Please enter your name and uID below.

Name: Devin White

uID: U0775759

Collaborators, if any, and how you collaborated:

Submission notes

- Due at 11:59 pm on Thursday, September 29.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for this problem set, you are allowed to collaborate in detail with your peers, as long as you cite them. However, you must write up your own solution, alone, from memory. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Narrow art gallery)

The recursive formula is as follows:

$$\text{nGall(r,k,uncloseable)}=\begin{cases} 0 & k=0 \\ -\infty & k>n-r \\ & k==n-r \\ max\begin{cases} (narrowGallery(r+1,k-1,0)+R[r,0] \\ (narrowGallery(r+1,k-1,1)+R[r,1] \end{cases} & uncloseable=-1 \\ (narrowGallery(r+1,k-1,0)+R[r,0] & uncloseable=0 \\ (narrowGallery(r+1,k-1,1)+R[r,1] & uncloseable=1 \\ & otherwise \\ max\begin{cases} (narrowGallery(r+1,k-1,0)+R[r,0] \\ (narrowGallery(r+1,k,-1)+R[r,1]+R[r,0] \end{cases} & uncloseable=0 \\ max\begin{cases} (narrowGallery(r+1,k-1,1)+R[r,1] \\ (narrowGallery(r+1,k,-1)+R[r,0]+R[r,1] \end{cases} & uncloseable=1 \\ max\begin{cases} (narrowGallery(r+1,k-1,0)+R[r,0] \\ (narrowGallery(r+1,k-1,1)+R[r,1] \\ (narrowGallery(r+1,k,-1)+R[r,1]+R[r,0] \end{cases} & uncloseable=-1 \end{cases}$$

This algorithm assumes that initially we have a 2d Array R[i][j] of size R[n][2] which holds the value of the applicable room. i represents the rows, while j represents 1 of the 2 possible columns, so R[0[0] would be the bottom left-most room. R[n][1] the topmost right. Essentially, during each recursive step, the algorithm is checking the values of the current room(or rooms if neither are closed) and taking the maximum value. If the room on the right is closed, then the room on the left can't be closed, and the room on the next row on the left also can't be closed to allow for passage, similarly on the right. However, if both doors on a given row are open, either the left or right doors on the next row may be closed if that leads to the maximum value. This is what the recursions "find out" as the optimal "path" through the gallery

In order to optimize this recursive algorithm and make it dynamic, we can create a table using a 3D array of Size T[n,r+1,3] this allows us to store the rows, the values, and whether the door on the L was closed, the R was closed, or neither was closed(the [3]), with both open being an index of 2 in this case). This memo can be used to store the results of the recursion, but also be used in changing the algorithm to a fully iterative dynamic one. When storing values within this table, it is filled bottom up, since the final values of the rooms rely upon those which are below them. Essentially, this can be done with 2 for loops, an inner loop j, and an outer loop i, looping through the inner loop j first until it reaches the number i(so once when on row 0, twice on row 1, etc) this accounts for the increasing possible "paths" as we move upwards throughout the gallery. The outer loop, representing the rows, we iterate through second as each j finishes. Ultimately looking something on each iteration like: T[i][j][0] = ..., T[i][j][1] = ..., and T[i][j][2] = ..., with each of the 3 possible "room states" calculated during each loop. The run-time of this algorithm comes out to $O(n^2)$ due to the nature of two nested loops. While j starts out looping a small amoun, it eventually goes larger and larger during each iteration until it loops through the entire size of n, while i loops through the entire size of n, giving us a runtime of Approximately $O(n^2)$ compared to the $O(2^n)$ runtime of the recursive version.

2. (String splitting)

   (a) Assumptions:

   - partition only increments if every word in the partition is a full word
   - can be split into subproblems E.g is A[0..k-1] a word? is A[k...n] a word? A[0..k-2]? etc.
   - The algorithm is dynamic(*efficient algorithm*), divided into subproblems
   - for any input i,j isWord() returns true if A[i..j] is a word. False otherwise.

   $$wordsplit(A[0..n]) = \begin{cases} 1 & n = 0 \\ V_{j=i}^{n-1} IsWord(i,j) wordsplit(j+1) & isWord(i,j) = true \\ 0 & otherwise \end{cases}$$

   This algorithm works as follows:
   It's a simple variation of the stringsplit algorithm provided in the book.(assuming I got the recursive formula correct) Basically, for recursion, there is a loop which increments through the array from i = 0 to n, if a call to isWord returns true, it means that there is a word there and there is a possible partition solution, and it recurses at j+1 to check for a larger range(for the DP solution, or tightening on i for the recursive solution), while the starting char increments with each loop iteration. if it reaches the end of the loop and there never is a word, then the count is never incremented, otherwise, the count is incremented when it reaches the end, as that is a viable partition. The counting either can be done with a global counter, or with the count incrementing from the value returned during one of the recursive calls.

   The recursive problem being solved can be thought of like so: you start at i of the given string and try to find the longest possible word, using Artistoil, it would essentially be asking for the first iteration, is a a word? no. Ok, then is ar a word? no ... until you get to art. Art is a word. so you find a partition there and keep note of that. artis isn't a word, but artist is a word, so it keeps track and so on. You then move the "start" to i+1 and repeat the process, so now it's is rt a word? rti? etc. until we are done with every possible string of characters from 0 to n(in order). In other words, your sub problems are essentially seeing the smallest range of what constitutes a "word" as that determines your partition point, with a series of words in a row determining the amount of possible partitions.

   In order to make this an efficient DP algorithm, the structure needed is a simple array of size [n+1][n+1](n+1 to account for base cases) to basically store the count for a possible "end paths"(remember, count is only incremented if the entire string array is split into words as determined by isWord). This is due to the fact that a word of n characters only ever has to account for those n characters and the base cases. This table would be filled with 2 for loops, i and j, storing the counter of possible partitions based upon looping through isWord.i starts at n-1 and ends at 0, while j would start at i, and end at n So for example, for the first iteration, it would be A[n][i] and check isword(n..n) then isword(n-2..n-1) and so on. until it ends the loop at isword(0...0), where our result is sotred in A[0], or the very first index, depending on whether it uses 0 or 1 based indexing.

   This algorithm would have a runtime of $O(n^2)$ due to in nested for loops. the outer i loop is only iterated over a single time, while the inner loop j is incremented over multiple times, Additionally, isWord is called n times for each loop. n times for i, and n times for j(as the amount of times it is called in j is reduced upon each iteration. it reduces in time for each increment of i, making the end result $O(n*n)$ or n^2

(b) Check if A[0..n] and B[0..n] have the same partitions:

$$wordsplit2(A[0..n], B[0..n]) = \begin{cases} False & LengthA! = LengthB \\ True & nA = 0 \text{ AND } nB = 0 \\ V_{j=i}^{n-1} isWord(A[i..j]) & isWord(A[0..n]) = True \\ isWord(B[i..j] wordsplit2(j+1) & \text{ AND } isWord(B[0..n]) = True \\ False & otherwise \end{cases}$$

The recursive version of this algorithm functions very similarly to, and is merely an altered version of the string split algorithm used in the book. The big change here in the recursive formula is the fact that this algorithm looking at the same indexes of Both A[0..n] and B[0..n] at the exact same time. If the loop is completed in its entirety, then it is true that both arrays can be split at the same indices, otherwise, it is not. Rather than simply looking at a single array, it now has twice the problems to go through on each recursion, making a recursive method EXTREMELY sub-optimal.
The recursive function being solved here is similar to the one in part A. However, the added conditional of needing them to be split at the exact same indices for the larger problem to be true means that there is a double requirement of both isWord(A[0..n]) AND isWord(B[0..n]) to be true. If the partition locations do not match exactly, they will never be true. Esentially this means it's checking whether A and B are/have words of the exact same lengths split at the same exact points, as they could only possibly be partitioned at the same indices if they have AT LEAST ONE WORD of the same length at the each of the possible partition points. if one array or the other has smaller or larger words at different indices or not is completely irrelevant in the grand scheme. The sub problem that is the concern of this algorithm is merely "Are wordA and WordB the same length at the same point in the array"

To convert the recursive version of this algorithm to a much more efficient DP algorithm, the memoization/value table would need to be a 2D array C of size C[n+1][n+1], this is because we need to know the results of the sub-problems(recursive calls) of both A[0.n] and B[0..n] as both must be true in the same points in order for our final solution to return true, otherwise there is no possible way for it to be true. as such, each A and B have their own subproblems to keep track of, which will be compared in the end to form the solution to our larger problem.

Similar to question 1, this array would be filled in decreasing index order, with a nested for loop, where i begins at n-1(inclusive) and ends at 0, and j begins at i and ends at n-1(inclusive). During each loop iteration, the results of A and B would be stored simultaneously at A[i][0] and B[i][1] respectively, in typical dynamic programming fashion, these results would be added to what would normally function as the recursive call during each iteration of the for loops until eventually, These results when compared and built upon themselves will culminate in our final answer when i reaches the 0th index, with the final solution being in the 0th index.

The running time of this is a little more interesting as isWord is called 2 TIMES for every single iteration of J. and since the algorithm is running inside a nested loop, we can expect that there will be n calls during the i loop, and two n calls during the j loops(since j iterations decrease as i increases), one for each of the two arrays we are comparing. This should give a run time something along the lines of $O(2n^2)$ this is much faster than the 2^n calls to isWord of the recursive version.

(optional second page for String splitting)

This class is so hard. :(