7️⃣

HW7

▼ **Question 1**

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".

Solution

Let OPT(k) = whether substring s(1..k) can be segmented using the given words in the dictionary. We can apply this recurrence formula -

$OPT(k) = max(OPT_{i<=k}(i), OPT_{k+1,j}(i))$. We partition the string at every index and check whether the substring is present in the given dictionary or not. Only if the substring is present, can the given string be segmented.

Base condition will be $O(0) = 1$ because in case there is no string present, it will always be present in the dictionary. Time complexity is $O(n^2)$

▼ **Question 2**

10. You're trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available.

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute, with the property that choices A and

B cannot appear in consecutive minutes. For example, if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i+1$ must be *move*, and then your choice for minute $i+2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem. Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
In minute 1, choose the machine achieving the larger of a_1, b_1
Set i = 2
While i ≤ n
   What was the choice in minute i − 1?
   If A:
      If b_{i+1} > a_i + a_{i+1} then
         Choose move in minute i and B in minute i + 1
         Proceed to iteration i + 2
      Else
         Choose A in minute i
         Proceed to iteration i + 1
      Endif
   If B: behave as above with roles of A and B reversed
EndWhile
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

Solution

a. Machine A = 2, 1, 1, 200 and Machine B = 1, 1, 20, 100 Here, the optimal solution will stay on A for all the 4 steps but the given algorithm will shift on Machine B for the final 2 steps. This is a counter example

b.

$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;
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.

Overall complexity is $O(n)$ i.e. $O(1)$ for each of n iterations.

▼ **Question 3**

24. *Gerrymandering* is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

Computers, it turns out, have been implicated as the source of some of the "villainy" in the news coverage on this topic: Thanks to powerful software, gerrymandering has changed from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process that it is today. Why is gerrymandering a computational problem? There are database issues involved in tracking voter demographics down to the level of individual streets and houses; and there are algorithmic issues involved in grouping voters into districts. Let's think a bit about what these latter issues look like.

Suppose we have a set of n *precincts* $P_1, P_2, \ldots, P_n$, each containing m registered voters. We're supposed to divide these precincts into two *districts*, each consisting of $n/2$ of the precincts. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose, for simplicity, that every voter is registered to one of these two.) We'll say that the set of precincts is *susceptible* to gerrymandering if it is possible to perform the division into two districts in such a way that the same party holds a majority in both districts.

Give an algorithm to determine whether a given set of precincts is susceptible to gerrymandering; the running time of your algorithm should be polynomial in n and m.

Example. Suppose we have $n = 4$ precincts, and the following information on registered voters.

Precinct	**1**	**2**	**3**	**4**
Number registered for party A	55	43	60	47
Number registered for party B	45	57	40	53

This set of precincts is susceptible since, if we grouped precincts 1 and 4 into one district, and precincts 2 and 3 into the other, then party A would have a majority in both districts. (Presumably, the "we" who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: Although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

Solution

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-problemM[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 fromM[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 n2;m2 sub-problems, the running time is O(n2m2).

▼ **Question 4**

4. You are given an $m \times n$ binary matrix. Each cell either contains a "0" or a "1". Find the largest square containing only 1's and return its area. Can you improve your solution to use $O(n + m)$ space?

Solution

Create a new 2D array of the same size as that of the given binary matrix. Copy the 1st row and column as it is.

The following DP formula can be applied

$$dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1])$$

as this will limit the largest square that can be created by the values that are limited by the calculations done previously. To decrease the number of times the 2D array is traversed, we can keep a variable that keeps a record of the maximum area that has been achieved up until that point (i, j)

This variable, $largestarea$ is the answer that returns the largest area of the $m * n$ binary matrix where $dp[i][j] = 1$

The solution can be improved to use $O(m+n)$ space by only keeping a 1D array instead of a 2D array, as we only need the $i-1$th row and $j-1$th column for calculating the value of $dp[i][j]$

Code

```
// let dp[1..n][1..n] be a new 2D array
// populate 1st row and 1st column with the same elements as in the origninal binary array
// define a variable to keep track of the largest area, largearea
for i in 1 to matrix.length do
  for j in 1 to matrix[0].length do
    dp[i][j] = 1 + Math.min(dp[i - 1][j], dp[i][j - 1])
    largearea = Math.max(largearea, dp[i][j]

return largearea
```