Homework 7

Due Oct. 21st, 2021

Note This homework assignment covers dynamic programming from Klienberg and Tardos.

1. Given a non-empty string s and a dictionary containing a list of unique words, design a dynamic programming algorithm to determine if s can be segmented into a space-separated sequence of one or more dictionary words. If s="algorithmdesign" and your dictionary contains "algorithm" and "design". Your algorithm should answer Yes as s can be segmented as "algorithmdesign".

   Let $s_{i,k}$ denote the substring $s_i s_{i+1} \ldots s_k$. Let $Opt(k)$ denote whether the substring $s_{1,k}$ can be segmented using the words in the dictionary, namely $OPT(k) = 1$ if the segmentation is possible and 0 otherwise. A segmentation of this substring $s_{1,k}$ is possible if only the last word (say $s_i \ldots s_k$) is in the dictionary the remaining substring $s_{1,i}$ can be segmented. Therefore,we have equation:

   $$Opt(k) = \max_{0<i<k \text{ and } s_{i+1,k} \text{ is a word in the dictionary}} Opt(i)$$

   We can begin solving the above recurrence with the initial condition that $Opt(0) = 1$ and then go on to compute $Opt(k)$ for $k = 1, 2, \ldots, n$. The answer corresponding to $Opt(n)$ is the solution and can be computed in $\Theta(n^2)$ time.

2. Solve Kleinberg and Tardos, Chapter 6, Exercise 10.

   (a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4 minutes are listed as follows (in time order):

- Machine A: 2, 1, 1, 200
- Machine B: 1, 1, 20, 100

The given algorithm will choose A then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

(b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i, you should not move in minute i, because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i, consider that if you are on machine A in minute i, you either (i) stay on machine A in minute $i-1$ or (ii) are in the process of moving from machine B to A in minute $i-1$. Now let $OPT_A(i)$ represent the maximum value of a plan in minute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute $i-1$, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have

$$OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\} :$$

Similarly, we get the recursive relation for $OPT_B(i)$:

$$OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\} :$$

The algorithm initializes $OPT_A(0) = 0, OPT_B(0) = 0, OPT_A(1) = a_1$ and $OPT_B(1) = b_1$. Then the algorithm can be written as follows:

```
$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;
**for** $i = 2, \cdots, n$ **do**
  $OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$;
  Record the action (either stay or move) in minute $i-1$ that achieves the maximum.
  $OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$;
  Record the action in minute $i-1$ that achieves the maximum.
**end for**
Return $\max\{OPT_A(n), OPT_B(n)\}$;
Track back through the arrays $OPT_A$ and $OPT_B$ by checking the action records from minute $n-1$ to minute 1 to recover the optimal solution.
```

It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.

3. Solve Kleinberg and Tardos, Chapter 6, Exercise 24.
   The basic idea is to ask: How should we gerrymander precincts 1 through j, for each j? To make this work, though, we have to keep track of a few extra things, by adding some variables. For brevity, we say that A-votes in a precinct are the voters for part A and B-voter are the votes for part B. We keep track of the following information about a partial solution.

   - How many precincts have been assigned to district 1 so far?
   - How many A-votes are in district 1 so far?
   - How many A-votes are in district 2 so far?

   So let $M[j, p, x, y] = true$ if it is possible to achieve at least x A-votes in distance 1 and y A-votes in district 2, while allocating p of the first j precincts to district 1. Now suppose precinct $j + 1$ has z A-votes. To compute $M[j+1, p, x, y]$, you either put precinct $j+1$ in district 1 (in which case you check the results of sub-problem $M[j, p-1, x-z, y]$) or in district 2 (in which case you check the results of sub-problem $M[j, p, x, y - z]$). Now to decide if there's a solution to the while problem, you scan the entire table at the end, looking for a value of $true$ in any entry from $M[n, n/2, x, y]$ where each of x and y is greater than $mn/4$. (Since each district gets $mn/2$ votes total).

   We can build this up in the order of increasing j, and each sub-problem takes constant time to compute, using the values of smaller sub-problems. Since there are n^2, m^2 sub-problems, the running time is $O(n^2m^2)$.

4. We initialize another matrix (dp) with the same dimensions as the original one initialized with all 0's.

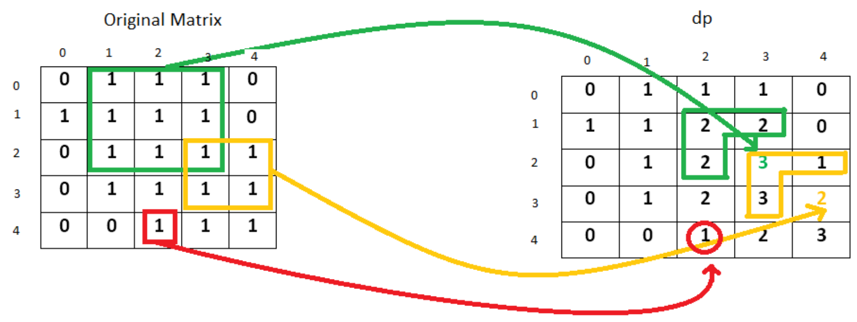
   dp(i,j) represents the side length of the maximum square whose bottom right corner is the cell with index (i,j) in the original matrix.

   Starting from index (0,0), for every 1 found in the original matrix, we update the value of the current element as

$$dp(i, j) = \min\big(dp(i-1, j), dp(i-1, j-1), dp(i, j-1)\big) + 1.$$

   We also remember the size of the largest square found so far. In this way, we traverse the original matrix once and find out the required maximum size. This gives the side length of the square (say maxsqlen). The required result is the area $maxsqlen^2$

   An entry 2 at (1, 3) implies that we have a square of side 2 up to that index in the original matrix. Similarly, a 2 at (1, 2) and (2, 2) implies that a square of side 2 exists up to that index in the original matrix. Now to make a square of side 3, only a single entry of 1 is pending at (2, 3). So, we enter a 3 corresponding to that position in the dp array.

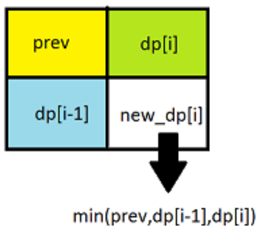


Now consider the case for the index $(3, 4)$. Here, the entries at index $(3, 3)$ and $(2, 3)$ imply that a square of side 3 is possible up to their indices. But, the entry 1 at index $(2, 4)$ indicates that a square of side 1 only can be formed up to its index. Therefore, while making an entry at the index $(3, 4)$, this element obstructs the formation of a square having a side larger than 2. Thus, the maximum sized square that can be formed up to this index is of size 2×2.

To reduce space complexity:

for calculating dp of i^{th} row we are using only the previous element and the $(i-1)^{th}$ row. Therefore, we don't need 2D dp matrix as 1D dp array will be sufficient for this.

Initially the dp array contains all 0's. As we scan the elements of the original matrix across a row, we keep on updating the dp array as per the equation $dp[j] = min(dp[j-1], dp[j], prev)$, where prev refers to the old dp[j-1]. For every row, we repeat the same process and update in the same dp array.


prev
dp[i]
dp[i-1]
new_dp[i]
min(prev,dp[i-1],dp[i])