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.

$j \rightarrow$

$i \downarrow$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

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:

$j \rightarrow$

$i \downarrow$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

So we can construct the recurrence as follows:

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

$LSq(i,j)$ 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 ← 1 to n
      LSq[n, i] ← A[n, i]
      LSq[i, n] ← A[i, n]
  for i ← n − 1 down to 1
      for j ← n − 1 down to 1
          if A[i, j] ≠ 0
              LSq[i, j] ← min{LSq[i + 1, j], LSq[i, j + 1], LSq[i + 1, j + 1]}
          else
              LSq[i, j] ← 0

      return max(LSq)
```

■

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

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

We need to compute $\max_{1 \le i \le n} MaxSum(i)$. We can memoize this function in an array $MaxSum[1..n]$, where $MaxSum[i]$ is assumed to memoize $MaxSum(i)$. 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 $MaxSum(i)$ for all $1 \le i \le n$. Since the maximization $\max_{1 \le i \le n} MaxSum(i)$ 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] ← 0    ⟨⟨Base case⟩⟩
  max ← MaxSum[n+1]
  for i ← n down to 1:
      MaxSum[i] ← max{0, A[i] + MaxSum[i+1]}
      max ← 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 $MaxProd^+(i)$ denote the maximum product of any subarray of A *that begins with* $A[i]$.
- Let $MaxProd^-(i)$ 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 $c' < 0$, $MaxProd^+$ and $MaxProd^-$ satisfy the following mutual recurrences:

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

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

We need to compute $\max_{1 \le i \le n} MaxProd^+(i)$. We can memoize this function in two one-dimensional arrays $MaxProd^+[1..n+1]$ and $MaxProd^-[1..n+1]$, where $MaxProd^\pm[i]$ is assumed to memoize $MaxProd^\pm(i)$. Each entry $MaxProd^\pm[i]$ 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 $\max_{1 \le i \le n} MaxProd^+(i)$ takes time $O(n)$ to compute after computing $MaxProd^+[1..n+1]$, the algorithm runs in ***time $O(n)$*** overall. The pseudocode for this algorithm is given below:

```
MaxProd(A[1..n]):
  MaxProd⁺[n+1] ← 1          ⟨⟨Base case⟩⟩
  MaxProd⁻[n+1] ← ∞          ⟨⟨Base case⟩⟩
  max ← MaxProd⁺[n+1]
  for i ← n down to 1:
      if A[i] ≥ 0:
          MaxProd⁺[i] ← max{1, A[i] · MaxProd⁺[i+1]}
          MaxProd⁻[i] ← A[i] · MaxProd⁻[i+1]
      else:          ⟨⟨A[i] < 0⟩⟩
          MaxProd⁺[i] ← max{1, A[i] · MaxProd⁻[i+1]}
          MaxProd⁻[i] ← A[i] · MaxProd⁺[i+1]
      max ← max{max, MaxProd⁺[i]}
  return max
```

■

(c) Suppose you are also given an arbitrary integer $X \geq 0$. Describe and analyze an algorithm to compute the *maximum sum* of any subarray of A *less than or equal to* X.

Solution: Let $S(i)$ be the sum of $A[1..i]$ for all $0 \leq i \leq n$. $S(i)$ can be computed and memoized into an array $S[1..n]$ in time $O(n)$ via the following algorithm:

```
COMPUTES(A[1..n]):
  sum ← 0
  S[0] ← sum
  for i ← 1 to n:
      sum ← sum + A[i]
      S[i] ← sum
  return S
```

Let $S(i,j)$ be the sum of array $A[i..j]$ for all integers i and j. Then

$$S(i,j) = \begin{cases} S(j) - S(i-1) & \text{if } 1 \leq i \leq j \leq n \\ 0 & \text{otherwise} \end{cases}$$

for all such i and j. We can compute and memoize $S(i,j)$ for all such i and j in time $O(n^2)$ using $S[1..n]$ memoized in the previous step. We then define $\overline{S}(i,j)$ as follows for all integers i and j:

$$\overline{S}(i,j) = \begin{cases} S(i,j) & \text{if } 1 \leq i \leq j \leq n \text{ and } S(i,j) \leq X \\ 0 & \text{otherwise} \end{cases}.$$

$\overline{S}(i,j)$ takes time $O(n^2)$ to compute and memoize for all such i and j using $S(i,j)$ memoized in the previous step. If we define $\max \varnothing = 0$, the final answer is given by

$$\max_{i,j} \overline{S}(i,j).$$

Since this maximization takes time $O(n^2)$ to compute after computing and memoizing $\overline{S}(i,j)$ for all such i and j, the overall algorithm runs in ***time $O(n^2)$***. It is given below in pseudocode:

```
MBS(A[1..n], X):
  A[0] ← 0
  maxSum ← 0          ⟨⟨Initialize max sum seen thus far⟩⟩
  outerSum ← 0        ⟨⟨Initialize sum of subarray A[1..j]⟩⟩
  for j ← 1 to n:
      outerSum ← outerSum + A[j]                ⟨⟨Compute sum of A[1..j]⟩⟩
      innerSum ← 0    ⟨⟨Initialize sum of subarray A[1..i]⟩⟩
      for i ← 1 to j:
          innerSum ← innerSum + A[i-1]          ⟨⟨Compute sum of A[1..i-1]⟩⟩
          subarraySum ← outerSum - innerSum     ⟨⟨Compute sum of A[i..j]⟩⟩
          if subarraySum ≤ X:                   ⟨⟨If sum of A[i..j] ≤ X⟩⟩
              maxSum ← max{maxSum, subarraySum}
                                    ⟨⟨Update max sum seen thus far⟩⟩
  return maxSum
```

■

Solution: Let $S(i)$ be defined as above for all $0 \le i \le n$. We optimize the above algorithm by using the following observations. Fix an integer j, define $S_j(i) = \overline{S}(i,j)$ for all integers i and fix $i^* \in \arg\max S_j(i)$. Observe that

$$S(i^*) = \min_{0 \le i \le j} \{S(i) \mid S(i) \overset{i}{\ge} S(j) - X\}.$$

Thus, if we define

$$\underline{S}(j) = \min_{0 \le i \le j} \{S(i) \mid S(i) \ge S(j) - X\}$$

for all $1 \le j \le n$, the maximum sum of a subarray of A at most X *that ends with* $A[j]$ is

$$\overline{S}(j) = S(j) - \underline{S}(j).$$

Thus, the maximum sum of a subarray of A at most X is given by

$$\max_{1 \le j \le n} \overline{S}(j).$$

It follows that if we use a data structure that enables the insertion of values into the data structure as well as the computation of a minimum value in the data structure above a given threshold in time $O(\log n)$, we can achieve time $O(n \log n)$ overall by simply computing $\underline{S}(i)$ (and also $\overline{S}(i)$) in time $O(\log n)$ for each $1 \le i \le n$. A binary search tree does the trick. We make the following assumptions on the operations of the binary search tree:

- BinarySearch(k) returns the smallest key *at least* k in any vertex of the binary search tree in time $O(\log n)$ and
- Insert(k) inserts a vertex into the tree with key k in time $O(\log n)$.

With these binary search tree operations, we can compute the maximum sum of a subarray of A at most X via the following algorithm:

```
MBS'(A[1..n], X):
  maxSum ← 0        ⟨⟨Initialize max sum seen thus far⟩⟩
  cumSum ← 0        ⟨⟨Initialize cumulative sum S(j) to 0⟩⟩
  Insert(cumSum)    ⟨⟨Insert vertex with key S(j) into BST⟩⟩
  for j ← 1 to n:
      cumSum ← cumSum + A[j]                    ⟨⟨Compute S(j)⟩⟩
      minAtLeast ← BinarySearch(cumSum − X)     ⟨⟨Compute S_(j)⟩⟩
      currSum ← cumSum − minAtLeast             ⟨⟨Compute S̄(j)⟩⟩
      maxSum ← max{maxSum, currSum}             ⟨⟨Update max sum if it's bigger⟩⟩
      Insert(cumSum)        ⟨⟨Insert vertex with key S(j) into BST⟩⟩
  return maxSum
```

Since each iteration of the for loop in this algorithm requires time $O(\log n)$, this algorithm runs in ***time $O(n \log n)$***. ■

(d) Describe a faster algorithm for part (c) when every element in the array A is nonnegative.

Solution: We can obtain a faster algorithm than in part (c) by observing the following fact in the case that A contains only nonnegative elements.

Claim 1. *Fix integers j and j' such that $1 \le j < j'$. Define $S_j(i) = \overline{S}(i,j)$ and $S_{j'}(i') = \overline{S}(i',j')$ for all integers i and i'. Fix $i^* \in \arg\max_{i \le j} S_j(i)$. There exists $i^* \le i'^* \le n$ such that $i'^* \in \arg\max_{i'} S_{j'}(i')$.*

Proof: Suppose otherwise. Then there exists an index $i'^* < i^*$ such that

$$\max_{i^* \le i'} S_{j'}(i') < \max_{i' \le j} S_{j'}(i') = S_{j'}(i'^*).$$

Then

$$\begin{aligned} S_j(i^*) = \overline{S}(i^*, j) &= \max_{i^* \le i' \le j'} \overline{S}(i', j) = \max_{i^* \le i' \le j'} \left[\overline{S}(i', j') - S(j+1, j')\right] \\ &= \left[\max_{i^* \le i' \le j'} \overline{S}(i', j')\right] - S(j+1, j') < \left[\max_{i' \le j'} \overline{S}(i', j')\right] - S(j+1, j') \\ &= \left[\max_{i' \le j'} S_{j'}(i')\right] - S(j+1, j') = S_{j'}(i'^*) - S(j+1, j') \\ &= \overline{S}(i'^*, j') - S(j+1, j') = \overline{S}(i'^*, j) = S_j(i'^*), \end{aligned}$$

which contradicts the assumption that $i^* \in \arg\max_{i \le j} S_j(i)$. □

The above claim implies that after computing an optimal i corresponding to a particular j with respect to the maximization of $\overline{S}(i,j)$, it is unnecessary to consider $i' < i$ as candidates for indices that maximize $\overline{S}(i',j')$ for all $j' > j$ and can thus be skipped over. This optimization corresponds to the following algorithm, which effectively maintains a sliding window $A[i..j]$ of elements whose sum is at most X:

```
MBS''(A[1..n], X):
  i ← 0                    ⟨⟨Initialize window left boundary⟩⟩
  A[i] ← 0                 ⟨⟨Add a sentinel value⟩⟩
  currSum ← 0              ⟨⟨Initialize window sum⟩⟩
  maxSum ← 0               ⟨⟨Initialize max sum seen thus far⟩⟩
  for j ← 1 to n:                    ⟨⟨Increment right boundary of window⟩⟩
      currSum ← currSum + A[j]       ⟨⟨Update window sum⟩⟩
      while currSum > X:             ⟨⟨While window sum is more than X⟩⟩
          currSum ← currSum − A[i]
          i ← i + 1 ⟨⟨Increment left boundary and update window sum until not⟩⟩
      maxSum ← max{maxSum, currSum} ⟨⟨Update max sum seen thus far⟩⟩
  return maxSum
```

Because i and j each only index through $O(n)$ values *over all iterations* of both loops in the above algorithm, this algorithm runs in ***time $O(n)$***.

Solution: A recursive implementation of the above algorithm is given below. Add sentinel values $A[0] = 0$ and $A[n+1] = \infty$. Let $\text{MBS}'''(A[0..n+1], X, i, j, currSum)$ denote the maximum sum at most X of a subarray of $A[i..n]$ where the sum of $A[i..j]$ is $currSum$. It is given below in pseudocode:

```
MBS'''(A[0..n+1], X, i, j, X, currSum):
  if i > n:                    ⟨⟨Subarray is empty; report that max sum is 0⟩⟩
      return 0
  else if currSum > X:         ⟨⟨If curr window sum is more than X⟩⟩
      return MBS'''(A, i+1, j, X, currSum − A[i])
                               ⟨⟨Report max bdd sum of window A[i'..j']
                                   for i' ≥ i and j' > j⟩⟩
  else:                        ⟨⟨If curr window sum is not more than X⟩⟩
      return max{currSum, MBS'''(A, i, j+1, X, currSum + A[j+1])}
                               ⟨⟨Report bigger of curr window sum and max bdd sum
                                   of window A[i'..j'] for i' ≥ i and j' > j⟩⟩
```

The final answer is given by $\text{MBS}'''(A, X, 0, 0, 0)$. Its runtime for an array of n elements satisfies the recurrence $T(n) = T(n-1) + O(1)$ which has solution $T(n) = O(n)$, so this algorithm also runs in ***time $O(n)$***. ■

3. We are given a tree $T = (V, E)$ with n vertices. Assume that the degree of all the vertices in T is at most 3. You are given a function $f : V \to \{0, 1, 2, 3\}$. The task is to compute a subset X of edges, such that for every node $v \in V$, there are at least $f(v)$ distinct edges in X that are adjacent to v.

   Describe an algorithm, as fast as possible, that computes the minimum size set $X \subseteq E$ that meets the needs of all the nodes in the tree. The algorithm should output both $|X|$ and X itself.

Solution: We root the tree at an arbitrary vertex r that is either of degree two or one (this takes $O(n)$ time to do) – since every tree has a leaf, this is always possible. Now, every vertex has pointers left(v), right(v), which are NULL if these children nodes do not exist.

We use dynamic programming to solve this problem, where the recursive function is defined as follows.

We use dynamic programming to solve this problem, where the recursive function is defined as follows.

$$Cover(v, s) = \begin{cases} 0 & v = \text{NULL} \\ \min\{Cover(\text{left}(v), \ell) + Cover(\text{right}(v), r) + s\} & \text{if } \Delta \geq f(v) \text{ for all } \ell, r \in \{0, 1\} \\ \infty & \text{otherwise} \end{cases}$$

$$\text{where, } \Delta = \text{s} + (\ell \times [\text{left}(v) \neq \text{NULL}]) + (\text{r} \times [\text{right}(v) \neq \text{NULL}])$$

A call to $cover(v, s)$ returns the cheapest feasible solution that meets all the demands in the subtree of v, where $s \in \{0, 1\}$ indicates whether or not the edge to the parent is included in the solution (and needed to be paid for). We compute the minimum size out of 4 combinations specified by the values of $\{\ell, r\}$ indicating wether or not edges to the left and right children are included respectively, if they exist.

```
COVER(v, s):
    b ← +∞, c ← ⟨0, 0⟩
    for ℓ = 0, 1 do
        for r = 0, 1 do
            Δ ← s + [left(v) ≠ NULL] · ℓ + [right(v) ≠ NULL] · r
            if Δ ≥ f(v) then
                    p_left ← 0, p_right ← 0
                    if left(v) ≠ NULL then
                            p_left ← cover(left(v), ℓ)
                    if right(v) ≠ NULL then
                            p_right ← cover(right(v), r)
                    α ← p_left + p_right + s
                    if α < b then
                            b ← α
                            c ← ⟨ℓ, r⟩

    sol[v, s] ← c      // For printing the solution later
    return b
```

We memoize this function into an array $cover[1..n, 0..1]$. There are $2n$ distinct recursive calls, so memoization readily yields an ***O(n) time*** algorithm using a postorder, reverse preorder or reverse level order traversal of the tree, since the recursive function takes $O(1)$ time ignoring the recursive calls. (Note that a level order traversal of a tree is equivalent to a breadth-first search (BFS) of the tree.)

Printing the solution is easy. We first call $cover(r, 0)$. Next, we call $printSolution(\text{NULL}, r, 0)$, where *printSolution* is defined as follows:

```
// p:  Parent of current node v
PRINTSOLUTION(p, v, s):
    if v = NULL then
        return
    if s = 1 then
        print "Edge pv in optimal solution"
    ⟨ℓ, r⟩ ← sol[v, s]
    if left(v) ≠ NULL then
        printSolution(v, left(v), ℓ)
    if right(v) ≠ NULL then
        printSolution(v, right(v), r)
```

■

4. 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), ...(x_n, y_n)], d):
  Let g ← Empty Graph with n nodes
  for i ← 1 to n − 1
      for j ← 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), ...(x_n, y_n)], d, src, dest):
  graph = BuildGraph(A, d)
  PathExists = BFSPath(graph, src, dest)
  return PathExists
```

■