1. **Largest Square of 1's** You are given a $n \times n$ bitonic array A and the goal is to find the set of elements within that array that form a square filled with only 1's.

$$
\begin{array}{cc}
 & j \rightarrow \\
i \downarrow & \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\end{array}
$$

Figure 1: Example: The output is the sidelength of the largest square of 1's (4 in the case of the graph above, yes there can be multiple squares of the greatest size).

Solution: We observe that a square of size n is composed of 3 squares of size $n-1$ plus the corner piece (assuming it's value is a 1). For example we can re-imagine the example above as:

$$
\begin{array}{cc}
 & j \rightarrow \\
i \downarrow & \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0
\end{bmatrix}
\end{array}
$$

So we can construct the recurrence as follows:

$$
LSq(i,j) = \begin{cases}
0 & \text{if } A[i,j] = 0 \qquad \text{(1a)} \\
A[i,j] & \text{if } i = n \text{ or } j = n \qquad \text{(1b)} \\
1 + \min \begin{Bmatrix} LSq(i+1,j) \\ LSq(i,j+1) \\ LSq(i+1,j+1) \end{Bmatrix} & \text{otherwise} \qquad \text{(1c)}
\end{cases}
$$

$LSq(i,j)$ describes the maximum square of 1's whose top left corner is at coordinate index $[i,j]$. Each of the recurrence cases can be described as:

- 1a is a base case. If $A[i,j] = 0$, then it can't be part of a square of 1's and hence the maximum square size is 0.

- 1b is another base case. The values on the bottom row can have a square (whose top-left is at a point on that row) or more than 1. So we set the values accordingly. Same logic applies for the rightmost column.
- 1c is the recurrence. If $A[i, j] = 1$, then there is the possibility we can connect it to the neighboring squares to form a new even larger square. We do this by taking the minimum sized square from the neighbors to bottom/right since we can only have 1's inside the new square.

The output is the max of all the possible square in the array max($LSq(1..n, 1..n)$)

We know that each computation of $LSq(1..n, 1..n)$ looks at the values to the bottom and right so we can memoize the array in reverse row-major order going from bottom to top, right to left. The pseudo-code looks-like:

```
LSq(A[1..n, 1..n]):
  LSq = zeros(n,n)
  for $i \leftarrow 1$ to $n$
      $LSq[n, i] \leftarrow A[n, i]$
      $LSq[i, n] \leftarrow A[i, n]$
  for $i \leftarrow n - 1$ down to 1
      for $j \leftarrow n - 1$ down to 1
          if $A[i, j] \neq 0$
              $LSq[i, j] \leftarrow \min\{LSq[i + 1, j], LSq[i, j + 1], LSq[i + 1, j + 1]\}$
          else
              $LSq[i, j] \leftarrow 0$

      return max($LSq$)
```

■

2. The traditional world chess championship is a match of 24 games. Each game ends in a win, loss, or draw (tie) where a win counts as 1 point, a loss as 0 point, and a draw as 1/2 point. The current champion retains the title in case he scores 12 or above. The players take turns playing white and black. In the first game, the champion plays white. The champion has probabilities w_w, w_d, and w_l of winning, drawing, and losing playing white, and has probabilities b_w, b_d, and b_l of winning, drawing, and losing playing black.

   (a) Write a recurrence for the probability that the champion retains the title. Assume that there are g games left to play in the match and that the champion needs to get i points (which may be a multiple of 1/2).

Solution: Let us define the recurrence with $\mathrm{P}_w(g,i)$ and $\mathrm{P}_b(g,i)$ that gives the probability of victory where the champion retains their title with g games left to play and with i points to be won, playing with white or black respectively.

The recurrence relation can be given by

$$P_w(g,i) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } g = 0 \\ w_w + w_d + w_l * P_b(g-1,i) & \text{if } i = 0.5,\ g \neq 1 \\ w_w + w_d & \text{if } i = 0.5,\ g = 1 \\ \{w_w P_b(g-1,i-1) + w_d P_b(g-1,i-0.5) & \\ \qquad\qquad + w_l P_b(g-1,i)\} & \text{otherwise} \end{cases} \tag{2}$$

$$P_b(g,i) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{if } g = 0 \\ b_w + b_d + b_l * P_w(g-1,i) & \text{if } i = 0.5,\ g \neq 1 \\ b_w + b_d & \text{if } i = 0.5,\ g = 1 \\ \{b_w P_w(g-1,i-1) + b_d P_w(g-1,i-0.5) & \\ \qquad\qquad + b_l P_w(g-1,i)\} & \text{otherwise} \end{cases} \tag{3}$$

- Case 1 $(i = 0)$: This means the champion has already won all the points needed to retain his title guaranteeing victory and hence $P_w(g,0) = 1$.
- Case 2 : $(g = 0)$: There are no more games left to be played and if $i \neq 0$, the champion still has not won all the points in the end resulting in failure, making $P_w(0,i) = P_w(0,i) = 0$.
- Case 3: $(i = 0.5, g \neq 1)$: If the champion only has 0.5 points left for victory, either winning(+1) or ending the game in a tie(+0.5) would give them enough points to win the title and there is no need to play further. But, if they lost the current game, they continue playing the next game but with the opposite color.
- Case 4: $(i = 0.5, g = 1)$: If the champion only has 0.5 points left for victory and there is only one more game left to be played, the probability of victory is only contributed by the probability of either winning that game or ending it in a tie and the champion can stop there.

- Otherwise: When the champion has more games to play and more than 0.5 points left, the probability for victory is a sum of the probabilities of all cases - a win, loss or a tie. If the champion wins a game, the total number of points to be won is reduced by 1 for the next game. Similarly, if the champion ends the current game in a tie, there are $(i-0.5)$ points left to be won and in case of a loss there is no change to the points left for victory for the consequent games. In all cases, we need to switch colours for the next game.

■

(b) Based on your recurrence, give a dynamic programming algorithm to calculate the champion's probability of retaining the title.

Solution: For the DP algorithm we use two memoization tables, one each for the colour being played, either black or white, where the rows represent the number of games left to be played and the columns represent the number of points to be won. Since the points can have a multiple of 1/2 we have $2n$ colums where the k^{th} column represents $k/2$ points. The champion needs to score atleast n/2 points to win n games as ties are also considered as a win. So the algorithm returns W[n][n].

Since we are given that the championship consists of 24 games, we call the algorithm as Win(24).

```
WIN($n$):
    W = zeros($n+1, 2n+1$)
    B = zeros($n+1, 2n+1$)
    for $g \leftarrow 0 : n$
        $W[g,0] = 1$
        $B[g,0] = 1$
    for $i \leftarrow 1 : n$
        $W[0,i] = 0$
        $B[0,i] = 0$
    for $g \leftarrow 0 : n$
        $W[g,1] = ww + wd + wl * B[g-1][1]$
        $B[g,1] = bb + bd + bl * W[g-1][1]$
        for $i \leftarrow 2 : n$ down to 1
            $W[g][i] = ww * B[g-1, i-2] + wd * B[g-1, i-1] + wl * B[g-1, i]$
            $B[g][i] = bw * W[g-1, i-2] + bd * W[g-1, i-1] + bl * W[g-1, i]$
    return $W[n][n]$
```

■

(c) Analyze its running time for an n game match where the champion needs to get $n/2$ points to retain the title.

Solution: The running time of our algorithm for a match with n games would be $O(n^2)$ ■

3. Plum blossom poles are a Kung Fu training technique, consisting of n large posts partially sunk into the ground, with each pole pi at position (xi, yi). Students practice martial arts techniques by stepping from the top of one pole to the top of another pole. In order to keep balance, each step must be more than d meters but less than 2d meters. Give an efficient algorithm to find a safe path from pole ps to pt if it exists.

Solution: We will have an input of list of n xy-coordinates, the value for minimum distance d, source coordinate and destination coordinate. We will use this data to build a graph by calculating the Euclidean distance between every pair of coordinates and adding an undirected edge between pairs where the distance lies in the range of d to 2d. This algorithm will take $O(n^2)$. We will use a BlackBox algorithm $BFSPath$ that takes a Graph, source and destination vertices and returns True if a path exists between the two points and False if path does not exist. The runtime complexity of running the BFS algorithm will be $O(V+E)$ where V is the number of vertices and E is the number of edges.

- Each vertex is (x_i, y_i) representing the xy coordinates of the Plum blossom poles
- An edge between 2 vertices indicates that the distance between the vertices is between d and $2d$. They are undirected.
- Since we are determining whether a path exists, we do not need to have a value associated with the edge.
- The problem we are trying to solve is whether a path lies between two points in the graph. Whether we can start at a given vertex and traverse through the graph and reach the destination vertex.
- We can use a DFS or BFS algorithm to check whether a path exists between two vertices.

```
BuildGraph($A[(x_1, y_1), (x_2, y_2), \dots(x_n, y_n)], d$):
    Let $g \leftarrow$ Empty Graph with n nodes
    for $i \leftarrow 1$ to $n-1$
        for $j \leftarrow i+1$ to $n$
            dist = $SquareRoot((x_i - x_j)^2 + (y_i - y_j)^2)$
            if $d <= dist <= 2d$
                add an Undirected edge between node i and j


    return $g$
```

```
PlumBlossomPath($A[(x_1, y_1), (x_2, y_2), \dots(x_n, y_n)], d, src, dest$):
    graph = $BuildGraph(A, d)$
    $PathExists = BFSPath(graph, src, dest)$
    return $PathExists$
```

■

4. Suppose you are given an array $A[1..n]$ of arbitrary real numbers. Recall a *subarray of an array A* is by definition a *contiguous* subsequence of A. Define the sum and product of an empty array to be 0 and 1, respectively. For any array $A[i..j]$ where $i \le j$, define its sum and product to be

$$\sum_{k=i}^{j} A[k] \quad \text{and} \quad \prod_{k=i}^{j} A[k],$$

respectively. For the sake of analysis, assume that comparing, adding and multiplying any pair of numbers takes $O(1)$ time.

   (a) Describe and analyze an algorithm to compute the *maximum sum* of any subarray of A.

Solution: This problem is a very widely used computer science technical interview question and the algorithm described below, which is the fastest known algorithm for this problem, is called Kadane's algorithm. See https://en.wikipedia.org/wiki/Maximum_subarray_problem#Kadane's_algorithm for more details.

Let $\mathit{MaxSum}(i)$ denote the maximum sum of any subarray of *A that begins with* $A[i]$. $\mathit{MaxSum}(i)$ satisfies the following recurrence:

$$\mathit{MaxSum}(i) = \begin{cases} 0 & \text{if } i > n \\ \max\{0, A[i] + \mathit{MaxSum}(i+1)\} & \text{otherwise} \end{cases}$$

We need to compute $\max\limits_{1 \le i \le n} \mathit{MaxSum}(i)$. We can memoize this function in an array $\mathit{MaxSum}[1..n]$, where $\mathit{MaxSum}[i]$ is assumed to memoize $\mathit{MaxSum}(i)$. Because the subproblem at index i depends only on the subproblem at index $i+1$, if the subproblems are evaluated in the order right to left, each subproblem will have its dependencies computed by the time the algorithm reaches it. Because each subproblem takes time $O(1)$ to evaluate and there are $O(n)$ subproblems, it takes time $O(n)$ to compute $\mathit{MaxSum}(i)$ for all $1 \le i \le n$. Since the maximization $\max\limits_{1 \le i \le n} \mathit{MaxSum}(i)$ takes time $O(n)$ to compute thereafter, the algorithm runs in ***time O(n)*** overall. The pseudocode for this algorithm is given below:

```
MaxSum($A[1..n]$):
    $MaxSum[n+1] \leftarrow 0$    ⟨⟨Base case⟩⟩
    $max \leftarrow MaxSum[n+1]$
    for $i \leftarrow n$ down to 1:
        $MaxSum[i] \leftarrow \max\{0, A[i] + MaxSum[i+1]\}$
        $max \leftarrow \max\{max, MaxSum[i]\}$
    return $max$
```

■

(b) Describe and analyze an algorithm to compute the *maximum product* of any subarray of $A[1\,..\,n]$.

Solution: We follow a similar approach as in Kadane's algorithm given above, but we track the *minimum* product *less than* 0 of subarrays starting at any particular index in addition to the maximum product. To this end, we define two functions:

- Let $\mathit{MaxProd}^+(i)$ denote the maximum product of any subarray of A *that begins with* $A[i]$.
- Let $\mathit{MaxProd}^-(i)$ denote the minimum product *less than* 0 of any subarray of A *that begins with* $A[i]$.

If we define $c \cdot \infty = \infty$ and $c' \cdot \infty = -\infty$ for all $c > 0$ and $c' < 0$, $\mathit{MaxProd}^+$ and $\mathit{MaxProd}^-$ satisfy the following mutual recurrences:

$$\mathit{MaxProd}^+(i) = \begin{cases} 1 & \text{if } i > n \\ \max\left\{1, A[i] \cdot \mathit{MaxProd}^+(i+1)\right\} & \text{if } i \le n \text{ and } A[i] \ge 0 \\ \max\left\{1, A[i] \cdot \mathit{MaxProd}^-(i+1)\right\} & \text{otherwise} \end{cases}$$

$$\mathit{MaxProd}^-(i) = \begin{cases} \infty & \text{if } i > n \\ A[i] \cdot \mathit{MaxProd}^-(i+1) & \text{if } i \le n \text{ and } A[i] \ge 0 \\ A[i] \cdot \mathit{MaxProd}^+(i+1) & \text{otherwise} \end{cases}$$

We need to compute $\max\limits_{1 \le i \le n} \mathit{MaxProd}^+(i)$. We can memoize this function in two one-dimensional arrays $\mathit{MaxProd}^+[1\,..\,n+1]$ and $\mathit{MaxProd}^-[1\,..\,n+1]$, where $\mathit{MaxProd}^\pm[i]$ is assumed to memoize $\mathit{MaxProd}^\pm(i)$. Each entry $\mathit{MaxProd}^\pm[i]$ depends only on entries in the next element of either the same array or the other array, so we can fill both arrays *in parallel* scanning right to left. Since the maximization $\max\limits_{1 \le i \le n} \mathit{MaxProd}^+(i)$ takes time $O(n)$ to compute after computing $\mathit{MaxProd}^+[1\,..\,n+1]$, the algorithm runs in ***time $O(n)$*** overall. The pseudocode for this algorithm is given below:

```
MaxProd($A[1\,..\,n]$):
    $\mathit{MaxProd}^+[n+1] \gets 1$        ⟨⟨Base case⟩⟩
    $\mathit{MaxProd}^-[n+1] \gets \infty$        ⟨⟨Base case⟩⟩
    $\mathit{max} \gets \mathit{MaxProd}^+[n+1]$
    for $i \gets n$ down to 1:
        if $A[i] \ge 0$:
            $\mathit{MaxProd}^+[i] \gets \max\{1, A[i] \cdot \mathit{MaxProd}^+[i+1]\}$
            $\mathit{MaxProd}^-[i] \gets A[i] \cdot \mathit{MaxProd}^-[i+1]$
        else:        ⟨⟨$A[i] < 0$⟩⟩
            $\mathit{MaxProd}^+[i] \gets \max\{1, A[i] \cdot \mathit{MaxProd}^-[i+1]\}$
            $\mathit{MaxProd}^-[i] \gets A[i] \cdot \mathit{MaxProd}^+[i+1]$
        $\mathit{max} \gets \max\{\mathit{max}, \mathit{MaxProd}^+[i]\}$
    return $\mathit{max}$
```

■