# CS 170 HW 7

Due **2021-10-18, at 10:00 pm**

## 1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, write "none".

**DP solution writing guidelines:**

Try to follow the following 4-part template when writing your solutions.

- Define a function $f(\cdot)$ in words, including how many parameters are and what they mean, and tell us what inputs you feed into $f$ to get the answer to your problem.
- Write the "base cases" along with a recurrence relation for $f$.
- Prove that the recurrence correctly solves the problem.
- Analyze the runtime and space complexity of your final DP algorithm. Can the bottom-up approach to DP improve the space complexity?

## 2 Saving the Spaceship

The famous spaceship captain Ha Nsolo has successfuly blown up the enemy's space station. Now, only one thing is left – to escape back to his base. He realizes that this will not be so easy because the surrounding space is swarmed with hostile fleet. Whenever Captain Nsolo encounters enemy spaceships on his way home, they will damage the hull of his ship, the Millenium Eagle.

However, the Millenium Eagle is not just an ordinary spaceship. In addition to being able to ordinarily fly through space, it can also perform jumps through the hyperspace. When jumping through the hyperspace, the ship does not encounter any enemies and it can go freely without receiving any damage. Captain Nsolo wants to rely on this mechanism to make sure that he can get home safely. But there is one tiny problem. The distance that the Millenium Eagle can travel through the hyperspace is a dynamic quantity that changes each time it performs a jump through hyperspace. Moreover, there is only a limited number of hyperspace jumps it can perform. So, Ha cannot go through hyperspace the whole way back. However, he has been a captain at the Millenium Eagle for a long time so he knows in advance how far each subsequent jump through the hyperspace will travel. He wants to plan his jumps ahead so as to get as little damage as possible on his way home.

The space is represented as a grid. Initially, Captain Nsolo starts at the initial position $(0, 0)$. We assume that he receives the damage at this square, so $W_{0,0}$ should be included in the damage that the Millenium Eagle receives. He wants to get to the square $(X-1, Y-1)$ for some numbers $X, Y \geq 1$, the width and the height of the grid. Each square $(i, j)$ on the grid $(0 \leq i \leq X-1, 0 \leq j \leq Y-1)$ has a positive number $W_{i,j}$ such that $0 \leq W_{i,j} \leq 9$. $W_{i,j}$ represents the damage dealt to the Millenium Eagle by the enemy spaceship on the square $(i, j)$. If $W_{i,j} = 0$, there is no enemy spaceship on the square $(i, j)$, so Ha can travel freely to that square. Captain Nsolo can only take steps to the right (in the positive X direction) or to the top (in the positive Y direction), one square at a time. However, he also knows a sequence $d_1, ..., d_m$ of distances that the Millenium Eagle can jump through hyperspace. Each $d_i$ is such that $1 \leq d_i \leq 9$. One jump through hyperspace is performed only in one direction (positive X or positive Y), so it cannot go diagonally. The jumps have to be used in the same order as given: the first conducted jump should be of length exactly $d_1$, the second jump should be of length exactly $d_2$ and so on. The Millenium Eagle cannot perform a jump of length $d_i$ before it performed the jumps of lengths $d_1, ..., d_{i-1}$. After it did the jumps of lengths $d_1, ..., d_m$, it cannot make any more jumps.

**Input format**:
The first line contains two integers, $X, Y (2 \leq X, Y \leq 100)$, separated by a space. They give the width and the height of the grid respectively.
The following $Y$ lines contain the description of the grid. Each line $j$ $(0 \leq j \leq Y-1)$ contains $X$ integers separated by a space: $W_{0,j}, ..., W_{X-1,j}$. (So the first line describes the bottom row of the grid, the second line describes the next-to-bottom row of the grid, and so on until the last line that describes the top row of the grid).
The next line contains integers $d_1, ..., d_m (1 \leq m \leq 10^5)$, every two adjacent integers are separated by a single space. $1 \leq d_i \leq 9$, $d_i$ is the length of the $i$'th jump of the Millenium Eagle.

**Output format**:
Output a single integer: the minimal damage that the Millenium Eagle can get as it travels from $(0, 0)$ to $(X-1, Y-1)$

**Note** that $W_{0,0}$ will **always** be included in this sum.

Go to https://hellfire.ocf.berkeley.edu/, access the contest for the current homework and provide a coding solution for the problem.

**Tips**

- When calculating the time your solution will take to run, you can roughly use the following guidelines: $10^8$ basic operations (such as an addition, for example) might run in a second, but might not. $10^7$ operations will most likely run in a second. Anything less than $10^7$, for example, $10^6$ will definitely run in a second.
- To read a line of integers separated by spaces, you can use something like
  d = list(map(int, input().split()))
- You can represent "infinity" in python by float('inf')

**Solution:**

We will use DP.

Let $f(x, y, k)$ = The smallest damage Captain Nsolo can get when he gets to cell $(x, y)$ using exactly $k$ jumps.

Then, the answer to the problem is $\min_k(f(X-1, Y-1, k))$.

The base cases are $f(0,0,0) = W_{0,0}$ since he starts out on the cell $(0,0)$ having made no jumps and $f(0,0,k) = \infty$ for $k > 0$ since he cannot get to the cell $(0,0)$ having made a jump. Also, if $x < 0, y < 0$ or $k < 0$, let $f(x, y, k) = \infty$.

For $k = 0$, the recurrence relation is

$$f(x, y, 0) = \min(f(x-1, y, 0), f(x, y-1, 0)) + W_{x,y}$$

since he can get to cell $(x, y)$ by two ways: from the left adjacent cell and from the bottom adjacent cell.

For $k > 0$, the recurrence relation is

$$f(x, y, k) = \min(f(x-1, y, k), f(x, y-1, k), f(x-d_k, y, k-1), f(x, y-d_k, k-1)) + W_{x,y}$$

since there are four ways to get to the cell $(x, y)$ having made exactly $k$ jumps:

1. Make $k$ jumps before, get to cell $(x, y)$ from the left adjacent cell.
2. Make $k$ jumps before, get to cell $(x, y)$ from the bottom adjacent cell.
3. Make the $k$'th jump from the left onto cell $(x, y)$.
4. Make the $k$'th jump from the bottom onto cell $(x, y)$.

Note that if "tracing back" the jump makes us go out of the board, the corresponding $f$ will be initialized to $\infty$.

We will be evaluating $f$ by first evaluating it for $k = 0$, then for $k = 1$ and so on until we get to the biggest $k$. Note that to calculate $f$ for a fixed $k$, we only need to know $f$ for $k-1$, so at each timestep we only need to store $f$'s calculated for the current $k$ and for $k-1$. Also note that **we don't always have to calculate the answer for all k's**. Once we get to a $k$ such that the jumps $d_1, ..., d_k$ are guaranteed to take Captain Nsolo out of the grid, we can stop since all subsequent jumps are irrelevant.

Now, we present the coding solution to this problem.

```
INFINITY = float('inf') # We will use this in the solution as
                        # the value for infinity.

# First, let us initialize all the inputs.
X, Y = list(map(int, input().split()))
grid = []
for j in range(Y):
    row = list(map(int, input().split()))
    grid.append(row)
jumps = list(map(int, input().split()))
m = len(jumps) # the number of jumps
```

```
# Now, we need to initialize the DP array.
# There will be two DP arrays, each with dimensions X by Y.
dp = []
prev_dp = []
for i in range(X):
    dp.append([INFINITY] * Y)
    prev_dp.append([INFINITY] * Y)

# Initialize the variable that will hold the minimal damage that
# the Millenium Eagle can get when travelling to (X - 1, Y - 1)
minimal_damage = INFINITY

# The function we will be using is f(i, j, k) that equals the minimal
# damage that the Millenium Eagle can get when travelling from (0, 0)
# to (i, j) making exactly k jumps.
# Note that to calculate f(i, j, k) we only need the values of f(i0, j0, k)
# for i0, j0 < i, j and the values of f(i, j, k - 1). So we do not need to
# store all the calculated values, but rather only the values for k jumps
# and for k - 1 jumps.
# The PREV_DP array stores the calculated values for k - 1 jumps and the DP
# array stores the values for k jumps.

for k in range(m + 1):
    # if k = 0, we are considering doing no jumps, so set jump_length to INFINITY
    # (this will result to an index being out of bounds of the array later, so
    # for k = 0, no jump will be made)
    jump_length = INFINITY if k == 0 else jumps[k - 1]

    # this variable will keep track whether at least one cell is reachable by jumps 1...k
    # if after we go through the whole array we have that for all i, j dp[i][j] = INFINITY,
    # then there is no point to iterate through k further
    saw_not_infinity = False

    for x in range(X):
        for y in range(Y):

            if x == 0 and y == 0:
                if k == 0:
                    dp[0][0] = grid[0][0] # base case
                    saw_not_infinity = True
                else:
                    dp[0][0] = INFINITY # cannot get to (0, 0) with a jump
                continue
            # Now, we will look at 4 cases:
            # Case 1: We came to this cell from the adjacent left cell
            # Case 2: We came to this cell from the adjacent bottom cell
            # Case 3: We came to this cell using a jump from a cell to the left
            # Case 4: We came to this cell using a jump from a cell to the bottom
            # The minimum of those will be the best

            # if we come to this cell from the adjacent left cell
            x_left, y_left = x - 1, y
            if x_left >= 0 and y_left >= 0:
                val_left = dp[x_left][y_left]
            else:
                val_left = INFINITY

            # if we come to this cell from the adjacent bottom cell
            x_bottom, y_bottom = x, y - 1
            if x_bottom >= 0 and y_bottom >= 0:
                val_bottom = dp[x_bottom][y_bottom]
            else:
                val_bottom = INFINITY

            # if we come to this cell using the jump from some left cell
            x_l_jump, y_l_jump = x - jump_length - 1, y
            if x_l_jump >= 0 and y_l_jump >= 0:
                val_l_jump = prev_dp[x_l_jump][y_l_jump]
            else:
                val_l_jump = INFINITY

            # if we come to this cell using the jump from some bottom cell
            x_b_jump, y_b_jump = x, y - jump_length - 1
            if x_b_jump >= 0 and y_b_jump >= 0:
                val_b_jump = prev_dp[x_b_jump][y_b_jump]
            else:
                val_b_jump = INFINITY

            # Note that if we are coming with a jump, we are looking at PREV_DP,
            # whereas if we are coming by an ordinary step, we are looking at DP.

            min_val = min(val_left, val_bottom, val_l_jump, val_b_jump)
            if min_val != INFINITY:
                saw_not_infinity = True
            dp[x][y] = min_val + grid[y][x]

    minimal_damage = min(minimal_damage, dp[X - 1][Y - 1])
    if not saw_not_infinity:
        break
    prev_dp = dp
    dp = []
    for i in range(X):
        dp.append([INFINITY] * Y)


print(minimal_damage)
```

## 3 Knightmare

Give a dynamic programming algorithm to find the number of ways you can place knights on an $N$ by $M$ ($M < N$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Clearly describe your algorithm and prove its correctness. Your algorithm's runtime can be exponential in $M$ but should be polynomial in $N$.

**(Please provide a 4-part DP solution as defined at the top of this homework)**

**Solution:** The first part of this solution is the old 3-part format. The 4-part solution is below it.

We use length $M$ bit strings to represent the configuration of rows of the chessboard (1 means there is a knight in the corresponding square and otherwise 0).

The main idea of the algorithm is as follows: we solve the subproblem of the number of valid configurations of $(n-1) \times M$ chessboard and use it to solve the $n \times M$ case. Note that as we iteratively incrementing $n$, a knight in the $n$-th rows can only affect configurations of rows $n+1$ and $n+2$. So we can denote $K(n, u, v)$ as the number of possible configurations of the first $n$ rows with $u$ being the $(n-1)$-th row and $v$ being the $n$-th row, and then use dynamic programming to solve this problem.

Let a list of bitstrings be valid if placing the knights in the first row according to the first bitstring, in the second row according to the second bitstring, etc. doesn't cause two knights to attack each other. Then we have $K(2, u, v) = 1$ if $u, v$ are valid and 0 otherwise for all $u, v$ pairs.

For $K(n, v, w)$ we have:

$$K(n, v, w) = \sum_{u: u,v,w \text{ are valid}} K(n-1, u, v)$$

**Proof of Correctness:** The only 2-row configuration of knights ending in row configurations $u, v$ is the configuration $u, v$ itself. So $K(2, v, w) = 1$ if $u, v$ are valid. Otherwise, $K(2, v, w) = 0$.

For $n > 2$, the above recurrence is correct because for any valid $n$-row configuration ending in $v, w$, the first $n-1$ rows must be a valid configuration ending in $u, v$ for some $u$, and for this same $u$, the last three rows $u, v, w$ must be also valid configuration. Moreover, this correspondence between $n$-row and $(n-1)$-row configurations is bijective.

**Runtime Analysis:** To bound the total runtime, first note that for each row, we iterate over all possible configurations of knights in the row and for each such configuration, we perform a sum over all possible valid configurations of the previous two rows. The time to check if a configuration is valid is $O(M)$. Therefore, the time taken to compute the sub-problems for a single row is $O(2^{3M}M)$ which gives us an overall runtime of $O(2^{3M}MN)$ operations. However, since there are a potentially exponential number of possible configurations (at most $2^{NM}$), we cannot assume that arithmetic on these large numbers is constant time. Summing together numbers up to $L$ takes $O(\log L)$ bit operations, so each of our operations takes at most $O(NM)$ time; this gives us a final runtime of $O(2^{3M}M^2N^2)$

**4-part solution:**

**Subproblems:** $f(n, u, v)$ is the number of possible valid configurations of the first $n$ rows with $u$ being the configuration of the $(n-1)$-th row and $v$ being the configuration of the $n$-th row.

**Recurrence and Base Cases:** For $n > 2$, $f(n, v, w) = \sum_{u:u,v,w \text{ are valid}} f(n-1, u, v)$. For the base case, $f(2, u, v) = 1$ if $u, v$ are valid and 0 otherwise for all $u, v$ pairs. No other base cases are needed.

**Proof of Correctness:** See proof of correctness in 3-part solution above.

**Runtime and Space Complexity:** See runtime above. For space complexity, note that there are only $O(N2^{2M})$ subproblems ($N$ rows, $2^M$ settings to the $N-1$-th row, and $2^M$ settings to the $N$th row). Each subproblem is a number of possible configurations, bounded above by $2^{MN}$, so our total space is $O(2^{2M}N^2M)$. Note that since each subproblem only depends on subproblems of the previous row, we could reduce this by a factor of $N$ to $O(2^{2M}NM)$ by recycling space from earlier rows.

## 4 Modeling: Tricks of the Trade

One of the most important problems in the field of *statistics* is the *linear regression problem.* Roughly speaking, this problem involves fitting a straight line to statistical data represented by points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ on a graph. Denoting the line by $y = a + bx$, the objective is to choose the constants $a$ and $b$ to provide the "best" fit according to some criterion. The criterion usually used is the *method of least squares*, but there are other interesting criteria where linear programming can be used to solve for the optimal values of $a$ and $b$.

Suppose instead we wish to minimize the sum of the absolute deviations of the data from the line, that is,

$$\min \sum_{i=1}^{n} |y_i - (a + bx_i)|$$

Write a linear program with variables $a, b$ to solve this problem.

*Hint: Create a new variable $z_i$ that will equal $|y_i - (a + bx_i)|$ in the optimal solution.*

**Solution:**

Note that the smallest value of $z$ that satisfies $z \geq x, z \geq -x$ is $z = |x|$.

Now, consider the following linear programming problem:

$$\min \sum_{i=1}^{n} z_i$$
$$\text{subject to} \begin{cases} y_i - (a + bx_i) \leq z_i & \text{for } 1 \leq i \leq n \\ (a + bx_i) - y_i \leq z_i & \text{for } 1 \leq i \leq n \end{cases}.$$

If for some solution we have that $z_i > |y_i - (a + bx_i)|$, then by setting $z_i = |y_i - (a + bx_i)|$ we will get a solution with a smaller value of the objective function, therefore the initial solution was not optimal.

This means that the optimal solution will set $z_i = |y_i - (a + bx_i)|$, so the new problem is indeed equivalent to the original problem. However, now it is a linear programming problem.