Maximum Submatrix Sum Problem

2023-9-28

Introduction

Maximum Submatrix Sum Problem is an extension of **Maximum Subsequence Sum Problem**. Using **Kadane's algorithm**, the time complexity of the maximum subsequence problem can be optimized to $O(n)$. In this project, the primary focus is on the **algorithmic complexity** analysis and **optimization of the maximum submatrix problem**.

Algorithms

Direct Solution

Introduction

The direct solution to Maximum Submatirx Sum Problem simply explores all the possible sub matrix and calculates the sum of all the elements using nested loops.

```
function directSolution
  for i in all the rows
    for j in all the colunms
      for k from i to end row
        for l from j to end colunm
          //(i,j)-(k,l) partion a submatrix
          currentSum = sum of elements in submatrix
          if currentSum > max
            max = currentSum
  return max
```

Complexity Analysis

In every single run for $i = x_1, j = y_1, k = x_2, l = y_2$, the number of caculation needed is

$$(x_2 - x_1) \times (y_2 - y_1) \tag{1}$$

The number of total calculation needed is

$$\sum_{x_1=0}^{N} \sum_{y_1=0}^{N} \sum_{x_2=x_1}^{N} \sum_{y_2=y_1}^{N} (x_2 - x_1) \times (y_2 - y_1) \tag{2}$$

Ignoring the terms with lower exponents, we have

$$(\sum_{x_1=0}^{N} sum(x_1, N)) \times (\sum_{y_1=0}^{N} sum(y_1, N)) \tag{3}$$

$sum(x_1, N)$ represents the sum of integers from $x1$ to N, has the complexity of N^2. Sum of N^2 has the complexity of N^3. **The time complexity in total is $\Theta(N^6)$.**

There is no other data structure using to store data. So, **the space complexity in total is** $\Theta(1)$.

Matrix Caculation Optimization

Introduction

In the explorition of all the matrixs. We could realize that some caculation is unnecessary, there is a more elegent way of cauculating the sum of a matrix's elements. That is to take previous sum as a condition.

```
function caculateOptimize
  //Create a matrix with the size of n+1 to store result
  //To simplify caculation, the left and upper element contains zero
  result = matrix(size+1,size+1)
  //set all element to zero
  memset(result,0,sizeof(int)*(size+1)*(size+1))
  for i in all the rows
    for j in all the colunms
      for k from i+1 to end row+1
        for l from j+1 to end colunm=1
          //caculate sum by previous result
          result[i][j][k][l] = result[i][j][k-1][l]+
                               result[i][j][k][l-1]
                               result[i][j][k-1][l-1]+
                               matrix[k-1][l-1]
          if result[i][j][k][l] > max
            max = result[i][j][k][l]
```

Complexity Analysis

The caculation for a single run has the time complexity of $O(1)$. So the total loops needed is

$$\sum_{x_1=0}^{N}\sum_{y_1=0}^{N}\sum_{x_2=x_1+1}^{N+1}\sum_{y_2=y_1+1}^{N+1} 1 \tag{4}$$

$$\sum_{x_1=0}^{N}\sum_{y_1=0}^{N}(N-x_1)\times(N-x_2) \tag{5}$$

Ignoring the terms with lower exponents, we have

$$sum(0,N)\times sum(0,N) \tag{6}$$

It's obvious that **the time complexity is** $\Theta(N^4)$.

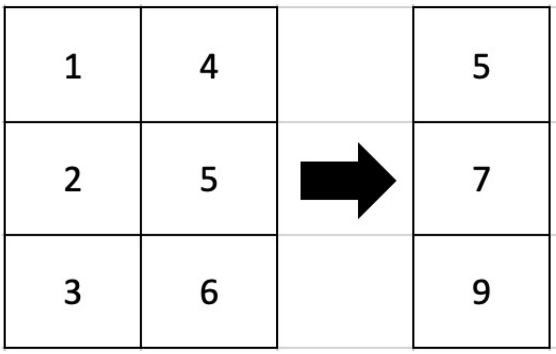
A two-dimension array with the size of (n+1)*(n+1) is needed. So **the space complexity is** $\Theta(N^2)$.

Algorithm of Kadane

Introduction

in the situation of single dimension, kadane algorithm is involved. The idea of ignoring the negative sum can also be involved in two-dimension situation. If the sum of a exsisting matrix is negative, it should be set to zero and restart caculating.

```
function Kadane
  for i in all the colunms
    //create a array and fill it with zero
    temp = array[size]
    memset(temp,0,sizeof(int)*temp)
    for j from i to end colunm
      //the loop restrict the left side and right side of colunm
      //temp is a one dimension array
      //each time of inner loop, temp[k] adds a new element, turns matrix sum to a one dimension array
      for k in all the rows
        temp[k] += matrix[j][k]
      for k in all rows
        sum += temp[k]
        if sum < 0
          sum = 0
        if sum > max
          max = sum
```



`temp[k] += matrix[i][j];`

Complexity Analysis

For certain start colunm `i` and a end colunm `j`. The add of `temp` needs n caculates. And a one dimension kadane algorithm has the complexity of $O(N)$,which only contains adding. So $2n$ caculation is needed in a single loop. The total caculation is

$$\sum_{x_1=0}^{N} \sum_{x_2=x_1+1}^{N+1} 2N \tag{7}$$

The time complexity is $\Theta(n^3)$.

As a array `temp` is needed in the caculation. **The space complexity is $\Theta(N)$.**

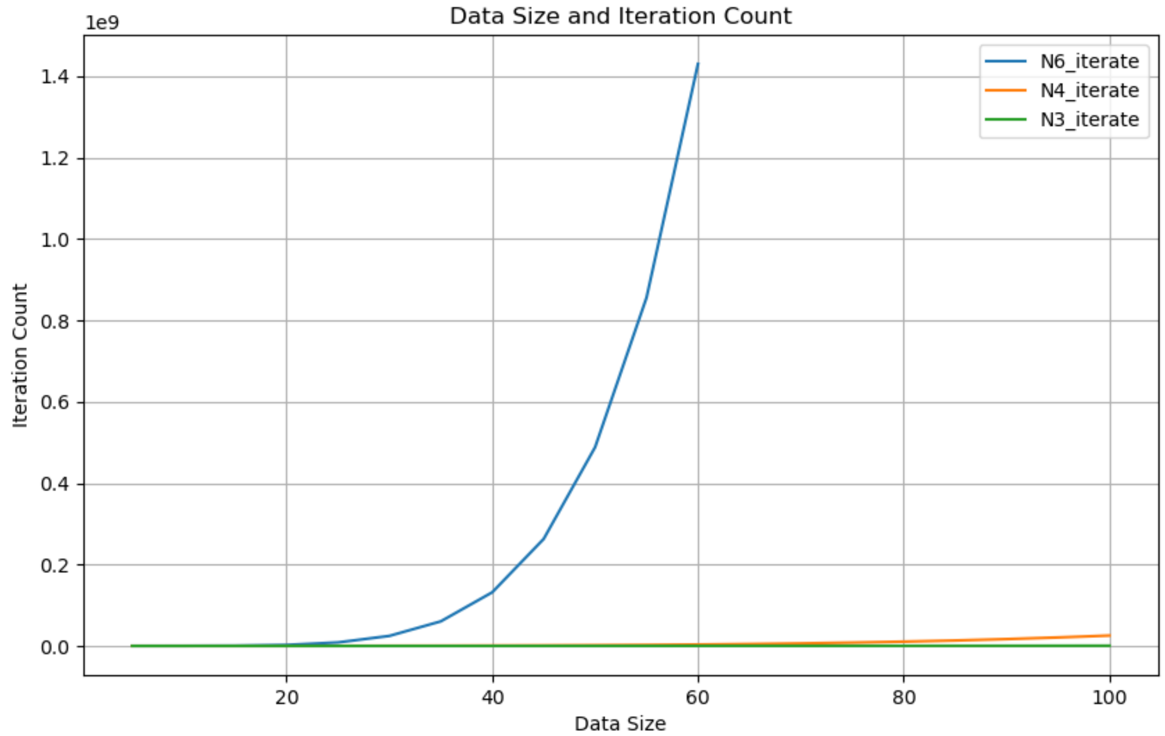
Comparisons of Algorithms

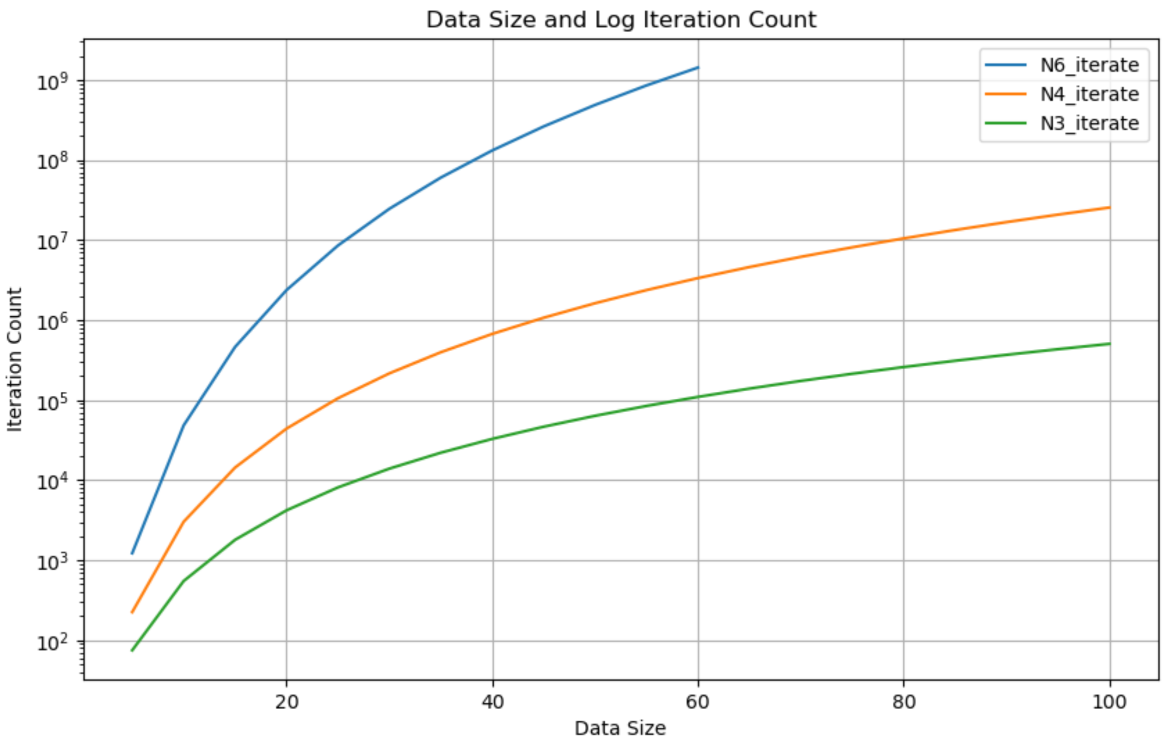
A tick equals to $1\mu s$. The quickest algorithm costs $3\mu s$, so all the tests are taken once, duration is the same as total time.

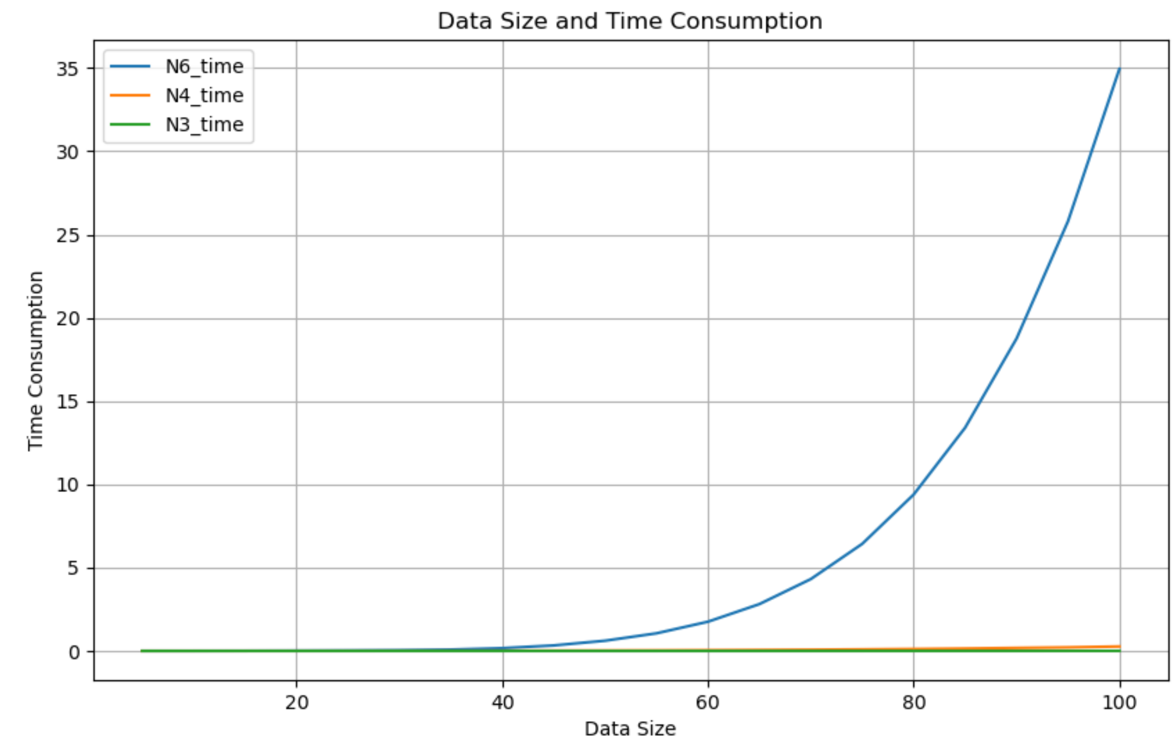
```
#define CLOCKS_PER_SEC ((clock_t)1000000)

[XSI]
扩展到:

((clock_t)1000000)
```

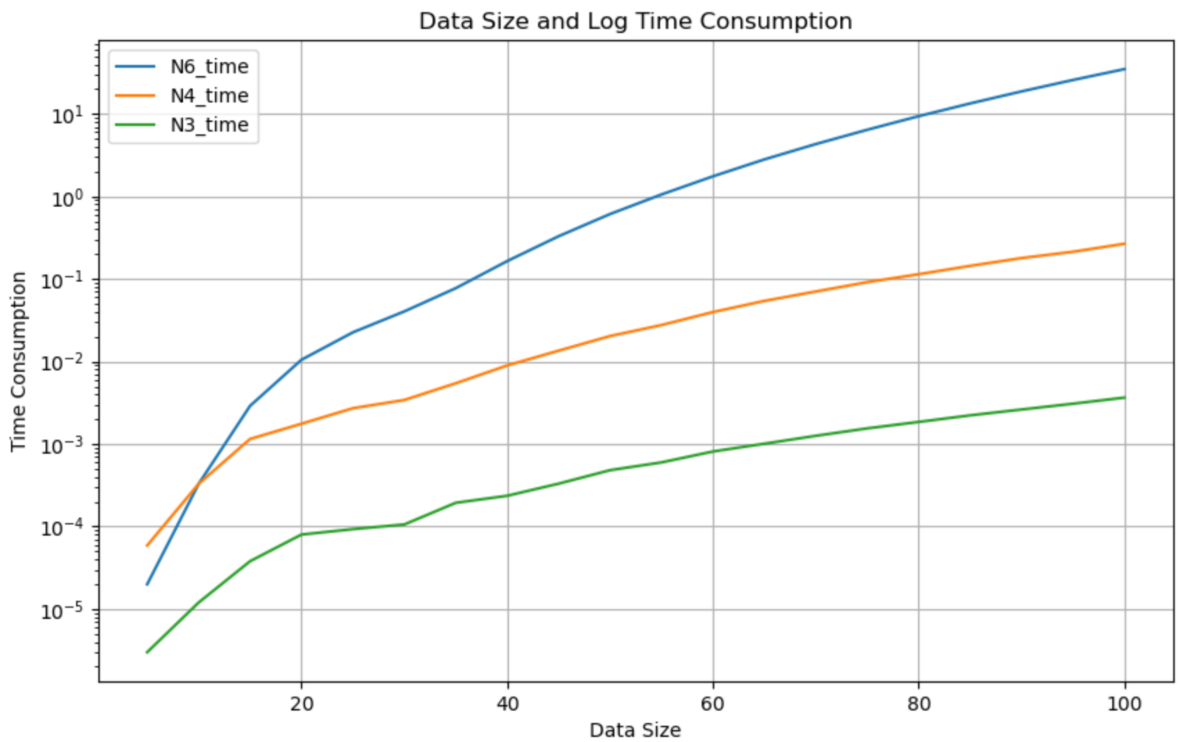
	N	5	10	30	50	80	100
$\Theta(N^6)$	Iterations(K)	1	1	1	1	1	1
	Ticks	20	329	4×10^4	6.1×10^5	9.3×10^6	3.5×10^7
	Total Time(sec)	0.00002	0.000329	0.04	0.613	9.39	34.95
	Duration(sec)	0.00002	0.000329	0.04	0.613	9.39	34.95
$\Theta(N^4)$	Iterations(K)	1	1	1	1	1	1
	Ticks	59	329	3.4×10^3	2×10^4	1.1×10^5	2.7×10^5
	Total Time(sec)	0.000059	0.000329	0.0034	0.02	0.114	0.267
	Duration(sec)	0.000059	0.000329	0.0034	0.02	0.114	0.267
$\Theta(N^3)$	Iterations(K)	1	1	1	1	1	1
	Ticks	3	12	100	482	1850	3650
	Total Time(sec)	0.000003	0.000012	0.0001	0.000482	0.00185	0.00365
	Duration(sec)	0.000003	0.000012	0.0001	0.000482	0.00185	0.00365

Plot the data using linear and logarithmic coordinates separately.


Data Size and Iteration Count
1e9
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Iteration Count
20
40
60
80
100
Data Size
N6_iterate
N4_iterate
N3_iterate


Data Size and Log Iteration Count
10^9
10^8
10^7
10^6
10^5
10^4
10^3
10^2
Iteration Count
20
40
60
80
100
Data Size
N6_iterate
N4_iterate
N3_iterate

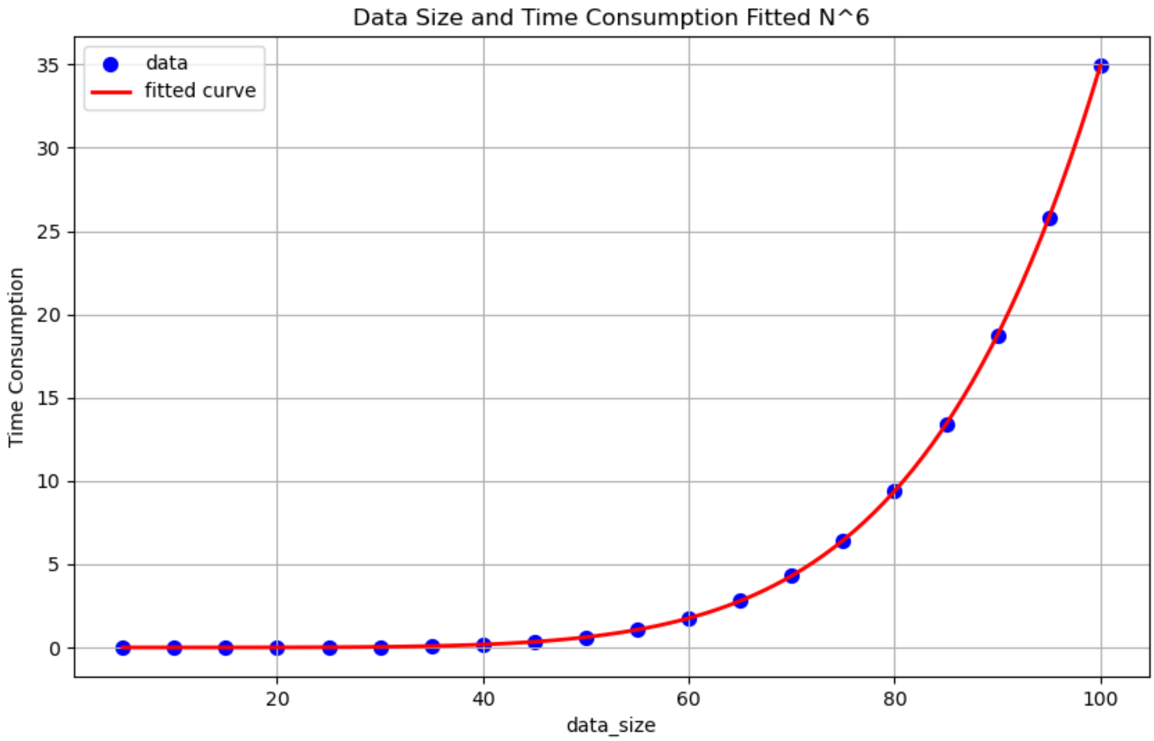


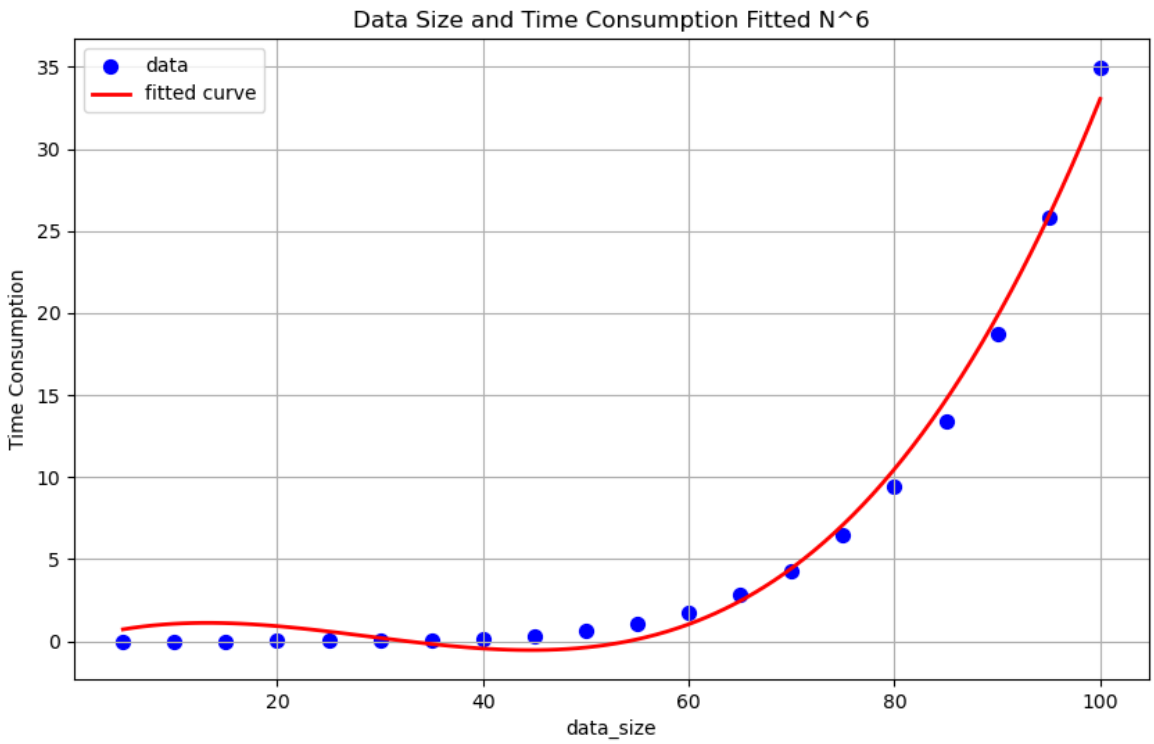


Difference is subtle when input data is in a small size. But significant differences occur in algorithms with different complexities as the input data size increases. Algorithms with higher time complexity costs hundreds of time more than optimized algorithm.

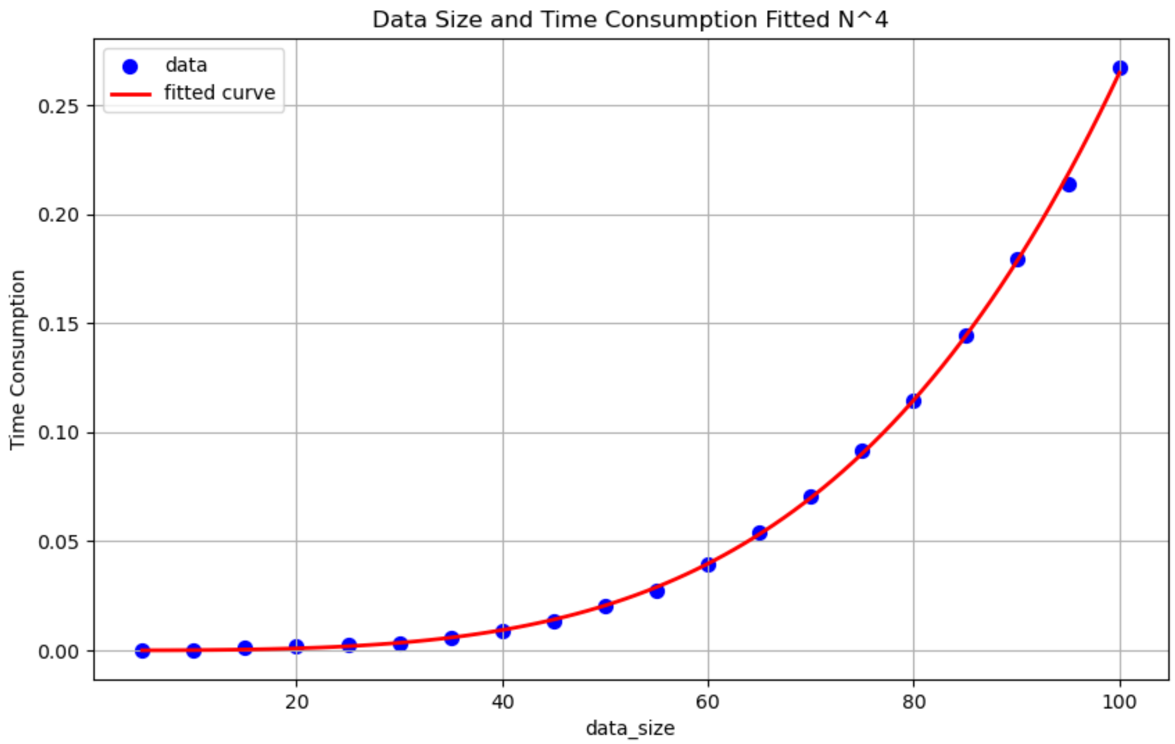
Besides, Kadane's algorithm has better performance in time-consuming comparing with other algorithms. The space it needed is merely $\Theta(N)$. It performs best in solving the problem.

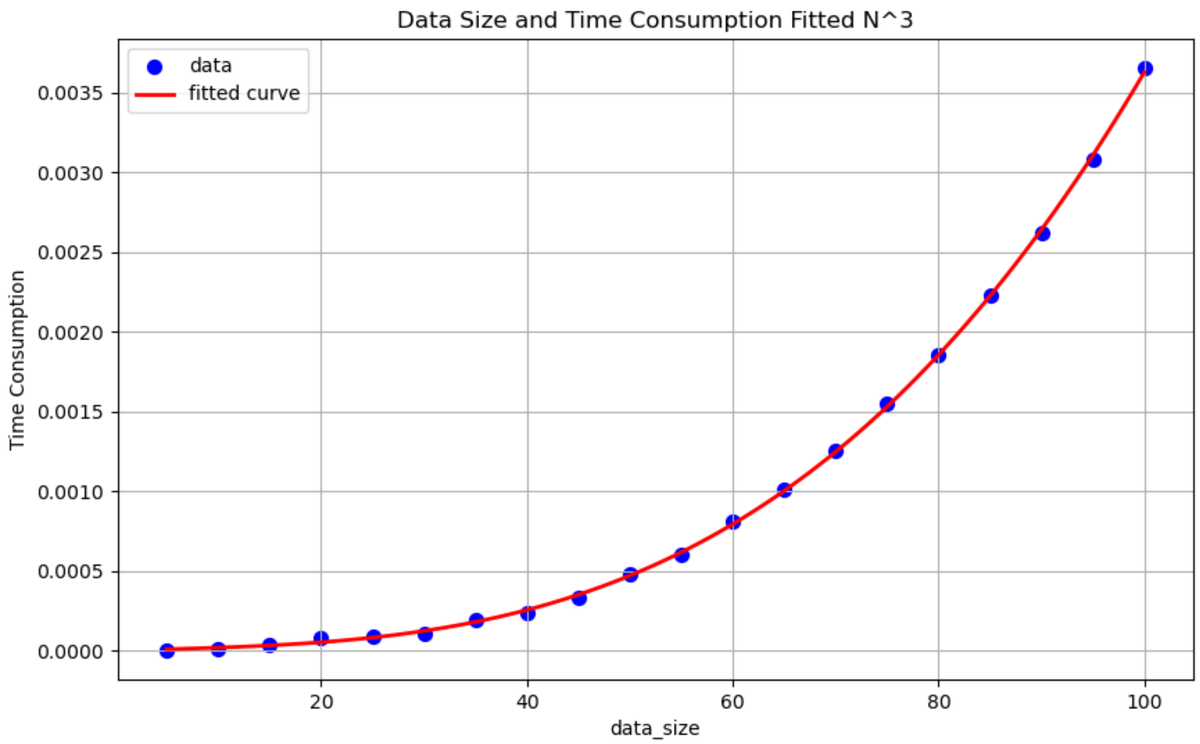
Finally, take $\Theta(N^6)$ algorthm as an example.





Fitting data separately using highest exponent of x^3 and x^6. x^6 reaches a better fitting, indicating that our analysis to time complexity has a high confidence.





Fitting the curve for the time complexity of N^4 and N^3 separately, also draws a result with high confidence.

Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

//time recorder
clock_t start,stop;
double duration;

//three modes deciding the way of input
//terminalInput:read element from terminal
//directInput:using globe varible Matrix
//randomInit:init a matrix with random varibles
enum mode
{
    terminalInput,
    directInput,
    randomInit,
};

//matrix process functions
int** matrixInput(enum mode m_mode);
int** matrixRead(int size);
int** generateRandomMatrix(int size);
void matrixPrint(int** matrix,int start_x,int start_y,
                              int end_x,  int end_y   );

//solving functions
int solve_on6(int** matrix,int size);
int solve_on4(int** matrix,int size);
int solve_on3(int** matrix,int size);

//size of globe varible Matirx should correspond to directMatrixSize
int directMatrixSize = 4;
int Matrix[4][4] = {
    { 0, -2, -7,  0},
    { 9,  2, -6,  2},
    {-4,  1, -4,  1},
    {-1,  8,  0, -2}
};

//correspond to the size of random matrix
int random_size = 5;
//mode init
enum mode m_mode = randomInit;

int main()
{
    //to store data
```

```
    //contains process time and iterate count
    double duration[3];
    int iterate[3];
    //initialize duration
    duration[0]=duration[1]=duration[2]=0;
    //init a matrix of certain size 'average' time
    //eg. average = 5 ; for all the matrix size, duration will be the average of 5
times
    int average = 1;
    int i,j;

    //create a file called "data.csv" storing data
    FILE *file = fopen("data.csv", "w");

    //the title of each colunm
    fprintf(file,
"data_size,N6_iterate,N6_time,N4_iterate,N4_time,N3_iterate,N3_time\n");

    //caculate from size=5 to size=100
    //caculate every size 'average' times
    //store the result in 'data.csv'
    for(random_size=5;random_size<=100;random_size+=5){
        for(i=0;i<average;i++){
            int** m_matrix = matrixInput(m_mode);

            //start the clock
            start = clock();
            //solve the matrix
            iterate[0] = solve_on6(m_matrix,random_size);
            //terminate the clock
            stop = clock();
            duration[0] += ((double)(stop - start)/CLOCKS_PER_SEC);

            //start the clock
            start = clock();
            //solve the matrix
            iterate[1] = solve_on4(m_matrix,random_size);
            //terminate the clock
            stop = clock();
            duration[1] += ((double)(stop - start)/CLOCKS_PER_SEC);

            //start the clock
            start = clock();
            //solve the matrix
            iterate[2] = solve_on3(m_matrix,random_size);
            //terminate the clock
            stop = clock();
            duration[2] += ((double)(stop - start)/CLOCKS_PER_SEC);
```

```
            //release memory in reverse order
            //to ensure there are no memory leaks
            //set the pointer to NULL to avoid dangling pointers
            for(j=0;j<random_size;j++){
                free((m_matrix)[j]);
            }
            free(m_matrix);
            m_matrix = NULL;
        }
        //write data to the file
        //use python pandas to read csv
        //use python matplotlib to plot the curve
        fprintf(file,"%d, %d, %lf, %d, %lf, %d, %lf\n",
                random_size,
                iterate[0], duration[0]/average,
                iterate[1], duration[1]/average,
                iterate[2], duration[2]/average);
        duration[0]=duration[1]=duration[2]=0;
    }

    fclose(file);
    printf("successfully store data into 'data.csv'");
}

// According to the mode, initialize and return a matrix
int** matrixInput(enum mode m_mode)
{
    int** mat;
    int size,i;
    switch (m_mode)
    {
    case terminalInput:
        scanf("%d",&size);
        //allocate memory dynamically according to matrix size
        mat = (int**) malloc(sizeof(int*)*size);
        for(i=0;i<size;i++){
            mat[i] = (int*) malloc(sizeof(int)*size);
        }
        mat = matrixRead(size);
        break;

    case directInput:
        //allocate memory dynamically according to matrix size
        mat = (int**) malloc(sizeof(int*)*directMatrixSize);
        for(i=0;i<directMatrixSize;i++){
            mat[i] = Matrix[i];
        }
        break;
```

```
    case randomInit:
        //generate random matrix using globe varible 'random_size'
        mat = generateRandomMatrix(random_size);
        break;
    }
    return mat;
}

//read input from terminal
//the number of elements should correspond to size*size
int** matrixRead(int size)
{
    int** matrix = (int**) malloc(sizeof(int*)*size);
    int i,j;
    for(i=0;i<size;i++){
        matrix[i] = (int*) malloc(sizeof(int)*size);
        for(j=0;j<size;j++){
            scanf("%d",&matrix[i][j]);
        }
    }
    return matrix;
}

//generate a random matrix
//matrix size correspond to input 'size'
int** generateRandomMatrix(int size)
{
    int** matrix = (int**) malloc(sizeof(int*)*size);
    int i,j;
    for(i=0;i<size;i++){
        matrix[i] = (int*) malloc(sizeof(int)*size);
        for(j=0;j<size;j++){
            matrix[i][j]=rand();
        }
    }
    return matrix;
}

//print matrix
//input the coordinate of top-left corner
//coordinate of bottom-left corner
//output matrix in terminal
void matrixPrint(int** matrix,int start_x,int start_y,
                              int end_x,  int end_y   )
{
    int i,j;
    for(i=start_x;i<=end_x;i++){
        for(j=start_y;j<=end_y;j++){
            printf("%2d ",matrix[i][j]);
```

```c
        }
        printf("\n");
    }
}

//solution with the time complexity of o(N^6)
//simply search all the possible submatrix
int solve_on6(int** matrix, int size)
{
    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the matrix is:\n");
    //matrixPrint(matrix,0,0,size-1,size-1);

    //varibles initialize
    int max = 0;
    int index_l_x = 0;
    int index_l_y = 0;
    int index_r_x = 0;
    int index_r_y = 0;
    int iterate = 0;

    //6 nested loop
    //directly find all the possible matrix
    int i, j, k, l, m, n;
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            for (k = i; k < size; k++) {
                for (l = j; l < size; l++) {
                    int sum = 0;
                    for (m = i; m <= k; m++) {
                        for (n = j; n <= l; n++) {
                            sum += matrix[m][n];
                            iterate ++;
                        }
                    }
                    //if current sum > maximum
                    //record current sum and record the coordinates of matrix
                    if (sum > max) {
                        max = sum;
                        index_l_x = i;
                        index_l_y = j;
                        index_r_x = k;
                        index_r_y = l;
                    }
                }
            }
        }
```

```
    }

    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the maximum submatrix is:\n");
    //matrixPrint(matrix,index_l_x,index_l_y,index_r_x,index_r_y);
    //printf("the maximum sum is: %d\n",max);

    return iterate;
}

//solution with the time complexity of o(N^4)
int solve_on4(int** matrix, int size)
{
    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the matrix is:\n");
    //matrixPrint(matrix,0,0,size-1,size-1);

    //varibles initialize
    int max = 0;
    int index_l_x = 0;
    int index_l_y = 0;
    int index_r_x = 0;
    int index_r_y = 0;
    int i, j, k, l;
    int iterate = 0;

    //malloc space dynamically
    //the size of element sum should be 'size'+1
    //the leftmost column and the topmost row are both filled with zeros
    //using calloc to initialize the value
    int**** element_sum;
    element_sum = (int****) calloc(size+1,sizeof(int***));
    for(i=0;i<size+1;i++){
        element_sum[i] = (int***) calloc(size+1,sizeof(int**));
        for(j=0;j<size+1;j++){
            element_sum[i][j] = (int**) calloc(size+1,sizeof(int*));
            for(k=0;k<size+1;k++){
                element_sum[i][j][k] = (int*) calloc(size+1,sizeof(int));
            }
        }
    }

    //caculating element sum by previous result
    //every elementsum consists of 4 parts
    //left part, upper part, left-top part and value of matrix
```

```
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            for (k = i+1; k < size+1; k++) {
                for (l = j+1; l < size+1; l++) {
                    element_sum[i][j][k][l] = element_sum[i][j][k-1][l]+
                                              element_sum[i][j][k][l-1]-
                                              element_sum[i][j][k-1][l-1]+
                                              matrix[k-1][l-1];
                    iterate ++;
                    //if sum > maximum
                    //record the sum and record the coordinate
                    if(element_sum[i][j][k][l]>max){
                        max = element_sum[i][j][k][l];
                        index_l_x = i;
                        index_l_y = j;
                        index_r_x = k-1;
                        index_r_y = l-1;
                    }
                }
            }
        }
    }

    //release memory in reverse order
    //to ensure there are no memory leaks
    //set the pointer to NULL to avoid dangling pointers
    for (int i = 0; i < size+1; i++) {
        for (int j = 0; j < size+1; j++) {
            for (int k = 0; k < size+1; k++) {
                free((element_sum)[i][j][k]);
            }
            free((element_sum)[i][j]);
        }
        free((element_sum)[i]);
    }
    free(element_sum);
    element_sum = NULL;

    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the maximum submatrix is:\n");
    //matrixPrint(matrix,index_l_x,index_l_y,index_r_x,index_r_y);
    //printf("the maximum sum is: %d\n",max);

    return iterate;
}

//solution with the time complexity of o(N^3)
```

```c
//return the count of iterate times
int solve_on3(int** matrix, int size)
{
    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the matrix is:\n");
    //matrixPrint(matrix,0,0,size-1,size-1);

    //initialize varibles
    int max = 0;
    int index_l_x = 0;
    int index_l_y = 0;
    int index_r_x = 0;
    int index_r_y = 0;
    int i, j, k;
    int iterate = 0;

    //create 'temp' filled with 0
    //using calloc to set elements to 0
    int* temp = (int*) calloc(size,sizeof(int));

    //using two nested loops
    //to constrain left colunm and right colunm
    for(i=0;i<size;i++){
        for(j=i;j<size;j++){
            int sum = 0;
            int indexStart = 0;
            int indexCount = 0;
            //using temp to transfer a 2-dimensions
            //to 1-dimension and involve kadane algoriothms
            for(k=0;k<size;k++) temp[k]+=matrix[k][j];
            for(k=0;k<size;k++){
                //kadane's algorithm
                sum += temp[k];
                indexCount ++;
                iterate ++;
                if(sum < 0){
                    sum=0;
                    indexStart = k;
                    indexCount = 0;
                }
                //if sum > max record sum of the new maximum
                //record the coordinates of the submatrix
                if(sum > max){
                    max = sum;
                    index_l_x = indexStart;
                    index_l_y = i;
                    index_r_x = indexStart+indexCount;
```

```
                    index_r_y = j;
                }
            }
        }
        //refill the temp array with 0
        memset(temp,0,sizeof(int)*size);
    }

    //the following code is used for debugging the program
    //comment out the piece of code
    //preventing it from affecting time caculating
    //printf("the maximum submatrix is:\n");
    //matrixPrint(matrix,index_l_x,index_l_y,index_r_x,index_r_y);
    //printf("the maximum sum is: %d\n",max);

    return iterate;
}
```

Declaration

I hereby declare that all the work done in this project is of my independent effort.