COT5405 Analysis of Algorithms: Programming Project

Rishabh Jaiswal (2109-5276)
Shreeya Deshpande (6523-9982)

April 6, 2022

Rishabh Jaiswal worked primarily on task 1, 2 and 3 while Shreeya Deshpande worked on task 4, 5 and 6. Both teammates contributed equally to the project, code and the report.

1 Problem Statement

Given a piece of land which can be divided into m x n cells. Each cell has certain air quality index *M[i,j]* (where i = 1,2,3,...,m and j = 1,2,3,...,n) which is recorded as matrix M of dimensions m x n. Our task is to find the a rectangular area on the land which has the maximum total air quality index.
To solve the above problem, we are breaking it into two parts discussed below:

- Problem 1: Consider a row in the matrix M discussed above. We have to find the continuous part of the row which has the maximum sum of values.
- Problem 2: Consider the whole matrix M and find a rectangular section which has the maximum sum of values.

Constraints:
For problem 1, $1 \leq n \leq 2^{31}$ $\forall i$ $-2^{15} \leq A[i] \leq 2^{15}$
For problem 2, $1 \leq m, n \leq 2^{15}$ $\forall i$ $-2^{15} \leq Cells[i][j] \leq 2^{15}$
where m,n are the number of rows and columns respectively.
Also, the value of m is in order$O(n)$.

2 Design and Analysis of Algorithms

2.1 ALG 1

The brute force algorithm to solve problem is described below:

Analysis:
In the above algorithm, we take combination of every possible indices i, j such that i≤j (line 5 and 7). For each such possible combination, we calculate the sum of the elements that lie in the sub-array starting at index i and ending at index j (lines 8-10).

For every i, loop 6-16 runs i times. And for every j, loop 9-11 runs $j - 1$ times. A range of [1, N] has $\binom{N}{2}$ /2 possible pairs of indices. Hence, finding each possible pair takes $O(n^2)$ time. Further, it takes $O(n)$ time to calculate the sum of values in a sub-array. Consequently, the algorithm has a time complexity of $O(n^3)$.

We are using only 4 variables to store our temporary as well as final result. Hence, the space complexity is constant i.e. $O(1)$.

Proof of correctness:
Let there be a non empty array A of size N. The algorithm follows the below loop invariants:

- Loop 1 (lines 4 to 16) maintains the loop invariant:

  At the start of each iteration, we have the maximum sum of sub-arrays starting from k such that $0 < k \leq i$.

Algorithm 1: Brute force O(n^3) Algorithm

```
1  Input: Size of the 1-dimensional array N, array of integers Cells
2  Output: Resulting sub-array (with maximum sum of values) represented as a set of values - starting
     index start, end index end, and sum of the sub-array maxSum.
3  Initialize: maxSum = −∞, start = 1, end = 1
4  begin
5      for i = 1 to N
6      begin
7          for j = i to N
8          Let currentSum = 0
9          begin
10             for k = i to j
11                 currentSum = currentSum + Cells[k]
12         begin
13             if currentSum > maxSum
14                 maxSum = currentSum
15                 start = i
16                 end = j
17 return start, end, maxSum
```

Initialization: Before the first iteration, we have not visited any elements of the array. The minimum element in the sub-array can be $-\infty$. Hence, the maximum possible sum in the worst case scenario (where all the elements are $-\infty$) can be $-\infty$ with a single element as the sub-array.

Maintenance: After the first iteration, we have checked the sum of elements of every possible sub-array starting from the first index. This is justified further from second invariant explained below. Thus, we know the maximum possible sum of elements and consequently the sub-array with that sum (since we have the end index of the sub-array) starting from the first element. Similarly, after 2^{nd} iteration, we will have checked the sum of elements of every possible sub-array starting from second index and compared it with the maximum sum obtained from the previous iteration. Hence, we have the maximum sum of sub-arrays starting at or before index 2. In this way, the invariant continues to follow until $i = N$.

Termination: In the last iteration of the loop, we have the sub-array with the maximum sum, starting at k such that $0 < k \leq N$. This implies that we have the sub-array with the maximum sum among all the sub-arrays that start from indices 0 to N. Consequently, we have the solution to our problem.

- Loop 2 (lines 6 to 16) maintains the below loop invariant:
  For every starting index i we have the maximum sum sub-array $A[i:k]$ where $i \leq k \leq j$.

  Initialization: Before the first iteration, we have not visited any elements of the array. As explained for loop 1, the maximum possible sum in the worst case scenario (where all the elements are $-\infty$) can be $-\infty$ with a single element as the sub-array.

  Maintenance: Let $i = 1$ and $1 \leq j \leq N$. Also, we know that $maxSum = -\infty$ (in this scenario). After the first iteration, we have visited $A[i:j] = A[1:1] = A[1]$. Since the minimum value of $A[i]$ can be $-\infty$, we can say that we have the maximum sum sub-array. Similarly, after the second loop we have visited $A[1]$ and $A[1:2]$. If both $A[1]$ and $A[2]$ are equal to $-\infty$, we still have the maximum sum sub-array as $A[1]$. If $A[2] > A[1] > -\infty$, $A[1] + A[1] > A[1]$. Hence, we still have the maximum sum sub-array as $A[1:2]$. In this way, the invariant is maintained for all $j|i \leq jN$.

  Termination: After the loop terminates, $j = N$ i.e. we have visited all the elements from index i to N and compared there sums. Hence, it is guaranteed that we have the maximum sum sub-array among all the sub-arrays starting from or before i. This also explains the invariant followed by loop 1.

Combining the two invariants, we observe that after the execution of both the loops, we will have a sub-array $A[s:e]$ such that $1 \leq s \leq N$ and $s \leq e \leq N$ which has the maximum sum among all the other possible sub-arrays. Hence, the algorithm is correct.

2.2 ALG 2

The $O(n^2)$ time dynamic programming algorithm to solve problem is described below:

Algorithm 2: DP O(n^2) Algorithm

Input: Size of the 1-dimensional array N, array of integers $Cells$
Output: Resulting sub-array (with maximum sum of values) represented as a set of values - starting index $start$, end index end, and sum of the sub-array $maxSum$.
Initialize: $dp = [][]$ with all values as 0
begin
    for $i = 1$ **to** N
    begin
        for $j = i$ **to** N
            $dp[i][j] = dp[i][j-1] + Cells[j]$
Initialize: $maxSum = -\infty$, $start = 1$, $end = 1$
begin
    for $i = 1$ **to** N
    begin
        for $j = 1$ **to** N
            if $maxSum < dp[i][j]$
                $maxSum = dp[i][j]$
                $start = i, end = j$
return $start$, end, $maxSum$

Analysis:
In the above algorithm, we calculate the sum of all the possible sub-arrays and store it in a result matrix $dp[i][j]$ such that i is the start index of the sub-array and j is the end index. After that, we traverse the $dp[][]$ array to get the maximum value along with the start and end indices of the maximum sum sub-array. To fill the $dp[][]$ array requires n^2 time as i iterates from 1 to N, and j iterates from i to N. Then, it requires n^2 time again to search for the optimal solution. Hence, total time complexity is of the order $O(n^2)$. Since a $dp[][]$ array of size NxN is maintained, the space complexity is also of the order of $O(n^2)$.

Proof of correctness:
The above problem can be broken into an optimal sub-problem which is:

$$dp[i][j] = dp[i][j-1] + A[j]$$

This can be proved by induction as below:

- Let $i = 1, j = 1$. $dp[1][1] = A[1:1] = A[1]$
- If $i = 1, j = 2$, $dp[1][2] = dp[1][1] + A[2] = A[1] + A[2]$
- For $i = 1, j = k$ $(k < N)$, $dp[1][k] = dp[1][k-1] + A[k] = dp[1]dp[k-1] + A[k-1] + A[k]$, and so on.
- For $i = 1, j = k+1$ $(k+1 < N)$, $dp[1][k+1] = dp[1][k] + A[k+1]$
- Hence, it is clear that the above pattern follows our sub-problem recursive formulation.

Therefore, the algorithm is correct.

2.3 ALG 3

The $O(n)$ time dynamic programming algorithm to solve problem is described below:

Analysis:
In the above algorithm, we are keeping track of the maximum sum possible for a sub-array using the variable $maxSum$ and the current running sum using the variable $currentSum$. In every iteration, we add the element to the $currentSum$ and keep it in $maxSum$ if it is the maximum sum we have encountered till now. If $currentSum$

Algorithm 3: DP $O(n)$ Algorithm

```
1  Input: Size of the 1-dimensional array N, array of integers Cells
2  Output: Resulting sub-array (with maximum sum of values) represented as a set of values - starting
     index start, end index end, and sum of the sub-array maxSum.
3  Initialize: maxSum = −∞, start = 1, end = 1, dp = Cells[1], currentSum = 0
4  begin
5      for i = 1 to N
6          ifcurrentSum < 0
7              currentSum = Cells[i]
8              if currentSum > dp
9                  dp = currentSum
10                 start = i, end = i
11         else
12             currentSum = currentSum + Cells[i]
13             if currentSum > dp
14                 dp = currentSum
15                 end = i
16 return start, end, maxSum
```

is negative, it suggests that this sub-array cannot be extended as adding any value to it will only decrease the value of overall sum. In such a situation, old sub-array is discarded and the current element becomes the new sub-array. This algorithm is also called *Kadane's Algorithm*. There is only one loop used in the algorithm, which means we are visiting each element in the array only once. This implies the time complexity of the algorithm to be $O(n)$.

Since only 4 extra variables are being used to store values of *currentSum*, *maxSum*, *start*, and *end*, the space complexity of the algorithm is constant i.e. $O(1)$.

Proof of correctness:
Given an array A we need to find the sub-array with maximum sum of elements. This can be solved as using below recursive formulation:

$$maxSum[i] = MAX(maxSum[i-1] + A[i], A[i])$$

where $maxSum[i]$ denotes the maximum local sum obtained for sub-array ending at i and $i \geq 1$.
Thus, for any index i, our task is to find the solution of this sub-problem. To prove that this recursion formulation is correct, we will use induction.

- Initially, for $i = 1$, $maxSum[1] = A[1]$.
- Now, let $i = 2$. $maxSum[2] = MAX(maxSum[1] + A[2], A[2]) = MAX(A[1] + A[2], A[1]) = A[1]$ if $A[2]$ is negative, or, $A[1] + A[2]$ if A[2] is positive.
- Considering that $A[2]$ is negative. Now, $maxSum[3] = MAX(maxSum[2] + A[3], A[3]) = MAX(A[2] + A[3], A[3]) = A[3]$ since $A[2]$ is negative. Thus, maximum sum from 1 to 3 will be A[3].
- Similarly, for $i = k$, $maxSum[k] = MAX(maxSum[k-1] + A[k], A[k])$.
- Now, let $i = k+1$. If $maxSum[k]$ is positive and $A[k+1]$ is also positive, $maxSum[k+1] = maxSum[k] + A[k+1]$. If $maxSum[k]$ is positive and $A[k+1]$ is negative, the maximum sum sub-array will end at k. Thus, new local maximum sum sub-array will only contain a single element $A[k+1]$ with $maxSum[k+1] = A[k+1]$. This implies, $maxSum[k+1] = MAX(maxSum[k] + A[k+1], A[k+1])$.
- Hence, the recursive formulation is proved.

Therefore, the algorithm is correct.

2.4 ALG 4

Brute force O(n^6) solution to solve the problem is described below:

Algorithm 4: Brute force O(n^6) Algorithm

```
1  Input: Size of the 2-dimensional array row * col , array of integers cells represented entered row-wise
2  Output: Resulting sub-square of a matrix (with maximum sum of values) represented as a set of
     corner values – left, right, top, bottom, and sum of the square maxSum.
3  Initialize: left_end = 0, right_end = 0, top_end = 0, bottom_end = 0, maxSum = −∞;
4  begin
5      for i = 0 to row
6      begin
7          for j = 0 to col
8          begin
9              for k = i to col
10             begin
11                 for l = j to col
12                     Let sum = 0
13         begin
14             for m = i to i
15             begin
16                 for n = j to l
17                     currentsum+ = matrix[m][n]
18                 begin
19                     if (currentSum > maxSum)
20                         update the corners
21 return left, right, top, bottom, maxSum
```

Analysis:
In the above algorithm, we take combination of every possible corners of the submatrix i, j, k, l such that i≤j (line 5, 7, 9 and 11). For each such possible combination, we calculate the sum of all the elements that lie in the sub-matrix (line 17).

For every i, loop 5-20 runs i times. And for every j, loop 7-20 runs j times. Similarly for k and l. This chooses all possible combinations of indices of corner. There are two loops m and n to calculate sum within the selected submatrix, Hence, finding each possible pair takes $O(n^4)$ time and summation takes $O(n^4)$. Consequently, the algorithm has a time complexity of $O(n^6)$.

There is no extra matrix/ array used to store temporary variables. Hence, the space complexity is constant i.e. $O(1)$.

Proof of correctness:
Let there be a non empty matrix A of size $N * M$. The algorithm follows the below loop invariants:

Outer Loop (lines 4 to 20) maintains the loop invariant:

At the start of each iteration, we have the maximum sum of sub-matrix for each sub-matrix with corners as i, j, k and l

Initialization: Before the first iteration, we have not visited any elements of the matrix. The minimum element in the sub-matrix can be $-\infty$. Hence, the maximum possible sum in the worst case scenario (where all the elements are $-\infty$) can be $-\infty$ with a single element as the sub-matrix.

Maintenance: After the first iteration, we have checked the sum of elements of every possible row in the sub-matrix starting from the first index. Thus, we know the maximum possible sum of elements and consequently the sub-matrix with that sum (since we have the end indexes of the sub-matrix) starting from the first left and top elements. Similarly, after 2^{nd} and further iterations, we will have checked the sum of elements of every possible sub-matrix and compared it with the maximum sum obtained from the previous iterations. Hence, we have the maximum sum of sub-arrays. In this way, the invariant continues to follow until $i = N$.

Termination: In the last iteration of the loop, we have the sub-matrix with the maximum sum. This implies that we have the sub-matrix with the maximum sum among all the sub-arrays that start from top-left of the matrix to the bottom-right. Consequently, we have the solution to our problem.

Similarly considering and combining all the six loops, we observe that after the execution of all the loops, we will have a sub-matrix which has the maximum sum among all the other possible sub-arrays. Hence, the algorithm is correct.

2.5 ALG 5

O(n^4) with O(N) extra space solution to solve the problem is described below:

Algorithm 5: O(n^4) Algorithm with O(N) extra space

```
1  Input: Size of the 2-dimensional array row * col , array of integers cells represented entered row-wise
2  Output: Resulting sub-square of a matrix (with maximum sum of values) represented as a set of
     corner values – left, right, top, bottom, and sum of the square maxSum.
3  Initialize: left_end = 0, right_end = 0, top_end = 0, bottom_end = 0, maxSum = −∞, start = 0,
     end = −1, temp[row];
4  begin
5      for l = 0 to col
6      begin
7          for i = 0 to row
8              Let temp[i] = 0
9          begin
10             for r = l to col for i = 0 to row
11                 temp[i]+ = M[i][r]
12             begin
13                 sum = kadanealgorithmtofindsumofasquare
14                 if (currentSum > maxSum)
15                     maxSum = currentSum
16                 update the corners
17 return left, right, top, bottom, maxSum
```

Analysis:
In the above algorithm, we use Kadane's algorithm (Algorithm 3). Kadane's algorithm for a 1-D array is used in this algorithm. The temporary array created (line 8) is used as an input to the Kadane's algorithm. This takes O(N) extra space. We take combination of all elements in row and columns (line 5 and 7). For each combination, we use Kadane's algorithm (line 13).

For every l, loop 5-16 runs *col* times, for every i, loop 7-16 runs *row* times, and for every r, loop 9-16 runs *row* times. Similarly Kadane's algorithm takes O(N) time. Hence, the algorithm has a time complexity of $O(n^4)$. There is one 1-D temporary array used and hence, the space complexity is constant i.e. $O(N)$.

Proof of correctness:
Let there be a non empty matrix A of size $N * M$. The algorithm follows the below loop invariants:

Outer Loop (lines 4 to 16) maintains the loop invariant:

At the start of each iteration, we have the maximum sum of sub-matrix as $-\infty$.

Initialization: Before the first iteration, we have not visited any elements of the matrix. The minimum element in the sub-matrix can be $-\infty$. Hence, the maximum possible sum in the worst case scenario (where all the elements are $-\infty$) can be $-\infty$ with a single element as the sub-matrix.

Maintenance: After the first iteration, we have checked the sum of elements of every possible row in the first column for iterator l. Thus, we know the maximum possible sum of elements and consequently the sub-matrix with that sum starting from the A[0][0] for the first column, all row elements. Similarly, after 2^{nd} and

further iterations, we will have checked the sum of elements of every possible sub-matrix and compared it with the maximum sum obtained from the previous iterations. Hence, we have the maximum sum of sub-arrays. In this way, the invariant continues to follow until $l = col$.

Termination: In the last iteration of the loop, we have the sub-matrix with the maximum sum. This implies that we have the sub-matrix with the maximum sum among all the sub-arrays that start from top-left of the matrix to the bottom-right. Consequently, we have the solution to our problem.

Similarly considering and combining all the loops, we observe that after the execution of all the loops, we will have a sub-matrix which has the maximum sum among all the other possible sub-arrays. Hence, the algorithm is correct.

2.6 ALG 6

O(n^3) with O(n*m) extra space solution to solve the problem is described below:

Algorithm 6: O(n^3) Algorithm with O(row*col) extra space

```
1  Input: Size of the 2-dimensional array row * col , array of integers cells represented entered row-wise
2  Output: Resulting sub-square of a matrix (with maximum sum of values) represented as a set of
     corner values – left, right, top, bottom, and sum of the square maxSum.
3  Initialize: leftEnd = 0, rightEnd = 0, topEnd = 0, bottomEnd = 0, maxSum = −∞, rowStart = 0,
     rowEnd = 0, columnStart = 0, columnEnd = 0, cumulative_sum[m][n]
4  begin
5      for i = 0 to col
6      begin
7          for j = 0 to row
8              cumulative_sum[i + 1][j] = cumulative_sum[i][j] + matrix[i][j]
9  begin
10     for i = 0 to row
11     begin
12         for j = i to row
13         Reinitialize sum = 0, currCol = 0
14         begin
15             for k = 0 to col
16                 sum+ = cumulative_sum[i + 1][j] − cumulative_sum[i][j];
17             begin
18                 if sum < 0
19                 Reset all row-col variables else if sum > maxSum
20                 Set all row-col variables
21 return left, right, top, bottom, maxSum
```

Analysis:
In the above algorithm, we store cumulative sum of all elements till a point in matrix when traversed top to bottom and left to right. At any point when the sum drops below zero and sum starts becoming negative, we reset the corners of the matrix. The prefix cumulative sum needs two loops to traverse the entire array (lines 5 and 7) to find cumulative sum matrix(line 8). Once we have this matrix, reset the sum and indices whenever the sum drops to zero or beyond. With these loops, the algorithm has a time complexity of $O(n^3)$. We also use a temporary matrix to store cumulative sum of all elements till a point in matrix when traversed top to bottom and left to right. Hence, O(N*M) extra space is used.

Proof of correctness:
Let there be a non empty matrix A of size $N * M$. The algorithm follows the below loop invariant:

Outer Loop (lines 9 to 18) maintains the loop invariant:

At the start of each iteration, we have the maximum sum of sub-matrix as $-\infty$.

Input Size	Task1	Task2	Task3a	Task3b
1000	113.812	15.296	11.44	11.083
2000	835.984	21.835	11.958	11.235
3000	2821.483	33.172	11.212	11.21
4000	6703.725	41.936	11.177	10.981
5000	13180.988	51.517	11.405	11.321

Table 1: Average Time Taken by Different Algorithms for Problem 1

Input Size	Task4	Task5	Task6
1000	7.343	0.396	0.321
2000	307.503	2.548	1.834
3000	3282.107	5.675	5.622
4000	6642.157	8.521	6.585
5000	38052.035	12.6	68.748

Table 2: Average Time Taken by Different Algorithms for Problem 2

Initialization: Before the first iteration, we have not visited any elements of the matrix. The minimum element in the sub-matrix can be $-\infty$. Hence, the maximum possible sum in the worst case scenario (where all the elements are $-\infty$) can be $-\infty$ with a single element as the sub-matrix.

Maintenance: After the first iteration, we have checked the sum of elements $i = 0$ and $0 < j < M$. Thus, we know the maximum possible sum of elements and consequently the sub-matrix with that sum starting from the A[0][0] for the first row, all column elements. Similarly, after 2^{nd} and further iterations, we will have checked the sum of elements of every possible sub-matrix and compared it with the maximum sum obtained from the previous iterations. Hence, we have the maximum sum of sub-arrays. In this way, the invariant continues to follow until $k = col$.

Termination: In the last iteration of the loop, we have the sub-matrix with the maximum sum. This implies that we have the sub-matrix with the maximum sum among all the sub-arrays that start from top-left of the matrix to the bottom-right. Consequently, we have the solution to our problem.

Similarly considering and combining all the loops, we observe that after the execution of all the loops, we will have a sub-matrix which has the maximum sum among all the other possible sub-arrays. Hence, the algorithm is correct.

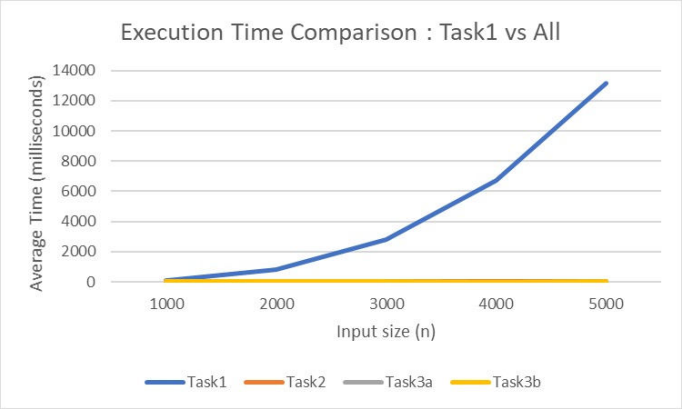
3 Experimental Comparative Study

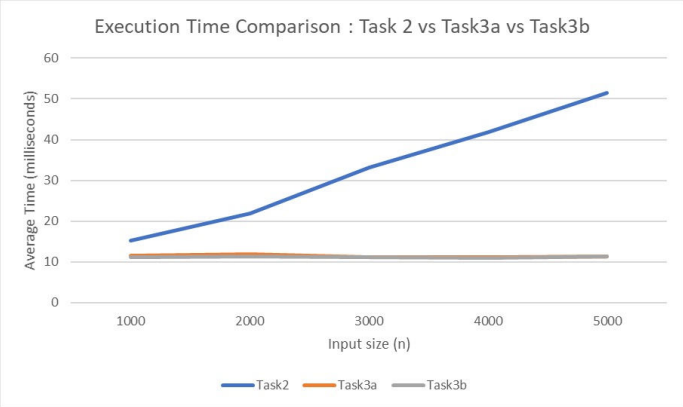
Graphs of the experimental data are shown on the next page.

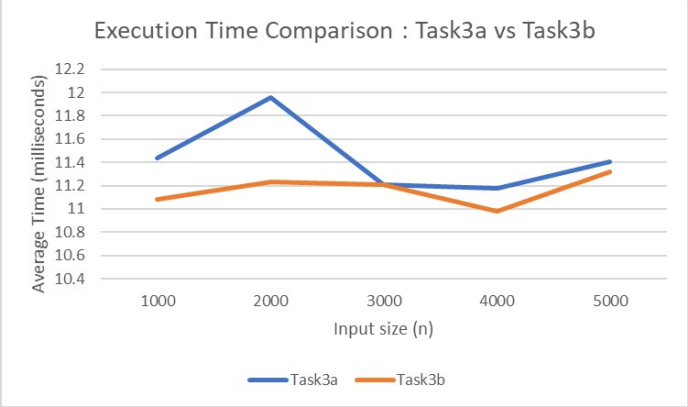
- The brute force method $O(n^3)$ took far more time in solving the problem than other methods.
- The calculated time complexity closely matched with the actual time taken for in the performed experiments.
- Iterative DP approach performs better than Recursive DP approach. This can be due to the memory stack used by recursion based implementations.
- For DP approach to solve Problem 1, the execution time increased steadily upto the N=2000 and then decreased beyond that along with slow rate.
- For DP approach to solve Problem 2, the execution time decreased till $N = 3000$ and increased with slow rate.

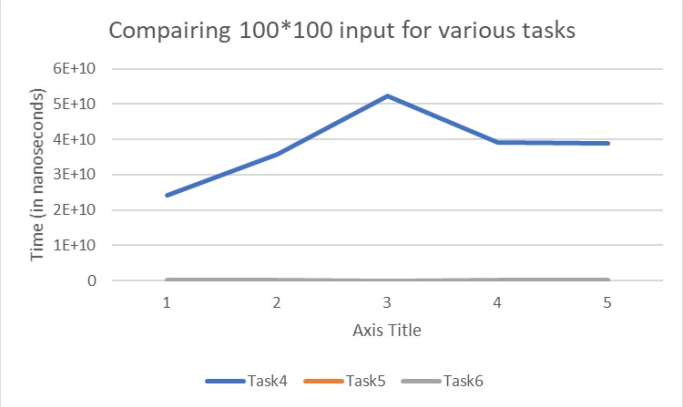
4 Conclusion

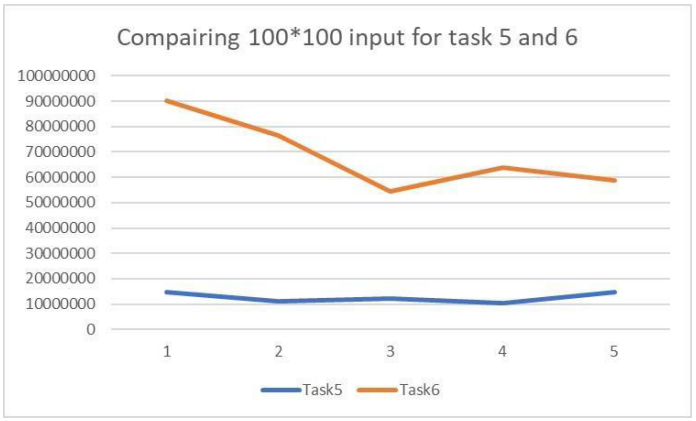
The algorithms with brute force approach take a very high amount of time than other better approaches. On comparing the results, even one degree lesser time complexity provides much faster results. Execution time increases exponentially with increasing complexity of the algorithm. In today's world, time is more valuable than storage space and hence it would a good practice to reduce time complexity even if it increases space complexity.

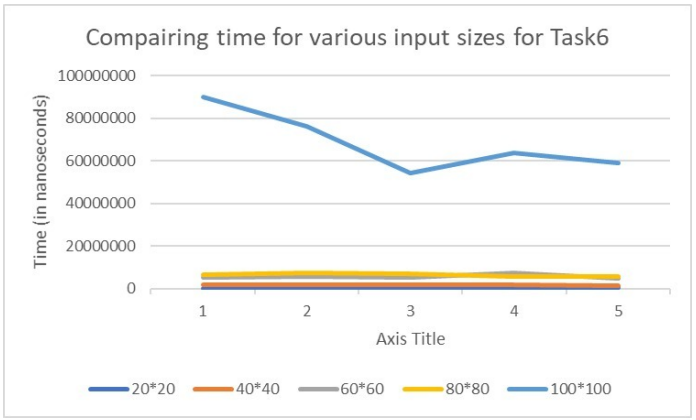

Execution Time Comparison : Task1 vs All
Average Time (milliseconds)
14000
12000
10000
8000
6000
4000
2000
0
1000
2000
3000
4000
5000
Input size (n)
Task1
Task2
Task3a
Task3b


Execution Time Comparison : Task 2 vs Task3a vs Task3b
Average Time (milliseconds)
60
50
40
30
20
10
0
1000
2000
3000
4000
5000
Input size (n)
Task2
Task3a
Task3b


Execution Time Comparison : Task3a vs Task3b
Average Time (milliseconds)
12.2
12
11.8
11.6
11.4
11.2
11
10.8
10.6
10.4
1000
2000
3000
4000
5000
Input size (n)
Task3a
Task3b


Compairing 100*100 input for various tasks
Time (in nanoseconds)
6E+10
5E+10
4E+10
3E+10
2E+10
1E+10
0
1
2
3
4
5
Axis Title
Task4
Task5
Task6


Compairing 100*100 input for task 5 and 6
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
1
2
3
4
5
Task5
Task6


Compairing time for various input sizes for Task6
Time (in nanoseconds)
100000000
80000000
60000000
40000000
20000000
0
1
2
3
4
5
Axis Title
20*20
40*40
60*60
80*80
100*100