1 Dynamic Programming

1. **Binary Strings.** Let $\#C(\ell)$ denote the number of binary strings with length ℓ that have no consecutive occurrences of a 1. For example $\#C(3) = 5$; we can list all binary strings of length 3 and determine by inspection that only the strings 000, 001, 010, 100, and 101 have no consecutive occurrences of a 1.

   (a) Compute $\#C(5)$.

   Solution: $\#C(5) = 13$.

   (b) Describe why a brute-force approach used to compute $\#C(5)$ above is problematic.

   Solution: Enumerating every binary string of length ℓ becomes intractible as ℓ becomes large - there are exponentially many of them.

   (c) Show that $\#C(\ell) = \#C(\ell-1) + \#C(\ell-2)$ for $\ell \geq 2$. Derive a recursive algorithm to compute $\#C(\ell)$.
   Hint: You will need to add base cases.

   Solution: Observe that for a string of length $\ell \geq 2$, it must either begin with a 1 or a 0. If it begins with a 1, then the next bit must be a 0, and the remaining string has length $\ell - 2$ and must have no consecutive occurrences of a 1. If it begins with a 0, then the next bit may be any bit, and the remaining string has length $\ell - 1$ and must have no consecutive occurrences of a 1.
   As such, we have that $\#C(\ell) = \#C(\ell-1) + \#C(\ell-2)$ for $\ell \geq 2$.
   If we want to compute this recursively, we only need to add base cases. Observe that $\#C$ is only defined for non-negative integers, so our base cases should be $\#C(0) = 1$ and $\#C(1) = 2$.

   Input: Unsigned integer n

```
1: function #C(n)
2:     if n = 0 then
3:         return 1
4:     if n = 1 then
5:         return 2
6:     return #C(n − 1) + #C(n − 2)
```

   (d) Use the bottom-up-table approach to improve your recursive algorithm.

   Solution: We improve the algorithm by storing values of $\#C(n)$ in an array; $\#C(n)$ goes in the n-th index of the array. For any n, we start with the base cases $\#C(0)$ and $\#C(1)$, and we compute each subsequent $\#C(i)$ from $\#C(i-1)$ and $\#C(i-2)$ until we get to $\#C(n)$.

This recurrence is exactly the recurrence for the Fibonacci sequence. The bottom-up algorithm for the two is exactly the same with the only difference being in the base cases.

Input: Nonnegative integer n

```
1: function #C(n)
2:     DP ← a table of n integers
3:     DP[0] ← 1
4:     DP[1] ← 2
5:     for i = 2 to n do
6:         DP[i] ← DP[i − 1] + DP[i − 2]
7:     return DP[n]
```

Note that as for the Fibonacci sequence, the whole table need not be stored, since we only need the last two computed values at any point in time.

2. **Odd Tiling.** Luffy wants to tile a path of length n using k types of blocks with distinct lengths $\ell_1, \ldots, \ell_k$ (note that we have an infinite supply of each type of block). He wants to know if there are an even or odd number of ways of tiling such a path. You aim to devise an algorithm for answering this question in $O(nk)$ time.

   As a starting point, let $ODD(i)$ = True if and only if there is an odd number of ways to tile a path of length i using the same k types of blocks.

   (a) Give the recursive formulation for $ODD(i)$ and briefly argue why it is correct. Be sure to include the base case(s).

   Solution: Our base case is $n = 0$. There is one way to tile a path of length $n = 0$: by using zero tiles. If there is one way, there is an odd number of ways, thus $ODD(0) = 1$.

   Now consider path length $i > 0$ and some arbitrary block length ℓ_j where $1 \leq j \leq k, i$. Assume that ℓ_j is the last tile placed in the path. Then $ODD[i - \ell_j]$ tells us whether the number of ways to tile the path in this manner is odd. We can try this for each of the k blocks that are at most the length of the path. Summing these calls to ODD and taking the mod 2 will tell us whether the sum of all the ways of tiling is odd, because even though we're not finding the actual number of tilings, we preserve the "oddness" of the number by returning 1 in odd cases and 0 in even cases. $ODD(i) = \Sigma_{j=1}^{k} ODD[i - \ell_j] \mod 2$ (only when $i - \ell_j \geq 0$)

   (b) Implement a bottom-up dynamic programming algorithm that solves this problem. Provide pseudocode and analyze the runtime of your algorithm.

   Solution: Below is the pseudo-code corresponding to the recurrence.

Input: Path length $n \geq 0$, Block lengths $\{\ell_1, ..., \ell_k\}$

```
1: function ODD(n, {ℓ_1, ..., ℓ_k})
2:     initialize array memo of length n
3:     memo[0] = 1
4:     for i = 1 to n do
5:         memo[i] = 0
6:         for j = 1 to k do
7:             if i − j_k ≥ 0 then
8:                 memo[i] = memo[i] + memo[i − j_k] mod 2.
9:     return memo[n]
```

The running time is $O(nk)$ because of the two nested loops of n and k iterations, respectively. Each iteration takes $O(1)$ time.

3. **Smallest Subarray Product.** Given an array of $n \geq 1$ *positive real numbers*, $A[1..n]$, we are interested in finding the smallest *product* of any subarray of A, i.e.,

$$\min\{A[i]A[i+1]\cdots A[j] \mid i \leq j \text{ are indices of } A\}.$$

Describe a dynamic programming algorithm to solve this problem in $O(n)$ time. Your solution should include a recurrence, an explanation of why it is correct, and a pseudocode implementation using a bottom-up table with runtime analysis.

Solution: Note that we'll use 1-based indexing for this solution. For each $1 \leq i \leq n$, let $S(i)$ be the smallest product that *ends* at position i of A. Then we see that

$$S(i) = \begin{cases} A[1] & i = 1 \\ \min\{S(i-1) \cdot A[i], A[i]\} & i > 1 \end{cases}.$$

Indeed, the key observation is that, since the product needs to end at position i, we should only extend the previous product if it doesn't hurt to do so (i.e., when $S(i) \leq 1$; this observation can actually lead to a simpler algorithm that does not require a table). Given this recurrence, we may compute the smallest product as $\min\{S(i) \mid 1 \leq i \leq n\}$. The bottom-up table implementation is as follows.

```
function MinProd(A[1..n])
    allocate S[1..n]
    S[1] ← A[1]
    for i = 2..n do
        S[i] ← min{S[i − 1] · A[i], A[i]}
    return min{S[i] | 1 ≤ i ≤ n}
```

We see that this algorithm consists of a loop across $n - 1 = O(n)$ elements with constant work being done in each iteration, so this is an $O(n)$ algorithm.

4. **Matrix Multiplication.** ($M[1, \ldots, n]$): Given a sequence of matrices to multiply together, write a recurrence to determine the minimum number of element-element multiplications required (you may disregard additions).

   As an example, suppose the sequence of matrices to multiply is ABC, where A is 2×3, B is 3×4, and C is 4×5.

   Computing $(AB)C$ requires $2 \cdot 3 \cdot 4 + 2 \cdot 4 \cdot 5 = 64$ multiplications. Computing $A(BC)$ requires $3 \cdot 4 \cdot 5 + 2 \cdot 3 \cdot 5 = 90$ multiplications. Therefore, the minimum number of multiplications required is 64.

   The following is a short example of how to perform matrix multiplication:
   If $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ and $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$, then $AB = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$.

   Remarks: Matrix multiplication is associative. Given two matrices with dimensions $m \times n$ and $n \times p$, we require mnp multiplications because the resulting matrix is of dimension $m \times p$ and each element requires n multiplications. You can read more about matrix multiplication here: `https://en.wikipedia.org/wiki/Matrix_multiplication`

   Solution: Let us define $L(i, j)$ to be the minimum number of multiplications required to multiply the i^{th} through j^{th} matrices. The base cases are when we only have a single matrix, i.e. $i = j$, in which case no multiplications are required: $L(i, i) = 0$ for each i.

   Let each $M[i]$ be of the form (r, c), where r is the number of rows and c is the number of columns in the matrix. Then

   $$L(i, j) = \min_{i \leq k < j} (L(i, k) + L(k + 1, j) + M[i].r \cdot M[k].c \cdot M[j].c).$$

   This recurrence will find the minimum by recognizing that each sequence of matrices that are multiplied have a final multiplication, therefore, by checking each final breakpoint, we can find the breakpoint that corresponds to the minimum number of multiplications.

5. **Longest Palindromic Subsequence.** Given a string S, write a recurrence relation to determine the length of a longest subsequence of S (not necessarily a substring) that is a palindrome. Recall that a palindrome is a string which is the same forwards and backwards (e.g., "racecar").

   Solution: For an input string $S[1 \ldots n]$, define the function $LPS[i, j]$ for $1 \leq i \leq j \leq n$ to be the length of a longest palindromic subsequence of $S[i \ldots j]$. This indicates that we are only interested in subproblems that correspond to substrings of S. Clearly, $S[1, n]$ is the length of a longest palindromic subsequence of all of S.

   We have the following base cases: $S[i, i] = 1$ for all i, because a single character is a palindrome, and $S[i, j] = 0$ when $i > j$.

   For the recursive case, if $S[i] = S[j]$, then a longest palindromic subsequence of $S[i \ldots j]$ includes those two characters at the beginning and end, with an l.p.s. of $S[i + 1, j - 1]$

between them. Otherwise, an l.p.s. of $S[i \dots j]$ can't include both of $S[i]$ and $S[j]$ (it wouldn't be a palindrome), so it is the longer of an l.p.s. of $S[i+1 \dots j]$ and of $S[i \dots j-1]$.

All this yields the recurrence

$$LPS[i,j] = \begin{cases} 2 + LPS[i+1, j-1] & \text{if } S[i] = S[j] \\ \max\{LPS[i+1, j], LPS[i, j-1]\} & \text{otherwise.} \end{cases}$$

6. **Michigania Tug-of-war.** Over the summer, Aayush took a job as a staff member at *Camp Michigania*. The campers complained that the traditional tug-of-war game between North camp and South camp was unfair (go North camp!) and now are trying to balance a new tug-of-war game. Aayush wants to split the campers in half to so that the total strength of the campers on each side is as even as possible. Each camper's strength is a positive integer, and each camper must be placed on either the North side or the South side.

   The problem is formalized as follows: given n campers, where camper i has strength $s_i \in \mathbb{N}$, find a subset $S \subseteq \{1, \dots, n\}$ to go on the South side (the rest go to the North side) so that the difference

   $$\left| \sum_{i \in S} s_i - \sum_{i \notin S} s_i \right|$$

   between the two sides' total strengths is minimized.

   (a) The naïve brute-force algorithm is: "try all possible subsets of campers for the South side; choose the one having the smallest strength difference." Express the running time of this algorithm in big-Theta notation.

   Solution: There are 2^n possible choices of subsets for the South side. For each choice, we then compare the strength of the campers on the south side to that of those on the north side. Finding these two sums takes $\Theta(n)$ time. So, finding the smallest difference takes $\Theta(n \cdot 2^n)$ time.

   (b) Give, with justification, a recurrence relation that is suitable for a dynamic programming solution to the problem. Be sure to identify the base case(s).

   Solution: To employ dynamic programming, we can observe that we only need to determine if a total can be reached by some subset of people. This is similar to the 0-1 knapsack problem where we considered subsets of items and attempted to maximize the total value of the subset.
   Let $A_j(s) = 1$ if some subset of the first j campers have total strength s, and let $A_j(s) = 0$ otherwise. Then we may observe that (similar to what we have seen in discussion), the following recurrence holds:

   $$A_j(s) = A_{j-1}(s) \vee A_{j-1}(s - s_j).$$

   Furthermore, we have

   $$A_j(s) = \begin{cases} 1 & \text{if } s = 0 \\ 0 & \text{if } j = 0 \text{ and } s \neq 0 \end{cases}$$

(c) Give pseudocode for a (bottom-up) dynamic programming algorithm that solves the problem, and analyze its running time.

Solution: Lines 1 through 8 fill in *memo* so that $memo[i][j] = 1$ if and only if there is a subset $S \subseteq \{1, \ldots, i\}$ such that $\sum_{i \in S} s_i = j$.
Lines 9 through 11 find the strength of one side such that

$$|(\text{strength on the south side}) - (\text{strength on the north side})|$$

would be minimized.

Input: Integer strengths $(s_1, s_2, \ldots, s_n)$, $C = \sum_{i=1}^{n} s_i$, and *memo* is an $(n+1) \times (C+1)$ array, initially consisting of 0's in each entry. For simplicity of notation, the rows and columns are 0 indexed (i.e. from 0 to n and from 0 to C).
Output:

```
1:  function MichiganiaPartition((s_1, s_2, ..., s_n), memo[n+1][C+1])
2:      memo[0][0] ← 1
3:      for i from 1 to n do
4:          for j from 0 to C do
5:              if memo[i−1][j] = 1 then
6:                  memo[i][j] = 1 // Represents excluding i
7:                  memo[i][j + s_i] = 1 // Represents including i
8:      b ← ⌊C/2⌋ // The best strength we can put in one side
9:      while memo[n][b] ≠ 1 do
10:         b ← b − 1
11:     return b
```

The running time of this algorithm is $O(nC)$ because there are $O(nC)$ loops, each of which takes $O(1)$ time.