PROJECT REPORT

Subarray Summit – Maximum Subarray Problem

Subject Name: Design and Analysis of Algorithms

Subject Code : 23CSH-301

Submitted to:

Er. Mohammad Shaqlain

(E17211)

Submitted by:

Name: Akshara Chauhan

UID: 23BCS11410

Section: KRG_2B

Aim:

To design and implement algorithms to solve the Maximum Subarray Problem, which involves finding the contiguous subarray within a one-dimensional array that has the largest sum.

Objectives:

1. To understand and implement different algorithmic approaches for solving the maximum subarray problem.
2. To compare the efficiency of Brute Force**,** Divide and Conquer, and **Kadane's Algorithm**.
3. To analyze the time complexity and identify the most optimal solution.
4. To demonstrate the use of algorithm design paradigms like **Divide & Conquer** and **Dynamic Programming**.

Problem Definition

Given an array of integers (both positive and negative), find the contiguous subarray which has the maximum possible sum**.**

Example:
Input Array = [−2, −3, 4, −1, −2, 1, 5, −3]
Output = 7
(Contiguous subarray = [4, −1, −2, 1, 5])

Algorithms Used

We implement three algorithms to show efficiency improvement:

1. Brute Force Approach

Compute the sum of all possible subarrays and track the maximum sum.
Time Complexity: O(n^2)
Space Complexity: O(1)

---

2. Divide and Conquer Approach

Divide the array into two halves.
Find:
Maximum subarray sum in the left half
Maximum subarray sum in the right half
Maximum subarray sum that crosses the midpoint

Return the maximum of these three.

Recurrence Relation:

$$T(n) = 2T(n/2) + O(n)$$

Solving gives: T(n) = O(n log n)
Space Complexity: O(log n) (due to recursion stack)

---

3. Kadane's Algorithm (Dynamic Programming)

Traverse array, maintaining a running sum and reset it to 0 if it becomes negative.
Keep track of the maximum sum seen so far.
Time Complexity: O(n)
Space Complexity: O(1)

C++ Implementation

```
#include <iostream>

#include <climits>
```

```
using namespace std;

// ---------- BRUTE FORCE APPROACH ----------
int maxSubarrayBruteForce(int arr[], int n) {
    int maxSum = INT_MIN;
    for (int i = 0; i < n; i++) {
        int currentSum = 0;
        for (int j = i; j < n; j++) {
            currentSum += arr[j];
            maxSum = max(maxSum, currentSum);
        }
    }
    return maxSum;
}

// ---------- DIVIDE AND CONQUER APPROACH ----------
int maxCrossingSum(int arr[], int l, int m, int h) {
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--) {
        sum += arr[i];
        left_sum = max(left_sum, sum);
    }

    sum = 0;
    int right_sum = INT_MIN;
```

```
    for (int i = m + 1; i <= h; i++) {
        sum += arr[i];
        right_sum = max(right_sum, sum);
    }

    return left_sum + right_sum;
}

int maxSubarrayDivideConquer(int arr[], int l, int h) {
    if (l == h)
        return arr[l];

    int m = (l + h) / 2;

    int left_max = maxSubarrayDivideConquer(arr, l, m);
    int right_max = maxSubarrayDivideConquer(arr, m + 1, h);
    int cross_max = maxCrossingSum(arr, l, m, h);

    return max({left_max, right_max, cross_max});
}

// ---------- KADANE'S ALGORITHM ----------
int maxSubarrayKadane(int arr[], int n) {
    int maxSoFar = INT_MIN, maxEndingHere = 0;

    for (int i = 0; i < n; i++) {
```

```
        maxEndingHere += arr[i];
        maxSoFar = max(maxSoFar, maxEndingHere);

        if (maxEndingHere < 0)
            maxEndingHere = 0;
    }
    return maxSoFar;
}

// ---------- MAIN FUNCTION ----------
int main() {
    int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Array: ";
    for (int x : arr) cout << x << " ";
    cout << "\n\n";

    cout << "Brute Force Max Sum: " << maxSubarrayBruteForce(arr, n) << endl;
    cout << "Divide & Conquer Max Sum: " << maxSubarrayDivideConquer(arr, 0, n - 1) << endl;
    cout << "Kadane's Algorithm Max Sum: " << maxSubarrayKadane(arr, n) << endl;

    return 0;
}
```

```
Array: -2 -3 4 -1 -2 1 5 -3

Brute Force Max Sum: 7
Divide & Conquer Max Sum: 7
Kadane's Algorithm Max Sum: 7
```

Figure 1: Output

Complexity Analysis

Approach	Time Complexity	Space Complexity	Remarks
Brute Force	$O(n^2)$	O(1)	Simple but inefficient
Divide & Conquer	O(n log n)	O(log n)	Uses recursion
Kadane's Algorithm	O(n)	O(1)	Most efficient

✓ **Recurrence Relation (for Divide & Conquer)**

$$T(n) = 2T(n/2) + O(n)$$

By applying the **Master Theorem**,

$$T(n) = O(n\log\ n)$$

Applications

1. **Financial Analysis:** To identify the most profitable period in stock price changes.
2. **Data Analytics:** Detect the most positive trend in sensor or signal data.

3. **Machine Learning:** Feature extraction for time-series data.
4. **Genomics:** Identifying regions with maximum activity in DNA sequences.
5. **Weather Forecasting:** Finding longest warm or cold streaks.

Conclusion

The **Maximum Subarray Problem** efficiently demonstrates the evolution of algorithmic design:

- From **Brute Force ($O(n^2)$)**
- To **Divide & Conquer ($O(n \log n)$)**
- To **Kadane's Algorithm ($O(n)$)**

Kadane's Algorithm provides the most optimal solution, making it ideal for large datasets and real-world applications where quick computation is essential.