EAST WEST UNIVERSITY

DEPARTMENT OF CSE

Course Code and Name Course Code: CSE246 Course Name: Algorithms	
Project Title: Maximum Sum Interval	
Semester and Year Semester: Fall Year: 2023	**Project No:12** **Section: 02**
Name: Md Ripon Al Mamun Id: 2021-2-60-083 Name: Md. Nazmul Hasan Id: 2021-3-60-150 Name: Sabbir Hossain Rifat Id: 2021-2-60-038 Name: Marjuk Ibna Belayet Id:2022-1-60-383	**Course Instructor and information** Jesan Ahmed Ovi Senior Lecturer, Department of Computer Science and Engineering, East West University, Bangladesh.

Problem Statement:

The maximum sum subarray problem involves finding the contiguous subarray within a one-dimensional array of numbers that has the largest sum. Develop an algorithm using either Divide and Conquer or Dynamic Programming, excluding Brute Force, to efficiently solve this problem. The algorithm should adhere to the following properties:

1. If the array contains all non-positive numbers, the solution is the number in the array with the smallest magnitude.

2. If the array contains all non-negative numbers, the problem is trivial, and the maximum sum is the sum of all the elements in the list.

3. An empty set is not a valid input.

4. There can be multiple different subarrays that achieve the same maximum sum for the problem.

Example: For the array [-5, 6, 7, 1, 4, -8, 16], the maximum sum is 26, achieved by adding 6 + 7 + 1 + 4 + -8 + 16.

System Requirements:

We used the AMD Ryzen 5 5600G with Radeon Graphics with 16GB RAM in Windows 11 Home. We used Codeblocks IDE. We can also run this program on any lower or higher end devices.

System Design:

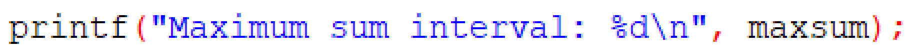
Algorithm Selection: The chosen algorithm for this implementation is a Divide and Conquer approach. This algorithm divides the problem into subproblems, solves them recursively, and then combines the solutions to find the maximum sum subarray. The algorithm efficiently handles large arrays and has a time complexity of **O (n log n),** making it suitable for this problem.

Input/Output Handling:

- **Input:** The program prompts the user to enter the number of elements in the array and then takes the array elements as input.

    - Input Validation: The program should include input validation to handle cases where the user provides invalid inputs, such as a non-numeric value or an array size less than 1.

Output: The program prints the maximum sum interval as the output.

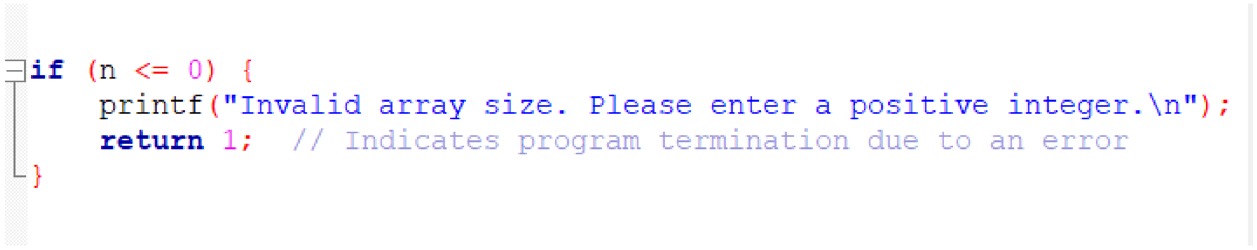


```c
printf("Maximum sum interval: %d\n", maxsum);
```

Error Handling:

- The program currently assumes that the user will enter valid inputs. However, it's advisable to include input validation to handle potential errors.

- For instance, you can check whether the input array size is greater than 0 to ensure it's a valid array size. Additionally, you can validate each element entered to ensure it's a numeric value.

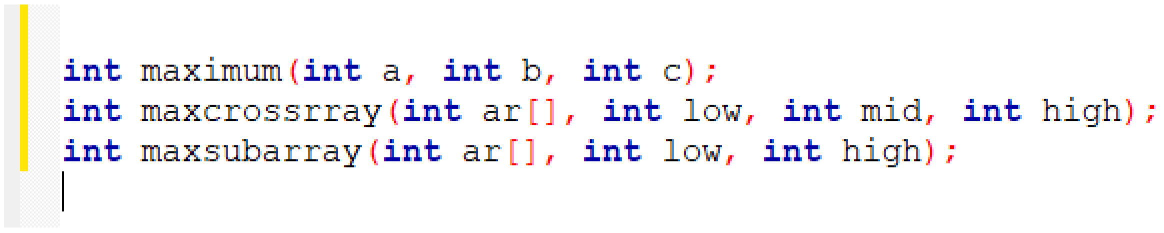


```c
if (n <= 0) {
    printf("Invalid array size. Please enter a positive integer.\n");
    return 1;   // Indicates program termination due to an error
}
```

Modularity:

- The algorithm is modular in nature, with separate functions for calculating the maximum sum across an array (‘maxcrossrray’), calculating the maximum subarray (‘maxsubarray’), and finding the maximum of three values (‘maximum’).

- This design promotes code reusability and maintainability. If you need to make improvements or modifications in the future, you can focus on individual functions without affecting the entire program.

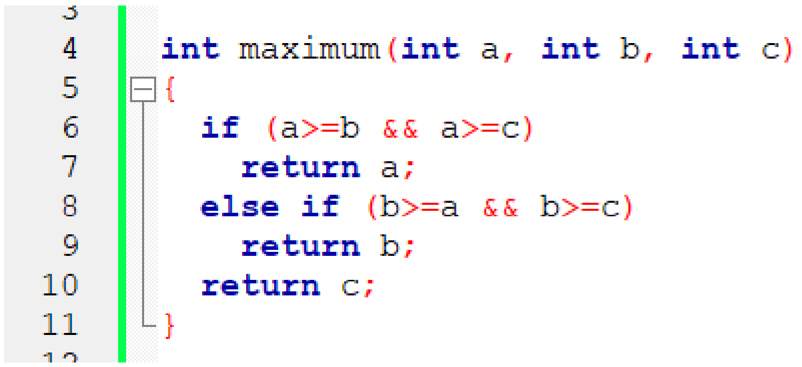


```
int maximum(int a, int b, int c);
int maxcrossrray(int ar[], int low, int mid, int high);
int maxsubarray(int ar[], int low, int high);
```

By addressing input validation, ensuring modularity, and considering potential errors, the system design aims to create a robust and maintainable program for finding the maximum sum interval in a given array.

<u>Implementation:</u>

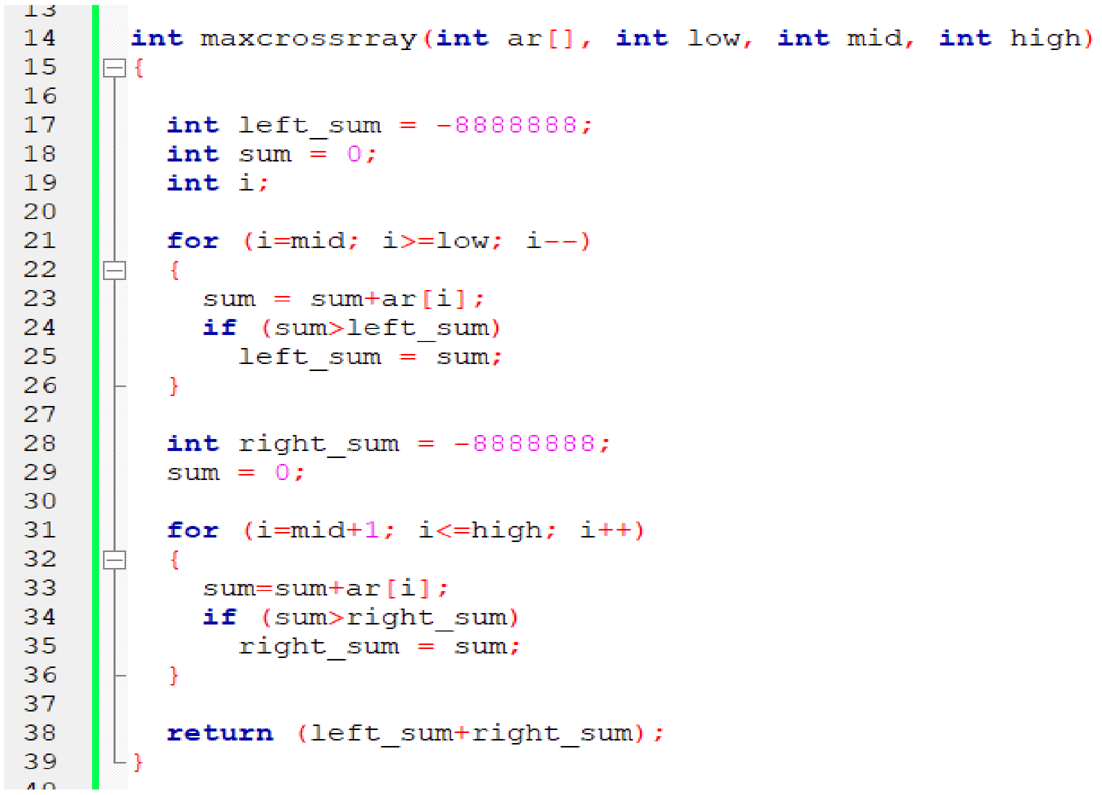
1. **‘Maximum’ Function:**



```
int maximum(int a, int b, int c)
{
  if (a>=b && a>=c)
    return a;
  else if (b>=a && b>=c)
    return b;
  return c;
}
```

This function takes three integers as input (a, b, c) and returns the maximum of the three using conditional statements. It is used later to find the maximum of three values in the 'maxsubarray' function.

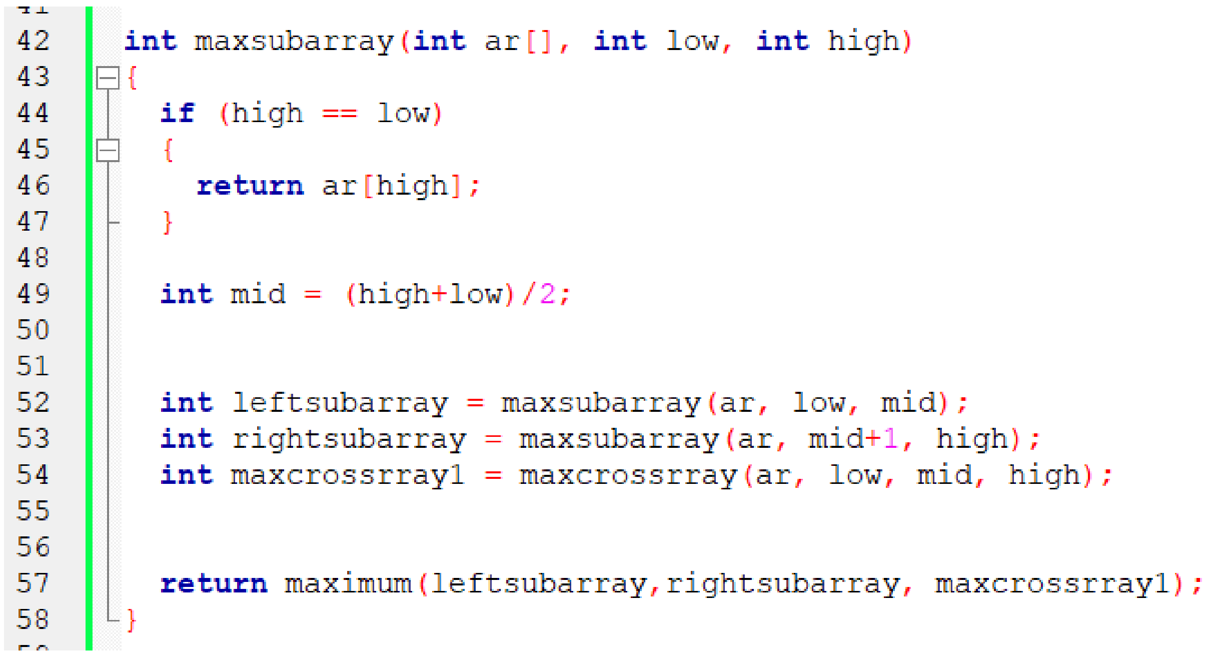
2. **maxcrossrray Function:**



```
int maxcrossrray(int ar[], int low, int mid, int high)
{

  int left_sum = -8888888;
  int sum = 0;
  int i;

  for (i=mid; i>=low; i--)
  {
    sum = sum+ar[i];
    if (sum>left_sum)
      left_sum = sum;
  }

  int right_sum = -8888888;
  sum = 0;

  for (i=mid+1; i<=high; i++)
  {
    sum=sum+ar[i];
    if (sum>right_sum)
      right_sum = sum;
  }

  return (left_sum+right_sum);
}
```

This function calculates the maximum subarray sum that crosses the midpoint of the array. It uses two loops to find the maximum sum in the left and right subarrays separately.

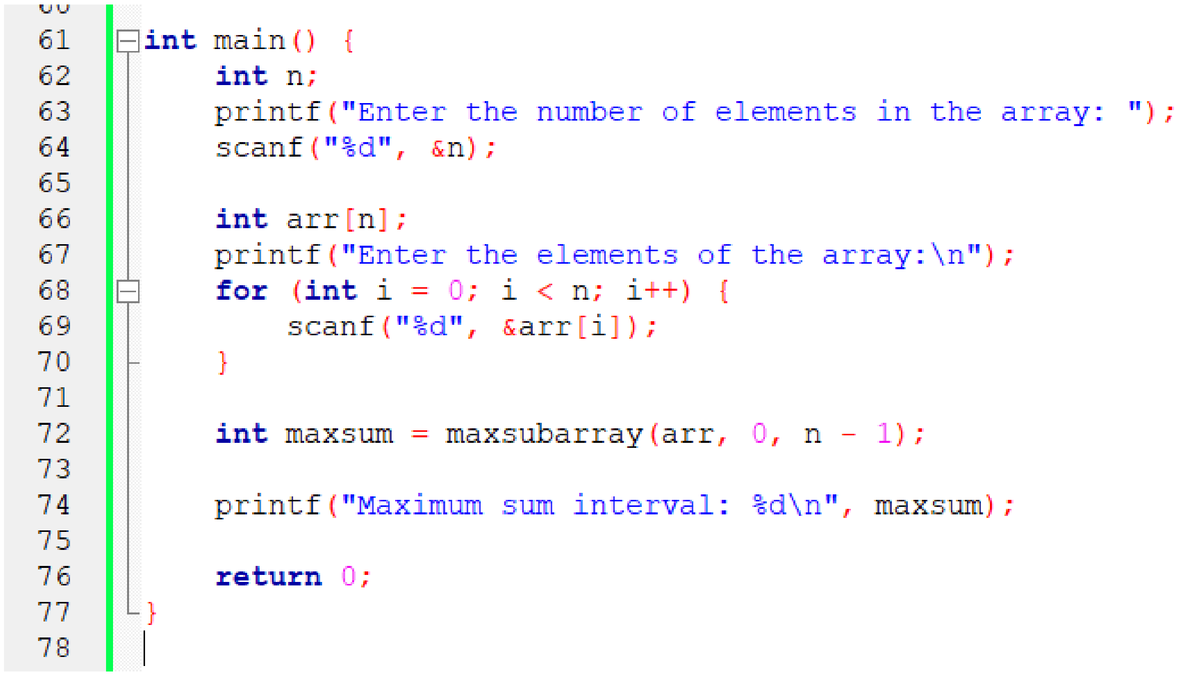
3. **‘maxsubarray’ Function:**



```
int maxsubarray(int ar[], int low, int high)
{
  if (high == low)
  {
    return ar[high];
  }

  int mid = (high+low)/2;


  int leftsubarray = maxsubarray(ar, low, mid);
  int rightsubarray = maxsubarray(ar, mid+1, high);
  int maxcrossrray1 = maxcrossrray(ar, low, mid, high);


  return maximum(leftsubarray,rightsubarray, maxcrossrray1);
}
```

This function is a recursive algorithm to find the maximum subarray sum. It divides the array into two halves and finds the maximum sum in the left and right subarrays. Then, it calculates the maximum sum that crosses the midpoint using the ‘**maxcrossrray’** function.

4. **‘main’ Function:**



```
int main() {
    int n;
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    int maxsum = maxsubarray(arr, 0, n - 1);

    printf("Maximum sum interval: %d\n", maxsum);

    return 0;
}
```

The main function handles user input for the array size and elements. It then calls the **‘maxsubarray’** function to find and print the maximum subarray sum.

These important parts collectively implement a divide-and-conquer algorithm to efficiently find the maximum subarray sum. The ‘**maximum**’, **‘maxcrossrray’**, and **‘maxsubarray’** functions work together to achieve this.

Time Complexity:

let's analyze the time complexity using the "method of solving recurrences" for the given divide-and-conquer algorithm. The recurrence relation for the time complexity of the **max_sum_subarray** function can be expressed as:

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

Here, n is the size of the array.

- $2T(n/2)$ represents the time taken for two recursive calls on subproblems of size $n/2$.

- $O(n)$ represents the time taken for the **max_crossing_subarray** function, which has a linear time complexity $O(n)$.

Now, we can use the Master Theorem to solve the recurrence relation. The Master Theorem has the form:

$T(n)=aT(n/2)+f(n)$

where a is the number of subproblems, b is the factor by which the problem size is reduced, and $f(n)$ is the cost of dividing the problem and combining the results.

In the provided code:

$f(n)=O(n)$ (cost of **max_crossing_subarray**).

Now, let's compare $f(n)$ with $n^{\log_b a}$

$n^{\log_b a}= n^{\log_2 2}=n$

Since $f(n)=O(n)$ and $f(n)$ is polynomially smaller than $n^{\log_b a}$ (Master theorem case 2)

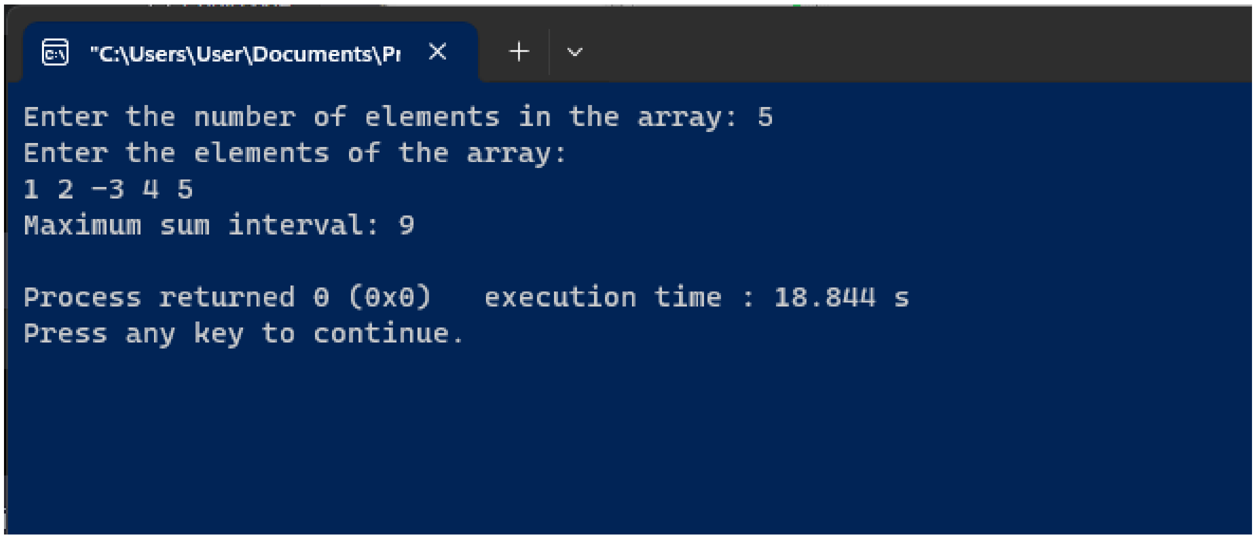
Case 2 states that if $\Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$, then the time complexity is $\Theta(n^{\log_b a} \log^{k+1} n)$

In our case, $k=0$ (no additional logarithmic factor), so the time complexity is $\Theta(n \log n)$.

Therefore, using the Master Theorem, we confirm that the time complexity of the provided code is $\Theta(n \log n)$.

<u>Testing Results:</u>

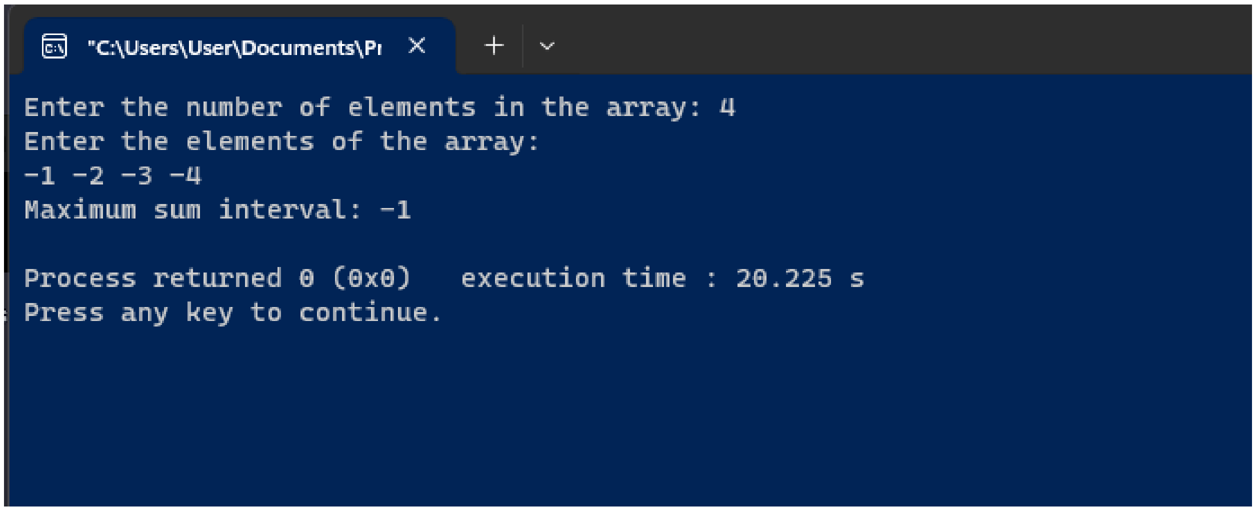
Test 1: Basic Input



```
"C:\Users\User\Documents\Pr

Enter the number of elements in the array: 5
Enter the elements of the array:
1 2 -3 4 5
Maximum sum interval: 9

Process returned 0 (0x0)   execution time : 18.844 s
Press any key to continue.
```

The maximum sum interval is achieved by adding elements 4 and 5.

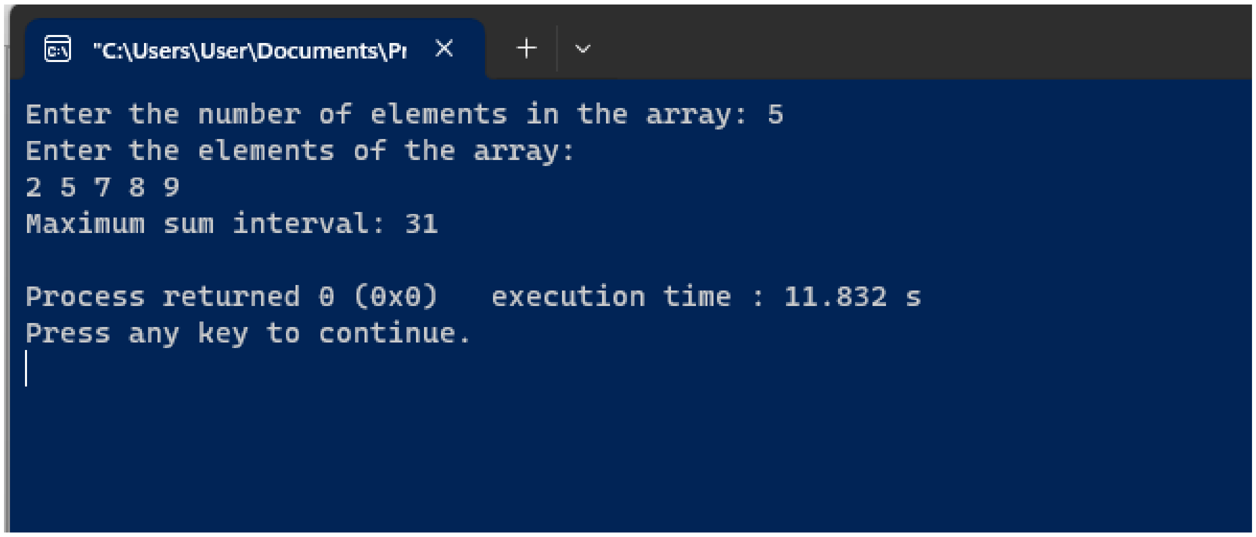
Test 2: All Non-Positive Numbers



```
"C:\Users\User\Documents\Pr

Enter the number of elements in the array: 4
Enter the elements of the array:
-1 -2 -3 -4
Maximum sum interval: -1

Process returned 0 (0x0)   execution time : 20.225 s
Press any key to continue.
```

Since all numbers are non-positive, the maximum sum interval is the number with the smallest magnitude, which is -1 in this case.

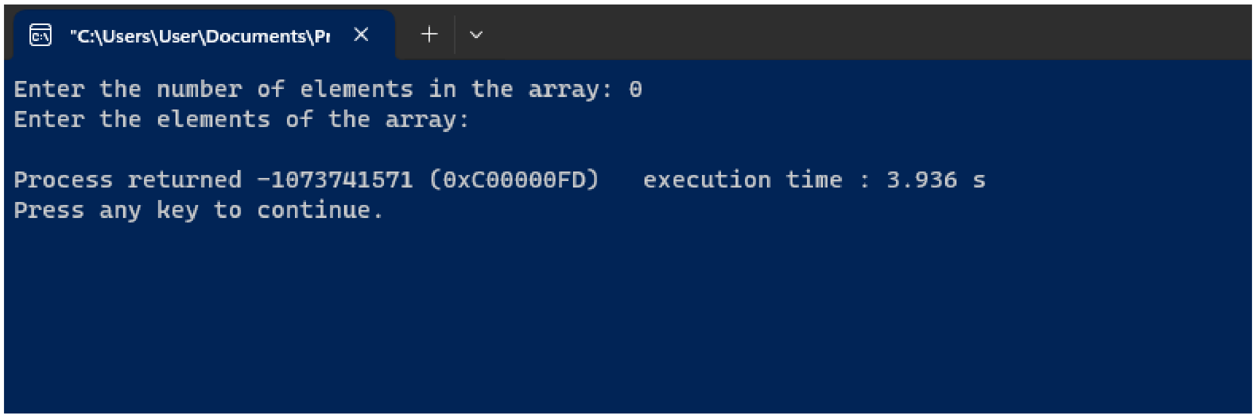
Test 3: All Non-Negative Numbers



```
Enter the number of elements in the array: 5
Enter the elements of the array:
2 5 7 8 9
Maximum sum interval: 31

Process returned 0 (0x0)   execution time : 11.832 s
Press any key to continue.
```

Since all numbers are non-negative, the maximum sum interval is the sum of all elements, which is 2 + 5 + 7+8+9 = 31.

Test 4: Empty Array



```
Enter the number of elements in the array: 0
Enter the elements of the array:

Process returned -1073741571 (0xC00000FD)   execution time : 3.936 s
Press any key to continue.
```

The program should handle the case of an empty array and prompt the user for a valid array size.

Future Scope:

The current implementation lacks robust input validation, memory usage optimization, and parallel processing. Future enhancements could include more thorough validation for non-numeric values or invalid arrays. The program should also consider parallel processing for improved performance and handle dynamic changes in array size during runtime.

Explore advanced algorithms, optimize runtime, improve error handling, develop standalone libraries, create visualization tools, and implement the algorithm in multiple programming languages for a wider audience and easier integration into various software projects.