# CS 170 Homework 6

Due **3/07/2022, at 10:00 pm (grace period until 11:59pm)**

## 1 Study Group

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

## 2 Egg Drop Revisited

Recall the Egg Drop problem from Homework 5:

*You are given $k$ identical eggs and an $n$ story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor $\ell$ or lower, and always breaks if dropped from floor $\ell+1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more.*

*Let $f(n, k)$ be the minimum number of egg drops that are needed to find $\ell$ (regardless of the value of $\ell$).*

Instead of solving for $f(n, k)$ directly, we define a new subproblem $M(d, k)$ to be the maximum number of floors for which we can always find $\ell$ in at most $d$ drops using $k$ eggs.

(a) Find a recurrence relation for $M(d, k)$ that can be computed in constant time given the previous subproblems. Briefly justify your recurrence.

(Hint: *As a starting point, what is the highest floor that we can drop the first egg from and still be guaranteed to solve the problem with the remaining $d-1$ drops and $k-1$ eggs if the egg breaks?*)

(b) Give an algorithm to compute $M(d, k)$ given $d$ and $k$ and analyze its runtime.

(c) Modify your algorithm from (b) to compute $f(n, k)$ given $n$ and $k$. (Hint: *If we can find $\ell$ when there are more than $n$ floors, we can also find $\ell$ when there are $n$ floors.*)

(d) Show that the runtime of the algorithm of part (c) is $O(nk)$.

(e) How can we implement the algorithm using $O(k)$ space?

**Solution:**

(a)

$$M(d, k) = M(d-1, k-1) + M(d-1, k) + 1$$

We will first deduce $i$, the optimal floor from which we will drop the first egg given we have $d$ drops and $k$ eggs. If the egg breaks after being dropped from floor $i$, we have reduced the problem to floors 0 through $i-1$, and the strategy can solve this subproblem

using $d-1$ drops and $k-1$ eggs. The optimal strategy for $d-1$ drops and $k-1$ eggs can distinguish between $M(d-1,k-1)+1$ floors, so we should choose $i = M(d-1,k-1)+1$.

On the other hand, if the egg doesn't break, we reduce the problem to floors $i$ to $n$ with $d-1$ drops and $k$ eggs. The maximum number of floors we can distinguish using this many drops and eggs is $M(d-1,k)+1$, so we can solve this subproblem for $n$ as large as $i + M(d-1,k) = M(d-1,k-1) + M(d-1,k) + 1$.

(b) For base cases, we'll take $M(0,k) = 0$ for any $k$ and $M(d,0) = 0$ for any $d$. Starting with $d = 1$, we compute $M(d,x)$ for all, $1 \le x \le k$, and do so again for increasing values of $d$, up until we compute $M(d,x)$ for all $1 \le x \le k$. We return $M(d,k)$.

We compute $dk$ subproblems, each of which takes constant time, so the overall runtime if $\Theta(dk)$.

(c) Again, starting with $d = 1$, we compute $M(d,x)$ for all, $1 \le x \le k$, and do so for increasing values of $d$. This time, we stop the first time we find that $M(d,k) \ge n$, and return this value of $d$.

(d) Because there are only $n$ floors, the optimal number of drops, $d$ will always be at most $n$. From part (b), we know the runtime is $\Theta(dk)$, so if $d \le n$, we know the runtime must be $O(nk)$.

(This is a very loose bound, but it is still much better than the naive $O(n^2k)$-time algorithm we'd get from using the recurrence on $f(n,k)$ directly.)

(e) While we're computing $M(d,x)$ for all $1 \le x \le k$, we only need to store $M(d-1,x)$ and $M(d,x)$ for all $x$, i.e. we only ever need to store $O(k)$ values. In particular, after computing $M(d,x)$ for all $x$, we can delete our stored values of $M(d-1,x)$.

## 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 $N$. Return your answer mod 1337.

For this problem, write your answer in the following 4-part format:

(a) 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.

(b) Write the "base cases" along with a recurrence relation for $f$.

(c) Prove that the recurrence correctly solves the problem.

(d) Analyze the runtime and space complexity of your final DP algorithm. Can the bottom-up approach to DP improve the space complexity? (For example, we saw in class that Knapsack can be solved in O(nW) space, but one can reduce that to O(W) space via the optimization of only including the last two rows.)

**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:

$$K(n, v, w) = \sum_{u : u,v,w \text{ are valid}} K(n-1, u, v) \bmod 1337$$

**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 subproblems for a single row is $O(2^{3M}M)$ which gives us an overall runtime of $O(2^{3M}MN)$ operations. Although there are a potentially exponential number of possible configurations (at most $2^{NM}$), we are only interested in the answer mod 1337, which means that all of our numbers are constant size, and so arithmetic on these numbers is constant time. This gives us a final runtime of $O(2^{3M}MN)$.

**4-part solution:**

**Subproblems:** $f(n, u, v)$ is the number of possible valid configurations mod 1337 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 $n > 2$, $f(n, v, w) = \sum_{u:u,v,w \text{ are valid}} f(n-1, u, v) \bmod 1337$. 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 mod 1337,

bounded above by 1337, so our total space is $O(2^{2M}N)$. Note that since each subproblem only depends on subproblems of the previous row, we could reduce this by a factor of $N$ to $O(2^{2M})$ by recycling space from earlier rows.

## 4 Balloon Popping Problem.

You are given a sequence of $n$-balloons with each one of a different size. If a balloon is popped, then it produces noise equal to $s_{left} \cdot s_{popped} \cdot s_{right}$, where $s_{popped}$ is the size of the popped balloon and $s_{left}$ and $s_{right}$ are the sizes of the balloons to its left and to its right. If there are no balloons to the left, then we set $s_{left} = 1$. Similarly, if there are no balloons to the right then we set $s_{right} = 1$, while calculating the noise produced.

After popping a balloon, the balloons to its left and right become neighbors. (Note that the total noise produced depends on the order in which the balloons are popped.)

Design a polynomial-time dynamic programming algorithm to compute the the maximum noise that can be generated by popping the balloons. *Hint: Read the section of the textbook on Matrix Chain Multiplication.*

Example:
Input (Sizes of the balloons in a sequence): ④ ⑤ ⑦
Output (Total noise produced by the optimal order of popping): 175

Walkthrough of the example:

- **Current State** ④ ⑤ ⑦
  *Pop Balloon* ⑤
  **Noise Produced** $= 4 \cdot 5 \cdot 7$

- **Current State** ④ ⑦
  *Pop Balloon* ④
  **Noise Produced** $= 1 \cdot 4 \cdot 7$

- **Current State** ⑦
  *Pop Balloon* ⑦
  **Noise Produced** $= 1 \cdot 7 \cdot 1$

- **Total Noise Produced** $= 4 \cdot 5 \cdot 7 + 1 \cdot 4 \cdot 7 + 1 \cdot 7 \cdot 1$.

(a) Define your subproblem.

(b) What are the base cases?

(c) Write down the recurrence relation for your subproblems. What is the runtime of a dynamic programming algorithm using this subproblem?

**Solution:**

(a) We want to form subtrees $C(i, j)$ representing the *maximum* amount of noise produced by popping balloons in the sublist $i, i+1, \ldots j$ first before the other balloons. The smallest subproblem is when there is only 1 balloon to pop. The size of the subproblem is the number of multiplications.

(b) Base case: When the size of sublist to pop out is only one balloon $s_{popped}$, return the noise $s_{left} \cdot s_{popped} \cdot s_{right}$. In addition, we need to initialize $s_0 = 1$ and $s_{n+1} = 1$ assuming the input is from 1 to $n$. In our algorithm we'll never actually pop these two balloons as they are just dummy balloons on the left and right.

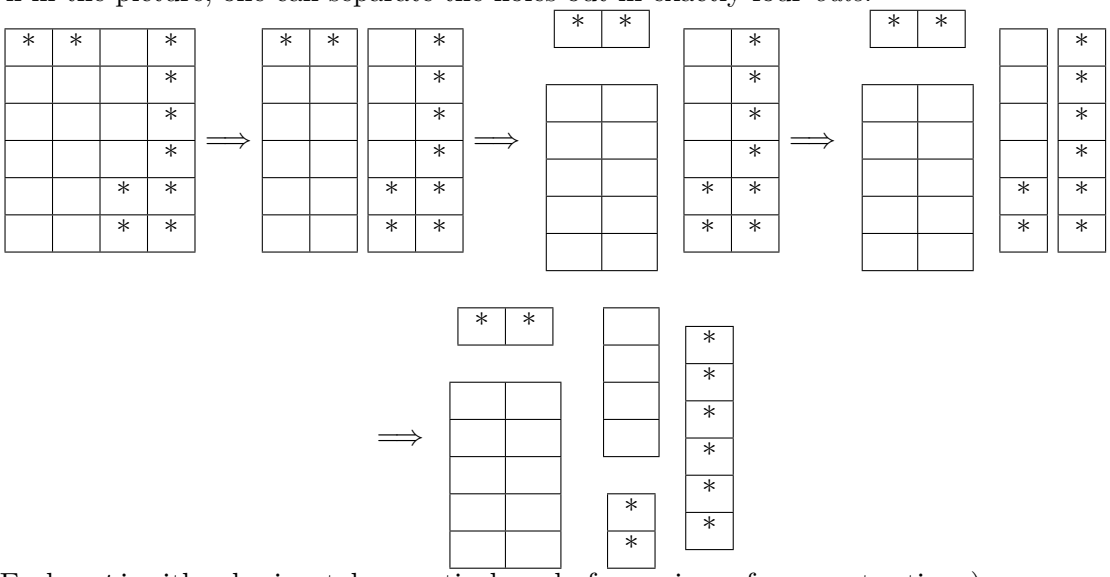
(c)
$$C(i, j) = \max_{i \le k \le j} \{C(i, k-1) + C(k+1, j) + s_{i-1} \cdot s_k \cdot s_{j+1}\}$$

**Justification:** The leaves of the tree represent the last balloon being popped, so here, $k$ represents the index of the last balloon being popped. Then we can recurse up, and at each level find the splitting point $k$ that maximizes the value of the subtree (noise produced for that sequence). Runtime of this algorithm is $O(n^3)$ because there are $O(n^2)$ subproblems, each of which requires $O(n)$ time to compute given the answers to smaller subproblems.

## 5 Paper Cutting

There is a piece of paper consisting of an $m \times n$ rectangular grid of squares. Some of the squares have holes in them, and you cannot use paper with a hole in it. You would like to *cut* the paper into pieces so as to separate all the squares with holes from all the intact squares.

For example, shown below is a $6 \times 4$ piece of paper with holes in squares marked *. As shown in the picture, one can separate the holes out in exactly four *cuts*.



(Each *cut* is either horizontal or vertical, and of one piece of paper at a time.)

Design a DP based algorithm to find the smallest number of cuts needed to separate all the holes out. Formally, the problem is as follows:

**Input:** Dimensions of the paper $m \times n$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the $ij^{th}$ square has a hole.

**Goal:** Find the minimum number of cuts needed to separate the holes out.

(a) Define your subproblem.

(b) Write down the recurrence relation for your subproblems.

(c) What is the time complexity of solving the above mentioned recurrence? Provide a justification.

**Solution:**

(a) We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of cuts needed to separate the submatrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of pieces with holes in them or intact pieces.

(b)

$$B[i_1, j_1, i_2, j_2] = \min \begin{cases} 0, & \text{if all entries of } A[i_1 \dots i_2, j_1 \dots j_2] \text{ are equal} \\ 1 + B[i_1, j_1, i_1 + k, j_2] + B[i_1 + k + 1, j_1, i_2, j_2] & \text{for any } k \in \{1, \dots, i_2 - i_1\} \\ 1 + B[i_1, j_1, i_2, j_1 + k] + B[i_1, j_1 + k + 1, i_2, j_2] & \text{for any } k \in \{1, \dots, j_2 - j_1\} \end{cases} \tag{1}$$

Alternatively, you could have also encapsulated the 0 base case in all single-square pieces, and determined if a piece was pure via the merging, see below.

(c) Two answers are acceptable: $O((m + n)m^2n^2)$ and $O(m^3n^3)$

We have $O(m^2n^2)$ total subproblems: $O(mn)$ possibilities for $(i_1, j_1)$, and $O(mn)$ possibilities for $(i_2, j_2)$. For each subproblem, we examine up to $m$ possible choices for horizontal splits, and $n$ possible choices for vertical splits. A single split consideration will result in two smaller subproblems, which we can assume have already been solved, so we just need to find the best split, which takes $O(n + m)$ time.

In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only intact paper, or contains only paper with holes). Brute force checking this takes $O(mn)$ time, for a total subproblem time of $O(mn + (m + n)) \to O(mn)$.

However, this $O(mn)$ factor can be reduced to $O(m + n)$ (this is not required to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Technically, for us to pre-compute faster than $O(m^3n^3)$, we'll need to compute the purities more intelligently than by brute-force. It is possible to do this in as fast as $O(\max\{m, n\}^2)$ time, although the details of this are complicated and won't be explained here. So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time, then we can reach an overall time of $O((m + n)m^2n^2)$.

Alternatively, we can initialize all min-cut values of single square pieces to be 0. Then, if it is possible to have some cut such that both resulting pieces have min-cut values of 0, and both resulting pieces are of the same type (intact-only or hole-only, and we can take any sample of either and compare them), then we ourself are a pure piece. This

would allow you to avoid the entire pre-computation business as mentioned before, and still achieve a runtime of $O((m+n)m^2n^2)$.