1. An array $A[0..n-1]$ of n distinct numbers is *bitonic* if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

Solution: Let BitonicMin($A[0..n-1]$) denote the index of the smallest element in a bitonic array $A[0..n-1]$. The pseudocode for an algorithm corresponding to this English description is given below:

```
BitonicMin($A[0..n-1]$):
    if $n < 10^{10}$:
        use brute force
    $m \leftarrow \lfloor n/2 \rfloor$
    if { $A[m] < A[m+1] < A[n-1] < A[0]$ or
         $A[m] > A[m+1] > A[n-1] > A[0]$ or
         $A[0] < A[n-1]$ and $A[m] < A[m+1]$ }
        return BitonicMin($A[0..m]$)
    else:
        return BitonicMin($A[m+1..n-1]$)
```

The correctness of this algorithm follows from the following claim.

Claim 1. *Fix an arbitrary bitonic array $A[0..n-1]$. BitonicMin(A) computes the index of the minimum element in A.*

Proof: We proceed by induction on n. Fix an integer $n \geq 0$ and a bitonic array $A[0..n-1]$. If $n < 10^{10}$, BitonicMin(A) computes the index of the minimum element in A because "use brute force" does. Suppose $n \geq 10^{10}$, $m = \lfloor n/2 \rfloor$ and BitonicMin($A[i..j]$) computes the index of the minimum element in any bitonic subarray $A[i..j]$ of A for any i and j such that $j - i < n - 1$. In the following analysis, we denote concatenation (*not product!*) by $\bullet$. We have the following cases:

- $A[m] < A[m+1] < A[n-1] < A[0]$. By the definition of bitonic, there must be *unique* indices $0 \leq i \leq j \leq m$ such that $A[j..n-1] \bullet A[0..i]$ is increasing and $A[i..j]$ is decreasing. Observe that such an index j is the index the minimum element in this case.
- $A[m] > A[m+1] > A[n-1] > A[0]$. By the definition of bitonic, there must be *unique* indices $0 \leq i \leq j \leq m$ such that $A[i..j]$ is increasing and $A[j..n] \bullet A[0..i]$

is decreasing. Observe that such an index i is the index the minimum element in this case.

- $A[m] < A[m+1]$ and $A[0] < A[n-1]$. By the definition of bitonic, there must be *unique* indices $0 \le i \le m$ and $m+1 \le j \le n-1$ such that $A[i \mathinner{..} j]$ is increasing and $A[j \mathinner{..} n] \bullet A[1 \mathinner{..} i]$ is decreasing. Observe that such an index i is the index the minimum element in this case.

The above analysis gives that the above conditions are sufficient for the minimum element in A to be in $A[0 \mathinner{..} m]$. By symmetry, we immediately have that these conditions are necessary and sufficient for the minimum element in A to be in $A[0 \mathinner{..} m]$. Since

- $A[0 \mathinner{..} m]$ and $A[m+1 \mathinner{..} n-1]$ are bitonic,
- $m-1 < n-1$ and $n-1-(m+1) = n-m < n-1$ (i.e., the recursive calls receive strictly smaller inputs) and
- $0 \le m$ and $m+1 \le n-1$ (i.e., the recursive calls receive only valid inputs),

BitonicMin($A[0 \mathinner{..} m]$) and BitonicMin($A[m+1 \mathinner{..} n-1]$) compute the index of the minimum element in $A[0 \mathinner{..} m]$ and $A[m+1 \mathinner{..} n-1]$, respectively. This completes the proof. $\square$

Computing BitonicMin(A) gives the index of the minimum element in A. Since this algorithm *is* binary search, it requires *time* $O(\log n)$.

2. Suppose we have a river and on either side are a number of cities numbered from 1 to n (North side: $N[1 \ldots n]$, South side: $S[1 \ldots n]$). The city planner wants to connect certain cities together using bridges and has a list of the desired crossings (x is a $2 \times k$ array where k is the number of planned bridges) . Unfortunately, as we know, bridges cannot cross one-another over water so the city planner must focus on building the most bridges from his plan that do not intersect. Describe an algorithm that finds the maximum number of non-intersecting bridges.

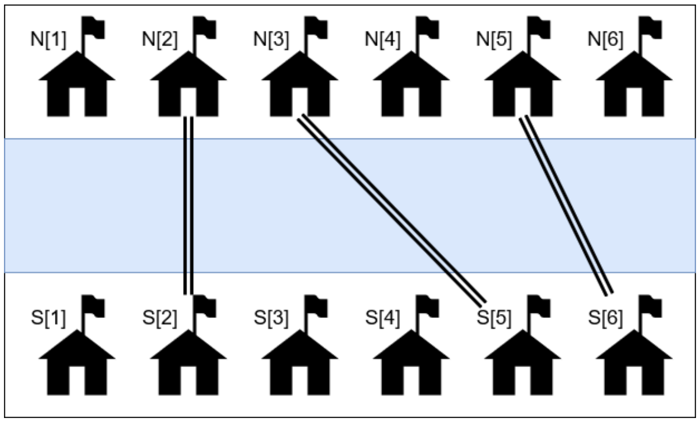

Figure 1. Assuming $n = 6$, $x = \begin{bmatrix} 1 & 5 & 6 & 2 & 3 \\ 4 & 6 & 1 & 2 & 5 \end{bmatrix}$, then the output should be 3 as shown above.

Solution: We can solve this problem by applying the LIS algorithm. Since the cities on the north and south banks are in ascending order of index $N[1...n]$ and we can not overlap bridges, we can sort either the South cites or the North cities in the given 2D bridges array first. Then we can apply the LIS algorithm on the unsorted part of the 2D array. Based on the results from the LIS algorithm, we can connect the bridges that are present in those combinations.

Given $x = \begin{bmatrix} 1 & 5 & 6 & 2 & 3 \\ 4 & 6 & 1 & 2 & 5 \end{bmatrix}$

Sorting based on North Cities, we get $x = \begin{bmatrix} 1 & 2 & 3 & 5 & 6 \\ 4 & 2 & 5 & 6 & 1 \end{bmatrix}$

After LIS is applied on the South Cities, we get $\begin{bmatrix} 2 & 5 & 6 \end{bmatrix}$ and the count is 3.

Pseodo Code:

```
BUILDBRIDGE($x, n$):
    sortedMatrix = SORT($x, 1$)  ⟨⟨Sorting the matrix $x$ based on the first row⟩⟩
    $A = sortedMatrix[2]$        ⟨⟨We are picking the bottom row of the matrix⟩⟩
    bridgeCount = LIS($A$)
    return bridgeCount
```

TLDR: Sort then LIS. ■

3. In lecture we defined the recurrence of the longest-increasing-subsequence(LIS) problem as:

$$LIS_{LEC}(i,j) = \begin{cases} 0 & i = 0 \\ LIS_{LEC}(i-1,j) & A[i] \geq A[j] \\ \max \begin{cases} LIS_{LEC}(i-1,j) \\ 1 + LIS_{LEC}(i-1,i) \end{cases} & A[i] < A[j] \end{cases} \tag{1}$$

But when we worked out the problem in lab looks like:

$$LIS_{LAB}(i,j) = \begin{cases} 0 & \text{if } i > n \\ LIS_{LAB}(i+1,j) & \text{if } i \leq n \text{ and } A[j] \geq A[i] \\ \max \begin{Bmatrix} LIS_{LAB}(i+1,j) \\ 1 + LIS_{LAB}(i+1,i) \end{Bmatrix} & \text{otherwise} \end{cases} \tag{2}$$

Is one of them wrong? If not, what's the difference? You solution so be a simple, **short**, english description of each recurrence. No long proofs for correctness are necessary. This is to make sure you understand how to describe a function (and no saying "LIS returns the longest increasing subsequence length." is not a sufficient description).

Solution: Yes they are both correct, they are simply computing the results from different directions:

- $LIS_{LEC}(i,j)$ returns the largest possible increasing subsequence in the **prefix** array $A[1\ldots i]$ assuming none of the values in that subsequence are larger than $A[j]$. That is why we state from the smallest possible prefix array $i = 1$ and work our way up.
- $LIS_{LAB}(i,j)$ returns the largest possible increasing subsequence in the **suffix** array $A[i\ldots n]$ assuming none of the values in that subsequence are smaller than $A[j]$. That is why the start from the largest possible

■

4. A *subsequence* is any sequence obtained from a sequence by taking some of its elements while keeping them in the same order. For instance, the strings 7, $F4$ and $WTF374$ are all subsequences of the string $WTF374$. A *supersequence* is any sequence obtained from a sequence by adding more elements, keeping the elements of the original sequence in the same order. For example, $WTF374$, $SMTWTFS247374$ and $WTF374473FTW$ are all supersequences of the string $WTF374$. For each part, we want efficient (i.e., polynomial time) algorithms.

   (a) Suppose $X[1..m]$ and $Y[1..n]$ are two arrays. A *common subsequence* of X and Y is another sequence that is a subsequence of both X and Y. Describe an algorithm to compute the *length* of a longest common subsequence of two given arbitrary arrays A and B.

Solution: Let $LCS(i,j)$ denote the length of a longest common subsequence of $A[i..m]$ and $B[j..n]$. LCS obeys the following recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i > m \text{ or } j > n \\ \max\left\{\begin{array}{r} LCS(i+1,j) \\ LCS(i,j+1) \\ 1+LCS(i+1,j+1) \end{array}\right\} & \text{if } i \le m, j \le n \text{ and } A[i]=B[j] \\ \max\left\{\begin{array}{r} LCS(i+1,j) \\ LCS(i,j+1) \end{array}\right\} & \text{otherwise} \end{cases}$$

We need to compute $LCS(1,1)$. We can memoize the function LCS into an array $LCS[1..m+1, 1..n+1]$. Each entry $LCS[i,j]$ depends only on entries in the next row $LCS[i+1,\cdot]$ or the next column $LCS[\cdot, j+1]$, so we can fill the array in reverse row-major order, scanning bottom to top in the outer loop, and right to left in the inner loop.

```
LCS($A[1..m]$,$B[1..n]$):
    for $i$ from 1 to $m+1$:   ⟨⟨Base cases⟩⟩
        $LCS[i,n+1] \leftarrow 0$
    for $j$ from 1 to $n+1$:   ⟨⟨Base cases⟩⟩
        $LCS[m+1,j] \leftarrow 0$
    for $i \leftarrow m$ down to 1:
        for $j \leftarrow n$ down to 1:
            $LCS[i,j] \leftarrow \max\{LCS[i,j+1], LCS[i+1,j]\}$
            if $A[i]=B[j]$:
                $LCS[i,j] \leftarrow \max\{LCS[i,j], 1+LCS[i+1,j+1]\}$
    return $LCS[1,1]$
```

The resulting algorithm runs in ***$O(mn)$ time***, where m and n are the number of elements in A and B, respectively. ■

(b) Suppose $X[1..m]$ and $Y[1..n]$ are two arrays. A *common supersequence* of X and Y is a sequence that contains both X and Y as subsequences. Describe an algorithm to compute the *length* of a shortest common supersequence of two given arbitrary arrays A and B.

Solution: Let $SCS(i,j)$ denote the length of a shortest common supersequence of $A[i..m]$ and $B[j..n]$. SCS obeys the following recurrence:

$$SCS(i,j) = \begin{cases} n-j+1 & \text{if } i > m \\ m-i+1 & \text{if } i \le m \text{ and } j > n \\ \min\begin{Bmatrix} 1+SCS(i+1,j) \\ 1+SCS(i,j+1) \\ 1+SCS(i+1,j+1) \end{Bmatrix} & \text{if } i \le m, j \le n \text{ and } A[i]=B[j] \\ \min\begin{Bmatrix} 1+SCS(i+1,j) \\ 1+SCS(i,j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

We need to compute $SCS(1,1)$. We can memoize the function SCS into a two-dimensional array $SCS[1..m+1, 1..n+1]$. Each entry $SCS[i,j]$ depends only on entries in the next row $SCS[i+1,\cdot]$ or the next column $SCS[\cdot, j+1]$, so we can fill the array in reverse row-major order, scanning bottom to top in the outer loop, and right to left in the inner loop.

```
SCS(A[1..m],B[1..n]):
    for $i \leftarrow 1$ to $m+1$:        ⟨⟨Base cases⟩⟩
        $SCS[i,n+1] \leftarrow m+1-i$
    for $j \leftarrow 1$ to $n+1$:        ⟨⟨Base cases⟩⟩
        $SCS[m+1,j] \leftarrow n+1-j$
    for $i \leftarrow m$ to 1:
        for $j \leftarrow n$ to 1:
            $SCS[i,j] \leftarrow \min\{1+SCS[i,j+1], 1+SCS[i+1,j]\}$
            if $A[i]=B[j]$:
                $SCS[i,j] \leftarrow \min\{SCS[i,j], 1+SCS[i+1,j+1]\}$
    return $SCS[1,1]$
```

The resulting algorithm runs in ***O(mn) time***, where m and n are the number of elements in A and B, respectively. ■

Solution: The length of the shortest common supersequence can also be found via the following algorithm, where LCS is the algorithm defined in part (a):

```
SCS($A[1..m]$,$B[1..n]$):
    return $m+n-$LCS($A,B$)
```

The correctness of this algorithm follows from the following claim:

Claim 2. *Fix arbitrary arrays $A[1..m]$ and $B[1..n]$ and define L and S to be a length of the longest common subsequence and a shortest common supersequence, respectively, of A and B. Then $L+S=m+n$.*

Proof: Suppose LCS is a longest common subsequence of A and B. Adding the $m+n-2L$ elements of A and B not used in LCS to LCS while preserving their order relative to the elements in LCS *necessarily* gives a supersequence of A and B of length $m+n-L$, giving that $S \leq m+n-L$. On the other hand, we have that $S \geq m+n-L$, because suppose otherwise. Then there is a supersequence of A and B of length $m+n-L' < m+n-L$ for some integer $L' \geq 0$. For this to happen, there *must* be a subsequence of length L' contained in A and B. In other words, there must be a common subsequence of length L' in A and B, which is a contradiction as L by definition is the length of a longest common subsequence of A and B and $L < L'$. The two inequalities above together prove the claim. □

Because LCS(A,B) in the above algorithm requires time $O(mn)$, the above algorithm runs in ***O(mn) time***, where m and n are the number of elements in A and B, respectively.

(c) A sequence $W[1..n]$ of numbers is *weakly increasing* if each element is larger than the average of its two previous elements (i.e., $2 \cdot W[i] > W[i-1] + W[i-2]$ for all $i > 2$). Describe an algorithm to compute the *length* of a longest weakly increasing subsequence of a given arbitrary array A of integers.

Solution: We use the approach used in the solution for the longest convex subsequence problem in the lab on 3/4 to solve this problem. Let $LWIS(i, j)$ denote the length *minus* 1 of a longest weakly increasing subsequence of a *nonempty* array $A[i..n]$ whose first element is $A[i]$ and whose second element (if any) is $A[j]$. $LWIS$ obeys the following recurrence:

$$LWIS(i,j) = 1 + \max\{LWIS(j,k) \mid j < k \leq n \text{ and } 2 \cdot A[k] > A[j] + A[i]\}$$

Here we define $\max\varnothing = 0$; this gives us a working base case. The final answer is given by

$$\text{LWIS}(A[1..n]) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + \max\limits_{1 \leq i < j \leq n} LWIS(i,j) & \text{otherwise} \end{cases}.$$

We can memoize the function $LWIS$ into a two-dimensional array $LWIS[1..n-1, 2..n]$, which we can fill in reverse row-major order in ***$O(n^3)$ time*** as follows:

```
LWIS($A[1..n]$):
    if $n = 0$:        ⟨⟨Base case of LWIS($A[1..n]$)⟩⟩
        return 0
    $\ell \leftarrow 1$
    for $i \leftarrow n-1$ down to 1:
        for $j \leftarrow n$ down to $i+1$:
            $LWIS[i,j] \leftarrow 0$        ⟨⟨Base case of LWIS($i,j$)⟩⟩
            for $k \leftarrow j+1$ to $n$:
                if $2A[k] > A[j] + A[i]$:
                    $LWIS[i,j] \leftarrow \max\{LWIS[i,j], 1 + LWIS[j,k]\}$
            $\ell \leftarrow \max\{\ell, LWIS[i,j] + 1\}$
    return $\ell$
```

■

(d) A sequence $O[1..n]$ of numbers is *oscillating* if $O[i] > O[i+1]$ for all odd i and $O[i] < O[i+1]$ for all even i. Describe an algorithm to compute the *length* of a shortest oscillating supersequence of a given arbitrary array A of integers.

Solution: The idea of the mutual recurrences below is to walk along the array from left to right. Either the recurrence adds only the current element of the array to the oscillating supersequence or in addition adds an additional element with value ∞ and $-\infty$ *before the current element*. It then maximizes the lengths over all oscillating supersequences it considers. Add a sentinel value $A[0] = -\infty$. We define two functions:

- Let $SOS^+(j)$ denote the length of a shortest oscillating supersequence of $A[j\,..\,n]$ whose first element (if any) is larger than $A[j-1]$ and whose second element (if any) is smaller than its first.
- Let $SOS^-(j)$ denote the length of a shortest oscillating supersequence of $A[j\,..\,n]$ whose first element (if any) is smaller than $A[j-1]$ and whose second element (if any) is larger than its first.

SOS^+ and SOS^- satisfy the following mutual recurrences:

$$SOS^+(j) = \begin{cases} 0 & \text{if } j > n \\ \min\left\{\begin{matrix} 1 + SOS^-(j+1) \\ 2 + SOS^+(j+1) \end{matrix}\right\} & \text{if } j \le n \text{ and } A[j] > A[j-1] \\ 2 + SOS^+(j+1) & \text{otherwise} \end{cases}$$

$$SOS^-(j) = \begin{cases} 0 & \text{if } j > n \\ \min\left\{\begin{matrix} 1 + SOS^+(j+1), \\ 2 + SOS^-(j+1) \end{matrix}\right\} & \text{if } j \le n \text{ and } A[j] < A[j-1] \\ 2 + SOS^-(j+1) & \text{otherwise} \end{cases}$$

We need to compute $SOS^+(1)$. We can memoize these functions into one-dimensional arrays $SOS^+[1\,..\,n+1]$ and $SOS^-[1\,..\,n+1]$. Each entry $SOS^\pm[i,j]$ depends only on the next entry of either the same array or the other array. So we can fill both arrays in parallel, scanning right to left.

```
SOS(A[1..n]):
  $A[0] \leftarrow -\infty$                  ⟨⟨Add a sentinel⟩⟩
  $SOS^+[n+1] \leftarrow 0$              ⟨⟨Base case⟩⟩
  $SOS^-[n+1] \leftarrow 0$              ⟨⟨Base case⟩⟩
  for $j \leftarrow n$ down to 1:
      $SOS^+[j] \leftarrow 2 + SOS^+[j+1]$
      $SOS^-[j] \leftarrow 2 + SOS^-[j+1]$
      if $A[j] > A[j-1]$:
          $SOS^+[j] \leftarrow \min\{SOS^+[j], 1 + SOS^-[j+1]\}$
      if $A[j] < A[j-1]$:
          $SOS^-[j] \leftarrow \min\{SOS^-[j], 1 + SOS^+[j+1]\}$
  return $SOS^+[1]$
```

The resulting algorithm runs in ***O(n) time***. ■