1. Suppose we have a river and on either side are a number of cities numbered from 1 to n (North side: $N[1 \dots n]$, South side: $S[1 \dots n]$). 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. You may assume that a city would not appear more than once in the plan.

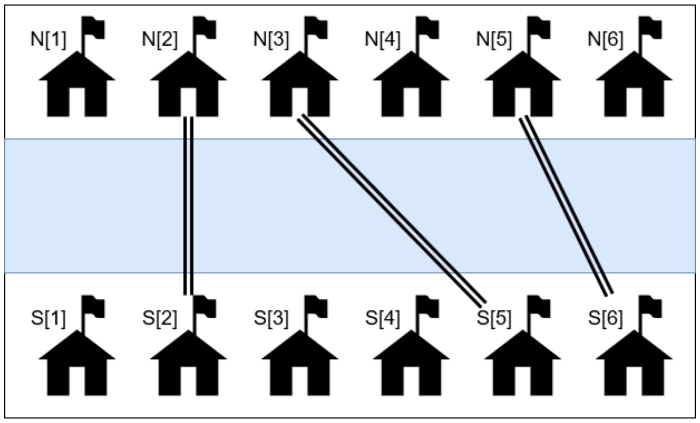


Figure 1. Assuming $n = 6$, $x = \begin{bmatrix} 1 & 5 & 6 & 2 & 3 \\ 4 & 6 & 1 & 2 & 5 \end{bmatrix}$, then the output should be 3 as shown above.

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 $N[1 \dots n]$ 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. ■

2. For each of the following recurrences, provide English description in terms of the parameters i and j.

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

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

   Your solution should 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:

(a) $LIS_{LEC}(i,j)$ 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.

(b) $LIS_{LAB}(i,j)$ 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 suffix array.

■

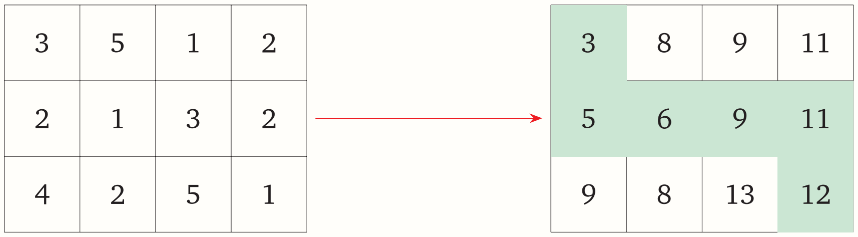
3. Given an $n \times m$ grid filled with non-negative numbers, find the minimal sum of all numbers along a path from top left $(1,1)$ to bottom right (n,m). You can only move either down $(++i)$ or right $(++j)$ at any point in time. Find an algorithmically efficient solution. What is the running time of your algorithm?

Solution: Brute Force Approach(Recursion): We will start from top left $(1,1)$ and for each element we will consider two paths, one to the right $(++j)$ and one downwards $(++i)$, and find the minimum sum of those two. This determines whether we should proceed with a rightward or downward step to minimize the sum. This algorithm will have a time complexity of $O(2^{(n+m)})$ because, we are considering two paths and calculating the sum at each element for those paths recursively.

Best Approach(Dynamic Programming): For element at (i,j), we will calculate the minimum sum to reach that element from top left $(1,1)$. The sum at (i,j) can either come from up or left, since it can only move down or right. We define a recurrence relation DP(i,j), which returns the minimum sum to reach the element at index (i,j) from top left.

$$DP(i,j) = \begin{cases} A[i][j] & \text{if } i = 1 \text{ and } j = 1 \quad (1a) \\ DP(i,j-1) + A[i][j] & \text{if } i = 1 \text{ and } j \neq 1 \quad (1b) \\ DP(i-1,j) + A[i][j] & \text{if } j = 1 \text{ and } i \neq 1 \quad (1c) \\ A[i][j] + \min \left\{ \begin{array}{l} DP(i-1,j) \\ DP(i,j-1) \end{array} \right\} & \text{Otherwise} \quad (1d) \end{cases}$$

Observe that for the first row we can move only right, and hence to reach any index in the first row the min path sum is the sum of all elements from the top left to that index (Case $3b$). Similarly, for the first column, we can move only down, and hence to reach any index in the first row the min path sum is the sum of all elements from the top left to that index (Case $3c$). For the other elements we will have two choices to reach the index (i,j) a) we reach by moving right (from index $(i,j-1)$) b) we reach by moving downwards (from index $(i-1,j)$). We will choose the minimum sum of these two options and then add the current element value to it to get the minimum sum at that element (Case $3d$).



```
MinSumPath($A[1..n][1..m]$):
  for ($i \leftarrow 1 : n$)
      for ($j \leftarrow 1 : m$)
          if $i = 1$ and $j = 1$
              continue
          else if $i = 1$ and $j =!1$        ⟨⟨First Row⟩⟩
              $A[i][j] = A[i][j-1] + A[i][j]$
          else if $j = 1$ and $i =!1$        ⟨⟨First Column⟩⟩
              $A[i][j] = A[i-1][j] + A[i][j]$
          else
              $A[i][j] = A[i][j] + min(A[i][j-1], A[i-1][j])$
  return $A[n][m]$
```

Here, we are changing the array A itself, you can initialize another array also rather than changing A. This algorithm will have a time complexity of $O(n * m)$ because of the nested for loop.(Outer loop : n ;Inner loop : m) ■

4. A certain string processing language allows the programmer to break a string into two pieces. It costs n units of time to break a string of n characters into two pieces, since this involves copying the old string. A programmer wants to break a string into many pieces, and the order in which the breaks are made can affect the total amount of time used. For example, suppose we wish to break a 20-character string after characters 3, 8, and 10. If the breaks are made in left-to-right order, then the first break costs 20 units of time, the second break costs 17 units of time, and the third break costs 12 units of time, for a total of 49 units. If the breaks are made in right-to-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, for a total of only 38 units.

   Give a dynamic programming algorithm that takes a list of character positions after which to break and determines the cheapest break cost in $O(n^3)$ time.

Solution: We define a recurrence relation Cost(i,j), which returns the minimum cost of cutting a string that starts from the index i and ends at the index j for all possibe cuts k that are within the string boundaries $i \leq k < l$. For every cut k, this cost to cut the string is essentially broken down into three parts :

1. Cost[i][k] - The total cost of further breaking the left half of the string
2. Cost[k+1][j] - The total cost of further breaking the right half of the string
3. (j-i+1) - The cost of breaking the string[i..j] after the kth character which is the length of the current string

This can be recursively called for all strings of length greater than 1. So, our base case is to return 0 when we reach i ≥ j as we cannot further cut strings that have length ≤ 1. Therefore our recurrence relation is :

$$\text{Cost(i,j)} = \begin{cases} 0 & i \geq j \\ \min_k \left\{ \text{Cost}(i,k) + \text{Cost}(k+1,j) + (j-i+1) \right. & i \leq k < j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

To find the total minimum cost of cutting a string of length N at the given list of breakpoints, we compute the values for our cost matrix starting from our base case of strings of length 1 and use our cost function to compute the cost for strings of increasing length 2 up to N. Since we compute the cost functions in the increasing order of the length, the cost of left and right parts of the broken string are of lesser length than the current string and we can make use of our cost table for the previously computed cost values for the smaller left and right pieces with their respective boundaries. In the end, Cost$[1][N]$ gives us the optimal minimum cost of cutting the entire string at the given breakpoints. Since we require a nested loop with the outer loop computing lengths from 2 to N and an inner loop for all pairs of the length, our algorithm takes $O(n^3)$.

For example, let's compute the cost matrix for a string of length = 5 and breakpoints at $1, 2, 4$ for lengths- 1(base-case) and 2.

length = 1

	1	2	3	4	5
1	0				
2	x	0			
3	x	x	0		
4	x	x	x	0	
5	x	x	x	x	0

$\longrightarrow$

length = 2

	1	2	3	4	5
1	0	2			
2	x	0	2		
3	x	x	0	0	
4	x	x	x	0	2
5	x	x	x	x	0

Now for length = 3, to compute Cost[1][3] in our example we have two possible cuts $1, 2$ as $1 \le 1, 2 < 3$.

For cut = 1, the currString[1..3] is broken into leftString[1..1] and rightString[2..3] and we need to add their costs to the current cut's cost along with a cutting cost=3(length). The cost of the leftString and RightString is given by Cost[1][1] = 0 and Cost[2][3] = 2 respectively, making the current cost as 5. Note that for computing strings of length=3 we make use of the already computed costs for strings of length=2, 1.

Similarly, the cost for cut=2 is cost[1][2] + cost[2][3] + 3 = 7. We take the min(5, 7), and store Cost[1][3] as 5. These costs are repetitively computed untill length=5 and we return Cost[1][5]=10 as the minimum cost to cut the entire string at the given breakpoints.

Algorithm:

```
CheapestBreakCost($A[1..N], BP[1..M]$)
  for ($i \leftarrow 1 : N$)
      for ($j \leftarrow 1 : N$)
          if ($i = j$) then Cost[$i$][$j$] $= 0$
  for (length $\leftarrow 2 : N$)
      for ($i \leftarrow 1 : N -$ length $+ 1$)
          $j = i +$ length $- 1$
          Cost[$i$][$j$] $= inf$
          for ($k \leftarrow 1 : M$)
              $kCut = BP[k]$
              if ($i \le kCut$ & $kCut < j$) then
                  currCost = Cost[$i$][$kCut$] + Cost[$kCut + 1$][$j$] + $(j - i + 1)$
                  Cost[$i$][$j$] = min{Cost[$i$][$j$] , currCost}
          if (Cost[$i$][$j$] $= inf$) then Cost[$i$][$j$] $= 0$
  return Cost[1][$N$]
```

■