Algorithms Programming Project

Raghav Gupta
Nishant Agarwal

Contributions

- In this Assignment, Raghav Gupta prepared the algorithms Task1,2,3 and 6 and implemented all the algorithms as asked in the assignment.
- Whereas Nishant Agarwal prepared the algorithms for Task 4 and 5 and has reported the time it takes for an implementation to run and compiled the data for comparative study.
- Preparation of the report was done as a collaborative effort.

Q) Humans are planning to build their second home on Mars. Suppose that the Mars rovers have
detected an area of $m \times n$ cells for us, and the air quality index of each cell is $M[i, j]$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. You are tasked to find a rectangle area to build a new base where
the total air quality index is maximized. For this, you shall solve the following two problems.
The solution to the first problem should help in designing an efficient solution for the second
problem. For the rest of this document assume that m is O(n).

- **Problem1 Given an array A of n integers (positive or negative), find a contiguous subarray whose sum is maximum.**
- **Problem2 Given a two-dimensional array M of size $m \times n$ consisting of integers (positive or negative), find a rectangle (two-dimensional sub-array) whose sum is maximum.**

-

2) Algorithm Design Tasks
Problem1- Given an array A of n integers (positive or negative), find a contiguous subarray whose sum is maximum.
Alg1 Design a $\Theta(n3)$ time brute force algorithm for solving Problem1

- This is a Naive approach, we will be using the two loops to compute all the possible subarrays and computer their sum in the next loop

Algo:
Maxsubarray (Array A)

```
1)  initialize max_sum
2)  Initialze sub_Array[1…..A.size()] be a new array
3)  for i = 1 to A.size()
4)          for  j = i to A.size()
5)              Intialize Sum = 0
6)              for k = i to k = j
7)                  sum += A[k]
8)                  if (sum>max_sum)
9)                      Max_sum = sum
10)                     left = i
11)                     right = j
12)  for int i =left to right
13)         print (A[i])
```

- In this approach as we can see that the approach uses three loops, the first or the outer loops try to find the leftmost element of the subarray that will give us the maximum sum. The second loop starts from where the first loop starts basically trying to find the rightmost element of the subarray. The third loop is simply keeping track of the sum generated from the sub-arrays and then comparing it with the maximum sum we encountered soo far to keep track of the maximum sum we have calculated from the sub-array. We also keep track of the left and the right element of the subarray in this loop.

<u>Correctness:</u>

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the array, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the array we encounter the values in the array and compare the encountered sum to the global maximum which we maintain in another variable.

Termination:

- When the traversal reaches the end of the array it has explored and evaluated all the sum of the subarrays and compares every sum to the global maximum value. In the end we would have done the comparison and updation we get the maximum sum subarray value of the array.

Alg2 Design a Θ(n2) time dynamic programming algorithm for solving Problem1

- **Better Approach** - In this approach, we will try to eliminate the innermost for loop which keeps track of the sum of the sub-array, instead we will update the sum from the second for loop itself.
  Algo:
  Maxsubarray (Array A)
  1. Initialze dp[1…..A.size()] be a new array
  2. dp [0] = A[0]
  3. for i = 1 to A.size()
  4. Initialize sum =0
  5. for j = i to A.size()
  6. sum += A[j]
  7. dp[i] = max(dp[i-1],sum)
  8. Return dp[A.length()-1]

- This approach performs much faster than the last approach because of the removal of one for loop which effectively reduces the time complexity from O(n^3) to O(n^2)

<u>Correctness:</u>

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the array, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the array we encounter the values in the array and compare the encountered sum to the global maximum which we maintain in a different array (let's say max_array). we choose the maximum out of the value in the array or the array value itself.

Termination:

- When the traversal reaches the end of the array it has explored and evaluated all the sum of the subarrays and compares every sum to the global maximum value which is stored in the latest explored index of the max-array . In the end we would have done comparison and updation we get the maximum sum subarray value of the array.

Alg3 Design a Θ(n) time dynamic programming algorithm for solving Problem1

- We will use a Dynamic programming array to save the maximum and track it using the array.

Algo:

max subarray(Array A)

1. Initialize maxSum = A[0]
2. Initialze dp[1…..A.size()] be a new array
3. dp[0] = A[0]
4. for (int i=1-> A.size())
5. maxSum = max(A[i],maxSum+A[i])
6. dp[i] = max(dp[i-1],maxSum)
7. Return dp[A.length()-1]

- In this approach, we are using array to save the last maximum value to solve the problem. We save the maximum value we found so far in the "dp" array and change it only when we encounter the max_sum to be greater than the maximum value we have encountered so far.
- This approach boils down to the time complexity of the entire approach to O(n).

Correctness:

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the array, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the array we encounter the values in the array and compare the encountered sum to the global maximum which we maintain in a different array (let's say max_array). we choose the maximum out of the value in the array or the array value itself.

Termination:

- When the traversal reaches the end of the array it has explored and evaluated all the sum of the subarrays and compares every sum to the global maximum value which is stored in the latest explored index of the max-array. In the end we would have done comparison and updation we get the maximum sum subarray value of the array.

Mathematical Recursive Formula:

- In this approach we make a choise between choosing the previous element or not. Upon this decision we make the decision of calculating the max_sum.
- For the choice when we do choose the previous element we will have to consider the maximum sum encountered till that element and add it to the element.

- For the choice when we don't choose the previous element we essentially decide that the current element is greater than the maximum sum encountered till the previous element.
- If we write the above mentioned statements in mathematical language it will translate to this:

$$dp[i] = A[i] + dp[i-1] \quad ,\text{if } dp[i-1] > 0$$
$$dp[i] = A[i] \quad ,\text{otherwise}$$

Problem2) - Given a two-dimensional array M of size m×n consisting of integers (positive or negative), find a rectangle (two-dimensional sub-array) whose sum is maximum.

Alg4 Design a Θ(n6) time brute force algorithm for solving Problem2
Algo:

```
vector<int> task4(vector<vector<int>> matrix, int n, int m)
{
        initialize res[0…m]
        initialize  maxSum to INT_MIN
        Initialize  leftr = 0, leftc = 0, rightr = 0, rightc = 0
        for (initialize i= 0to  n)
                for (initialize j = 0 to m)
                        for (initialize k = i to  n)
                                for (initialize l = j to m)
                                initalize  curSum = 0
                                        for (initialize row = i to  k)
                                                for (initialize col = j to  l)
                                                        curSum += matrix[row][col]
                                                        if (curSum > maxSum)
                                                                maxSum = curSum;
                                                                leftr = i
                                                                leftc = j
                                                                rightr = row
                                                                rightc = col
                                                        Push leftr + 1 into res array
                                                        Push leftc + 1 into res array
                                                    Push rightr + 1into res array
                                                    Push rightc + 1 into res array
                                                   Push maxSum into res array
        return res;
```

- This is the brute force approach in which we choose the left most element in the first two for loops and the right most element in the next two for loops.
- We use the final two loops to calculuate the sum and update the maximum sum if applicable.
- The lower bound of this algorithm can be Ω(row*col) and upper bound of this algorithm will have to run all the elements of the O((row*col)^6)

Correctness:

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the matrix, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the matrix we try to calculate all the possible sub-matrices from the left-most index. We follow the same procedure by saving the maximum sum value encountered so far in a variable,which we shall return in at termination.

Termination:

- When the traversal of the left index reaches the last element of the matrix, we would have explored all the sub-matrices of the matrix and calculated their corresponding sums. We get the maximum sum at the termination of the program.

Alg5 Design a Θ(n4) time algorithm for solving Problem2 using dynamic programming Alg3

```
int findMaxSumSubmatrix(2d array mat[][])
        if (mat.size() == 0)
        return 0
        int S[M+1][N+1];                // preprocess the matrix to fill `S`
        for (int i = 0 to  M )
                for (int j = 0 to  N)
                        if (i == 0 || j == 0)
                                S[i][j] = 0
                        else
                        S[i][j] = S[i-1][j] + S[i][j-1] - S[i-1][j-1] + mat[i-1][j-1]
        initialize  maxSum as INT_MIN
        initialize rowStart, rowEnd, colStart, colEnd;
        for (int i = 0; i < M; i++)
                for (int j = i; j < M; j++)
                        for (int m = 0; m < N; m++)
                                for (int n = m; n < N; n++)
                                initializr submatrix_sum = S[j+1][n+1] - S[j+1][m] - S[i][n+1] + S[i][m];
                                if (submatrix_sum > maxSum)
                                maxSum = submatrix_sum;
                                rowStart = i;
                                rowEnd = j;
                                colStart = m;
                                colEnd = n;
```

- In this approach we use pre-processed sum matrix containing the sums to help our brute force implementation.
- The approach follows dp approach to solve the maximum sum of the sub matrices.
- Pre computation of sum helps us in getting the time complex to O(n^4)

Correctness:

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the matrix, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the matrix we try to calculate all the possible sub-matrices from the left-most index. We follow the same procedure by saving the maximum sum value encountered so far in a variable,which we shall return in at termination.

Termination:

- When the traversal of the left index reaches the last element of the matrix, we would have explored all the sub-matrices of the matrix and calculated their corresponding sums. We get the maximum sum at the termination of the program.

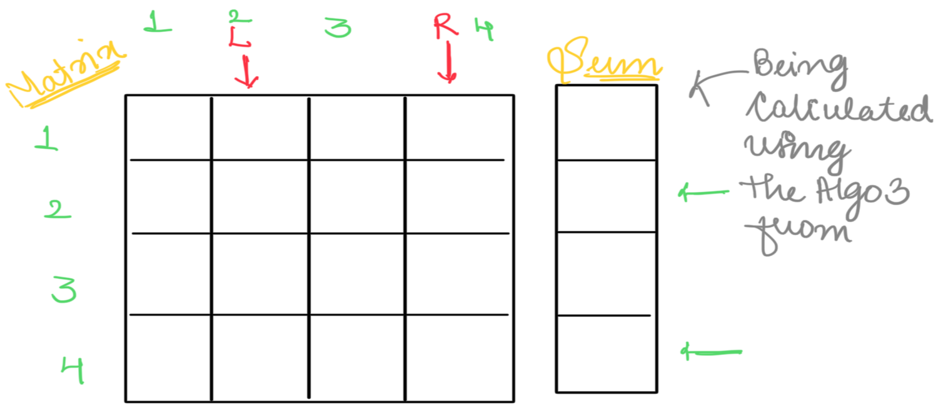
Alg6 Design a Θ(n3) time algorithm for solving Problem2 using dynamic programming Alg3

Algo:

```
task3b(Array nums)
        int T[0……nums.size()]
        T[0] = nums[0]
        int m = T[0]
        for (initialize i = 1 to nums.size())
                T[i] = max(nums[i], nums[i] + T[i - 1])
                if (T[i] > m)
                        m = T[i]
        return m;
```

```
task6(2d Array matrix[][], int row, int col)
        Int res[0…..col]
        initialize area  INT_MIN;
        for(initialize left = 0 to vol)
                int sum[row];
                for (initialize right = left to col)
                        for (initialize row = 0 to row)
                                sum[row] += matrix[r][row]
                        ar = max(ar, 3b(sum))
return ar
```

- This is the optimized approach of finding the maximum area in a 2d rectangle.
- In this approach we move the left and right pointers on the column which is being done by the first two for loops which will also determine from where our array will begin and end for every scope.
- We initialize the sum array with the size equals to the number of the rows, we save the maximum sum in each implementation using the algorithm we used in the task 3. The values given out by the task 3 algo will define the value of the rows we will choose.
- And we update the maximum value of the area we have encountered so far.



Correctness:

Initialization:

- When we first initialize the algorithm the only value that we have is the first element of the matrix, which will also be the maximum sum value for that scope.

Maintenance:

- As we traverse the matrix we try to calculate all the possible sub-matrices from the left-most index to the right most index. For every row and for every left and right index we save the maximum sum of that sub array in a different array and save the maximum subarray sum of that array which is essentially our maximum sum of the matrix.

Termination:

- When the traversal of the left index reaches the last element of the matrix, we would have explored all the sub-matrices of the matrix and calculated their corresponding sums. We get the maximum sum at the termination of the program.

3) Programming Tasks

Once you complete the algorithm design tasks, you should have an implementation for each of the following programming procedures:

Task1 Give an implementation of Alg1:

Code:

```
int maxSubArray(vector<int>& A) {
        int max_sum=INT_MIN,left=0,right=0;
        for (int i=0;i<A.size();i++){
        for(int j=i;j<A.size();j++){
        int sum = 0;
        for(int k=i;k<=j;k++){
                sum += A[k];
                if (sum>max_sum)
                {   max_sum = sum;
                left =i;
                right = j;}

                }
        }
        }
        cout<< left+1 << " " << right+1<< "\n";
        return max_sum;
        }
```

Time complexity: O(n^3)
Space Complexity: O(1)

Task2 Give an implementation of Alg2.

Code:
```
int maxSubArray(vector<int>& nums) {
        int maxSum = nums[0],left =0,right=0;
        for (int i = 0; i < nums.size() ; i++){
        int sum = 0;
                for (int j = i; j < nums.size(); j++){
                sum += nums[j];
                if(maxSum < sum)
                {
                maxSum = sum;
                left =i;
                right =j;
                }
        }
        }
        cout << left+1<< " " <<right+1 << "\n";
        return maxSum;
        }
```

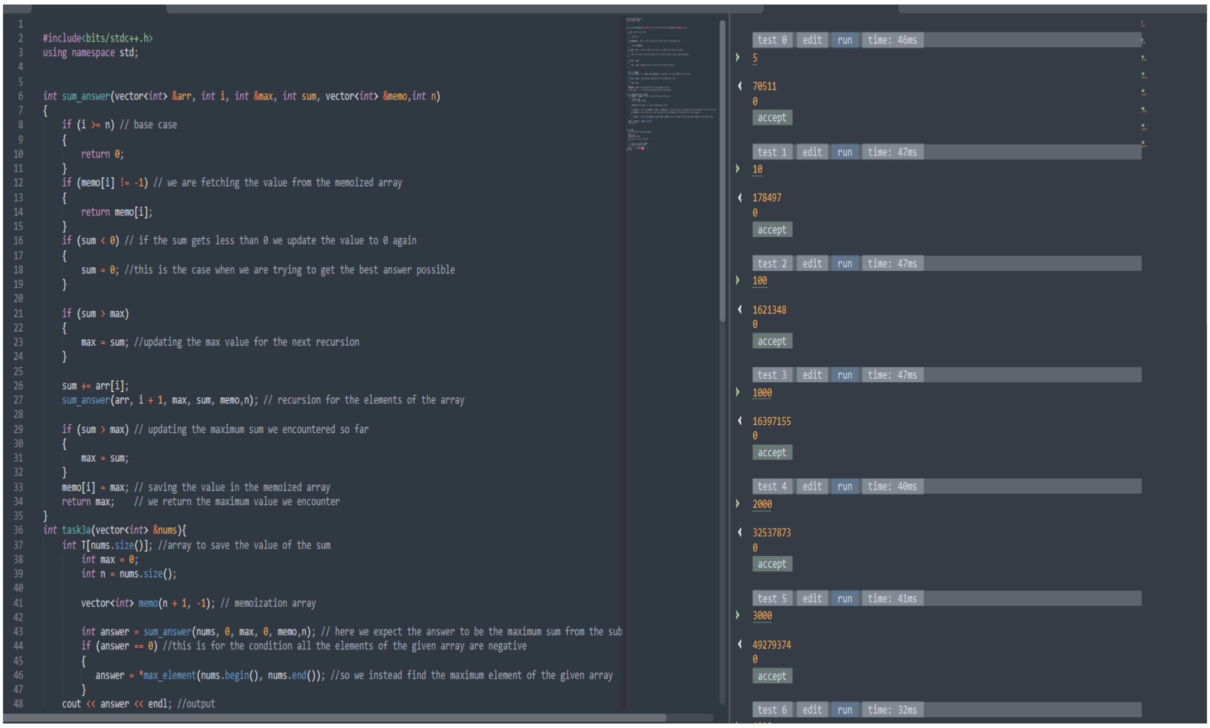
Time complexity: O(n^2)

Space Complexity: O(1)

Task3a Give a recursive implementation of Alg3 using Memoization.

```
int sum_answer(vector<int> &arr, int i, int &max, int sum, vector<int> &memo,int n)
{
	if (i >= n) // base case
	{
	return 0;
	}
	if (memo[i] != -1) // we are fetching the value from the memoized array
	{
	return memo[i];
	}
	if (sum < 0) // if the sum gets less than 0 we update the value to 0 again
	{
	sum = 0; //this is the case when we are trying to get the best answer possible
	}

	if (sum > max)
	{
	max = sum; //updating the max value for the next recursion
	}

	sum += arr[i];
	sum_answer(arr, i + 1, max, sum, memo,n); // recursion for the elements of the array

	if (sum > max) // updating the maximum sum we encountered so far
	{
	max = sum;
	}
	memo[i] = max; // saving the value in the memoized array
	return max;	// we return the maximum value we encounter
}
int task3a(vector<int> &nums){
	int T[nums.size()]; //array to save the value of the sum
	int max = 0;
	int n = nums.size();

	vector<int> memo(n + 1, -1); // memoization array

	int answer = sum_answer(nums, 0, max, 0, memo,n); // here we expect the answer to
be the maximum sum from the sub array.
	if (answer == 0) //this is for the condition all the elements of the given array are
negative
```

```
        {
        answer = *max_element(nums.begin(), nums.end()); //so we instead find the
maximum element of the given array
        }
        cout << answer << endl; //output
        return 0;
}
```



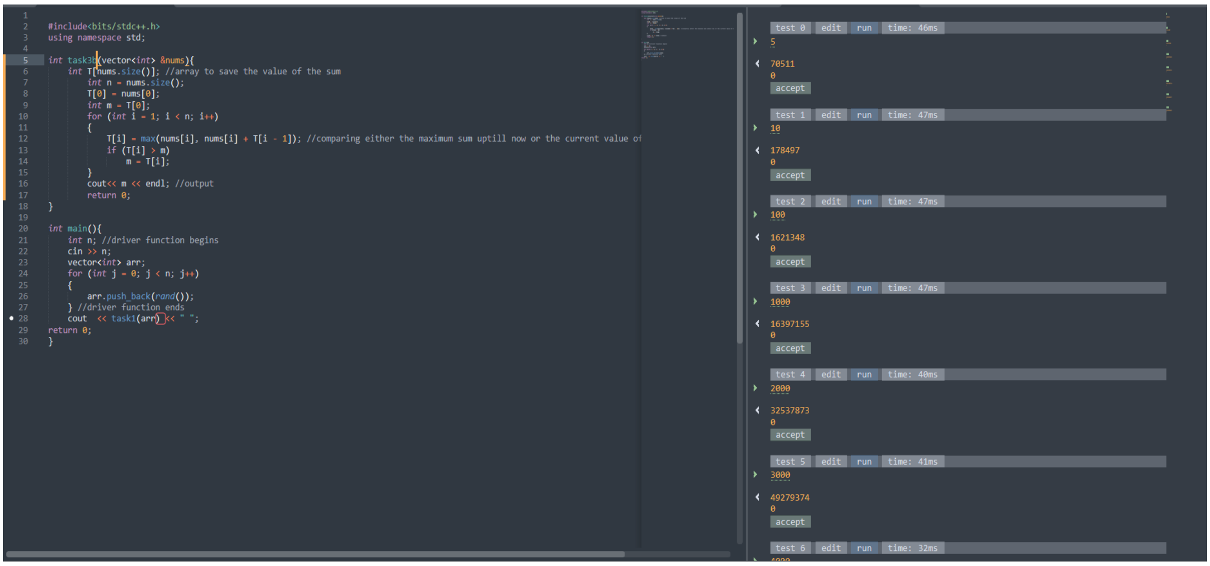
```
#include<bits/stdc++.h>
using namespace std;


int sum_answer(vector<int> &arr, int i, int &max, int sum, vector<int> &memo,int n)
{
    if (i >= n) // base case
    {
        return 0;
    }
    if (memo[i] != -1) // we are fetching the value from the memoized array
    {
        return memo[i];
    }
    if (sum < 0) // if the sum gets less than 0 we update the value to 0 again
    {
        sum = 0; //this is the case when we are trying to get the best answer possible
    }

    if (sum > max)
    {
        max = sum; //updating the max value for the next recursion
    }

    sum += arr[i];
    sum_answer(arr, i + 1, max, sum, memo,n); // recursion for the elements of the array

    if (sum > max) // updating the maximum sum we encountered so far
    {
        max = sum;
    }
    memo[i] = max; // saving the value in the memoized array
    return max;    // we return the maximum value we encounter
}
int task3a(vector<int> &nums){
    int T[nums.size()]; //array to save the value of the sum
        int max = 0;
        int n = nums.size();

        vector<int> memo(n + 1, -1); // memoization array

        int answer = sum_answer(nums, 0, max, 0, memo,n); // here we expect the answer to be the maximum sum from the sub
        if (answer == 0) //this is for the condition all the elements of the given array are negative
        {
            answer = *max_element(nums.begin(), nums.end()); //so we instead find the maximum element of the given array
        }
    cout << answer << endl; //output
```

test 0 edit run time: 46ms
5
70511
0
accept
test 1 edit run time: 47ms
10
178497
0
accept
test 2 edit run time: 47ms
100
1621348
0
accept
test 3 edit run time: 47ms
1000
16397155
0
accept
test 4 edit run time: 40ms
2000
32537873
0
accept
test 5 edit run time: 41ms
3000
49279374
0
accept
test 6 edit run time: 32ms

Time complexity: O(n)
Space Complexity: O(n)

Task3b Give an iterative bottom-up implementation of Alg3.

```
int maxSubArray(vector<int> &nums)
{
        int T[nums.size()];
        T[0] = nums[0];
        int m = T[0];
        for (int i = 1; i < nums.size(); i++)
        {
        T[i] = max(nums[i], nums[i] + T[i - 1]);
        if (T[i] > m)
        m = T[i];
        }
        return m;
}
```



Time complexity: O(n)
Space Complexity: O(n)

Task4 Give an implementation of Alg4 using O(1) extra space.
Code:

```
 int maxSum = -2147483647; //asssigning the maximum value to INT_MIN
int leftr = 0, leftc = 0, rightr = 0, rightc = 0;

for (int i = 0; i < n; ++i) //left most row value
{
for (int j = 0; j < m; j++) //left most column value
{
for (int k = i; k < n; k++) //right most row value
{
	for (int l = j; l < m; l++) //right most column value
	{
	int curSum = 0;
	for (int row = i; row <= k; row++) //computing the sum in the next two for
loops
	{
	for (int col = j; col <= l; col++)
	{
		curSum += mat[row][col];
		if (curSum > maxSum)
		{
		//updating the values of maximum sum, leftmost row and column and
rightmost row and column.
		maxSum = curSum;
		leftr = i;
		leftc = j;
		rightr = row;
		rightc = col;
		}
	}
	}
	}
}
}
}
cout << leftr+1 << " " << leftc+1 << " "<< rightr+1<< " "<<rightc+1<< "
"<<maxSum<<endl; //output
return 0;
}
```

```cpp
#include<bits/stdc++.h>
using namespace std;

int maxSumBruteForce(vector<vector<int>> matrix,int n,int m ) {
    int maxSum = INT_MIN;
    int leftr =0,leftc =0,rightr=0,rightc=0;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; j++) {
            for (int k = i; k < n; k++) {
                for (int l = j; l < m; l++) {
                    int curSum =0;
                    for (int row = i; row <= k; row++) {
                        for (int col = j; col <= l; col++) {
                            curSum += matrix[row][col];
                            if(curSum>maxSum){
                            maxSum=curSum;
                            leftr = i;
                            leftc = j;
                            rightr = row;
                            rightc= col;
                            }
                        }
                    }
                }
            }

        }

    }
   cout << leftr+1 << " " << leftc+1<< " "<< rightr+1<< " " << rightc+1 << endl;
    return maxSum;
}
```

```
you.cpp -run

test 0   edit   run   time: 31ms
{21, 3, -17, -14},
{15,-14,-31,-28},
{11,-21,24,-6},
{-2,23,-23,23}
1 1 3 1
47
decline

test 1   edit   run   time: 31ms
{0,5,-11,-61},
{ -41,-88,-24,-65},
{53,-18,29,-37},
{-38,52,0,5}
3 1 4 2
78
decline

test 2   edit   run   time: 31ms
{-1,-1,-1,-1},
{0,0,0,0},
{-1,-1,-1,-1},
{0,0,0,0}
2 1 2 1
0
decline

next test
```

Time Complexity : O (n^6)
Space Complexity : O(1)

Task5 Give an implementation of Alg5 using O(mn) extra space.

code :

```
int findMaxSumSubmatrix(vector<vector<int>> const &mat)
{
	// base case
	if (mat.size() == 0) {
	return 0;
	}

	// `M × N` matrix
	int M = mat.size();
	int N = mat[0].size();

	// `S[i][j]` stores the sum of submatrix formed by row 0 to `i-1`
	// and column 0 to `j-1`
	int S[M+1][N+1];

	// preprocess the matrix to fill `S`
	for (int i = 0; i <= M; i++)
	{
	for (int j = 0; j <= N; j++)
	{
	if (i == 0 || j == 0) {
		S[i][j] = 0;
	}
	else {
		S[i][j] = S[i-1][j] + S[i][j-1] - S[i-1][j-1] + mat[i-1][j-1];
	}
	}
	}

	int maxSum = INT_MIN;
	int rowStart, rowEnd, colStart, colEnd;

	// consider every submatrix formed by row `i` to `j`
	// and column `m` to `n`
	for (int i = 0; i < M; i++)
	{
	for (int j = i; j < M; j++)
	{
	for (int m = 0; m < N; m++)
	{
		for (int n = m; n < N; n++)
```

```
            {
            // calculate the submatrix sum using `S[][]` in O(1) time
            int submatrix_sum = S[j+1][n+1] - S[j+1][m] - S[i][n+1] + S[i][m];

            // if the submatrix sum is more than the maximum found so far
            if (submatrix_sum > maxSum)
            {
            maxSum = submatrix_sum;
            rowStart = i;
            rowEnd = j;
            colStart = m;
            colEnd = n;
            }
            }
        }
        }
        }

        cout << "The maximum sum submatrix is\n\n";
        for (int i = rowStart; i <= rowEnd; i++) {
        vector<int> row;
        for (int j = colStart; j <= colEnd; j++) {
        row.push_back(mat[i][j]);
        }
        printVector(row);
        }

        return maxSum;
}
```

```cpp
void printVector(vector<int> const &input)
{
    return;
}

// Find the maximum sum submatrix present in a given matrix
int findMaxSumSubmatrix(vector<vector<int>> const &mat)
{
    // base case
    if (mat.size() == 0) {
        return 0;
    }

    // `M x N` matrix
    int M = mat.size();
    int N = mat[0].size();

    // `S[i][j]` stores the sum of submatrix formed by row 0 to `i-1`
    // and column 0 to `j-1`
    int S[M+1][N+1];

    // preprocess the matrix to fill `S`
    for (int i = 0; i <= M; i++)
    {
        for (int j = 0; j <= N; j++)
        {
            if (i == 0 || j == 0) {
                S[i][j] = 0;
            }
            else {
                S[i][j] = S[i-1][j] + S[i][j-1] - S[i-1][j-1] + mat[i-1][j-1];
            }
        }
    }

    int maxSum = INT_MIN;
    int rowStart, rowEnd, colStart, colEnd;

    // consider every submatrix formed by row `i` to `j`
    // and column `m` to `n`
    for (int i = 0; i < M; i++)
    {
        for (int j = i; j < M; j++)
        {
            for (int m = 0; m < N; m++)
            {
                for (int n = m; n < N; n++)
                {
                    // calculate the submatrix sum using `S[][]` in O(1) time
```

```
59106997
accept

test 6 | edit | run | time: 1125ms
70

The maximum sum submatrix is

80987506
accept

test 7 | edit | run | time: 979ms
80

The maximum sum submatrix is

105908945
accept

test 8 | edit | run | time: 1061ms
90

The maximum sum submatrix is

133889244
accept

test 9 | edit | run | time: 885ms
100

The maximum sum submatrix is

165075799
accept

test 10 | edit | run | time: 2694ms
40

The maximum sum submatrix is

26084805
accept

next test
```

Time complexity: O(n^4)
Space Complexity: O(nm)

Task6 Give an implementation of Alg6 using O(mn) extra space.

```
int maxSubArray(vector<int> A)
{
	int maxSum = A[0];
	int n = A.size();
	int dp[A.size()];
	dp[0] = A[0];
	for (int i = 1; i < n; i++)
	{
	maxSum = max(A[i], maxSum + A[i]);
	dp[i] = max(dp[i - 1], maxSum);
	}
	return dp[A.size() - 1];
}
	int ar = INT_MIN;
	for (int l = 0; l < m; l++)
	{
	vector<int> sum(n);
	for (int r = l; r < m; r++)
	{
		for (int row = 0; row < n; row++)
		{
		sum[row] += mat[r][row];
		}
		ar = max(ar, maxSubArray(sum));
	}
	}
```

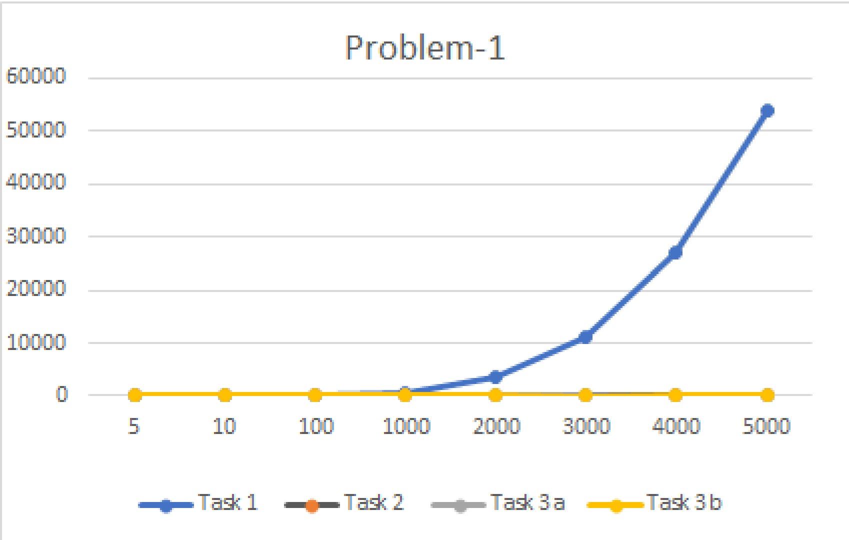
```cpp
#include<bits/stdc++.h>
using namespace std;


 int maxSubArray(vector<int> A){
      int maxSum = A[0];
      int dp[A.size()];
      dp[0] = A[0];
        for(int i=1;i<A.size();i++){
          maxSum = max(A[i],maxSum+A[i]);
           dp[i] = max(dp[i-1],maxSum);
        }
    return dp[A.size()-1];

    }

int maxSumOP(vector<vector<int>> matrix,int n,int m ) {

    int ar = INT_MIN;
    for(int l=0;l<m;l++){
        vector<int> sum (n);
        for(int r=l;r<m;r++){
            for(int row=0;row<n;row++){
                sum[row]+= matrix[r][row];

            }
            ar = max(ar,maxSubArray(sum));

        }
    }
    return ar;
}
```

```
test 0   edit   run   time: 31ms
{0,5,-11,-61},
{ -41,-88,-24,-65},
{53,-18,29,-37},
{-38,52,0,5}
78
decline

test 1   edit   run   time: 31ms
{21, 3, -17, -14},
{15,-14,-31,-28},
{11,-21,24,-6},
{-2,23,-23,23}
47
decline

test 2   edit   run   time: 31ms
{-1,-1,-1,-1},
{0,0,0,0},
{-1,-1,-1,-1},
{0,0,0,0}
0
decline

next test
```

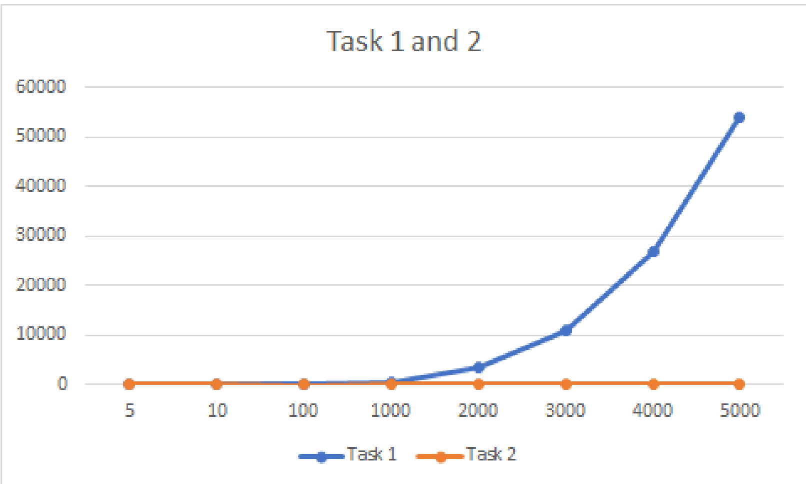
Time complexity: O(n^3)
Space Complexity: O(nm)

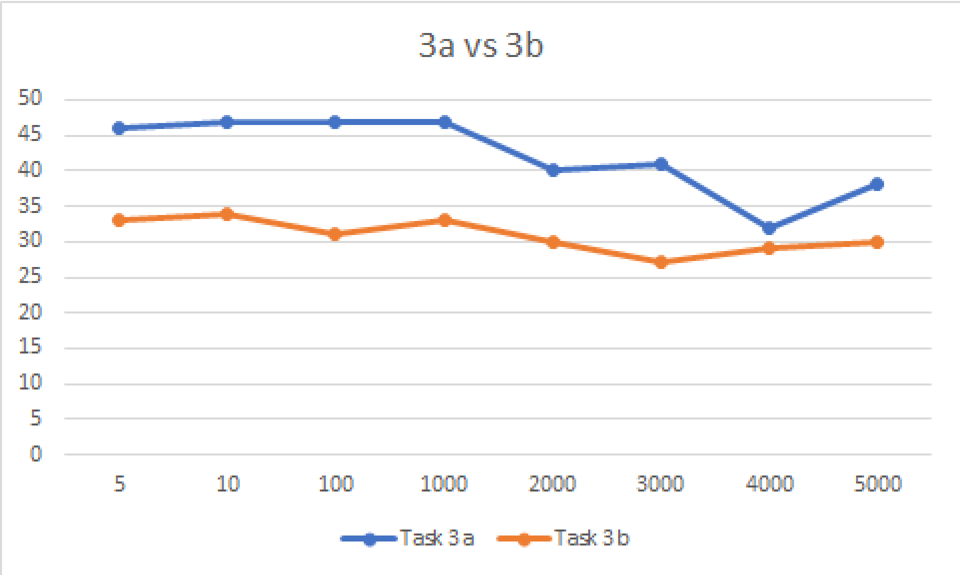
Comparative study

Problem-1	5	10	100	1000	2000	3000	4000	5000
Task 1	34	32	30	467	3507	11000	27000	54000
Task 2	35	32	30	32	37	43	52	60
Task 3a	46	47	47	47	40	41	32	38
Task 3b	33	34	31	33	30	27	29	30

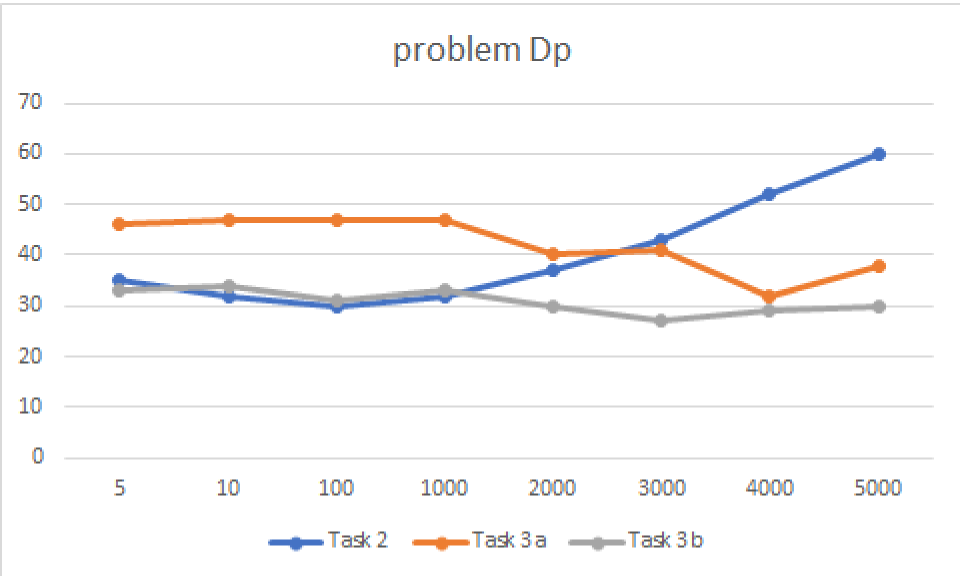


- Task 1 grows explosively as we start comparing it for bigger inputs compared to other tasks.



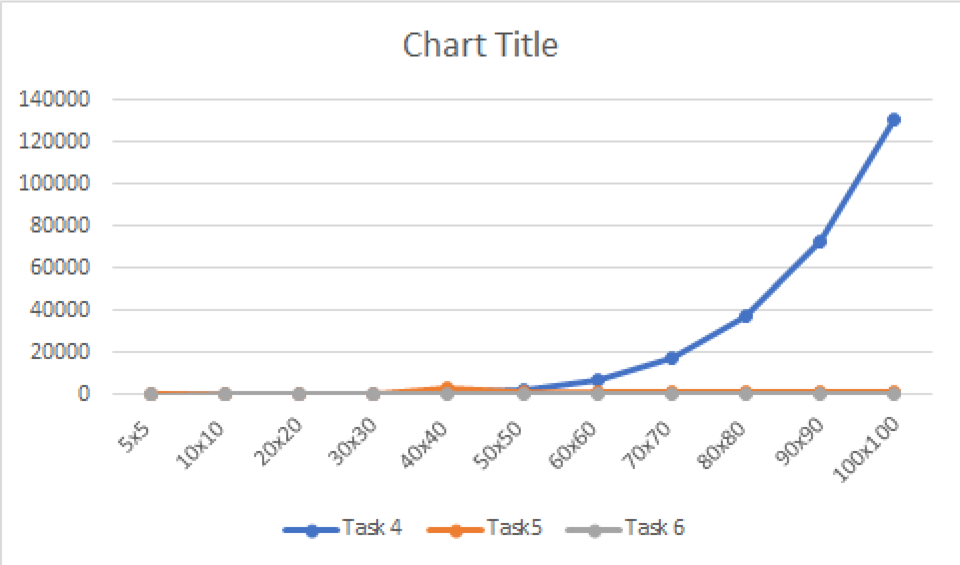


- In the plot above we can clearly see that task 3a clearly takes longer time than task 3b ,this happens because of recursion. Calling stack function one after the other slows down the recursive approach.

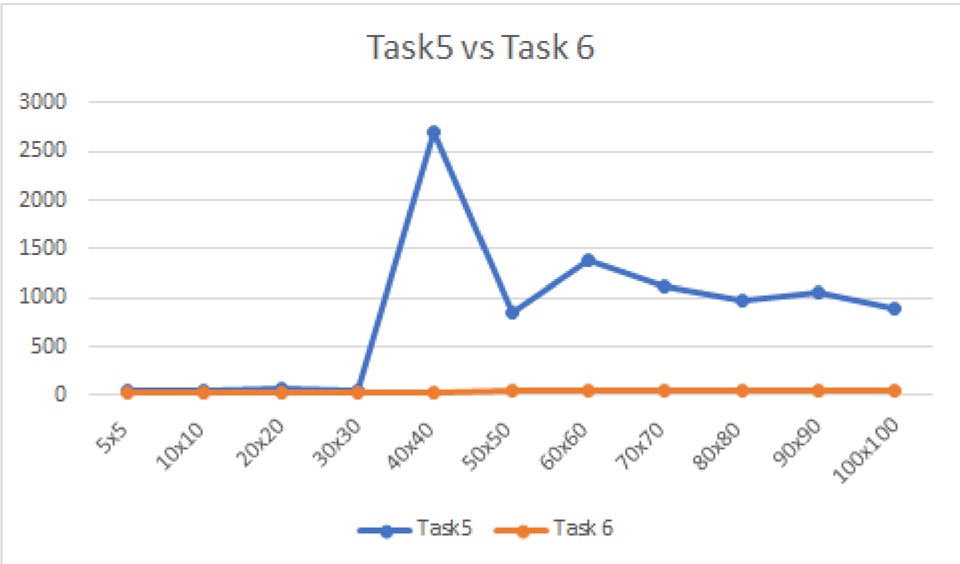


- Task 2 initially takes somewhere around the same amount of time as tasks 3a and tasks 3b but as soon as the input grows exponentially task 2's growth also grows explosively.

Problem-2	5x5	10x10	20x20	30x30	40x40	50x50	60x60	70x70	80x80	90x90	100x100	
Task 4	51	54	75	187	740	2559	7000	17000	37000	73000	131000	
Task 6	31	35	35	35	38	41	42	41	44	42	47	



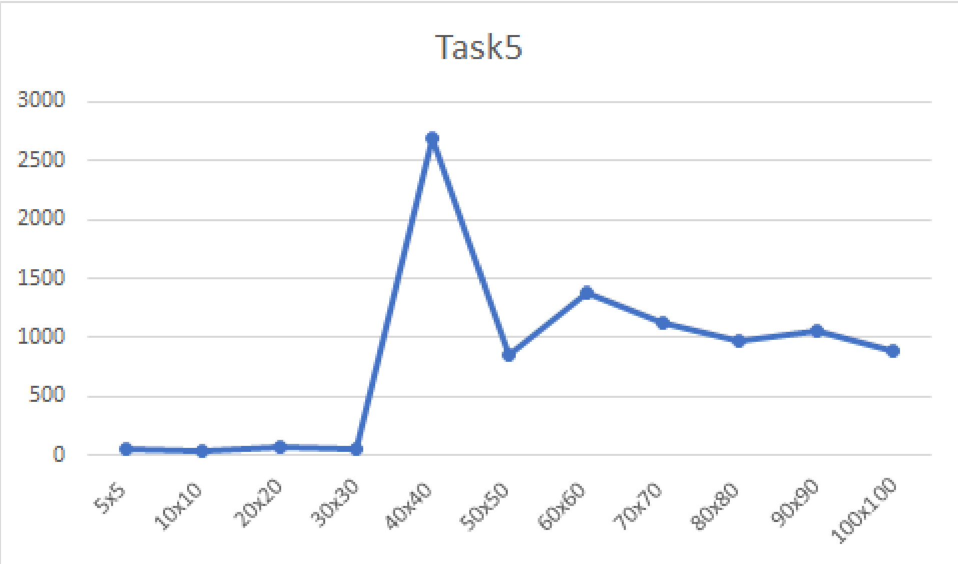
- Here as the size of the input matrix grows Task4 starts growing expolosively where as task6 still fairly quick.

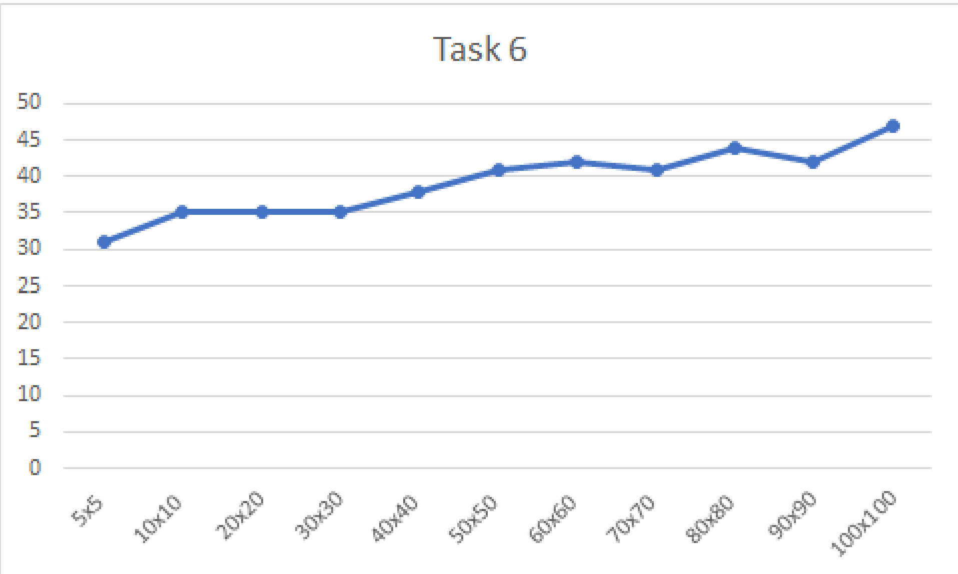


Task 5 performs poorly as the input size increases.

-


Task 4
140000
120000
100000
80000
60000
40000
20000
0
5x5
10x10
20x20
30x30
40x40
50x50
60x60
70x70
80x80
90x90
100x100


Task5
3000
2500
2000
1500
1000
500
0
5x5
10x10
20x20
30x30
40x40
50x50
60x60
70x70
80x80
90x90
100x100


Task 6
50
45
40
35
30
25
20
15
10
5
0
5x5
10x10
20x20
30x30
40x40
50x50
60x60
70x70
80x80
90x90
100x100
,

Conclusion

- This assignment takes a deep dive into the applications of dynamic programming. The tasks given made me understand the true process of solving any problem. The brute force approach followed by a better approach and eventually trying to build the optimized approach on top of the previous approaches used.
- Task 1 was fairly easy to implement as the brute force was very straightforward and was implemented with the use of simple for loops.
- Task 2 was a little more trickier as we eliminate a for loop and start using variables to keep a track instead.
- Task 3
    - 3a was one of the most trickiest part of this assignment as solving recursion needs creativity and to land on the final approach I had to try several other approaches.
    - 3b was easier than 3a but it still took a good amount of time to come to and understanding of the solution.
- Task 4 again was fairly simple as soon as I was able to visualize the 2d matrix and how the iterators move along the matrix. But once that part clicked I was able to solve the task easily.
- Task 5 still remains the hardest task for me, Using four loops exactly with algo 3 posed a lot of difficulties.
- Task 6 was fairly easy once i understood that we can use algo 3 on the maximum sum we encountered.