# MORE EXERCISES: ARRAYS

You can check your solutions in Judge



## 1. ENCRYPT, SORT, AND PRINT ARRAY

Write a program that reads a **sequence of strings** from the console. Encrypt every string by summing:

- The code of **each vowel multiplied by the string length**
- The code of **each consonant divided by the string length**

**Sort** the **number** sequence in ascending order and print it in the console.

On the first line, you will always receive the number of strings you have to read.

### EXAMPLES

| Input | Output | Comments |
|---|---|---|
| 4<br>Peter<br>Maria<br>Katya<br>Todor | 1032<br>1071<br>1168<br>1532 | Peter = 1071<br>Maria = 1532<br>Katya = 1032<br>Todor = 1168 |
| 3<br>Sofia<br>London<br>Washington | 1396<br>1601<br>3202 | Sofia = 1601<br>London = 1396<br>Washington = 3202 |

```
using System;
namespace _01.EncryptSortAndPrintArray
{
   internal class Program
   {
      static void Main(string[] args)
      {
         int n = int.Parse(Console.ReadLine()); // Number of strings to read
         int[] encryptedValues = new int[n];    // Use an array instead of a List
         for (int i = 0; i < n; i++)
         {
            string input = Console.ReadLine();
            int stringLength = input.Length;
            int sum = 0;
            foreach (char ch in input)
            {
               char v = char.ToLower(ch);

               if (v == 'a' || v == 'e' || v == 'i' || v == 'o' || v == 'u')
               {
                  sum += ch * stringLength;
               }
```

```
            else
            {
                sum += ch / stringLength;
            }
        }
        encryptedValues[i] = sum;
    }
    Array.Sort(encryptedValues); // Sort the array in ascending order
    foreach (int value in encryptedValues)
    {
        Console.WriteLine(value);
    }
  }
 }
}
```

## 2. PASCAL TRIANGLE

The triangle may be constructed in the following manner: In row 0 (the topmost row), there is a unique nonzero entry 1. Each entry of each subsequent row is constructed by adding the number above and to the left with the number above and to the right, treating blank entries as 0. For example, the initial number in the first (or any other) row is 1 (the sum of 0 and 1), whereas the numbers 1 and 3 in the third row are added to produce the number 4 in the fourth row.
If you want more info about it: https://en.wikipedia.org/wiki/Pascal's_triangle
Print each row element separated with whitespace.

### EXAMPLES

| Input | Output |
|---|---|
| 4 | 1<br>1 1<br>1 2 1<br>1 3 3 1 |
| 13 | 1<br>1 1<br>1 2 1<br>1 3 3 1<br>1 4 6 4 1<br>1 5 10 10 5 1<br>1 6 15 20 15 6 1<br>1 7 21 35 35 21 7 1<br>1 8 28 56 70 56 28 8 1<br>1 9 36 84 126 126 84 36 9 1<br>1 10 45 120 210 252 210 120 45 10 1<br>1 11 55 165 330 462 462 330 165 55 11 1<br>1 12 66 220 495 792 924 792 495 220 66 12 1 |

### HINTS

- The input number **n** will be `1 <= n <= 60`.
- Think about the proper **type** for the elements of the array.
- Don't be scared to use **more and more arrays**.

```
using System;

namespace _02.PascalTriangle
{
    internal class Program
    {
        static void Main(string[] args)
```

```
        {
            int n = int.Parse(Console.ReadLine());
                // Read the number of rows for Pascal's Triangle
            long[][] triangle = new long[n][];
                // Create a jagged array to hold the triangle
            for (int i = 0; i < n; i++)
            {
                triangle[i] = new long[i + 1];
                      // Initialize each row with the appropriate size
                triangle[i][0] = 1; // The first element of each row is always 1
                triangle[i][i] = 1; // The last element of each row is also always 1
                // Fill in the values for the current row
                for (int j = 1; j < i; j++)
                {
                    triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
                }
            }
            // Print the triangle
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j <= i; j++)
                {
                    Console.Write(triangle[i][j] + " "); // Print each value in the row
                }
                Console.WriteLine(); // Move to the next line after printing a row
            }

        }
    }
}
```

## 3. RECURSIVE FIBONACCI

The Fibonacci sequence is a quite famous sequence of numbers. Each member of the sequence is calculated from the sum of the two previous elements. The **first two** elements are 1, 1. Therefore the sequence goes like 1, 1, 2, 3, 5, 8, 13, 21, 34…
The following sequence can be generated with an array, but that's easy, so your task is to implement recursively.
So if the function **`GetFibonacci(n)`** returns the n$^{th}$ Fibonacci number we can express it using **`GetFibonacci(n) = GetFibonacci(n-1) + GetFibonacci(n-2).`**
However, this will never end and in a few seconds, a StackOverflow Exception is thrown. For the recursion to stop, it has to have a "**bottom**". The bottom of the recursion is **`GetFibonacci(2)`** should return 1 and **`GetFibonacci(1)`** should return 1.

### INPUT

- On the only line in the input, the user should enter the wanted Fibonacci number.

### OUTPUT

- The output should be the n$^{th}$ Fibonacci number counting from 1.

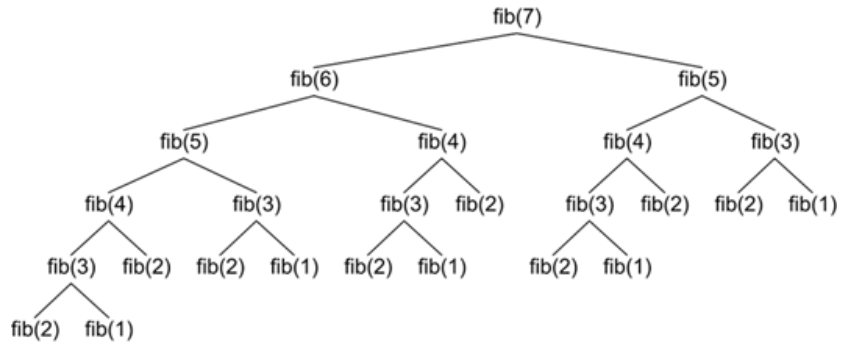
### CONSTRAINTS

- 1 ≤ N ≤ 50

### EXAMPLES

| Input | Output |
|---|---|
| 5 | 5 |
| 10 | 55 |
| 21 | 10946 |

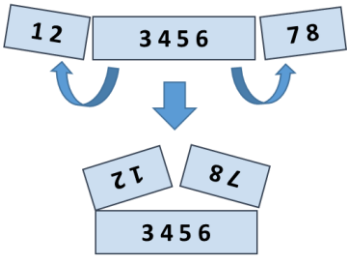
For the N$^{th}$ Fibonacci number, we calculate the N-1$^{th}$ and the N-2$^{th}$ number, but for the calculation of the N-1$^{th}$ number we calculate the N-1-1$^{th}$(N-2$^{th}$) and the N-1-2$^{th}$ number, so we have a lot of repeated calculations.




If you want to figure out how to skip those unnecessary calculations, you can search for a technique called memoization.

## 4. FOLD AND SUM

Read an array of **4*k** integers, fold it like shown below, and print the sum of the upper and lower two rows (each holding 2*k integers):




### EXAMPLES

| Input | Output | Comments |
|---|---|---|
| 5 **2 3** 6 | 7 9 | 5 6 +<br>2 3 =<br>7 9 |
| 1 2 **3 4 5 6** 7 8 | 5 5 13 13 | 2 1 8 7 +<br>3 4 5 6 =<br>5 5 13 13 |
| 4 3 -1 **2 5 0 1 9 8** 6 7 -2 | 1 8 4 -1 16 14 | -1 3 4 -2 7 6 +<br>2 5 0 1 9 8 =<br>1 8 4 -1 16 14 |

### HINTS

- Create the **first row** after folding: the first **k** numbers reversed, followed by the last **k** numbers reversed.
- Create the **second row** after folding: the middle **2*k** numbers.
- **Sum** the first and the second rows.

```
using System;

namespace _04.FoldAndSum
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = Array.ConvertAll(Console.ReadLine().Split(), int.Parse);
                                // Read and parse the input numbers
            int k = numbers.Length / 4;
                        // Calculate k based on the length of the input array
                       // Split the array into two parts: left and right
            int[] leftPart = new int[k];
            int[] rightPart = new int[k];
            Array.Copy(numbers, 0, leftPart, 0, k);
                        // Copy the first k elements to leftPart
```

```
            Array.Copy(numbers, 3 * k, rightPart, 0, k);
                        // Copy the last k elements to rightPart
                        // Reverse both parts
            Array.Reverse(leftPart);
            Array.Reverse(rightPart);
                        // Create the folded array by combining left and right parts
            int[] foldedArray = new int[2 * k];
            for (int i = 0; i < k; i++)
            {
                foldedArray[i] = leftPart[i] + numbers[k + i];
                        // Add corresponding elements from left part and middle part
                foldedArray[k + i] = rightPart[i] + numbers[2 * k + i];
                        // Add corresponding elements from right part and middle part
            }
            Console.WriteLine(string.Join(" ", foldedArray)); // Print the result
        }
    }
}
```

## 5. LONGEST INCREASING SUBSEQUENCE (LIS)

Read a **list of integers** and find the **longest increasing subsequence** (LIS). If several such exist, print the **leftmost**.

### EXAMPLES

| Input | Output |
|---|---|
| **1** | 1 |
| 7 **3 5** 8 -1 0 **6 7** | 3 5 6 7 |
| **1 2** 5 **3 5** 2 4 1 | 1 2 3 5 |
| **0** 10 20 30 30 40 **1** 50 **2 3 4 5 6** | 0 1 2 3 4 5 6 |
| 11 12 13 **3** 14 **4** 15 **5 6 7 8** 7 **16** 9 8 | 3 4 5 6 7 8 16 |
| **3** 14 **5** 12 15 **7 8 9 11** 10 1 | 3 5 7 8 9 11 |

### HINTS

- Assume we have **n** numbers in an array **nums[0…n-1]**.
- Let **len[p]** hold the length of the longest increasing subsequence (LIS) ending at position **p**.
- In a for loop, we shall calculate **len[p]** for **p** = **0** … **n-1** as follows:
  - Let **left** be the leftmost position on the left of **p** (**left** < **p**), such that **len[left]** is the largest possible.
  - Then, **len[p]** = **1** + **len[left]**. If **left** does not exist, **len[p]** = **1**.
  - Also, save **prev[p]** = **left** (we hold in **prev[]** the previous position, used to obtain the best length for position **p**).
- Once the values for **len[0…n-1]** are calculated, restore the LIS starting from position **p** such that **len[p]** is maximal and go back and back through **p** = **prev[p]**.
- The table below illustrates these computations:

| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **nums[]** | **3** | **14** | **5** | **12** | **15** | **7** | **8** | **9** | **11** | **10** | **1** |
| **len[]** | 1 | 2 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 6 | 1 |
| **prev[]** | -1 | 0 | 0 | 2 | 3 | 2 | 5 | 6 | 7 | 7 | -1 |
| **LIS** | {3} | {3,14} | {3,5} | {3,5,12} | {3,5,12,15} | {3,5,7} | {3,5,7,8} | {3,5,7,8,9} | {3,5,7,8,9,11} | {3,5,7,8,9,10} | {1} |

```
using System;
namespace _05.LongestIncreasingSubsequence
{
    internal class Program
    {
        static void Main(string[] args)
        {
```

```
        int[] nums= Array.ConvertAll(Console.ReadLine().Split(), int.Parse); // Read and parse the input numbers
        int n = nums.Length; // Get the length of the input array
        int[] len = new int[n]; // Create a DP array to store the lengths of LIS ending at each index
        int[] prev = new int[n]; // Create an array to store the previous index in the LIS
        for (int i = 0; i < n; i++)
        {
            len[i] = 1; // Initialize the length of LIS ending at each index to 1
            prev[i] = -1; // Initialize the previous index to -1 (no previous element)
        }
        int maxLength = 1; // Variable to keep track of the maximum length of LIS found
        int endIndex = 0; // Variable to keep track of the index where the maximum LIS ends
        for (int i = 1; i < n; i++) // Iterate through the array starting from the second element
        {
            for (int j = 0; j < i; j++) // Check all previous elements
            {
                if (nums[i] > nums[j] && len[i] < len[j] + 1) // If current element is greater and can extend the LIS
                {
                    len[i] = len[j] + 1; // Update the length of LIS ending at i
                    prev[i] = j; // Update the previous index to j
                }
            }
            if (len[i] > maxLength) // If we found a longer LIS
            {
                maxLength = len[i]; // Update the maximum length
                endIndex = i; // Update the end index of the LIS
            }
        }
        // Reconstruct the longest increasing subsequence
        int[] lis = new int[maxLength]; // Create an array to hold the LIS
        int currentIndex = endIndex; // Start from the end index of the LIS
        for (int i = maxLength - 1; i >= 0; i--) // Fill the LIS array in reverse order
        {
            lis[i] = nums[currentIndex]; // Add the current element to the LIS
            currentIndex = prev[currentIndex]; // Move to the previous index in the LIS
        }
        Console.WriteLine(string.Join(" ", lis)); // Print the longest increasing subsequence
    }
  }
}
```