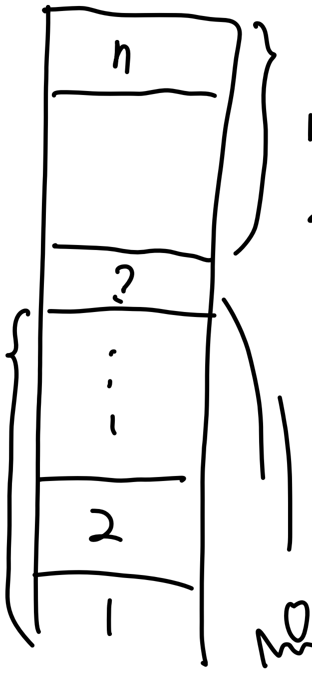
1. Study Group

Jialiang Tang (myself), SID: 3039758308

Yuanzhe Wang, SID: 3039749520

2. (a) $M(x, m) = M(x-1, m-1) + M(x-1, m) + 1$



After we drop an egg on floor f, it will break or not. If it breaks, then we should use $x-1$ drops and $m-1$ eggs to decide l below floor f, else we should use $x-1$ drops and m eggs to decide l above floor f

To ensure the two can both be successful, $f-1$ should be at most $M(x-1, m-1)$ while $n-f$ should be at most $M(x-1, m)$.

So $n = (n-f) + (f-1) + 1 \leq M(x-1, m-1) + M(x-1, m) + 1$

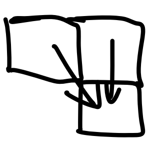
to guarantee we can solve the problem.

So $M(x, m) = M(x-1, m-1) + M(x-1, m) + 1$

(b) Our algorithm is also based on dynamic programming. Using the subproblem just described, base cases are:

$M(x, 0) = 0$ for all x and $M(0, m) = 0$ for all m.

We compute $M(1, 1), M(1, 2), \cdots, M(1, m)$



then $M(2, 1), M(2, 2), \cdots, M(2, m)$

$\cdots$

until $M(x, 1), \cdots, M(x, m)$. (Given $1 \le x \le m$)

So finally we can get $M(x, m)$.

Runtime: There are $O(xm)$ subproblems

Each subproblem takes $O(1)$ to solve.

So ~~total~~ time complexity is $O(xm)$

(c) We only need to find the first $M(x, m) \ge n$ for $1 \le x \le m$. So we should do as (b) does to compute all $M(x, m)$ from $x=1$ until we get some $M(x', m) \ge n$ and we just output x'.

(d) The runtime of Alg in (c) is obviously $\Theta(x'm)$

On the other hand we know $x' \le n$.

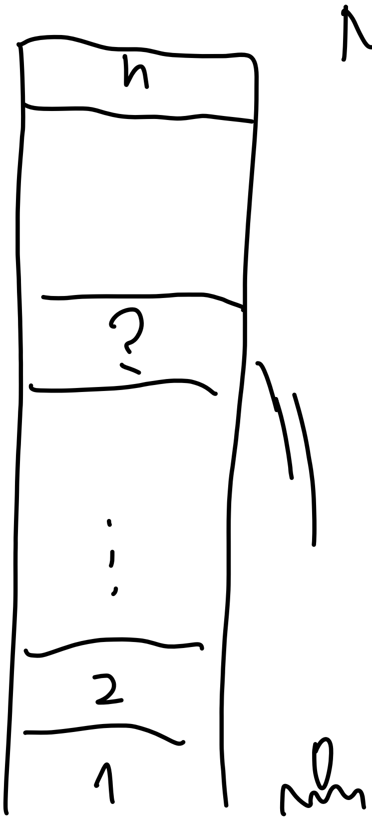
So runtime of Alg in (c) is $O(nm)$.

The runtime of alg last week is $O(n^2 m)$.

So the alg (c) gives is faster than that of last week.

(e) Since we only need $M(x-1, m-1)$ and $M(x-1, m)$ to compute $M(x, m)$, we only need to store 2 lines of matrix M at once. The space complexity is $O(m)$.

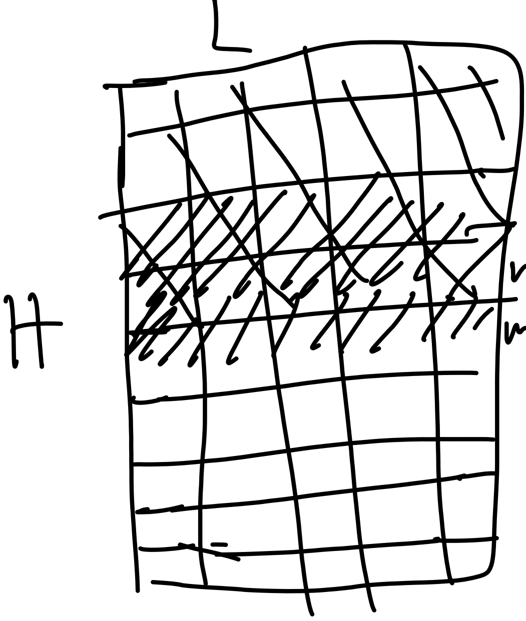
(f) No need to redefine M, just compute $M(x, 2m)$ in (c) alg to find the first x' s.t. $M(x, 2m) > n$. Output x'.



$$M(x, i) = \underbrace{M(x-1, i)}_{\text{No change!}} + M(x-1, i-1) + 1.$$

$(0 \le i \le 2m)$

3. Algorithm Description: (State Compression)



① Subproblem: L-bit number to represent which lines have m knights. We can solve the problem about $x \times L$ chessboard and then solve $(x+1) \times L$ chessboard problem. $dp[x][l_1][l_2]$ is the answer of problem about first x rows with the $(m-1)$-th row is l_1 and m-th row is l_2.

② Recurrence:

$$dp[x][l_1][l_2] = \sum_{\substack{l' \text{ with} \\ \text{no confliction} \\ \text{with } l_1 \text{ and } l_2}} dp[x-1][l'][l_1] \bmod 1337 \quad (x \geq 3)$$

Base Cases:

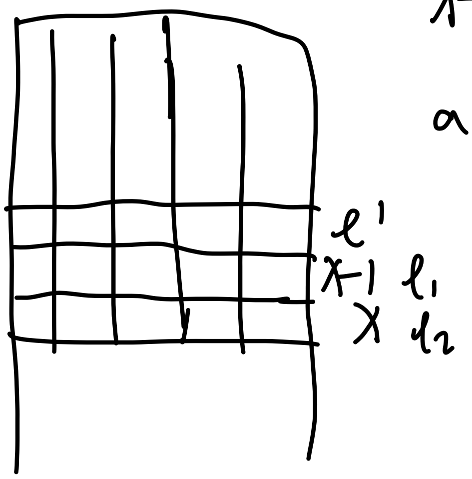
$$dp[2][l_1][l_2] = \begin{cases} 1, & \text{for no-conflicted } (l_1, l_2) \\ 0, & \text{otherwise} \end{cases}$$

③ Ordering: Just compute dp in increasing order of x. For a given x, compute all $dp[x][l_1][l_2]$ for all no-conflicted (l_1, l_2).

Proof of Correctness: Situation of 2-row: the solution is just l_1 and l_2, so $dp[2][l_1][l_2]$

$$= \begin{cases} 1, & l_1 \text{ has no confliction with } l_2 \\ 0, & \text{otherwise} \end{cases}$$

Situation of X-row ($X > 2$): A solution for



X-row with l_1 and l_2 can be got by a solution for (X-1)-row with l' and l_1 which have no confliction with l_2.

So $dp[X][l_1][l_2] = \sum_{\substack{l',\ (l', l_1, l_2) \\ \text{has no confliction}}} dp[X-1][l'][l_1] \mod 1337$

(It's a bijective map between 2 solution)

Runtime Analysis: H rows

Each subproblem we need to compute all (l_1, l_2) pairs, which is $O(2^{2L})$. Computing all (X-1)-row takes $O(2^L)$. Checking confliction takes $O(L)$. So each row we need to take $O(2^{3L} \cdot L)$ to compute.

Total Runtime: $O(2^{3L} \cdot L \cdot H)$.

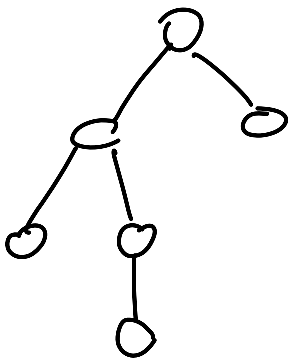
Space Complexity: Only $H \cdot 2^{2L}$ subproblems.

So space complexity is $O(H \cdot 2^{2L})$.

By rolling array optimization, it can be reduced to $O(2^{2L})$. (Same technique as T2 (e))

4. (a) <u>Algorithm Description</u>: ① Subproblem: We can firstly solve the problem with and without the root node of current subtree. Then use it to solve a bigger subtree problem. $dp[v][k][0/1]$: The max weight of ~~sb~~ subtree with root v and at most k nodes in independent set, and 0: root isn't in set, 1: root is in set.

② Recurrence: $dp[v][k][1] = A[v] + \max\left\{ \sum_{\substack{0 \le i \le \#\text{ of child of } v \\ \sum k_i = k-1}} dp[\text{child}_i \text{ of } v][k_i][0] \right\}$



$$dp[v][k][0] = \max_{\sum k_i = k} \left\{ \sum_{0 \le i \le \#\text{ of child of } v} dp[\text{child}_i \text{ of } v][k_i][0/1] \right\}$$

Base Cases: $dp[\text{leaf}][0][0/1] = 0$

$dp[\text{leaf}][k][0] = 0 \quad (k \ge 1)$

$dp[\text{leaf}][k][1] = \max\{A[\text{leaf}], 0\} \quad (k \ge 1)$

③ Ordering: Compute according to the post value in increasing order. For given v, compute all dp values of k.

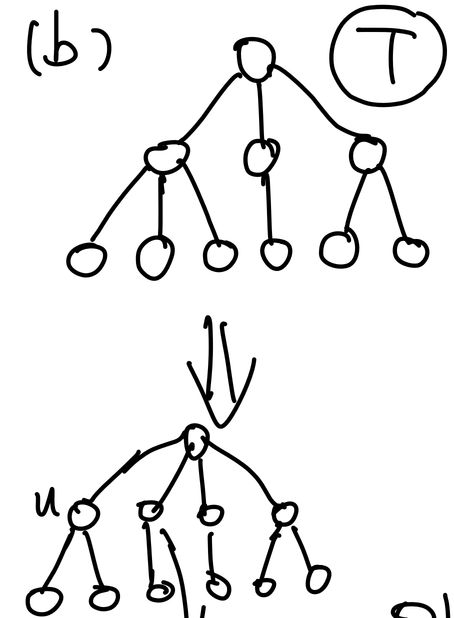
<u>Runtime Analysis</u>: There are $2nk$ subproblems.

Each subproblem we need $O(k)$ to compute.

($O(k)$ ways to assign k_i)

Total time is $O(nk^2)$.

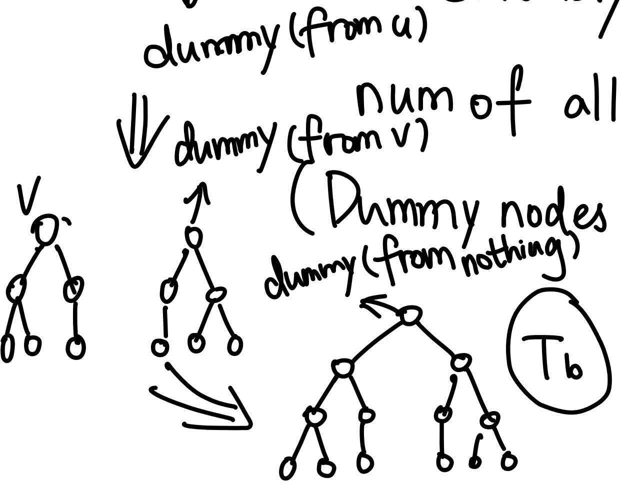
(b)



We do such modification on T :

While there's node N having more than 2 children:

add dummy node as a sibling of N to 'rob' at most 2 children of N. then make them same parent.



Obviously num of dummy nodes won't exceed num of all origin nodes, which is $O(n)$.

(Dummy nodes added at depth $i \leq$ # of nodes at depth $i+1$)

↓

a loose bound!

(c) <u>Algorithm Description</u>: Record which dummy node each origin node makes. Whenever we choose the origin node, we choose its dummy node too. Following this rule we just run alg in (a) on Tb. But dummy nodes don't consume # of nodes we can choose. Also, we never choose root of Tb.

<u>Runtime Analysis</u>: $O(2n)$ nodes so ~~total~~ runtime is $O(2nk^2) = O(nk^2)$