Divide and Conquer Algorithms

MD. MUKTAR HOSSAIN

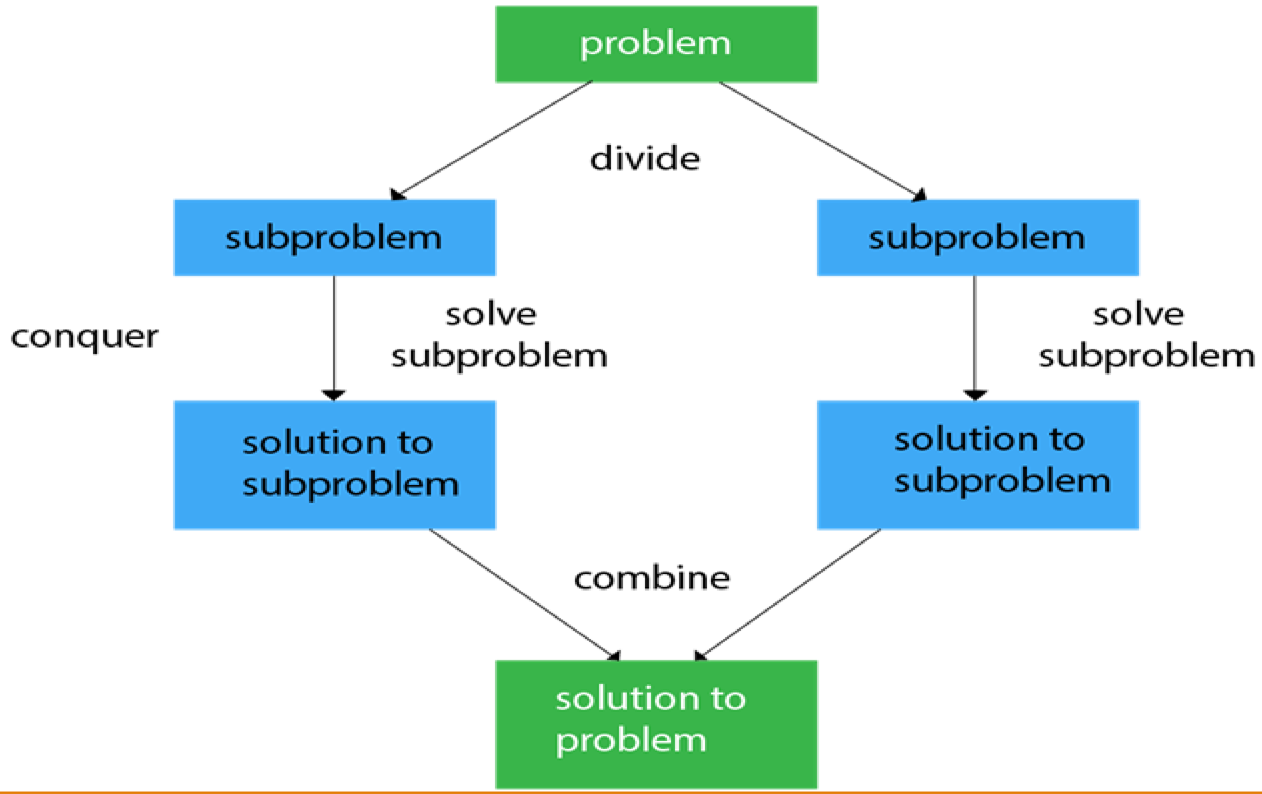
LECTURER

DEPT. OF CSE

VARENDRA UNIVERSITY

Divide and Conquer

Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually and then merging them to find solution to the original problem.



Divide and Conquer Cont’d

General Steps of Divide and Conquer:

Divide: Split the problem into two or more smaller subproblems.

Conquer: Solve each of the subproblems. If the subproblem is small enough, solve it directly (often recursively).

Combine: Combine the solutions to the subproblems to solve the original problem.

Key Characteristics:

- ***Recursion***: Divide and conquer typically uses recursion to solve subproblems.
- ***Divide and Combine Operations***: The problem is split into subproblems, and the subproblem results are merged to solve the original problem.
- ***Efficiency***: Many divide and conquer algorithms, especially when used in sorting or searching, can achieve optimal time complexity.

Divide and Conquer Cont’d

Pros

Efficiency: Many divide and conquer algorithms (like binary search, merge sort and quick sort) are very efficient compared to simpler approaches.

Parallelism: The independent subproblems in divide and conquer are often suitable for parallel execution, leading to faster algorithms on multi-core machines.

Cons

Overhead: Recursive calls can lead to overhead, especially if the recursion depth is high.

Space Complexity: Some divide and conquer algorithms (e.g., merge sort) require additional memory space, leading to higher space complexity.

Maximum Subarray Sum

Maximum Subarray Sum refers to the largest sum that can be obtained from a contiguous subarray within a given array.

Maximum Subarray Sum problem can be solved with different time complexities.

1. **O(n²)** – Quadratic Time Complexity (Brute Force)
2. **O(n log n)** – Divide and Conquer
3. **O(n)** – Kadane’s Algorithm (Optimal Solution)

Maximum Subarray Sum – $O(n^2)$ (Brute Force Approach)

Brute Force approach considers all possible subarrays and computes their sums.

Algorithm:

1. Iterate over all possible starting indices.
2. Iterate over all possible ending indices.
3. Compute the sum for each subarray and track the maximum sum.

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	**-4**	**5**	**-2**	**4**	**-3**

sum = 0

maxSum = **–2147483648**

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++)
{
        int sum = 0;
        for (int j = i; j < n; j++)
{
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum;
}
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

i 3 j	-4	5	-2	4	-3

sum = 3

maxSum = –2147483648

maxSum = 3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++)
{
        int sum = 0;
        for (int j = i; j < n; j++)
{
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum;
}
```

Maximum Subarray Sum – $O(n^2)$ (Brute Force Approach)

i 3	-4 j	5	-2	4	-3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

sum = -1

maxSum = 3

maxSum = 3

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

i 3	-4	5 j	-2	4	-3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

sum = 4

maxSum = 3

maxSum = 4

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

i 3	-4	5	-2 j	4	-3

sum = 2

maxSum = 4

maxSum = 4

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
      int sum = 0;
      for (int j = i; j < n; j++) {
            sum += arr[j];
            maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2)
(Brute Force Approach)

i 3	-4	5	-2	4 j	-3

sum = 6

maxSum = 4

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

i 3	-4	5	-2	4	-3 j

sum = 3

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
      int sum = 0;
      for (int j = i; j < n; j++) {
             sum += arr[j];
             maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

🟥 3	i -4 j	5	-2	4	-3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

sum = 0

maxSum = 6

Maximum Subarray Sum – O(n^2)
(Brute Force Approach)

3	i -4 j	5	-2	4	-3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

sum = -4

maxSum = 6

maxSum = 6

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	i -4	5 j	-2	4	-3

sum = 1

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	i -4	5	-2 j	4	-3

sum = -1

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	i -4	5	-2	4 j	-3

sum = 3

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	i -4	5	-2	4	-3 j

sum = 0

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
      int sum = 0;
      for (int j = i; j < n; j++) {
             sum += arr[j];
             maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	i 5 j	-2	4	-3

sum = 0

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	i 5 j	-2	4	-3

sum = 5

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                 sum += arr[j];
                 maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	i 5	-2 j	4	-3

sum = 3

maxSum = 6

maxSum = 6

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                 sum += arr[j];
                 maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	i 5	-2	4 j	-3

sum = 7

maxSum = 6

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	i 5	-2	4	-3 j

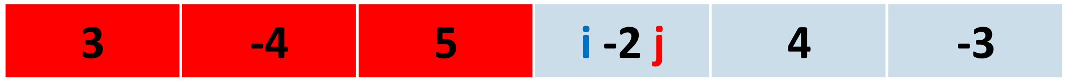
sum = 4

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)



sum = 0

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	i -2 j	4	-3

sum = -2

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – $O(n^2)$ (Brute Force Approach)

3	-4	5	i -2	4 j	-3

sum = 2

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	i -2	4	-3 j

sum = -1

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – $O(n^2)$
(Brute Force Approach)

3	-4	5	-2	i 4 j	-3

sum = 0

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	-2	i 4 j	-3

sum = 4

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	-2	i 4	-3 j

sum = -1

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
      int sum = 0;
      for (int j = i; j < n; j++) {
             sum += arr[j];
             maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	-2	4	i -3 j

sum = 0

maxSum = 7

maxSum = 7

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	-2	4	i -3 j

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = i; j < n; j++) {
                sum += arr[j];
                maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

sum = -3

maxSum = 7

maxSum = 7

Maximum Subarray Sum – O(n^2) (Brute Force Approach)

3	-4	5	-2	4	-3

```
int maxSubarraySum(int arr[], int n)
{
int maxSum = INT_MIN;
for (int i = 0; i < n; i++) {
      int sum = 0;
      for (int j = i; j < n; j++) {
             sum += arr[j];
             maxSum = max(maxSum, sum);
}
}
return maxSum; }
```

maxSum = 7

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

Divide and Conquer approach divides the array into two halves, solves the problem for each half recursively, and combines the results.

Algorithm:

1. Find the maximum subarray sum in the left half.
2. Find the maximum subarray sum in the right half.
3. Find the maximum subarray sum that crosses the midpoint.
4. Return the maximum of the three.

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

3	-4	5	-2	4	-3

```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

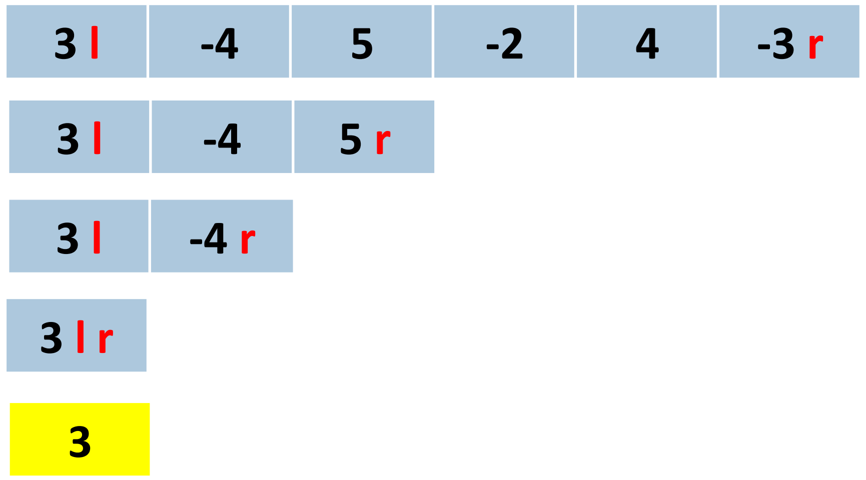
leftMax =

rightMax =

crossMax =

max =

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

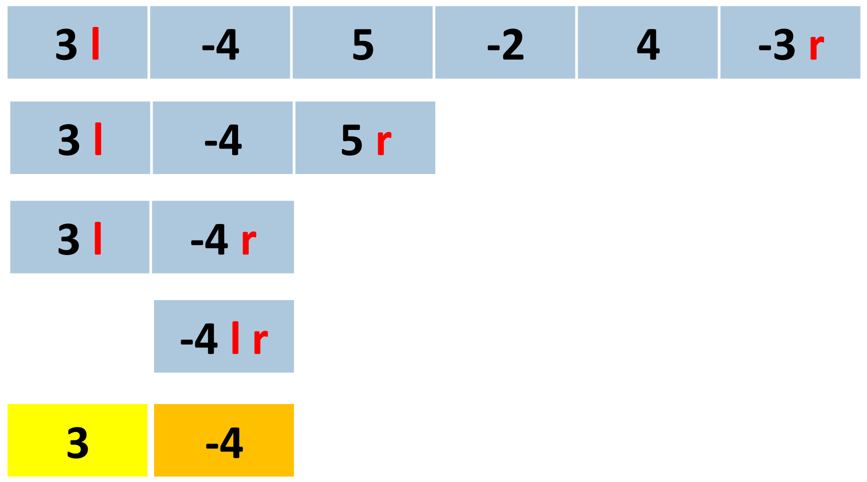
leftMax =

rightMax =

crossMax =

max =

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

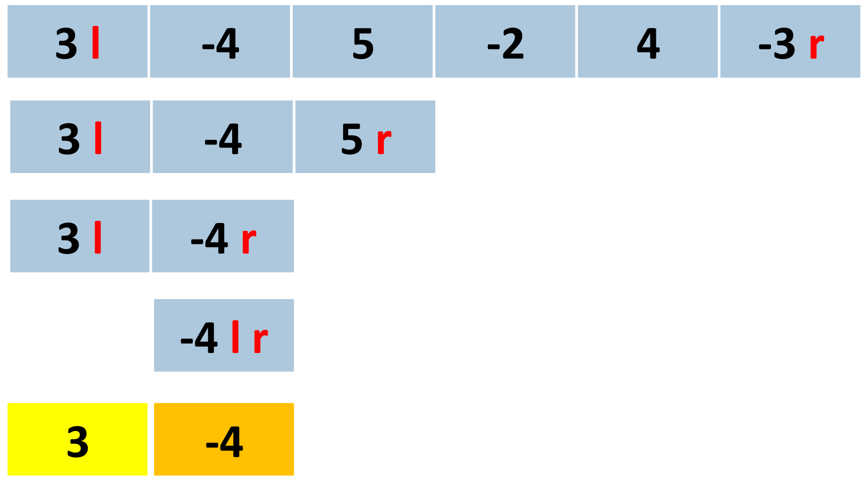
leftMax = 3

rightMax =

crossMax =

max =

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

leftMax = 3

rightMax = -4

crossMax =

max =

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

3 l	-4	5	-2	4	-3 r

3 l	-4	5 r

3 l	-4 r

3	-4

```
int maxCrossingSum(int arr[], int left, int mid, int
right) {

int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;

for (int i = mid; i >= left; i--) {
          sum += arr[i];
          leftSum = max(leftSum, sum);     }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
          sum += arr[i];

          rightSum = max(rightSum, sum);     }

return leftSum + rightSum;}
```

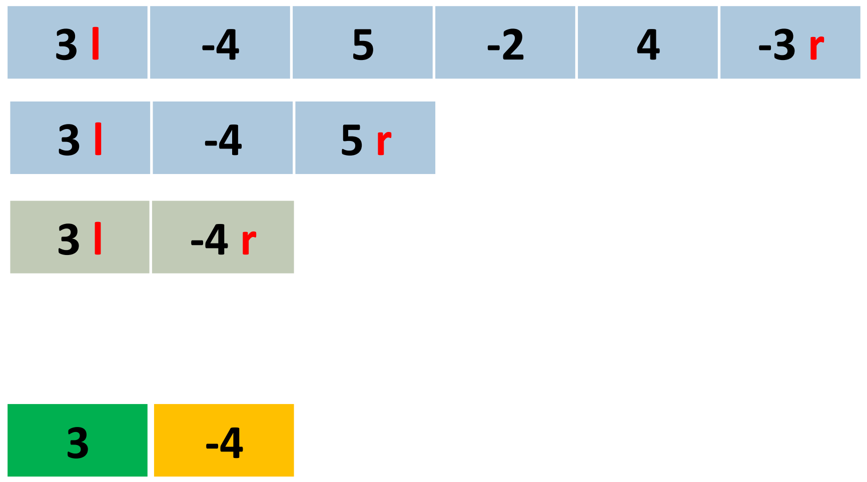
leftMax = 3

rightMax = -4

crossMax = -1

max =

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

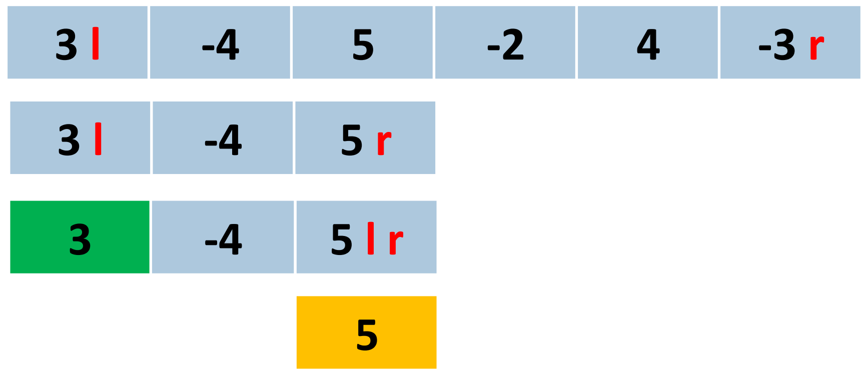
leftMax = 3

rightMax = -4

crossMax = -1

max = 3

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

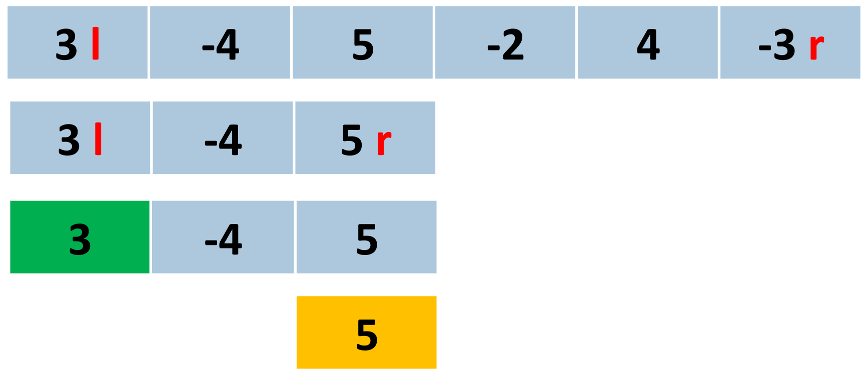
leftMax = 3

rightMax = -4

crossMax = -1

max = 3

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxCrossingSum(int arr[], int left, int mid, int
right) {

int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;

for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);     }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];

        rightSum = max(rightSum, sum);     }

return leftSum + rightSum;}
```

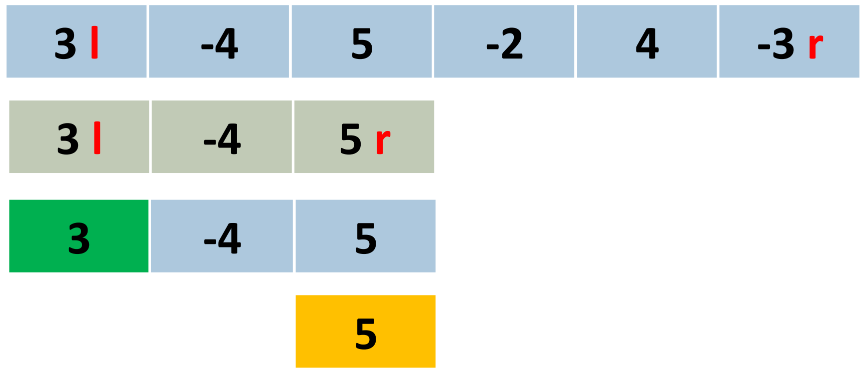
leftMax = 3

rightMax = 5

max = 3

crossMax = -1

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxCrossingSum(int arr[], int left, int mid, int
right) {
int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;
for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);    }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        rightSum = max(rightSum, sum);    }
return leftSum + rightSum;}
```

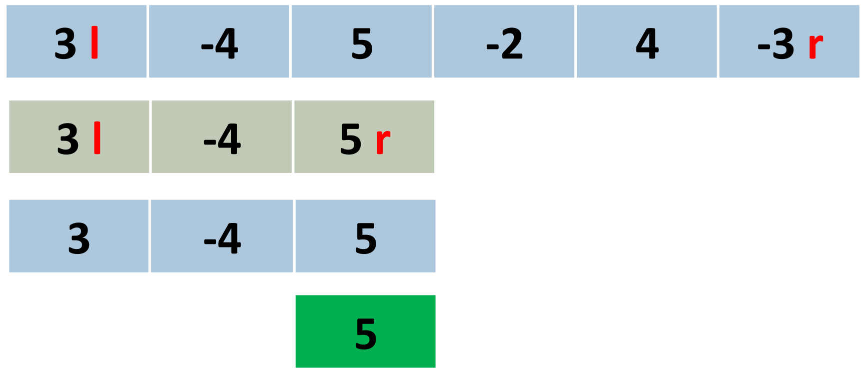
leftMax = 3

rightMax = 5

crossMax = 4

max = 3

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

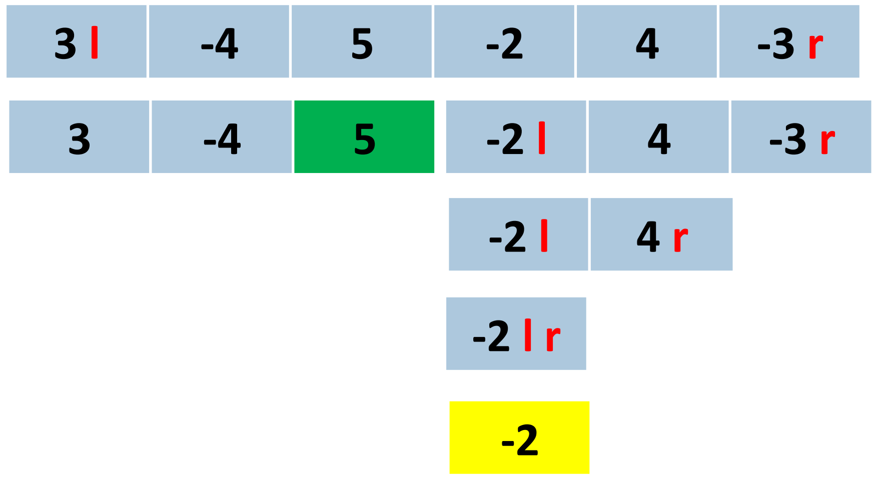
leftMax = 3

rightMax = 5

crossMax = 4

max = 5

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

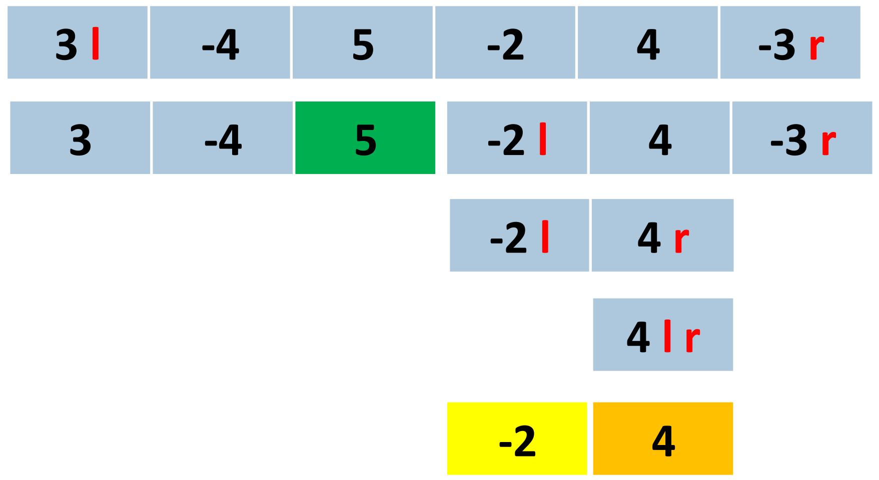
leftMax = 3

rightMax = 5

crossMax = 4

max = 5

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

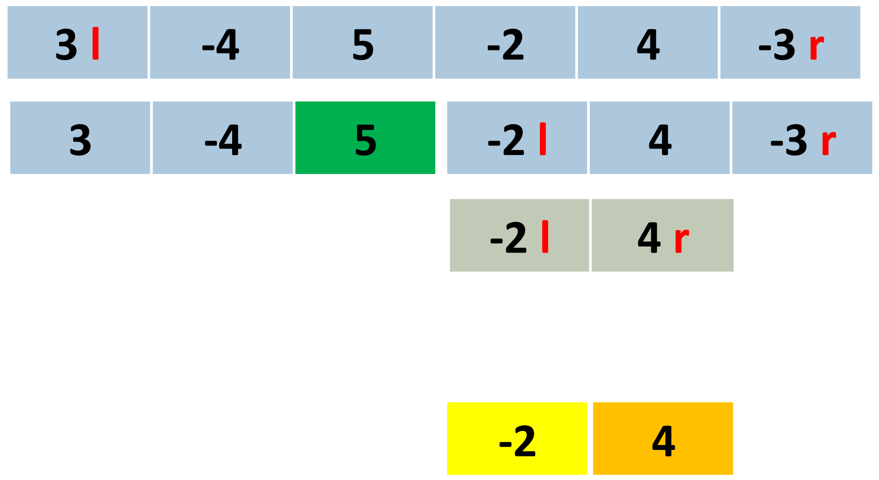


```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
        return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

```
leftMax = -2
rightMax = 4
crossMax = 4
```

```
max = 5
```

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxCrossingSum(int arr[], int left, int mid, int
right) {

int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;

for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);    }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];

        rightSum = max(rightSum, sum);    }

return leftSum + rightSum;}
```

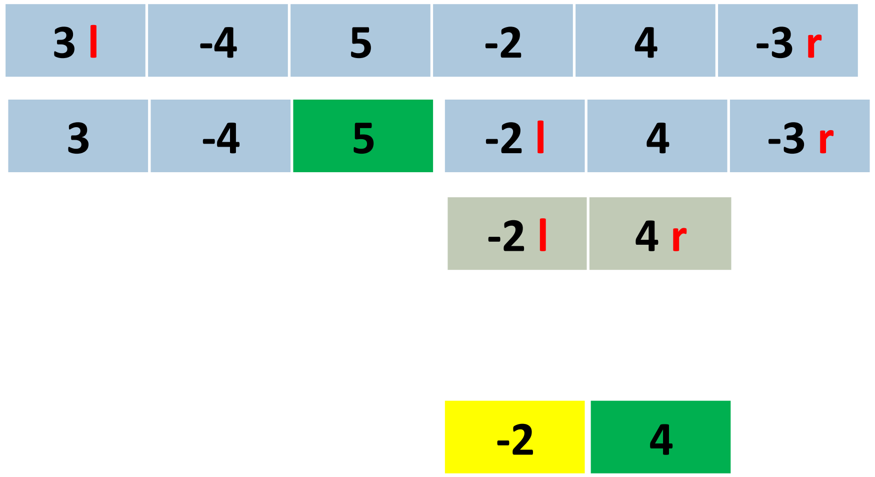
leftMax = -2

rightMax = 4

max = 5

crossMax = 2

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
        return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

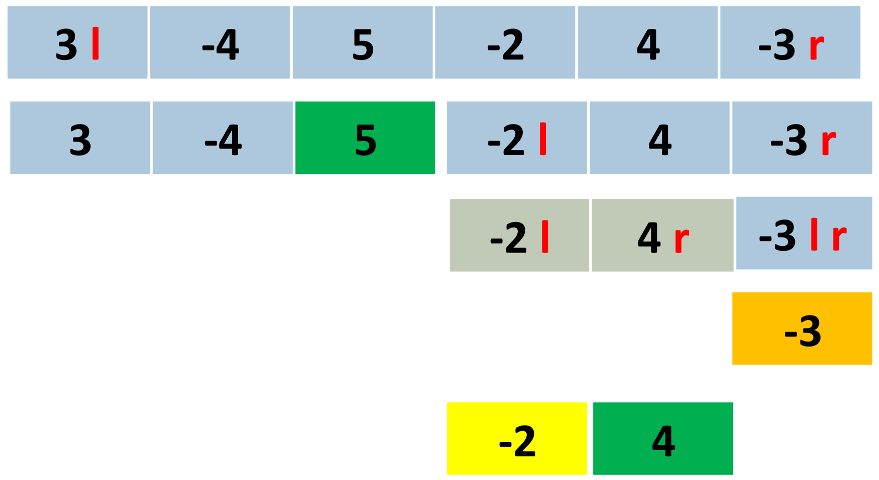
leftMax = -2

rightMax = 4

crossMax = 2

max = 4

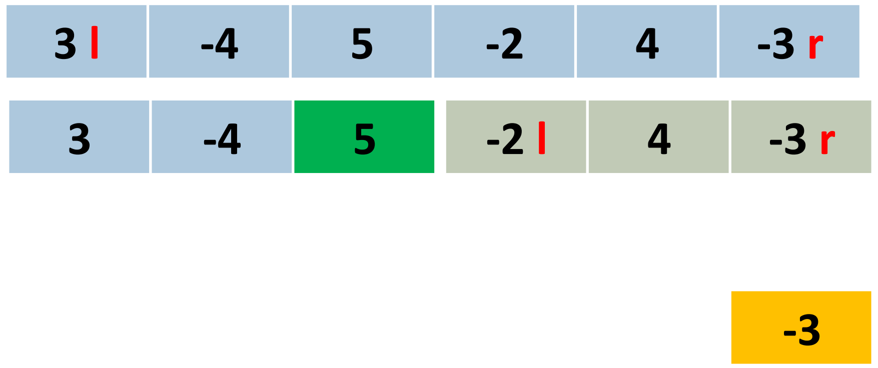
Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

```
leftMax = -2

rightMax = 4          max = 4

crossMax = 2
```

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxCrossingSum(int arr[], int left, int mid, int
right) {

int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;

for (int i = mid; i >= left; i--) {
            sum += arr[i];
            leftSum = max(leftSum, sum);     }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
            sum += arr[i];

            rightSum = max(rightSum, sum);     }

return leftSum + rightSum;}
```

```
leftMax = -2

rightMax = -3          max = 4

crossMax = 2
```

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

3 l	-4	5	-2	4	-3 r
3	-4	5	-2 l	4	-3 r

```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
       return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

leftMax = -2

rightMax = -3

crossMax = 2

max = 2

Maximum Subarray Sum – O(n logn) (Divide and Conquer)

3 l	-4	5	-2	4	-3 r

3	-4	5	-2	4	-3

```
int maxCrossingSum(int arr[], int left, int mid, int
right) {

int leftSum = INT_MIN, rightSum = INT_MIN, sum = 0;

for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = max(leftSum, sum);    }
sum = 0;
for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];

        rightSum = max(rightSum, sum);    }

return leftSum + rightSum;}
```

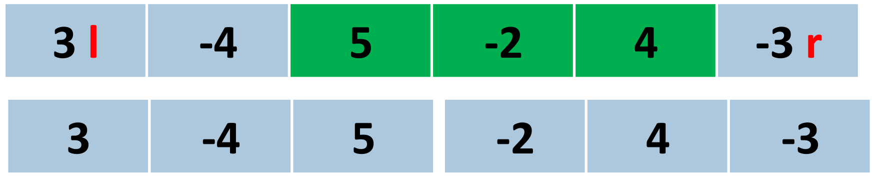
leftMax = 5

rightMax = 2

crossMax = 7

max = 2

Maximum Subarray Sum – O(n logn) (Divide and Conquer)



```
int maxSubarraySumDC(int arr[], int left, int right) {

if (left == right)
        return arr[left];

int mid = (left + right) / 2;

int leftMax = maxSubarraySumDC(arr, left, mid);

int rightMax = maxSubarraySumDC(arr, mid + 1, right);

int crossMax = maxCrossingSum(arr, left, mid, right);

return max({leftMax, rightMax, crossMax});}
```

leftMax = 5

rightMax = 2

crossMax = 7

max = 7

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

Kadane's algorithm efficiently finds the maximum subarray sum in O(n) time by maintaining a running sum.

Algorithm:

1. Initialize *maxSum* as the smallest possible integer.
2. Initialize *currentSum* to 0.
3. Iterate through the array:
   - Add the current element to *currentSum*.
   - If *currentSum* exceeds *maxSum*, update *maxSum*.
   - If *currentSum* drops below 0, reset it to 0.
4. Return *maxSum*.

Maximum Subarray Sum – O(n) (Kadane’s Algorithm)

3	-4	5	-2	4	-3

currentSum = 0

maxSum = **–2147483648**

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	**-4**	**5**	**-2**	**4**	**-3**

currentSum = 0

maxSum = **–2147483648**

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

🟨 3	-4	5	-2	4	-3

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

currentSum = 3

maxSum = 3

Maximum Subarray Sum – O(n) (Kadane’s Algorithm)

3	-4	5	-2	4	-3

currentSum = -1

maxSum = 3

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 0

maxSum = 3

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 0

maxSum = 3

```
int kadane(int arr[], int n) {

int maxSum = INT_MIN, currentSum = 0;

for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;

    }

return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 5

maxSum = 5

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 5

maxSum = 5

```
int kadane(int arr[], int n) {

int maxSum = INT_MIN, currentSum = 0;

for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;

    }

return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	**-4**	**5**	**-2**	**4**	**-3**

currentSum = 3

maxSum = 5

```
int kadane(int arr[], int n) {

int maxSum = INT_MIN, currentSum = 0;

for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;

    }

return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane’s Algorithm)

3	-4	5	-2	4	-3

currentSum = 3

maxSum = 5

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 7

maxSum = 7

```
int kadane(int arr[], int n) {

int maxSum = INT_MIN, currentSum = 0;

for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;

    }

return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

3	-4	5	-2	4	-3

currentSum = 7

maxSum = 7

```
int kadane(int arr[], int n) {

int maxSum = INT_MIN, currentSum = 0;

for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;

    }

return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)

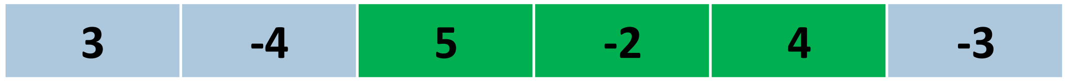
3	-4	5	-2	4	-3

currentSum = 4

maxSum = 7

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Maximum Subarray Sum – O(n) (Kadane's Algorithm)



currentSum = 4

maxSum = 7

```
int kadane(int arr[], int n) {
int maxSum = INT_MIN, currentSum = 0;
for (int i = 0; i < n; i++) {
        currentSum += arr[i];
        maxSum = max(maxSum, currentSum);
        if (currentSum < 0) currentSum = 0;
    }
return maxSum;}
```

Comparison of Approaches

Algorithm	Time Complexity	Space Complexity	Approach
Brute Force	$O(n^2)$	$O(1)$	Checks all subarrays
Divide & Conquer	$O(n \log n)$	$O(\log n)$	Recursively divides array
Kadane’s Algorithm	$O(n)$	$O(1)$	Iterative approach

THANK YOU