Hard Performance Measurement(MSS)

2023-10-7

Chapter 1: Introduction

Given an *N*×*N* integer matrix $((a_{ij})_{N} *_{N})$, find the maximum value of $\sum_{k=i}^{m}\sum_{l=j}^{n}a_{kl}$ for all $1 \le i \le m \le N$ and $1 \le j \le n \le N$. For convenience, the maximum submatrix sum is 0 if all the integers are negative.

To complete the algorithm 2 and algorithm 3, I learn the prefix and. This way can use $O(N^2)$ to get the add of continuous lines. Using this way can decrease the complexity into $O(N^2)$. And after the traversal, the complexity is $O(N^4)$ in algorithm 2.

1 temp[0]
12 temp[1]
123 temp[2]
1234 temp[3]
2 temp[1]-temp[0]
23 temp[2]-temp[0]
234 temp[3]-temp[0]
3 temp[2]-temp[1]
34 temp[3]-temp[1]
4 temp[3]-temp[2]
These are my ways to analyze how to write the circulation.

Also, I learn a useful and wonderful algorithm, which can get maximum subsequence sum in $O(N)$. The algorithm use a magic way to deal with number, which is that when this_sum<0, this_sum comes to 0, and when this_sum>0, this_sum += the next number. Using this and prefix and can make the time complexity into $O(N^3)$.

Chapter 2: Algorithm Specification

Algorithm 1 ($O(N^6)$)

The algorithm is similar to which in Section 2.4.3. I think its principle is easy but it's extremely complex. First, the first two circulations are used to record the start position in the matrix. Then, the next two circulations are used to record the end position. Until now the time complexity is $O(N^4)$. Finally, the last two circulations are used to add the number into `this_sum`. And if `this_sum`> `max_sum`, I change the number and note the position.

The key point is use $O(N^4)$ to determine the start position and end position and use $O(N^2)$ to add the numbers.

Algorithm 2 ($O(N^4)$)

The algorithm uses prefix and to simplify the algorithm. First, I use two functions whose complexity is $O(N^2)$ to calculate the prefix and of each line. Then, as the thinking process in chapter 1, I use a $O(N^2)$ circulation to calculate the add of continuous lines. Finally, similar to up, I use $O(N^2)$ circulation to calculate and compare the result.

Algorithm 3($O(N^3)$)

The algorithm is similar to last one, which also uses prefix and. The differences between the two algorithms is when comparing the columns, I use a special way. It is that when this_sum<0, this_sum comes to 0, and when this_sum>0, this_sum += the next number. Using this and prefix and can make the time complexity into $O(N^3)$.

Extra complement

All the algorithms are not using special data structures except array.

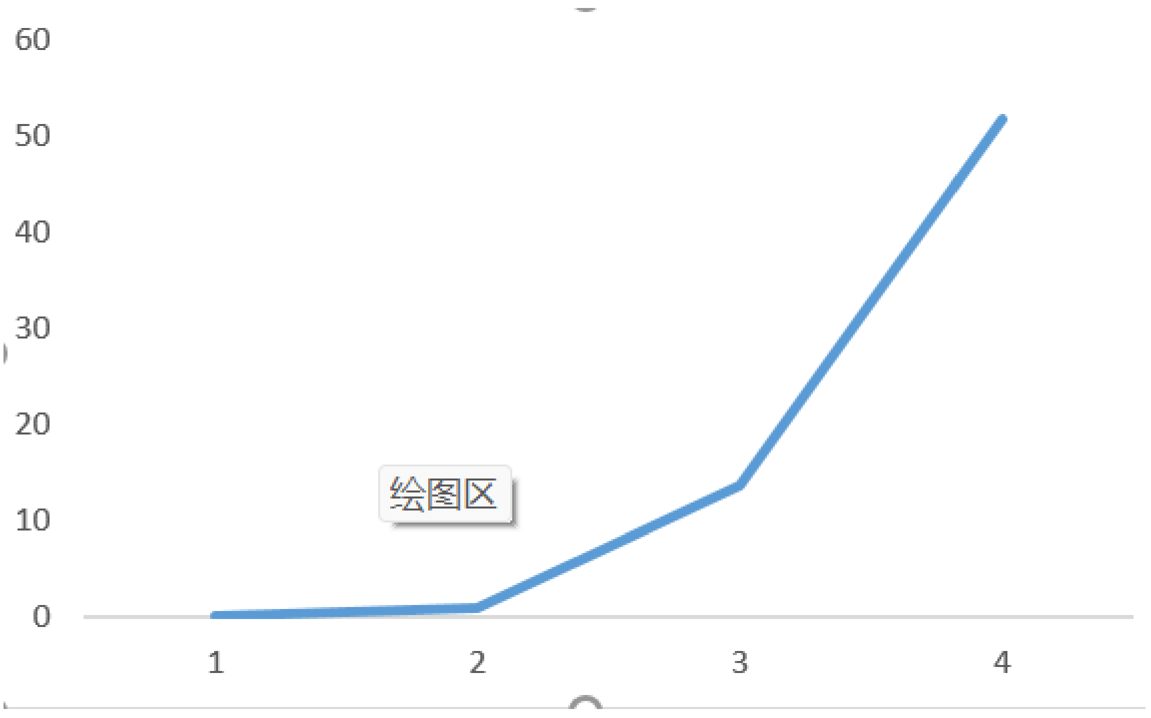
All the space complexity are $O(N^2)$.

Chapter 3: Testing Results

	N	5	10	30	50	80	100
O(N^6)	Iterations(K)	5000	2000	100	10	10	1
O(N^6)	Ticks	11	165	4218	8649	135782	51850
O(N^6)	Total Time(sec)	0.011	0.165	4.218	8.649	135.782	51.85
O(N^6)	Duration(sec)	$2.2 * 10^-6$	$8.25 * 10^-5$	0.04218	0.8649	13.5872	51.85
O(N^4)	Iterations(K)	5000	2000	100	10	10	1
O(N^4)	Ticks	5	27	98	71	333	91
O(N^4)	Total Time(sec)	0.005	0.027	0.098	0.071	0.333	0.091
O(N^4)	Duration(sec)	$1 * 10^-6$	$1.35 * 10^-5$	$9.8 * 10^-4$	$7.1 * 10^-3$	0.0333	0.091
O(N^3)	Iterations(K)	5000	2000	100	10	10	1
O(N^3)	Ticks	1	4	4	1	4	1
O(N^3)	Total Time(sec)	0.001	0.004	0.004	0.001	0.004	0.001
O(N^3)	Duration(sec)	$2 * 10^-7$	$2 * 10^-6$	$4 * 10^-5$	$1 * 10^-4$	$4 * 10^-4$	$1 * 10^-3$

Chapter 4: Analysis and Comments

Through the table, it's clear that the time of complete the algorithm have very big difference. I learn that simplifying the algorithm is very important when N is big and implement times are huge.



Similar to the graph, when N increases, the time used in programming increased rapidly, similar to e^x.

Appendix: Source Code (in C)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
clock_t start, stop;
double duration, ticks;
int n = 99;    // give the length of matrix
int total = 3; // the repeat times
int a[100][100];
void init_matrix()
{
    int i, j;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            a[i][j] = rand() % 10;
        }
    }
}
int maxinumsubmatrix_6()
{
    int this_sum, max_sum = 0, best_i1, best_i2, best_j1, best_j2, i1, i2, j1,
j2, k1, k2; // define variables
    for (i1 = 0; i1 < n; i1++)
    {
        for (j1 = 0; j1 < n; j1++) // the circulation of the start row and
column
        {
            for (i2 = i1; i2 < n; i2++)
            {
                for (j2 = j1; j2 < n; j2++) // the circulation of the end row
and column
                {
                    this_sum = 0; // clear this_sum and start adding
                    for (k1 = i1; k1 <= i2; k1++)
                    {
                        for (k2 = j1; k2 <= j2; k2++) // culculate these number
in the matrix
                        {
                            this_sum += a[k1][k2];
                            if (this_sum > max_sum) // record the result
                            {
                                max_sum = this_sum;
                                best_i1 = i1;
                                best_i2 = i2;
                                best_j1 = j1;
                                best_j2 = j2;
                            }
                        }
                    }
                }
            }
        }
    }
    return max_sum;
}
int maxinumsubmatrix_4()
{
    int temp[100][100], result[4] = {0, 0, 0, 0}; // use temp[4][4] to record
prefix and; use result[4] to record submatrix
    int this_sum, max_sum = 0, best_i1, best_i2, best_j1, best_j2, i1, i_, j1,
j2, f = 0;
    for (i1 = 0; i1 < n; i1++)
    {
        for (j1 = 0; j1 < n; j1++)
        {
            temp[i1][j1] = a[i1][j1]; // copy a matrix to keep prefix and
        }
    }
    for (i1 = 1; i1 < n; i1++)
    {
        for (j1 = 0; j1 < n; j1++)
        {
            temp[i1][j1] += temp[i1 - 1][j1]; // complete culculating prefix and
        }
    }
    for (i1 = -1; i1 < n - 1; i1++)
    {
        for (i_ = i1 + 1; i_ < n; i_++) // these circulation calculate the add
of lines by prefix and
        {
            for (j1 = 0; j1 < n; j1++) // start evaluating the add of column
            {
                for (f = 0; f < n; f++)
                {
                    result[f] = 0;
                }
                this_sum = 0;               // clear the sum
                for (j2 = j1; j2 < n; j2++) // culculate the add of submatrix
sum
                {
                    if (i1 == -1)
                    {
                        result[j2] = temp[i_][j2];
                    }
                    else
                    {
                        result[j2] = temp[i_][j2] - temp[i1][j2];
                    }
                    this_sum += result[j2]; // use this_sum to keep the answer
                    if (this_sum > max_sum) // record the result
                    {
                        max_sum = this_sum;
                        best_i1 = i1 + 1;
                        best_i2 = i_;
                        best_j1 = j1;
                        best_j2 = j2;
                    }
                }
            }
        }
    }
    return max_sum;
}
int maxinumsubmatrix_3()
{
    int temp[100][100]; // use temp[4][4] to record prefix and
    int this_sum, max_sum = 0, best_i1, best_i2, i1, i_, j1;
    for (i1 = 0; i1 < n; i1++)
    {
        for (j1 = 0; j1 < n; j1++)
        {
            temp[i1][j1] = a[i1][j1]; // copy a matrix to keep prefix and
        }
    }
    for (i1 = 1; i1 < n; i1++)
    {
        for (j1 = 0; j1 < n; j1++)
        {
            temp[i1][j1] += temp[i1 - 1][j1]; // complete culculating prefix and
        }
    }
    for (i1 = -1; i1 < n - 1; i1++)
    {
        for (i_ = i1 + 1; i_ < n; i_++) // these circulation calculate the add
of lines by prefix and
        {
            this_sum = 0; // init this_sum
            for (j1 = 0; j1 < n; j1++)
            {
                if (i1 == -1) // if i1==-1,not need to subtract last line
                {
                    if (this_sum < 0)
                    {
                        this_sum = temp[i_][j1]; // if this_sum<0, update
this_sum
                    }
                    else
                    {
                        this_sum += temp[i_][j1]; // if this_sum>0, the sum will
increase, so still add
                    }
                }
                else // if i1!=-1, subtract temp[i1][j1] to get the lines
                {
                    if (this_sum < 0)
                    {
                        this_sum = temp[i_][j1] - temp[i1][j1]; // if
this_sum<0, update this_sum
                    }
                    else
                    {
                        this_sum = this_sum + temp[i_][j1] - temp[i1][j1]; // if
this_sum>0, the sum will increase, so still add
                    }
                }
                if (this_sum > max_sum) // record the result
                {
                    max_sum = this_sum;
                    best_i1 = i1 + 1;
                    best_i2 = i_;
                }
            }
        }
    }
    return max_sum;
}
int main()
{
    init_matrix(); // use random number to generate matrix
    start = clock();
    int i;
    for (i = 0; i < total; i++) // implement the function for total times
    {
        // maxinumsubmatrix_6();
        // maxinumsubmatrix_4();
        // maxinumsubmatrix_3();
    }
    stop = clock();
    printf("Algorithm 2: n = %d, times = %d, ", n, total); // help print the
answer
    ticks = ((double)(stop - start));
    duration = ((double)(stop - start)) / CLK_TCK;
    printf("ticks = %f, time = %f seconds", ticks, duration);
    return 0;
```

```
}
```

Declaration

I hereby declare that all the work done in this project titled "MAXIMUM SUBMATRIX SUM" is of my independent effort.