# **Diamond Shaped Array**

**Task:** Create a function that returns an array that expands by 1 from 1 to the value of the input, and then reduces back to 1. Items in the lists will be the same as the length of the lists.

**Examples:**

diamondArrays(1) ➞ [[1]]

diamondArrays(2) ➞ [[1], [2, 2], [1]]

diamondArrays(5) ➞ [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4], [5, 5, 5, 5, 5], [4, 4, 4, 4], [3, 3, 3], [2, 2], [1]]

# **Multiplication Tables**

**Task:** create an N x N multiplication table, of size **n** provided by a parameter.

For example, when **n** is 5, the multiplication table is:

1, 2, 3, 4, 5

2, 4, 6, 8, 10

3, 6, 9, 12, 15

4, 8, 12, 16, 20

5, 10, 15, 20, 25

Output:

**[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]**

**Examples:**

multiplicationTable(1) ➞ [[1]]

multiplicationTable(3) ➞ [[1, 2, 3], [2, 4, 6], [3, 6, 9]]

# **Solve a Linear Equation**

**Task:** The function is given a string representing a linear equation with one unknown. Find the value of x that is a valid solution of the equation. Return the string x=val. Three outcomes are possible:

* x\_val is an integer or a float (round x\_val to 2 decimals)
* "Infinite solutions" for situations 0\*x=0
* "No solution" for situations 0\*x=num, (num != 0)

**Examples:**

FindX("4x-7=x+11") ➞ "x=6"

FindX("3x=2x+x") ➞ "Infinite solutions"

FindX("3x=3x+2") ➞ "No solution"

**Notes:** All input numbers should only have at most 3 digits.

# **String Cycling**

**Task:** Given two strings, repeatedly cycle through all of the letters in the first string until it is the same length as the second string.

**Examples:**

stringCycling("abc", "hello") ➞ "abcab"

stringCycling("programming", "edabit") ➞ "progra"

stringCycling("ha", "good morning") ➞ "hahahahahaha"

# **Mini Peaks**

**Task:** Write a function that returns all the elements in an array that are strictly greater than their adjacent left and right neighbors. If no such numbers exist, return an empty array. Do not count boundary numbers, since they only have one left/right neighbor.

**Examples:**

miniPeaks([4, 5, 2, 1, 4, 9, 7, 2]) ➞ [5, 9]

miniPeaks([1, 2, 1, 1, 3, 2, 5, 4, 4]) ➞ [2, 3, 5]

miniPeaks([1, 2, 3, 4, 5, 6]) ➞ []

# **Longest Common String**

**Task:** Given two strings, write a program that efficiently finds the longest common substring. If there are no common substrings, return an empty string.

**Example:**

longestCommonString(“Secretary”,”Intersection”) ➞ “sect”

longestCommonString(“Polymorphism”,”Abundant”) ➞ “”

# **Prime Numbers**

**Task:** Create a function that returns true if a number is prime, and false otherwise. A prime number is any positive integer that is evenly divisible by only two divisors: 1 and itself. 1 is not considered a prime number.

# **Interprime Numbers**

An interprime number is a composite number which is equidistant from two consecutive primes. For example, the interprime 6 is 1 point after 5, a prime, and 1 point before the next prime, 7. Another interprime is 93, which lies midway between primes 89 and 97.

**Task:** Create a function that takes a number **n** as input. If **n** is an interprime number, return an array containing the two consecutive primes between which it lies. If it isn't, return an empty array.

**Examples:**

interprime(6) ➞ [5, 7]

interprime(9) ➞ [7, 11]

interprime(8) ➞ []

**Notes:** Interprimes cannot be prime themselves (otherwise the primes would not have been consecutive).

# **Missing Alphabets**

**Task:** Create a function that takes a string containing only letters from a to z and returns the missing letter(s) in alphabetical order a-z.

**Examples:**

MissingAlphabets("abcdefghijklmnopqrstuvwxy") ➞ "z"

// "z" is missing.

MissingAlphabets("edabit") ➞ "cfghjklmnopqrsuvwxyz"

**Note:** If the string contains all letters from a-z, return an empty string ""

# **Triangle Words**

The nth term of the sequence of triangle numbers is given by, t(n) = ½n(n+1); so the first ten triangle numbers are: **1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...**

By converting each letter in a word to a number corresponding to its alphabetical position and adding these values we form a word value. For example, the word value for SKY is 19 + 11 + 25 = 55 = t(10). If the word value is a triangle number then we shall call the word a triangle word.

**Task:** Write a function that will take in a string and return whether the word is a triangle word or not.

# **Array of Multiples**

**Task:** Create a function that takes two numbers as arguments (num, length) and returns an array of multiples of num until the array length reaches length.

**Examples:**

arrayOfMultiples(7, 5) ➞ [7, 14, 21, 28, 35]

arrayOfMultiples(12, 10) ➞ [12, 24, 36, 48, 60, 72, 84, 96, 108, 120]

arrayOfMultiples(17, 6) ➞ [17, 34, 51, 68, 85, 102]

*Note that “num” is also included in the result*

# **Positive Count / Negative Sum**

**Task:** Create a function that takes in an array of ints and returns the number of positive numbers and the sum of the negative numbers, respectively.

**Examples:**

countPosSumNeg([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -11, -12, -13, -14, -15]) ➞ [10, -65]

countPosSumNeg([92, 6, 73, -77, 81, -90, 99, 8, -85, 34]) ➞ [7, -252]

countPosSumNeg([91, -4, 80, -73, -28]) ➞ [2, -105]

countPosSumNeg([]) ➞ []

**Notes:**

* If given an empty array, return an empty array: []
* 0 is not positive

# **Expanded Form of a Number**

**Task:** Create a function that expands a number into a string as shown below:

25 ➞ "20 + 5"

70701 ➞ "70000 + 700 + 1"

685 ➞ "600 + 80 + 5"

**Examples:**

expandedForm(70304) ➞ "70000 + 300 + 4"

expandedForm(1037903) ➞ "1000000 + 30000 + 7000 + 900 + 3"

expandedForm(802539) ➞ "800000 + 2000 + 500 + 30 + 9"

# **Consecutive Numbers**

**Task:** Create a function that determines whether elements in an array can be re-arranged to form a consecutive list of numbers where each number appears *exactly once*.

**Examples:**

ConsecNum([5, 1, 4, 3, 2]) ➞ true

// Can be re-arranged to form [1, 2, 3, 4, 5]

ConsecNum ([5, 1, 4, 3, 2, 8]) ➞ false

ConsecNum ([5, 6, 7, 8, 9, 9]) ➞ false

// 9 appears twice

# **Parabolic Javelin Throw**

**Task:** Write a function that takes initial speed (v in m/s) and throw angle (a in degrees) and returns the maximum height and maximum range reached by javelin as a string. Assume there is no air resistance.

**Examples:**

javelinThrow(36.7, 45) ➞ "Ymax=34m, Xmax=137m"

javelinThrow(51.3, 20) ➞ "Ymax=16m, Xmax=172m"

javelinThrow(100.1, 89) ➞ "Ymax=511m, Xmax=36m"

**Notes:**

* Javelin starts moving at h=0m.
* Gravitational acceleration is g=9.81 m/s^2.
* All results should be rounded to the nearest whole number.
* When an object in projectile motion reaches its maximum height, vertical velocity = 0

# **Unique Characters**

**Task:** Given two strings, create a function that returns the total number of unique characters from the combined string.

**Examples:**

countUnique("apple", "play") ➞ 5

// "appleplay" has 5 unique characters:

// "a", "e", "l", "p", "y"

countUnique("sore", "zebra") ➞ 7

// "sorezebra" has 7 unique characters:

// "a", "b", "e", "o", "r", "s", "z"

countUnique("a", "soup") ➞ 5

**Notes:** All input should be lower case

# **Quadratic Equation**

**Task:** Create a function to find only the root value of **x** in any quadratic equation ax^2 + bx + c. The function will take three arguments:

* **a** as the coefficient of **x^2**
* **b** as the coefficient of **x**
* **c** as the constant term

**Examples:**

quadraticEquation(1, -12, -28) ➞ 14

quadraticEquation(2, -7, 3) ➞ 3

quadraticEquation(1, 2, -3) ➞ 1

**Notes:**

* Quadratic equation is always guaranteed to have a root.
* Calculate only the root that sums the square root of the discriminant, not the one that subtracts it.
* Round the value / return only integer value.

# **How Many Solutions Does This Quadratic Have?**

A quadratic equation a x² + b x + c = 0 has either 0, 1, or 2 distinct solutions for real values of **x**.

**Task:** Given **a**, **b** and **c**, you should return the number of solutions to the equation.

**Examples:**

solutions(1, 0, -1) ➞ 2

// x² - 1 = 0 has two solutions (x = 1 and x = -1).

solutions(1, 0, 0) ➞ 1

// x² = 0 has one solution (x = 0).

solutions(1, 0, 1) ➞ 0

// x² + 1 = 0 has no real solutions.

**Notes:**

* You do not have to calculate the solutions, just return how many there are.
* “**a**” will always be non-zero.

# **Palindromic Substring**

**Task:** Write a function that finds the longest palindromic substring of a given string, if there is none, return an empty string

**Example:** palindromicSubstring(“different”) ➞ “ere”

# **Vertex of a Quadratic**

Every quadratic curve y = a x² + b x + c has a vertex point: the turning point where the curve stops heading down and starts going up.

**Task:** Given the values **a**, **b** and **c**, where **a** is never 0, you need to return the coordinates of the vertex. Return your answers rounded to 2 decimal places (a double).

**Examples:**

FindVertex(1, 0, 25) ➞ [0, 25]

// The vertex of y=x²+25 is at (0, 25).

FindVertex(-1, 0, 25) ➞ [0, 25]

// The vertex of y=-x²+25 is at (0, 25).

FindVertex(1, 10, 4) ➞ [-5, -21]

// The vertex of y=x²+10x+4 is at (-5, -21).

# **Sum of Quadratic Roots**

**Task:** Given **a**, **b** and **c**, find the roots of the equation ax^2 +bx +c and then add them together.

**Examples:**

findRootsSum(2, 4, -6) ➞ -2.00

findRootsSum(3, 4, -3) ➞ -1.33

findRootsSum(4, 3, -8) ➞ -0.75

**Notes:**

* Round your answer to two decimal places.
* If there is only one real root return 1.
* If there are no real roots, return 0.

# **Distance to Nearest Vowel**

**Task:** Write a function that takes in a string and for each character, returns the distance to the nearest vowel in the string. If the character is a vowel itself, return **0**.

**Examples:**

DistanceToNearestVowel("aaaaa") ➞ [0, 0, 0, 0, 0]

DistanceToNearestVowel("babbb") ➞ [1, 0, 1, 2, 3]

DistanceToNearestVowel("abcdabcd") ➞ [0, 1, 2, 1, 0, 1, 2, 3]

DistanceToNearestVowel("shopper") ➞ [2, 1, 0, 1, 1, 0, 1]

**Notes:**

* All input strings will contain at least one vowel.
* Strings will be lowercased.
* Vowels are: a, e, i, o, u.

# **Moran Numbers**

A Harshad number is a number which is divisible by the sum of its digits. For example, 132 is divisible by 6 (1+3+2).

A subset of the Harshad numbers are the Moran numbers. Moran numbers yield a prime when divided by the sum of their digits. For example, 133 divided by 7 (1+3+3) yields 19, a prime.

**Task:** Create a function that takes a number and returns "M" if the number is a Moran number, "H" if it is a (non-Moran) Harshad number, or "Neither".

**Examples:**

Moran(132) ➞ "H"

Moran(133) ➞ "M"

Moran(134) ➞ "Neither"

# **Nth Fibonacci Number**

**Task:** Create a function to return the Nth number in the Fibonacci sequence as a string.

**Examples:**

Fibonacci(10) ➞ "55"

Fibonacci(20) ➞ "6765"

Fibonacci(30) ➞ "832040"

Fibonacci(40) ➞ "102334155"

Fibonacci(50) ➞ "12586269025"

Fibonacci(60) ➞ "1548008755920"

**Notes:** This function is expected to calculate numbers greater than **UInt64.MaxValue** where **n** can be as large as but not greater than 200.

# **Binary Search**

**Task:** Create a function that finds a target number in a list of prime numbers. Implement a binary search algorithm in your function. The target number will be from 2 through 97. If the target is prime then return "yes" else return "no".

**Examples:**

int[] primes = { 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 }

IsPrime(primes, 3) ➞ "yes"

IsPrime(primes, 4) ➞ "no"

IsPrime(primes, 67) ➞ "yes"

# **Make it a Palindrome**

You are given a string “S” containing lowercase English letters. Your task is to calculate the minimum number of letters that need to be removed in order to make it possible to build a palindrome from the remaining letters. When building the palindrome, you can rearrange the remaining letters in any way.

**Task:** Write a function **int minPalindrome(string s)**, which, given a string of length “N”, returns the minimum number of letters that need to be removed.

**Examples:**

Given S = "ervervige", your function should return 2. After removing the letter "g" and one "e", we may create a word "reviver", which is a palindrome.

Given S = "aaabab", your function should return **O**. We may create a word "aabbaa", which is a palindrome and uses all of the letters.

Given S = "x", your function should return **O**. String "x" is a palindrome itself, so we do not have to delete any letters.

**Notes:**

* “S” contains only lowercase English letters.
* “N” is an integer within the range [1..200,000]

# **Is the String a Palindrome?**

**Task:** Given a word, create a function that checks whether it is a palindrome.

# **Is the Number a Palindrome?**

**Task:** Create a function that returns true if a number is a palindrome.

# **Integral of a Function**

**Task:** Create a function that takes numbers **b**, **m**, and **n** as arguments and returns the definite integral of the function **f(x)=(b+1)\*x^b** with respect to **x** from **x=m** **to x=n**, where **b**, **m**, and **n** are constants.

**Examples:**

Integral(0, 2, 5) ➞ 3

Integral(2, 4, 7) ➞ 279

Integral(5, 9, 3) ➞ -530712

**Notes:**

* **^** in the context of this challenge means "to the power of", also known as the "exponent" operator.
* Assume that **b** will be an integer greater than or equal to 0.
* **m** and **n** can be any integer, both positive and negative.

# **Perfect Number**

**Task:** Create a function that tests whether or not an integer is a perfect number. A perfect number is a number that can be written as the sum of its factors, (equal to sum of its proper divisors) excluding the number itself.

For example, 6 is a perfect number, since 1 + 2 + 3 = 6, where 1, 2, and 3 are all factors of 6. Similarly, 28 is a perfect number, since 1 + 2 + 4 + 7 + 14 = 28.

# **Maximum Travel Distance**

**Task:** Write a function that takes **fuel** (liters), **fuelUsage** (liters/100km), **passengers**, **airCon** (boolean) and returns maximum distance that car can travel.

* **fuel** is the number of liters of fuel in the fuel tank.
* **fuelUsage** is basic fuel consumption per 100 km (with the driver inside only).
* Every additional passenger is increasing basic fuel consumption by 5%.
* If the air conditioner is ON (true), its increasing the total (not basic) fuel consumption by 10%.

**Examples:**

totalDistance(70.0, 7.0, 0, false) ➞ 1000.0

totalDistance(36.1, 8.6, 3, true) ➞ 331.8

totalDistance(55.5, 5.5, 5, false) ➞ 807.3

**Notes:**

* **fuel** and **fuelUsage** are always greater than 1.
* **passengers** are always greater or equal to 0.
* Round your answer to the nearest tenth.

# **Sort Positives, Keep Negatives**

**Task:** Write a function that sorts the positive numbers in ascending order, and keeps the negative numbers untouched.

**Examples:**

posNegSort([6, 3, -2, 5, -8, 2, -2]) ➞ [2, 3, -2, 5, -8, 6, -2]

posNegSort([6, 5, 4, -1, 3, 2, -1, 1]) ➞ [1, 2, 3, -1, 4, 5, -1, 6]

posNegSort([-5, -5, -5, -5, 7, -5]) ➞ [-5, -5, -5, -5, 7, -5]

**Notes:**

* If given an empty array, you should return an empty array.
* Constraint: zeros are not included in input

# **Valid Hex Code**

A hex code must begin with a pound key # and is exactly 6 characters in length. Each character must be a digit from 0-9 or an alphabetic character from A-F. All alphabetic characters may be uppercase or lowercase.

**Task:** Create a function that determines whether a string is a valid hex code.

**Examples:**

IsValidHexCode("#CD5C5C") ➞ true

IsValidHexCode("#EAECEE") ➞ true

IsValidHexCode("#eaecee") ➞ true

IsValidHexCode("#CD5C58C") ➞ false

// Length exceeds 6

IsValidHexCode("#CD5C5Z") ➞ false

// Not all alphabetic characters in A-F

IsValidHexCode("#CD5C&C") ➞ false

// Contains unacceptable character

IsValidHexCode("CD5C5C") ➞ false

// Missing #

# **The Karaca's Encryption Algorithm**

**Task:** Make a function that encrypts a given input with these steps:

Step 1: Reverse the input

Step 2: Replace all vowels using the following chart:

**a => 0**

**e => 1**

**i => 2**

**o => 2**

**u => 3**

Step 3: Add "aca" to the end of the word

**Examples:**

Encrypt("banana") ➞ "0n0n0baca"

Encrypt("karaca") ➞ "0c0r0kaca"

Encrypt("burak") ➞ "k0r3baca"

Encrypt("alpaca") ➞ "0c0pl0aca"

**Notes:** All inputs are strings, no uppercases and all output must be strings.

# **Quartic Equation**

**Task:** Create a function that returns the number of (real) solutions of ax^4+bx^2+c=0. The function will take three arguments: **a** as the coefficient of x^4, **b** as the coefficient of x^2, and **c** as the constant term.

**Examples:**

quarticEquation(1, -5, 4) ➞ 4

quarticEquation(4, 3, -1) ➞ 2

quarticEquation(1, 10, 9) ➞ 0

**Notes:** Try substitution t=x^2.

# **Matrix Transpose**

In linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by A^T.

**Task:** Create a function that takes a 2D array (matrix A) and returns a 2D array (matrix A^T).

**Examples:**

transpose([

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]) ➞ [

[1, 4, 7],

[2, 5, 8],

[3, 6, 9]

]

transpose([

[1, 2],

[3, 4],

[5, 6]

]) ➞ [

[1, 3, 5],

[2, 4, 6]

]

# **Matrix Multiplication/Division**

**Task:** Write two functions, both taking in a 2d array of integers representing matrices. One will multiply the given matrices, the other will divide them. (Hint: division of matrix **A** by matrix **B** really means multiply matrix **A** by the inverse of matrix **B**)

# **Matrix Determinants**

In linear algebra, the determinant is a scalar value that can be computed from elements in a square matrix. The determinant is useful for solving linear equations, capturing how linear transformation change area or volume, and changing variables in integrals.

**Task:** Create a function that takes a square 2d array and returns the determinant

**Examples:**

determinant([[3]]) ➞ 3

determinant([

[1, 0]

[5, 4]

]) ➞ 4

determinant([

[4, 8, 6]

[2, 4, 3]

[6, 2, 1]

]) ➞ 0

**Notes:** All inputs must be *square* integer matrices

# **Matrix Rotate**

**Task:** Create a function to rotate a square two-dimensional matrix of integer elements **num** times, where if **num** is positive, the rotation is clockwise, and if not, counterclockwise.

**Examples:**

rotateTransform({

{2, 4},

{0, 0}

}, 1) ➞ {

{0, 2},

{0, 4}

}

# **Area of a Polygon**

**Task:** Given ordered coordinates of a polygon with **n** vertices, write a function to find the area of the polygon. Ordered means that the coordinates are given either in a clockwise manner from first vertex to last.

# **Character Permutations**

**Task:** Given a string, create a function that returns a list that contains all permutations using the characters in the string. Sort the list in alphabetical order.

**Examples:**

permutations("AB") ➞ {"AB", "BA"}

permutations("CD") ➞ {"CD", "DC"}

permutations("NOT") ➞ {"NOT", "NTO", "ONT", "OTN", "TNO", "TON"}

permutations("YAW") ➞ {"AWY", "AYW", "WAY", "WYA", "YAW", "YWA"}

# **Area of Overlapping Rectangles**

**Task:** Create a function that returns the area of the overlap between two rectangles. The function will receive two rectangles, each with the coordinates of the lower left corner followed by the width and the height: **int[] { x, y, width, height }**

**Examples:**

overlappingRectangles(new int[] { 2, 1, 3, 4 }, new int[] { 3, 2, 2, 5 }) ➞ 6

overlappingRectangles(new int[] { 2, -9, 11, 5 }, new int[] { 5, -11, 2, 9 }) ➞ 10

overlappingRectangles(new int[] { -8, -7, 4, 7 }, new int[] { -5, -9, 4, 7 }) ➞ 5

**Notes:** If the rectangles do not overlap, return -1

# **Maximum Occurrence**

**Task:** Given a string, write a function that returns the character with the highest frequency. If more than 1 character has the same highest frequency, return all those characters as a comma separated string. If there is no repetition in characters, return "No Repetition".

**Examples:**

maxOccur("Computer Science") ➞ "e"

maxOccur("Edabit") ➞ "No Repetition"

maxOccur("system admin") ➞ "m, s"

maxOccur("the quick brown fox jumps over the lazy dog") ➞ " "

**Notes:** Characters are case sensitive, so, for example, "C" and "c" are counted separately

# **Recursive Reverse**

**Task:** Given a string s, write a function that will reverse that string using recursion

# **Triangle Words**

The nth term of the sequence of triangle numbers is given by, t(n) = ½n(n+1); so the first ten triangle numbers are: **1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...**

By converting each letter in a word to a number corresponding to its alphabetical position and adding these values we form a word value. For example, the word value for SKY is 19 + 11 + 25 = 55 = t(10). If the word value is a triangle number then we shall call the word a triangle word.

**Task:** Write a function that will take in a string and return whether the word is a triangle word or not

# **Morse Code Translator**

**Task:** Write two functions, one that takes in a string of morse code and translates it into the English alphabet and one that does the opposite. For the input, each *word* in morse should be distinguished by a single space and each *letter* in morse should be distinguished by a forward slash, “/” with spaces on either side of it.

**Examples:**

morseToEnglish(“.... . .-.. .-.. --- / .-- --- .-. .-.. -..”) ➞ “Hello world”

englishToMorse(“I love code”) ➞ ” .. / .-.. --- ...- . / -.-. --- -.. .”

# **Point Within a Triangle**

**Task:** Create a function that takes four pairs. The first three are (x, y) coordinates of three corners of a triangle. Return true if the fourth tuple — the (x, y) coordinates of a test point — lies within the triangle, and false if it doesn't.

**Examples:**

withinTriangle([1, 4], [5, 6], [6, 1], [4, 5]) ➞ true

withinTriangle([1, 4], [5, 6], [6, 1], [3, 2]) ➞ false

withinTriangle([-6, 2], [-2, -2], [8, 4], [4, 2]) ➞ true

# **Missing Integer**

**Task:** Write a function that takes an array **A** of **N** integers and returns the smallest positive integer that does not occur in **A**

**Examples:**

* Given **A** = {1, 3, 6, 4, 1, 2}, the function should return **5**
* Given **A** = {1, 2, 3}, the function should return **4**
* Given **A** = {-1, -3}, the function should return **1**

**Assume that:**

* 1 <= **N** <= 100,000
* Each element of array **A** is in the range of [-1,000,000, 1,000,000]

# **Vowel Substrings**

**Task:** Given a string of lowercase English letters and an integer of the substring length, determine the substring of that length that contains the most vowels. Vowels are in the set {a, e, i, o, u}. If there is more than one substring with the maximum number of vowels, return the one that starts at the lowest index. If there are no vowels in the input string, return the string “Not found!”.

**Examples:**

vowelSubstring(“caberqiitefg”, 5) ➞ “erqii”

vowelSubstring(“aeiouia”, 3) ➞ “aei”

# **Forgotten Rolls**

You have just rolled a dice several times. The **N** roll results that you remember are described by an array **A**. However, there are **F** rolls whose results you have forgotten. The arithmetic mean of all of the roll results (the sum of all the roll results divided by the number of rolls) equals **M**.

**Task:** Write a function that, given an array **A** of length **N**, an integer **F** and an integer **M**, returns an array containing possible results of the missed rolls. The returned array should contain **F** integers from 1 to 6 (valid dice rolls). If such an array does not exist then the function should return [O].

**Examples:**

Given A = [3, 2, 4, 3], F = 2, M = 4 ➞ [6,6]

Given A = [1, 5, 6], F = 4, M = 3 ➞ [2, 1, 2, 4] or [6, 1, 1, 1] (there may be multiple solutions)

Given A = [1, 2, 3, 4], F = 4, M = 6 ➞ [0]

**Assume the following for this problem:**

* **M** is an integer within the range [1..6]
* Each element of array **A** is an integer within the range [1..6]
* **N** and **F** are integers within the range [1..100,000]

# **Array Summation**

**Task:** Write a function that takes two integer arrays as inputs. Your function will return TRUE if any two of the numbers in the first array, inputs, add up to any of the numbers in the second array, tests, and FALSE otherwise.

**Examples:**

arraySummation({-1, 8, 3}, {3, 7, 2}) ➞ TRUE

arraySummation({9, 6, 12}, {1, 2, 3}) ➞ FALSE

# **Clock Angle Problem**

**Task:** Given a time in “**hh:mm**” format, calculate the shorter angle between the hour and minute hand in an analog clock

**Examples:**

Input: 5:30 Output: 15°

Input: 9:00 Output: 90°

Input: 12:00 Output: 80

**Notes:**

* hh:60 should be considered as (hh+1):0.
* The idea is to consider the rate of change of the angle in degrees per minute. The hour hand of a 12-hour analog clock turns 360° in 12 hours, and the minute hand rotates through 360° in 60 minutes. So, we can calculate the angle in degrees of the hour hand minute hand separately and return their difference using the following formula:
  + Degree(hr) = Hx (360/12) + (Mx360)/(12x60) Degree(min) = Mx(360/60)
* Here, H is the hour, and M is the minutes past the hour. The angle should be in degrees and measured clockwise from the 12 o'clock position of the clock. If the angle is greater than 180° , take its difference with 360.

# **Left Rotation**

A left rotation operation on an array shifts each of the array's elements 1 unit to the left. For example, if 2 left rotations are performed on array *[1, 2, 3, 4, 5]*, then the array would become *[3, 4, 5, 1, 2]*. Note that the lowest index item moves to the highest index in a rotation. This is called a circular array.

**Task:** Given an array ***a*** of integers and a number ***r***, perform ***r*** left rotations on the array. Return the updated array to be printed as a single line of space-separated integers.

**Examples:**

**Given: a = {1, 2, 3, 4, 5}, n = 4**

**Output: {5, 1, 2, 3, 4}**

# **Sliding Blocks**

A Sliding Block Puzzle is a type of puzzle in which an **N**x**M** rectangular grid contains a single blank space and (**N**\***M**)-1numbered tiles. The goal of the puzzle is to arrange the puzzle such that the numbers are ordered left-to-right, top-to-bottom, with the empty space in the top left. This is achieved by repeatedly sliding adjacent tiles into the empty space and thus rearranging the puzzle.

**Task:** Implement a function that will take a puzzle (represented as a 2d array of integers, with 0 representing the blank space) and return the minimum number of moves to bring it to a solution. If the puzzle has no solution, you should return -1.

**Examples:**

An example of a move:

[1, 6, 3] [1, 6, 3]

[8, 7, 2] 🡪 [8, 0, 2]

[4, 0, 5] [4, 7, 5]

Solved puzzle:

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

**Notes:**

* Puzzles will be rectangular, and will contain each number from 0 to (M\*N) exactly once
* N and M will be at least two and no more than 5

# **Making Anagrams**

An anagram is defined as: a word or phrase made by transposing the letters of another word or phrase. In other words, an anagram is formed by taking the letters of one word/phrase and turning them into a different word/phrase. For example: the word silent can be formed by rearranging the letters of the word listen.

**Task:** Given two strings, **A** and **B**, that may or may not be of the same length, determine the minimum number of character deletions required to make them anagrams. Any characters can be deleted from either of the strings.

**Examples:**

Given: A = “abc”, B = “cde” Output: 4

Given: A = “cde”, B = “dcf” Output: 2