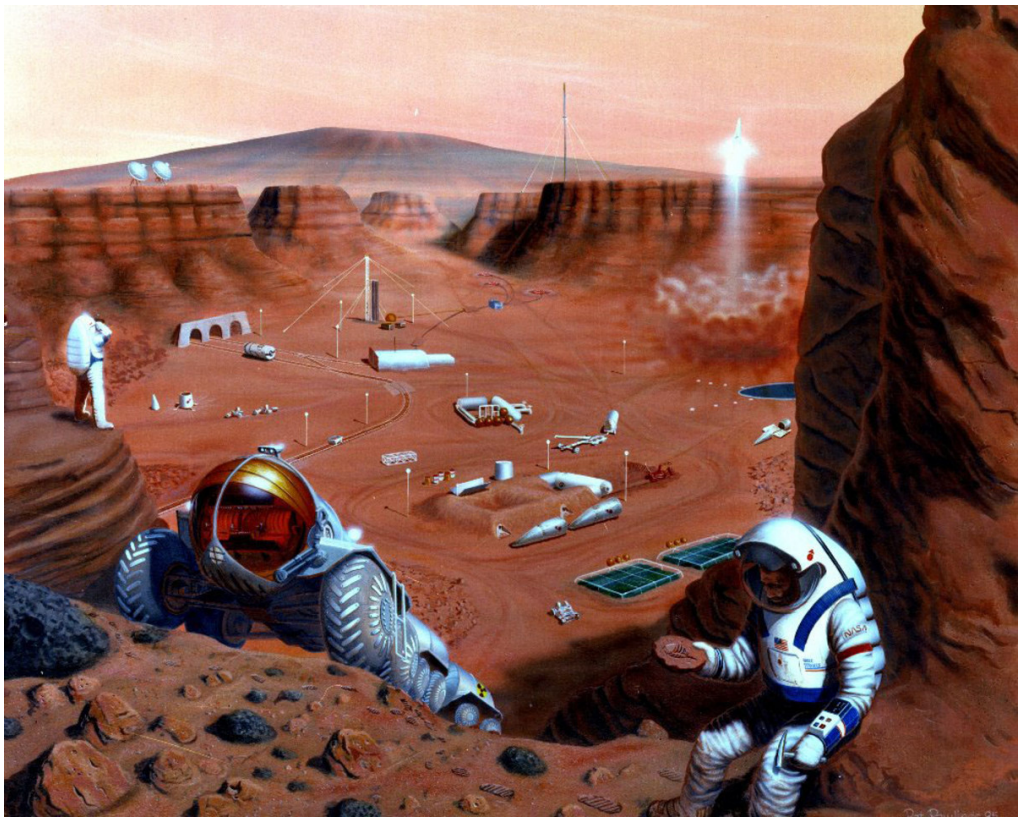
Algorithms Programming Project

Jaswant Reddy Kankanala
(j.kankanala@ufl.edu)

Manish Kumar Reddy Gangula
(gangula.m@ufl.edu)



Abstract

Getting started, this project entails about in-depth understanding & nuances involved in the implementation of efficient design strategies for solving possible real-world situations. The problem statement describes that, due to the recent advancements and scope towards colonization on mars, several metrics regarding the climatic conditions are being taken into consideration for Human survival on Mars. Some factors are listed below:

- Air Quality Index *M[i, j]* is a rating or metric used to determine the quality of air, to select building new home, for all *(i, j)* represents the location on the designated space. Values assigned can be either *Positive/ Negative*
- Provided a suitable Rectangular Area.
- Total Air Quality Index is summation of all included indices in the rectangular area.

Problem Statements:

1. Design an Algorithm subject to the condition of obtaining the Maximum sum of the Air Quality Index in the Contiguous subarray locations.
2. Design an Algorithm subject to a similar condition as above with an additional criterion of finding the Maximum sum of the Air Quality Index in 2-Dimensional Array of size *m x n.*

Design Approaches:

A traditional approach is to first identify the approach to solve the problem and then later try to refine/ optimize it such that it consumes the minimal number of resources (Time and Space). The same strategy is followed and each of the problem statement is solved with three variants (Brute Force and Dynamic Programming with different Flavors) of both the Time and Space Complexities.

Language of Choice:

We are using Java as the programming language as choice to work with it. Also, the project includes a makefile which in turn helps to run the project to execute with the desired strategy.

Problem 1: Given an array A of n integers (positive or negative), find a contiguous subarray whose sum is maximum.

Algorithm 1: Using Brute force Algorithm to solve it in $\theta(n^3)$

Firstly, Brute force Algorithm is used to list out all the possible subarrays. Secondly, finding the sum of each of the subarray of the air quality index values and noting down the start and end positions of the subarray. Next, picking the highest sum subarray as the optimal solution to the problem statement.

Algorithm:

```
intMaxBoundSumContiguousSubArray(arr):
        globalMaxSum = integer minimum value // −2^31
        start = 0, end = 0 //used to store the start and end indices of subarray
        for(startIndex = 0;startIndex<arr.length; startIndex++):
              for (endIndex = startIndex; endIndex < n; endIndex++):
                    localSum = 0
                    for (k = startIndex; k <= endIndex; k++):
                          localSum += arr[k]
                          if(globalMaxSum<localSum):
                          globalMaxSum= localSum, start = startIndex+1,
                          end = endIndex+1
        return start, end, globalMaxSum
```

From the above snippet, the function intMaxBoundSumContiguousSubArray() is passed with an Integer array. Variables like "`start`", "`end`", "`globalMaxSum`" are used to capture things like start of the process for performance visualization, and the size of the array dealing with, and finally the output array to store the indices and the max sum value to return to the console.

In here, the logic is to iterate over all the combinations of subarrays possible using the input array "`arr` []" and using the For loop we iterate over the array given using two index values as startIndex with an initial value 0 and ranging up till the size of the array. Another index value to capture the end value to define the end of each of the iteration as endIndex with an initial value equal to startIndex and with the size of the subarray varying to through a maximum being size of the array.

Post the process, we now have successfully created the start and end Indices for all the possible combinations. Let us try to depict this using an example below.

Array: A[19, 25, 23, -19, 24, -9, -34, -22]

Array Size: 8

Start Index	End Index	Sub Array	Size
0	0	[19]	1
0	1	[19, 25]	2
0	2	[19,25, -19]	3
.	.	.	.
.	.	.	.
7	7	[-34]	1
7	8	[-34, -22]	2
8	8	[-22]	1

Table: 1.1

From the above Table 1.1, we now iterate over all the subarray Indices and calculate the sum of the elements present and store it in the array as `globalMaxSum`. For the base case we assign the Integer Min Value to the max sum for the first iteration which is –2147483648 (as the possible input values can range from negative maximum to positive maximum of an Integer) and moving further we compare the same with the evaluated sum for each of the possible subarray, compare with the base value and update array if the computed sum is larger than the existing sum, also captured the start and end points for retrieving the index values contributing to the scenario.

Start Index	End Index	Sub Array	Size	Evaluating Sum	Sum Value	Global Max Sum
0	0	[19]	1	19	19	19
0	1	[19, 25]	2	19+25	44	44
0	2	[19,25, -19]	3	19+25+(-19)	25	44
.	.	.	.	.		
.	.	.	.	.		
7	7	[-34]	1	-34	-34	44
7	8	[-34, -22]	2	(-34)+(-22)	-56	44
8	8	[-22]	1	-22	-22	44

Table 1.2

From the above table we can depict that the Global sum is initially set to a minimum amount. As we continue with the iterations, we update it with the Sum that is larger than the assigned value. During the 3rd iteration we observe that the Sum is 25 and Global Max Sum in the 2nd iteration is 44. So, we retain 44 as the max sum and continue to the next iteration, stating that the optimal solution available till now is in the subarray [19, 25] with a sum of 44. The same process is continued till we exhaust all the possible subarrays, and hence we are guaranteed to get an optimal solution.

Correctness of the Algorithm: In the brute force approach mentioned, we observe that we go through each iteration finding the each and every possible subarray that might lead to the possible max sum sub array. We compare each time the solution with existing solution thereby holding the correctness of the algorithm true.

Asymptotic Analysis:

1. Time Complexity:

In the above-mentioned algorithm, we are iterating over the subarray size twice to extract all the possible combinations and in-order-to compute sum we once again iterate over the size of the subarray elements making the Time Complexity $\theta(n^3)$

2. Space Complexity:

In the above-mentioned algorithm, several variables are used to capture the vital performance metrics which are assigned on θ (1).

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task1.java
Enter size of input
8
Enter the integer set
19 25 23 -19 24 -9 -31 -22
1 5 72
```

Case2:

```
C:\Users\Public\Manish\AOA Test>java Task1.java
Enter size of input
8
Enter the integer set
-99 -38 -81 -29 -91 -3 -36 -46
6 6 -3
```

Case3:

```
C:\Users\Public\Manish\AOA Test>java Task1.java
Enter size of input
8
Enter the integer set
-1 -2 -1 0 0 -1 0 0
4 4 0
```

Algorithm 2: Using Dynamic Programming to solve it in $\theta(n^2)$

The process of finding the Maximum sum is similar to the previously discussed algorithm. From Table 1.1, we obtain all the possible subarrays. There is a repetitive process of preforming the same mathematical calculation of adding the subarray values to obtain the sum of elements in it, where one can use the result from the previously computed subarray as the pattern involves just an additional element at the end of the "k" sized subarray. We shall discuss it in more detail in the example below.

Start Index	End Index	Sub Array	Size	Sum	Global Max Sum
0	0	[19]	1	19	19
0	1	[19, 25]	2	19+25	44
0	2	[19,25, -19]	3	44+(-19)	44
.	.	.	.	.	
.	.	.	.	.	
7	7	[-34]	1	-34	44
7	8	[-34, -22]	2	(-34)+(-22)	44
8	8	[-22]	1	-22	44

Table 1.3

From the above Table 1.3, by using the previously computed sum value (displayed in RED INTEGERS) we can get rid of the re-evaluation of the same result instead referring to it and can save time.

Algorithm:

```
maxContiguousDPQuadratic(arr):
      globalMaxSum = integer minimum value // −2^31
      start = 0, end = 0 //used to store the start and end indices of subarray
      for (startIndex = 0; startIndex < arr. length; startIndex++):
                localSum = 0
          for(endIndex=startIndex; endIndex<arr.length; endIndex++):
              localSum += localSum[endIndex]
              if (globalMaxSum < localSum):
                  globalMaxSum = localSum, start = startIndex + 1
                  end = endIndex+1
        return start, end, globalMaxSum
```

In the algorithm described above, all the possible subarrays are listed out with help of the two 'for' loops and storing the computed sum of all the subarray elements into "`localSum`" variable. The variable is reused to just add the next contiguous element in the subarray in the next iteration.

Using this logic, we have managed to fill the Table 1.3, the value of the column: Sum in the 2nd row is computed not by iterating it across the subarray, instead using the stored value '19' to compute the sum to

be ‘44’. Similarly, for the 3rd row, Sum of the elements from the index positions 0-2 is evaluated by using the sum of indices from 0-1 and performed addition 44+(-19) instead of evaluating 19+25+(-19). Extending the strategy for the large input size of “1 million” elements of the subarray, instead of adding 1,000,000 elements we can use the sum of the 999,999-element subarray sum and just add the next contiguous element to it. This helps us in saving a ton of time in re-doing the same activity again and again. Also, this helps to get rid of an extra for loop and reduces the Asymptotic complexity.

Correctness of the Algorithm: Unlike the brute force approach but holding the ideas of possibilities check intact we find that we go through each iteration finding the each and every possible subarray that might lead to the possible max sum sub array. But we do not consider iterating again to check the previous value, rather, we check it in every subarray iteration which proves the correctness of the algorithm and thereby optimizing the brute force approach.

Asymptotic Analysis:

1. Time Complexity:

In the above-mentioned algorithm, we are iterating over the subarray size twice to extract all the possible combinations and in-order-to compute sum we make use of the previously computed sum of (n-1) subarray elements and add the nth element of the subarray for evaluating the Max sum for Size-n Subarray with a Time Complexity $\theta\ (n^2)$.

2. Space Complexity:

In the above-mentioned algorithm, several variables are used to capture the vital performance metrics which are assigned on $\theta\ (1)$.

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task2.java
Enter size of input
8
Enter the integer set
19 25 23 -19 24 -9 -31 -22
1 5 72
```

Case2:

```
C:\Users\Public\Manish\AOA Test>java Task2.java
Enter size of input
8
Enter the integer set
-99 -38 -81 -29 -91 -3 -36 -46
6 6 -3
```

Case3:

```
C:\Users\Public\Manish\AOA Test>java Task2.java
Enter size of input
8
Enter the integer set
-1 -2 -1 0 0 -1 0 0
4 4 0
```

Algorithm 3: Using Dynamic Programming Algorithm to solve it in $\theta(n)$

Task 3A: Recursive Implementation using Memoization.

The Recursive approach of Dynamic Programming is one of the Paradigm techniques to obtain the optimal solution which has Overlapping Subproblems and using DP helps to get the solution with ease. One such approach is to use recursive calls to derive the necessary values. In here, we are going to extrapolate the recursive concepts to identify the effective subarray which ultimately leads to the optimal substructure. From the below snippet, we are using the recursive function "maxContSumMemoized()" which takes input parameters as the input[] and array size.

Algorithm:

```
globalMax =0, startIndex=0, endIndex=0, localStartIndex = 0

computeMaxSum(arr):
      globalMax = arr[0]
      maxContSumMemoized(arr)

 maxContSumMemoized (arr, arrSize):
      //handle the base case
       if (arr.length is 1):
            return arr [0]
       localMax = maxContSumMemoized(arr, arr.Length-1) //Recursive call
       if (localMax < 0):
           localMax = arr[arr.length -1]
           localStartIndex = arr.length -1
       else:
           localMax += arr[arr.length -1]
      if (globalMax < localMax):
           globalMax = localMax, startIndex = localStartIndex
           endIndex = arr.length -1
      return startIndex, endIndex, localMax
```

The substructure property of this approach is maxContSumMemoized(input[], array_size) for all array_size belongs to {1, 2, 3, …. , size -1} which returns the local maximum (`localMax`) value from each of the recursive calls to evaluate the global maximum(`globalMax`).

$$maxContSumMemoized(i) = \begin{cases} arr[i], & i = 0 \\ \max\left(arr[i], maxContSumMemoized(i-1) + arr[i]\right), & i > 0 \end{cases}$$

Correctness of the Algorithm: In the approach we are recursively checking for the max of arr[i] with the sum of max sum till the array index i-1 (overlapping subproblem) and the current element arr[i]. Every-time we check for a solution at an index, it is therefore a case of the maximum of either the sum of already existing and proven maximum of i-1 subproblem with current index, or the solution of previous sub problem. Which holds the correctness of the algorithm intact using the recursive subproblem solving approach (DP)

Optimal Substructure for the recursive Dynamic Programming paradigm to find the Maximum Contiguous Sum across the Array. To obtain the value of `maxContSumMemoized (arr, arrSize,` we are recursively calling its sub-problems and we obtain the Optimal solutions if and only if we manage to get the optimal solution to its sub-problems thus, it is making the use of the Optimal sub-structure property as we can observe from the above function. Let us understand it with an example below.

Array: A[19, 25, 23, -19, 24, -9, -34, -22]

Array Size: 8

Array Size	Local Max	Global Max	Start Index	End Index
8	32 - 22 = 10	72	0	4
7	63 - 31 = 32	72	0	4
6	72 - 9 = 63	72	0	4
5	48+24 = 72	72	0	4
4	67-19 = 48	67	0	2
3	44+23 = 67	67	0	2
2	19+25 = 44	44	0	1
1	19	19	0	0

Table: 1.4

We can clearly see from the above table that while deriving the "localMax" for the third iteration, we need to compute {19+ 25+ 23 = 67} while summation of some portion of the elements in the example {19+25 = 44} is already computed in the second iteration of the array. This shows that the problem statement has the ***Overlapping Sub-Problems*** in it making it a good example to use Dynamic Programming to effectively get the solution.

The recursive calls to the maxContSumMemoized(input[], array_size) makes calls to itself until the base case conditions are met. From the above table and code snippet, the base case is when the array_size = 1 i.e., each recursive calls passes the parameters with one less than the array_size it has until it reaches to size = 1. The values for localMax = 19 is obtained and it is proceeded further to evaluate globalMax by performing the checks with its initial value to localMax and is assigned with localMax for the first Iteration. Also, as per the problem statement, we need to keep track of the start and end indices. The values are updated when there is a change in the globalMax. Also, a case when both are equal, we compare the start and end

indices with current recursions with the array_size to keep track of the changes. From the above table, with the Dynamic Programming recursion on the array_size of 8 and filled the table with the values we see that GlobalMax of 72 from the subarray starting from 1-5 index positions.

Asymptotic Analysis:

1. Time Complexity:

With the Recursive Dynamic Programming algorithm, we are recursively calling the function with the array size less than the previous one, we perform the n calls with the size from {1, 2, 3, …, n} making the Time complexity to be θ (n).

2. Space Complexity:

In the above-mentioned algorithm, several variables are used to capture the vital performance metrics which are assigned on θ (n).

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task3a.java
Enter size of input
8
Enter the integer set
19 25 23 -19 24 -9 -31 -22
1 5 72
```

Case 2:

```
C:\Users\Public\Manish\AOA Test>java Task3a.java
Enter size of input
8
Enter the integer set
-99 -38 -81 -29 -91 -3 -36 -46
6 6 -3
```

Case 3:

```
C:\Users\Public\Manish\AOA Test>java Task3a.java
Enter size of input
8
Enter the integer set
-1 -2 -1 0 0 -1 0 0
4 4 0
```

Task 3B: Iterative Bottom-Up Dynamic Programming Implementation.

The Iterative Bottom-Up approach of Dynamic Programming Paradigm techniques used obtain the efficient solution which has Overlapping Subproblems and using DP helps to get the solution with ease. One such approach is to iteratively derive the necessary summation values and update them accordingly. In here, we are going with the Iterative approach to identify the optimal subarray with maximum sum which ultimately leads to the optimal substructure. From the below snippet, we are using the iterative function "IterativeBottomUpDP ()" which takes input parameters as the input[] and array size.

The main idea behind this is to check the sum of the traversed subarray of any length over the process has to contribute towards finding the optimal solution, in here we wanted to obtain the maximum sum in the subarray, so the sum of the traversed subarray should be at least positive to make the difference. This formulates to the very first test strategy we employed to test to whether consider the solution or not?

Algorithm:

```
maxContBFMatrix(arr, rows, columns):
      globalMaxSum = localMaxSum = arr[0]
      globalStart=0, globalEnd=0, localStart=0
      for (loop=1; loop < arr.length; loop++):
            if (localMaxSum + arr [loop] < arr [loop]):
                  localMaxSum = arr[loop]
                  localStart = loop
            else:
                  localMaxSum += arr[loop]
            if (globalMaxSum < localMaxSum):
                  globalMaxSum = localMaxSum, globalStart = localStart
                  globalEnd = loop
      return globalStart, globalEnd, globalMaxSum
```

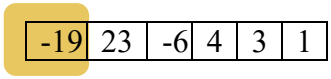
For example, let us consider an array [-19, 23, 5, …., 10]. So, starting from the left index position, we store the local sum obtained upon traversing the array. We have `localMaxSum` as "-19" and is negative and upon going to the second index position, we have a positive value "23" when both (0, 1) indices are considered we get the reduced value as because of the element at '0' position. Thus, it is better to avoid the first and start to consider the values from index '1'. By induction, we are going to implement the same to the traversed subarray and we find the `localMaxSum` and perform the check if greater than the combined element value of the subarray and include the element, if not, we discard the traversed subarray and update the start Index point to the current index value. Also, for each iteration we perform update of the "`globalMaxSum`" comparing with the "`localMaxSum`". When we reach the end of the array, we have a set of variables which hold the information related the Maximum sum one can obtain from the selecting contiguous subarray in the respective array using "`globalStart`", "`globalEnd`" and "`globalMaxSum`".

Sample Array

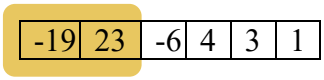
-19	23	-6	4	3	1

Let us understand the process with the help of an example.

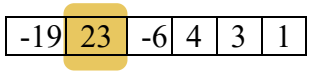
In the first step,



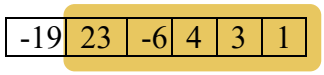
We have -19 loaded for the subarray traversal and updating the variable `localMaxSum` to -19. Going for the next element i.e., 23.



Now, we have two elements with us {-19, 23} and adding them together we obtain `localMaxSum` as 4, it indeed is increased the value from the previous iteration. But when considered the other way around if we consider {23} only we are going to have `localMaxSum` as 23 and yes, it is larger than considering the both the elements. Thus, whenever we obtain the `localMaxSum` as a negative value it decreases the total sum value, and it must be discarded and have move on with the next element as below.



Traversing ahead to the next index, we have -6 and `localMaxSum` becomes 17 which is a positive number and `localStart` to index 1. Continuing till the end of the array we obtain the values as



`globalStart`: 2

`globalEnd`: 6

`globalMaxSum`: 25

Correctness of the Algorithm: Similar to the previous approach of recursive formulation, but in a bottom-up fashion, the solution of the algorithm depends on the iteration that we go through, that is, the solution from i to j+1 depends on the solution from i to j (intuitively from i to j-1 and so on) comparing which would be the maximum among the two, thereby maintaining the correctness of the approach.

Asymptotic Analysis:

1. Time Complexity:

With the Iterative algorithm in Dynamic Programming approach, we are iterating over the array size and perform the global and local summation checks and track the start and end indices making Time complexity to be $\theta(n)$.

2. Space Complexity:

In the above-mentioned algorithm, several variables are used to capture the vital performance metrics which are assigned on θ (1).

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task3b.java
Enter size of input
8
Enter the integer set
19 25 23 -19 24 -9 -31 -22
1 5 72
```

Case2:

```
C:\Users\Public\Manish\AOA Test>java Task3b.java
Enter size of input
8
Enter the integer set
-99 -38 -81 -29 -91 -3 -36 -46
6 6 -3
```

Case3:

```
C:\Users\Public\Manish\AOA Test>java Task3b.java
Enter size of input
8
Enter the integer set
-1 -2 -1 0 0 -1 0 0
4 4 0
```

Problem 2: Given a 2-D array Array M of size *m x n* of integers (positive or negative), We need to find a rectangle whose sum is maximum.

Algorithm 4: Using Brute force Algorithm to solve it in $\theta(n^6)$

Based on the Problem 1, Brute force is used to list out all the possible subarrays and then finding the Max sum of the air quality Indices gave us the solution. With the same idea and extrapolating it to across 2-D Array, we can list down all the possible rectangles by Iterating through all possible span of elements using "4-For loops" in the 2-D array by identifying the indices of the start and end of both the rows and columns.

Next, iterating through over all the elements present in the rectangular portion and is summed with all the elements to know the possible sum. Using the similar idea in Problem 1, Task 1, we maintain a global matrix sum as "maxBFMatSum" and the global index positions as "leftIndex", "rightIndex", "topIndex", "endIndex" and are compared with each of the values obtained from the rectangular portion derived and are updated with the highest values. Let us understand better with an example stated below.

Algorithm:

```
maxContBFMatrix(arr, rows, columns) {
   globalMaxSum = integer minimum value // −2^31
   leftIndex=0, rightIndex=0, topIndex=0, endIndex=0
   for (startLeft = 0; startLeft < rows; ++startLeft):
      for (startTop = 0; startTop < columns; ++startTop):
        for (endLeft = startLeft; endLeft < rows; ++endLeft):
          for (endTop = startTop; endTop < columns; ++endTop):
            localSum = 0
            for (rowLoop = startLeft; rowLoop <= endLeft; ++rowLoop):
              for (colLoop = startTop; colLoop <= endTop; ++colLoop):
                  localSum += arr [rowLoop][colLoop]
            if (localSum > globalMaxSum):
                  globalMaxSum=localSum, leftIndex=startTop+1,
                  rightIndex=endTop+1, topIndex=startLeft+1,
                  endIndex=endLeft+1
   return topIndex, leftIndex, endIndex, rightIndex, globalMaxSum
```

Example:

Input: 4 4

$$\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

Based on the above input, we have 4 rows and 4 columns, and the next input corresponds to the matrix elements in *4x4* matrix. Below is the table we obtain based on the algorithm we discussed earlier, and we will investigate some of the rows for better understanding and correctness of the algorithm.

Start Left	Start Top	End Left	End Top	Local Sum	MaxBFMatSum	Left	Right	Top	End
0	0	0	0	21	21	1	1	1	1
0	0	0	1	21+3 = 24	24	1	2	1	1
0	0	0	2	21+3+(-17)= 7	24	1	2	1	1
0	0	0	3	-7	24	1	2	1	1
0	0	1	0	36	36	1	1	2	1
0	0	1	1	25	36	1	1	2	1
0	0	1	2	-23	36	1	1	2	1
0	0	1	3	-65	36	1	1	2	1
0	0	2	0	47	47	1	1	3	1
0	0	2	1	15	47	1	1	3	1
0	0	2	2	-9	47	1	1	3	1
0	0	2	3	-57	47	1	1	3	1
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
3	1	3	1	23	47	1	1	3	1
3	1	3	2	0	47	1	1	3	1
3	1	3	3	23	47	1	1	3	1
3	2	3	2	-23	47	1	1	3	1
3	2	3	3	0	47	1	1	3	1
3	3	3	3	23	47	1	1	3	1

Table 1.5

Let us consider that the index positions start from 0,1,2,3. We shall follow the algorithmic instructions step by step and prove the correctness.

1. Firstly, let us consider the initial step where the indices of the for loop are (0, 0, 0, 0) this makes us to select the first element of the matrix as shown below.

$$\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

with the above input, we select only the first element in the matrix and the variables “LocalSum” and “MaxBFMatSum” are evaluated and assigned, and the positions related to the rectangle are also captured as (1, 1, 1, 1). Now repeated the process to further increasing the boundary of the rectangle.

2. For the second iteration, (0, 0, 0, 1) is chosen as the positions for the matrix making it to expand it to the right-hand side as below.

$$\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

With the respective positions we obtain the rectangle as the above yellow region and "LocalMax" is computed as the summation of all the elements in the yellow region as (21+3) = 24. Now the value of "MaxBFMatSum" is compared with the "LocalMax" and if it is less than it is updated to 24 from 21 and also the positional indices of the rectangle is also updated from (1, 1, 1, 1) to (1, 2, 1, 1).

3. Now for the third iteration, (0, 0, 0, 2) is chosen as positions of the matrix and is represented as

$$\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

With the new positions of the rectangle, we need to compute the values of "LocalMax" as summation of elements as (21+3+(-17)) = 7, which is less than the value of "MaxBFMatSum" and there will be no updates for the positions and MaxBFMatSum.

4. Let us consider the case (0, 0, 2, 2) and the matrix and the rectangular portion looks like this.

$$\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

With the new positions we have got the rectangle with positions as (1, 1, 3, 3) and "LocalMax" is computed as (21+3+(-17)+15+(-14)+(-31)+11+(-21)+24) = -9. We then repeat the same process of comparing it with "MaxBFMatSum" which holds 47 as its value and it is greater and does not update anything. By induction, we proceed till the last possible index positions (3, 3, 3, 3) with "MaxBFMatSum" as 47 at (1, 1, 3, 1) position.

Correctness of the Algorithm: Similar to the 1D array brute force approach, we find all possible sub-Matrices of the input matrix in finding the left, right, top and bottom start in four loops, and comparing them in other two loops. As we utilize all possible sub matrices, this proves the correctness of the algorithm

Asymptotic Analysis:

1. Time Complexity:

With the Brute force approach, we are iterating over the 2-D array to extract all the possible combinations and in order to compute summation of all the elements inside it, and we once again iterate over the rows and columns of the 2-D array elements making the Time Complexity $\theta\ (n^6)$.

2. Space Complexity:

In the above-mentioned algorithm, several variables are used to capture the vital performance metrics which are assigned on $O\ (1)$.

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task4.java
Enter size of input Array
4 4
21 3 -17 -14
15 -14 -31 -28
11 -21 24 -6
-2 23 -23 23
1 1 3 1 47
```

Case2:

```
C:\Users\Public\Manish\AOA Test>java Task4.java
Enter size of input Array
4 4
0 5 -11 -61
-41 -88 -24 -65
53 -18 29 -37
-38 52 0 5
3 1 4 3 78
```

Case3:

```
C:\Users\Public\Manish\AOA Test>java Task4.java
Enter size of input Array
4 4
-1 -1 -1 -1
0 0 0 0
-1 -1 -1 -1
0 0 0 0
2 1 2 1 0
```

Algorithm 5: Using Dynamic Programming approach, we try to evaluate the summation of the elements in the rectangular elements from the 2-D array in θ (n^4) with an extra space of O(mn)

In the previous algorithm, Table1.5, "MaxBFMatSum" is computed each-and-every time instead if we rely on the previously computed and stored value, we can exclude the exponential complexity to make the algorithm run faster. Here, we are using a matrix of the same size as *(m+1 x n+1)* to store the computed values in the previous iterations.

Algorithm:

```
maxContDPn4Matrix(arr, rows, columns):
        new blank array temp[rows][columns]
        for(row=1; row<rows+1; row++):
            for(int col=1; col<columns+1; col++):
                 // Considering the elements in the previous columns
                 if(row > 0):
                   temp [row][col]=arr[row-1][col-1]+ temp [row-1][col]
                 if(col > 0):
                    temp[row][col]=temp[row][col]+ temp[row][col-1];
                 // Removing the elements which are added twice along the diagonal.
                 if(row > 0 && col > 0){
                    temp[row][col]=temp[row][col]-temp[row-1][col-1]
        globalMaxDPn4 = Integer.MIN_VALUE; // −2^31
        startRowIndex=0, startColIndex=0, endRowIndex=0, endColumnIndex=0

        // Iterating over all the possibilities to frame the rectangular elements.
        for(rowLoop=1; rowLoop<rows+1; rowLoop++):
            for (columnLoop = 1; columnLoop < columns+1; columnLoop++):
                for (rLoop = rowLoop; rLoop < rows+1; rLoop++):
                    for (cLoop = columnLoop; cLoop < columns+1; cLoop++):
                        subProblemSum = temp[rLoop][cLoop]

                        if (rowLoop > 0)
                            subProblemSum -= temp[rowLoop - 1][cLoop]
                        if (columnLoop > 0)
                            subProblemSum -= temp[rLoop][columnLoop - 1]
                        if (rowLoop > 0 && columnLoop > 0)
                            subProblemSum += temp[rowLoop - 1][columnLoop - 1]
                        if (subProblemSum > globalMaxDPn4) {
                            startRowIndex=rowLoop, startColIndex=columnLoop
                            endRowIndex = rLoop, endColumnIndex = cLoop
                            globalMaxDPn4 = subProblemSum
        return startRowIndex, startColIndex, endRowIndex, endColumnIndex, globalMaxDPn4
```

In this approach we are trying to use an extra 2-D array space with a size *(m+1 x n+1)* for storing the summation of the possible rectangular element values from the input 2-D array of size *m x n.* We are adding

one row & col (typically first row and column) to the temp matrix filled with Zeros '0' in order to incorporate the same logic to the edge cases dealing with the first row and first column. We begin with filling the values into the temp 2-D array from index (1,1) and let us work on an example :

Input: 4 4

$$InputArray(4x4)\begin{bmatrix}21 & 3 & -17 & -14\\ 15 & -14 & -31 & -28\\ 11 & -21 & 24 & -6\\ -2 & 23 & -23 & 23\end{bmatrix} \qquad TempArray(5x5)\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

1. We begin with referring to the Temp array starting from index (1,1) marked in yellow on the top, we iterate over all the combinations by checking the values as below:

$$Temp[r][c] = \begin{cases} Input[r-1][c-1] + Temp[r-1][c] \ continue, & r > 0 \\ Temp[r][c] + Temp[r][c-1] \ continue, & c > 0 \\ Temp[r][c] - Temp[r-1][c-1] \ continue, & r > 0 \ and \ c > 0 \end{cases}$$

For (1,1):

1) Firstly, performing the first condition: $Input[1-1][1-1] + Temp[1-1][1], \ \&r > 0$ gives us:
   a) Temp[1][1] = 0
   b) Input[0][0] = 21
   c) Temp[0][1] = 0

The values are color coded above in the matrix and are computed (Yellow = Green + Red)and filled in the Temp array.

2) Nextly, we perform the second equation: $Temp[1][1] + Temp[1][1-1], \ \&c > 0$ similar to the previous step and update the value from the step 1.1.
3) Finally, evaluated third equation: $Temp[1][1] - Temp[1-1][1-1], \ \&r > 0 \ \&\& \ and \ c > 0$ to get the final output value to be filled at Temp[1][1] is 21.
4) $$InputArray(4x4)\begin{bmatrix}21 & 3 & -17 & -14\\ 15 & -14 & -31 & -28\\ 11 & -21 & 24 & -6\\ -2 & 23 & -23 & 23\end{bmatrix} \qquad TempArray(5x5)\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 21 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\end{bmatrix}$$

For (1,2):

1) Firstly, performing the first condition: $Input[1-1][2-1] + Temp[1-1][2], \ \&r > 0$ gives us:

a) Temp[1][2] = 0
b) Input[0][1] = 3
c) Temp[0][1] = 0

The values are color coded above in the matrix and are computed (Yellow = Green + Red)and filled in the Temp array.

2) Nextly, we perform the second equation: $Temp[1][2] + Temp[1][2-1]$, $\&c > 0$ similar to the previous step and update the value from the step 1.1.
3) Finally, evaluated third equation: $Temp[1][2] - Temp[1-1][2-1]$, $\&r > 0$ && $and\ c > 0$ to get the final output value to be filled at Temp[1][2] is 24.
4) $InputArray(4x4)\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$ $TempArray(5x5))\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 21 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

In this way we finally fill all the values in the Temp 2-D array, where the required summation of all the rectangular elements of the 2-D Matrix is colored in yellow.

$TempArray(5x5))\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 21 & 24 & 7 & -7 \\ 0 & 36 & 25 & -23 & -65 \\ 0 & 47 & 15 & -9 & -57 \\ 0 & 45 & 36 & -11 & -36 \end{bmatrix}$

2. The size of the Temp[r+1][c+1] corresponds to O({*m*+*1*}{*n*+*1*}) = O({*mn* + *m* + *n* + *1*}) which is of the complexity O(*mn*). Now we are using the same approach as discussed in the previous context for finding the maximum sum sub-rectangle in an Array.

$$SumOfRectangle = Temp[1][1]$$

$$SumOfRectangle = \begin{cases} SumOfRectangle - Temp[r-1][c]\ continue, & r > 0 \\ SumOfRectangle - Temp[r][c-1]\ continue, & c > 0 \\ SumOfRectangle - Temp[r-1][c-1]\ continue, & r > 0\ and\ c > 0 \end{cases}$$

We follow with assigning the initial values for "SumOfRectangle" to Temp[1][1] and iterate over all of the possibilities and compare the SumOfRectangle with "globalMaxSum" to get the Max Sum value across the entire 2-D array, thus we get elements back as like in the Input Array using the above reverse logic applied earlier on the Input array to obtain the Temp Array and update the start and end positions of the rectangular elements.

$TempArray(5x5)$

0	0	0	0	0
0	21	24	7	−7
0	36	25	−23	−65
0	47	15	−9	−57
0	45	36	−11	−36

For the last rectangular value, we find it consider by removing the summation of all the colored regions yellow and red (36, -65) and which are intersecting (25) and have to add the values with the green element(-36) to give out as 18 as this is less than the global maximum of 47. Thus after exhausting all the options, we get the result as 47 with (1, 1, 3, 1) as it's ordinal positions for the rectangle.

Correctness of the Algorithm: Similar to the brute force Matrix approach we find all possible sub-Matrices of the input matrix in finding the left, right, top and bottom start in four loops but instead of two loops for comparing, we store the max sum in a new matrix, thereby increasing space but making it efficient. This proves the correctness of the algorithm

Asymptotic Analysis:

1. Time Complexity:

With the Dynamic Programming approach, we are iterating over the extracted 2-D array of sum of all the possible combinations into Temp and we are iterating to find the Optimal solution in the Temp on the rows and columns of the 2-D array elements making the Time Complexity $\theta\ (n^4)$.

2. Space Complexity:

In the above-mentioned algorithm, we are using an additional space for storing the cumulative sum of the elements of the Input array, when combined to form $O\ (mn)$.

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task5.java
Enter size of input Array
4 4
21 3 -17 -14
15 -14 -31 -28
11 -21 24 -6
-2 23 -23 23
1 1 3 1 47
```

Case 2:

```
C:\Users\Public\Manish\AOA Test>java Task5.java
Enter size of input Array
4 4
0 5 -11 -61
-41 -88 -24 -65
53 -18 29 -37
-38 52 0 5
3 1 4 3 78
```

Case 3:

```
C:\Users\Public\Manish\AOA Test>java Task5.java
Enter size of input Array
4 4
-1 -1 -1 -1
0 0 0 0
-1 -1 -1 -1
0 0 0 0
2 1 2 1 0
```

Algorithm 6: Using Dynamic Programming approach, we try to evaluate the summation of the elements in the rectangular elements from the 2-D array in θ (n^3) with an extra space of O(mn)

In the previous algorithm, Table1.5, "MaxBFMatSum" is computed each-and-every time instead if we rely on the previously computed and stored value, we can exclude the exponential complexity to make the algorithm run faster. Here, we are using a matrix of the same size as *(m+1 x n+1)* to store the computed values in the previous iterations.

In this approach we are trying to use an extra 2-D array space with a size *(m+1 x n+1)* for storing the summation of the possible rectangular element values from the input 2-D array of size *m x n.* We are adding one row & col (typically first row and column) to the temp matrix filled with Zeros '0' in-order-to incorporate the same logic to the edge cases dealing with the first row and first column. We begin with filling the values into the temp 2-D array from index (1,1) and let us work on an example.

Algorithm:

```
maxContDPn6Matrix(arr, rows, columns):
      rowStartIndex = 0, rowEndIndex = 0, colStartIndex = 0, colEndIndex = 0
      negativeRowIndex = 0, negativeColIndex = 0
      globalMinDPn3 = Integer.MIN_VALUE // −2^31
      globalMaxDPn3 = -1
      new 2D array initAdd[rows + 1][columns]
      for (rowLoop = 0; rowLoop < rows; rowLoop++):
        for (colLoop = 0; colLoop < columns; colLoop++):
          initAdd[rowLoop+1][colLoop]=initAdd[rowLoop][colLoop]+ arr[rowLoop][colLoop]
      for (rowStart = 0; rowStart < rows; rowStart++:
        for (row = rowStart; row < rows; row++):
            localSum = 0, localColumnStart = 0
            for (col = 0; col < columns; col++):
              localSum += initAdd[row + 1][col] - initAdd[rowStart][col];
              if (localSum < 0):
                  if (globalMinDPn3 < localSum):
                        globalMinDPn3=localSum, negativeRowIndex=row
                        negativeColIndex = col
                  localSum = 0, localColumnStart = col + 1
              else if (globalMaxDPn3 < localSum):
                  globalMaxDPn3 = localSum, rowStartIndex = rowStart, rowEndIndex = row
                  colStartIndex = localColumnStart, colEndIndex = col;
      if (globalMaxDPn3 == -1):
            globalMaxDPn3 = globalMinDPn3,
            rowStartIndex = rowEndIndex = negativeRowIndex,
            colStartIndex = colEndIndex = negativeColIndex

      return rowStartIndex, colStartIndex, rowEndIndex, colEndIndex, globalMaxDPn3
```

The problem statement can be solved with the Dynamic Programming strategy, where we are going to use an extra 2-D array Temp[r+1][c] of size (*m+1 x n*) to store the values. Below is the function used to fill the values into the Temp.

$$Temp[i][j] = \begin{cases} 0, & i = 0 \\ Input[i-1][j], & i > 0 \end{cases}$$ for all *(i, j)* ranging from *(0,0)* to *(r+1, c) in Temp*

$$InputArray(4x4)\begin{bmatrix} 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix} \qquad TempArray(5x4)\begin{bmatrix} 0 & 0 & 0 & 0 \\ 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

1. Now, we iterate over the rows and columns with respect to all the rows present in the Temp array for finding the maximum sum of the sub rectangle formed by the spanned with different rows forming as *(rowStart, row, col).*

   For the Iteration: *(0, 0, 0)*

   a. We calculate the sum using the above function defined for “localSum” as the cumulative sum over all the column elements with respect to the “rowStart” and “row”. Let us understand more about it using the example.

$$TempArray(5x4)\begin{bmatrix} 0 & 0 & 0 & 0 \\ 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

   We have represented two colored regions with yellow and red for calculating the difference of the sum of the elements. This evaluates localSum to -7 and we compare with the globalMinSum (initially min value of Integer and globalMaxSum to -1) is less than the assigned value, it is assigned as -7 . We perform the same to the next rows following from 2 to row+1.

$$TempArray(5x4)\begin{bmatrix} 0 & 0 & 0 & 0 \\ 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

   Now, when the row is iterated over all the possible row values, we are trying to get the maximum sum of the row elements w.r.t to the rowStart. By doing the same we are trying to find the highest value from

the rows and its positional indices. Here, we get the "localSum" to be as -58 and it is less than the globalMinSum we don't change the value and we repeat the process until the last row.

2. We proceed the same with changing the rowStart Values from 0 - rows+1. Performing the check between the rows to check if they correspond to any of the highest sum between them.

$$TempArray(5x4)\begin{bmatrix} 0 & 0 & 0 & 0 \\ 21 & 3 & -17 & -14 \\ 15 & -14 & -31 & -28 \\ 11 & -21 & 24 & -6 \\ -2 & 23 & -23 & 23 \end{bmatrix}$$

With the last iteration we happen to obtain the Maximum Sum of the 2-D Array from "globalMaxSum" or "globalMinSum" when the value is -1 or else, we pick the globalMaxSum value along with its respective index positional values.

Correctness of the Algorithm: Similar to the brute force Matrix approach we find all possible sub-Matrices of the input matrix in finding the left, right, top and bottom start in four loops but instead of two loops for comparing, we store the max sum in the array increasing space but making it efficient. This proves the correctness of the algorithm

Asymptotic Analysis:

1. Time Complexity:

With the Dynamic Programming approach, we are iterating over the extracted 2-D array of sum of all the possible combinations into Temp but unlike the previous approach, we store the previously computed max values into an additional row. This reduces an iteration in our iterative approach to find the Optimal solution in the new Array on the rows and columns of the 2-D array elements, thus making the Time Complexity $\boldsymbol{\theta\ (n^3)}$.

2. Space Complexity:

In the above-mentioned algorithm, we are using an additional space for storing the cumulative sum of the elements of the Input array, when combined to form $\boldsymbol{O\ (mn)}$.

Output:

Case1:

```
C:\Users\Public\Manish\AOA Test>java Task6.java
Enter size of input Array
4 4
21 3 -17 -14
15 -14 -31 -28
11 -21 24 -6
-2 23 -23 23
1 1 3 1 47
```

Case 2:

```
C:\Users\Public\Manish\AOA Test>java Task6.java
Enter size of input Array
4 4
0 5 -11 -61
-41 -88 -24 -65
53 -18 29 -37
-38 52 0 5
3 1 4 3 78
```

Case 3:

```
C:\Users\Public\Manish\AOA Test>java Task6.java
Enter size of input Array
4 4
-1 -1 -1 -1
0 0 0 0
-1 -1 -1 -1
0 0 0 0
2 1 2 1 0
```

Experimental Comparative Study:

Based on the experience obtained on understanding how different strategies work to re-produce the same output result with a trad-off between Space and Time complexities, we have extended the range of inputs to understand the algorithms and their behavior w.r.t to varying inputs ranging in thousands. Below are some of the screenshots based on the survey performed.

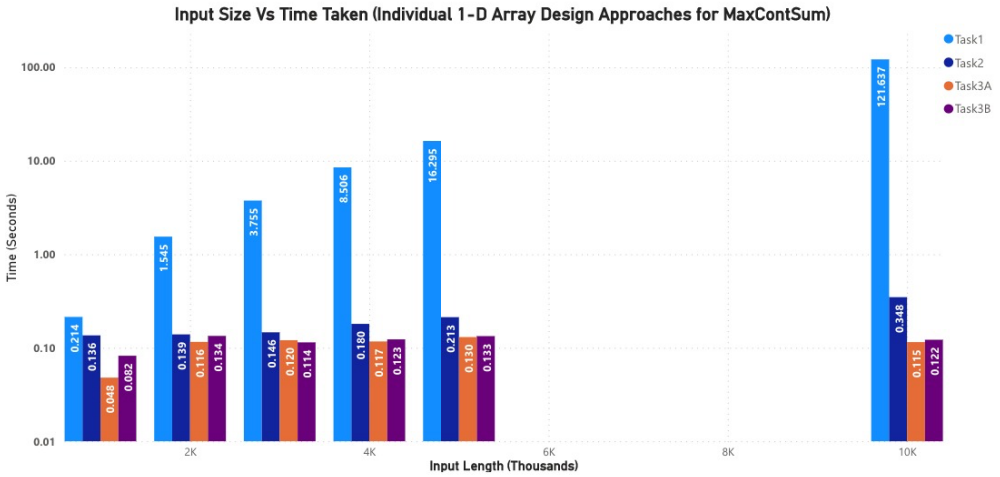
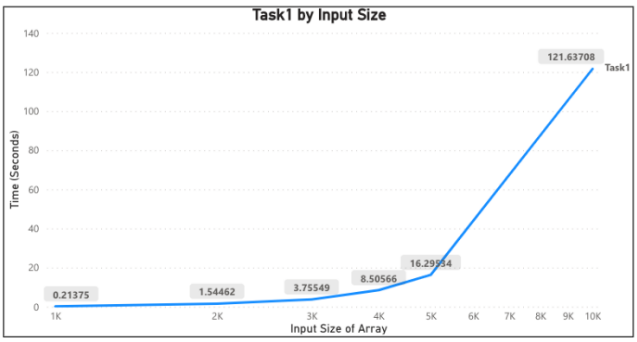
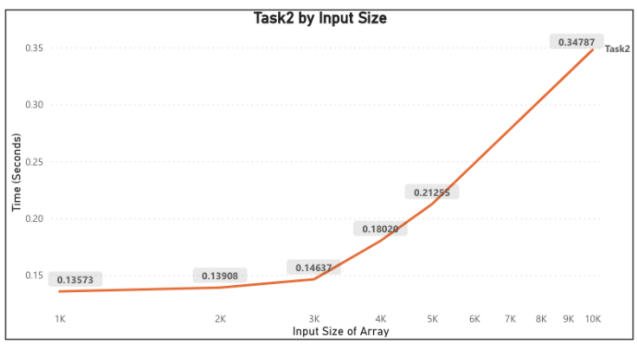


Figure 1.1

Figure 1.1 shows the comparative analysis of Three Algorithms with varying input values. It depicts that if we increase the input size, time taken to the Algorithm is increasing based on the Asymptotic expression of the algorithm. It means, for the Algorithm that has complexity in the order of n^2 takes large amount when compared with n. For Example, with the size 10,000 (randomly generated and evenly distributed across both sides of Zero, and same inputs for both the algorithms) the time taken for the Brute force Task-1 with the Complexity order n^3 took around 120 seconds whereas the Dynamic Programming Task-3A/B roughly took around 0.1 seconds, Task-1 has increased 56,000% of its original time and Task-3A/B has increased 140% of its original value. Also, a detailed analysis w.r.t individual algorithms is provided below for better understanding in the rapid change of the proportions of time taken for processing times.





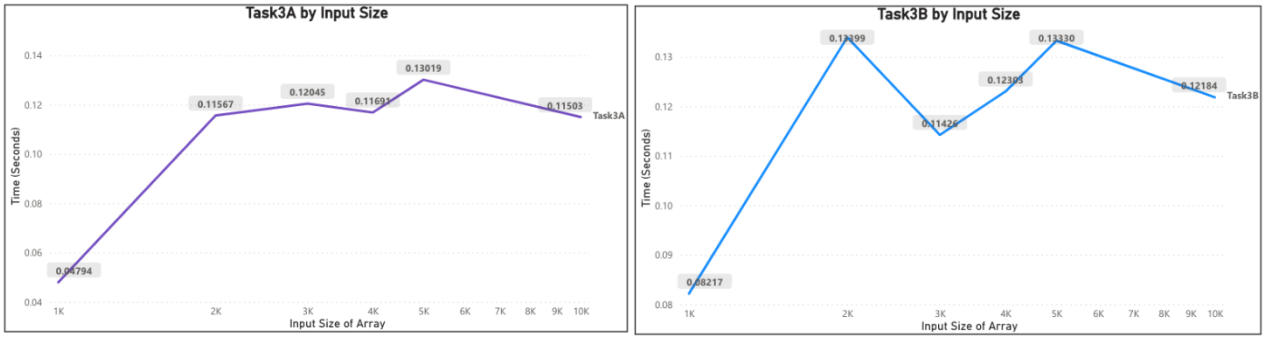


Figure 1.2

We see that the Higher Order Complexity Algorithms like Task-1 and Task-2 has a clear uphill in the time taken for computing with the increase in the input size. Whereas, for the Task-3A/B we experience a little ups and downs due to several parameters like the CPU being utilized by other background instructions at that time and also we see some incline in these graphs too.

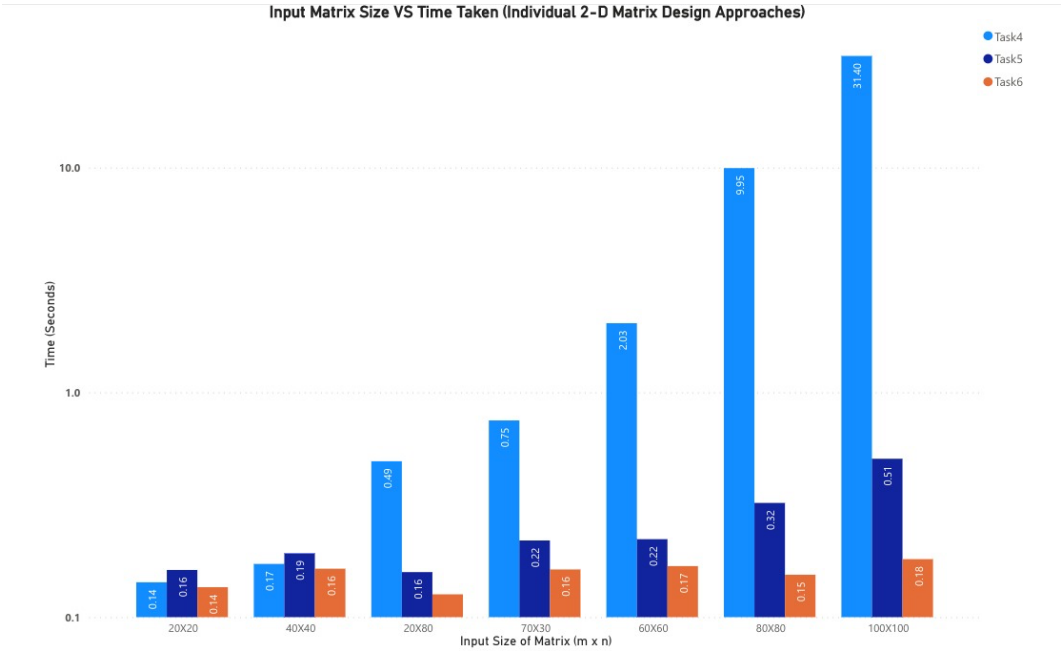


Figure 1.3

From the above figure, we clearly depict the same analysis, with the increase in the input size (randomly generated and evenly distributed across both sides of Zero, and same inputs for both the algorithms) of the rectangular matrix *(m x n)* we also observe there is a direct proportional change in the time required for obtaining the result. Less complexity algorithms, like Dynamic Programming, are performing well even with the matrix range of order of Hundreds with a bare increase of about 30% and the higher order complexity algorithms, like the Brute force, increased by 22,000% which a huge jump when compared with the CPU time.

Conclusion:

With the exposure to the theoretical aspects of the Design strategies for solving the problems, it is a good learning experience to obtain the practical exposure and co-relate the real-world situations and to deal it with the trade-offs that contribute to Optimal Solutions.

Based on the statistical analysis and the correctness provided we conclude that Higher Order Complexity Algorithms like Naïve/ Brute force Algorithms provide solutions considering all the possible outcomes which might not be a satisfactory solution at times. By observing the repetitive patterns and overlapping sub-structure we can reduce the overhead to re-compute the same logic multiple times using the Dynamic Programming Paradigms, this gives us an understanding to approach to the scenario in a more suitable and reliable approach. Statistics show that with the correct utilization of the computing capabilities we can deal highly complicated real-world scenarios can be solved considering both time and space as a constraint.

Approaches derived for the Naïve solutions can framed easily with the direct understanding to the situation, the very first solution to begin with. The challenges one may face is often overlooked with the short-sightedness of less inputs or limiting to an easy input. This puts everything on-line both time and space. It can be overcome if one may look in to refine the process to work effectively, by finding ways to reduce the repetitive computation of same problem. We understood a way to refine a Naïve/ Brute force process into a industry standard algorithm by finding patterns one can depict and implement using Iterative and recursive strategies gives us much more effective results for even exponential increase of input sizes.