Lab 1:Maximum Submatrix Sum Problem Report

October 16, 2023

浙江大學

[t]

Chapter 1:Introduction

The Maximum Submatrix Sum problem is a well-known computational challenge that involves finding the maximum sum of a submatrix within an $N \times N$ integer matrix. This report aims to explore and analyze three different algorithms for solving this problem: one with a time complexity of $O(N^6)$, one with a time complexity of $O(N^3)$, and another with a time complexity of $O(N^4)$. The motivation behind this study is to understand the efficiency trade-offs between these algorithms and their performance under varying matrix sizes.

Chapter 2:Algorithm Overview

overview of the program

```
1. Declare global variables:
   - int a[100][100] # To store the matrix
   - int N # Size of the matrix (N*N)
   - long long num # Counter variable

2. Define a function to generate a random matrix generateMatrix(n):
   - Use srand and rand to generate random numbers to fill matrix a

3. Define a function to compute the maximum submatrix sum with N^4
   complexity max_matrix_of_N4(s[100][100]):
   - Determine the number of iterations num_of_iteration based on
      different N values
   - Repeat num_of_iteration times:
     - Copy input matrix s to a temporary matrix b
     - Calculate prefix sums in the horizontal direction
     - Use four nested loops to compute the maximum submatrix sum
   - Output the maximum submatrix sum max

4. Define a function to compute the maximum submatrix sum with N^6
   complexity max_matrix_of_N6(s[100][100]):
   - Determine the number of iterations num_of_iteration based on
      different N values
   - Repeat num_of_iteration times:
     - Copy input matrix s to a temporary matrix b
     - Use six nested loops to compute the maximum submatrix sum
   - Output the maximum submatrix sum max

5. Define a function to compute the maximum submatrix sum with N^3
   complexity max_matrix_of_N3(s[100][100]):
   - Determine the number of iterations num_of_iteration based on
      different N values
   - Repeat num_of_iteration times:
     - Copy input matrix s to a temporary matrix b
     - Calculate prefix sums in the horizontal direction
     - Use three nested loops to compute the maximum submatrix sum
   - Output the maximum submatrix sum max

6. Main function main():
   - Loop until the input N equals -1:
```

```
    - Read the size of the matrix N
    - Call generateMatrix(N) to generate a random matrix
    - Loop through the elements of the matrix to display them
    - Call max_matrix_of_N6(a) to compute the maximum submatrix sum
      with N^6 complexity
    - Output the computed result, execution time, and other
      statistics
    - Call max_matrix_of_N3(a) to compute the maximum submatrix sum
      with N^3 complexity
    - Output the computed result, execution time, and other
      statistics
    - Call max_matrix_of_N4(a) to compute the maximum submatrix sum
      with N^4 complexity
    - Output the computed result, execution time, and other
      statistics

7. End of the program
```

O(N^6) Algorithm

```

    pseudocode of function max_matrix_of_N6(matrix s):
    max = 0;
        // Find the maximum submatrix sum using N^6 complexity
        for i from 0 to N-1:
            for j from 0 to N-1:
                for k from i to N-1:
                    for l from j to N-1:
                        sum = 0
                        for m from i to k:
                            for n from j to l:
                                sum += b[m][n]
                                //Make 'sum' the sum of the submatrix
                                   from (i, j) to (k, l)

                        if sum > max:max = sum
    return max
```

Analysis:
The algorithm is a brute force algorithm that iterates through all possible submatrices and calculates their sums.
The algorithm employs six nested loops to iterate through all potential submatrices and calculate their sums. Thus ,it has a time complexity of $O(N^6)$ and a space complexity of $O(N^2)$.This exhaustive search guarantees that no submatrix is left unexamined, ensuring the discovery of the maximum submatrix sum.

O(N^4) Algorithm

```

    pseudocode of function max_matrix_of_N4(matrix s):

        // Copy the input matrix to a temporary matrix b
```

```
        b = copy_matrix(s)

        // Calculate prefix sums in the horizontal direction
        for i from 0 to N-1:
            for j from 1 to N-1:
                b[i][j] = b[i][j] + b[i][j - 1]

        // Find the maximum submatrix sum using N^4 complexity
        for i from 0 to N-1:
            for j from 0 to N-1:
                for k from i to N-1:
                    sum = 0
                    for l from j to N-1:
                        sum += (b[l][k] - b[l][i] + s[l][i])
                        //translates the 2 dimentions array to 1

                        if sum > max:
                            max = sum

    return max
```

Analysis:
In the `max_matrix_of_N4` function, the algorithm efficiently explores all possible starting and ending rows and columns of submatrices within the given matrix s to calculate the maximum submatrix sum with a time complexity of $O(N^4)$. The key to achieving this efficiency is the use of dynamic programming to precompute and store the cumulative sums of elements in the horizontal direction. This allows the algorithm to calculate submatrix sums in constant time, rather than recomputing them from scratch for each submatrix.
Here's the specific process:
1. First, determine the number of iterations based on the matrix size N.
2. Create a copy of the input matrix s as a temporary matrix b.
3. Calculate prefix sums for each row of b in the horizontal direction. This is done using a single loop that iterates through the columns of b. The prefix sum $b[i][j]$ at each cell (i, j) represents the cumulative sum of elements in row i from column 0 to j:

$$b[i][j] = \sum_{k=0}^{j} s[i][k]$$

4. To explore submatrices efficiently, two nested loops iterate over all possible starting and ending rows, i and k, respectively. These loops define the top and bottom rows of the submatrix.
5. Within these nested loops, there is another set of nested loops iterating over all possible starting and ending columns, j and l, respectively. These loops define the left and right columns of the submatrix.
6. For each combination of starting and ending rows and columns, the algorithm efficiently calculates the sum of elements within the submatrix using the precomputed prefix sums:
$sum = b[l][k] - b[l][i] + s[l][i]$
This calculation exploits the prefix sums to compute the sum of elements within the submatrix in constant time, avoiding the need for nested loops to sum individual elements.
7. The maximum submatrix sum, max, is updated whenever a greater sum is found.
8. The algorithm continues to iterate through all possible submatrices, ensuring that no submatrix is overlooked.
9. After all iterations, the maximum submatrix sum is found, and it is printed as the result.

This approach guarantees that the algorithm explores all submatrices efficiently, leveraging dynamic programming to optimize the computation of submatrix sums.

O(N^3) Algorithm

```

        pseudocode of function max_matrix_of_N3(matrix s):

        // Copy the input matrix to a temporary matrix b
        b = copy_matrix(s)

        // Calculate prefix sums in the horizontal direction
        for i from 0 to N-1:
            for j from 1 to N-1:
                b[i][j] = b[i][j] + b[i][j - 1]

        // Find the maximum submatrix sum using N^3 complexity
        for k from 0 to N-1:
            for j from k to N-1:
                sum = 0
                for i from 0 to N-1:
                        sum0 = b[i][j] - b[i][k] + s[i][k];
                        sum+=sum0;
                        if sum > max:
                            max = sum
                        else if sum < 0
                            sum = 0
    return max
```

The overall implementation approach of the N^3 function is similar to that of N^4. Specifically, the addition of a mechanism that sets 'sum' to zero if it becomes negative , it focus on optimizing the process of confirming the submatrix , which allows for a further reduction in the level of nested loops

Chapter 3 : Performance Testing

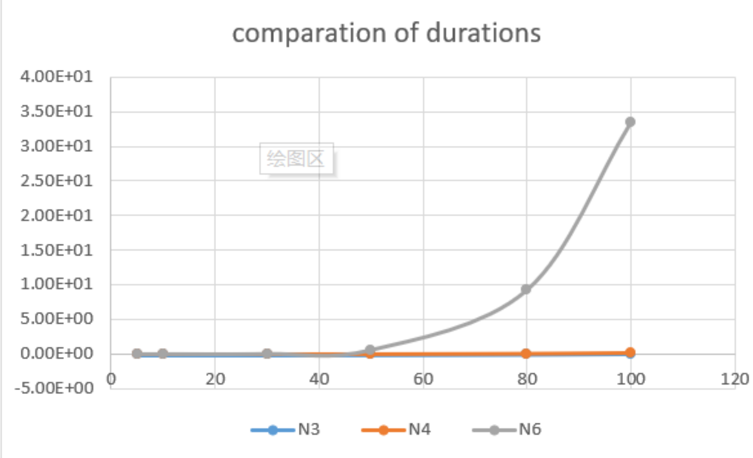
To measure the performance of three algorithms, we conducted a series of tests for different N values: 5, 10, 30, 50, 80, and 100. We used C's standard library time.h to record execution times.And every element in the matrix generated is an integer between -100 and +100,aiming to guarantee the max-sum in the range of type int .
When N is relatively small ,it's difficlult to test the exact durations.Thus,so I made them run K times and averaging them to get the accurate duration.
Here is the chart of times:

An illustration of the run time chart is also added below.

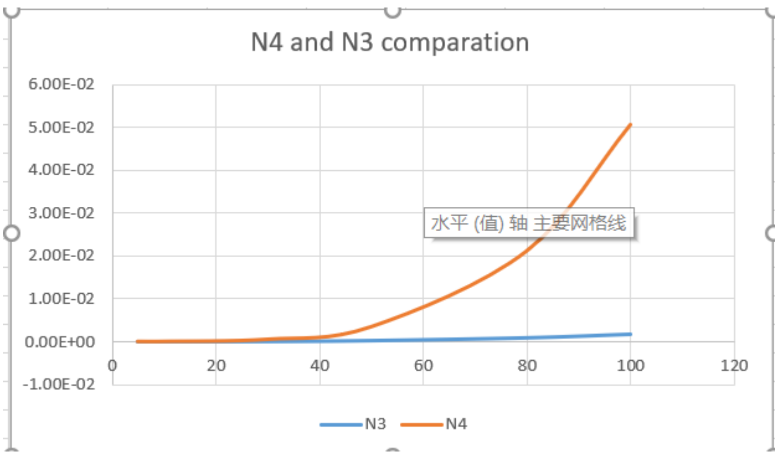
Table 1: Sample Table

	N	5	10	30	50	80	100
O(N^3)	Iterations(K)	1000000	1000000	100000	10000	10000	1000
	Ticks	345	2313	4655	2526	9116	1747
	Duration(sec)	3.45e-007	2.31e-006	4.66e-005	2.53e-004	9.12e-004	1.75e-003
	Total time(sec)	0.345000	2.313000	4.655000	2.526000	9.116000	1.747000
	RET(sec)	2.76e-009	2.31e-009	1.72e-009	2.02e-009	1.78e-009	1.75e-009
O(N^4)	Iterations(K)	1000000	100000	10000	1000	1000	100
	Ticks	779	878	5045	3435	21159	5081
	Duration(sec)	7.79e-007	8.78e-006	5.04e-004	3.44e-003	2.12e-002	5.08e-002
	Total time(sec)	0.779000	0.878000	5.045000	3.435000	21.159000	5.081000
	RET(sec)	1.25e-010	8.78e-010	6.23e-010	5.50e-010	5.17e-010	5.08e-010
O(N^6)	Iterations(K)	1000000	100000	1000	10	10	1
	Ticks	2510	7796	32035	5907	92478	33422
	Duration(sec)	2.51e-006	7.80e-005	3.20e-002	5.91e-001	9.25e+000	3.34e+001
	Total time(sec)	2.510000	7.796000	32.035000	5.907000	92.478000	33.422000
	RET(sec)	1.61e-010	7.80e-011	4.39e-011	3.78e-011	3.53e-011	3.34e-011

duration of three functions



The chart shows that maxmatrixofN6 cost much more time when N grows. However,it's difficult for us to distinguish between maxmatrixofN4 and maxmatrixofN3,since both Algorithm can run rapidly while the maximum N is 100. To make the testing results more clear, I make an another illustartion which only contains the perforamance of maxmatrixofN4 and maxmatrixofN3. The illustration is shown below.



Now the whole results are clear, maxmatrixofN4 runs truly slower than maxmatrixofN3.And when grows larger, the difference become even more unbelievable,that means, when is sufficiently large, the advantage of maxmatrixofN3 can be shown more clear.

Chapter 4 : Analysis and Comments

O(N^6)

Space Complexity: O(N^2)
A temporary matrix b of size N * N is created to store the copy of the input matrix s.And the matrix s itself uses size N * N. Integer variables max, sum, i, j, k, l, m, n, and man do not contribute significantly to space complexity, as they use constant space.
Thus , the Algorithm ought to runs O(N^6)(analyze in chapter 2) in and has O(N^2) memory. Analysis the data and verify the time complexity.If it truly runs in O(N^6), then $\frac{t}{N^6}$ ought to be roughly an constant
It seems that there is a mistake, because when N is small, the result's error can't be ignored. But when we turn to larger N , the error seems quite small. The reason causes the big error when is relatively small is that the time of reading the datas remains O(1) ,and this time cannot be ignored when N is small.
However, When N begins to grow larger, $\frac{t}{N^6}$ seems to be stable.Hence,we can assume that the Algorithm runs in O(N^6).

Table 2: TIME COMPLEXITY EVALUATION 1

N	5	10	30	50	80	100
Duration(sec)	2.51e-006	7.80e-005	3.20e-002	5.91e-001	9.25e+000	3.34e+001
RET(sec)	1.61e-010	7.80e-011	4.39e-011	3.78e-011	3.53e-011	3.34e-011

O(N^4)

Space Complexity: O(N^2)
A temporary matrix b of size N * N is created to store the copy of the input matrix s.And the matrix s itself uses size N * N. Integer variables max, sum, i, j, k, l, m, n, and man do not contribute significantly to space complexity, as they use constant space.
Thus the Algorithm ought to runs in O(N^4) and has O(N^2) memory. Analysis the data and verify the time complexity.If it truly runs in O(N^4), then $\frac{t}{N^4}$ ought to be roughly an constant
When N begins to grow larger, $\frac{t}{N^4}$ seems to be stable.Hence,we can assume that the Algorithm runs in O(N^4).

Table 3: TIME COMPLEXITY EVALUATION

N	5	10	30	50	80	100
Duration(sec)	7.79e-007	8.78e-006	5.04e-004	3.44e-003	2.12e-002	5.08e-002
RET(sec)	1.25e-010	8.78e-010	6.23e-010	5.50e-010	5.17e-010	5.08e-010

O(N^3)

Space Complexity: O(N^2)
A temporary matrix b of size N * N is created to store the copy of the input matrix s.And the

matrix s itself uses size N * N. Integer variables max, sum, i, j, k, l, m, n, and man do not contribute significantly to space complexity, as they use constant space.
Thus , the Algorithm ought to runs O(N^3) in and has O(N^2) memory. Analysis the data and verify the time complexity.If it truly runs in O(N^3), then $\frac{t}{N^3}$ ought to be roughly an constant

When N begins to grow larger, $\frac{t}{N^3}$ seems to be stable.Hence,we can assume that the Algorithm runs in O(N^3).

Table 4: TIME COMPLEXITY EVALUATION

N	5	10	30	50	80	100
Duration(sec)	3.45e-007	2.31e-006	4.66e-005	2.53e-004	9.12e-004	1.75e-003
RET(sec)	2.76e-009	2.31e-009	1.72e-009	2.02e-009	1.78e-009	1.75e-009

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

clock_t start, stop;
double duration;
int a[100][100];
int N; // Size of the matrix (N*N)
long long num = 0;

// Function to find the maximum submatrix sum for N^4
   complexity
int max_matrix_of_N4(int s[100][100]) {
    int num_of_iteration;

    // Determine the number of iterations based on the matrix
       size N
    if (N == 5)
        num_of_iteration = 1000000;
    else if (N == 10)
        num_of_iteration = 100000;
    else if (N == 30)
        num_of_iteration = 10000;
    else if (N == 50)
        num_of_iteration = 1000;
    else if (N == 80)
        num_of_iteration = 1000;
    else if (N == 100)
        num_of_iteration = 100;

    int i, j, k, l;
    int max = 0;
    int sum = 0;
```

```
    int b[100][100];
    int man = num_of_iteration;

    while (num_of_iteration--) {
        // Copy the input matrix to b
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                b[i][j] = s[i][j];
            }
        }

        // Calculate prefix sums in the horizontal direction
        for (i = 0; i < N; i++) {
            for (j = 1; j < N; j++) {
                b[i][j] = b[i][j] + b[i][j - 1];
            }
        }

        // Find the maximum submatrix sum using N^4 complexity
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                for (k = i; k < N; k++) {
                    sum = 0;
                    for (l = j; l < N; l++) {
                        // Calculate the sum efficiently
                        sum += (b[l][k] - b[l][i] + s[l][i]);
                        num++;
                        if (sum > max) {
                            max = sum;
                        }
                    }
                }
            }
        }
    }
    printf("Maximum submatrix sum with N^4 complexity: %d\n",
       max);

    return man; // Return the number of iterations performed
}

// Function to find the maximum submatrix sum for N^6
   complexity
int max_matrix_of_N6(int s[100][100]) {
    int num_of_iteration;

    // Determine the number of iterations based on the matrix
       size N
    if (N == 5)
        num_of_iteration = 1000000;
    else if (N == 10)
        num_of_iteration = 100000;
```

```
        else if (N == 30)
            num_of_iteration = 1000;
        else if (N == 50)
            num_of_iteration = 10;
        else if (N == 80)
            num_of_iteration = 10;
        else if (N == 100)
            num_of_iteration = 1;

        int i, j, k, l, m, n;
        int max = 0;
        int sum = 0;
        int b[100][100];
        int man = num_of_iteration;

        while (num_of_iteration--) {
            // Copy the input matrix to b
            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    b[i][j] = s[i][j];
                }
            }

            // Find the maximum submatrix sum using N^6 complexity
            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    for (k = i; k < N; k++) {
                        for (l = j; l < N; l++) {
                            sum = 0;
                            for (m = i; m <= k; m++) {
                                for (n = j; n <= l; n++) {
                                    // Calculate the sum
                                       efficiently
                                    sum += b[m][n];
                                    num++;
                                }
                            }
                            if (sum > max) {
                                max = sum;
                            }
                        }
                    }
                }
            }
        }
        printf("Maximum submatrix sum with N^6 complexity: %d\n",
           max);

        return man; // Return the number of iterations performed
    }

    // Function to find the maximum submatrix sum for N^3
```

```
    complexity
int max_matrix_of_N3(int s[100][100]) {
    int num_of_iteration;

    // Determine the number of iterations based on the matrix
       size N
    if (N == 5)
        num_of_iteration = 1000000;
    else if (N == 10)
        num_of_iteration = 1000000;
    else if (N == 30)
        num_of_iteration = 100000;
    else if (N == 50)
        num_of_iteration = 10000;
    else if (N == 80)
        num_of_iteration = 10000;
    else if (N == 100)
        num_of_iteration = 1000;

    int i, j, k, l;
    int max = 0;
    int sum = 0;
    int sum0 = 0;
    int b[100][100];
    int man = num_of_iteration;

    while (num_of_iteration--) {
        // Copy the input matrix to b
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                b[i][j] = s[i][j];
            }
        }

        // Calculate prefix sums in the horizontal direction
        for (i = 0; i < N; i++) {
            for (j = 1; j < N; j++) {
                b[i][j] = b[i][j] + b[i][j - 1];
            }
        }

        // Find the maximum submatrix sum using N^3 complexity
        for (k = 0; k < N; k++) {
            sum = 0;
            for (j = k; j < N; j++) {
                sum = 0;
                for (i = 0; i < N; i++) {
                    if (j == k) {
                        sum0 = s[i][j];
                    } else {
                        sum0 = b[i][j] - b[i][k] + s[i][k];
                    }
```

```
                        num++;
                        sum += sum0;
                        if (sum > max) {
                            max = sum;
                        } else if (sum < 0) {
                            sum = 0;
                        }
                    }
                }
            }
        }
        printf("Maximum submatrix sum with N^3 complexity: %d\n",
           max);

        return man; // Return the number of iterations performed
    }

    void generateMatrix(int n) {
        srand(time(NULL));

        // Generate the matrix
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                a[i][j] = rand() % 201 - 100; // Generate a random
                   number between -100 and 100
            }
        }
    }

    int main() {
        int i, j;

            scanf("%d", &N); // Read the size of the matrix (N*N)

            generateMatrix(N);

            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    printf("%d ", a[i][j]); // Read the elements of
                       the matrix
                }
                printf("\n");
            }

            start = clock();

            // Call the functions to find maximum submatrix sums
            int m = max_matrix_of_N6(a); // Set m to the number of
               iterations

            stop = clock();
            duration = ((double)(stop - start)) / CLOCKS_PER_SEC;
```

```
            clock_t ticks = stop - start;

            printf("The duration of N6 function is %.2e s\n", duration / m);
            printf("The Total time of N6 function is %f s\n", duration);
            printf("The ticks of N6 function is %ld ms\n", (long) ticks);
            printf("The ret of N6 function is %.2e s\n", duration / m / pow(N, 6)); // The time of each iteration

            start = clock();
            m = max_matrix_of_N3(a);
            stop = clock();

            duration = ((double)(stop - start)) / CLOCKS_PER_SEC;
            ticks = stop - start;

            printf("The duration of N3 function is %.2e s\n", duration / m);
            printf("The Total time of N3 function is %f s\n", duration);
            printf("The ticks of N3 function is %ld ms\n", (long) ticks);
            printf("The ret of N3 function is %.2e s\n", duration / m / pow(N, 3));

            start = clock();
            m = max_matrix_of_N4(a);
            stop = clock();

            duration = ((double)(stop - start)) / CLOCKS_PER_SEC;
            ticks = stop - start;

            printf("The time of N4 function is %.2e s\n", duration / m);
            printf("The Total time of N4 function is %f s\n", duration);
            printf("The ticks of N4 function is %ld ms\n", (long) ticks);
            printf("The ret of N4 function is %.2e s\n", duration / m / pow(N, 4));
        }

        system("pause");
        return 0;
```

Declaration

I hereby declare that all the work done in this project titled ”Maximum Submatrix Sum Problem” is of my independent effort.