**More Exercises: Arrays**

Problems for more exercises and homework for the ["Technology Fundamentals" course @ SoftUni](https://softuni.bg/courses/technology-fundamentals).

You can check your solutions in [Ju HYPERLINK "https://judge.softuni.bg/Contests/1268/" HYPERLINK "https://judge.softuni.bg/Contests/1268/" HYPERLINK "https://judge.softuni.bg/Contests/1268/"d HYPERLINK "https://judge.softuni.bg/Contests/1268/" HYPERLINK "https://judge.softuni.bg/Contests/1268/" HYPERLINK "https://judge.softuni.bg/Contests/1268/"ge](https://judge.softuni.bg/Contests/1268/).

* **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 on the console.

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

**Examples**

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

* **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 elements separated with whitespace.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 | 1  1 1  1 2 1  1 3 3 1 |
| 13 | 1  1 1  1 2 1  1 3 3 1  1 4 6 4 1  1 5 10 10 5 1  1 6 15 20 15 6 1  1 7 21 35 35 21 7 1  1 8 28 56 70 56 28 8 1  1 9 36 84 126 126 84 36 9 1  1 10 45 120 210 252 210 120 45 10 1  1 11 55 165 330 462 462 330 165 55 11 1  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 proper **type** for elements in the array
* Don’t be scared to use **more and more arrays**
* **Recursive Fibonacci**

The Fibonacci sequence is quite a 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 as 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. In order 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 Format:**

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

**Output Format:**

* 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 Nth Fibonacci number, we calculate the **N-1th** and the **N-2th** number, but for the calculation of **N-1th** number we calculate the **N-1-1th(N-2th)** and the **N-1-2th** number, so we have a lot of repeated calculations.



* **Jump Around**

You will receive an **integer** **array** from the console. You **start** from the **beginning** of the array and try to **move** **right** by a **step**, equal to the **value** at position **0**. If that is **possible** you should **collect** the **value** from the **index** on which you landed, and try to move to the **right** by **its** **value**, if that is **not** possible – try to move to the **left**. If that is also **not** possible **stop** the program and print the **sum** of the collected **values**. Example:

Example: We have the array [**3 7 12 2 10]**. We **start** from **3** and move **3 indices** to **2**. We have to move **2 indices**, but we **can’t** **move** to the **right**, so we move to the **left** to **7**. From there we **cannot** move **anywhere** and we **stop** the program and we print the sum of the collected cells: **3 + 2 + 7 =** **12**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 7 | 12 | 2 | 10 |

**Input**

The input consists of **single** line, which will be an **array** of **integers**.

**Constraints**

* The array will have at most **50** elements
* The elements in the array will be in the interval **[1…50]**

**Examples**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| 10 50 7 30 8 5 | 10 |  | 2 3 5 7 5 4 8 4 | 18 |

* **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]** holds 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** is 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 if **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} |