CS 170 Homework 7

Due **Saturday 3/15/2025, 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, explicitly write "none".

2 Egg Drop

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

Let $f(n, m)$ be the minimum number of egg drops that are needed to find b (regardless of the value of b).

(a) Find $f(1, m)$, $f(0, m)$, $f(n, 1)$, and $f(n, 0)$. Briefly explain your answers.

Hint: use ∞ to denote that it is impossible to find b.

(b) Consider dropping an egg at floor h when there are n floors and m eggs left. Then, it either breaks, or doesn't break. In either scenario, determine the minimum remaining number of egg drops that are needed to find b in terms of $f(\cdot, \cdot)$, n, m, and/or h.

(c) Find a recurrence relation for $f(n, m)$.

Hint: whenever you drop an egg, call whichever of the egg breaking/not breaking leads to more drops the "worst-case event". Since we need to find b regardless of its value, you should assume the worst-case event always happens.

(d) Write pseudocode to compute $f(n, m)$. In particular, make sure that your code processes the subproblems in the correct order.

(e) Analyze the runtime complexity of your DP algorithm.

(f) Analyze the space complexity of your DP algorithm.

(g) Is it possible to modify your algorithm above to use less space? If so, describe your modification and re-analyze the space complexity. If not, briefly justify.

Solution:

(a) We have that:

- $f(1, m) = 1$, since we can drop the egg from the single floor to determine if it breaks on that floor or not.
- $f(0, m) = 0$, since there is only one possible value for b.

- $f(n, 1) = n$, since we only have one egg, so the only strategy is to drop it from every floor, starting from floor 1 and going up, until it breaks.
- $f(n, 0) = \infty$ for $n > 0$, since the problem is unsolvable if we have no eggs to drop.

(b) If the egg breaks, we only need to consider floors 1 to $h - 1$, and we have $m - 1$ eggs left since an egg broke, in which case we need $f(h - 1, m - 1)$ more drops. If the egg doesn't break, we only need to consider floors $h + 1$ to n, and there are m eggs left, so we need $f(n - h, m)$ more drops.

(c) The recurrence relation is

$$f(n, m) = 1 + \min_{h \in \{1 \ldots n\}} \max\{f(h - 1, m - 1), f(n - h, m)\}.$$

When we drop an egg at floor h, in the worst case, we need $\max\{f(h-1, m-1), f(n-h, m)\}$ drops. Then, the optimal strategy will choose the best of the n floors, so we need $\min_{h \in \{1 \ldots n\}} \max\{f(h - 1, m - 1), f(n - h, m)\}$ more drops.

(d) We solve the subproblems in increasing order of m, n, i.e.:

```
for j in range(m+1):
    for i in range(n+1):
        solve f(i, j)
```

(e) We solve nm subproblems, each subproblem taking $O(n)$ time. Thus, the overall runtime is $O(n^2 m)$.

(f) The only thing we have to store is the DP array f, which contains nm elements. Thus, the overall space complexity is $O(nm)$.

(g) Yes, it is possible! Notice that in our recurrence relation in part (c), we only need the values of $f(\cdot, m)$ and $f(\cdot, m-1)$. So we can just store the last two "columns" computed so far. The pseudocode for this would look approximately as follows:

```
def eggdrop(n, m):
    if m == 0: # base case for m=0
        return float("inf")
    if n == 0:
        return 0

    curr = [i for i in range(n+1)] # base case for m=1

    for j in range(2, m+1):
        prev = copy(curr)

        for i in range(j+1):
            curr[i] = i
        for i in range(j+1, n+1):
            curr[i] = 1 + min([
                                max(prev[h-1], curr[i-h])
                                for h in range(1, i+1)
```



```
                ])

    return curr[n]
```

3 Longest Common Subsequence

In lecture, we covered the longest increasing subsequence problem (LIS). Now, let us consider the longest common subsequence problem (LCS), which is a bit more involved. Given two arrays A and B of integers, you want to determine the length of their longest common subsequence. If they do not share any common elements, return 0.

For example, given $A = [1, 2, 3, 4, 5]$ and $B = [1, 3, 5, 7]$, their longest common subsequence is $[1, 3, 5]$ with length 3.

We will design an algorithm that solves this problem in $O(nm)$ time, where n is the length of A and m is the length of B.

(a) Define your subproblem in words.

Hint: looking at the subproblem for Edit Distance may be helpful.

(b) Write your recurrence relation and describe your base cases. (A fully correct recurrence relation will always have the base cases specified.)

(c) In what order do we solve the subproblems?

(d) What is the runtime of this dynamic programming algorithm?

(e) Analyze the space complexity of your DP algorithm. Show how to reduce the space complexity of your algorithm to $O(\min\{m, n\})$ additional memory (i.e. not including the input).

Remark: for all space complexity analyses in this class, we only consider writeable auxiliary memory, which does not *include any input if kept read-only.*

Solution:

Algorithm Description: Let $L[i][j]$ be the length of the LCS between the first i characters of A and j characters of B. (i.e between $A[0 : i]$ and $B[0 : j]$). Then the final answer will be $L[n][m]$. The recurrence is given as follows:

$$L[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1 & \text{if } A[i] = B[j] \\ \max\{L[i-1][j], L[j-1][i]\} & \text{otherwise} \end{cases} \quad (1)$$

Correctness: The correctness follows by induction. Observe that if the last characters of A and B match, then any longest common subsequence should have the same last character. Otherwise, at least one of $A[i-1]$ and $B[j-1]$ will not be in the solution, so we take the maximum over the possibilities.

Runtime: Our DP has $(n+1)(m+1)$ states and calculating each state takes $O(1)$. Hence, the algorithm runs in $O(nm)$.

Space: Naively, we can just store all the states, which would take up $O(nm)$ space. However, we can note that to compute any $L[i][j]$, we only need the subproblems involving $i-1$, $j-1$,

and i, j. Thus, we simply need to store the last 2 "rows" or "columns" of the DP table, yielding a space complexity of $O(2\min\{n, m\}) = O(\min\{n, m\})$.

4 Maximum Subarray Sum Revisited

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \le j} \sum_{k=i}^{j} A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$.

In Discussion 2, we saw how to find the maximum subarray sum in $O(n \log n)$ time using divide and conquer. This problem can actually be solved in $O(n)$ time using dynamic programming.

(a) Devise a dynamic programming algorithm solving this problem in $O(n)$ time.

(i) Define your subproblem $f(\cdot)$ in words. Make sure to include how many parameters there are and what they mean, and tell us what input(s) you feed into f to get the answer to your problem.

(ii) Write the recurrence relation for f, as well as the base cases.

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

Hint: for almost all DP correctness proofs, we suggest induction.

(iv) Analyze the runtime and space complexity of your final DP algorithm. Note that the top-down and bottom-up approaches to DP have the same runtime complexity; however, bottom-up can potentially yield a better space complexity.

(b) Describe an $O(n)$-time dynamic programming algorithm to find the maximum subarray sum among subarrays that have an odd number of odd numbers (i.e. $[-3, 4]$ and $[-1, 2, 1, -5]$, but not $[-2], [-2, 1, -3]$, or the empty array). You do not need to provide a proof of correctness or an analysis of runtime or space complexity.

Solution:

(a) **Algorithm Description:**

- Subproblem Definition: $dp[i]$ denotes the maximum sum subarray that ends at index i (non-inclusive of the empty subarray).
- Base Case(s): $dp[0] = A[0]$
- Recurrence Relation: we compute the dp values in increasing order of i (for $i = 1 \dots n-1$) with the recurrence,

$$dp[i] = \max(0, dp[i-1]) + A[i]$$

- Final Answer: for the final answer we output the largest $dp[i]$ or 0 if it is negative.

Here is pseudocode illustrating this algorithm:

```
def max_subarray_sum(A):
    n = len(A)
    dp = [0 for i in range(n)]

    # base case
    dp[0] = A[0]

    # using dynamic programming to compute the rest of dp array
    for i in range(1, n):
        dp[i] = max(0, dp[i-1]) + A[i]

    return max(max(dp), 0)
```

Note that this solution uses $O(n)$ space, as we need to store the entire dp array. However, when computing a given $dp[i]$, we only use the previous $dp[i-1]$, so we can optimize the space complexity of this solution by just storing a "running sum", as follows:

```
def max_subarray_sum(A):
    n = len(A)

    # base case
    max_sub_sum = running_sum = A[0]

    # using dynamic programming to compute the rest of dp array
    for i in range(1, n):
        running_sum = max(0, running_sum) + A[i]
        max_sub_sum = max(max_sub_sum, running_sum)

    return max(max_sub_sum, 0)
```

This solution uses $O(1)$ space since it only has to keep track of three integers n, `max_sub_sum`, and `running_sum`! (Note that we do not count the space complexity of input A when determining this)

Also, if you've done a lot of leetcode before, you may recognize this algorithm as **Kadane's algorithm**.

Proof of Correctness: We use induction to show that $dp[i]$ is correct for all $i \geq 0$.

- Base Case ($i = 1$): when there's just a single element, the only subarray option is $A[0]$.
- Inductive Hypothesis ($i = k$): assume that $dp[k]$ is the correct maximum sum subarray that ends at index k.
- Inductive Step ($i = k \implies i = k + 1$): when finding $dp[k + 1]$, we either want to use $dp[k]$ or ignore it (and start a new subarray sum). Thus, we break it down into two cases:

- When $dp[k] \geq 0$, it's optimal to use it and simply add $A[k+1]$ to try to potentially maximize $dp[k+1]$.
- When $dp[k] < 0$, it's optimal to forget about it and just start a new subarray sum at $k+1$. This is because when $dp[k] < 0$, we have that $dp[k] + A[k+1] < 0 + A[k+1] = A[k+1]$.

The two cases are summarized below:

$$\begin{aligned} dp[k+1] &= \begin{cases} dp[k] + A[k] & dp[k] \geq 0 \\ A[k] & dp[k] < 0 \end{cases} \\ &= \max(dp[k], 0) + A[k] \end{aligned}$$

which is exactly our recurrence relation.

Runtime Analysis: Since we perform $O(n)$ iterations of constant-time operations, the overall runtime is $O(n)$

Space Analysis: The unoptimized DP algorithm uses $O(n)$ space, as we're storing the entire dp array. The optimized DP algorithm (i.e. Kadane's algorithm) uses $O(1)$ space since it only has to keep track of three integers.

(b) **Subproblem Definition:** $dp_o[i]$ denotes the maximum sum subarray with an odd number of odd numbers ending at index i, and $dp_e[i]$ denotes the maximum sum subarray that ends at index i with an even number of odd numbers.

Base Cases: $dp_o[0] = -\infty$, $dp_e[0] = A[0]$.

Recurrence Relation: we compute the dp_o and dp_e values in increasing order of i (for $i = 1 \ldots n-1$) with the recurrence:

$$dp_o[i] = \begin{cases} dp_o[i-1] + A[i] & \text{if } A[i] \text{ is even} \\ \max(A[i], dp_e[i-1] + A[i]) & \text{if } A[i] \text{ is odd} \end{cases},$$

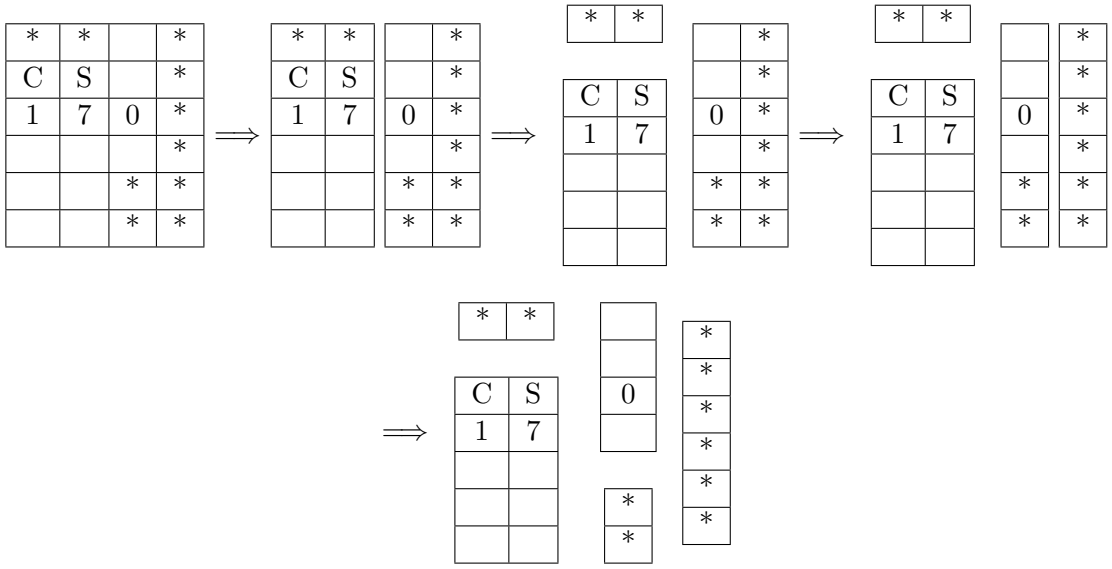
$$dp_e[i] = \begin{cases} \max(dp_e[i-1] + A[i], A[i]) & \text{if } A[i] \text{ is even} \\ dp_o[i-1] + A[i] & \text{if } A[i] \text{ is odd} \end{cases}.$$

Final Answer: for the final answer, we output the largest $dp_o[i]$ over all i.

5 My Dog Ate My Homework

One morning, you wake up to realize that your dog ate some of your CS 170 homework paper, which is an $\ell \times w$ rectangular grid of squares. Some of the squares have holes chewed through them, and you cannot use paper that has a hole in it. You would like to cut the paper into pieces so as to separate all the tattered squares from all the clean, un-bitten squares. You want to do this so that you can save as much as your work as possible.

For example, shown below is a 6×4 piece of paper where the bitten squares are marked with *. As shown in the picture, one can separate the bitten parts out in exactly four cuts.



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

Formally, the problem is as follows:

> **Input:** Dimensions of the paper $\ell \times w$ and an array $A[i, j]$ such that $A[i, j] = 1$ if and only if the ij^{th} square has holes bitten into it.
>
> **Goal:** Find the minimum number of cuts needed so that the $A[i, j]$ values of each piece are either all 0 or all 1.

Design a DP-based algorithm to find the smallest number of cuts needed to separate all the bitten parts out in $O(\ell^3 w^3)$ time. For **extra credit**, try to optimize your algorithm to achieve a runtime of $O((\ell + w)\ell^2 w^2)$.

(a) Define your subproblem.

Hint: try making any arbitrary cut. What two subproblems do you now have? What parameters do you need to properly handle recursing on these two resulting subproblems?

(b) Write down the recurrence relation for your subproblems as well as your base cases. Provide a brief justification of your recurrence relation.

(c) Describe the order in which we should solve the subproblems in your DP algorithm.

(d) What is the runtime complexity of your DP algorithm? Provide a justification.

(e) What is the space complexity of your algorithm? Provide a justification.

Solution:

(a) **Subproblem Definition:** We define $B[i_1, j_1, i_2, j_2]$ to be the minimum number of cuts needed to separate the sub-matrix $A[i_1 \leq i_2, j_1 \leq j_2]$ into pieces consisting either entirely of bitten pieces or clean pieces.

(b) **Recurrence Relation:**

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

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) **Subproblem Order:** we solve them in increasing order of $(i_2 - i_1 + 1)(j_2 - j_1 + 1)$. In other words, we solve all the smallest subproblems first (e.g. containing one square) and build our DP array up to our result $B[1, \ell, 1, w]$, which covers the entire paper.

(d) **Runtime Analysis:** Two answers are acceptable: $O((\ell + w)\ell^2 w^2)$ and $O(\ell^3 w^3)$

We have $O(\ell^2 w^2)$ total subproblems: $O(\ell w)$ possibilities for (i_1, j_1), and $O(\ell w)$ 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 $O(\ell + w)$ time.

In addition, for a subproblem, we also want to check the base case for if the piece is "pure" (contains only clean paper, or contains only bitten paper). Brute force checking this takes $O(\ell w)$ time, for a total subproblem time of $O(\ell w + (\ell + w)) = O(\ell w)$.

Thus, the overall (accepted) runtime is $O(\ell^2 w^2) \cdot O(\ell w) = O(\ell^3 w^3)$.

However, this $O(\ell w)$ factor per subproblem can be reduced to $O(\ell + w)$ (this is not required to receive full points). We can precompute the purities of every single possible subrectangle and store it in a table. Brute-force performs the pre-computation in $O(\ell^3 w^3)$ time, but using prefix sums allows us to do this in just $O(\ell w)$ time. So to solve our recurrence relation, if we can determine purity/impurity in $O(1)$ time (after doing some pre-computation), then we can reach an overall time of $O((\ell + w)\ell^2 w^2)$.

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 (clean-only or bitten-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 $O((\ell + w)\ell^2 w^2)$.

(e) **Space Complexity Analysis:** we have to store the entire DP array for our recurrence relation to work, so the space complexity is $O(\ell^2 w^2)$.

6 [Coding] Dynamic Programming

For this week's homework, you'll implement some dynamic programming algorithms. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw07` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-sp25-coding`

   and navigate to the `hw07` folder. Refer to the `README.md` for local setup instructions.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 7 Coding Portion."
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem threads. To ensure others can help you, make sure to:

  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.

- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.

