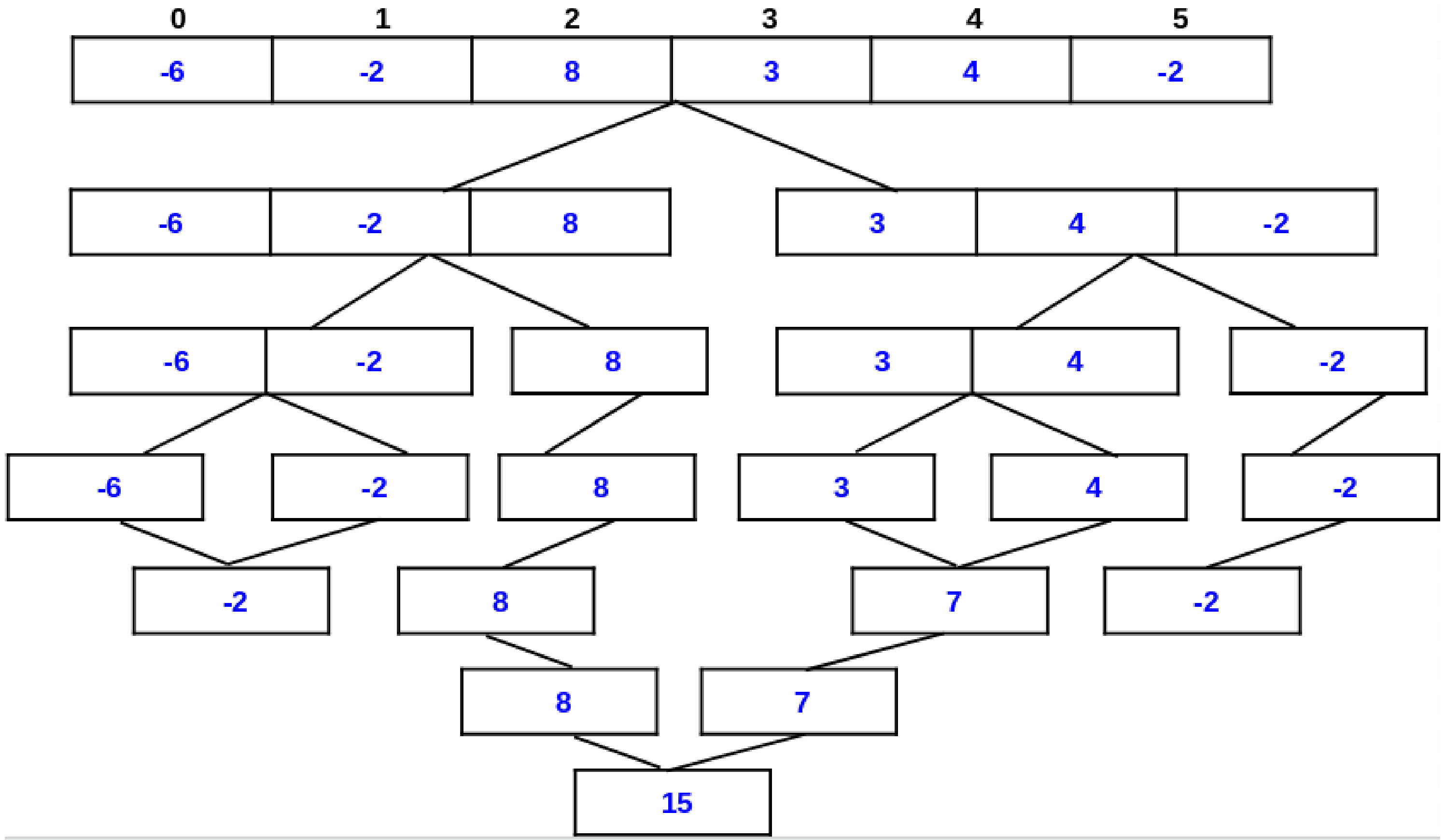
max_subarray(arr,low,high)=max(max_subarray(arr,low,mid),max_subarray(arr,mid+1,high),max_crossing_subarray(arr,low,mid,high))

Here, max_crossing_subarray(arr,low,mid,high) finds the maximum subarray that crosses the midpoint.

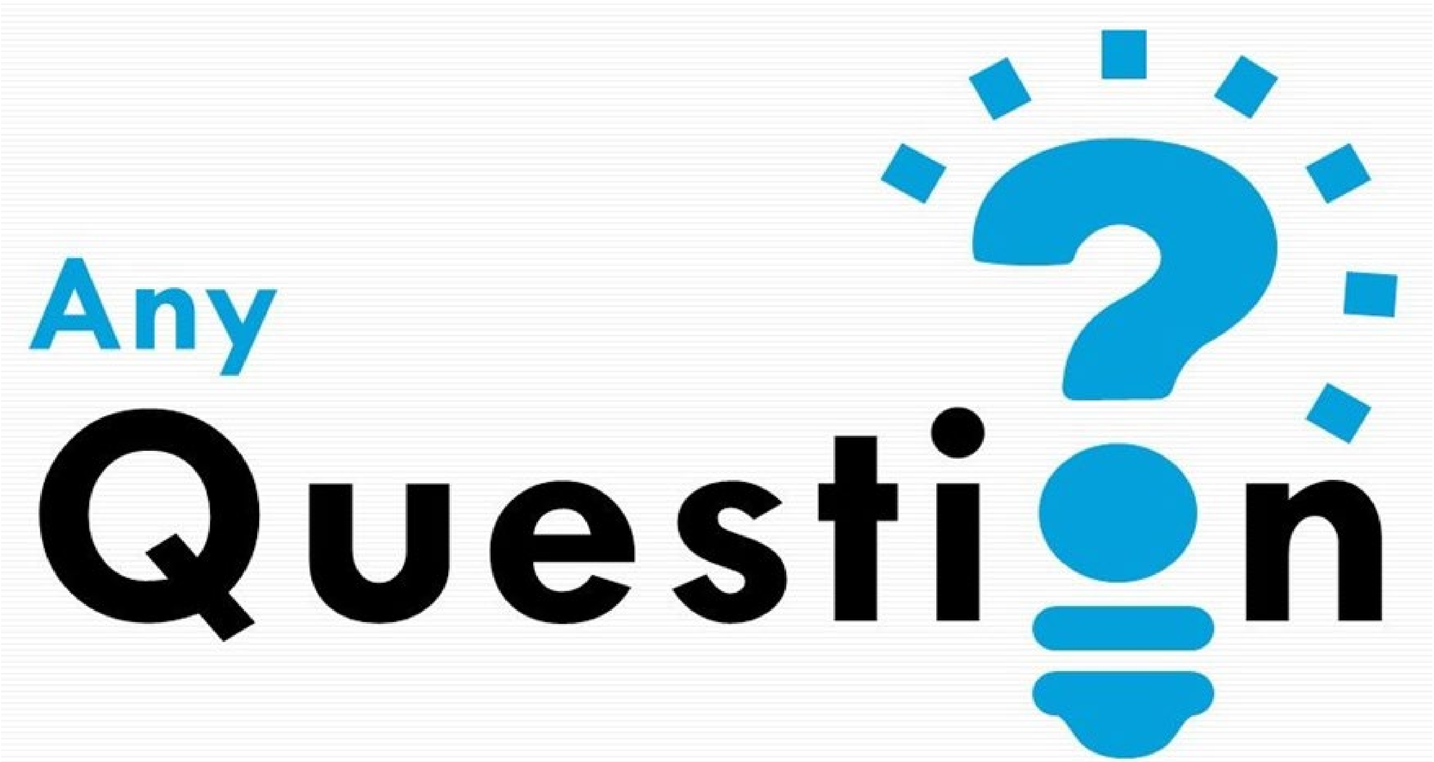
Maximum Subarray

Another example:

- Input array =[-6,-2,8,3,4,-2].
- here maximum contiguous subarray sum from left-side is 8 .
- maximum contiguous subarray sum from right-side is 7.
- midpoint cross subarray sum is 15.
- hence the maximum subarray sum is 15.



Karatsuba faster integer multiplication algorithm


Any
Question


PresenterMedia


Thank You!


FOR YOUR
ATTENTION