1. **Herding a Dumb Horse Across a Chessboard** One is interested in the paths of the knight (according to the chess game) over a board with $n > 4$ rows and $m > 3$ columns. One wants to know the number of distinct ways to go from the square $(1,1)$ (departure) to the square (n,m) (arrival). By convention, (i,j) stands for the square of the board located on row i and column j. In contrast to the chess game where the knight has eight possible moves (except if it would go outside the board), it is imposed that the knight moves only in order to increase the column index (the second one), as shown in Figure 1.

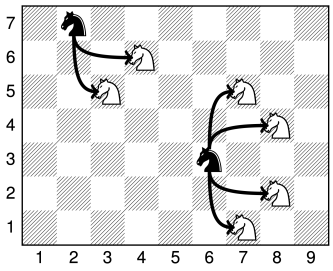

Figure 1. The knight located in $(7,2)$ (respectively $(3,6)$), plotted in black, can move to the two (respectively four) positions, plotted in white.

Let $\text{nbRout}(i,j)$ be the count of distinct paths starting in $(1,1)$ and ending in (i,j). Give the complete recurrence for calculating $\text{nbRout}(i,j)$.

Solution:

$$\text{nbRout}(i,j) = \begin{cases} 1 & \text{if } (i,j) = (1,1) \\ \text{nbRout}(i-2,j-1)+ & \\ \text{nbRout}(i+2,j-1)+ & \\ \text{nbRout}(i-1,j-2)+ & \\ \text{nbRout}(i+1,j-2) & \text{if within board boundaries} \\ 0 & \text{otherwise} \end{cases}$$

Suppose Sumedh was a horse on the board. Neigh. Neigh. Suppose he was exactly one step move away from reaching (i,j) on the board. What are all the places he could be standing at?

Right before his last move, he could be:

- one space up and 2 spaces to the left: $(i-1,j-2)$
- one space down and 2 spaces to the left: $(i+1,j-2)$
- two spaces up, and one space to the left: $(i-2,j-1)$
- two spaces down, and one space to the left: $(i+2,j-1)$

Now suppose by magic he knew how many paths there were to his second to last location. I know, I know. Just suppose he knew.

If he knows how many paths there are takes from $(1,1)$ to his second to last location, well he just needs to add 1 to get the total number of paths to his second to last

location. But wait, how do we compute the number of steps from $(1,1)$ to the second to last location? We call nbRout

If he ends up off the board after his last move, well, that means he took an invalid path, so we add o to the count of valid paths.

■

2. **Don't lift the Da*n Pen!** We want to draw n doubly nested squares as follows.

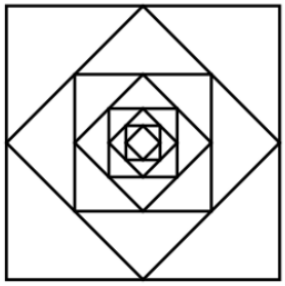

Figure 2. A drawing involving four nested double squares.

- at the end of the drawing, the pen must be back to its starting position,
- the pen must not be raised during the drawing and no segment can be drawn more than once (neither idle time, nor useless work).

a. Identify the possible starting point(s) of the drawing.

b. Write a recursive algorithm to make this drawing.

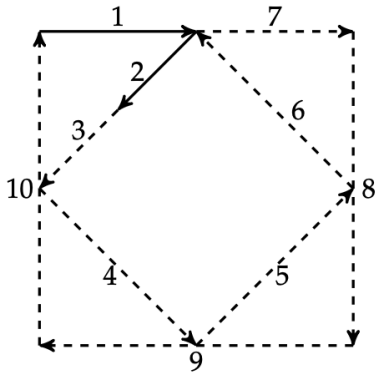

Figure 3. An example of a drawing trajectory for nested double squares.

Solution: The possible starting points include all 4 corners of any non-rotated square or any point of contact bewteen two sqaures. Why? We need to start the drawings of size n and size $n-1$ at similar points, geometrically speaking. This means that from the starting point of the size n drawing, we should be able to reach the corresponding point in the size $n-1$ drawing. For this to happen, the starting point for the size $n-1$

drawing must be a common point between the two drawings.

Intuition: For square n, start from a corner. Go right for $\frac{l}{2}$ steps, then rotate trajectory -45 degrees. Recursively draw square $n-1$. Un-rotate 45 degrees. Finish square n. Figure 3

NestDbSqDr(n) draws n nested squares such that the midpoints of the edges of the outer square form the $(n-1)^{th}$ square.

Answer: Simply call **NestDbSqDr(n)**

$$\text{NestDbSqDr}(n) \rightarrow \begin{cases} \text{elementary (nothing to draw)} & n = 0, \\ \text{start the drawing of the 'outermost" double square} \\ + \\ \text{NestDbSqDr}(n-1) \\ + \\ \text{finish the drawing of the 'outermost" double square} & n > 0. \end{cases}$$

■

3. **Array-nageddon: The Ultimate Backtracking Hellscape** Let $T[1..n]$ ($n \geq 1$) be an array of real non-negative values ($T \in 1..n \to \mathbb{R}^+$). There are at least two indices, i and j, defining the interval $i..j$, with $1 \leq i \leq j \leq n$, such that the value of the expression $T[j] - T[i]$ is maximum. We are looking for this maximum value (the value of the best interval). The special case where $i = j$ characterizes a monotonic array strictly decreasing: the value searched for is then zero. Give a recurrence to solve this problem! Make sure to give an English description.

Solution: MaxDiff(k) finds the maximum value of $T[j] - T[i]$ for $k \leq i \leq j \leq n$ in $T[k \ldots n]$.

Intuition Let's say I'm trying to find the maximum difference in the suffix of the array T, starting from k and ending at n (i.e., the slice $T[k \ldots n]$).

Suppose Lord Rama appeared and told me the maximum difference between any two elements in the array slice from $T[k+1]$ to $T[n]$.

Then, the largest difference is either (1) the value given by Lord Rama, or (2) the difference between the smallest number in $T[k \ldots n]$ and $T[k]$.

Oh wait! Rama's number is just MaxDiff($k+1$). So, I try both and take the maximum value. Also, the maximum difference between $T[n] - T[n]$ is just 0.

$$\text{MaxDiff}(k) = \begin{cases} 0 & \text{if } k = n \\ \max(\text{MaxDiff}(k+1), \max_{k \leq m \leq n}(T[m] - T[k])) & \text{if } k < n \end{cases}$$

■

4. **Turning Your Recursion into a Memoization Mess** Describe a memorization order and data structure to convert your answer from to Problem 3 from a recursive backtracking solution into a Dynamic Programming Solution.

Solution: We can use a 1D array of size n. Since MaxDiff(k) depends on MaxDiff($k+1$), we need MaxDiff($k+1$) pre-computed before MaxDiff(k). Therefore, we fill the 1D array from right to left. Note that we can use a second array, let's say max_val, where max_val[k] stores the maximum value up until index k to speed up the computation. ■

5. **Turning Alphabet Soup into a Clusterf*k** Let C be a set of m words on the alphabet Σ, all with lengths less than or equal to k (C is called the code). We also have another word D of length n on the alphabet Σ, which we try to encode using the fewest possible occurrences of words of C. For example, if $C = \{a, b, ba, abab\}$ and $D = babbaababa$, a possible encoding of D is $ba\ ba\ b\ ba$, using six occurrences of C. There may be no solution, as for the encoding of $D = abbc$ with the code $C = \{a, bc\}$. A sufficient condition for any string to be encoded (and thus admit optimal encoding) is that Σ be included (in a broad sense) in C.

Solution: MS(k) computes the minimum number of occurrences of words from C needed to split D[1 ... k], or returns infinity if D is not decomposable.

Intuition I'm trying to find the minimum number of words from C needed to split $D[1 \ldots k]$. Suppose, by magic, I knew the word w, with length len(w), that would minimize the number of splits needed.

So, I know that the minimum number of splits for $D[1 \ldots k]$ is 1+ the minimum number of splits needed for $D[1 \ldots k - \text{len}(w)]$.

But how do I find w? Just try all of them.

If $k - \text{len}(w) < 0$, I went too far back and there is no split – so return infinity.

$$\text{MS}(k) = \begin{cases} 0 & \text{if } k = 0 \\ \infty & \text{if } k < 0 \\ 1 + \min\{\text{MS}(k - \text{len}(w)) \mid w \in C \text{ and } D[k - \text{len}(w) + 1 \ldots k] = w\} & \text{if } k > 0 \end{cases}$$

Answer: MS(|D|) computes the minimum number of occurrences of words from C needed to split D[1 ... |D|], or returns infinity if D is not decomposable.

Runtime: There are $O(|D|)$ total ways to call the function. Each call requires $O(|C|)$ checks to find the appropriate w. Each check to see if w is an exact substring takes $O(\max\{\text{len}(w) \mid w \in C\})$. So, clearly, the total runtime is:

$$O(|D| \times |C| \times \max\{\text{len}(w) \mid w \in C\})$$

Memoization and Space Complexity There are $O(|D|)$ ways to call MS. We use a 1D array. Since MS(k) depends on smaller values of MS, we must compute and fill in these prior values first. Therefore, the table is filled from left to right. The space complexity is $O(|D|)$.

■