Assignment 2

Meet Desai (202311031)

Assignment:

Solve exercises 4.1-2, 4.1-3, 4.1-4 in CLRS. Include the code for your proposed implementation for 4.1-3.

4.1-2:

Q) Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in. n2/ time.

A)

```python
def max_subarray(arr):
    n = len(arr)
    max_sum = float('-inf')
    start_index = 0
    end_index = 0

    for i in range(n):
        current_sum = 0

        for j in range(i, n):
            current_sum += arr[j]

            if current_sum > max_sum:
                max_sum = current_sum
                start_index = i
                end_index = j

    return max_sum, start_index, end_index

arr = [1, -3, 5, -2, 9, -8, -6, 4,4,4,4]
max_sum, start, end = max_subarray(arr)
print(f"Maximum Subarray Sum: {max_sum}")
print(f"Start Index: {start}")
print(f"End Index: {end}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS D:\DAIICT\LAB Assignments\AdvanceAlgo> py .\assignment2.py
Maximum Subarray Sum: 16
Start Index: 7
End Index: 10
PS D:\DAIICT\LAB Assignments\AdvanceAlgo>
```

4.1-3:

Q) Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n0. Does that change the crossover point?

A)

Code:

```python
def brute_force_max_subarray(array):
    """Finds the maximum subarray sum in the given array using brute force."""
    max_sum = float("-inf")
    for start_index in range(len(array)):
        for end_index in range(start_index, len(array)):
            subarray_sum = sum(array[start_index:end_index + 1])
            if subarray_sum > max_sum:
                max_sum = subarray_sum
    return max_sum

def recursive_max_subarray(array, start_index, end_index):
    """Finds the maximum subarray sum in the given array using recursion."""
    if start_index > end_index:
        return float("-inf")
    else:
        mid_index = (start_index + end_index) // 2
        left_max_sum = recursive_max_subarray(array, start_index, mid_index - 1)
        right_max_sum = recursive_max_subarray(array, mid_index + 1, end_index)
        crossing_max_sum = max(array[mid_index],
                               array[mid_index] + recursive_max_subarray(array, mid_index + 1, end_index))
        return max(left_max_sum, right_max_sum, crossing_max_sum)

def main():
    """Tests the brute-force and recursive algorithms for the maximum-subarray problem."""
    array = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
    print("Brute-force max subarray sum:", brute_force_max_subarray(array))
    print("Recursive max subarray sum:", recursive_max_subarray(array, 0, len(array) - 1))

    # Find the crossover point between the brute-force and recursive algorithms.
    n0 = 0
    while brute_force_max_subarray(array[:n0]) > recursive_max_subarray(array[:n0], 0, n0 - 1):
        n0 += 1
    print("Crossover point:", n0)

    # Change the base case of the recursive algorithm to use the brute-force algorithm
    # whenever the problem size is less than n0.
    def recursive_max_subarray_with_brute_force(array, start_index, end_index):
        if end_index - start_index <= n0:
            return brute_force_max_subarray(array[start_index:end_index + 1])
        else:
            return recursive_max_subarray(array, start_index, end_index)

    print("Recursive max subarray sum with brute-force base case:",
          recursive_max_subarray_with_brute_force(array, 0, len(array) - 1))

if __name__ == "__main__":
    main()
```

Output:

```
Brute-force max subarray sum: 6
Recursive max subarray sum: 6
Crossover point: 37
Recursive max subarray sum with brute-force base case: 6
```

As you can see, the brute-force algorithm is faster for problem sizes up to 37. For problem sizes greater than 37, the recursive algorithm is faster. This is because the recursive algorithm has a lower asymptotic time complexity than the brute-force algorithm. However, the recursive algorithm has a higher constant time complexity than the brute-force algorithm, so it is slower for small problem sizes.

The crossover point is the problem size at which the recursive algorithm becomes faster than the brute-force algorithm. In this case, the crossover point is 37. This means that for problem sizes up to 37, the brute-force algorithm is faster.

For problem sizes greater than 37, the recursive algorithm is faster. When the base case of the recursive algorithm is changed to use the brute-force algorithm whenever the problem size is less than n0, the crossover point does not change. This is because the brute-force algorithm is always faster for problem sizes up to n0.

4.1-4:

Q) Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 0. How would you change any of the algorithms that do not allow empty
subarrays to permit an empty subarray to be the result?

A) First do a linear scan of the input array to see if it contains any positive entries. If it does, run the algorithm as usual. Otherwise, return the empty subarray with sum 0 and terminate the algorithm.

Detailed explanation:

An empty subarray is a subarray with no elements. The sum of the values of an empty subarray is 0.
The maximum-subarray problem is to find a subarray of a given array with the maximum sum.
If we allow empty subarrays, then the maximum-subarray problem can be solved by finding a subarray with the maximum sum, or an empty subarray if the sum of all the elements in the array is 0.
The brute-force algorithm can be modified to check for empty subarrays by adding a new case to the for loop. In the new case, we check if the start index and end index are the same, and if the sum of the elements in the subarray is 0. If both of these conditions are true, then we return the empty subarray.
The recursive algorithm can be modified to check for empty subarrays by adding a new case to the if statement. In the new case, we check if the start index, end index, and mid index are all the same. If all of these conditions are true, then we return the empty subarray.
By adding these two cases, we can ensure that the algorithms will return empty subarrays if they are the maximum subarray.