Algorithm HW4 산정 김관호

1.

Let dp[n] is minimum # of coins of which sum is equal to the amount n.

- define subproblem

$$dp[n] = \begin{cases} 0 & \text{if } i == 0 \\ \min\{dp[n-coin]\} + 1 & \text{where } coin \in (1,4,6) \text{ and } coin \le n \end{cases}$$

- Recurrence relation

At first, initialize dp array -1 from index 1 to n.
and dp[0] = 0

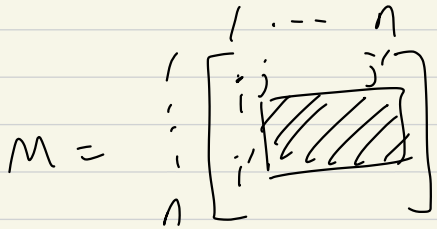
- solve(n)

```
if n == 0 then
    return 0
if dp[n] != -1 then
    return dp[n]

int temp = INF
for coin in (1,4,6) do
    if coin > n then
        continue
    temp = min(temp, solve(n-coin)+1)
dp[n] = temp
return dp[n]
```

The answer is solve(n) when amount of money is n.

2.

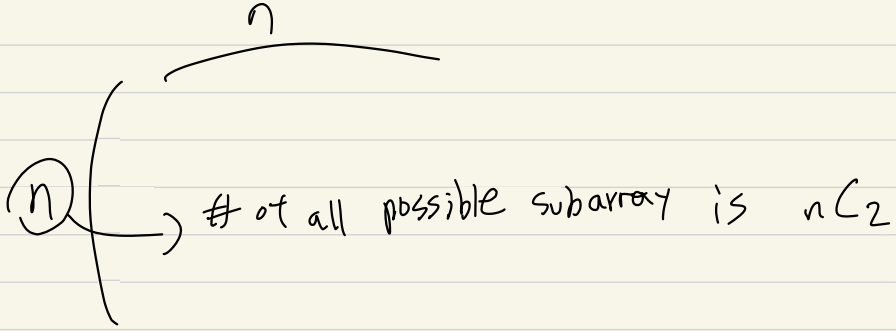


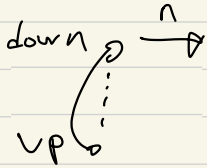
It's possible to find largest sum of subarray in 1D in O(n) time using Kadane's Algorithm.

we'll simplify this problem dealing with 2D array into 1D array while considering all possible 1D array.

That means, we'll consider # of $_nC_2$ 1D arrays. Bounded by row with up and down variables.

So, we can solve this problem in $O(_nC_2 \cdot n) = O(n^3)$





we'll see sub column from row 'down' to 'up' as sole value of array.
Then moving, calculate largest sum of subarray.

Let's see pseudo code.

```
maxSum = -INF
for up from 1 to n do                          -> nC2 subarray
    colSum[n] = {0, }
    for down from up to n do
        for col from 1 to n do
            colSum[col] += M[down][col]

    sum = 0
    for i = 1 ~ n do
        sum += colSum[i]
        maxSum = max(maxSum, sum)
        sum = max(sum, 0)
```

Finally, maxSum is largest sum of rectangular subarray.

First two for loops make $_nC_2$ subarray in aspect of row. And next for loop with 'col' variable fills colSum[] array which helps calculate subarray sum.

In next for loop in same region with previous for loop calculates subarray sum using colSum[] and update max sum if sum is bigger than max sum. And if sum is negative, reset sum zero to guarantee find largest subarray sum.

In this algorithm, time complexity is $O(\frac{n(n-1)}{2} \cdot n \cdot (n+n))$

$= O(n^3)$

3.

Notice the point that robot can't move left or right once it chose direction to move left or right.

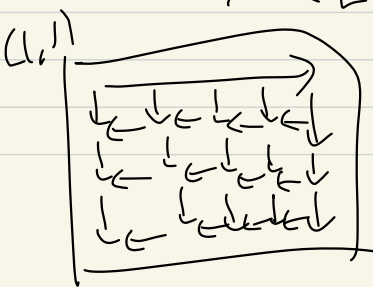
Let define 2D array dp[n][m]
dp[i][j] means maximum sum of values in G when reached entry (i,j). Then, we need to find dp[n][m] which is bottom-right position.

Assume G is one-based array.

And let L[i][j], R[i][j] which contain max sum value while taking only left and right direction starting from 2nd row.

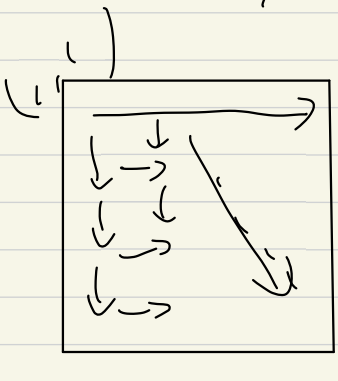
On first row, there is no choice to turn left, only robot can do is go right or down. From 2nd row, robot can choose it's direction.
So L[i][j], R[i][j] define like below

$$L[i][j] = \begin{cases} G[i][j] & \text{where } i=j=1 \\ L[i][j-1] + G[i][j] & \text{where } i=1,\ j>1 \\ L[i-1][j] + G[i][j] & \text{where } i>1,\ j=N \\ \max(L[i-1][j], L[i][j+1]) + G[i][j] & \text{else} \end{cases}$$



How to fill L[i][j]

$$R[i][j] = \begin{cases} G[i][j] & \text{where } i=j=1 \\ R[i][j-1] + G[i][j] & \text{where } i=1,\ j>1 \\ R[i-1][j] + G[i][j] & \text{where } i>1,\ j= \\ \max(R[i-1][j], R[i][j-1]) + G[i][j] & \text{else} \end{cases}$$



The way till R[i>[j]

Finally, combining L[][], R[][] we can fill dp table.

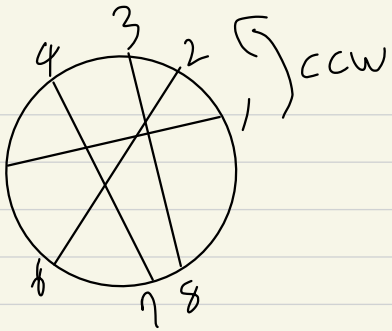
dp[i][j] = max(L[i][j], R[i][j])

Initialize dp, L, R array zero
for i from 1 to n do
  for j from 1 to m do
    fill L, R array, when fill L array from 2nd row, the array should fill last index firstly. that is, L[i][m-j+1] not L[i][j].

then use another 2-for loops and fill dp table.

Time complexity is O(n·m) because of two for loops.

4.



(a) Let's divide n points of set L into $2n$ by choosing any point and labeling $1, \cdots, 2n$ CCW direction. And memorize pair of points that means connection of point $1-5$, $2-6$, $3-8$, $4-7$ above image so that we can know which point is corresponded.

Define 2D dp array size $(2n+1) \cdot (2n+1)$

dp[i][j] means maximum # of line segments in which they aren't intersected. (considering from 'point i' to 'point j')

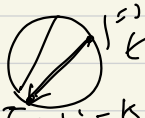
Initialize dp table zero. dp and set of points are one based.
Let k is corresponding point with point j which means $(k, j) \in L$ (k: index)

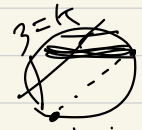
starting from first point to last point with stride 1 to $2n-1$.

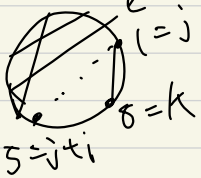
$$dp[i][j] = \begin{cases} dp[i+1][j-1]+1 & \text{if } j = k \\ \max(dp[i+1][j], dp[i][k-1]+dp[k+1][j]+1) & \text{if } i < k < j \\ dp[i+1][j] & \text{else} \end{cases}$$

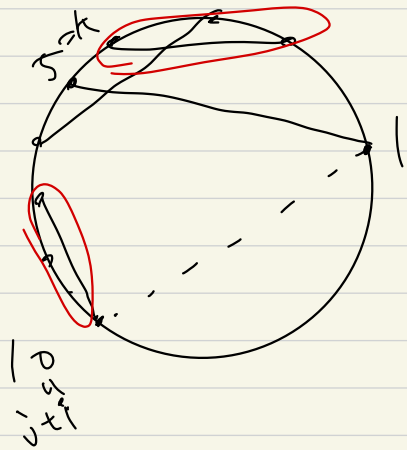
stride ← start point

for i from 1 to 2n−1 do
  for j from 1 to 2n−i do
    Let 'k' is index of corresponding point with j in line.
    if k == j then
      dp[j][j+i] = dp[j+1][j+i−1] + 1
    else if i < k < j then
      dp[j][j+i] = max(dp[j+1][j+i],
                       dp[j][k−1] + dp[k+1][j+i] + 1)
    else dp[j][j+i] = dp[j+1][j+i]







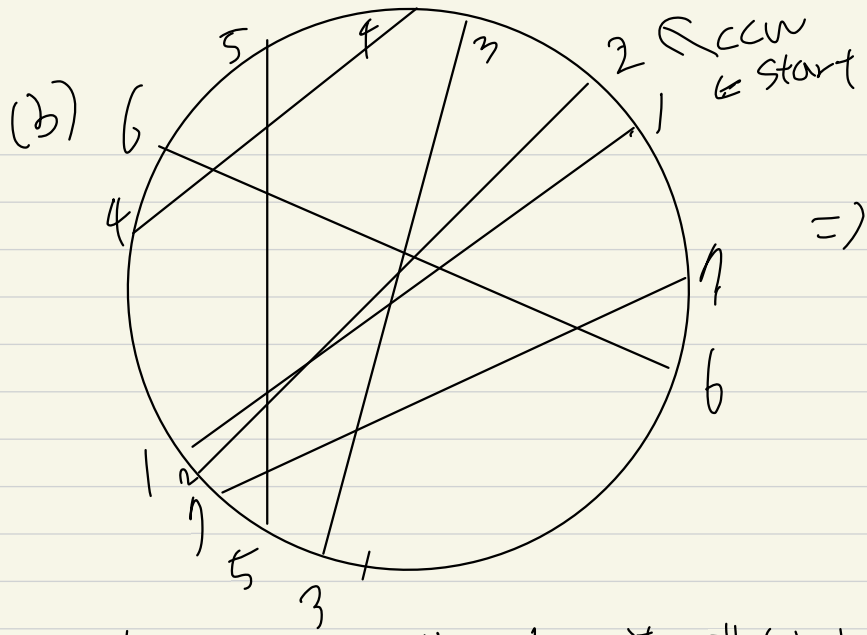


⇒ In this condition, all we have to see is two zone divided by k. And choose max value compared dp[j+1][j+i]
This is when i < k < j

Using above way, we can derive maximum # of line segment in which they aren't intersected.

- Time complexity
  - sorting by angle $(2n)\log(2n)$
  - two for loops (stride, start point) $(2n)^2$

bounded to $O(n^2)$



sorting CCW direction and write all label CCW direction.

* 1 2 3 4 5 6 4 / 1 2 n 5 3 6 n

1 — A — n^{th} — $(n+1)^{th}$ — B — $2n^{th}$

Find LCS A and B => 1236 or 1256

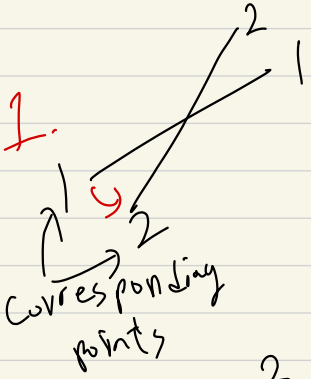
size of LCS is size of largest subset of L in which every pair of segments intersects.

At first, sorting every 2n points CCW direction from any point. And label each point 1 to n. Give same label for two points in line segment.

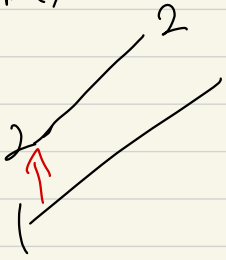
After labeling, start from point 1. while going through whole points append each point's label to string variable.
Then length of that string is 2n.
split string half into A and B.

Since we sort points CCW direction,

condition for intersection every pair of lines is below

1. 

=) when corresponding later point crosses over previous corresponding point, they intersect.

2. 

=) when corresponding later point doesn't cross over previous corresponding point, they don't intersect.

when we see completed string 1212 and 1221 in case 1 and 2, first half of string in 1 is '12' and last half is '12'. Their LCS (Longest common substring) is '12' and it's length is 2 which is the same as # of intersected lines.

In case 2 A = '12' B = '21' and size of LCS is 1. In the case size 1, there is no intersection.

This is because points are sorted by angle and to intersect, later point label must cross over previous point firstly and lastly.

So, all we have to do is derive LCS between first half of written string and other which we have already learned.

— Time complexity.

- sorting CCW => $(2n)\log(2n)$
- write string => $2n$
- Derive LCS of first half and other by Dynamic Programming => n^2
  (first half: size n; other: size n)

So, it is bounded to drawing LCS $O(n^2)$