Report

Contents

1. Team Members

Team members names :

1. Maheswari Karlapudi
2. Ananya Kolla

Contribution of Maheswari Karlapudi :

- Designed ALGORITHM1
- Designed ALGORITHM2
- Designed ALGORITHM3
- Written Code implementation for TASK3B
- Written Code implementation for TASK4
- Written Code implementation for TASK5
- Written Code implementation for TASK6

Contribution of Ananya Kolla :

- Designed ALGORITHM4
- Designed ALGORITHM5
- Designed ALGORITHM6
- Written Code implementation for TASK1
- Written Code implementation for TASK2
- Written Code implementation for TASK3A
- Written Code implementation for TASK3B

Report,plots and makefile was done by both of us.

2. Design and Analysis of Algorithms

- **ALGORITHM1**

  We perform brute force by taking every possible subset and finding the sum of those subsets.
  Initially we loop from i to j (i being start of sub array and j being the end of sub array).
  Then we sum over elements from arr[i] to arr[j].
  If the sum>maxsum-seentillnow
  We set res[2]=sum,
  res[0]=i(left_index),
  res[1]=j(right_index)
  return result.
  The time complexity of this algorithm is O(n^3)
  The Space complexity of this algorithm is O(1)

- **Algorithm 2**

  We can observe that we need not calculate the sum each time from i to j.
  So, instead of initializing the sum to 0 each time, we just increment it.
  result contains start_index, end_index, max_sum.
  Set the start_index and end_index to -1
  Set the maximum sum seen till now to – INFINITY
  Initialize the curr_sum to 0.
  for i=1 to i<n
  set curr_sum=0;
  for j=i to j<n
  set curr_sum+=arr[j];
  if the curr_sum > result[2]
  we set result[2] = curr_sum;
  result[0] = i+1;
  result[1] = j+1;
  return result;
  The time complexity of this algorithm is O(n^2).
  The space complexity of this algorithm is O(1).

- **ALGORITHM3**

We will use dynamic programming to find the maximum sum of sub array. To prove we will use the four step process according to CLRS:

1. Characterize the structure of an optimal solution:

Let A be the given array. In this step, we need to figure out the structure of maximum sum of sub array. Let's define the structure of the problem as maxSubArray gives the max sum in the sub array from A[0:i] which must has A[i] as the end element. This implies that the element at index i is a part of the optimal solution. We store this result in dp[i]

The final result would be maximum{dp[i]} where i belongs to [0,len(A)]

We can build the optimal solution to MaxSubArray problem by finding if we can find the maximum sum by including index i or not.

2. Recursively define the value of optimal solution

The recursive formula can be given as,

maxSubArray(A, i) = A[i] + maxSubArray(A, i - 1) if maxSubArray(A, i - 1)>0

= A[i] otherwise

If the sum obtained from previous index is negative, adding it to current index will make it only much less. So, if the max sum obtained in previous index is less than zero we skip it.

3. Compute the value of an optimal solution

We can observe that this generates an overlapping subproblems. So, we use dynamic programming to store the value of maxSubArray(A,i) as dp[i]. And, compute the values of dp[0] to dp[n].

4. Constructing an optimal solution from the computed information

From step 2, it is understood that if maxSubArray(A,i-1)≤0 then we simple discard the value. This implies that we start a new subarray starting from i. We store this result in s. If the sum of subarray starting from s to i is greater than max_sum(Stored in result[2]), we update the left to s, right to i and result[2] = dp[i].
The time complexity of this algorithm is O(n)
The space complexity of this algorithm is O(n)

- **ALGORITHM4**

We use brute force method to find the maximum sum of submatrix A of size m*n. The idea is to iterate through each row and each columm and find the sum of elements in between them.

So, we will initially loop for each row(representing the starting row), each column(representing the starting column). In the loop we iterate from start row to m(representing ending row) and starting column to n(representing ending column)

Now we find the sum of elements from (start_row,start_column) to (end_row,end_column). If the sum of elements in this sub matrix is greater than previous sum then we update the result.

The Time complexity of the algorithm is O(n^6)
The Space complexity of the algorithm is O(1)

- ## ALGORITHM5

  Instead of calculating the sum for each sub matrix, we store the sum of elements in prefix_sum. So, for each sub matrix it only takes O(1) time to calculate the sum and do the necessary updates.
  The time complexity of this algorithm is O(n^4)
  The space complexity of this algorithm is O(mn)

- ## ALGORITHM6

  We optimize the algorithm even further by fixing left and right of the submatrix. We copy the sum of elements in each row from left to right to prefix_sum. Simply put, prefix_sum[i] contains sum of elements from matrix[i][left] to matrix[i][right]. Now we pass the prefix_sum to Dynamic Programming algorithm computed in Alg3. The result returned from the algorithm contains the start and end indices of maximum sum subarray in prefix_sum. It can be understood that the start and end indices returned would be the top and bottom indices of the matrix.

  If the sum obtained from ALG3(prefix_sum[left:right]) is greater than maximum_sum_seen(stored in res[4]), we make the necessary updates.
  The time complexity of this algorithm is O(n^3).
  The space complexity of this algorithm is O(m).

3. Experimental Comparative Study

We have created randomly generated input files of various sizes for measuring the performance and correctness of Task1, Task2, Task3A, Task3B, Task4, Task5, Task6. We have attached the input files in the zip folder.

The names of the input files used for Task1,Task2,Task3A,Task3B are as follows:

- Problem1_n1000 where we chose n=1000 and generated the test case.
- Problem1_n2000 where we chose n=2000 and generated the test case.
- Problem1_n3000 where we chose n=3000 and generated the test case.
- Problem1_n4000 where we chose n=2000 and generated the test case.
- Problem1_n5000 where we chose n=2000 and generated the test case.

The names of the input files used for Task4,Task5,Task6 are as follows:

- Problem2_20_20 where we chose m=20 and n=20 to generate the data set.
- Problem2_40_40 where we chose m=40 and n=40 to generate the data set.
- Problem2_60_60 where we chose m=60 and n=60 to generate the data set.
- Problem2_80_80 where we chose m=80 and n=80 to generate the data set.
- Problem2_100_100 where we chose m=100 and n=100 to generate the data set.

Below is the table to represent the time taken in nanoseconds for executing Task1, Task2, Task3A, Task3B, Task4 against the input files Problem1_n1000, Problem1_n2000, Problem1_n3000, Problem1_n4000, Problem1_n5000.

Test Sizes	**Time taken for Task1 (ns)**	**Time taken for Task2 (ns)**	**Time taken for Task3A(ns)**	**Time taken for Task3B(ns)**
n=1000	173526822	4378496	442737	656270
n=2000	1310406548	13148341	431263	800985
n=3000	2958084670	8598114	580675	976636
n=4000	6115082103	11597386	603957	1298577
n=5000	10875643210	29316562	682442	1442358

Below is the table to represent the time taken in nanoseconds for executing Task4, Task5 and Task6 against the input files Problem2_20_20, Problem2_40_40, Problem2_60_60, Problem2_80_80, Problem2_100_100.

m & n values	Time taken for Task4 (ns)	Time taken for Task5 (ns)	Time taken for Task6 (ns)
m=20,n=20	23587972	20570985	22307111
m=40,n=40	182039334	29762063	24411658
m=60,n=60	1453960753	43573139	36503166
m=80,n=80	6659851580	58468492	17330195
m=100,n=100	9651821620	230866681	64562673

We have conducted a performance comparison between Task1 and Task2 by generating a two dimensional plot of running time (y-axis) against input size (x-axis).The plot is as shown in the figure 1.

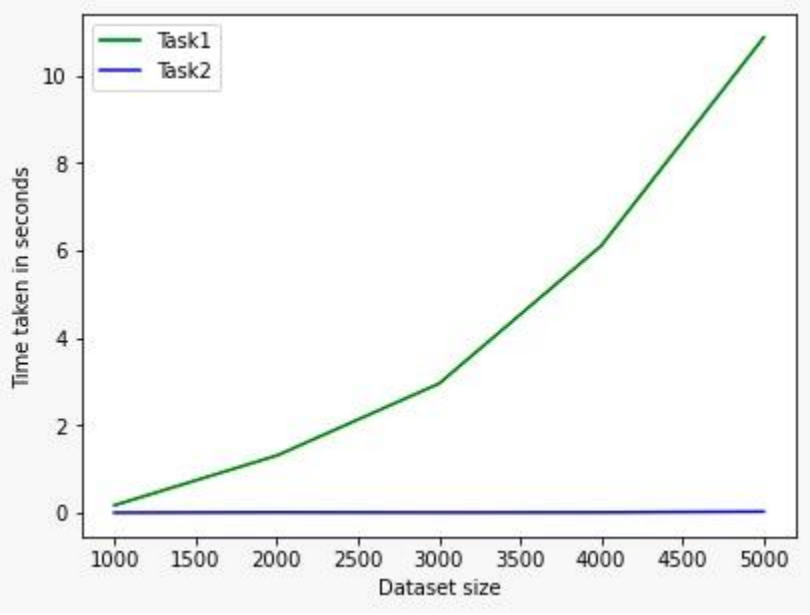

Figure 1

We have conducted a performance comparison between Task2 and Task3A by generating a two dimensional plot of running time (y-axis) against input size (x-axis).The plot is as shown in the figure 2.

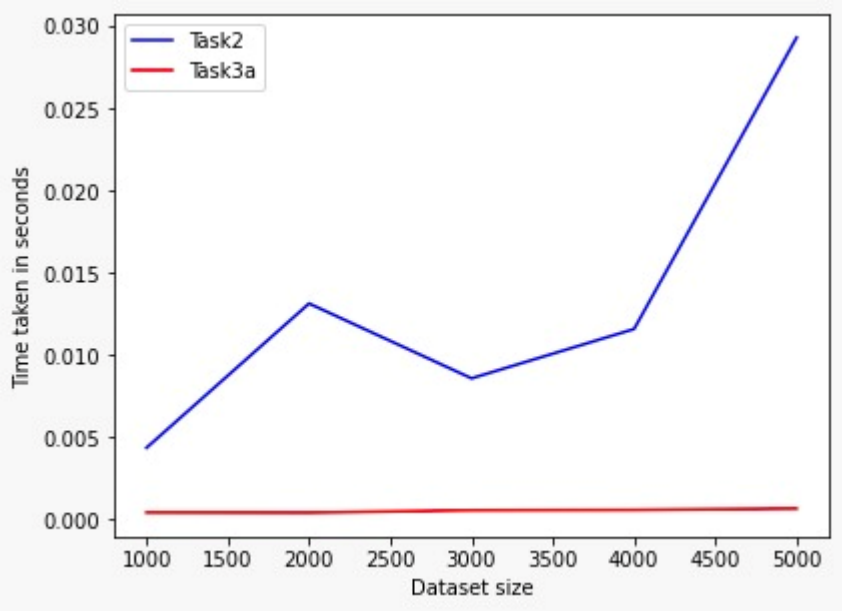

Figure 2

We have conducted a performance comparison between Task3A and Task3B by generating a two dimensional plot of running time (y-axis) against input size (x-axis).The plot is as shown in the figure 3.

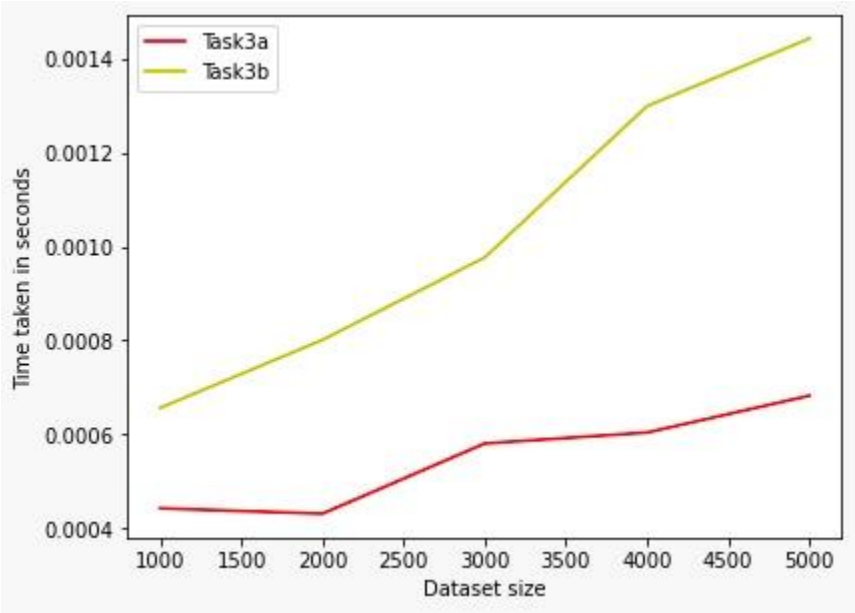

Figure3

We have conducted a performance comparison between Task5 and Task6 by generating a two dimensional plot of running time (y-axis) against input size (x-axis).The plot is as shown in the figure 4.

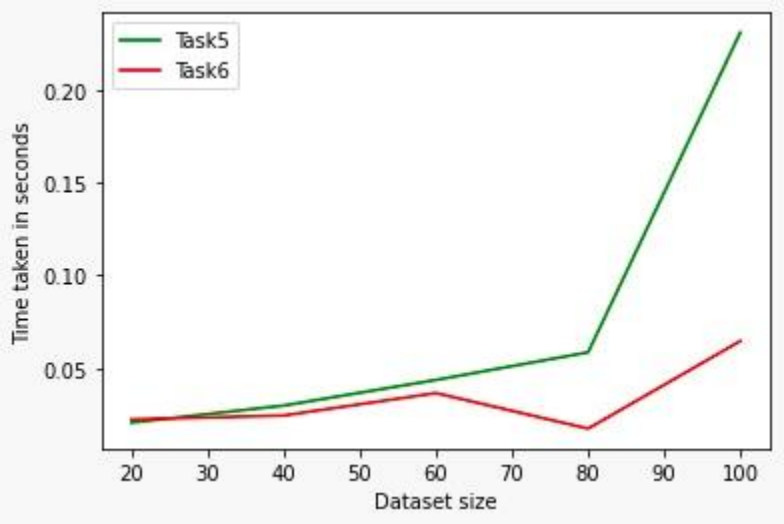

Figure 4

We have conducted a performance comparison between Task4,Task5 and Task6 by generating a two dimensional plot of running time (y-axis) against input size (x-axis).The plot is as shown in the figure 5.

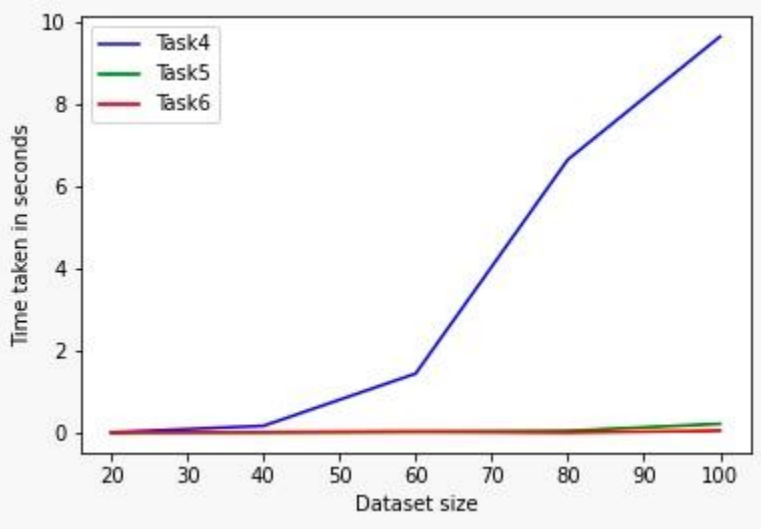

Figure 5

Conclusion

ALGORITHM1:

Since, we are using brute force, this algorithm is not efficient for larger input sizes. It takes a lot of time to execute on larger input sizes.

ALGORITHM2:

From the plots, we can observe a significant improvement in time taken to execute the program. But it is not suitable for input sizes > 100000

ALGORITHM3:

Clearly, Alg3 is best suitable for this problem. We implemented the dynamic programming approach in 2 ways: Top-down and Bottom-up. The memoization method out performs bottom-up model as there is implicit time and space required for the recursion stack.

ALGORITHM4:

The brute force method computed in this algorithm is not at all efficient. The time complexity is O(n^6), so it takes lot of time to compute, and thus is unsuitable for real applications.

ALGORITHM5:

Alg5 is the slightly optimized version of Alg4. It performs comparatively better than Alg4 but can be improved much as in Alg6

ALGORITHM6:

Applying the Alg3 to Alg4 reduces the time complexity greatly from O(n^6) to O(n^3). This is the best algorithm for problem-2.