CS 170 Homework 7

Due **3/11/2024, 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".

Solution: I worked on this homework with the following collaborators:

- Lakshya Nagal, SID: 3037935253

4-Part Solutions

For all (and only) dynamic programming problems in this class, we would like you to follow a 4-part solution format:

1. **Algorithm Description:** since dynamic programming algorithms can be difficult to explain, you should follow the template below to optimize clarity.
   (a) *Define your subproblem.* In words, define a function f so that the evaluation of f on a certain input gives the answer to the stated problem.
   You should clearly state how many parameters f has, what those parameters represent, what f evaluated on those parameters represents, and what inputs you should feed into f to get the answer to the stated problem.
   (b) *Provide your recurrence relation.* More precisely, give a recurrence relation showing how to compute f recursively, and make sure to provide base cases. If you need to use certain data structures to make computation of f faster, you should say so.
   (c) *Subproblem Ordering:* describe the order in which you should solve the subproblems to obtain the final answer.
2. **Proof of Correctness:** provide some inductive proof that shows why your DP algorithm computes the correct result.
3. **Runtime Analysis:** analyze the runtime of your algorithm.
4. **Space Analysis:** analyze the space/memory complexity of your algorithm.

2 Not Too Much DP

(a) Given an array A with positive or negative integers (i.e. non-zero), we want to find the subarray (i.e. contiguous sequence of elements in the array) that creates the maximum product. We will use 1-dimensional DP to approach this problem where $dp[i]$ will return the maximum product and minimum product seen so far using elements up to (and

including) i-th index from the array. Notice here we need to keep track of the minimum product as well; in case the next element in the array is negative, the minimum product might become the maximum product after new multiplication.

Given the DP subproblems below, perform the following:

0	1	2	3	4	5
$(-2,-2)$	$(3,-6)$	$(24,-12)$	$(48,-24)$	$(24,-48)$	$(96,-192)$

(i) Recover the original array.
Solution:

```
def max_product_iter(A, n):
    max_p = min_p = A[0]
    dp = [(0, 0) for _ in range(n)]
    dp[0] = (A[0], A[0])

    for i in range(1, n):
        if A[i] < 0:
            max_p, min_p = min_p, max_p
        max_p = max(A[i], max_p * A[i])
        min_p = min(A[i], min_p * A[i])
        dp[i] = (max_p, min_p)
    return dp, dp[n-1]
```

Through trial and error and knowing that A[0] = -2 and A[1] = 3, I arrived at the array $A = [-2, 3, -4, 2, -1, 4]$ which did in fact result in the table above.

(ii) Identify the subarray that produces the maximum product.
Solution: The subarray that results in the max product of 96 given A is $[3, -4, 2, -1, 4]$

(b) Given strings s_1, s_2, and s_3, find whether s_3 can be formed by an interleaving of s_1 and s_2. s_3 is defined to be an interleaving of s_1 and s_2 if s_3 contains all of the characters of s_1 and s_2 and only those characters. Additionally, the order of the characters of s_1 and s_2 are preserved in s_3.

Let l_1, l_2, l_3 be the lengths of s_1, s_2 and s_3 respectively. We use 2-dimensional DP to approach this problem where $dp[i][j] = True$ if the substring $s_3[: i + j]$ is an interleaving of substrings $s_1[: i]$ and $s_2[: j]$, and False otherwise.

(i) For this subpart, let s_1 = "cbadb", s_2 = "badda", s_3 = "cbbadadadb". Using those inputs, fill in the missing grids in the following table:

-	0	1	2	3	4	5
0	T	T	T	F	F	F
1	F	T	T	F	F	F
2	F	F	T	F	F	F
3	F	F	T	T	F	F
4	F	F	F	T	F	F
5	F	F	F	T	T	T

Note that for this table, the columns correspond to the characters of s_1 and the rows to s_2. Hence $dp[row][col]$ is true if $s_3[:row+col]$ is a valid interleaving of $s_1[:col]$ and $s_2[:row]$. Additionally, $dp[row][0]$ corresponds to checking whether $s_3[:row] == s_2[:row]$, and $dp[0][col]$ corresponds to checking whether $s_3[:col] == s_1[:col]$.

Solution:

$dp[5][2] = s_1[:5] + s_2[:2] =$ "cbadb" + "bad" $\neq$ "cbbadad" $= s_3[:7] \rightarrow F$
$dp[3][3] = s_1[:3] + s_2[:3] =$ "cba" + "bad" = "cbbada' $= s_3[:6] \rightarrow T$
$dp[2][4] = s_1[:2] + s_2[:4] =$ "cb" + "badd" $\neq$ "cbbada" $= s_3[:6] \rightarrow F$
$dp[4][4] = s_1[:4] + s_2[:4] =$ "cbad" + "bbad" $\neq$ "cbbadada" $= s_3[:8] \rightarrow F$
$dp[5][4] = s_1[:5] + s_2[:4] =$ "cbadb" + "badd" $\neq$ "cbbadadad" $= s_3[:9] \rightarrow F$
$dp[2][5] = s_1[:2] + s_2[:5] =$ "cb" + "badda" $\neq$ "cbbadad" $= s_3[:7] \rightarrow F$
$dp[4][5] = s_1[:4] + s_2[:5] =$ "cbad" + "badda" = "cbbadadad" $= s_3[:9] \rightarrow T$

(ii) For this subpart, you are given s_3 = "sponpdaens" and part of the DP table. Using those information, recover s_1 and s_2. (Note there might be multiple s_1, s_2 combinations that will produce the same table. If multiple combinations are possible, list out all of them.)

-	0	1	2	3	4	5
0	T	-	-	-	-	-
1	-	T	-	-	-	-
2	-	-	T	F	F	-
3	-	-	-	-	T	-
4	-	-	-	-	T	F
5	-	-	-	-	-	T

Solution: The string possibilities are:

s_1 ="podan"	s_2 = "snpes"
s_1 ="pndan"	s_2 = "sopes"
s_1 ="sodan"	s_2 = "pnpes"
s_1 ="sndan"	s_2 = "popes"

We arrive at this conclusion by noting:
$dp[1][1] = T \rightarrow s_3[:2] =$ sp, then...

s_1 ="s"	s_2 = "p"
s_1 ="p"	s_2 = "s"

$dp[2][2] = T \to s_3[:4] =$ spon, and $s_1[:2]$ and $s_2[:2]$ then...

$s_1 =$"po"	$s_2 =$ "sn"
$s_1 =$"pn"	$s_2 =$ "so"
$s_1 =$"so"	$s_2 =$ "pn"
$s_1 =$"sn"	$s_2 =$ "po"

$dp[3][2] = F \to s_3[:5] =$ sponp, and $s_1[:3]$ and $s_2[:2]$ then...

$s_1 =$"po"	$s_2 =$ "snp"
$s_1 =$"pn"	$s_2 =$ "sop"
$s_1 =$"so"	$s_2 =$ "pnp"
$s_1 =$"sn"	$s_2 =$ "pop"

$dp[4][2] = F \to s_3[:6] =$ sponpd, and $s_1[:4]$ and $s_2[:2]$ then...

$s_1 =$"poda"	$s_2 =$ "snpp"
$s_1 =$"pnda"	$s_2 =$ "sopp"
$s_1 =$"soda"	$s_2 =$ "pnp"
$s_1 =$"snda"	$s_2 =$ "pop"

We continue on like this eventually arriving at the mentioned possibilities for s_1 and s_2.

(iii) For this subpart, determine whether the following subtables are possible (subtable is simply a small part of the entire table). Give a brief justification/reasoning to your answer.

1.
T	T
T	T

Solution: This can be true as can be seen in the table from part (i).

2.
T	F
F	T

Solution: This is true. Consider the strings from above,"podan", "snpes". We have $dp[4][4] = T, dp[5][4] = F, dp[5][5] = T$, and $dp[4][5]$ = interleaving "poda" and "snpes" the best we can do is "sponpdae" $\neq$ "sponpdaen" $= s_3[:9] \to F$ giving us a subtable like the one here.

3.
T	F
T	F

Solution: This can be true as can be seen in the table from part (i).

3 Egg Drop

You are given m identical eggs and an n story building. You need to figure out the highest floor $h \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor h or lower, and always breaks if dropped from floor $h+1$ or higher. ($h = 0$ 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 h (regardless of the value of h).

(a) Find $f(1, m)$, $f(0, m)$, $f(n, 1)$, and $f(n, 0)$. Briefly explain your answers.
Solution:
$f(1, m) = 1$. We have two possible results, it either breaks on the first floor or it doesn't, in either case the minimum number of egg drops is 1.
$f(0, m) = 0$. If we are on floor 0 the minimum number of egg drops needed is 0.
$f(n, 1) = n$. If we only have 1 egg, then we have to drop the egg starting from the first floor, until he breaks or we reach the n^{th} floor.
$f(n, 0) = -\infty$. If we have 0 eggs, then the minimum number of egg drops we can perform is undefined, which will be represented by $-\infty$.

(b) Consider dropping an egg at floor x 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 h in terms of $f(\cdot, \cdot)$, n, m, and/or x.
Solution:

- $f(n - x, m)$ if the egg does not break at the x^{th} floor, we know h is at a floor above, so check $x + 1, x + 2, \ldots, n$.
- $f(x - 1, m - 1)$ if the egg does break, we know we have found h, or its at a lower floor, so we check floor $x - 1, x - 2, \ldots, 1$.

(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 h regardless of its value, you should assume the worst-case event always happens.
Solution:
$f(n, m) = 1 + min\{max\{f(n - x, m), f(x - 1, m - 1)\}\}$, regardless of whether the egg the broke, we used an egg so add we add 1. We take the max because in the worst case we have n floors and we have 1 egg. In this case we have to linearly check every floor which will give us the worst number of egg drops but guarantees that we find h with the given egg. Then we take the minimum over that to find the minimum number of egg drops, after considering every floors possibilities.

(d) If we want to use dynamic programming to compute $f(n, m)$ given n and m, in what order do we solve the subproblems?
Solution: We would want to solve the problems in a bottom up way so that we solve all the smaller subproblems before solving the bigger subproblems in order to use the

information from the smaller subproblems to efficiently compute the larger ones. In other words, find all the bases cases first and use that to solve $f(1,1), f(2,1), ..., f(n,m)$

(e) Based on your responses to previous parts, analyze the runtime complexity of your DP algorithm.
Solution: We are going to have $O(nm)$ subproblems with each subproblem taking at most $O(n)$ to compute giving us a total runtime of $O(n^2m)$.

(f) Analyze the space complexity of your DP algorithm.
Solution: We need to store all the intermediate values of n, m so our space complexity is $O(nm)$.

(g) (**Extra Credit**) 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:

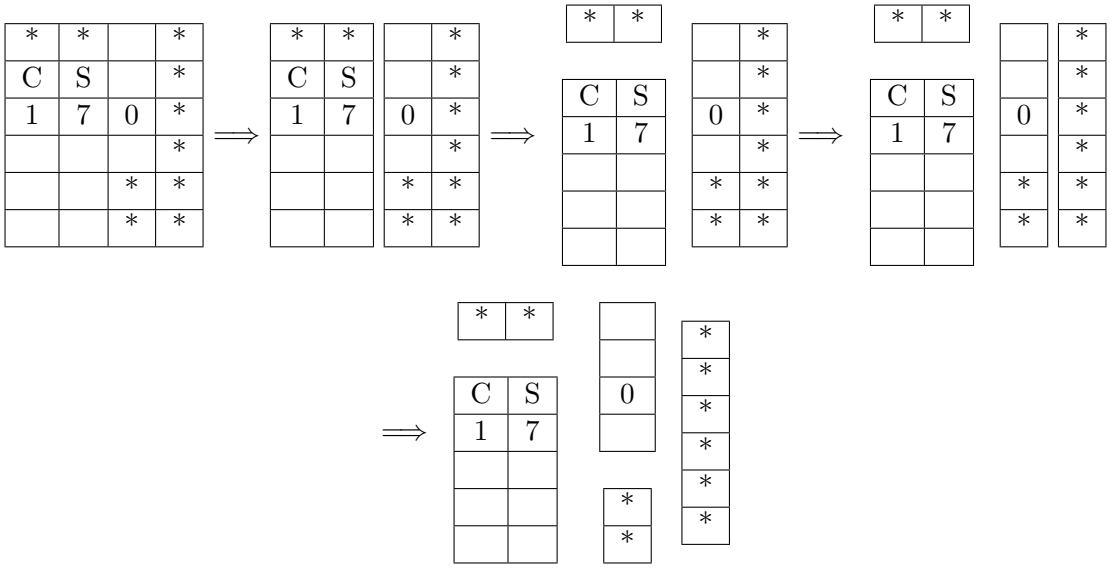
-	0	1	2	3	$\cdots$	m
0	0	0	0	0	$\cdots$	0
1	$-\infty$	1	1	1	$\cdots$	1
2	$-\infty$	2	★			
3	$-\infty$	3				
$\vdots$	$\vdots$	$\vdots$				
n	$-\infty$	n				

For ★ at $2,2$ or subproblem $f(2,2)$, we only need to consider the two suproblems $f(2,1)$ and $f(2,2)$, generally for, subproblem $f(n,m)$ we only need to consider subproblems $f(n-k,m)$ for some $0 \leq k \leq n$, and $f(n-1,m-1)$, so we only need to store the current m, and previous n floors which is bounded by $O(n)$.

4 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 $m \times n$ 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.)

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

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

(a) Define your subproblem.

Hint: try making any arbitrary cut. What two subproblems do you now have?

Solution: Let $f(m, n, x, y)$ be the minimum number of cuts given a matrix of size $m \times n$ and the top right corner coordinate (x, y). The subproblems arise when making horizontal or vertical cuts, resulting in submatrices with dimensions $m' \times n$ or $m \times n'$.

(b) Write down the recurrence relation for your subproblems. A fully correct recurrence relation will always have the base cases specified.

Solution:

$$f(m,n,x,y) = \begin{cases} 0, \text{ if } n = 0 \text{ or } m = 0 \\ 0, \text{ if } \forall_{x,y} P[x][y] = 0 \text{ or } \forall_{x,y} P[x][y] = 1 \\ \min \begin{cases} 1 + f(m,n',x',y') + f(m,n',x',y') \\ 1 + f(m',n,x',y') + f(m',n,x',y') \end{cases} \end{cases}$$

Where $m \times n', m' \times n$ are the new dimensions of the a submatrix when performing a vertical and horizontal cut, and (x', y') are the new top right corner coordinates of the submatrices after the cut. To find the new coordinates we can use the indices of the matrix where we are performing the cut to denote the left submatrix or bottom submatrix, then we can subtract the length of the new right or top submatrix to find the starting coordinates of that submatrix.

(c) Describe the order in which we should solve the subproblems in your DP algorithm.
Solution: The problems should be solved bottom up to find the minimum cuts with a smaller subarrays and use this information to solve for bigger subarrays.

(d) What is the runtime complexity of your DP algorithm? Provide a justification.
Solution: we have $O(mn)$ subproblems, for each subproblem we need to identify whether or not preform a cut, which is bounded by $O(mn)$, then we need to find the minimum which could cost $O(n)$. For a total of $O(m^2n^3)$.

(e) What is the space complexity of your algorithm? Provide a justification.
Solution: We store the results in an $m \times n$ array, so our space complexity is $O(mn)$.

5 Knightmare

Give a dynamic programming algorithm to find the number of ways you can place knights on an L by H ($L < H$) chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). Knights can move in a 2×1 shape pattern in any direction.

Provide a 4-part solution. Your algorithm's runtime should be $O(2^{3L}LH)$, and return your answer mod 1773.

Hint: if a knight is on row i, what rows on the chessboard can it affect?

Solution:

Algorithm Description:

(a) Let $f(row, bitString_{row}, bitString_{row-1})$ be the number of ways we can place knights on row of the grid, where $bitString_{row}$ and $bitString_{row-1}$ is a possible configurations of row and previous row.

(b)

$$f(i, bitString_i, bitString_{i-1})_{0 \le i < H} = \begin{cases} 1, \text{ if } row = 0 \\ 2^L, \text{ if } row = 1 \\ f(i-1, bitString_{i-1}, bitString_i) + \\ f(i-2, bitString_{i-2}, bitString_i), \text{otherwise} \end{cases}$$

where $f(i-1, bitString_{i-1}, bitString_i) = \sum \mathbb{1}_{is\_valid(bitString_i, bitString_{i-1})}$ over all possible bit strings for $bitString_i$ and $bitString_{i-1}$. $is\_valid(a, b)$ checks if two bit strings are a valid configuration of knights. Additionally we return this mod 1773.

(c) The order in which the subproblems should be solved should be bottom up, so that we can solve the smaller subproblems before trying to solve the bigger subproblems. In this manner, we can take advantage of the dp table to compute larger subproblems.

Proof of Correctness: Essentially what is happening, is that I am generating every possible string for a row n, then I am fixing that string and checking all possible configurations for the $n-1$ row that are valid with the current configuration of row n, and checking that it is also valid for all possibilities for $n-2$ row. I make use of that fact that for any row n, I only need to ensure that the $n-1, n-2, n+1, n+2$ rows are valid given row n's configuration.
Runtime Analysis: The time to check if two strings of length L are a valid configuration is $O(L)$, for each of the three rows we compare, there are 2^L possibilities which is $O(2^{3L})$. We also iterate through every row which is O(H), giving us a total runtime of $O(2^{3L}LH)$.
Space Complexity: We maintain an $H \times L$ grid, so our space complexity is $O(HL)$.