ITERATION

WORKSHEET

CENG 111

2019

GÖKTÜRK ÜÇOLUK

DIFFICULTY

Check if string is palindrome ⭐

[Was present in the RECURSION WORKSHEET, now you are expected to do it iteratively]

Description

Write a function, named **`is_palindrom`**, that takes a string as argument and returns **`True`** if the string is a palindrome, **`False`** otherwise.

Example

```
>>> is_palindrom(“BABA”)
False

>>> is_palindrom(“BABAB”)
True

>>> is_palindrom(“LEBLEBI”)
False

>>> is_palindrom(“LEBLEBI”)
False

>>> is_palindrom(“Rotator”)
False

>>> is_palindrom(“RotatoR”)
True
```

First Recamán's Sequence ⭐

[Was present in the RECURSION WORKSHEET, now you are expected to do it iteratively]

Introduction

The First Recamán's Sequence R is defined as

- R(0) = 0
- R(n) = R(n − 1) − n, if n > 0 ∧ the RHS is positive ∧ R(n) is not already in the sequence,
- R(n) = R(n − 1) + n, otherwise

which can be succinctly defined as "subtract if you can, otherwise add."
The first few terms are 0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, ..

Description

Write a function, named **`recaman`**, that takes a positive integer (**`n`**) as argument and returns R(n).

Example

```
>>> recaman(6)
13
```

Note

Care about efficiency !

Greatest Common Divisor ⭐

[Was present in the RECURSION WORKSHEET, now you are expected to do it iteratively]

Description

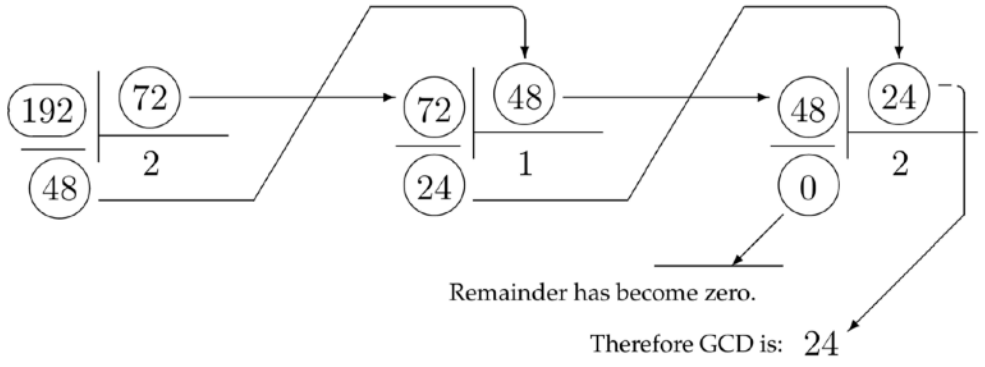
Write a function, named **`gcd`**, that takes two positive integers (**`k,n`**) as argument and returns their greatest common divisor.

Example

```
>>> gcd(192,72)
24

>>> gcd(72,192)
24

>>> gcd(16,15)
1
```



Remove duplicates ⭐

[Was present in the RECURSION WORKSHEET, now you are expected to do it iteratively]

Description

Write a function, named **`remove_duplicate`**, that takes a single string argument (**`s`**). The function removes all all but the last of adjacent duplicates.

Example

```
>>> remove_duplicate('geeksforgeeeeek')
'geksforgek'

>>> remove_duplicate('acaaabbbacdddd')
'acabacd'
```

Coalesce ⭐⭐⭐

[Was present in the RECURSION WORKSHEET, now you are expected to do it iteratively]

Introduction

Some elements of a set may be equivalent. Such equivalences can be expressed using a list of pairs of equivalent elements. Equivalence is symmetric and transitive. Equivalence classes are subsets, all elements of which are equivalent.

Description

Write a function **`coalesce`** which takes a list of pairwise equivalences and return a list of equivalence classes.

Example

```
>>> coalesce([['a','e'],['z','f'],['m','b'], \
['p','k'], ['e','i'],['f','s'],['b','d'],['t','p'], \
['i','o'],['s','v'],['d','g'],['k','p'],['o','u'], \
['v','z'],['g','m'],['p','t']])

[['a','e','i','o','u'],['f','s','v','z'],
['b','d','g','m'],['k','p','t']]
```

Convert positive integer to binary 0/1 string⭐

Description

Write a function, named **`convert_to_bin`**, that takes a single string argument. The function converts the argument integer into its binary representation expressed as a string of “`1`” and “`0`” s.

Example

```
>>> convert_to_bin(456)
'111001000'

>>> remove_duplicate(56324432345224525)
'110010000001101011000101110101100001110011010101010100110
1'
```

Note

- There is a built-in function **`bin()`**, you can make use of this function to check your result. Certainly it is not wise to use it in cooking up your solution: that would not serve the purpose of the worksheet.
- Python assumes unlimited size in integers. So do not make any presumption on the count of digits, etc. On the other hand, all arithmetics is working for these ‘unlimited’ integers.

Rotate a sub-square in a matrix by multiples of 90^0 ⭐⭐⭐

Description

Write a function, named **`rotate_sub_mat()`**, that takes four arguments:

- **1. argument:** A list with *n* elements, each of which are themselves lists of *m* elements. This is a $n \times m$ (square) matrix in row order.
- **2. argument:** A tuple *(i, j)* that points to the upper-left element of sub-matrix which is going to be rotated, where *i* is the row and *j* is the column (counting starts at zero)
- **3. argument:** A positive integer *k* which is the width (and height) of the sub-matrix that is going to be rotated. ($i+k \leq n$ and $j+k \leq m$) is assured.
- 4**. argument:** An integer *r* in the range [1,3]. This is the multiplicative factor for 90^0, how much the sub-matrix is going to be rotated clockwise.

Example

$$\begin{bmatrix} 45 & -6 & 10 & 26 & 92 & -2 \\ 1 & 0 & 0 & 7 & 99 & -1 \\ 19 & 46 & 0 & 66 & 13 & 0 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 17 & 27 & 37 & 47 & 57 \\ -2 & -1 & 0 & 1 & 2 & 3 \end{bmatrix}$$

A 3×90^0 ($=270^0$) rotation will yield:

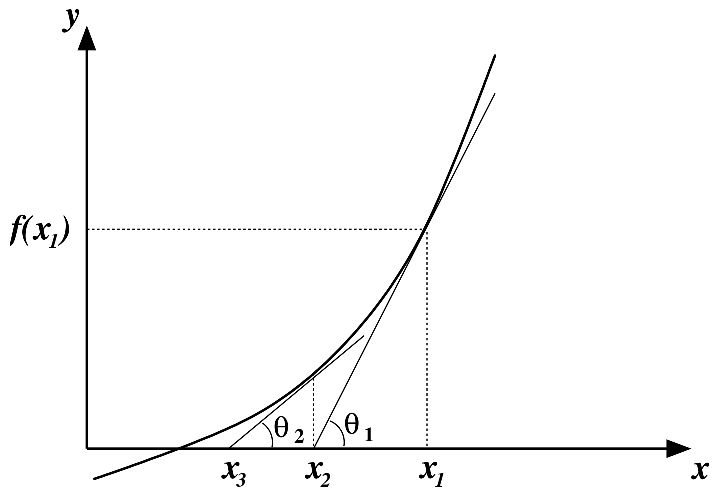
$$\begin{bmatrix} 45 & -6 & 10 & 26 & 92 & -2 \\ 1 & 0 & 99 & 13 & 5 & -1 \\ 19 & 46 & 7 & 66 & 4 & 0 \\ 1 & 2 & 0 & 0 & 3 & 6 \\ 7 & 17 & 27 & 37 & 47 & 57 \\ -2 & -1 & 0 & 1 & 2 & 3 \end{bmatrix}$$

```
>>> rotate_sub_mat([[45,-6,10,26,92,-2],[1,0,0,7,99,-1],
[19,46,0,66,13,0],[1,2,3,4,5,6],[7,17,27,37,47,57],
[-2,-1,0,1,2,3]], (1,2), 3, 3)
[[45,-6,10,26,92,-2],[1,0,99,13,5,-1],[19,46,7,66,4,0],
[1,2,0,0,3,6],[7,17,27,37,47,57],[-2,-1,0,1,2,3]]
```

Root finding by Newton-Raphson method ⭐⭐⭐

Introduction

A so called Newton-Raphson method is a good and robust way to determine numerically the positions of the roots in a given interval. The technique makes use of the concept of the tangent line of a function. Unless the slope of a tangent line is zero, it is bound to intersect the x-axis.



This point of intersection is easily calculated:

$$\tan\theta_1 = \frac{f(x_1)}{x_1 - x_2} = f'(x_1)$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

x_2 is closer to the root compared to x_1. It can be proven that if this process is iteratively continued in the limit x_n will meet the root.
So, generalizing

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In the application of this iterative algorithm it is important to choose the initial point x_1 in the close proximity to the root.

The derivative is also to be calculated in numerical form. As you know the value of a derivative at a given point a is defined by:

$$\left.\frac{df(x)}{dx}\right|_{x=a} = f'(a) = \lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$$

Of course in real life to take the limit is unrealistic.
h has to be chosen as a small value. To approximate the real limit, it is wise to take the average of the right hand side where h is first a positive small value and then a negative one.
Let us assume the magnitude of this small value is δ, then the numeric derivative value at a is calculated as:

$$f'(a) = \frac{f(a+\delta)-f(a-\delta)}{2\delta}$$

Combining this tool of numerical differentiation with the Newton-Raphson method we are able to compute the roots of an analytic function in a given interval.

Description

Write a function **`find_root()`** that has:

- **1. argument:** The name of the function that's root of which is going to be seeked.
- **2. argument:** A tuple of two floats, `(lower, upper)`, that defines the search interval.
- **3. argument:** A floating point that defines the tolerance.
- **Return value:** A list of roots found.

Example

```
>>> def f(x): return x*x * sin(x+1) + x

>>> find_root(f, (-8.7,11.1), 0.001)
[-7.142721, -4.372346, 0.000008, 2.545359, 5.085252,
8.542115]
```

Sum of a sub interval of a list ⭐-⭐⭐

Description

The task is to write a function **`sub_interval_sum()`**, where you are given a list of numbers L as the first argument. The second argument is a list containing a number of index intervals each in the form of $(start\_index_i, end\_index_i)$. The return value is a list with the values:

$$\left[\sum_{k=start\_index_1}^{end\_index_1} L_k, \sum_{k=start\_index_2}^{end\_index_2} L_k, \ldots, \sum_{k=start\_index_n}^{end\_index_n} L_k\right]$$

Tasks

- **The simple solution:** If the first argument is of length N and the second is of length n do it in $\Theta(N \times n)$ complexity.
- **The profi solution:** Do it in $\Theta(N + n)$ complexity.

Example

```
>>> L = [-45, 38, 47, 21, 45, -36, 19, -22, 4, 31, -38,
-24, 39, 49, 28, -36, -5, -3, 23, -27]

>>> intervals = [(0,3),(1,9),(10,15),(0,19),(7,17)]

>>> sub_interval_sum(L,intervals)
[61, 147, 18, 108, 23]
```

Sieve of Eratosthenes⭐⭐

Introduction

Read material at:
`en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

Description

Write a function, **`generate_primes()`**, that takes a single argument *n*, and returns a list of all primes less then *n* by making use of the Sieve method.

Example

```
>>> generate_primes(200)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199]
```

K-Means clustering⭐⭐⭐⭐

Introduction

Assume you are given a set of points in the plane as (x_i, y_i).
You want to partition these points into k clusters.

A cluster is a group of points which are near to each other.
There is an excellent introductory material that explains K-means clustering. We advise you to read it:
`bigdata-madesimple.com/possibly-the-simplest-way-to-explain-k-means-algorithm/`

A more complete and advanced cover can be found at:
`en.wikipedia.org/wiki/K-means_clustering`

A run-through example can be found at:
`mnemstudio.org/clustering-k-means-example-1.htm`

Description

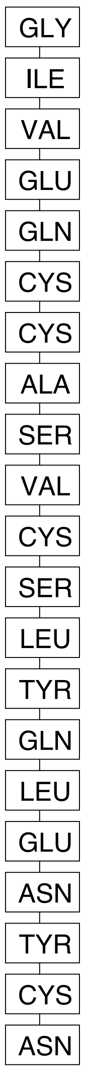
Write a function, **`k_means()`**, that takes two arguments, a list of data points and an integer *k* (number of cluster), and returns a list with *k* sublists. Each sublist holds the points of one cluster.

Example

```
>>> k_means([(1.0,1.0),(3.0,4.0),(5.0,7.0),(3.5,5.0),
(4.5,5.0),(3.5,4.5),(1.5,2.0)], 2)
[[(1.0,1.0),(1.5,2.0)],[(3.0,4.0),(5.0,7.0),(3.5,5.0),
(4.5,5.0),(3.5,4.5)]]
```

DNA ⭐⭐⭐⭐

Introduction



The life form on our planet is primarily based on a molecule that we call DNA. It carries within its structure the heredity information that determines the structure of proteins and the instructions that direct cells to grow and divide. So are the messages that bring about the differentiation of fertilized eggs into the multitude of specialized cells that are necessary for the successful functioning of higher plants and animals.

In DNA molecules, *nucleotides* are linked together to form long chains by bonds. The order and sequence of this chain is the information content of the DNA. Each DNA molecule contains many subparts that we call *gene*. A gene is the smallest functional unit and serves as a 'program' that synthesizes a protein (a chain of amino-acids).

On the left you see a part of such a synthesed protein: The A-chain of an *insulin*.

Just for your information the names of the building blocks of proteins namely the Amino acids are:

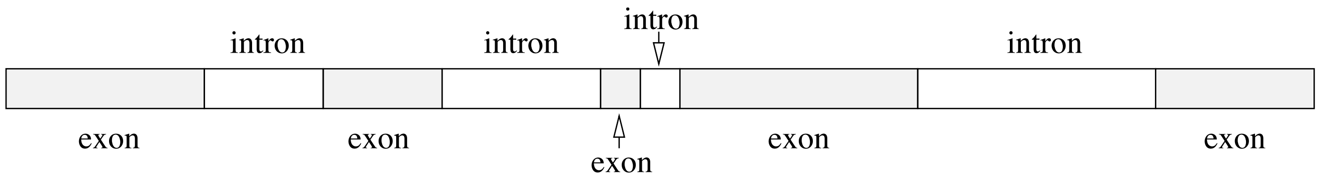
Glycine	GLY	Lysine	LYS
Alanine	ALA	Arginine	ARG
Valine	VAL	Asparagine	ASN
Isoleucine	ILE	Glutamine	GLN
Leucine	LEU	Cysteine	CYS
Serine	SER	Methionine	MET
Threonine	THR	Tryptophan	TRP
Proline	PRO	Phenilalanine	PHE
Asparic acid	ASP	Tyrosine	TYR

A nucleotide is one of the the four molecules:

A	Adenine
C	Cytosine
G	Guanine
T	Thymine

In a gene, each group of three successive nucleotides is called a *codon*. Each codon is responsible of synthesizing an amino acid (one of the 20 amino acids given in the table above).

Although a gene is responsible of synthesis of a protein, it has parts which are `garbage' and parts which are `functional'. These parts are called *introns* and *exons*, respectively. So, it is actually the exon zones which produces the protein and introns can practically be omitted from the sequence.



When it is time to produce a protein, a kind of a mask is generated from the gene. This mask is called the messenger RNA (mRNA) which also consists of only four kind of nucleotides. These are

A	Adenine
C	Cytosine
G	Guanine
U	Urasil

It is only the useful parts that are masked, namely the exons. The introns are simply ignored. This mask generation is called *transcription*. Each nucleotide in the gene is represented by a one-to-one correspondence.

Here is the table of transcription:

Original DNA nucleotide	Transcripted into mRNA nucleotide
A	U
C	G
G	C
T	A

After the transcription is completed, the codons of the mRNA are used to manufacture the protein. Since there are 4 nucleotides, as you might have realized already, there are 4×4×4 possibilities for them to group in four. That means there are 64 codons. But as we know they synthesize only 20 amino acids and some codons stop the process. So, evidently, some codons must synthesize the same amino-acid. That is true, and is called *degeneracy*.

First position	Second position				Third position
	U	C	A	G	
U	PHE	SER	TYR	CYS	U
	PHE	SER	TYR	CYS	C
	LEU	SER	stop	stop	A
	LEU	SER	stop	TRP	G
C	LEU	PRO	HIS	ARG	U
	LEU	PRO	HIS	ARG	C
	LEU	PRO	GLN	ARG	A
	LEU	PRO	GLN	ARG	G
A	ILE	THR	ASN	SER	U
	ILE	THR	ASN	SER	C
	ILE	THR	LYS	ARG	A
	MET	THR	LYS	ARG	G
G	VAL	ALA	ASP	GLY	U
	VAL	ALA	ASP	GLY	C
	VAL	ALA	GLU	GLY	A
	VAL	ALA	GLU	GLY	G

WHICH MRNA CODON IS SYNTHESIZING WHICH AMINO ACID.

Description

You will be given a string of nucleotides which represent a gene. You know also that there is exactly 3 exons and 2 introns. The gene starts with an exon and ends with an exon. You don't know where the intermediate exon is located. You also know that the both of the introns are non-empty. Since exons are sequences of codons (three nucleotides together) they are a multiple of 3. This is not so for introns, since they are garbage their length has not to be a multiple of 3.

You are expected to find the introns' start and end positions. The answer shall be given a list of two-tuples. Where each tuple is a pair of two positions, namely an intron's sart and end position. The function that will do as described shall be named as **`find_intron()`.** It will receive the gene sequence as a string from its first argument, and the protein as a list of strings from the second argument. If you discover that this protein is not the product of the gene given, then you shall return **`None`**.

Example

```
>>> gene =
"TCTGCAGCAGAGGGGCCGTCGGCAGAAGGAGGGCTCGGGCAGGCTCTGCGACTCG
TAGGCACCAGGCGTGAGACCTGTAGCCCCCGATCACCATGTACAGCTTCATGGGTG
GTGGCCTGTTCTGTGCCTGGGTGGGGACCATCCTCCTGGTGGTGGCCATGGCAACA
GACGGGGCCAAGGACACCTGTATTCCAGATGGAGAACTCTGCGGCTCAAAGAGGGA
AAGGGAGCAACCCAAGGTCACTCAGCGGAGGCTGACTCCTGGTCCTAGGCTGGAAG
GAGGAAGAATAGGGCCCATGGGAGGGAGCTGAGAAGACT"

>>> protein =
['ARG','ARG','ARG','LEU','PRO','GLY','SER','ARG','LEU','
PRO','PRO','GLU','PRO','VAL','ARG','ASP','ALA','GLU','LE
U','VAL','VAL','HIS','VAL','GLU','VAL','PRO','THR','THR'
,'GLY','GLN','ASP','THR','ASP','PRO','PRO','LEU','VAL','
GLY','GLY','PRO','PRO','PRO','VAL','PRO','LEU','SER','PR
O','PRO','THR','GLU','ASP','GLN','ASP','PRO','THR','PHE'
,'LEU','LEU','LEU','ILE','PRO','GLY','THR','LEU','PRO','
ARG','LEU','PHE','stop']

>>> find_intron(gene,protein)
[(55,85),(170,249)]
```

Gaussian Elimination ⭐⭐⭐⭐

Introduction

Read material at:
`en.wikipedia.org/wiki/Gaussian_elimination`

Description

Write a function, **`gaussian_elimination()`** that implements *Gaussian Elimination*. This function shall takes two arguments. The fist argument is a list of lists, representing the coefficient (square) matrix in row form. The second is the right hand side column matrix (a vector), expressed as a list. The entries, both of the coefficient matrix and the vector, are represented NOT as floating points, but fractions. You will represent fractions by two-tuples: `(`*numerator*`,`*denominator*`)`, the *denominator* is always positive and GCD of *numerator* and *denominator* is cancelled out. **`gaussian_elimination()`**, is expected to return a list of fractions, the solution to the system.

Example

```
>>> gaussian_elimination([[(8,1),(1,1)(-1,1)],[(-12,1),
(-1,1),(2,1)],[(-8,1),(1,1),(2,1)]], [(8,1),(-11,1)
(-3,1)])
[(1,2),(3,1)(-1,1)]
```

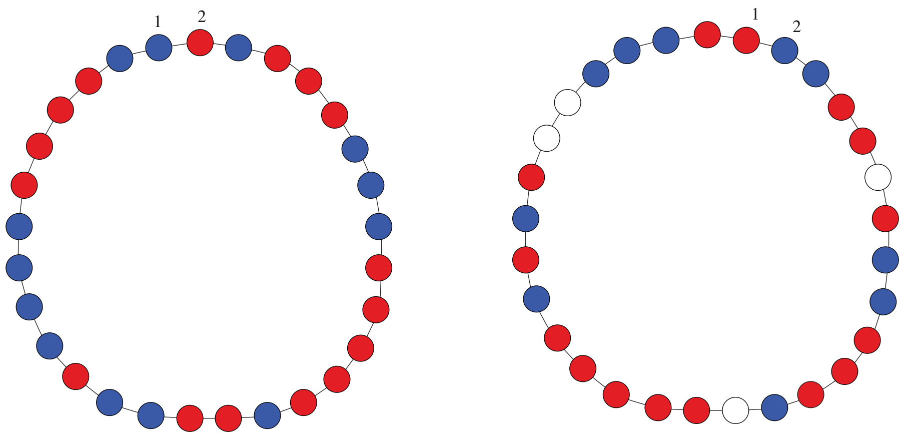
Note

Do not attempt to do the arithmetic in floating points and then, finally, convert it into fractions. That won't work. You have to implement fraction arithmetic, and perform all arithmetical operations in this domain.

Neclace ⭐⭐⭐

Introduction

You have a necklace of n beads (n <100) some of which are red, others blue and others white, arranged at random. Let's see two examples for n = 29:



(The beads considered first and second in the text that follows have been marked in the picture with 1 and 2)

The configuration in the left figure may be represented as a string of **`b`**'s and **`r`**'s, where **`b`** represents a blue bead and **`r`** a red one, as follows:
`brbrrrbbbrrrrrbrrbbrbbbbrrrrb`

Suppose you are to break the necklace, lay it out straight, and then collect beads of the same color from one end until you reach a bead of a different color, and do the same for the other end (which may not be of the same color as the beads collected before this).

Determine the point where the necklace should be broken so that the **most number of beads can be collected.**

For example, for the necklace in the left figure above, 8 beads can be collected, with the breaking point either between bead 9 and bead 10, or between bead 24 and bead 25.

In some necklaces, white beads had been included as shown in right figure above. When collecting beads, a white bead that is encountered may be treated as either red or blue, and painted with the desired color. The string that represents this configuration will include the symbols: **`r`**, **`b`** and **`w`**.

Description

Write a function, **`break_neclace()`**, which takes one argument, a string that represents a neclace configuration, and returns a two-tuple, namely the maximum number of beads collectable, along with a breaking point (the bead number after which the break is performed)

Example

```
>>> break_neclace("brbrrrbbbrrrrrbrrbbrbbbbrrrrb")
(8,9)

>>> break_neclace("bbwbrrrwbrbrrrrrb")
(10,16)
```

Island⭐⭐⭐⭐⭐

Introduction

The SEA is represented by an N × N grid. Each ISLAND is a **"*"** on that grid. The task is to reconstruct a MAP of islands only from some CODED INFORMATION about the horizontal and vertical distribution of the islands. To illustrate this code, consider the following map:

```
*   * *       1 2
  * * *   *   3 1
*   *   *     1 1 1
  * * * * *   5
* *   *   *   2 1 1
      *       1

1 1 4 2 2 1
1 2   3   2
1
```

The numbers on the right of each row represent the order and size of the groups of islands in that rows. For example, **"1 2"** in the first row means that this row contains a group of one island followed by a group of two islands; with sea of arbitrary length to the left and right of each island group. Similarly, the sequence **"1 1 1"** below the first column means that this column contains three groups with one island each, etc.

Description

Write a function **`decode_map()`**, that takes two list as arguments. Both lists are lists of lists. Each sublist is the information about a row/column. **`decode_map()`** is expected to print the map similar to the example above. Each blank must be represented by a pair of spaces. Each island should be represented by a `'*'` followed by a space.

Example

```
>>> decode_map([[1,2],[3,1],[1,1,1],[5],[2,1,1],[1]],
[[1,1,1],[1,2],[4],[2,3],[2],[1,2]])

*   * *       1 2
  * * *   *   3 1
*   *   *     1 1 1
  * * * * *   5
* *   *   *   2 1 1
      *       1

1 1 4 2 2 1
1 2   3   2
1
```