Problem #1

Dynamic programming is similar to divide-and-conquer in that it solves problems by dividing a large problem into subproblems.

Divide-and-conquer might solve subproblems more than once and hence has more time consumption. But, in dynamic programming each subproblem is solved only once, which means no recursions occur. By memorizing subproblems in containers, we don't have to calculate the same subproblem twice but reuse it. So, this significantly reduces an exponential running time of divide-and-conquer.

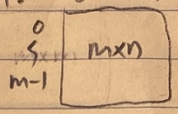
In divide-and-conquer, subproblems are independent of each other. Conversely, in dynamic programming, subproblems are interdependent of each other and a subproblem might be needed to solve other subproblems. But solution of one subproblem may not be part of the solution for other subproblems to the same problem.

Problem #2

Problem#2

Some prepositions are required to clearly approach to this problem.

- Chessboard is a m×n grid, and its columns and rows are indexed from 0 to n-1 and m-1 respectively, starting from top-left of the board. 0~n-1



- The chessboard is empty, which means there are no obstacles in any paths when a rook go through.

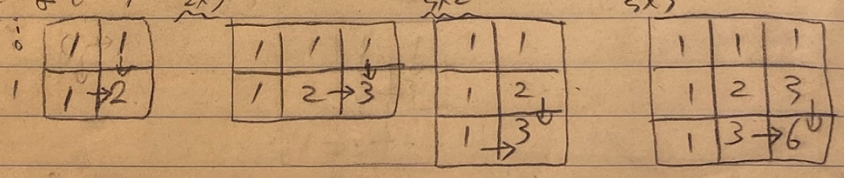
- A rook's movement is limited to right or down because we want to get the shortest paths.

a) To solve this problem, I'm referring to 5 steps to dynamic programming from MIT's lectures.

Let C[i,j] is a grid displaying the number of shortest paths where i = 0, 1, 2 ... , m-1, j = 0, 1, 2 ... , n-1
i is a index of rows and j is a index of columns.

① subproblems: C[i,j] = the number of shortest paths of (i+1)×(j+1) grid.

e.g. 2×2 3×2 3×3



To solve C[2,2], we use C[1,2] and C[2,1].

<8×8 chessboard?>

	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	1	2	3	4	5	6	7	8
2	1	3	6	10	15	21	28	36
3	1	4	10	20	35	56	84	120
4	1	5	15	35	70	126	210	330
5	1	6	21	56	126	252	462	792
6	1	7	28	84	210	462	924	1716
7	1	8	36	120	330	792	1716	3432

$C[7,7] = C[6,7] + C[7,6]$

solution subproblem subproblem

number of subproblems $= \Theta(m \cdot n)$

② choices

- two special cases:
  - $C[i, 0] = 1$, $i = 0, 1, 2 \cdots, 7$
  - $C[0, j] = 1$, $j = 0, 1, 2 \cdots 7$

  This is because we have only one shortest path to get to other square when $i = 0$ or $j = 0$.

- one general case:

  $C[i, j] = c[i-1][j] + C[i][j-1]$

  where $i = 1, 2 \cdots 7$ and $j = 1, 2, \cdots, 7$

③ time of each subproblem is $\Theta(1)$

④ $i = 0 \cdots m$, $j = 0 \cdots n$

total time : $\Theta(mn)$

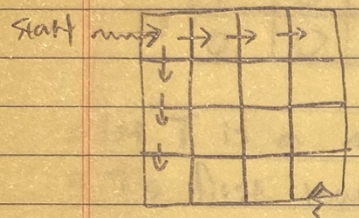
⑤ orignial problem : $C[m, n]$

- Algorithm pseudocode

```
countSortestPath (m, n):
    C= two dimensional array [m, n]
    C[i,0] = 1  for 0≤i<m          // special case
    C[0,j] = 1  for 0≤j<n          // special case

    for i  from 1 to m-1           // general case
      for j from 1 to n-1
        c[i,j] = c[i-1,j] + c[i,j+1]
    return C[m-1, n-1]
```

b) For mxn chessboard, the shortest path should be $(m-1)+(n-1)$ moves where $m-1$ is the number of down moves, and $n-1$ is the number of right moves. E.G. for 4x4 chessboard, the shortest path should be consisted of $3+3=6$ moves.



Let R is a right move.
Let D is a down move.

RRRDDD reaches to the end point.
and RDDRDR reaches to the end point.
RRDRDD also reaches to the endpoint.

So, we can get the number of paths by using combination. We can select a right move from the collection of $(m-1)+(n-1)$ moves.

$$_{(m-1)+(n-1)}C_{(m-1)} = \binom{(m-1)+(n-1)}{m-1}$$

$$= \frac{[(m-1)+(n-1)]!}{(m-1)!(n-1)!}$$

For 8x8 chessboard,

$$_{14}C_7 = \binom{14}{7} = \frac{14!}{7!7!} = 3432.$$

Problem #3

Problem#3

- Let B is a 4 x 5 boolean grid.

B:	0	1	2	3	4
0	1	0	0	1	1
1	1	0	0	0	0
2	1	0	0	0	1
3	0	0	0	0	0

- Let T is a 4x5 boolean grid.
- T's first row and first column copy the B's values reversely. This is an initial setting of T. (0 to 1, 1 to 0)

T:					
	0	1	1	0	0
	0	1	2		
	0				
	1				

- T's cell indicate the length of the maximum square whose values are zero.

- B[1,1] is 0. So, it's a candidate of the largest square matrix. Then we would get the minimum value among T[0,0], T[0,1], and T[1,0]. This will tell us the size of the submatrix of 0 in T[1,1].

- B[1,2] is 0. T[1,2]'s value is a minimum of T[0,1], T[0,2], and T[1,1]. We update 2 for this position, which implies that a square of size 2 is a submatrix up to these indices.

□ When B[m, n]'s value is 1, we update T[m, n] to 0. When we fill up all values in T,

T:

0	1	1	0	0
0	1	2	1	1
0	1	2	2	0
1	1	2	(3)	1

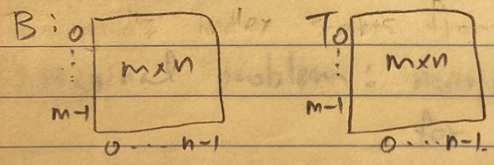
.

↳ This is the largest length of a square submatrix in B. So, we can conclude that the largest square submatrix of zero has a length of 3 for its width and height. The submatrix looks like this

0	0	0
0	0	0
0	0	0

.

○ 5 steps to dynamic programming.
Let B is a m×n grid. B[m, n].
Let T is a m×n grid. T[m, n].



①Subproblems: T[i, j] for $0 \le i < m$, $0 \le j < n$.

number of subproblems : $\Theta(m \cdot n)$

② choices.

- special cases

$T[i, 0]$ = if $M[i, 0]$ is 0 then 1
else then 0. for $0 \le i < m$

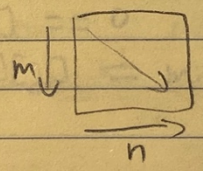
$T[0, j]$ = if $M[0, j]$ is 0 then 1
else then 0. for $0 \le j < n$.

- two choices.

$T[i, j]$ = if $B[i, j]$ is 0, then
$\text{minimum}(T[i-1, j-1], T[i, j-1], T[i-1, j]) + 1$
else, then
0.

③ time of each subproblem is $\Theta(1)$

④ $0 \cdots m$, $0 \cdots n$



Total time = $\Theta(m \cdot n)$

⑤ original problem: maximum $(T[i, j])$
for $0 \le i < m$ and $0 \le j < n$.

This algorithm's time efficiency is $\Theta(m \cdot n)$ where m is a length of rows and n is a length of columns. We need to iterate through all rows and columns of B to check, calculate and fill out lengths of submatrices.

□ Algorithm pseudocode.

```
findMaximumSubmatrix(B):
    m = length of B's row
    n = length of B's column
    T = two dimensional array of m rows and n columns
        and set initial values to 0.

    for i from 0 to m-1        // change 0 to 1
        if B[i][0] == 0 then T[i][0] = 1
    for j from 0 to n-1        // change 0 to 1
        if B[0][j] == 0 then T[0][j] = 1

    for i from 1 to m-1
        for j from 1 to n-1
            if B[i][j] == 0
                then T[i][j] = minimum(T[i-1][j-1],
                                       T[i-1][j],
                                       T[i][j-1]) + 1
            else then T[i][j] = 0

    maxLength = T[0][0].       // finding maximum square
        for i from 0 to m-1
            for j from 0 to n-1
                maxLength = maximum(maxLength, T[i][j])
    // largest square submatrix
    output = two dimensional array of maxLength rows
             and columns filled with 0.
    return output
```

□

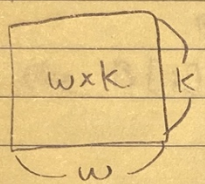
Problem #4

Problem #4

a) Let S is a list of weight-value pairs of items.
k is the number of items.
w = weight , v = value
then $S = [(w_1, v_2), (w_2, v_2), \cdots, (w_k, v_k)]$

Let B is a matrix that states the maximum value for total capacity w and items (1, 2, ... k)

B ⇒



b) $B[k][w] = \begin{cases} B[k-1][w] & \text{if } w_k > w \\ \max\{B[k-1][w], B[k-1][w-w_k] + v_k\} & \text{if } w_k \le w \end{cases}$

where $0 \le k <$ number of items,
$0 \le w <$ weight.

c)
- B[k-1][w] is when the value of item is too big. ($w_k > w$)
- B[k-1][w] is when we don't use item k. ($w_k \le w$)
- B[k-1][w-w_k] + v_k is when we use item k.

a) pseudocode

```
knapsack (B, k, w, S):
    for i from 1 to k
        for j from 1 to w
            if wi ≤ j
                if vi + B[i-1][j-wi] > B[i-1][j]
                    B[i][j] = vi + B[i-1][j-wi]
                else
                    B[i][j] = B[i-1][j]
            else B[i][j] = B[i-1][j]
```

b)

j (weight)

i (item)	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	25	25	25	25
2	0	0	20	25	25	45	45
3	0	15	20	35	40	45	60
4	0	15	20	35	40	55	60
5	0	15	20	35	40	55	65

b) The instance has one optimal subset that is $65.

c) We can trace back the steps, B[i-1][j] or B[i-1][j-wi] all the way up to the start point, and check if there's the same value equivalent to the optimal subset.

e)

We need to fill $n \times W$ matrix and it takes constant time to compute a value for each cell.

- Iterating a matrix and filling cell(i, j) whose $1 \le i \le n$, $1 \le j \le W$ takes $n \cdot W$ times. Deciding if we should put item to knapsack (subproblem) takes $\Theta(1)$. So, $\Theta(\underbrace{1}_{\text{constant}} \cdot n \cdot W) = \Theta(n \cdot W)$ is time efficiency.
- We need a $n \times W$ matrix that stores the maximum values of each cell, and this matrix should remain during the process. So, we have $\Theta(n \cdot W)$ space efficiency.
- To find composition of an optimal subset, we need to compare $B[i][j]$ and $B[i-1][j]$ at most n times because we only have n items (for i from n to 1). So, from filled table, we can find the composition in $O(n)$ time.

- The reason of using big-theta for time and space efficiency is that the bounds depend on the implementation and language you're using. So, $O(n \cdot W)$ and $\Omega(n \cdot W)$ are true.

<Citations>

- https://researchideas.ca/wmt/c6b3.html
- MITOpencourse
- https://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/