Analysis of Algorithms

Project Report

COT 5045 FALL 2020

Yuze zhu | UFID: 75087354| 12/1/20

1. Programming methods

There are 3 algorithms to complete AlgoTowers.

- ALG1 uses dynamic programming: for a position (x, y), there exist a required square block with side length l, if for position (x, y), (x, y+1), (x+1, y), and (x+1, y+1), there exist a with side length l-1.
- ALG2 uses enumeration method: enumerate all submatrix of the given matrix, and test whether each submatrix satisfies the conditions and whether its size is the biggest among the satisfied.
- ALG3 uses dynamic programming: for a rectangle area at position (x, y), with width a, height b, if for position (x, y+1) there exist a rectangle area with width a, height c, then there exist a width a+1, height b>c?c:b rectangle area at position (x, y). In a nutshell, width problem could be decomposed into width subproblems.

TIME COMPLEXITY ANALYSIS

- ALG1: in a subproblem with side length a, a traversal of O(mn) matrix is needed to compose the next subproblem with side length a+1. And the traversal should be taken k times, where k is the side length of possible result squares. Hence the complexity is O(mn)
- ALG2: for a mxn matrix, there are mxn(O(mn)) top-leftist point choices and O(mn) bottom-rightist point choices to form a submatrix. And for a submatrix, a traversal with O(mn) is needed to check if it satisfies the conditions. Hence the complexity is O(m^3n^3).
- ALG3: for every column(m in total), an array record is maintained to note the max-height below which all elements all elements satisfy the conditions. And on this array(n in total), traversal with “slide” operation could be made to check if there exists a matrix with width w, and it could use 2 subproblems with width w-1. And the loop should stop in problem of width k which has a sub-problem of width k-1 where there is less than 2 matrices. Hence the complexity is O(mn).

2. Task details and space analysis

Pre: for all tasks, an update function is set in the most inside layer of loops, to update the current largest square/rectangle, which is not repeatedly stated below.

1. Task 1: a mxn matrix is needed to record whether a square satisfies the conditions taking this position as top-leftist of square submatrix. And loop in order to make it record whether (x, y), (x+1, y), (x, y+1), (x+1, y+1) all satisfy the last subproblem, until there is less than 4 “true” in a loop. Hence O(mn) extra space is required.

```
bool** mem = new bool*[m];
for (int i = 0; i < m; i++) {
    mem[i] = new bool[n];
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] >= h) mem[i][j] = true;
        else mem[i][j] = false;
    }
}
int count = 0, side = 1, currentSide = 0;
for (int k = 0; k < m; k++) {
    for (int i = 0; i < m - side; i++) {
        for (int j = 0; j < n - side; j++) {
            mem[i][j] = mem[i][j] && mem[i][j+1] && mem[i+1][j] && mem[i+1][j+1];
            if (mem[i][j]) {
                count++;
                |
            }
        }
    }
    if (count < 4) break;
    side++;
    count = 0;
}
```

2. Task 2: an n array is needed to record a height, below which above current position, all elements satisfy the condition. And to check if there exists a required square with width w, height records could be used to quickly judge if it is possible (w < height). And checks of (x, y), (x-1, y), (x, y+1), (x-1, y+1) for square with w-1 (subproblem) are needed. In every loop, width is added by 1, and loop is stopped where there is less than 2 in current-line or last-line subproblems. Hence O(n) extra space is required

```
int* mem = new int[n];
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] >= h) mem[j] += 1;
        else mem[j] = 0;
    }
    int counter = 0;
    for (int tside = 0; tside < n; tside++) {
        for (int j = 0; j < n - tside; j++) {
            bool fit = true;
            for (int k = 0; k < tside + 1; k++) {
                fit = fit && (mem[j+k] > tside);
            }

        }
        if (counter < 2) break;
        counter = 0;
    }
}
delete[] mem;
return results;
```

3. Task 3: Just Loop mxn times to choose top-leftist point of submatrix, and then loop O(mn) times to choose bottom-rightist point. Then one bool is required to check if all elements of submatrix satisfy the condition. Hence O(1) extra space is required.

```
bool mark;
int size = 0;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        for (int x = i; x < m; x++) {
            for (int y = j; y < n; y++) {
                mark = true;
                for (int a = i; a <= x; a++) {
                    for (int b = j; b <= y; b++) {
                        mark = mark && (matrix[a][b] >= h);
                        if (!mark) {
                            break;
                            break;
                        }
                    }
                }

            }
        }
    }
}
return results;
```

4. Task 4: an mxn array is needed to record a height (or width), below which above corresponding position in original matrix, all elements satisfiy the condition. Then the problem changed to finding the largest possible rectangle on an axis with values. Loop with increasing width w in a row, from left to right, for each element choose the less one of itself and its right element as its new value. In this operation, the value of a position stands for a max up-right rectangle height with loop width w. And the loop should stop when there is less than 2 values with positive integers.

```
int** mem = new int*[m];
for (int i = 0; i < m; i++) {
    mem[i] = new int[n];
    int flag = 0;
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] >= h) {
            mem[i][j] = j - flag + 1;

        } else {
            flag = j + 1;
            mem[i][j] = 0;
        }
    }
}
int count = 0;
for (int height = 1; height < m; height++) {
    for (int i = 0; i < m - height; i++) {
        for (int j = 0; j < n; j++) {
            mem[i][j] = (mem[i][j] < mem[i+1][j])?mem[i][j]:mem[i+1][j];
            int tsize = mem[i][j] * (height + 1);
            if (tsize > 0) count++;

        }
    }
    if (count < 2) break;
    count = 0;
}
return results;
```

5. **Task 5: an n array is needed to record a height**, below which above current position, all elements satisfy the condition. And for each row, **copy that n array to operate**. The operating method is similar to that in task 4. **However only 2 n-array space is required. (require O(n) extra space).**

```
int* mem = new int[n];
for (int i = 0; i < n; i++) mem[i] = 0;
int* temp = new int[n];
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (matrix[i][j] >= h) mem[j] += 1;
        else mem[j] = 0;
        temp[j] = mem[j];

    }
    int counter = 0;
    for (int width = 1; width < n; width++) {
        for (int j = 0; j < n - width; j++) {
            temp[j] = (temp[j] < temp[j+1])?temp[j]:temp[j+1];
            if (temp[j] > 0) counter++;
            int tsize = (width + 1) * temp[j];

        }
        if (counter < 2) break;
        counter = 0;
    }
}
delete[] mem;
delete[] temp;
return results;
```

3. Substructure correctness argument

For square:

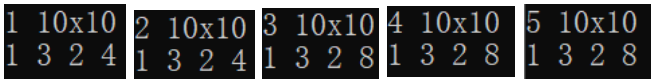
If a square with side s at (x, y) exists, then 4 squares with side s-1 at (x, y), (x, y+1), (x+1, y), (x+1, y+1) must exist, which is a subproblem.

For rectangle:

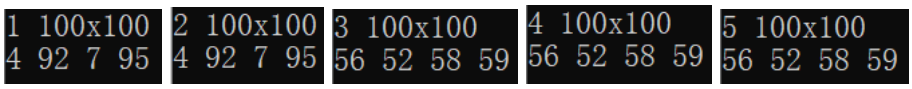
If a rectangle with width w, height h at (x, y) exists, then 2 rectangles with width w-1 at (x, y), (x+1, y) must exist, and their height must >= h.

4. Results for test cases

For the given 10x10 case:



100x100 case:



1000x1000case: (Task 3 is too long to wait for).

5. Experimental comparative study

I used the provided python code to generate several square matrices as input. All data has a scale between 0 to 10, and threshold is set 3 in order to make less variable. And I use ctime lib to record time used for a kind of operation. And I got table below. (unit: cloct_t unit)

Input size	Task 1	Task 2	Task 3	Task 4	Task 5
10x10	0	0	0	0	0
20x20	0	0	3	0	0
30x30	0	0	31	0	0
40x40	0	0	165	0	0
60x60	0	0	1795	0	0
80x80	1	0	9792	1	0
100x100	1	1	36460	1	1
500x500	10	7	null	44	18
1000x1000	39	23	null	189	75
2000x2000	170	105	null	902	353

As the table shows, first of all It's quite obvious that ALG2-Task3 is absolutely deficient. Its running time grows so rapidly that I have to abandon it early when size reaches 100x100.

And since all rest tasks are O(mn), their running time does not rise fast. And according to the square-shape data size I chose where m = n, here we can easily observe that all rest tasks are O(n^2): from 500x500 to 1000x1000, from 1000x1000 to 2000x2000, running time grows around 4 times larger.

Additionally, we can find that running time of task 2 is less than that of task 1, and running time of task 5 is less than that of task 4. One possible reason could be that with less extra space, memory operation could be faster. Besides, since all arrays in my program are pointer arrays, another important reason could be the allocation and release of memory.