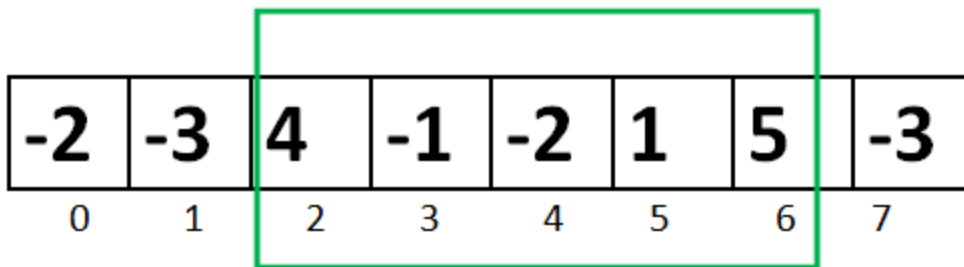


Finding Maximum Sum Subarray



Design and Analysis of Algorithms (CS 311)

First Semester, term 231

Sec# 1469

Prepared By:

Nour Al Akhras – 220410351

Haifa Zain Eddin – 220410581

Supervised By:

Dr. Helene Kanso

Table of Contents:

| | |
|---|-----------|
| 1. Introduction..... | 3 |
| 2. Problem Statement..... | 3 |
| 3. Real-World Application:..... | 3 |
| 3. Proposed Algorithms..... | 5 |
| Brute Force:..... | 5 |
| Divide-and-Conquer:..... | 6 |
| Dynamic Programming:..... | 8 |
| Classical Dynamic Programming:..... | 8 |
| Kadane's Algorithm..... | 9 |
| 4. System Design..... | 9 |
| 5. User Interface Design..... | 10 |
| Maximum Sum Subarray Interface (index.html):..... | 10 |
| Exploring with Random Arrays Interface (generate_random.html):..... | 10 |
| 6. Implementation..... | 11 |
| 1. Brute Force..... | 11 |
| 2. Divide and Conquer..... | 12 |
| 3. Classical Dynamic Programming..... | 13 |
| 4. Kadane's Dynamic Programming..... | 14 |
| 5. Calculating the time elapsed for each algorithm..... | 14 |
| 7. Experimental Results..... | 16 |
| 1. Input an array of 500 integers..... | 16 |
| 1.1. Brute Force..... | 16 |
| 1.2. Divide and Conquer..... | 17 |
| 1.2. Classical Dynamic Programming..... | 18 |
| 1.4 Kadane's Dynamic Programming..... | 19 |
| 2. Generating an array of size 10000..... | 20 |
| 2.1 Brute Force..... | 20 |
| 2.2 Divide and Conquer..... | 20 |
| 2.3 Classical Dynamic Programming..... | 20 |
| 2.4 Kadane's Dynamic Programming..... | 20 |
| 2. Generating an array of size 50000..... | 20 |
| 2.1 Brute Force..... | 20 |
| 2.2 Divide and Conquer..... | 20 |
| 2.3 Classical Dynamic Programming..... | 20 |
| 2.4 Kadane's Dynamic Programming..... | 20 |
| 3. Generating an array of size 100000..... | 20 |
| 2.1 Brute Force..... | 20 |
| 2.2 Divide and Conquer..... | 21 |
| 2.3 Classical Dynamic Programming..... | 21 |

| | |
|---------------------------------------|-----------|
| 2.4 Kadane's Dynamic Programming..... | 21 |
| 8. Similar Problem..... | 21 |
| 8. References..... | 21 |

1. Introduction

In the realm of algorithmic problem-solving, the quest for determining the maximum sum subarray is a fundamental and intriguing challenge. This problem entails identifying the contiguous subarray within a given array that yields the highest sum. This concept finds application in various domains, such as finance, data analysis, and signal processing, where identifying the most significant subarray sum can yield valuable insights.

2. Problem Statement

Given an array of integers, the task is to find the contiguous subarray with the largest sum.

Formally, if the input is an array `nums` of length `n`, the objective is to find a contiguous subarray, denoted by indices `i` and `j` (where $0 \leq i \leq j < n$), such that the sum of the elements in the subarray `nums[i:j+1]` is maximized.

Example 1:

Input: `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]`

Output: 6

Explanation: The subarray `[4, -1, 2, 1]` has the maximum sum of 6.

Example 2:

Input: `nums = [1, -2, 3, -4, 5, -6, 7, -8, 9]`

Output: 9

Explanation: The subarray `[9]` has the maximum sum of 9.

3. Real-World Application:

Brief Explanation:

One of the Real - World Applications is finding the maximum profit possible by buying or selling stocks on different days. In this application a dynamic approach of solving this problem is taken to maintain efficiency hence it's a real-life problem.

Statement:

Given an array of daily stocks prices, there has to be a maximum profit that can be achieved by buying or selling the stock within a single transaction where you can only buy once and sell once.

Example: Daily Stock Prices: [-2,1,-3,4]

Max_sum = -2 which is the first element

Current_sun = -2

➤ For the first iteration:

The current element is: 1

Potential current sum would be $\max(1, -2+1) = 1$

Current_Sum will be updated to 1 and the Max_sum will be updated to 1.

➤ For the second iteration:

The Current_element will become -3.

Potential current sum would be $\max(-3, 1-3) = -2$

Current_Sum will be updated to -2 and the Max_sum remains 1.

➤ For the third iteration:

The Current_element will become 4.

Potential current sum would be $\max(4, -2+4) = 4$

Current_Sum will be updated to 4 and the Max_sum will be updated to 4.

➤ For the Fourth iteration:

The execution will stop and return the maximum sum of the subarray which is 4 and the subarray which is [1,-3, 4].

3. Proposed Algorithms

Brute Force:

The brute force algorithm is the basic algorithm used to find the maximum sum of a subarray in a given array using nested loops.

Explanation of the algorithm:

1. The algorithm will start with checking the base case which is the case of having no elements and it will return 0.
2. Otherwise it will iterate through the input array and check all possible subarrays using nested loops.
3. The outer loop will determine the starting index of a subarray whilst the inner loop will iterate from starting index till the end of the array.
4. The current_sum will be updated cumulatively, if it surpasses a previously calculated max_sum then the max_sum will be updated accordingly.
5. Once the loops are completed, the algorithm will return the max_sum.

Pseudocode:

MaxSum_bruteforce(Arr):

```
    if Arr is empty:
        return 0

    n = len(Arr)
    max_Sum = - infinity
    current_sum=0

    from i till n-1:
        from j till n-1:
            current_sum +=Arr[j]
            if current_sum > max_sum:
                max_sum = current_sum
    return max_sum
```

Time Complexity:

The running time of the brute force algorithm is $O(n^2)$ as it can be seen that nested loops are required for this algorithm where the outer loop will run linear time n and the inner loop will run at most n for each iteration of the outer one.

Space Complexity:

The space complexity for this algorithm is $O(1)$ which is constant time since no additional space is used in this algorithm and the variables used don't depend on the size of the input array.

Divide-and-Conquer:

The divide-and-conquer approach can be used to find the maximum sum of a subarray in a given array by finding the mid-point that divides the array into two halves, computing the maximum sum of each half and of the subarray that crosses the mid-point recursively and finally merging the results and returning the maximum sum of the subarrays.

Explanation of the algorithm:

`max_subarray_sum_divide_conquer(arr, low, high):`

1. Base case: if the array consists of one element then return that element as the maximum sum.
2. Divide: the array is divided into two halves (left / right) by finding the mid-point (mid), then the algorithm will call itself recursively to find the maximum sum of a subarray on each side.
2. Conquer: the maximum sum of the entire array is going to be either the maximum sum of the subarray on the left half, or on the right half, or the sum of the subarray that crosses the mid-point.

`max_crossing_sum(arr, low, mid, high):`

1. This method will calculate the maximum sum of a subarray that crosses the mid-point by iterating linearly from the mid-point till the start/end point of the array.
2. Then, it will find the maximum sum on both sides left and right.
3. The results will be combined to find the maximum sum of the subarray that crosses the mid-point.
4. It will return the subarray that crosses the mid-point and the sum of left and right subarrays.
5. The method will call itself recursively to divide this problem into smaller subproblems and combine their solutions.

Pseudocode:

`max_crossing_sum(arr, low, mid, high):`

```
    left_sum = - infinity
    sum_left = 0
    max_left_idx = mid
    for i from mid to low - 1:
        sum_left += arr[i]
        if sum_left > left_sum:
            left_sum = sum_left
            max_left_idx = i
```

```
    right_sum = - infinity
    sum_right = 0
    max_right_idx = mid + 1
```

```

for i from mid+1 till high:
    sum_right += arr[i]
    if sum_right > right_sum:
        right_sum = sum_right
        max_right_idx = i
return arr[max_left_idx : max_right_idx +1], left_sum + right_sum

```

```

max_subarray_sum_divide_conquer(arr, low, high):
    if low = high:
        return arr[low]
    mid = floor((low+high)/2)
    left_sum, = max_subarray_sum_divide_conquer(arr, low, mid)
    right_sum = max_subarray_sum_divide_conquer(arr, mid+1,high)
    crossing_sum = max_crossing_sum(arr, low, mid, high)
    if left_sum >= right_sum and left_sum >= crossing_sum:
        return left_sum
    elif right_sum >= left_sum and right_sum >= crossing_sum:
        return right_sum
    else:
        return crossing_sum

```

Time Complexity:

The time complexity for this algorithm is $O(n \log n)$ by Master's Theorem where the recursive function is of $T(n) = 2 * T(n/2) + O(n)$. Since it follows the form $T(n) = a * T(n/b) + f(n)$ where:

➤ $a = 2$ which is ≥ 1 , $b = 2$ which is > 1 , and $f(n) = O(n) = O(n^1)$.

1. Compare $f(n)$ with $n^{\log_b a}$.

➤ $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, so the Master's theorem can be applied.

2. By checking case two of Master's Theorem which is :

➤ If $f(n) = \Theta(n \log_b a)$, then $T(n) = \Theta(n \log_b a * \log n)$, Then the solution will be $\Theta(n \log n)$.

Space Complexity:

The space complexity for this algorithm is $O(\log n)$ due to the depth of recursive call $\log n$ where n is the size of the array plus constant time $O(1)$ for the variables.

Dynamic Programming:

Classical Dynamic Programming:

The classical dynamic programming approach for finding the maximum subarray sum involves maintaining an auxiliary array `maxSumEnding`, where each element represents the maximum subarray sum ending at that position. The final result is obtained by finding the maximum value in this array.

Explanation of the algorithm:

1. Initialize an array `maxSumEnding` of size n to store the maximum subarray sum ending at each position, with the first element set to the corresponding value in the input array X .
2. Iterate through the array from index 1 to $n-1$.
3. At each step, update `maxSumEnding[i]` based on whether adding the current element to the previous subarray sum is beneficial. If the previous sum is negative, start a new subarray.
4. Find the maximum value in the `maxSumEnding` array to get the overall maximum subarray sum.
5. Return the maximum subarray sum.

Pseudocode:

```
function classical_Dynamic_programming(arr):  
    n = length of X  
    maxSumEnding = array of size n  
    maxSumEnding[0] = X[0]  
  
    for i from 1 to n - 1:  
        if maxSumEnding[i - 1] > 0:  
            maxSumEnding[i] = X[i] + maxSumEnding[i - 1]  
        else:  
            maxSumEnding[i] = X[i]  
  
    maxSubarraySum = -infinity  
    for i from 0 to n - 1:  
        maxSubarraySum = max(maxSubarraySum, maxSumEnding[i])  
  
    return maxSubarraySum
```

Time Complexity:

The time complexity of this dynamic programming approach is $O(n)$, where n is the length of the input array. This is because we iterate through the array once, performing constant time operations at each step.

Space Complexity:

The space complexity is $O(n)$ because we use an additional array `dp` of size n to store the intermediate results.

Kadane's Algorithm

Kadane's Algorithm is a more optimized version of finding the maximum subarray sum, using only constant space. It iterates through the array, maintaining two variables (maxsumSoFar and maxsumEndingHere) to keep track of the maximum subarray sum.

Explanation of the algorithm:

1. Initialize maxsumSoFar and maxsumEndingHere to the first element of the array.
2. Iterate through the array, updating maxsumEndingHere based on whether adding the current element to the previous subarray sum is beneficial. If the previous sum is negative, start a new subarray.
3. Update maxsumSoFar if maxsumEndingHere becomes greater than the current maximum.

Pseudocode:

```
function kadane_Dynamic_programming(X):  
    n = length of X  
    maxsumSoFar = X[0]  
    maxsumEndingHere = X[0]  
  
    for i from 1 to n - 1:  
        maxsumEndingHere = max(maxsumEndingHere + X[i], X[i])  
        if maxsumSoFar < maxsumEndingHere:  
            maxsumSoFar = maxsumEndingHere  
  
    return maxsumSoFar
```

Time Complexity:

The time complexity of Kadane's Algorithm is $O(n)$ because we iterate through the array once.

Space Complexity:

The space complexity is $O(1)$ because we only use a constant amount of extra space (max_sum and current_sum) regardless of the size of the input array.

4. System Design

The system is designed to prompt the user to input an array. It then applies various algorithms to find the maximum sum subarray. Main data structures include arrays for input and output.

5. User Interface Design

Maximum Sum Subarray Interface (index.html):

- Allows users to input an array of numbers and select an algorithm.
- Finds the maximum sum subarray using the selected algorithm.
- Displays the maximum sum, subarray, and elapsed time.

Maximum Sum Subarray

Generate random arrays

Enter an array of numbers (comma-separated):

Select the algorithm:

Brute Force Algorithm

Submit

Exploring with Random Arrays Interface (generate_random.html):

- Allows users to input the size of an array and select an algorithm.
- Generates a random array based on user input.
- Displays the generated random array, maximum sum, subarray, and elapsed time

Exploring with Random Arrays

Give your own array

Enter the size of the array (n):

Select an algorithm:

Brute Force

Submit

6. Implementation

<https://github.com/NourAlakhras/algorithms.git>

1. Brute Force

```
1
2 def brute_force(arr):
3     n = len(arr)
4     max_sum = float('-inf')
5     start, end = None, None # Initialize start and end to None
6     #For the nested loop it will be a time complexity of  $O(n^2)$ , and space complexity of  $O(1)$ 
7     for i in range(n): # Outer loop that runs n times linearly from index 0 till n-1
8         current_sum = 0
9         for j in range(i, n): #Inner loop will run at most n times starting from j = i till n-1
10            current_sum += arr[j]
11            if current_sum > max_sum: #Ensuring that the sum is the maximum and updating accordingly
12                max_sum = current_sum
13                start = i
14                end = j
15
16 #then it should return the new indecies that have the sub-array with maximum sum and the its max-sum.
17 return max_sum, arr[start:end+1]
18
```


2. Divide and Conquer

```
1 def divide_and_conquer(arr, low, high):
2     if low == high:
3         return arr[low], arr[low]
4
5     mid = (low + high) // 2
6
7     # Recursively find the maximum subarray sum in the left and right halves O(n/2)
8     left_sum, left_subarray = divide_and_conquer(arr, low, mid)
9     right_sum, right_subarray = divide_and_conquer(arr, mid + 1, high)
10
11    # Find the maximum subarray sum that crosses the midpoint (Merging O(n))
12    crossing_sum, crossing_subarray = max_crossing_sum(arr, low, mid, high)
13
14    # Return the result with the maximum sum
15    if left_sum >= right_sum and left_sum >= crossing_sum:
16        return left_sum, left_subarray
17    elif right_sum >= left_sum and right_sum >= crossing_sum:
18        return right_sum, right_subarray
19    else:
20        return crossing_sum, crossing_subarray
21
22
23 def max_crossing_sum(arr, low, mid, high):
24     left_sum = float('-inf')
25     sum_left = 0
26     max_left_idx = mid
27     #The total time complexity for the loops below is still considered as linear O(n)
28     for i in range(mid, low - 1, -1): #This loop will run linearly, it will iterate till half of the main array therefore n/2.
29         sum_left += arr[i]
30         if sum_left > left_sum:
31             left_sum = sum_left
32             max_left_idx = i #Max on the left side
33
34     right_sum = float('-inf')
35     sum_right = 0
36     max_right_idx = mid + 1
37
38     for i in range(mid + 1, high + 1): #Similarly this loop will run linearly, it will iterate till half of the main array therefore n/2.
39         sum_right += arr[i]
40         if sum_right > right_sum:
41             right_sum = sum_right
42             max_right_idx = i #Max on the right side
43
44     return left_sum + right_sum, arr[max_left_idx:max_right_idx + 1]
45
```

3. Classical Dynamic Programming

```
1 def classical_Dynamic_programming(arr):
2     n = len(arr)
3     maxSumEnding = [0] * n
4     maxSumEnding[0] = arr[0]
5
6     start, end = 0, 0
7
8     for i in range(1, n):
9         if maxSumEnding[i - 1] > 0:
10             maxSumEnding[i] = arr[i] + maxSumEnding[i - 1]
11         else:
12             maxSumEnding[i] = arr[i]
13             start = i # Start a new subarray
14
15         # Update end index if the current subarray sum is the maximum
16         if maxSumEnding[i] > maxSumEnding[end]:
17             end = i
18
19     maxSubarraySum = max(maxSumEnding)
20     subarray = arr[start:end + 1]
21     return maxSubarraySum, subarray
22
```

4. Kadane's Dynamic Programming



```
1 def kadane_dynamic_programming(arr):
2     n = len(arr)
3     maxsumSoFar = arr[0]
4     maxsumEndingHere = arr[0]
5     start, end, tempStart = 0, 0, 0
6
7     for i in range(1, n):
8         if maxsumEndingHere + arr[i] < arr[i]:
9             maxsumEndingHere = arr[i]
10            tempStart = i
11        else:
12            maxsumEndingHere = maxsumEndingHere + arr[i]
13
14        if maxsumSoFar < maxsumEndingHere:
15            maxsumSoFar = maxsumEndingHere
16            start = tempStart
17            end = i
18
19    subarray = arr[start:end + 1]
20    return maxsumSoFar, subarray
```

5. Calculating the time elapsed for each algorithm

The `time_it` function is a utility function designed to measure the execution time of a given function (`func`) with specified arguments (`*args`).



```
1
2 def time_it(func, *args):
3     start_time = time.time()
4     result = func(*args)
5     end_time = time.time()
6     elapsed_time = (end_time - start_time) * 1000 # Convert to milliseconds
7     return result, elapsed_time
```



```
1
2 match request.form['algorithm']:
3     case 'bruteforce':
4         result, elapsed_time = time_it(brute_force, arr)
5     case 'divideandconquer':
6         result, elapsed_time = time_it(divide_and_conquer, arr, 0, len(arr) - 1)
7     case 'dynamicprogramming':
8         result, elapsed_time = time_it(classical_dynamic_programming, arr)
9     case 'kaden':
10        result, elapsed_time = time_it(kadane_dynamic_programming, arr)
```


1.2. Divide and Conquer

Maximum Sum Subarray

Generate random arrays

Enter an array of numbers (comma-separated):

-96, -69, -76, -97, -9, -49, -53, -38, 12, 11, -66, 70, 91, 51, 27, 39, -40, -37, -56, 88, -18, 49, -20, 72, 65, 30, -37, 17, 46, 9, -3, -3, -47, -73, -6, 78, 50, 7, 18, 74, -7, -1, -93, -26, -50, -3, 79, -23, 92, -12, -6, 16, -1, 39, 94, 27, -89, -74, -72, 27, 84, 22, 16, 43, 52, 18, -97, 0, -10, 93, -88, -9, 28, 75, 89, -48, 5, -20, 35, -45, -1, -61, -79, -17, 86, 36, -69, 30, 86, -77, -91,

Select the algorithm:

Divide and Conquer Algorithm

Submit

Maximum Sum:

1846

Subarray:

97, 80, 36, 11, 54, 76, 43, -42, 85, 20, -80, -7, -65, -82, 73, -19, -11, -29, 49, 47, -43, -68, 79, 60, 23, 43, 96, -51, 99, 23, -27, -36, -1, -54, -40, 75, 48, 100, 85, -11, 53, -76, -16, -1, 54, 12, -43, 73, -43, 17, 18, 0, -19, 52, 100, 65, 2, -83, 46, 62, 37, -85, 54, -23, -82, 55, -12, 85, 62, 91, -35, 49, 9, -42, 99, 40, 78, -92, -33, -7, -79, 45, -95, -52, 74, 47, 17, 73, 36, -9, -31, -32, -94, -6, 44, -6, -37, 10, -90, 3, -9, 3, -81, -47, 19, 35, -68, 50, -97, -42, 82, 64, 40, 50, 89, 84, 65, -89, -82, 9, 17, -47, 11, 0, 86, 47, -74, -12, 90, -94, -71, -27, -4, -76, -68, -17, 54, -20, 60, 90, -99, -53, -5, -44, 50, -52, 20, -99, 92, 22, 42, 95, -1, -13, 82, 10, -39, 42, 87, 59, 17, 43, -2, -98, -29, -40, -94, -47, -93, -50, -31, -59, -31, 38, 65, 69, -58, 13, -83, -69, -100, 31, 73, -70, -10, -100, 7, -26, -55, -88, 77, -16, 76, -73, -69, 70, -22, 93, -42, 74, 15, 65, 23, 76, -83, 55, -68, -60, 15, -39, 51, -46, -97, 95, 25, -79, -50, 53, 25, 59, 18, -79, 21, 68, 44, -8, 52, 4, 70, 68, -9, -15, -17, -60, 3, -33, 93, 40, 88, -20, 53, 78, 42, 75, -25, -66, -10, 87, 3, 16, 57, 18, 26, -98, 41, -11, -25, -27, 77, 53, 100, -66, 30, -54, -65, 98, -56, -38, 93, 26, 71, 85, 65, 83, -90, 34, 83, -9, 98, 49

Elapsed Time:

[illegible]

Generate random arrays

-31, 38, 65, 69, -58, 13, -83, -69, 100, 31, 73, -70, -10, -100, 7, -26, -55, -88, 77, -16, 76, -73, -69, 70, -22, 93, -42, 74, 15, 65, 23, 76, -83, 55, -68, -60, 15, -39, 51, -46, -97, 95, 25, -79, -50, 53, 25, 59, 18, -79, 21, 68, 44, -8, 52, 4, 70, 68, -9, -15, -17, -60, 3, -33, 93, 40, 88, -20, 53, 78, 42, 75, -25, -66, -10, 87, 3, 16, 57, 18, 26, -98, 41, -11, -25, -27, 77, 53, 100, -66, 30, -54, -65, 98, -56, -38, 93, 26, 71, 85, 65, 83, -90, 34, 83, -9, 98, 49, -1, -33, -15, 37

Classical Dynamic Programming Algorithm

Submit

1846

97, 80, 36, 11, 54, 76, 43, -42, 85, 20, -80, -7, -65, -82, 73, -19, -11, -29, 49, 47, -43, -68, 79, 60, 23, 43, 96, -51, 99, 23, -27, -36, -1, -54, -40, 75, 48, 100, 85, -11, 53, -76, -16, -1, 54, 12, -43, 73, -43, 17, 18, 0, -19, 52, 100, 65, 2, -83, 46, 62, 37, -85, 54, -23, -82, 55, -12, 85, 62, 91, -35, 49, 9, -42, 99, 40, 78, -92, -33, -7, -79, 45, -95, -52, 74, 47, 17, 73, 36, -9, -31, -32, -94, -6, 44, -6, -37, 10, -90, 3, -9, 3, -81, -47, 19, 35, -68, 50, -97, -42, 82, 64, 40, 50, 89, 84, 65, -89, -82, 9, 17, -47, 11, 0, 86, 47, -74, -12, 90, -94, -71, -27, -4, -76, -68, -17, 54, -20, 60, 90, -99, -53, -5, -44, 50, -52, 20, -99, 92, 22, 42, 95, -1, -13, 82, 10, -39, 42, 87, 59, 17, 43, -2, -98, -29, -40, -94, -47, -93, -50, -31, -59, -31, 38, 65, 69, -58, 13, -83, -69, -100, 31, 73, -70, -10, -100, 7, -26, -55, -88, 77, -16, 76, -73, -69, 70, -22, 93, -42, 74, 15, 65, 23, 76, -83, 55, -68, -60, 15, -39, 51, -46, -97, 95, 25, -79, -50, 53, 25, 59, 18, -79, 21, 68, 44, -8, 52, 4, 70, 68, -9, -15, -17, -60, 3, -33, 93, 40, 88, -20, 53, 78, 42, 75, -25, -66, -10, 87, 3, 16, 57, 18, 26, -98, 41, -11, -25, -27, 77, 53, 100, -66, 30, -54, -65, 98, -56, -38, 93, 26, 71, 85, 65, 83, -90, 34, 83, -9, 98, 49

[illegible]

1.4 Kadane's Dynamic Programming

Maximum Sum Subarray

Generate random arrays

Enter an array of numbers (comma-separated):

65, 23, 76, -83, 55, -68, -60, 13, -39, 51, -46, -97, 95, 25, -79, -50, 53, 25, 59, 18, -79, 21, 68, 44, -8, 52, 4, 70, 68, -9, -15, -17, -60, 3, -33, 93, 40, 88, -20, 53, 78, 42, 75, -25, -66, -10, 87, 3, 16, 57, 18, 26, -98, 41, -11, -25, -27, 77, 53, 100, -66, 30, -54, -65, 98, -56, -38, 93, 26, 71, 85, 65, 83, -90, 34, 83, -9, 98, 49, -1, -33, -15, 37

Select the algorithm:

Kadane's Dynamic Programming Algorithm

Submit

Maximum Sum:

1846

Subarray:

97, 80, 36, 11, 54, 76, 43, -42, 85, 20, -80, -7, -65, -82, 73, -19, -11, -29, 49, 47, -43, -68, 79, 60, 23, 43, 96, -51, 99, 23, -27, -36, -1, -54, -40, 75, 48, 100, 85, -11, 53, -76, -16, -1, 54, 12, -43, 73, -43, 17, 18, 0, -19, 52, 100, 65, 2, -83, 46, 62, 37, -85, 54, -23, -82, 55, -12, 85, 62, 91, -35, 49, 9, -42, 99, 40, 78, -92, -33, -7, -79, 45, -95, -52, 74, 47, 17, 73, 36, -9, -31, -32, -94, -6, 44, -6, -37, 10, -90, 3, -9, 3, -81, -47, 19, 35, -68, 50, -97, -42, 82, 64, 40, 50, 89, 84, 65, -89, -82, 9, 17, -47, 11, 0, 86, 47, -74, -12, 90, -94, -71, -27, -4, -76, -68, -17, 54, -20, 60, 90, -99, -53, -5, -44, 50, -52, 20, -99, 92, 22, 42, 95, -1, -13, 82, 10, -39, 42, 87, 59, 17, 43, -2, -98, -29, -40, -94, -47, -93, -50, -31, -59, -31, 38, 65, 69, -58, 13, -83, -69, -100, 31, 73, -70, -10, -100, 7, -26, -55, -88, 77, -16, 76, -73, -69, 70, -22, 93, -42, 74, 15, 65, 23, 76, -83, 55, -68, -60, 15, -39, 51, -46, -97, 95, 25, -79, -50, 53, 25, 59, 18, -79, 21, 68, 44, -8, 52, 4, 70, 68, -9, -15, -17, -60, 3, -33, 93, 40, 88, -20, 53, 78, 42, 75, -25, -66, -10, 87, 3, 16, 57, 18, 26, -98, 41, -11, -25, -27, 77, 53, 100, -66, 30, -54, -65, 98, -56, -38, 93, 26, 71, 85, 65, 83, -90, 34, 83, -9, 98, 49

Elapsed Time:

0.000000000 milliseconds

2. Generating an array of size 10000

2.1 Brute Force

Elapsed Time:

[illegible]

2.2 Divide and Conquer

Elapsed Time:

63.78412246704101562500 milliseconds

2.3 Classical Dynamic Programming

Elapsed Time:

```
1.9998550415039062500000000000000000000000000000000000000000000 milliseconds
```

2.4 Kadane's Dynamic Programming

Elapsed Time:

[illegible]

2. Generating an array of size 50000

2.1 Brute Force

Elapsed Time:

[illegible]

2.2 Divide and Conquer

Elapsed Time:

[illegible]

2.3 Classical Dynamic Programming

Elapsed Time:

```
30.570268630981445312500000000000000000000000000000000000000000000000 milliseconds
```

2.4 Kadane's Dynamic Programming

Elapsed Time:

[illegible]

3. Generating an array of size 100000

2.1 Brute Force

It took more than 15 minutes

2.2 Divide and Conquer

Elapsed Time:

[illegible]

2.3 Classical Dynamic Programming

Elapsed Time:

[illegible]

2.4 Kadane's Dynamic Programming

Elapsed Time:

47.7674007415771484375000 milliseconds

8. Similar Problem

A similar problem is that given an array of positive integers n, return the maximum possible sum of an ascending subarray in n, where the subarray is a contiguous sequence of integer in an array.

- Say that a subarray $[\text{nums}[l], \text{nums}[l+1], \dots, \text{nums}[r-1], \text{nums}[r]]$ is ascending if for all i where $l \leq i < r$, $\text{nums}[i] < \text{nums}[i+1]$. And a subarray of size 1 is ascending..

This type of problem can be solved using the dynamic programming approach which in terms of complexity it would also be of time complexity of $O(n)$ which is the complexity of our problem.

8. References

- Rojek, T. . (2017). MAXIMUM SUBARRAY PROBLEM OPTIMIZATION FOR SPECIFIC DATA. *Informatyka, Automatyka, Pomiary W Gospodarce I Ochronie Środowiska*, 7(4), 62–65.
<https://doi.org/10.5604/01.3001.0010.7507>
- Malik, H., & Daescu, O. (2018). k-Maximum Subarrays for Small k: Divide-and-Conquer made simpler. *arXiv preprint arXiv:1804.05956*. <https://arxiv.org/abs/1804.05956>
- Kirubaharan, L. (n.d.). Understanding Kadens’s Algorithm: Solving Real-Life Problems. Medium.
<https://medium.com/@laaveniyakirubaharan/understanding-kadaness-algorithm-solving-real-life-problems>
- Wade, N. (n.d.). Maximum Ascending Subarray. Medium.
<https://medium.com/codex/leetcode-1800-maximum-ascending-subarray-python-solution>