Problem 6.

(1) Let the board be $W[n][n]$. First, construct two $n \times n$ arrays A, B. $A[i][j] = \sum_{k=0}^{j} W[i][k]$ and $B[i][j] = \sum_{k=0}^{i} W[k][j]$. The time used to construct A, B is $O(n^2)$.

Next, read k rectangles. When we read the upper left point $a = (a_i, a_j)$ and the lower left point $b = (b_i, b_j)$ of a rectangle R. The weight of R is equal to

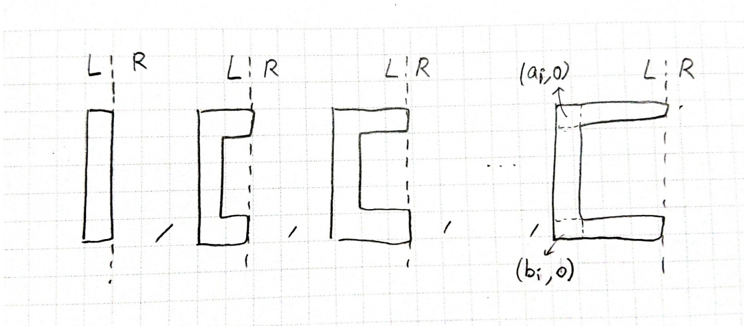
$$\begin{aligned}
&+ A[b_i][b_j - 1] - A[b_i][a_j - 1] \\
&+ A[a_i][b_j] - A[a_i][a_j] \\
&+ B[b_i][b_j] - B[a_i][b_j] \\
&+ B[b_i - 1][a_j] - B[a_i - 1][a_j]
\end{aligned}$$

It takes $O(1)$ time to compute the weight of R. Because there are k rectangles, it take $O(k)$ time to compute all weight. Hence this method takes $O(n^2 + k)$ time.

(2) First, we construct A, B as (1). It takes $O(n^2)$ time. Iterate through all rectangles in W. There are $(C_2^n)^2$ rectangles in total, $(C_2^n)^2 = O(n^4)$. For each rectangle R with upper left point (a_i, a_j) and lower right point (b_i, b_j), we take $O(1)$ to compute its perimeter $2(b_i + b_j - a_i - a_j)$ and $O(1)$ to compute its weight as (1). It takes $O(1)$ to compare perimeter with L and $O(1)$ to update the maximum weight if perimeter is no more than L. The overall time taken is $O(n^2) + O(n^4) * [O(1) + O(1) + O(1) + O(1)] = O(n^4)$.

(3) First, we construct A, B as (1). It takes $O(n^2)$ time. Let $T(n, m)$ be the time to find the rectangle with maximum weight in $n \times m$ board. Use divide and conquer. First we split $n \times m$ board W into two $n \times m/2$ small boards L, R.

First, we find the rectangles R_L, R_R with maximum weight in L, R respectively. This takes $2T(n, m/2)$ time. Next, we find the rectangle R_C with maximum weight that crosses L, R. Suppose the upper left point and lower right point of R_C is (a_i, a_j) and (b_i, b_j) respectively. The left part of R_C is the one that have largest weight in



If we denote the weight of these figures as $w_{\text{mid}-1}, w_{\text{mid}-m/2+1}, \ldots, w_{\text{mid}-m/2}$ respectively, then weight of left part of R_c is $\max\{w_{\text{mid}-m/2}, \ldots, w_{\text{mid}-2}, w_{\text{mid}-1}\}$. Right part of R_C is similar, its weight is $\max\{w_{\text{mid}}, \ldots, w_{\text{mid}+m/2-1}, w_{\text{mid}+m/2}\}$. If we already know a_i, b_i, then we can find the left and right part of R_C in $O(n)$. (with the help of A, B) But we do not know a_i and b_i, so we have to iterate through all possible (a_i, b_i) pairs. There are $O(n^2)$ possibles, each can be done in $O(m)$, so the time used to find R_C is $O(n^2)O(m)$. Then we return the maximum weight of R_L, R_R, R_C, which takes $O(1)$ time.

The recursion relation is $T(n,m) = 2T(n,m/2) + O(n^2)O(m)$ with $T(1) = O(1)$. Because of symmetry (dividing rows or columns are the same), $T(n,m) = 2T(n/2,m) + O(m^2)O(n)$. We have $T(n,n) = 2T(n,n/2)+O(n^2)O(n) = 2(2T(n/2,n/2)+O(n^2)O(n))+O(n^2)O(n) = 4T(n/2,n/2) + 3O(n^3)$. By master theorem, $T(n,n) = O(n^3)$.

(4) We modify the method in (3) to solve (4). First, we construct A, B. It takes $O(n^2)$ time. Let $T(n,m)$ be the time to find the rectangle with maximum weight with perimeter no greater than L in $n \times m$ board. Use divide and conquer. To solve the problem in a $n \times m$ board, we first split it into left and right parts, which are both $n \times m/2$ boards. We find the rectangles R_L, R_R with maximum weight in L, R respectively. This takes $2T(n,m/2)$ time. Next, we find the rectangles R_C with maximum weight that crosses L, R whose perimeter is no greater than L. Suppose the upper left point and lower right point of R_C are (a_i, a_j) and (b_i, b_j) respectively. Construct two arrays W_L, W_R of size $m/2$, for $0 \le i < m/2$,

$$W_L[i] = \max_{0\le j\le i} w_{\text{mid}-j}$$

$$W_R[i] = \max_{0\le j\le i} w_{\text{mid}+j}$$

The construction of W_L, W_R can be done in $O(m)$ time. Suppose for the row a_i and b_i, we need to consider rectangles that have width no greater than C. Then the weight of R_C is equal to $\max\limits_{0\le i\le C}(W_L[i] + W_R[C-i])$, this can be compute in $O(m)$. Since we do not know what $a_i.b_i$ are, we have to iterate through all possibles ($O(n^2)$). So the recursion formula is $T(n,m) = 2T(n,m/2) + O(n^2)O(m)$. This recursion formula is same to (3), so $T(n,n) = O(n^3)$.

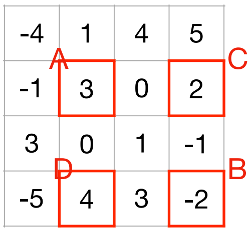
Problem 6

(1) First, examine the n*n cells for $O(n^2)$ time, storing the accumulating weights (including itself) from up, left directions for each cell.

-4	1	4	5
-1	$\boxed{3}$	0	2
3	0	1	-1
-5	4	3	-2

cell[1][1].up = 4
cell[1][1].left = 2

When given a rectangle, we calculate its weight by following steps:
Example: position of upper-left cell A = (1,1)
position of lower-right cell B = (3,3)
step1: find the other two vertices C=(1,3) and D=(3,1) $O(1)$ time



step2: calculate weight and return $O(1)$ time

weight = (C.left - A.left) + (D.up - A.up) + A.weight
+(B.up - C.up) +(B.left - D.left)

The time-complexity of calculating the weight of a rectangle is $O(1)$.
Here we're given k (k ≫ n) rectangles. Since k ≫ n, k can't be ignored.
This algorithm is $O(n^2 + k)$.

(2) In this problem , we test all the possible permutations of the upper-left cell A and lower-right cell B of rectangles, which takes $O(n^2 * n^2) = O(n^4)$ time.
In every loop, we first check the perimeters of the rectangle. If it $> L$, then we break this loop. And every when the weight of one rectangle is obtained, we check whether it's larger than the weight of the rectangle with max weight . If it is, we update the rectangle with max weight to be the current rectangle.

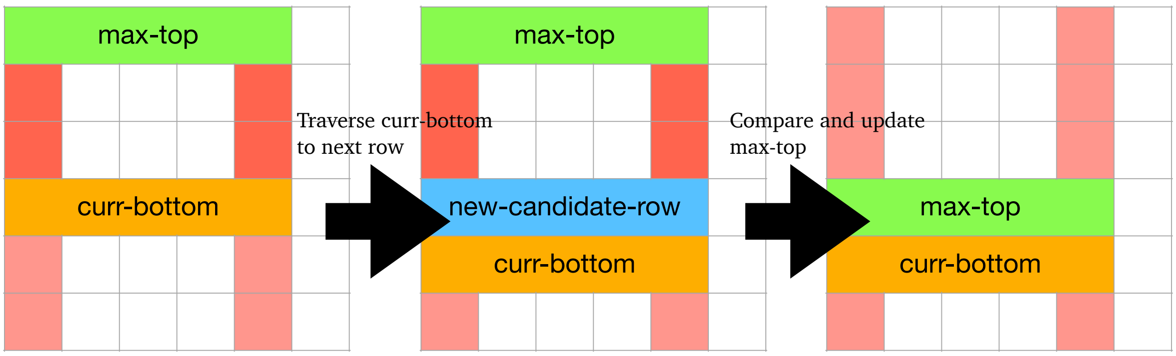
Suppose we totally compute weights of K rectangles, then $K = O(n^4)$.
Substituting $k = K$ into $O(n^2 + k)$, the complexity of computing weight of k rectangles, we know that it takes $O(n^2 + n^4)$ time to compute every possible rectangles.
Hence this algorithm is $O(n^4)$ and at last return the rectangle with max weight whose perimeter $\leq L$.

(3) First, same as (1), we examine the n*n cells for $O(n^2)$ time, storing the accumulating weights (including itself) from up, left directions for each cell, so that we can get the weight sum of any cells that are continuous and in the same row or same column in $O(1)$ time.

Second, we select two columns to be left and right sides of the rectangle with all possible permutations, which take $C_2^n = O(n^2)$ time. Every when we select two columns, there is something we do to find the maximum rectangle(explained in the following), so the overall time complexity will be $O(n^2 *$ the complexity of what we do next).

In the beginning of each loop, we first set row[0] as max-top and row[1] as the current-bottom and check whether it's the rectangle with maximal weight. Then we traverse the current-bottom from row[2] to row[n-1], taking $O(n)$ time. When we change the current-bottom to row[i], row[i-1] becomes a new candidate row for max-top. So we compare the rectangle with row[i-1] as top and the rectangle with original max-top as top, and then update the maximal weight(global variable) and max-top in this loop.

At last, we can find the maximal weight of rectangle.
This algorithm is$O(n^2 + n^2 * n) = O(n^3)$.



reference: b08902028

(4) Similar to (3), but add a ~~queue~~ deque Q to store the rows which are above curr-bottom, arranging from up to down, and from larger weight to smaller weight.

The difference with (3) is described below:
First we push_back row[0] to the Q.
And in the process of traversing the current-bottom from row[2] to row[n-1], we keep compare Q.back() and the new-candidate-row. If the weight of the rectangle with Q.back() as top < the weight of the rectangle with new-candidate-row as top, we pop Q.back(). Keep doing this until Q becomes empty orthe weight of the rectangle with Q.back() as top > the weight of the rectangle with new-candidate-row as top. After that we push_back new-candidate-row to Q. Also, when current-

bottom moves down, we have to recursively pop out Q.front() if the perimeters of the rectangle with Q.front() as top $> L$. So that we can make sure all the rows(elements) in Q are available and arranged decreasingly.(The perimeters of Those rectangle with them as top won't exceed L.)

For every distinct curr-bottom, Q.front() is the max-top. The remaining parts which is not mentioned are still same as (3). Every distinct curr-bottom has its max-weight(the weight of the rectangle with max-top as top). In the loop of $O(n^3)$, we compare them and find the maximal one.

Since every rows are at most pushed and popped respectively once, the operations with Q is $O(1)$ time for each row. The overall time complexity remains $O(n^3)$.