Analysis of Algorithm Project

Yinuo Yang
UFID: 5725-3171

In this project, I use C++ to realize the 3 required algorithms and 5 tasks with required time complexity and memory. Further running time comparisons are also included to help understanding the time complexity.

ALG1 (task1&task2)

Definition: Create a table D with size $(m+1)\times(n+1)$. Let $D[i,j]$ denote the length of largest square area block whose bottom-right corner is at $cell(i-1,j-1)$ and all p inside the block is no less than h. For $i=0$ or $j=0$, $D[i,j]$ has no real meaning but to help building our algorithm, I'll let them equals to zero. Here is the recursive formulation expression:

$$D[i,j]=\begin{cases}0, & if\ i=0\ or\ j=0;\\ 0, & elseif\ p[i-1,j-1]<h;\\ \min\{D[i,j-1],D[i-1,j],D[i-1,j-1]\}+1, & else\ if\ p[i-1,j-1]\geq h;\end{cases}$$

Correctness proof: (1) For $i=0$ or $j=0$, trivial case; (2) Elseif $p[i-1,j-1]<h$, there's no block bottom-right cornered at $cell(i-1,j-1)$, thus $D[i,j]=0$ holds. (3) For $p[i-1,j-1]\geq h$, I'll use cut-and-paste strategy to prove the optimal solution to current problem must contain the optimal solution to three subproblems, which is also the minimum value in the equation, let's suppose the minimum value to be x. If given optimal solution $D[i,j]$, the calculated x is not from optimal solutions to subproblems, then there must exist a larger minimum value y from optimal solutions to subproblems, where $y>x$. For $cell(i-1,j-1)$, $D[i,j-1]$ covers area $cell(i-1,j-2)\,\&\,cell(i-1,j-1-y)\,\&\,cell(i-1-y,j-2)\,\&\,cell(i-1-y,j-1-y)$. $D[i-1,j]$ covers area $cell(i-2,j-1)\,\&\,cell(i-2,j-1-y)\,\&\,cell(i-1-y,j-1)\,\&\,cell(i-1-y,j-1-y)$. $D[i-1,j-1]$ covers area $cell(i-2,j-2)\&\,cell(i-2,j-1-y)\&cell(i-1-y,j-2)\&cell(i-1-y,j-2)\&\,cell(i-1-y,j-1-y)$. Considering $cell[i-1,j-1]$ is also covered, a square with side length $(y+1)$ is covered, which is a better solution to the original optimal solution $D[i,j]=x+1$, since $y>x$. This contradicts to our original condition, thus this optimal solution do contain the optimal solution to the subproblems. Proof done.

Analysis: In this algorithm, two for loop is used to traverse $i\in[0,m]\,\&\,j\in[0,n]$, so the time complexity is $O(mn)$. In **task1**, a table with size $(m+1)\times(n+1)$ is used, the space complexity is also $O(mn)$. In **task2**, I use a table with size $2\times(n+1)$ to only record the current row and previous row of $D[i,j]$, the previous row will be override if go to the next row. So the space complexity is $O(n)$

Comparison between task1 & task2: According to **Figure1**, both task1 and task2 running in linear time with the increase of input size ($m \times n$). This is in accordance with the time complexity of $O(mn)$ as we analyzed above. The fact that task2 faster than task1 on constant level may come from the details of code difference, but it does not affect they are both in same time complexity.

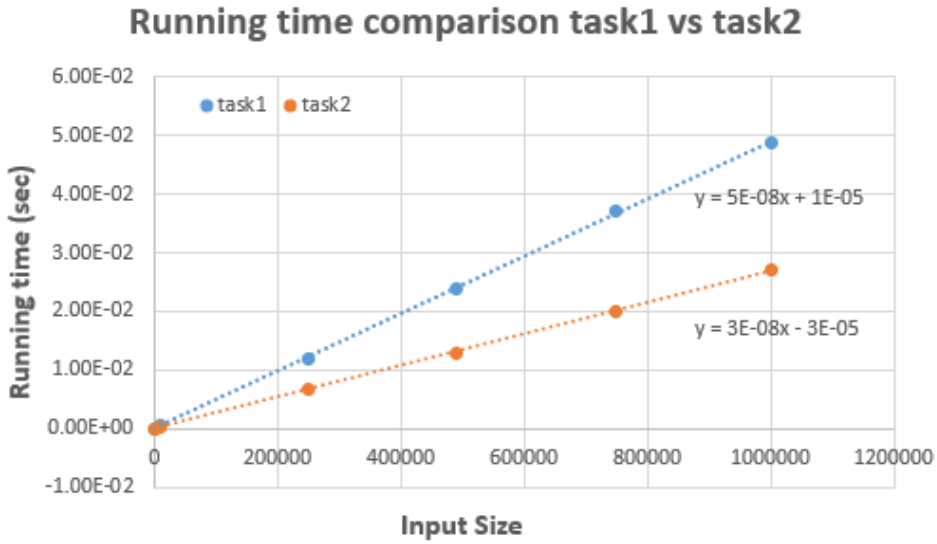

Figure 1. Running time comparison between task1 and task2

ALG2 (task3)

A simple bottom up brute force algorithm is applied to traverse all possible rectangles and record the largest area. In task3, six for loops are used to traverse all starting row i and column j and all possible ending row $i + a$ and column $j + b$. Check all the p value to decide if this rectangle satisfy the requirement that all $p \geq h$, if true record to area and finally get the maximum area. Therefore the time complexity is $O(m^3n^3)$ and takes $O(1)$ memory.

ALG3(task4 and task5)

Definition: Create a table H with size $m \times n$.Let $H[i,j]$ denote the number of consecutive cells inclusively above $cell(i,j)$ whose p value is no less than h. We can image it as the height of cells that can be potentially used to build the rectangle at $cell(i,j)$. We can use a dynamic programming to derive the table H, here is the recursive formulative expression:

$$H[i,j] = \begin{cases} 0, & if\ i = 0; \\ 0, & else\ if\ p[i,j] < h; \\ H[i-1,j] + 1, & else\ if\ p[i,j] \geq h; \end{cases}$$

Correctness proof: This proof is intuitive. Situation (1) and (2) are trivial. For situation (3) where $p[i,j] \geq h$, the optimal solution $H[i,j]$ to current problem must contain the optimal solution to subproblem $H[i-1,j]$ because all p values from $cell(i,j)$ above to $cell(i - H[i-1,j],j)$ are no less than h, so the satisfying height of cell is $H[i-1,j] + 1$, situation

(3) holds.

Further explanation: Giving this table H achieved by dynamic programming, we need to take a second step to get the area for maximum rectangle. Imagine each row $H[i]$ as an array with histogram plot at row i, we can intuitively think of the maximum rectangle area of this plot as the solution if the bottom line of the maximum rectangle is located at row i. In my code, I use a function called '*task5_hist_maxarea*' to realize this function. If we keep track of the maximum rectangle area while traversing all rows, we will get the optimal solution.

Analysis: For dynamic programming step1, it needs two for loop to traverse each cell with $i \in [0, m-1]$ & $j \in [0, n-1]$, so the time complexity is $O(mn)$. For second step2, a stack is used to smartly help store every starting point and ending point of potential rectangle. All it needs to do is traverse the row array once, so each time calling function '*task5_hist_maxarea*', it cost $O(n)$. Since we call this function for m times, in total step2 also take $O(mn)$. The total time complexity is $O(mn)$. The difference in **task4** and **task5** comes from the memory use. In former task I use a table $H[i,j]$ as above explained, so it needs memory $O(mn)$. In the latter task I only use an array to store $H[j]$ for each row i, and I'll override this array when traverse all row m, so it only needs $O(n)$ storage.

Comparison between task3 -5: According to **Figure2**, the running time of task3 increases much faster than task4 and task5 with the increase of input size ($m \times n$). More specifically task3 shows a perfect cubic polynomial fitting, which is in according with the time complexity. While task4 and task5 show a linear trend, which matches the time complexity $O(mn)$. This comparison clearly shows the power of a good dynamic programming especially when the input size is unlimited.

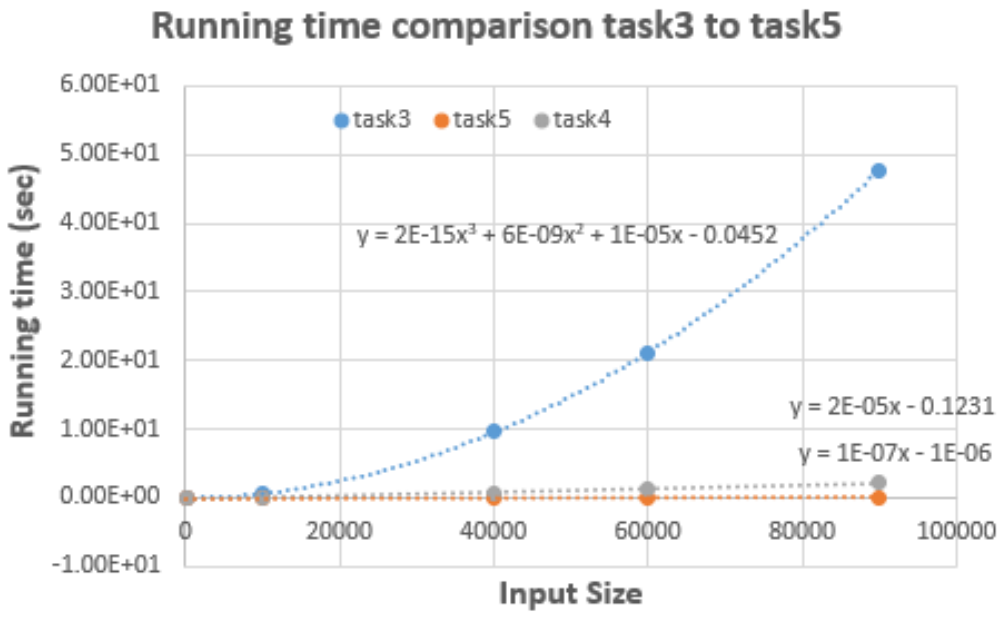

Figure 2. Running time comparison between task3, task4 and task5

Conclusion

In this project, an easy of problem maximum square area is firstly studied with dynamic programming algorithm ALG1, task1 and task2 use a memoization and bottom up approaches separately. Since task2 takes less memory, it would be a better approach for this problem. Then, to gradually study the rectangle's version of this problem, a simply bottom up brute force approach firstly applied in task3 and further dynamic programming approach applied in task4 and task5. Form the comparison, we can see task4 and task5 shows much better performance than task3. From this comparison, dynamic programming shows its power in solving problems.