Dynamic Programming

1. In a top-down approach to dynamic programming, the larger subproblems are solved before the smaller ones. True/False

   Solution: False. The larger problems depend on the smaller ones, so the smaller ones need to be solved first. The smaller problems get solved (and memoized) recursively as part of a larger problem.

2. Any Dynamic Programming algorithm with n subproblems will run in O(n) time. True False

   Solution**:** False. The subproblems may take longer than constant time to compute.

3. Imagine starting with the given number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The *square-depth* S(n) of n is defined to be the maximum number of square numbers you can arrange to see along the way. For example, S(32492) = 3 via the sequence

   $$32492 \to 3249 \to 324 \to 24 \to 4$$

   since 3249, 324, and 4 are squares, and there is no better sequence of chops giving a larger score (although you can do as well other ways, since 49 and 9 are both squares).

   Describe an efficient algorithm to compute the *square-depth*, S(n), of a given number n, written as a d-digit decimal number $a_1a_2 \ldots a_d$. Analyze your algorithm's running time.
   Your algorithm should run in time polynomial in d. You may assume a function IS-SQUARE(x) that returns, in constant time, 1 if x is square, and 0 if not.

   Solution: We can solve this using dynamic programming.

   First, we define some notation: let $n_{ij} = a_i \ldots a_j$. That is, n_{ij} is the number formed by digits i through j of n. Now, define the subproblems by letting $D[i, j]$ be the square-depth of n_{ij}.

   The solution to $D[i, j]$ can be found by solving the two subproblems that result from chopping off the left digit and from chopping off the right digit, and adding 1 if n_{ij} itself is square. We can express this as a recurrence:

   $$D[i, j] = \max(D[i+1, j], D[i, j-1]) + \text{IS-SQUARE}(n_{ij})$$

   - The base cases are $D[i, i]$ = IS-SQUARE(a_i), for all $1 \le i \le d$.
   - The solution to the general problem is S(n) = $D[1, d]$.

   There are $\Theta(d^2)$ subproblems, and each takes $\Theta(1)$ time to solve, so the total running time is $\Theta(d^2)$.

4. The Borg want to find the ***largest square parking space*** in which to park the Borg Cube. That is, find the largest k such that there exists a $k \times k$ square in A containing all ones and no zeros.

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 1 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 0 & 0 \\
4 & 1 & 1 & 1 & 0 & 1
\end{array}$$

In the example figure, the solution is 3, as illustrated by the 3×3 highlighted box.

Describe an efficient algorithm that finds the size of the largest square parking space in A. Analyze the running time of your algorithm.

Hint: Call $A[0][0]$ be the *top-left* of the parking lot, and call $A[n-1][n-1]$ the *bottom-right*. Use dynamic programming, with the subproblem $S[i, j]$ being the side length of the largest square parking space whose bottom-right corner is at (i, j).

Solution:

The base cases are the positions along the top and left sides of the parking lot. In each of these positions, you can fit a 1×1 square parking space if and only if the space is unoccupied. Thus, for the base cases ($i = 0$ or $j = 0$), we have $S[i, j] = A[i][j]$.

Now for the general case. Again, if $A[i][j] = 0$, then $S[i, j] = 0$, so we'll only consider the case where $A[i][j] = 1$. There is a parking space of size x with bottom-right corner at (i, j) *if and only if* there are three (overlapping) spaces of size $x - 1$ at each of the locations $(i - 1, j)$, $(i, j - 1)$, and $(i - 1, j - 1)$. Thus, the largest possible parking space is

$S[i, j]=1 + min(S[i-1, j], S[i, j-1], S[i-1, j-1])$ The desired answer is $max_{ij}\, S[i, j]$, which requires $O(n^2)$ to compute.

There are n^2 subproblems, and each takes $O(1)$ time to solve, for a time of $O(n^2)$. This, added to the $O(n^2)$ to extract the desired answer, gives a total $O(n^2)$ running time, which is clearly optimal because all the data must be examined.

5. In dynamic programming, we derive a recurrence relation for the solution to one sub- problem in terms of solutions to other subproblems. To turn this relation into a bottom- up dynamic programming algorithm, we need an order to fill in the solution cells in a table, such that all needed subproblems are solved before solving a subproblem. For each of the following relations, give such a valid traversal order, or if no traversal order is possible for the given relation, briefly justify why.

   i. $A(i,j)=F(A(i,j-1), A(i-1,j-1), A(i-1,j+1))$
   ii. $A(i,j)=F(A(\min\{i,j\}-1,\min\{i,j\}-1), A(\max\{i,j\}-1,\max\{i,j\}-1))$
   iii. $A(i,j)=F(A(i-2,j-2), A(i+2,j+2))$

Solution:

i. Solve $A(i, j)$ for (i from 0 to n: for(j from 0 to n))
ii. Solve $A(k, k)$ for (k from 0 to n) then solve rest in any order
iii. Impossible: cyclic.

7. Describe an algorithm, using dynamic programming, to find the number of binary strings of length N which do not contain any two consecutive 1's. For example, for N=2, the algorithm should return 3 as the possible binary strings are 00, 01 and 10. State the set of subproblems that you will use to solve this problem and the corresponding recurrence relation to compute the solution.

Solution:

Let a[i] be the number of binary strings of length i which do not contain any two consecutive 1's and which end in 0. Similarly, let b[i] be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

$$a[i] = a[i-1] + b[i-1] \quad b[i] = a[i-1]$$

The base cases of our recurrence are a[1] = b[1] = 1. The total number of strings of length i is just a[i] + b[i].

8. Obert Ratkins is having dinner at an upscale tapas bar, where he will order many small plates which will comprise his meal. There are n plates of food on the menu, with each plate p_i having integer volume v_i, integer calories c_i, and may or may not be sweet. Obert is on a diet: he wants to eat no more than C calories during his meal, but wants to fill his stomach as much as he can. He also wants to order exactly s sweet plates, for some $s \in [1, n]$, without purchasing the same dish twice. Describe a dynamic programming algorithm to find a set of plates that maximizes the volume of food Obert can eat given his constraints.

Be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subproblems, and analyze running time.

Solution:

- Let s_i be 1 if plate p_i is sweet and 0 otherwise
- $x(i, j, k)$: largest volume of food orderable from plates p_i to p_n, using at most j calories, ordering exactly k sweet plates

- Either we order plate p_i or not. Guess!
  - If order p_i, get v_i in volume but use c_i calories
  - If p_i is sweet, need to order one fewer sweet plate

$$x(i, j, k) = \max \begin{cases} v_i + x(i+1, j-c_i, k-s_i) & \text{if } c_i \le j \text{ and } s_i \le k \\ x(i+1, j, k) & \text{otherwise} \end{cases}$$

- Note that subproblems $x(i, j, k)$ only depend on strictly larger i so acyclic, so we can solve in order of decreasing i, then j and k in any order
- Base case: for all $j \in [\,0, C]$, $x(n+1, j, 0) = 0$ and $x(n+1, j, k) = -\infty$ for $k \neq 0$
- Solution given by $x(1, C, s)$
- Time Analysis: (# Subproblems) × (work per problem) = $O(n\ C\ s) \times O(1) = O(n\ C\ s)$

Reference:

From "6.006: Introduction to Algorithms", MIT.