Exercises for Elementary Algorithmic Programming in Python

There are three ways to learn computer programming: *practicing, practicing, and practicing.* So, get on with it! This document contains a list of exercises intended to practice your basic, algorithmic programming skills in Python.

The focus of these exercises is on the *algoritmic* aspects of programming. Some features of the Python language and its libraries hide some of these fundamental aspects. Many innocent-looking expressions hide linear complexities. For example, you can check whether an array A contains an element x with a simple expression `x in A`, or you can get (or "splice") values or entire sequences in and out of arrays with expressions such as `A[1:]`, `A[2:3]`, `A[1:1] = B`. And of course, many algorithms are already implemented in standard library functions (e.g., `min()`, `max()`, `sort()`). Here we deliberately avoid all that.

You must therefore solve all these exercises using only a limited subset of the Python language and libraries. In particular, You may only use the following built-in types:

- numeric types, such as `int`
- sequence types, such as arrays, tuples, and strings (so, no sets or dictionaries)

With arrays or other sequence types, you may only use the following features:

- direct access to an element by index, as in `print(A[7])` or `A[i+1] = A[i]`
- append an element, as in `A.append(10)`
- delete the last element, as in `del A[-1]` or `A.pop()`; deleting any other element of a sequence is not permitted
- read the length, as in `n = len(A)`
- shrink to a given length, as in `del A[length:]`

You may also use the `range` function, typically in a for-loop, as in `for i in range(10)`.

You may not use any library or external function other than the ones listed above.

Median Value

Write a function `median_value(a,b,c)` that, given three numbers a, b, c returns their median value.

Examples

```
>>> median_value(1,2,3)
2
>>> median_value(3,2,1)
2
>>> median_value(7, 3, 21)
7
>>> median_value(7, 3, 5)
5
>>> median_value(7, 3, 3)
3
>>> median_value(7, 3, 7)
7
```

Solution: here

Leap Year

Write a function `leap_year(y)` that, given a year number y in the Gregorian calendar, return `True` if y is a leap year, or `False` otherwise. Recall that a leap year is one whose number is divisible by 4, excluding the year numbers divisible by 100, but including the year numbers divisible by 400.

Examples

```
>>> leap_year(2000)
True
>>> leap_year(1969)
False
>>> leap_year(2023)
False
>>> leap_year(1984)
True
>>> leap_year(2022)
False
>>> leap_year(2200)
False
>>> leap_year(2400)
True
>>> leap_year(1900)
False
```

Solution: here

Classify Triangle

Write a function `classify_triangle(a,b,c)` that, given three positive numbers representing the lengths of three segments, respectively, output a classification of the triangle obtained by connecting the three segments. The output consists of one or two words printed on a single line and separated by a single space. The first word is one of `acute`, `right`, `obtuse`, or `impossible`. `impossible` indicates that it is impossible to form a triangle with the given segment lengths, in which case the output ends there. `acute`, `right`, and `obtuse` indicate that the resulting triangle has all acute angles, one right angle, or one obtuse angle. In these cases, the output must contain a second word that can be either `scalene`, `isosceles`, or `equilateral`, indicating the type of triangle.

Examples

```
>>> classify_triangle(10,10,10)
acute equilateral
>>> classify_triangle(4,3,5)
right scalene
>>> classify_triangle(4,3,8)
impossible
>>> classify_triangle(3,4,3)
acute isosceles
>>> classify_triangle(3,5,3)
obtuse isosceles
>>> classify_triangle(5,5,7)
acute isosceles
```

Solution: here

Minimum

Write a function `minimum(A)` that, given an array A of numbers, returns the minimum value in A. You may assume that the sequence contains at least one element. Recall that you may not use the built-in function `min()`.

Examples

```
>>> minimum([1,2,3])
1
>>> minimum([3,2,1])
1
>>> minimum([100,2,55,4,3,2,67,3])
2
```

```
>>> minimum([7])
7
```

Solution: here

Position in Sequence

Write a function `position_in_sequence(A,x)` that returns the first position in which value x appears in A. Positions start from 0. Return -1 if x does not appear in A.

Examples

```
>>> position_in_sequence([6, 7, 10, -2, 3], 7)
1
>>> position_in_sequence([3, 2, 1, 1, 3, 2, 2, 3, 1], 1)
2
>>> position_in_sequence([], 1)
-1
>>> position_in_sequence([2, 3, 4, 5], 1)
-1
>>> position_in_sequence([2, 3, 4, 5], 2)
0
```

Solution: here

Count Lower

Write a function `count_lower(A,x)` that returns the number of elements in A whose value is less than x.

Examples

```
>>> count_lower([0,7,10,-2,3], 7)
3
>>> count_lower([3, 2, 1, 1, 3, 2, 2, 3, 1], 2)
3
>>> count_lower([3, 2, 1, 1, 3, 2, 2, 3, 1], 10)
9
>>> count_lower([], 1000000000000000)
0
>>> count_lower([2, 3, 4, 5], 1)
0
```

Solution: here

Multiples of Three

Write a function `multiples_of_three(A)` that takes an array A of integers and prints the count of all the elements of A that are multiples of three.

Examples

```
>>> multiples_of_three([34, 31, 45, 5, 38, 19, 19, 26, 25, 19,
2
>>> multiples_of_three([7, 2, 0])
0
```

Check Sorted

Write a function `check_sorted(A)` that returns `True` if the given sequence A is sorted in non-decreasing order, or `False` otherwise.

Examples

```
>>> check_sorted([1,2,3])
True
>>> check_sorted([1,3,2])
False
>>> check_sorted([])
True
>>> check_sorted([7])
True
>>> check_sorted([50,50,50])
True
>>> check_sorted([43,51,51])
True
>>> check_sorted([43,51,50,51,70])
False
```

Solution: here

Monotonic Sequence

Write a function `is_monotonic(A, i, j)` that, given an array of numbers, A, returns

`True` if A contains a *monotonic* sub-sequence starting at position i and ending at position j, or `False` otherwise. A monotonic sequence is one that is either in non-decreasing or non-increasing order.

Examples

```
>>> is_monotonic([1,2,3], 0, 2)
True
>>> is_monotonic([1,1,7,7,9], 0, 4)
True
>>> is_monotonic([9,9,5], 0, 2)
True
>>> is_monotonic([6,6,6,6,6,6], 2, 4)
True
>>> is_monotonic([1,1,1,2,1,3], 0, 5)
False
>>> is_monotonic([1,1,1,2,1,3], 0, 3)
True
>>> is_monotonic([3,2,1,3,2,1], 0, 5)
False
>>> is_monotonic([3,2,1,3,2,1], 2, 3)
True
>>> is_monotonic([3,2,1,3,2,1], 2, 4)
False
>>> is_monotonic([3,2,1,3,2,1], 3, 5)
True
>>> is_monotonic([7,4,7], 0, 2)
False
>>> is_monotonic([7,4,7], 1, 2)
True
```

Solution: here

Explain, Analyze, and Improve an Algorithm!

Consider the following algorithm:

```
def algorithm_x (A):
    x = 0
    for i in range(len(A)):
        for j in range(i+1, len(A)):
            if is_monotonic (A, i, j):
                if x < j - i:
                    x = j - i
    return x
```

Question 1

Explain what `algorithm_x` does. Do not paraphrase the code. Instead, explain the high-level semantics of the algorithm.

Question 2

Analyze the complexity of `algorithm_x`. Give your best characterization of the complexity of the algorithm.

Question 3

Write an algorithm `better_algorithm_x(A)` that is functionally identical to `algorithm_x(A)` but with a strictly better time complexity.

Question 4

Write an algorithm `linear_algorithm_x(A)` that is functionally identical to `algorithm_x(A)` and that runs in time $O(n)$.

Solution: here

Find the Peak

Write an algorithm `find_peak(A)` that, given a "peak" sequence A, finds the maximal value x in A. A "peak" sequence A consists of an increasing sequence $A_{1,\ldots,i}$ attached to a decreasing sequence $A_{i,\ldots,n}$, for some index i. These two sequences share exactly one element, namely A_i.

Examples

```
>>> find_peak([1,2,3])
3
>>> find_peak([1,2,7,8,9])
9
>>> find_peak([9,6,5])
9
>>> find_peak([1,2,1,-3])
2
>>> find_peak([1,2,1,0,-3])
2
>>> find_peak([1,2,3,2,1])
3
>>> find_peak([1,2,3,4])
```

```
4
>>> find_peak([1,2])
2
>>> find_peak([4,7,4])
7
>>> find_peak([7])
7
>>> find_peak([7,4])
7
>>> find_peak([4,7])
7
```

Log Base Two

Write a function `log_base_two(n)` that, given a positive integer n, returns the integer logarithm base-two of n, that is, the maximal integer k such that $2^k \leq n$.

Examples

```
>>> log_base_two(1)
0
>>> log_base_two(2)
1
>>> log_base_two(5)
2
>>> log_base_two(1000)
9
```

Maximal Difference

Write a function `maximal_difference(A)` that returns the maximal difference between any two elements of the given sequence A.

Examples

```
>>> maximal_difference([2, 1, 5, 9, 4, 10, 8])
9
>>> maximal_difference([1])
0
>>> maximal_difference([1, 1, 1])
0
>>> maximal_difference([10,-3, 4, 11, 0, 9])
```

```
14
```

Minimal Sum

Write a function `minimal_sum(A, x)` that returns `True` if there are some elements in A whose sum is greater or equal to x, or `False` otherwise.

Examples

```
>>> minimal_sum([], 1)
False
>>> minimal_sum([1], 1)
True
>>> minimal_sum([3, 2], 4)
True
>>> minimal_sum([3, -2], 4)
False
>>> minimal_sum([2, 1, 5, -3, 9, 4], 20)
True
>>> minimal_sum([32,-3,10,7,-4,18,25], 50)
True
>>> minimal_sum([32,-3,10,7,-4,18,25], 94)
False
```

Isolated Elements

Write a function `isolated_elements(A)` that prints, in order, all the elements that are different from all their adjacent elements, that is, each element that is different from the previous element (if that exists) and the next element (if that exists).

Examples

```
>>> isolated_elements([])
>>> isolated_elements([1])
1
>>> isolated_elements([7,3])
7
3
>>> isolated_elements([2, 2])
>>> isolated_elements([2, 2, 3, 2, 3])
3
2
```

```
3
>>> isolated_elements([-2, 2, 2, 2, -2])
-2
-2
>>> isolated_elements([1,2,1,2,1,2,1,2])
1
2
1
2
1
2
1
2
```

Repeated Elements

Write a function `repeated_adjacent_elements(A)` that prints, in order, all the elements that are repeated in sub-sequences of two or more elements. For every maximal subsequence of identical values, `repeated_adjacent_elements(A)` must print that value *only once.*

Examples

```
>>> repeated_adjacent_elements([])
>>> repeated_adjacent_elements([1])
>>> repeated_adjacent_elements([1, 2])
>>> repeated_adjacent_elements([3, 3])
3
>>> repeated_adjacent_elements([1,-1,7,7,-1,7,1,7,7,7,7,2,2])
7
7
2
```

Compression

Write a function `compress_sequence(A)` that prints a "compressed" version of the given sequence A. Compression is obtained by printing three or more consecutive elements with equal values with a line containing $x * k$, where x is the value of those elements and k is the number of consecutive equal elements.

Examples

```
>>> compress_sequence([-1,1,1,1,7,7,7,7,5,5,1,1,4,1])
-1
1 * 3
7 * 4
5
5
1
1
4
1
```

Maximal Increasing Subsequence

Write a function `maximal_increasing_subsequence(A)` that returns the maximal length of any strictly increasing sequence of contiguous elements of A. For $A = -1, 1, 7, 5, -2, 1, 2, 7, 7, 5, 0, 1, 3, 4, 1$, the result is 4, since there is at least one increasing subsequence of length 4 (e.g., $-2, 1, 2, 7$) but there is no increasing subsequence of length 5.

Examples

```
>>> maximal_increasing_subsequence([-1,1,7,5,-2,1,2,7,7,5,5,1,1
4
```

Partition Even/Odd

Write a function `partition_even_odd(A)` that takes an array of integers A and sorts the elements of A so that all the even elements precede all the odd elements. The function must sort A *in-place.* This means that it must operate directly on A just by swapping elements, without using an additional array.

Examples

```
>>> A = [-1,1,7,5,-2,1,2,7,7,5,5,1,1,4,1]
>>> partition_even_odd(A)
>>> print(A)
[-2,2,4,-1,1,7,5,1,7,7,5,5,1,1,1]
```

Notice that in general the solution is not unique. For example, the following result would be equally correct:

```
>>> A = [-1,1,7,5,-2,1,2,7,7,5,5,1,1,4,1]
>>> partition_even_odd(A)
>>> print(A)
[2,4,-2,5,1,5,1,7,7,7,-1,1,1,5,1]
```

Like or dislike

Write a function `like_or_dislike(A)` that takes a sequence of ratings expressed with the strings `'like'` and `'dislike'`. The function must print `'like'` if the most prevalent rating is `'like'`, or `'dislike'` if the most prevalent rating is `'dislike'`, or `'undecided'` otherwise.

Examples

```
>>> like_or_dislike(['like'])
like
>>> like_or_dislike(['dislike','like','dislike'])
dislike
>>> like_or_dislike(['like','dislike'])
undecided
>>> like_or_dislike([])
undecided
```

Stops and Inversions

A car moves along a road. For simplicity you may assume that the road is perfectly linear and that the position of the car is given as the distance from a certain reference point on the road. Write a function `stops_and_inversions(P)` that, given an array P of time-ordered positions of the car, prints the number of times the car stopped and the the number of times that the car inverted the direction of motion. $P = p_1, p_2, p_3, \ldots$ is time ordered in the sense that the car is at position p_1 *before* it is at position p_2, it is at p_2 before it is at p_3, and so on.

Examples

```
>>> stops_and_inversions([])
stops: 0
inversions: 0
>>> stops_and_inversions([5,5,5])
stops: 0
inversions: 0
>>> stops_and_inversions([5,5,5,6,7,8])
```

```
stops: 0
inversions: 0
>>> stops_and_inversions([5,5,5,6,7,8,8])
stops: 1
inversions: 0
>>> stops_and_inversions([1,5,10,12,12,12,15,20,24])
stops: 1
inversions: 0
>>> stops_and_inversions([1,5,10,12,12,11,8,7])
stops: 1
inversions: 1
>>> stops_and_inversions([1,5,10,12,12,11,8,7,9,11,20,30])
stops: 1
inversions: 2
>>> stops_and_inversions([1,5,10,12,12,11,8,7,9,11,20,30,30,30,
stops: 3
inversions: 2
```

Minimal Bounding Rectangle

Consider a sequence of n points $p_1, p_2, \ldots, p_n$ in a plane identified by their Cartesian coordinates. The coordinates are given in the form of two arrays X and Y both containing of n numbers, such that point p_i has coordinates `X[i]` and `Y[i]`.

Write a function called `minimal_bounding_rectangle_area(X,Y)` that, given two arrays of coordinates X and Y representing n points, returns the area of the smallest axis-aligned rectangle that covers all the n points. An axis-aligned rectangle is such that its sides are parallel to the X or Y axis.

Examples

```
>>> minimal_bounding_rectangle_area([3,2,10,4],[5,5,3,2])
24
>>> minimal_bounding_rectangle_area([2],[89])
0
minimal_bounding_rectangle_area([-8,8,9,-9,4,-4,4,-9,7],[10,1,-
306
```

Distance Between Points

Write a function called `distance_between_points(X,Y,d)` that, given two arrays of coordinates X and Y representing n points, and a number d, return `True` if and only if two of the given points are at a distance d from each other, or `False` otherwise.

Examples

```
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],5)
True
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],1)
True
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],6)
False
>>> distance_between_points([3,2,1,10,5,4],[5,5,4,3,1,2],4)
False
```

Find a Square

Write a function called `find_a_square(X,Y)` that, given two arrays of coordinates X and Y representing n points, and a number d, return `True` if and only if four of those points are the vertices of a square, or `False` otherwise.

Examples

```
>>> find_a_square([0,1,2,3],[3,1,4,2])
True
>>> find_a_square([0,1,2,1],[3,1,3,5])
False
```

Most Common Digit

Write a function called `most_common_digit(n)` that takes an integer n, and returns the most common digit in the decimal representation of n. If there are two or more most common digits, the function must return the smallest one.

Examples

```
>>> most_common_digit(1969)
9
>>> most_common_digit(44272)
2
>>> most_common_digit(362237715388253317 23)
3
>>> most_common_digit(0)
0
```

Sums of Squares

Write a function called `sums_of_squares(n)` that prints all the pairs of squares that sum to n. That is, `sums_of_squares(n)` prints a line a^2 + b^2 for all the distinct pairs $a \leq b$ such that $a^2 + b^2 = n$.

Examples

```
>>> sums_of_squares(25)
9 + 16
>>> sums_of_squares(21)
>>> sums_of_squares(125)
4 + 121
25 + 100
>>> sums_of_squares(178)
9 + 169
>>> sums_of_squares(179)
>>> sums_of_squares(17993241)
176400 + 17816841
```