Moving Median

Top Rated User Solution

Moving Median

Have the function MovingMedian(arr) read the array of numbers stored in arr which will contain a sliding window size, N, as the first element in the array and the rest will be a list of numbers. Your program should return the Moving Median for each element based on the element and its N-1 predecessors, where N is the sliding window size. The final output should be a string with the moving median corresponding to each entry in the original array separated by commas.

Note that for the first few elements (until the window size is reached), the median is computed on a smaller number of entries. For example: if arr is [3, 1, 3, 5, 10, 6, 4, 3, 1] then your program should output "1,2,3,5,6,6,4,3"

Examples

Input: [5, 2, 4, 6]

Output: 2,3,4

Input: [3, 0, 0, -2, 0, 2, 0, -2]

Output: 0,0,0,0,0,0,0

```
function MovingMedian(arr) {
let buildArr = [];
let winLength = arr.shift();
let len = arr.length;

for (let i = 0; i < len; i++) {
  let firstIndex = Math.max(0, i - winLength + 1);
  let med = getMedian(arr.slice(firstIndex, i + 1));
  buildArr.push(med);
}
return buildArr.join(',');
}

let getMedian = (arr) => {
  arr.sort((val1, val2) => {return val1 - val2});
  let len = arr.length;
  if (len % 2) {
    return arr[Math.floor(len/2)];
  } else {
    return (arr[len/2 - 1] + arr[len/2]) / 2;
  }
}

// keep this function call here
MovingMedian(readline());
```

Group Totals

Have the function `GroupTotals(strArr)` read in the `strArr` parameter containing key:value pairs where the key is a string and the value is an integer. Your program should return a string with new key:value pairs separated by a comma such that each key appears only once with the total values summed up.

For example: if `strArr` is ["B:-1", "A:1", "B:3", "A:5"] then your program should return the string **A:6,B:2**.

Your final output string should return the keys in alphabetical order. Exclude keys that have a value of 0 after being summed up.

Examples

Input: ["X:-1", "Y:1", "X:-4", "B:3", "X:5"]
Output: B:3,Y:1

Input: ["Z:0", "A:-1"]
Output: A:-1

```
function GroupTotals(strArr) {
  let resObject = {};
  let parsingRegExp = /(\w+):(-?\d+)/;

  strArr.forEach(val => {
      const matches = val.match(parsingRegExp);
      const key = matches[1];
      const numVal = Number(matches[2]);

      if (resObject[key] || resObject[key] === 0) {
        resObject[key] = resObject[key] += numVal;
      } else {
        resObject[key] = numVal;
      }
  });

  return Object.keys(resObject)
    .sort()
    .map(val => {
      return resObject[val] ? (val + ":" + resObject[val]) : '';
    })
    .filter(val => {
      return val;
    })
    .join(',');
}

// keep this function call here
GroupTotals(readline());
```

String Changes

Have the function `StringChanges(str)` take the `str` parameter being passed, which will be a string containing letters from the alphabet, and return a new string based on the following rules. Whenever a capital M is encountered, duplicate the previous character (then remove the M), and whenever a capital N is encountered remove the next character from the string (then remove the N). All other characters in the string will be lowercase letters. For example: "abcNdgM" should return "abcgg". The final string will never be empty.

Examples

Input: "MrtyNNgMM"
Output: rtyggg

Input: "oMoMkkNrrN"
Output: ooookkr

```
function StringChanges(str) {
	const strArray = str.split('');
	let len = strArray.length;

	for (let i = 0; i < len; i++) {
		if (strArray[i] === 'M') {
			if (!i) {
				strArray.shift();
				len -= 1;
				i -= 1;
			} else {
				strArray.splice(i, 1, strArray[i - 1]);
			}
		} else if (strArray[i] === 'N') {
			if (i === len - 1) {
				strArray.pop();
			} else {
				strArray.splice(i, 2);
				len -= 2;
				i -= 2;
			}
		}
	}
	return strArray.join('');
}

// keep this function call here
StringChanges(readline());
```

FizzBuzz

Have the function `FizzBuzz(num)` take the `num` parameter being passed and return all the numbers from 1 to `num` separated by spaces, but replace every number that is divisible by 3 with the word "Fizz", replace every number that is divisible by 5 with the word "Buzz", and every number that is divisible by both 3 and 5 with the word "FizzBuzz". For example: if `num` is 16, then your program should return the string "1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16". The input will be within the range 1 - 50.

Examples

Input: 3
Output: 1 2 Fizz

Input: 8
Output: 1 2 Fizz 4 Buzz Fizz 7 8

```
function FizzBuzz(num) {
	const returnArray = [];
	for (let i = 1; i <= num; i++) {
		if (!(i % 15)) {
			returnArray.push('FizzBuzz');
		} else if (!(i % 5)) {
			returnArray.push('Buzz');
		} else if (!(i % 3)) {
			returnArray.push('Fizz');
		} else {
			returnArray.push(i);
		}
	}
	return returnArray.join(' ');
}

// keep this function call here
FizzBuzz(readline());
```

H Distance

Have the function `HDistance(strArr)` take the array of strings stored in `strArr`, which will only contain two strings of equal length and return the number of characters at each position that are different between them. For example: if `strArr` is ["house", "hours"] then your program should return **2**. The string will always be of equal length and will only contain lowercase characters from the alphabet and numbers.

Examples

Input: ["10011", "10100"]
Output: 3

Input: ["abcdef", "defabc"]
Output: 6

```
function HDistance(strArr) {

const string1 = strArr[0];
	const string2 = strArr[1];
	const len = string1.length;
	let count = 0;
```

```
        for (let i = 0; i < len; i++) {
                if (string1[i] !== string2[i]) {
                        count += 1;
                }
        }
        return count;
}

// keep this function call here
HDistance(readline());
```

Different Cases

Have the function `DifferentCases(str)` take the `str` parameter being passed and return it in upper camel case format where the first letter of each word is capitalized. The string will only contain letters and some combination of delimiter punctuation characters separating each word.

For example: if `str` is "Daniel LikeS-coding" then your program should return the string **DanielLikesCoding**.

Examples

Input: "cats AND*Dogs-are Awesome"
Output: CatsAndDogsAreAwesome

Input: "a b c d-e-f%g"
Output: ABCDEFG

```
function DifferentCases(str) {
     const charTest = /[a-zA-Z]/;
        let returnString = '';
        const len = str.length;
        const baseString = str.toLowerCase();

        for (let i = 0; i < len; i++) {
                if (i === len - 1 && !charTest.test(baseString[i])) {
                        return returnString;
                }
                if (i === 0 && charTest.test(baseString[i])) {
                        returnString += baseString[i].toUpperCase();
                        continue;
                }
                if (baseString[i - 1] && charTest.test(baseString[i])) {
                        if (!charTest.test(baseString[i - 1])) {
                                returnString += baseString[i].toUpperCase();
                        } else {
                                returnString += baseString[i];
                        }
                }
        }
        return returnString;
}
```

```
// keep this function call here
DifferentCases(readline());
```

Equivalent Keypresses

Have the function `EquivalentKeypresses(strArr)` read the array of strings stored in `strArr` which will contain 2 strings representing two comma separated lists of keypresses. Your goal is to return the string **true** if the keypresses produce the same printable string and the string **false** if they do not. A keypress can be either a printable character or a backspace represented by **-B**. You can produce a printable string from such a string of keypresses by having backspaces erase one preceding character.

For example: if `strArr` contains ["a,b,c,d", "a,b,c,c,-B,d"] the output should return **true** because those keypresses produce the same printable string. The array given will not be empty. The keypresses will only contain letters from the alphabet and backspaces.

Examples

Input: ["a,b,c,d", "a,b,c,d,-B,d"]
Output: true

Input: ["c,a,r,d", "c,a,-B,r,d"]
Output: false

```
function EquivalentKeypresses(strArr) {
  function reduce(str) {
    let result = "";
    str = str.split(",");
    str.forEach(function (item, index) {
      if (str[index + 1] !== "-B" && str[index] !== "-B") {
        result += item;
      }
    })
    return result;
  }
  // code goes here
  return reduce(strArr[0]) == reduce(strArr[1]);

}

// keep this function call here
console.log(EquivalentKeypresses(readline()));
```

Prime Time

Have the function `PrimeTime(num)` take the `num` parameter being passed and return the string **true** if the parameter is a prime number, otherwise return the string **false**. The range will be between 1 and 2^16.

Examples

Input: 19
Output: true

Input: 110
Output: false

```
function PrimeTime(num) {
    var hinge = Math.floor(Math.sqrt(num));
    var i = 2;
    var test = true;

    //hardcode correct answers for 1 and 2
    if (num === 1) {
        return false
    }
    else if (num === 2) {
        return true
    }
    //run a loop to see if the number has any integral factors (aside from 1)
    else {
        while (i <= hinge && test === true) {
            if (num % i !== 0) {
                i++;
            }
            else {
                test = false;
            }
        }
        return test
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
PrimeTime(readline());
```

Run Length

Have the function `RunLength(str)` take the `str` parameter being passed and return a compressed version of the string using the Run-length encoding algorithm. This algorithm works by taking the occurrence of each repeating character and outputting that number along with a single character of the repeating sequence. For example: "wwwggopp" would return **3w2g1o2p**. The string will not contain any numbers, punctuation, or symbols.

Examples

Input: "aabbcde"
Output: 2a2b1c1d1e

Input: "wwwbbbw"
Output: 3w3b1w

```
function RunLength(str) {
  strarr = str.split("");
  resarr = [];  //a place to put my results as they are determined;
  arrlen = strarr.length;
  count = 1;

  for (var i = 0; i <strarr.length; i++) {
    if (strarr[i] == strarr[i+1]) {
        count ++;
    }
        else {
        var entry = count + strarr[i];
        resarr.push(count + strarr[i]);
        count = 1;
        }
  }


  return resarr.join("");

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
RunLength(readline());
```

Prime Mover

Have the function `PrimeMover(num)` return the `num`th prime number. The range will be from 1 to 10^4. For example: if `num` is 16 the output should be **53** as 53 is the 16th prime number.

Examples

Input: 9

Output: 23

Input: 100

Output: 541

```
function PrimeMover(num) {
    var counter = 0;
    var testNum = 0;

    while (counter < num) {
        if (primeTest(testNum + 1)) {
            counter++;
        }
        testNum++;
    }
    return testNum

    function primeTest(int) {
        if (int === 1) {
            return false
        }
        else if (int === 2) {
            return true
        }
        else {
            var i = 2;
            var pivot = Math.ceil(Math.sqrt(int));
            while (i <= pivot) {
                if (int % i === 0) {
                    return false
                }
                i++;
            }
            return true
        }
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
PrimeMover(readline());
```

Palindrome Two

Have the function `PalindromeTwo(str)` take the **`str`** parameter being passed and return the string **true** if the parameter is a palindrome, (the string is the same forward as it is backward) otherwise return the string **false**. The parameter entered may have punctuation and symbols but they should not affect whether the string is in fact a palindrome. For example: "Anne, I vote more cars race Rome-to-Vienna" should return **true**.

Examples

Input: "Noel - sees Leon"

Output: true

Input: "A war at Tarawa!"

Output: true

```
function PalindromeTwo(str) {
var str = str.toLowerCase();
var strlen = str.length;
var arr = str.split("")
var newarr = [];
var x;
  for (var i = 0; i < strlen; i++) {
    if (arr[i].charCodeAt(0) > 96 && arr[i].charCodeAt(0)< 123) {
      newarr.push(arr[i]);
    }
  }

  for (var j = 0; j < newarr.length; j++) {
    if (newarr[j] != newarr[newarr.length - j - 1]) {
      x = false;
      break;
    }
    else {
    x = true;
    }
  }
return x;
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
PalindromeTwo(readline());
```

Division

Have the function `Division(num1,num2)` take both parameters being passed and return the **Greatest Common Factor**. That is, return the greatest number that evenly goes into both numbers with no remainder. For example: 12 and 16 both are divisible by 1, 2, and 4 so the output should be 4. The range for both parameters will be from 1 to 10^3.

Examples

Input: 7 & num2 = 3

Output: 1

Input: 36 & num2 = 54

Output: 18

```
function Division(num1,num2) {
var bignum = Math.max(num1, num2);
var smallnum = Math.min(num1, num2);

  for (var i = 1; i <= smallnum; i++) {
    if (bignum%i == 0 && smallnum%i == 0) {
     var maxi = i;
    }
  }
  return maxi;

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
Division(readline());
```

String Scramble

Have the function `StringScramble(str1,str2)` take both parameters being passed and return the string **true** if a portion of `str1` characters can be rearranged to match `str2`, otherwise return the string **false**. For example: if `str1` is "rkqodlw" and `str2` is "world" the output should return **true**. Punctuation and symbols will not be entered with the parameters.

Examples

Input: "cdore" & str2= "coder"

Output: true

Input: "h3llko" & str2 = "hello"

Output: false

```
function StringScramble(str1,str2) {

//first, knock the strings to lower case and put them in arrays for manipulation

arr1 = str1.toLowerCase().split("");
arr2 = str2.toLowerCase().split("");

//next, sort the arrays by alphabetical order
arr1.sort();
arr2.sort();

//loop through arr2, checking to see if each letter is in arr1, then removing it from
//arr2 and it and all previous letters from arr1.

var j = 0;
while (j < arr1.length) {
  if (arr2[0] == arr1[j]) {
    arr2.shift();
    arr1.splice(0,j+1);
    j = 0;
    }
  else {
 j++;
  }
}

if (arr2 == "") {
  return true;
}

else{
  return false;
}

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
StringScramble(readline());
```

Arith Geo II

Have the function `ArithGeoII(arr)` take the array of numbers stored in `arr` and return the string "**Arithmetic**" if the sequence follows an arithmetic pattern or return "**Geometric**" if it follows a geometric pattern. If the sequence doesn't follow either pattern return **-1**. An arithmetic sequence is one where the difference between each of the numbers is consistent, where as in a geometric sequence, each term after the first is multiplied by some constant or common ratio. Arithmetic example: [2, 4, 6, 8] and Geometric example: [2, 6, 18, 54]. Negative numbers may be entered as parameters, 0 will not be entered, and no array will contain all the same elements.

Examples

Input: [5,10,15]

Output: Arithmetic

Input: [2,4,16,24]

Output: -1

```
function ArithGeoII(arr) {
    var len = arr.length;
    //establish the relationship between two consecutive array elements
    var mathConstant = arr[1] - arr[0];
    var geoConstant = arr[1] / arr[0];

    //test the array to see if the difference between elements is the same between
each pair of consecutive elements.  If any pair fails, set flag to true and quit
    for (var i = 0; i < len - 1; i++) {
        if (arr[i + 1] - arr[i] !== mathConstant) {
            var mathTest = true;
            break;
        }
    }
    //if the above loop went all the way through, then return answer "Arithmetic."
If not, then loop through the array testing each pair.  If any pair fails, return -1
since it has failed both tests.  If it makes it all the way, return 'Geometric.'
    if (!mathTest) {
        return 'Arithmetic';
    }
    else {
        for (var j = 0; j < len - 1; j++) {
            if (arr[j + 1] / arr[j] !== geoConstant) {
                return -1;
            }
        }
        return 'Geometric';
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
ArithGeoII(readline());
```

Array Addition

Have the function `ArrayAddition(arr)` take the array of numbers stored in `arr` and return the string **true** if any combination of numbers in the array (excluding the largest number) can be added up to equal the **largest number** in the array, otherwise return the string **false**. For example: if `arr` contains [4, 6, 23, 10, 1, 3] the output should return **true** because 4 + 6 + 10 + 3 = 23. The array will not be empty, will not contain all the same elements, and may contain negative numbers.

Examples

Input: [5,7,16,1,2]

Output: false

Input: [3,5,-1,8,12]

Output: true

```
function ArrayAddition(arr) {
    var target;
    var addArr = arrayPrep(arr);
    var len = addArr.length;
    var permNum = Math.pow(2, len);

    for(var i = 0; i <= permNum; i++) {
        permStr = (i).toString(2);
        strlen = permStr.length;
        var counter = 0;
        for(var j = 0; j < strlen; j++) {
            if(permStr[j] === '1') {
                counter += addArr[j];
            }
        }
        if (counter === target) {
            return true
        }
    }
    return false

    function arrayPrep(arr2) {
        arr.sort(function(a, b){
            return a - b
        });
        target = arr2.pop()
        return arr2
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
ArrayAddition(readline());
```

Binary Converter

Have the function `BinaryConverter(str)` return the decimal form of the binary value. For example: if **101** is passed return **5**, or if **1000** is passed return **8**.

Examples

Input: "100101"

Output: 37

Input: "011"

Output: 3

```
function BinaryConverter(str) {
var counter = 0;
var numleng = str.length;

  for (var i = numleng-1; i >= 0; i--) {
    var j = numleng-1-i;
    var digt = parseInt(str.charAt(i));
    var addit = digt * Math.pow(2, j);
    counter = counter + addit;
  }

  return counter;

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
BinaryConverter(readline());
```

Letter Count

Have the function `LetterCount(str)` take the `str` parameter being passed and return the first word with the greatest number of repeated letters. For example: "Today, is the greatest day ever!" should return **greatest** because it has 2 e's (and 2 t's) and it comes before **ever** which also has 2 e's. If there are no words with repeating letters return **-1**. Words will be separated by spaces.

Examples

Input: "Hello apple pie"

Output: Hello

Input: "No words"

Output: -1

```
function LetterCount(str) {

  str = str.replace(/[^a-zA-zs]/g, "");
  str = str.toLowerCase();

 var wordarr = str.split(" ");

  var totescore = {};
      for (var i = 0; i < wordarr.length; i++) {  //for each word in string
        var scorearr = [];
        for (var j = 0; j < wordarr[i].length; j++) {  //for each letter in word
          var count = 0;
          var x = wordarr[i].charAt(j);
          for (var k = 0; k < wordarr[i].length; k++) {
           if  (wordarr[i][k] === x) {
                count++;
          }
        }
          scorearr.push(count);
     }
        scorearr.sort(function(a, b) {return (b-a)});
        totescore[wordarr[i]] = scorearr[0];
        var newarr = [];
      }
        for (var cnt in totescore) {
          newarr.push([cnt, totescore[cnt]]);
          newarr.sort(function(a, b) {return b[1] - a[1]});
      }
  if (newarr[0][1] === 1) {
    return -1;
  }
  else {
    return newarr[0][0];
  }

}


// keep this function call here
// to see how to enter arguments in JavaScript scroll down
LetterCount(readline());
```

Caesar Cipher

Have the function `CaesarCipher(str,num)` take the `str` parameter and perform a Caesar Cipher shift on it using the `num` parameter as the shifting number. A Caesar Cipher works by shifting each letter in the string N places down in the alphabet (in this case N will be `num`). Punctuation, spaces, and capitalization should remain intact. For example if the string is "Caesar Cipher" and `num` is **2** the output should be "Ecguct Ekrjgt".

Examples

Input: "Hello" & num = 4
Output: Lipps

Input: "abc" & num = 0
Output: abc

```
function CaesarCipher(str,num) {

  var arr = str.split("");

  for (var i = 0; i < str.length; i++) {


    var x = arr[i].charCodeAt(0);


    if (x > 64 && x < 91) {
      x = x - 65;
      x = x + num;
      x = x%26;
      x = x + 65;
      arr[i] = String.fromCharCode(x);

    }
    else if (x > 96 && x < 123) {
      x = x - 97;
      x = x + num;
      x = x%26;
      x = x + 97;
      arr[i] = String.fromCharCode(x);
    }

    else {

  }

  }
  var y = arr.join("");
  return y;

}
```

```
// keep this function call here
// to see how to enter arguments in JavaScript scroll down
CaesarCipher(readline());
```

Simple Mode

Have the function `SimpleMode(arr)` take the array of numbers stored in `arr` and return the number that appears most frequently (the mode). For example: if `arr` contains [10, 4, 5, 2, 4] the output should be **4**. If there is more than one mode return the one that appeared in the array first (**ie.** [5, 10, 10, 6, 5] should return **5** because it appeared first). If there is no mode return **-1**. The array will not be empty.

Examples

Input: [5,5,2,2,1]

Output: 5

Input: [3,4,1,6,10]

Output: -1

```
function SimpleMode(arr) {
  len = arr.length;
  var newobj = {};
  var testarr = [];
  for (var i = len-1; i >= 0; i--) {
    count = 0;
    for (var j = 0; j < len; j++) {
      if (arr[j] === arr[i]) {
      count++;
      }
    }
    newobj[arr[i]] = count;
  }
  for (x in newobj) {
    testarr.push([x, newobj[x]]);
  }
  testarr.sort(function(a, b) {return b[1] - a[1]});

  if (testarr[0][1] === 1) {
   return -1
  }
  else {
   return testarr[0][0];
  }


}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
SimpleMode(readline());
```

Consecutive

Have the function `Consecutive(arr)` take the array of integers stored in `arr` and return the minimum number of integers needed to make the contents of `arr` consecutive from the lowest number to the highest number. For example: If `arr` contains [4, 8, 6] then the output should be **2** because two numbers need to be added to the array (5 and 7) to make it a consecutive array of numbers from **4** to **8**. Negative numbers may be entered as parameters and no array will have less than 2 elements.

Examples

Input: [5,10,15]
Output: 8

Input: [-2,10,4]
Output: 10

```
function Consecutive(arr) {
    const onceArray = Array.from(new Set(arr))
             .sort((val1, val2) => val1 - val2);
       const finalIndex = onceArray.length - 1;
       return onceArray[finalIndex] - onceArray[0] - (onceArray.length - 1);
}

// keep this function call here
Consecutive(readline());
```

Formatted Division

Have the function `FormattedDivision(num1,num2)` take both parameters being passed, divide `num1` by `num2`, and return the result as a string with properly formatted commas and 4 significant digits after the decimal place. For example: if `num1` is **123456789** and `num2` is **10000** the output should be "**12,345.6789**". The output must contain a number in the one's place even if it is a zero.

Examples

Input: 2 & num2 = 3
Output: 0.6667

Input: 10 & num2 = 10
Output: 1.0000

```
function FormattedDivision(num1,num2) {
    //this gets most of the problem done in one simple step!
    var divisionResult = (num1 / num2).toFixed(4);
    //split the string into an array with two items: integer, decimal
    var numParts = divisionResult.split('.');
    var intArray = numParts[0].split('');
    var len = intArray.length;
    var intString;

    for (var i = len; i > 0; i--) {
        if (i < (len) && (len - i + 1) % 3 === 1) {
            intArray.splice(i, 0, ',');
            intString = intArray.join('');
        }
        else {
            intString = (intString)? intString : intArray.join('');
        }
    }
    return intString.concat('.', numParts[1]);
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
FormattedDivision(readline());
```

Counting Minutes

Have the function `CountingMinutes(str)` take the `str` parameter being passed which will be two times (each properly formatted with a colon and am or pm) separated by a hyphen and return the total number of minutes between the two times. The time will be in a 12 hour clock format. For example: if `str` is **9:00am-10:00am** then the output should be **60**. If `str` is **1:00pm-11:00am** the output should be **1320**.

Examples

Input: "12:30pm-12:00am"

Output: 690

Input: "1:23am-1:08am"

Output: 1425

```
function CountingMinutes(str) {
      const tester = /(\d+):(\d+)([pa]m)-(\d+):(\d+)([pa]m)/;
      const timeArray = str.match(tester);
      const time1 = timeArray[3] === 'am'
            ? (parseInt(timeArray[1], 10) * 60) + parseInt(timeArray[2], 10)
            : (parseInt(timeArray[1], 10) * 60) + parseInt(timeArray[2], 10) + 720;
      const time2 = timeArray[6] === 'am'
            ? (parseInt(timeArray[4], 10) * 60) + parseInt(timeArray[5], 10)
            : (parseInt(timeArray[4], 10) * 60) + parseInt(timeArray[5], 10) + 720;
      return ((time2 - time1) + 1440) % 1440;
}

// keep this function call here
CountingMinutes(readline());
```

Permutation Step

Have the function `PermutationStep(num)` take the `num` parameter being passed and return the next number greater than `num` using the same digits. For example: if `num` is 123 return **132**, if it's 12453 return **12534**. If a number has no greater permutations, return -1 (**ie.** 999).

Examples

Input: 11121

Output: 11211

Input: 41352

Output: 41523

```
function PermutationStep(num) {
    var numArray = num.toString().split('').reverse();
    numArray = numArray.map(function(val) {
        return parseInt(val);
    })
    var test = true;
    var len = numArray.length;
    newArray = [];
    while(test) {
        if (!newArray[0]) {
            newArray[0] = numArray.shift();
        }
        else if (newArray.every(function(val) {
            return val <= numArray[0];
            })) {
            newArray.push(numArray.shift())
        }
        else {
            if (!numArray[0]) {
                return '-1';
            }
            test = false;
        }
    }
    newArray.sort(function(a, b) {return a - b});
    var numHolder = numArray.shift();

    for (var i = 0; i < newArray.length; i++) {
        if (newArray[i] > numHolder) {
            numArray.unshift(newArray[i]);
            newArray[i] = numHolder;
            break;
        }
    }
    newArray.sort(function(a, b) {return b - a});

    var resultArray = newArray.concat(numArray);

    return resultArray.reverse().join('');
}

PermutationStep(readline());
```

Prime Checker

Have the function `PrimeChecker(num)` take `num` and return **1** if any arrangement of `num` comes out to be a prime number, otherwise return **0**. For example: if `num` is 910, the output should be **1** because 910 can be arranged into 109 or 019, both of which are primes.

Examples

Input: 98

Output: 1

Input: 598

Output: 1

```
function PrimeChecker(num) {
    //the initialize function converts the number into an array of n! 2-item arrays,
where n is the number of digits in num.
    //The array has the form ['',['1', '2', '3']].

    workingArray = initialize(num);
    var arrayLen = workingArray.length;

    while (workingArray[0][1].length > 0){
    //permStep is the function that moves the digits into their spots in an ordered
manner, resulting in
    //an array of n! elements of the form ['123',[]].
        permStep(workingArray);
    }
    //this tidies up the array elements.
    workingArray = workingArray.map(function(val){
        return val[0];
    });
    //primeTest is a function to convert each array element into a string, then test
to see if it is a prime.
    for (var i = 0, WAlen = workingArray.length; i < WAlen; i++){
        if (primeTest(workingArray[i])){
            return 1;
        }
    }
    return 0

    function initialize(num) {
        var arr = num.toString().split('')
        var resArr = [];
        for (var i = 0, len = factorial(arr.length); i < len; i++) {
            resArr.push(['', arr]);
        }
        return resArr;
    }
    function factorial(num) {
        if (num <= 1) {
            return 1;
        }
        else {
            return num * factorial(num - 1)
        }
    }
    function permStep(arr) {
        var counter = 0;
        var len = arr[0][1].length;
        while (counter < arrayLen) {
            var targetArray = arr[counter][1];
            for (var i = 0; i < len; i++) {
                for (var j = 0; j < factorial(len - 1); j++){
                    var copyArray = targetArray.map(function(val){
                        return val;
                    });
                    var holder = copyArray.splice(i, 1);
                    arr[counter][0] = arr[counter][0].concat(holder[0]);
                    arr[counter][1] = copyArray;
                    counter++;
                }
            }
        }
        return arr;
    }
    function primeTest(stringNum) {
        stringNum = parseInt(stringNum);
        pivot = Math.ceil(Math.sqrt(stringNum));
        if (stringNum === 1) {
            return false;
        }
        if (stringNum === 2) {
            return true;
        }
        else {
            for (var i = 2; i <= pivot; i++) {
                if (stringNum % i === 0) {
                    return false;
                }
            }
            return true;
        }
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
PrimeChecker(readline());
```

Dash Insert II

Have the function `DashInsertII(str)` insert dashes ('-') between each two odd numbers and insert asterisks ('*') between each two even numbers in `str`. For example: if `str` is **4546793** the output should be **454*67-9-3**. Don't count zero as an odd or even number.

Examples

Input: 99946

Output: 9-9-94*6

Input: 56647304

Output: 56*6*47-304

```
function DashInsertII(num) {
       const resString = num.toString(10);
       return resString
              .replace(/([2468])(?=[2468])/g, '$1*')
              .replace(/([13579])(?=[13579])/g, '$1-');
}

// keep this function call here
DashInsertII(readline());
```

Swap II

Have the function `SwapII(str)` take the `str` parameter and swap the case of each character. Then, if a letter is between two numbers (without separation), switch the places of the two numbers. For example: if `str` is "6Hello4 -8World, 7 yes3" the output should be **4hELLO6 -8wORLD, 7 YES3**.

Examples

Input: "Hello -5LOL6"
Output: hELLO -6lol5

Input: "2S 6 du5d4e"
Output: 2s 6 DU4D5E

```
let helpers;

function SwapII(str) {
    const switchString = helpers.caseSwap(str);
      return switchString.replace(/(\d)([A-Za-z]+)(\d)/g, '$3$2$1');
}

helpers = {
      caseSwap(str) {
            return str
                  .split('')
                  .map((letter) => {
                        if (/[A-Z]/.test(letter)) {
                              return letter.toLowerCase();
                        }
                        return letter.toUpperCase();
                  })
                  .join('');
      }
};

// keep this function call here
SwapII(readline());
```

Number Search

Have the function `NumberSearch(str)` take the `str` parameter, search for all the numbers in the string, add them together, then return that final number divided by the total amount of letters in the string. For example: if `str` is "Hello6 9World 2, Nic8e D7ay!" the output should be **2**. First if you add up all the numbers, 6 + 9 + 2 + 8 + 7 you get 32. Then there are 17 letters in the string. 32 / 17 = 1.882, and the final answer should be rounded to the nearest whole number, so the answer is 2. Only single digit numbers separated by spaces will be used throughout the whole string (So this won't ever be the case: hello44444 world). Each string will also have at least one letter.

Examples

Input: "H3ello9-9"
Output: 4

Input: "One Number*1*"

Output: 0

```
function NumberSearch(str) {
    var matchArr = str.match(/d/g);
    if (matchArr) {
        var matchArr = matchArr.map(function(val) {
            return parseInt(val);
        });
        var sum = matchArr.reduce(function(post, pre){
            return pre + post
        });
    }
    else {
        var sum =  0;
    }

    var letterArr = str.match(/[a-zA-Z]/g);

    return Math.round(sum / letterArr.length);

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
NumberSearch(readline());
```

Triple Double

Have the function `TripleDouble(num1,num2)` take both parameters being passed, and return **1** if there is a straight triple of a number at any place in `num1` and also a straight double of the same number in `num2`. For example: if `num1` equals **451999277** and `num2` equals **41177722899**, then return **1** because in the first parameter you have the straight triple *999* and you have a straight double, *99*, of the same number in the second parameter. If this isn't the case, return **0**.

Examples

Input: 465555 & num2 = 5579

Output: 1

Input: 67844 & num2 = 66237

Output: 0

```
function TripleDouble(num1,num2) {
    var holdArr = []
    num1 = num1.toString();
    for (var i = 0; i <= 9; i++) {
        i = i.toString();
        var regEx = new RegExp (i+i+i);
        if (regEx.test(num1)){
            holdArr.push(i);
        }
    }
    if (holdArr === []) {
        console.log(holdArr)
        return 0;
    }
    else {
        for (var j = 0, len = holdArr.length; j < len; j++){
            var double = new RegExp (holdArr[j] + holdArr[j]);
            console.log(double);
            if (double.test(num2)) {
                return 1;
            }
        }
        return 0;
    }
    console.log(holdArr);
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
TripleDouble(readline());
```

Bracket Matcher

Have the function `BracketMatcher(str)` take the `str` parameter being passed and return **1** if the brackets are correctly matched and each one is accounted for. Otherwise return **0**. For example: if `str` is "(hello (world))", then the output should be **1**, but if `str` is "((hello (world))" the the output should be **0** because the brackets do not correctly match up. Only "(" and ")" will be used as brackets. If `str` contains no brackets return **1**.

Examples

Input: "(coder)(byte))"

Output: 0

Input: "(c(oder)) b(yte)"

Output: 1

```
function BracketMatcher(str) {
    var strArray = str.split('');
    var counter = 0;
    for (var i = 0, len = strArray.length; i < len; i++) {
        if (strArray[i] === "(") {
            counter++;
        }
        else if (strArray[i] === ")") {
            counter--;
        }
        if (counter < 0) {
            return 0;
        }
    }
    if (counter === 0) {
        return 1;
    }
    return 0;
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
BracketMatcher(readline());
```

String Reduction

Have the function `StringReduction(str)` take the **`str`** parameter being passed and return the smallest number you can get through the following reduction method. The method is: Only the letters a, b, and c will be given in **`str`** and you must take two different adjacent characters and replace it with the third. For example "ac" can be replaced with "b" but "aa" cannot be replaced with anything. This method is done repeatedly until the string cannot be further reduced, and the length of the resulting string is to be outputted. For example: if **`str`** is "cab", "ca" can be reduced to "b" and you get "bb" (you can also reduce it to "cc"). The reduction is done so the output should be **2**. If **`str`** is "bcab", "bc" reduces to "a", so you have "aab", then "ab" reduces to "c", and the final string "ac" is reduced to "b" so the output should be **1**.

Examples

Input: "abcabc"

Output: 2

Input: "cccc"

Output: 4

```
function StringReduction(str) {

  while (str.match(/(ab|ba|ac|ca|bc|cb)/) != undefined) {
  str = str.replace(/(ab|ba)/, "c");
  str = str.replace(/(bc|cb)/, "a");
  str = str.replace(/(ac|ca)/, "b");
  }

  // code goes here
  return str.length;

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
StringReduction(readline());
```

ThreeFive Multiples

Have the function `ThreeFiveMultiples(num)` return the sum of all the multiples of 3 and 5 that are below **`num`**. For example: if **`num`** is 10, the multiples of 3 and 5 that are below 10 are 3, 5, 6, and 9, and adding them up you get 23, so your program should return **23**. The integer being passed will be between 1 and 100.

Examples

Input: 6

Output: 8

Input: 1

Output: 0

```
function ThreeFiveMultiples(num) {
    var arr = [];
    for (var i = 0; i <= num - 1; i++) {
        if (i % 3 === 0 || (i % 5 === 0 && i % 3 !== 0)) {
            arr.push(i);
        }
    }
    return arr.reduce(function(hold, val) {
        return hold + val;
    });
}


// keep this function call here
// to see how to enter arguments in JavaScript scroll down
ThreeFiveMultiples(readline());
```

Binary Search Tree LCA

Have the function `BinarySearchTreeLCA(strArr)` take the array of strings stored in `strArr`, which will contain 3 elements: the first element will be a binary search tree with all unique values in a preorder traversal array, the second and third elements will be two different values, and your goal is to find the lowest common ancestor of these two values. For example: if `strArr` is ["[10, 5, 1, 7, 40, 50]", "1", "7"] then this tree looks like the following:

```
      10
     /  \
    5    40
   / \     \
  1   7     50
```

For the input above, your program should return **5** because that is the value of the node that is the LCA of the two nodes with values 1 and 7. You can assume the two nodes you are searching for in the tree will exist somewhere in the tree.

Examples

Input: ["[10, 5, 1, 7, 40, 50]", "5", "10"]
Output: 10

Input: ["[3, 2, 1, 12, 4, 5, 13]", "5", "13"]
Output: 12

```
function BinarySearchTreeLCA(strArr) {
    //take the first item in the argument array and turn it into an array of integers
    let nodeArray = strArr[0]
        .replace(/[[]]/g, '')
        .split(/,s/)
        .map(val => parseInt(val, 10));

    //take the other items in the argument array and convert into integers
    let num1 = parseInt(strArr[1], 10);
    let num2 = parseInt(strArr[2], 10);

    //find the positions of the given numbers in the number array
    let ind1 = nodeArray.findIndex(val => val === num1);
    let ind2 = nodeArray.findIndex(val => val === num2);

    //determine the farther of the two positions, we are not interested in elements
past that
    let rightEdge = Math.max(ind1, ind2);

    //see if there are any items to the left of rightEdge that split the two given
numbers, that will be the answer
    let result = nodeArray.filter((val, ind) => (val >= Math.min(num1, num2) && val
<= Math.max(num1, num2) && ind <= rightEdge));

    //if not any, then return the item that is farthest to the left
    if (result.length === 0) return ind1 < ind2 ? strArr[1] : strArr[2];

    //if there is, then return it as a string
    return result[0].toString();
}

// keep this function call here
BinarySearchTreeLCA(readline());
```

Coin Determiner

Have the function `CoinDeterminer(num)` take the input, which will be an integer ranging from 1 to 250, and return an integer output that will specify the **least number** of coins, that when added, equal the input integer. Coins are based on a system as follows: there are coins representing the integers 1, 5, 7, 9, and 11. So for example: if `num` is 16, then the output should be **2** because you can achieve the number 16 with the coins 9 and 7. If `num` is 25, then the output should be **3** because you can achieve 25 with either 11, 9, and 5 coins or with 9, 9, and 7 coins.

Examples

Input: 6

Output: 2

Input: 16

Output: 2

```
function CoinDeterminer(num) {

    var arr = [0, 1, 2, 3, 4, 1, 2, 1, 2, 1, 2, 1, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2]


    if (num <= 22) {
        return arr[num]
    }

    else {
        var turns = Math.floor((num - 11) / 22) * 2;
        var remain = num - (turns * 11);
        if (remain > 22) {
            turns++;
            remain -= 11;
        }
        return turns + arr[remain];
    }
}


// keep this function call here
// to see how to enter arguments in JavaScript scroll down
CoinDeterminer(readline());
```

Fibonacci Checker

Have the function `FibonacciChecker(num)` return the string **yes** if the number given is part of the Fibonacci sequence. This sequence is defined by: Fn = Fn-1 + Fn-2, which means to find Fn you add the previous two numbers up. The first two numbers are 0 and 1, then comes 1, 2, 3, 5 etc. If `num` is not in the Fibonacci sequence, return the string **no**.

Examples

Input: 34

Output: yes

Input: 54

Output: no

```
function FibonacciChecker(num){

var seed1 = 0;
var seed2 = 1;
var counter = 0;
    while (counter < num) {
        counter = seed1 + seed2;
        if (counter === num) {
            return "yes";
        }
        seed1 = seed2;
        seed2 = counter;
    }

return "no";
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
FibonacciChecker(readline());
```

Multiple Brackets

Have the function `MultipleBrackets(str)` take the `str` parameter being passed and return **1 #ofBrackets** if the brackets are correctly matched and each one is accounted for. Otherwise return **0**. For example: if `str` is "(hello [world])(!)", then the output should be **1 3** because all the brackets are matched and there are 3 pairs of brackets, but if `str` is "((hello [world])" the the output should be **0** because the brackets do not correctly match up. Only "(", ")", "[", and "]" will be used as brackets. If `str` contains no brackets return **1**.

Examples

Input: "(coder)[byte)]"

Output: 0

Input: "(c([od]er)) b(yt[e])"

Output: 1 5

```
function MultipleBrackets(str) {
    var regex = /[()[]]/;

    if (!regex.test(str)) return 1;

    var countParen = 0;
    var countBrack = 0;
    var countOpen = 0;
    for (var i = 0, len = str.length; i < len; i++) {
        switch(str.charAt(i)) {
            case '(':
                countOpen++;
                countParen++;
                break;
            case '[':
                countOpen++;
                countBrack++;
                break;
            case ')':
                countParen--;
                break;
            case ']':
                countBrack--;
                break;
        }
        if (countParen < 0 || countBrack < 0) return 0;
    }

    if (countParen === 0 && countBrack === 0) return ('1 ' + countOpen);

    else return 0;
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
MultipleBrackets(readline());
```

Most Free Time

Have the function `MostFreeTime(strArr)` read the `strArr` parameter being passed which will represent a full day and will be filled with events that span from time X to time Y in the day. The format of each event will be **hh:mmAM/PM-hh:mmAM/PM**. For example, `strArr` may be ["10:00AM-12:30PM","02:00PM-02:45PM","09:10AM-09:50AM"]. Your program will have to output the longest amount of free time available between the start of your first event and the end of your last event in the format: **hh:mm**. The start event should be the earliest event in the day and the latest event should be the latest event in the day. The output for the previous input would therefore be **01:30** (with the earliest event in the day starting at **09:10AM** and the latest event ending at **02:45PM**). The input will contain at least 3 events and the events may be out of order.

Examples

Input: ["12:15PM-02:00PM","09:00AM-10:00AM","10:30AM-12:00PM"]
Output: 00:30

Input: ["12:15PM-02:00PM","09:00AM-12:11PM","02:02PM-04:00PM"]
Output: 00:04

```
function MostFreeTime(strArr) {
      strArr = strArr.map(function(val) {
        return convert12to24(val);
    });

    strArr = strArr.map(function(val){
        return convertToMinutes(val);
    })

    strArr.sort(function(a, b){
        return parseInt(a.match(/d{3,4}/) - b.match(/d{3,4}/))
    })

    strArr = strArr.map(function(val){
        valArr = val.split('-');
        valArr = valArr.map(function(val){
            return parseInt(val);
        });
        return valArr;
    });
    var len = strArr.length;

    var counter = 0;
    for (var i = 0; i < len - 1; i++) {
        var time = strArr[i+1][0] - strArr[i][1];
        if (time > counter) {
            counter = time;
        }
    }

    return returnToTime(counter);

    function returnToTime(counter){
        var mins = (counter % 60).toString();
        var hrs = (Math.floor(counter / 60).toString());
        if (mins < 10) {
            mins = '0' + mins;
        }
        if (hrs < 10) {
            hrs = '0' + hrs;
        }
        return (hrs + ':' + mins);
    }

    function convertToMinutes(strVal) {
        tempArr = strVal.split('-');
        tempArr = tempArr.map(function(val){
            var hrs = parseInt(val.slice(0,2));
            var mins = parseInt(val.slice(3));
            return hrs * 60 + mins;
        });

        return tempArr.join('-');
    }
    function convert12to24(strVal) {
        var tempArr = strVal.split('-');
        tempArr = tempArr.map(function(val){
            if (/am/i.test(val)) {
                val = val.replace('12', '00');
                return val.slice(0, 5);
            }
            else {
                val = val.replace('12', '00');
                var hour = parseInt(val.slice(0, 2));
                val = val.slice(2);
                var newHour = (hour + 12).toString();
                val = newHour.concat(val)
                return val.slice(0, 5);
            }
        });
        return tempArr.join('-');
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
MostFreeTime(readline());
```

Overlapping Rectangles

Have the function `OverlappingRectangles(strArr)` read the `strArr` parameter being passed which will represent two rectangles on a Cartesian coordinate plane and will contain 8 coordinates with the first 4 making up rectangle 1 and the last 4 making up rectange 2. It will be in the following format: ["(0,0),(2,2),(2,0),(0,2),(1,0),(1,2),(6,0),(6,2)"] Your program should determine the area of the space where the two rectangles overlap, and then output the number of times this overlapping region can fit into the **first rectangle**. For the above example, the overlapping region makes up a rectangle of area 2, and the first rectangle (the first 4 coordinates) makes up a rectangle of area 4, so your program should output **2**. The coordinates will all be integers. If there's no overlap between the two rectangles return 0.

Examples

Input: ["(0,0),(0,-2),(3,0),(3,-2),(2,-1),(3,-1),(2,3),(3,3)"]
Output: 6

Input: ["(0,0),(5,0),(0,2),(5,2),(2,1),(5,1),(2,-1),(5,-1)"]
Output: 3

JAVASCRIPT

```
let helpers;

const OverlappingRectangles = (strArr) => {
      const rectangles = helpers.rectangArrays(strArr);
      const rec1Points = helpers.recPoints(rectangles[0]);
      const rec2Points = helpers.recPoints(rectangles[1]);

      // created the values object to give names to the rectangle sides to make them
      // easier to visualize.
      const values = {
            rec1Top: rec1Points[1][1],
            rec1Right: rec1Points[1][0],
            rec1Bottom: rec1Points[0][1],
            rec1Left: rec1Points[0][0],
            rec2Top: rec2Points[1][1],
            rec2Right: rec2Points[1][0],
            rec2Bottom: rec2Points[0][1],
            rec2Left: rec2Points[0][0]
        };
        const rec1Area = (values.rec1Top - values.rec1Bottom) * (values.rec1Right -
values.rec1Left);

        /* test for overlap */
        if (values.rec2Bott >= values.rec1Top ||
            values.rec1Bott >= values.rec2Top ||
            values.rec1Left >= values.rec2Right ||
            values.rec2Left >= values.rec1Right) {
            return 0;
        }

        const overLapTop = Math.min(values.rec1Top, values.rec2Top);
        const overLapBottom = Math.max(values.rec1Bottom, values.rec2Bottom);
        const overLapRight = Math.min(values.rec1Right, values.rec2Right);
        const overLapLeft = Math.max(values.rec1Left, values.rec2Left);
        const overLapArea = (overLapTop - overLapBottom) * (overLapRight -
overLapLeft);
        return overLapArea === 0 ? 0 : Math.trunc(rec1Area / overLapArea);
};

helpers = {
        /* the function RectangArrays ta105816
        kes the input of this problem and returns a
    two-element array, each element being an array of the points in the first or
    second rectangle, respectively. */
        rectangArrays(strArr) {
            const str = strArr[0];
            let bigArray = str.split('),(');
            bigArray = bigArray.map((val) => {
                const cleanVal = val.replace(/[()]/g, '');
                const pointArray = cleanVal.split(',');
                pointArray[0] = parseInt(pointArray[0], 10);
                pointArray[1] = parseInt(pointArray[1], 10);
                return pointArray;
            });
            return [bigArray.splice(0, 4), bigArray];
        },
        /*
        the function recPoints takes an array of points on the cartesian grid (of an
    aligned rectangle) and returns an array of two points, representing the lower
    left corner and the upper right corner of the rectangle.
*/
        recPoints(arr) {
            const yVals = arr.map(val => val[1]);
            const xVals = arr.map(val => val[0]);
            const yMin = Math.min(...yVals);
            const yMax = Math.max(...yVals);
            const xMin = Math.min(...xVals);
            const xMax = Math.max(...xVals);
            return [[xMin, yMin], [xMax, yMax]];
      }
};

// keep this function call here
OverlappingRectangles(readline());
```

Look Say Sequence

Have the function `LookSaySequence(num)` take the `num` parameter being passed and return the next number in the sequence according to the following rule: to generate the next number in a sequence *read off* the digits of the given number, counting the number of digits in groups of the same digit. For example, the sequence beginning with 1 would be: 1, 11, 21, 1211, ... The 11 comes from there being "one 1" before it and the 21 comes from there being "two 1's" before it. So your program should return the next number in the sequence given `num`.

Examples

Input: 1211

Output: 111221

Input: 2466

Output: 121426

```
function LookSaySequence(num) {
    var numArr = prepFunc(num);
    var newArr = [];
    var storeArr = [];

    while (numArr.length > 0) {
        if (!newArr.length) {
            newArr.push(numArr.shift());
            if (!numArr.length) {
                storeArr.push('1');
                storeArr.push(newArr[0]);
                return parseInt(storeArr.join(''));
            }
        }
        else if (newArr[newArr.length - 1] === numArr[0]) {
            newArr.push(numArr.shift());
            if (!numArr.length) {
                storeArr.push(newArr.length.toString());
                storeArr.push(newArr[0]);
                return parseInt(storeArr.join(''));
            }
        }
        else {
            storeArr.push(newArr.length.toString());
            storeArr.push(newArr[0]);
            newArr = [];
        }
    }
        console.log(storeArr);
```

```
  // code goes here
  return num;
    function prepFunc(num) {
        var str = num.toString();
        var arr = str.split('');
        return arr;
    }
}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
LookSaySequence(readline());
```

Distinct List

Have the function `DistinctList(arr)` take the array of numbers stored in `arr` and determine the total number of duplicate entries. For example if the input is [1, 2, 2, 2, 3] then your program should output **2** because there are two duplicates of one of the elements.

Examples

Input: [0,-2,-2,5,5,5]

Output: 3

Input: [100,2,101,4]

Output: 0

```
function DistinctList(arr) {
      const strippedArr = new Set(arr);
      return arr.length - strippedArr.size;
}
DistinctList(readline());
```

Number Encoding

Have the function `NumberEncoding(str)` take the `str` parameter and encode the message according to the following rule: encode every letter into its corresponding numbered position in the alphabet. Symbols and spaces will also be used in the input. For example: if `str` is "af5c a#!" then your program should return **1653 1#!**.

Examples

Input: "hello 45"

Output: 85121215 45

Input: "jaj-a"

Output: 10110-1

```
function NumberEncoding(str) {
    str = str.toLowerCase();
    var arr = str.split("");
    var len = arr.length;
    for (var i = 0; i < len; i++) {
        if (/[a-z]/.test(arr[i])) {
            arr[i] = arr[i].replace(arr[i].charAt(0), (arr[i].charCodeAt(0) -
96).toString());
        }
    }

    str = arr.join("");

  // code goes here
  return str;

}

// keep this function call here
// to see how to enter arguments in JavaScript scroll down
NumberEncoding(readline());
```

Stock Picker

Have the function `StockPicker(arr)` take the array of numbers stored in `arr` which will contain integers that represent the amount in dollars that a single stock is worth, and return the maximum profit that could have been made by buying stock on day **x** and selling stock on day **y** where **y** > **x**. For example: if `arr` is [44, 30, 24, 32, 35, 30, 40, 38, 15] then your program should return **16** because at index 2 the stock was worth $24 and at index 6 the stock was then worth $40, so if you bought the stock at 24 and sold it at 40, you would have made a profit of $16, which is the maximum profit that could have been made with this list of stock prices.

If there is not profit that could have been made with the stock prices, then your program should return **-1**. For example: `arr` is [10, 9, 8, 2] then your program should return **-1**.

Examples

Input: [10,12,4,5,9]

Output: 5

Input: [14,20,4,12,5,11]

Output: 8

```
function StockPicker(arr) {
    var maxProfit = 0;
    var len = arr.length;

    while (arr.length > 1) {
        var start = arr.shift();
        var max = Math.max.apply(null, arr);
        var profit = max - start;
        if (profit > maxProfit) {
            maxProfit = profit;
        }
    }

    return maxProfit === 0 ? -1 : maxProfit;
}

// keep this function call here
StockPicker(readline());
```

Max Subarray

Have the function `MaxSubarray(arr)` take the array of numbers stored in `arr` and determine the largest sum that can be formed by any contiguous subarray in the array. For example, if `arr` is [-2, 5, -1, 7, -3] then your program should return **11** because the sum is formed by the subarray [5, -1, 7]. Adding any element before or after this subarray would make the sum smaller.

Examples

Input: [1, -2, 0, 3]

Output: 3

Input: [3, -1, -1, 4, 3, -1]

Output: 8

```
function MaxSubarray(arr) {
    //we use this number a few times
    let count = Math.max(...arr),
        indexArray = [];

    //take care of case with no positive numbers
    if (count <= 0) {
        return count;
    }

    //get a list (indexArray) of the indexes of all positive entries
    arr.forEach((val, ind) => {
        if (val > 0) {
            indexArray.push(ind)
        }
    });
    //we know that a maximum must have a positive number at each end of the subarray
(if there are multiple
    //positive numbers. Of course, the subArray could have length of one.  So just
compare all possible
    //subarrays.
    let leng = indexArray.length;
    for (let i = 0; i < leng; i++) {
        for (let j = i + 1; j < leng; j++) {
            let subArraySum = arr
                .slice(indexArray[i], indexArray[j] + 1)
                .reduce((val1, val2) => val1 + val2);
            //update count if the subArraySum is larger
            count = Math.max(count, subArraySum);
        }
    }
    return count;
}
```

```
// keep this function call here
MaxSubarray(readline());
```

Missing Digit

Have the function `MissingDigit(str)` take the `str` parameter, which will be a simple mathematical formula with three numbers, a single operator (+, -, *, or /) and an equal sign (=) and return the digit that completes the equation. In one of the numbers in the equation, there will be an **x** character, and your program should determine what digit is missing. For example, if `str` is "3x + 12 = 46" then your program should output **4**. The **x** character can appear in any of the three numbers and all three numbers will be greater than or equal to 0 and less than or equal to 1000000.

Examples

Input: "4 - 2 = x"
Output: 2

Input: "1x0 * 12 = 1200"
Output: 0

```
function MissingDigit(str) {
    for (let i = 0; i < 10; i++) {
          const newString = str.replace(/x/, i.toString()).replace(/=/, '===');
            if (eval(newString)) {
                  return i;
            }
      }
      return 'failed';
}

// keep this function call here
MissingDigit(readline());
```

K Unique Characters

Have the function `KUniqueCharacters(str)` take the `str` parameter being passed and find the longest substring that contains **k** unique characters, where **k** will be the first character from the string. The substring will start from the second position in the string because the first character will be the integer **k**. For example: if `str` is "2aabbacbaa" there are several substrings that all contain 2 unique characters, namely: ["aabba", "ac", "cb", "ba"], but your program should return "aabba" because it is the longest substring. If there are multiple longest substrings, then return the first substring encountered with the longest length. **k** will range from 1 to 6.

Examples

Input: "3aabacbebebe"
Output: cbebebe

Input: "2aabbcbbbadef"
Output: bbcbbb

```
function KUniqueCharacters(str) {
    const count = parseInt(str.slice(0,1), 10);
    const subjectString = str.slice(1);
    const len = subjectString.length;
    let maxCount = 0;

    for (let i = 0; i < len; i++) {
        counter = 0;
        let holder = [];
        let charArray = [];
        for (let j = i; j < len; j++) {
            let char = subjectString.charAt(j);
            if (charArray.some(val => val === char)) {
                holder.push(char);
                counter++;
                if (j === len - 1) {
                    if (counter > maxCount) {
                        result = Array.from (holder);
                    }
                    maxCount = Math.max(counter, maxCount);
                }
                continue;
            } else {
                if (charArray.length < count) {
                    counter++;
                    charArray.push(char);
                    holder.push(char)
                    continue;
                } else {
                    if (counter > maxCount) {
                        result = Array.from(holder);
                    }
                    maxCount = Math.max(counter, maxCount);
                    break;
                }
            }
        }
    }
    return result.join('');
}

// keep this function call here
KUniqueCharacters(readline());
```

Bitmap Holes

Have the function `BitmapHoles(strArr)` take the array of strings stored in **`strArr`**, which will be a 2D matrix of 0 and 1's, and determine how many holes, or contiguous regions of 0's, exist in the matrix. A contiguous region is one where there is a connected group of 0's going in one or more of four directions: up, down, left, or right. For example: if **`strArr`** is ["10111", "10101", "11101", "11111"], then this looks like the following matrix:

```
1 0 1 1 1
1 0 1 0 1
1 1 1 0 1
1 1 1 1 1
```

For the input above, your program should return **2** because there are two separate contiguous regions of 0's, which create "holes" in the matrix. You can assume the input will not be empty.

Examples

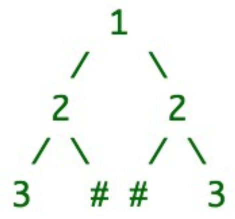
Input: ["01111", "01101", "00011", "11110"]
Output: 3

Input: ["1011", "0010"]
Output: 2

```
function BitmapHoles(strArr) {
    const thisObj = {};

    thisObj.fullArray = strArr.map(row => row.split(''));

    thisObj.fullArray.forEach((row, rowNum, arr) => {
        row.forEach((val, colNum) => {

            //can get rid of hotTest and just use crawler here.
            if (val === '1') {
            } else {
                arr[rowNum][colNum] = 'H';
                crawler([[rowNum,colNum]])
            }
        });
    });
    return thisObj.fullArray.reduce((row1, row2) => {
      return row1.concat(row2);
    }).filter(val => val === 'H').length;


    function crawler(pointsArray) {
        let newArray = [];
        pointsArray.forEach(point => {

            if(point[0] > 0 && thisObj.fullArray[point[0] - 1][point[1]] === '0') {
                thisObj.fullArray[point[0] - 1][point[1]] = '1';
                newArray.push([point[0] - 1, point[1]]);
            }

            if (point[0] < thisObj.fullArray.length - 1 && thisObj.fullArray[point[0]
+ 1][point[1]] === '0') {
                thisObj.fullArray[point[0] + 1][point[1]] = '1';
                newArray.push([point[0] + 1, point[1]]);
            }

            if (point[1] > 0 && thisObj.fullArray[point[0]][point[1] - 1] === '0') {
                thisObj.fullArray[point[0]][point[1] - 1] = '1';
                newArray.push([point[0], point[1] - 1]);
            }

            if (point[1] < thisObj.fullArray[0].length - 1 &&
thisObj.fullArray[point[0]][point[1] + 1] === '0') {
                thisObj.fullArray[point[0]][point[1] + 1] = '1';
                newArray.push([point[0], point[1] + 1]);
            }
        });

        if (newArray.length === 0) {
            return;
        }
        crawler(newArray);
    }
}


// keep this function call here
BitmapHoles(readline());
```

Symmetric Tree

Have the function `SymmetricTree(strArr)` take the array of strings stored in `strArr`, which will represent a binary tree, and determine if the tree is symmetric (a mirror image of itself). The array will be implemented similar to how a binary heap is implemented, except the tree may not be complete and NULL nodes on any level of the tree will be represented with a #. For example: if `strArr` is ["1", "2", "2", "3", "#", "#", "3"] then this tree looks like the following:



For the input above, your program should return the string **true** because the binary tree is symmetric.

Examples

Input: ["4", "3", "4"]

Output: false

Input: ["10", "2", "2", "#", "1", "1", "#"]

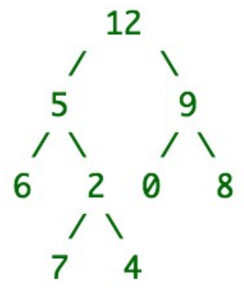
Output: true

```
function SymmetricTree(strArr) {
  let x = 1;
  let count = 0;
  while (strArr.length) {
    x++;
    if (x > 10) break;
    let length = Math.pow(2, count);
    let newArray = strArr.splice(0, length);
    let revArray = Array.from(newArray).reverse();
    if(!sameAs(newArray, revArray)) {
      return false;
    }
    count = newArray.filter(val => val !== '#').length;
  }
  return true;
}

function sameAs(arr1, arr2) {
  return arr1.every((val, ind) => val === arr2[ind]);
}

// keep this function call here
SymmetricTree(readline());
```

Binary Tree LCA

Have the function `BinaryTreeLCA(strArr)` take the array of strings stored in **`strArr`**, which will contain 3 elements: the first element will be a binary tree with all unique values in a format similar to how a binary heap is implemented with NULL nodes at any level represented with a #, the second and third elements will be two different values, and your goal is to find the lowest common ancestor of these two values.

For example: if **`strArr`** is ["[12, 5, 9, 6, 2, 0, 8, #, #, 7, 4, #, #, #, #]", "6", "4"] then this tree looks like the following:



For the input above, your program should return **5** because that is the value of the node that is the LCA of the two nodes with values 6 and 4. You can assume the two nodes you are searching for in the tree will exist somewhere in the tree.

Examples

Input: ["[5, 2, 6, 1, #, 8, #]", "2", "6"]
Output: 5

Input: ["[5, 2, 6, 1, #, 8, 12, #, #, #, #, #, #, 3, #]", "3", "12"]
Output: 12

```
function BinaryTreeLCA(strArr) {
    //first, convert the string representing the tree into an array of numbers
    let arrList = strArr[0]
        .replace(/[[]]/g, '')
        .split(/,s*/)
        .map(val => val !== '#' ? parseInt(val, 10) : "#");

    //convert the given numbers (strings) into numbers
    let num1 = parseInt(strArr[1], 10);
    let num2 = parseInt(strArr[2], 10);

    //get the indexes of the given numbers. This is really what we need
    let ind1 = Math.max(arrList.findIndex(val => val === num1) + 1,
arrList.findIndex(val => val === num2) + 1);
```

```
    let ind2 = Math.min(arrList.findIndex(val => val === num1) + 1,
arrList.findIndex(val => val === num2) + 1);

    //get the two numbers onto the same depth in the tree
    while (Math.trunc(Math.log2(ind1)) !== Math.trunc(Math.log2(ind2))) {
      ind1 = Math.trunc(ind1 / 2);
    }

    //find the common ancestor in the tree
    while (ind1 !== ind2) {
      ind1 = Math.trunc(ind1 / 2);
      ind2 = Math.trunc(ind2 / 2);
    }

    //return the number corresponding to the determined index
    return arrList[ind2 - 1].toString();


}

// keep this function call here
BinaryTreeLCA(readline());
```

LRU Cache

Have the function `LRUCache(strArr)` take the array of characters stored in **`strArr`**, which will contain characters ranging from A to Z in some arbitrary order, and determine what elements still remain in a virtual cache that can hold up to 5 elements with an LRU cache algorithm implemented. For example: if **`strArr`** is ["A", "B", "C", "D", "A", "E", "D", "Z"], then the following steps are taken:

(1) A does not exist in the cache, so access it and store it in the cache.
(2) B does not exist in the cache, so access it and store it in the cache as well. So far the cache contains: ["A", "B"].
(3) Same goes for C, so the cache is now: ["A", "B", "C"].
(4) Same goes for D, so the cache is now: ["A", "B", "C", "D"].
(5) Now A is accessed again, but it exists in the cache already so it is brought to the front: ["B", "C", "D", "A"].
(6) E does not exist in the cache, so access it and store it in the cache: ["B", "C", "D", "A", "E"].
(7) D is accessed again so it is brought to the front: ["B", "C", "A", "E", "D"].
(8) Z does not exist in the cache so add it to the front and remove the least recently used element: ["C", "A", "E", "D", "Z"].

Now the caching steps have been completed and your program should return the order of the cache with the elements joined into a string, separated by a hyphen. Therefore, for the example above your program should return **C-A-E-D-Z**.

Examples

Input: ["A", "B", "A", "C", "A", "B"]
Output: C-A-B

Input: ["A", "B", "C", "D", "E", "D", "Q", "Z", "C"]
Output: E-D-Q-Z-C

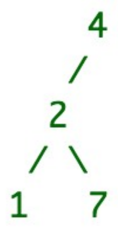
```
function LRUCache(strArr) {
    let cache = [];
    //very simple, go through the array given and operate on the cache
    strArr.forEach(val => {
        if (!cache.includes(val)) {
            //fun! One of the first times I have ever taken advantage of the fact
            //that push() returns the length of the array after the push, not the
            //array itself
            if (cache.push(val) > 5) {
                cache.shift();
            }
        } else {
            cache.splice(cache.findIndex(cacheVal => val === cacheVal), 1);
            cache.push(val);
        }
    });

    return cache.join('-');

}

// keep this function call here
LRUCache(readline());
```

medium

Discussion Solutions

Tree Constructor

Have the function `TreeConstructor(strArr)` take the array of strings stored in `strArr`, which will contain pairs of integers in the following format: **(i1,i2)**, where **i1** represents a child node in a tree and the second integer **i2** signifies that it is the parent of **i1**. For example: if `strArr` is ["(1,2)", "(2,4)", "(7,2)"], then this forms the following tree:



which you can see forms a proper binary tree. Your program should, in this case, return the string **true** because a valid binary tree can be formed. If a proper binary tree cannot be formed with the integer pairs, then return the string **false**. All of the integers within the tree will be unique, which means there can only be one node in the tree with the given integer value.

Examples

Input: ["(1,2)", "(2,4)", "(5,7)", "(7,2)", "(9,5)"]
Output: true

Input: ["(1,2)", "(3,2)", "(2,12)", "(5,2)"]
Output: false

```
//the following method throws out trees that fail any of three tests: i) is there
only one top node, ii) does any node have more
```

```
//than 2 children, or iii) are any node values repeated.  If it passes all of these
tests, then it should represent a binary tree.

function TreeConstructor(strArr) {
    //remove spaces from input (one of the tests had bad input)
    strArr = strArr.map(val => val.replace(/s/g, ''));

    let regExChildPattern = /((d+),d+)/
    let regExParentPattern = /(d+,(d+))/

    let children = strArr.map(val => regExChildPattern.exec(val)[1]);
    let parents = strArr.map(val => regExParentPattern.exec(val)[1]);

    //check to make sure all children are unique
    let childSet = new Set(children);
    if (children.length !== childSet.size) {
        return false;
    }

    //test whether any parent node has more than 2 children
    let parentObj = {};
    parents.forEach(val => {
        if (!parentObj[val]) {
            parentObj[val] = 1;
        } else {
            parentObj[val]++;
        }
    })
    for (let myKey in parentObj) {
        if (parentObj[myKey] > 2) return false;
    }

    //make certain there is one, and only one, top dog
    let uniqParents = Array.from(new Set(parents))

    let topDogs = uniqParents.filter(val => !children.includes(val));
    if (topDogs.length !== 1) return false;

    return true;

}


// keep this function call here
TreeConstructor(readline());
```

Array Min Jumps

Have the function `ArrayMinJumps(arr)` take the array of integers stored in `arr`, where each integer represents the maximum number of steps that can be made from that position, and determine the least amount of jumps that can be made to reach the end of the array. For example: if `arr` is [1, 5, 4, 6, 9, 3, 0, 0, 1, 3] then your program should output the number **3** because you can reach the end of the array from the beginning via the following steps: 1 -> 5 -> 9 -> END or
1 -> 5 -> 6 -> END. Both of these combinations produce a series of 3 steps. And as you can see, you don't always have to take the maximum number of jumps at a specific position, you can take less jumps even though the number is higher.

If it is not possible to reach the end of the array, return **-1**.

Examples

Input: [3, 4, 2, 1, 1, 100]
Output: 2

Input: [1, 3, 6, 8, 2, 7, 1, 2, 1, 2, 6, 1, 2, 1, 2]
Output: 4

```
function ArrayMinJumps(arr) {
	const newArr = Array.from(arr);
	const len = newArr.length;
	for (let i = len - 1; i >= 0; i--) {
		const toGo = len - (i + 1);
		const reach = newArr[i];
		if (toGo === 0) {
			newArr[i] = {
				ind: i,
				moves: 0
			};
		} else if (reach >= toGo) {
			newArr[i] = {
				ind: i,
				moves: 1
			};
		} else {
			const subArr = newArr.slice(i + 1);
			const subArrLen = subArr.length;
			const countHolder = [];
			for (let j = 0; j < subArrLen; j++) {
				if (typeof subArr[j] === 'object' && newArr[i] > j) {
					countHolder.push(subArr[j].moves);
				}
			}
			if (countHolder.length) {
				newArr[i] = {
					ind: i,
					moves: Math.min(...countHolder) + 1
				};
			}
		}
	}
	return typeof newArr[0].moves === 'undefined' ? -1 : newArr[0].moves;
}

// keep this function call here
ArrayMinJumps(readline());
```

Nearest Smaller Values

Have the function `NearestSmallerValues(arr)` take the array of integers stored in `arr`, and for each element in the list, search all the previous values for the nearest element that is smaller than the current element and create a new list from these numbers. If there is no element before a certain position that is smaller, input a **-1**. For example: if `arr` is [5, 2, 8, 3, 9, 12] then the nearest smaller values list is [-1, -1, 2, 2, 3, 9]. The logic is as follows:

For 5, there is no smaller previous value so the list so far is [-1]. For 2, there is also no smaller previous value, so the list is now [-1, -1]. For 8, the nearest smaller value is 2 so the list is now [-1, -1, 2]. For 3, the nearest smaller value is also 2, so the list is now [-1, -1, 2, 2]. This goes on to produce the answer above. Your program should take this final list and return the elements as a string separated by a space: **-1 -1 2 2 3 9**

Examples

Input: [5, 3, 1, 9, 7, 3, 4, 1]

Output: -1 -1 -1 1 1 1 3 1

Input: [2, 4, 5, 1, 7]

Output: -1 2 4 -1 1

```
function NearestSmallerValues(arr) {
       const resultsArray = [];
       arr.forEach((val, ind) => {
              const preArray = arr.slice(0, ind).reverse();
              resultsArray.push(preArray.find(item => item <= val) !== undefined ?
              preArray.find(item => item <= val) : -1);
       });
       return resultsArray.join(' ');
}
NearestSmallerValues(readline());
```

Matrix Spiral

Have the function `MatrixSpiral(strArr)` read the array of strings stored in `strArr` which will represent a 2D N matrix, and your program should return the elements after printing them in a clockwise, spiral order. You should return the newly formed list of elements as a string with the numbers separated by commas. For example: if `strArr` is "[1, 2, 3]", "[4, 5, 6]", "[7, 8, 9]" then this looks like the following 2D matrix:

```
1 2 3
4 5 6
7 8 9
```

So your program should return the elements of this matrix in a clockwise, spiral order which is: **1,2,3,6,9,8,7,4,5**

Examples

Input: ["[1, 2]", "[10, 14]"]

Output: 1,2,14,10

Input: ["[4, 5, 6, 5]", "[1, 1, 2, 2]", "[5, 4, 2, 9]"]

Output: 4,5,6,5,2,9,2,4,5,1,1,2

```
function MatrixSpiral(strArr) {
       //first, create an array of arrays of nubers [ [ '1', '2', '3' ], [ '4', '5',
'6' ], [ '7', '8', '9' ] ]
       strArr = strArr.map(val => {
              return val.replace(/[\[\]\s]/g, '').split(',');
       })
```

```
        //next, create a string to hold the values
        let res = [];
        let holder = [];
        //make a copy of the array, just in case;
        let arr = Array.from(strArr);

        let rightWall = (val) => {
              res.push(val.pop());
        };

        let leftWall = (val) => {
              holder.push(val.shift());

        };

        while(arr.length) {
              res.push(...arr.shift());
              if (!arr.length) {
                    break;
              }
              arr.forEach(rightWall);
              if (!arr.length) {
                    break;
              }
              res.push(...arr.pop().reverse());
              if (!arr.length) {
                    break;
              }
              arr.forEach(leftWall);
              res.push(holder.reverse());
              holder = [];
        }

        return res.join(',');
}

// keep this function call here
MatrixSpiral(readline());
```

Word Split

Have the function `WordSplit(strArr)` read the array of strings stored in `strArr`, which will contain 2 elements: the first element will be a sequence of characters, and the second element will be a long string of comma-separated words, in alphabetical order, that represents a dictionary of some arbitrary length. For example: `strArr` can be: ["hellocat", "apple,bat,cat,goodbye,hello,yellow,why"]. Your goal is to determine if the first element in the input can be split into two words, where both words exist in the dictionary that is provided in the second input. In this example, the first element can be split into two words: **hello** and **cat** because both of those words are in the dictionary.

Your program should return the two words that exist in the dictionary separated by a comma. So for the example above, your program should return **hello,cat**. There will only be one correct way to split the first element of characters into two words. If there is no way to split string into two words that exist in the dictionary, return the string **not possible**. The first element itself will never exist in the dictionary as a real word.

Examples

Input: ["baseball", "a,all,b,ball,bas,base,cat,code,d,e,quit,z"]

Output: base,ball

Input: ["abcgefd", "a,ab,abc,abcg,b,c,dog,e,efd,zzzz"]
Output: abcg,efd

```
function WordSplit(strArr) {
	let target = strArr[0];
	let parts = strArr[1].split(/,/);
	let starters = [];
	let enders = [];
	let res = '';
	parts.forEach(val => {
		regEx1 = new RegExp(`\^${val}`);
		regEx2 = new RegExp(`${val}\$`);
		if (regEx1.test(target)) {
			starters.push(val);
		}
		if (regEx2.test(target)) {
			enders.push(val);
		}
	});
	starters.forEach(start => {
		enders.forEach(end => {
			if (start + end === target) {
				res = `${start},${end}`;
			}
		})
	})
	return res || 'not possible';
}

// keep this function call here
WordSplit(readline());
```

Pair Searching

Have the function `PairSearching(num)` take the `num` parameter being passed and perform the following steps. First take all the single digits of the input number (which will always be a positive integer greater than 1) and add each of them into a list. Then take the input number and multiply it by any one of its own integers, then take this new number and append each of the digits onto the original list. Continue this process until an adjacent pair of the same number appears in the list. Your program should return the least number of multiplications it took to find an adjacent pair of duplicate numbers.

For example: if `num` is **134** then first append each of the integers into a list: [1, 3, 4]. Now if we take 134 and multiply it by 3 (which is one of its own integers), we get 402. Now if we append each of these new integers to the list, we get: [1, 3, 4, 4, 0, 2]. We found an adjacent pair of duplicate numbers, namely 4 and 4. So for this input your program should return **1** because it only took 1 multiplication to find this pair.

Another example: if `num` is **46** then we append these integers onto a list: [4, 6]. If we multiply 46 by 6, we get 276, and appending these integers onto the list we now have: [4, 6, 2, 7, 6]. Then if we take this new number, 276, and multiply it by 2 we get 552. Appending these integers onto the list we get: [4, 6, 2, 7, 6, 5, 5, 2]. Your program should therefore return **2** because it took 2 multiplications to find a pair of adjacent duplicate numbers (5 and 5 in this case).

Examples

Input: 8
Output: 3

Input: 198
Output: 2

```
function PairSearching(num) {
	let numArray = [num];
	let count = 0;
	let flag = false;

	let searching = (arr) => {
		let res = false;
		let numsArr = [];
		arr.forEach(val1 => {
			let numList = String(val1).split('');
			numList.forEach(val2 => {
				let product = val1 * Number(val2);
				let bigNumList =
numList.concat(String(product).split(''));
				if (repeatNum(bigNumList)) {
					res = true;
				} else {
					numsArr.push(product);
				}
			});
		});
		return res || numsArr;
	}

	let repeatNum = (numArr) => {
		for (let i = 0, len = numArr.length; i < len - 1; i++) {
			if (numArr[i] === numArr[i + 1]){
				return true;
			}
		}
		return false;
	}

	while (!flag) {
		count++;
		if (searching(numArray) === true) {
			return count;
		} else {
			numArray = searching(numArray);
		}
	}
}

// keep this function call here
PairSearching(readline());
```

Boggle Solver

Have the function `BoggleSolver(strArr)` read the array of strings stored in `strArr`, which will contain 2 elements: the first element will represent a 4x4 matrix of letters, and the second element will be a long string of comma-separated words each at least 3 letters long, in alphabetical order, that represents a dictionary of some arbitrary length. For example: `strArr` can be: ["rbfg, ukop, fgub, mnry", "bog,bop,gup,fur,ruk"]. Your goal is to determine if all the comma separated words as the second parameter exist in the 4x4 matrix of letters. For this example, the matrix looks like the following:

```
r b f g
u k o p
f g u b
m n r y
```

The rules to make a word are as follows:

1. A word can be constructed from sequentially adjacent spots in the matrix, where adjacent means moving horizontally, vertically, or diagonally in any direction.
2. A word cannot use the same location twice to construct itself.

The rules are similar to the game of Boggle. So for the example above, all the words exist in that matrix so your program should return the string **true**. If all the words cannot be found, then return a comma separated string of the words that cannot be found, in the order they appear in the dictionary.

Examples

Input: ["aaey, rrum, tgmn, ball", "all,ball,mur,raeymnl,tall,true,trum"]
Output: true

Input: ["aaey, rrum, tgmn, ball", "all,ball,mur,raeymnl,rumk,tall,true,trum,yes"]
Output: rumk,yes

```
function BoggleSolver(strArr) {
    //format the grid into an array of arrays
      gridArr = strArr[0].split(',').map(val => {
            return val.trim().split('');
      });
      //put localArr here to give it scope throughout the BoggleSolver function
      let localArr = [];

    //create an array of search terms
      let needleArr = strArr[1].split(',').map(val => {
            return val.trim();
      });

    //create an array of failing strings
      let res = needleArr.filter(val => {
            return !findWord(val);
      });


      return res.length ? res.join(',') : 'true';

      //-----------------helpers------------------
      //findWord checks to see if the given word (str) is in the grid
      function findWord (str) {
            //get a fresh, independent copy of the array
            localArr = makeArray(gridArr);

            //find the locations of the first letter, return false if none
            let hotSpots = findLetters(str[0]);
            if (!hotSpots) {
                  return false;
```

```
        }

        for (let i = 1, len = str.length; i < len; i++) {
                let newSpots = [];
                hotSpots.forEach(val => {
                        localArr[val[0]][val[1]] = '*';
                        newSpots.push(...adjacents([val[0], val[1]], str[i]));
                });
                hotSpots = newSpots;

                if (!hotSpots.length) {
                        return false;
                }
        }
        return true;
}

//adjacnts returns an array of all the points contiguous to a given point
function adjacents (arr, char) {
        let res = [];
        res.push(
                [arr[0] - 1, arr[1]],
                [arr[0] - 1, arr[1] + 1],
                [arr[0], arr[1] + 1],
                [arr[0] + 1, arr[1] + 1],
                [arr[0] + 1, arr[1]],
                [arr[0] + 1, arr[1] - 1],
                [arr[0], arr[1] - 1],
                [arr[0] - 1, arr[1] - 1]
        );

        res = res.filter(val => {
                return val[0] >= 0 && val[1] >= 0 && val[0] <= 3 && val[1] <= 3;
        }).filter(val => {
                return localArr[val[0]][val[1]] === char;
        });

        return res;
}


function findLetters(char) {
        let res = [];
        for (let row = 0; row < 4; row++) {
                for (let col = 0; col < 4; col++) {
                        if (gridArr[row][col] === char) {
                                res.push([row, col]);
                        }
                }
        }
        return res.length ? res : null;
}

function makeArray(arr) {
        let newArr = [];
```

```
            arr.forEach(val => {
                  newArr.push(val.slice(0));
            });
            return newArr;
      }
}
// keep this function call here
BoggleSolver(readline());
```

HTML Elements

Have the function `HTMLElements(str)` read the **`str`** parameter being passed which will be a string of HTML DOM elements and plain text. The elements that will be used are: `b, i, em, div, p`. For example: if **`str`** is "<div><b><p>hello world</p></b></div>" then this string of DOM elements is nested correctly so your program should return the string **true**.

If a string is not nested correctly, return the first element encountered where, if changed into a different element, would result in a properly formatted string. If the string is not formatted properly, then it will only be one element that needs to be changed. For example: if **`str`** is "<div><i>hello</i>world</b>" then your program should return the string **div** because if the first `<div>` element were changed into a `<b>`, the string would be properly formatted.

Examples

Input: "<div><div><b></b></div></p>"
Output: div

Input: "<div>abc</div><p><em><i>test test test</b></em></p>"
Output: i

```
function HTMLElements(str) {
      const inputArray = str.match(/(<\/?\w+>)/g);
      const len = inputArray.length;
      const HTMLStack = [];
      // go through the list of elements and push them onto the stack, or pull off
      // if closing tags.
      for (let i = 0; i < len; i++) {
            const isOpenTag = !(/\//.test(inputArray[i]));
            const tag = inputArray[i].replace(/[<>/]/g, '');
            if (isOpenTag) {
                  HTMLStack.push(tag);
            } else {
                  const popped = HTMLStack.pop();
                  if (tag !== popped) {
                        return popped;
                  }
            }
      }
      return HTMLStack.length ? HTMLStack.pop() : true;
}

// keep this function call here
HTMLElements(readline());
```

Missing Digit II

Have the function `MissingDigitII(str)` take the **`str`** parameter, which will be a simple mathematical formula with three numbers, a single operator (+, -, *, or /) and an equal sign (=) and return the two digits that complete the equation. In two of the numbers in the equation, there will be a single **?** character, and your program should determine what digits are missing and return them separated by a space. For example, if `str` is "38?5 * 3 = 1?595" then your program should output **6 1**.

The **?** character will always appear in both the first number and the last number in the mathematical expression. There will always be a unique solution.

Examples

Input: "56? * 106 = 5?678"

Output: 3 9

Input: "18?1 + 9 = 189?"

Output: 8 0

```
function MissingDigitII(str) {
	for (let i = 0; i <= 9; i++) {
		for (let j = 0; j <= 9; j++) {
			let newStr = str.replace(/\?/, String(i)).replace(/\?/,
String(j)).replace(/=/, '===');
			if (eval(newStr)) {
				return `${i} ${j}`;
			}
		}
	}
}

// keep this function call here
MissingDigitII(readline());
```

Palindromic Substring

Have the function `PalindromicSubstring(str)` take the `str` parameter being passed and find the longest palindromic substring, which means the longest substring which is read the same forwards as it is backwards. For example: if `str` is "abracecars" then your program should return the string **racecar** because it is the longest palindrome within the input string.

The input will only contain lowercase alphabetic characters. The longest palindromic substring will always be unique, but if there is none that is longer than 2 characters, return the string **none**.

Examples

Input: "hellosannasmith"

Output: sannas

Input: "abcdefgg"

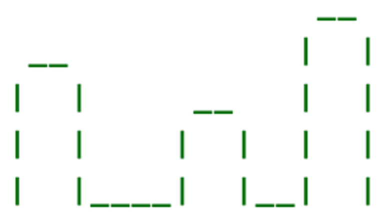
Output: none

```
function PalindromicSubstring(str) {
	for (let i = str.length; i > 2; i--) {
		for (let j = 0, len = str.length; j <= len - i; j++) {
			let newSlice = str.substr(j, i);
			if (isPalindrome(newSlice)) {
```

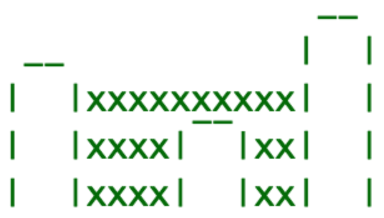
```
                                return newSlice;
                        }
                }
        }
        return 'none';
}

function isPalindrome(str) {
        return str === str.split('').reverse().join('');
}

// keep this function call here
PalindromicSubstring(readline());
```

Trapping Water

Have the function `TrappingWater(arr)` take the array of non-negative integers stored in `arr`, and determine the largest amount of water that can be trapped. The numbers in the array represent the height of a building (where the width of each building is 1) and if you imagine it raining, water will be trapped between the two tallest buildings. For example: if `arr` is [3, 0, 0, 2, 0, 4] then this array of building heights looks like the following picture if we draw it out:



Now if you imagine it rains and water gets trapped in this picture, then it'll look like the following (the x's represent water):



This is the most water that can be trapped in this picture, and if you calculate the area you get 10, so your program should return **10**.

Examples

Input: [1, 2, 1, 2]
Output: 1

Input: [0, 2, 4, 0, 2, 1, 2, 6]
Output: 11

```
function TrappingWater(arr) {
        let counter = 0;
        for (let i = 1, len = arr.length; i < len - 1; i++) {
                let hotSpot = arr[i];
                let preWall = Math.max(...arr.slice(0, i));
```

```
            let postWall = Math.max(...arr.slice(i + 1));
            let height = Math.min(preWall, postWall);
            if (hotSpot < height) {
                  counter += (height - hotSpot);
            }
      }
      return counter;
}
// keep this function call here
TrappingWater(readline());
```

Matrix Path

Have the function `MatrixPath(strArr)` take the `strArr` parameter being passed which will be a 2D matrix of 0 and 1's of some arbitrary size, and determine if a path of 1's exists from the top-left of the matrix to the bottom-right of the matrix while moving only in the directions: up, down, left, and right. If a path exists your program should return the string **true**, otherwise your program should return the number of locations in the matrix where if a single 0 is replaced with a 1, a path of 1's will be created successfully. If a path does not exist and you cannot create a path by changing a single location in the matrix from a 0 to a 1, then your program should return the string **not possible**. For example: if `strArr` is ["11100", "10011", "10101", "10011"] then this looks like the following matrix:

```
1 1 1 0 0
1 0 0 1 1
1 0 1 0 1
1 0 0 1 1
```

For the input above, a path of 1's from the top-left to the bottom-right does not exist. But, we can change a 0 to a 1 in 2 places in the matrix, namely at locations: [0,3] or [1,2]. So for this input your program should return **2**. The top-left and bottom-right of the input matrix will always be 1's.

Examples

Input: ["10000", "11011", "10101", "11001"]
Output: 1

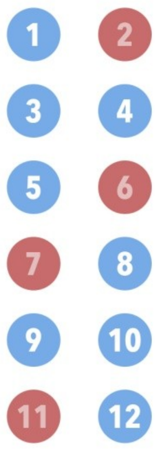
Input: ["1000001", "1001111", "1010101"]
Output: not possible

```
function MatrixPath(strArr) {
      let width = strArr[0].length;
      let height = strArr.length;
      // create a copy of our array, just in case.
      let newArr = copyArray(strArr);
      // create starting point
      newArr[height - 1][width - 1] = 'T'
      let stop = false;

      while(true) {
            let testArr = copyArray(newArr);
            for (let i = 0; i < height; i++) {
                  for (let j = 0; j < width; j++) {
                        if (testArr[i][j] === '1' && isAdjacent('T', testArr, [i,
j])) {
                              testArr[i][j] = 'T';
                        } else if (testArr[i][j] === '0' && isAdjacent('T',
testArr, [i, j])) {
                              testArr[i][j] = 'F';
                        }
                    }
                }
                if (isEqualArray(testArr, newArr)) {
                    newArr = testArr;
                    break;
                }
                newArr = testArr;
            }

            if (newArr[0][0] === 'T') {
                return 'true';
            }

            newArr[0][0] = 'G'

            while(true) {
                let testArr = copyArray(newArr);
                for (let i = 0; i < height; i++) {
                    for (let j = 0; j < width; j++) {
                        if (testArr[i][j] === '1' && isAdjacent('G', testArr, [i,
        j])) {
                            testArr[i][j] = 'G';
                        } else if (testArr[i][j] === '0' && isAdjacent('G',
        testArr, [i, j])) {
                            testArr[i][j] = 'F';
                        }
                    }
                }
                if (isEqualArray(testArr, newArr)) {
                    newArr = testArr;
                    break;
                }
                newArr = testArr;
            }
            let counter = 0;

            for (let i = 0; i < height; i++) {
                for (let j = 0; j < width; j++) {
                    if (newArr[i][j] === 'F' && isAdjacent('G', newArr, [i, j]) &&
        isAdjacent('T', newArr, [i, j])) {
                        counter ++;
                    }
                }
            }
            return counter ? counter : 'not possible';


        // ------------- helpers ------------------------------

            //    isAdjacent returns true if the array contains the designated char in a
        position
            //    adjacent to the designated position
            function isAdjacent (char, arr, position) {
                return [
                        [position[0] - 1, position[1]],
                        [position[0], position[1] + 1],
                        [position[0] + 1, position[1]],
                        [position[0], position[1] - 1]
                ].filter(val => {
                        return (val[0] >= 0 && val[0] < height && val[1] >= 0 && val[1] <
width);
                }).some(val => {
                        return arr[val[0]][val[1]] === char;
                });
        }

        // copyArray makes a new copy of a 2-deep array
        function copyArray (arr) {
        return Array.from(arr).map((val, ind) => {
                return Array.from(arr[ind]);
        });
}

        // isEqualArray checks to see if two 2-deep arrays have identical items
        function isEqualArray (arr1, arr2) {
        let height = arr1.length;
        let width = arr1[0].length;
        for (let i = 0; i < height; i++) {
                for (let j = 0; j < width; j++) {
                        if (arr1[i][j] !== arr2[i][j]) {
                                return false;
                        }
                }
        }
        return true;
}

        return strArr;
}

// keep this function call here
MatrixPath(readline());
```

Seating Students

Have the function `SeatingStudents(arr)` read the array of integers stored in `arr` which will be in the following format: [K, r1, r2, r3, ...] where K represents the number of desks in a classroom, and the rest of the integers in the array will be in sorted order and will represent the desks that are already occupied. All of the desks will be arranged in 2 columns, where desk #1 is at the top left, desk #2 is at the top right, desk #3 is below #1, desk #4 is below #2, etc. Your program should return the number of ways 2 students can be seated next to each other. This means 1 student is on the left and 1 student on the right, or 1 student is directly above or below the other student.

For example: if `arr` is [12, 2, 6, 7, 11] then this classrooms looks like the following picture:



Based on above arrangement of occupied desks, there are a total of 6 ways to seat 2 new students next to each other. The combinations are: [1, 3], [3, 4], [3, 5], [8, 10], [9, 10], [10, 12]. So for this input your program should return **6**. K will range from 2 to 24 and will always be an even number. After K, the number of occupied desks in the array can range from 0 to K.

Examples

Input: [6, 4]

Output: 4

Input: [8, 1, 8]

Output: 6

```
function SeatingStudents(arr) {
       const numSeats = arr[0];
       const occupied = arr.slice(1);
       let counter = 0;
       for (let i = 1; i <= numSeats; i++) {
              if (!occupied.includes(i)) {
                     if (!occupied.includes(i - 2) && i > 2) {
                            counter++;
                     }
                     if (!occupied.includes(i + 2) && i < (numSeats - 1)) {
                            counter++;
                     }
                     if (((i % 2 === 1) && !occupied.includes(i + 1)) || ((i % 2 ===
0) && !occupied.includes(i - 1))) {
                            counter++;
                     }
              }
       }
       return counter / 2;
}
SeatingStudents(readline());
```

Longest Matrix Path

Have the function `LongestMatrixPath(strArr)` take the array of strings stored in `strArr`, which will be an NxM matrix of positive single-digit integers, and find the longest increasing path composed of distinct integers. When moving through the matrix, you can only go up, down, left, and right. For example: if `strArr` is ["345", "326", "221"], then this looks like the following matrix:

```
3 4 5
3 2 6
2 2 1
```

For the input above, the longest increasing path goes from: 3 -> 4 -> 5 -> 6. Your program should return the number of connections in the longest path, so therefore for this input your program should return **3**. There may not necessarily always be a longest path within the matrix.

Examples

Input: ["12256", "56219", "43215"]
Output: 5

Input: ["67", "21", "45"]
Output: 3

```
function LongestMatrixPath(strArr) {
       let height = strArr.length;
       let width = strArr[0].length;
       //We will loop through each number, and return the biggest result
       //result will hold this number, and will be the return value.
       let returnVal = 0;
       strArr = strArr.map(val => {
              return val.split('');
       })

       //loop over the entire grid, and find the length for each place, by
       //calling crawler
       for (let i = 0; i < height; i++) {
              for (let j = 0; j < width; j++) {
                     let val = crawler([[strArr[i][j], [i, j]]]);
                     returnVal = Math.max(returnVal, val);
              }
       }
       return returnVal;

//------------------------ helpers ----------------------------------
       //adjacents takes a position on the grid and returns adjacent positions
       function adjacents(pos) {
              let newPositions = [
                     [pos[0], pos[1] + 1],
                     [pos[0], pos[1] - 1],
                     [pos[0] + 1, pos[1]],
                     [pos[0] - 1, pos[1]]
              ].filter (val => {
                     return val[0] >= 0 && val[1] >= 0 && val[0] < height && val[1] <
width;
              })
              return newPositions;
```

```
	}
	//this is the main function where things get done.  We don't need to worry
about
	//going backwards, since we must always increase.  So, we just see what the
possible
	//next steps are and quit when there are no more.
	function crawler(arr) {
		let counter = 0;
		while(arr.length) {
			let newArr = [];
			arr.forEach(val => {
				let pos = val[1];
				let newPos = adjacents(pos).filter(val => {
					return strArr[pos[0]][pos[1]] < strArr[val[0]]
[val[1]];
				}).map(val => {
					return [strArr[val[0]][val[1]], val];
				});
				newArr.push(...newPos);
			});
			if (newArr.length) {
				counter++;
			}
			arr = newArr;
		}
		return counter;
	}
}

// keep this function call here
LongestMatrixPath(readline());
```

Min Window Substring

Have the function `MinWindowSubstring(strArr)` take the array of strings stored in `strArr`, which will contain only two strings, the first parameter being the string N and the second parameter being a string K of some characters, and your goal is to determine the smallest substring of N that contains all the characters in K. For example: if `strArr` is ["aaabaaddae", "aed"] then the smallest substring of N that contains the characters a, e, and d is "dae" located at the end of the string. So for this example your program should return the string **dae**.

Another example: if `strArr` is ["aabdccdbcacd", "aad"] then the smallest substring of N that contains all of the characters in K is "aabd" which is located at the beginning of the string. Both parameters will be strings ranging in length from 1 to 50 characters and all of K's characters will exist somewhere in the string N. Both strings will only contains lowercase alphabetic characters.

Examples

Input: ["ahffaksfajeeubsne", "jefaa"]
Output: aksfaje

Input: ["aaffhkksemckelloe", "fhea"]
Output: affhkkse

```
function MinWindowSubstring(strArr) {
	let str = strArr[0];
	let needle = strArr[1].split('');

	//start with the smallest possible substrings, then go up
	for (let i = needle.length, len = str.length; i <= len; i++ ) {
		for (j = 0; j <= len - i; j++) {
			let mySlice = str.substr(j, i);
			if (isContained(mySlice)) {
				return mySlice;
			}
		}
	}
	return 'Not in string';

// ---------------------- helpers ----------------------------
	//isContained checks to see if all the chars in the needle are in the given
string
	function isContained(str) {
		let arr = str.split('');
		for (let i = 0, len = needle.length; i < len; i++) {
			let place = arr.findIndex(val => {
				return val === needle[i]
			});
			if (place === -1) {
				return false;
			} else {
				arr.splice(place, 1);
			}
		}
		return true;
	}
}

// keep this function call here
MinWindowSubstring(readline());
```

Matrix Chains

Have the function `MatrixChains(arr)` read the array of positive integers stored in `arr` where every pair will represent an NxM matrix. For example: if `arr` is [1, 2, 3, 4] this means you have a 1x2, 2x3, and a 3x4 matrix. So there are N-1 total matrices where N is the length of the array. Your goal is to determine the least number of multiplications possible after multiplying all the matrices. Matrix multiplication is associative so (A*B)*C is equal to A*(B*C).

For the above example, let us assume the following letters represent the different matrices: A = 1x2, B = 2x3, and C = 3x4. Then we can multiply the matrices in the following orders: (AB)C or A(BC). The first ordering requires (1*2*3) = 6 then we multiply this new 1x3 matrix by the 3x4 matrix and we get (1*3*4) = 12. So in total, this ordering required 6 + 12 = 18 multiplications. Your program should therefore return **18** because the second ordering produces more multiplications. The input array will contain between 3 and 30 elements.

Examples

Input: [2, 3, 4]
Output: 24

Input: [1, 4, 5, 6, 8]
Output: 98

```
function MatrixChains(arr) {
	let newArr = Array.from(arr);
	let counter = 0;
	let index = 0;
	for (let i = 0, len = newArr.length; i < len; i++) {
		if (arr[i] < newArr[index]) {
			index = i;
		}
	}
	console.log('index', index);
	//multiply to the right from the index, to extent possible
	moveRight(index);
	//multiply to the left from the index, to extent possible
	moveLeft(index);

	if (newArr.length === 3) {
		counter += newArr[0] * newArr[1] * newArr[2];
	}

	return counter;

	//------------------- helpers -----------------------------------
	//takes the smallest value and multiplies arrays going righttward
	function moveRight(index) {
		console.log('right');
		let move = newArr.length - index > 2;
		while(move) {
			counter += newArr[index] * newArr[index + 1] * newArr[index + 2];
			newArr.splice(index + 1, 1);
			if (newArr.length - index === 2) {
				move = false;
			}
		}
	}
	//takes the smallest value and multiplies going leftward
	function moveLeft(index) {
		console.log('left', index);
		let move = index > 1;
		while(move) {
			counter += newArr[index] * newArr[index - 1] * newArr[index - 2];
			newArr.splice(index - 1, 1);
			index--;
			if (newArr.length === 3) {
				move = false;
			}
		}
	}
}

// keep this function call here
MatrixChains(readline());
```

Histogram Area

Have the function `HistogramArea(arr)` read the array of non-negative integers stored in `arr` which will represent the heights of bars on a graph (where each bar width is 1), and determine the largest area underneath the entire bar graph. For example: if `arr` is [2, 1, 3, 4, 1] then this looks like the following bar graph:

```
          __
       __|  |
 __   |xx|xx|
|  |__|xx|xx|__
|  |  |xx|xx|  |
```

You can see in the above bar graph that the largest area underneath the graph is covered by the x's. The area of that space is equal to 6 because the entire width is 2 and the maximum height is 3, therefore 2 * 3 = 6. Your program should return **6**. The array will always contain at least 1 element.

Examples

Input: [6, 3, 1, 4, 12, 4]
Output: 12

Input: [5, 6, 7, 4, 1]
Output: 16

```
function HistogramArea(arr) {
	let maxRes = 0, len = arr.length;
	for (let i = 1; i <= len; i++) {
		for (let j = 0; j <= len - i; j++) {
			let arrSlice = arr.slice(j, j + i);
			let area = i * Math.min(...arrSlice);
			maxRes = Math.max(area, maxRes);
		}
	}
	return maxRes;
}

// keep this function call here
HistogramArea(readline());
```

Matching Characters

Have the function `MatchingCharacters(str)` take the `str` parameter being passed and determine the largest number of unique characters that exists between a pair of matching letters anywhere in the string. For example: if `str` is "ahyjakh" then there are only two pairs of matching letters, the two **a's** and the two **h's**. Between the pair of a's there are 3 unique characters: h, y, and j. Between the h's there are 4 unique characters: y, j, a, and k. So for this example your program should return **4**.

Another example: if `str` is "ghececgkaem" then your program should return **5** because the most unique characters exists within the farthest pair of **e** characters. The input string may not contain any character pairs, and in that case your program should just return **0**. The input will only consist of lowercase alphabetic characters.

Examples

Input: "mmmerme"
Output: 3

Input: "abccdefghi"
Output: 0

```
function MatchingCharacters(str) {
	maxRes = 0;
	let len = str.length;
	for (let i = 0; i < len - 1; i++) {
		let end = str.lastIndexOf(str[i]);
		let mySlice = str.slice(i + 1, end);
		maxRes = Math.max(maxRes, countUniq(mySlice));
	}
	return maxRes;

	function countUniq(str) {
		let arr = str.split('');
		let mySet = new Set(arr);
		return mySet.size;
	}
}

// keep this function call here
MatchingCharacters(readline());
```

Ternary Converter

Have the function `TernaryConverter(num)` take the `num` parameter being passed, which will always be a positive integer, and convert it into a ternary representation. For example: if `num` is 12 then your program should return 110.

Examples

Input: 21

Output: 210

Input: 67

Output: 2111

```
//The following takes advantage of built-in javascript method
//but is sort of cheating
//function TernaryConverter(num) {
	//simply taking advantage of a javascript built-in capability;
//	return num.toString(3);
//}

//Alternate, actually doing some work
function TernaryConverter(num) {
    //find what maximum power of three fits in num
	let powNum = Math.floor(Math.log(num) / Math.log(3));
	let res = '';

	for (let i = powNum; i >= 0; i--) {
		let nextDigit = Math.trunc(num / 3 ** i);
		num = num % (3 ** i);
		res += nextDigit;
	}
```

```
    return res;
}

// keep this function call here
TernaryConverter(readline());
```

Linear Congruence

Have the function `LinearCongruence(str)` read the `str` parameter being passed which will be a linear congruence equation in the form: "ax = b (mod m)" Your goal is to solve for x and return the number of solutions to x. For example: if `str` is "32x = 8 (mod 4)" then your program should return 4 because the answers to this equation can be either 0, 1, 2, or 3.

Examples

Input: "12x = 5 (mod 2)"

Output: 0

Input: "12x = 4 (mod 2)"

Output: 2

```
function LinearCongruence(str) {
    let components = str.match(/^(\d+)x\D+(\d+)\D+(\d+)\)$/);
    let multiplier = components[1];
    let factor = components[2];
    let mod = components[3];
    let counter = 0;

    for (let i = 0; i < mod; i++) {
        if (((multiplier * i - factor) % mod) === 0) {
            counter++;
        }
    }
    return counter;
}

// keep this function call here
LinearCongruence(readline());
```

Formatted Number

Have the function `FormattedNumber(strArr)` take the `strArr` parameter being passed, which will only contain a single element, and return the string **true** if it is a valid number that contains only digits with properly placed decimals and commas, otherwise return the string **false**. For example: if `strArr` is ["1,093,222.04"] then your program should return the string **true**, but if the input were ["1,093,22.04"] then your program should return the string **false**. The input may contain characters other than digits.

Examples

Input: ["0.232567"]

Output: true

Input: ["2,567.00.2"]
Output: false

```
function FormattedNumber(strArr) {
      const strNum = strArr[0];
      const hasDecimal = strNum.includes('.');
      const pattern = hasDecimal ? /^(?:\d{0,3})(?:,\d{3})*\.\d*$/ : /^(?:\d{0,3})
(?:,\d{3})*$/;
      return pattern.test(strNum);
}

// keep this function call here
FormattedNumber(readline());
```

Largest Row Column

Have the function `LargestRowColumn(strArr)` read the `strArr` parameter being passed which will be a 2D matrix of some arbitrary size filled with positive integers. Your goal is to determine the largest number that can be found by adding up three digits in the matrix that are within the same path, where being on the same path means starting from one of the elements and then moving either up, down, left, or right onto the next element without reusing elements. One caveat though, and that is when you calculate the sum of three digits, you should split the sum into two digits and treat the new digits as a row/column position in the matrix. So your goal is actually to find the sum of three digits that sums to the largest position in the matrix without going out of the bounds. For example: if `strArr` is ["345", "326", "221"] then this looks like the following matrix:

3 **4** 5
3 **2** **6**
2 2 1

The solution to this problem is to sum the bolded elements, 4 + 2 + 6, which equals 12. Then you take the solution, 12, and split it into two digits: 1 and 2 which represents row 1, column 2 in the matrix. This is the largest position you can get in the matrix by adding up 3 digits so your program should return **12**. If you for example added up 4 + 5 + 6 in the matrix you would get 15 which is larger than 12, but row 1, column 5 is out of bounds. It's also not possible with the current matrix to sum to any of the following numbers: 20, 21, 22. If you find a sum that is only a single digit, you can treat that as row 0, column N where N is your sum.

Examples

Input: ["234", "999", "999"]
Output: 22

Input: ["11111", "22222"]
Output: 4

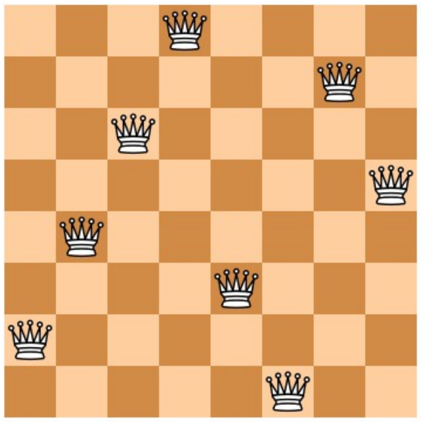
```
function LargestRowColumn(strArr) {


    let matrix = strArr.map(row => row.split(''));

    const adjacentElements = [
        [[1, 0], [0, 1]], // (one right), (one down)
        [[1, 0], [2, 0]], // right right
        [[1, 0], [1, 1]], // right down
        [[0, 1], [1, 1]], // down right
        [[0, 1], [0, 2]], // down down
        [[0, -1], [-1, 0]] // (one up), (one left)
    ];

    let largestValue = 0;
    let largestSum = 0;

    // Quickly done and very messy, please don't look.  :)

    matrix.forEach((row, rowIndex) => {
        row.forEach((col, colIndex) => {
            adjacentElements.forEach(ae => {
                //console.log(ae);
                const combos = [];
                combos.push(matrix[rowIndex][colIndex]);

                if (
                    rowIndex + ae[0][1] < matrix.length &&
                    rowIndex + ae[0][1] >= 0 &&
                    colIndex + ae[0][0] < matrix[0].length &&
                    colIndex + ae[0][0] >= 0
                ) {
                    combos.push(
                        matrix[rowIndex + ae[0][1]][colIndex + ae[0][0]]
                    );
                }
                if (
                    rowIndex + ae[1][1] < matrix.length &&
                    rowIndex + ae[1][1] >= 0 &&
                    colIndex + ae[1][0] < matrix[0].length &&
                    colIndex + ae[1][0] >= 0
                ) {
                    combos.push(
                        matrix[rowIndex + ae[1][1]][colIndex + ae[1][0]]
                    );
                }

                if (combos.length === 3) {
                    let sumThreeStr = combos
                        .reduce((sum, num) => (sum += Number(num)), 0)
                        .toString();

                    // now split digits of sum and check real value in matrix
                    let newRow, newCol;
                    if (sumThreeStr.length === 1) {
                        newRow = 0;
                        newCol = Number(sumThreeStr[0]);
                    } else {
                        [newRow, newCol] = sumThreeStr.split('');
                    }

                    if (
                        newRow >= 0 &&
                        newRow < matrix.length &&
                        newCol >= 0 &&
                        newCol < matrix[0].length
                    ) {
                        let value = matrix[newRow][newCol];
                        if (
                            // ??? spec says to use largest value, but it fails
                            // tests if I do this?
                            //value >= largestValue &&
                            Number(sumThreeStr) > largestSum
                        ) {
                            largestValue = value;
                            largestSum = Number(sumThreeStr);
                        }
                    }
                }
            });
        });
    });

    //console.log(largestValue, largestSum);

    return largestSum;
}


// keep this function call here
LargestRowColumn(readline());


function LargestRowColumn(strArr) {
	let height = strArr.length;
	let width = strArr[0].length;
	let helpers = helperMethods();

	//modify the strArr to an array of row arrays
	let newArr = helpers.reviseArr(strArr);

	let allSums = [];

	for (let i = 0; i < height; i++) {
		for (let j = 0; j < width; j++) {
			allSums.push(...helpers.createSums([i, j]));
		}
	}

	allSums = helpers.uniqArr(allSums).sort((val1, val2) => {return val1 - val2});

	//create an array of sums, converted into array positions
	let convertedSums = helpers.convertSums(allSums);

	//remove all points that don't exist on the grid
	let qualifyingSums = convertedSums.filter(val => {
		return parseInt(val[0], 10) < height && parseInt(val[1], 10) < width;
	});

	return parseInt(qualifyingSums.pop().join(''));
```

```
}

function helperMethods() {
	return {
		arr: [],

		//return an array stripped of all duplicate elements
		uniqArr(arr) {
			let mySet = new Set(arr);
			return Array.from(mySet);
		},

		//convert arr into an array of arrays containing integers
		reviseArr(arr) {
			let newArr = [];
			arr.forEach(val => {
				newArr.push(val.split('').map(val => {
					return parseInt(val, 10);
				}));
			});
			return this.arr = newArr;
		},

		//input a point, and get all valid adjacent points
		getPoints(point) {
			return [
				[point[0] + 1, point[1]],
				[point[0] - 1, point[1]],
				[point[0], point[1] + 1],
				[point[0], point[1] - 1]
			].filter(val => {
				return val[0] >= 0 && val[0] < this.arr.length && val[1]
>= 0 && val[1] < this.arr[0].length;
			})
		},

		//given a point, return a sorted array of values obtained by
		//adding numbers at adjacent points
		createSums(point) {
			let nextPoints = this.getPoints(point);
			let holder = [];
			nextPoints.forEach(val => {
				termPoints = this.getPoints(val).filter(val2 => {
					return (val2[0] !== point[0] || val2[1] !==
point[1]);
				});
				termPoints.forEach(val3 => {
					holder.push(this.arr[point[0]][point[1]] +
this.arr[val[0]][val[1]] + this.arr[val3[0]][val3[1]]);
				});
			});
			return this.uniqArr(holder).sort((val1, val2) => { return val1 -
val2});
		},
                convertSums(sums) {
                        return sums.map(val => {
                                let str = val.toString();
                                return str.length === 1 ? '0' + str : str;
                        }).map(val => {
                                return [val[0], val[1]];
                        });
                }
        }
}

// keep this function call here
LargestRowColumn(readline());
```

Eight Queens

Have the function `EightQueens(strArr)` read `strArr` which will be an array consisting of the locations of eight Queens on a standard 8x8 chess board with no other pieces on the board. The structure of `strArr` will be the following: ["(x,y)", "(x,y)", ...] where (x,y) represents the position of the current queen on the chessboard (x and y will both range from 1 to 8 where 1,1 is the bottom-left of the chessboard and 8,8 is the top-right). Your program should determine if all of the queens are placed in such a way where none of them are attacking each other. If this is true for the given input, return the string **true** otherwise return the first queen in the list that is attacking another piece in the same format it was provided.

For example: if `strArr` is ["(2,1)", "(4,2)", "(6,3)", "(8,4)", "(3,5)", "(1,6)", "(7,7)", "(5,8)"] then your program should return the string **true**. The corresponding chessboard of queens for this input is below (taken from Wikipedia).



Examples

Input: ["(2,1)", "(4,3)", "(6,3)", "(8,4)", "(3,4)", "(1,6)", "(7,7)", "(5,8)"]
Output: (2,1)

Input: ["(2,1)", "(5,3)", "(6,3)", "(8,4)", "(3,4)", "(1,8)", "(7,7)", "(5,8)"]
Output: (5,3)

```
const helpers = {};
function EightQueens(strArr) {
        // fix up the data as an array of arraySetup
        const newArr = helpers.arraySetup(strArr);
        const len = newArr.length;
        for (let i = 0; i < len; i++) {
                for (let j = 1 + i; j < len; j++) {
                        if (helpers.isAttacking(newArr[i], newArr[j])) {
                                return `(${newArr[i].slice(0, 2)})`;
                        }
                }
        }
        return true;
}
Object.assign(helpers, {
        arraySetup(strArr) {
                return strArr.map(val => JSON.parse(val
                        .replace(/\(/g, '[')
                        .replace(/\)/g, ']')));
        },
        isAttacking(pos1, pos2) {
                return (pos1[0] === pos2[0] ||
                        pos1[1] === pos2[1] ||
                        Math.abs(pos1[0] - pos2[0]) === Math.abs(pos1[1] - pos2[1]));
        }
});
// keep this function call here
EightQueens(readline());


// The structure of strArr will be the following: ["(x,y)", "(x,y)", ...]
// This seems to be wrong... in the example it shows y, x

function EightQueens(strArr) {

    let firstMatch = null;
    let matches = strArr.some(function(loc, index) {
        if (canAttack(strArr, loc[1], loc[3])) {
            firstMatch = index;
            return true;
        }
    });

    if (matches) {
        return strArr[firstMatch];
    }
    return 'true';


    function canAttack(strArr, x, y) {
        return strArr.some(function(loc){
            let coords = loc.substr(1, loc.length-2).split(',');
            // Check for same piece
            if (coords[0] === x && coords[1] === y) {
                return false;
            }
            // Check for horizontal moves
            if (coords[0] === x) {
                return true;
            }
            // Check for vertical moves
            if (coords[1] === y) {
                return true;
            }
            // Check for diagonal moves
            if (Math.abs(coords[0] - x) === Math.abs(coords[1] - y)) {
                return true;
            }
            return false;
        });

    }

}

// keep this function call here
EightQueens(readline());
```

Three Points

Have the function `ThreePoints(strArr)` read the array of strings stored in `strArr` which will always contain 3 elements and be in the form: ["(x1,y1)", "(x2,y2)", "(x3,y3)"]. Your goal is to first create a line formed by the first two points (that starts from the first point and moves in the direction of the second point and that stretches in both directions through the two points), and then determine what side of the line point 3 is on. The result will either be **right**, **left**, or **neither**. For example: if `strArr` is ["(1,1)", "(3,3)", "(2,0)"] then your program should return the string **right** because the third point lies to the right of the line formed by the first two points.

Examples

Input: ["(0,-3)", "(-2,0)", "(0,0)"]
Output: right

Input: ["(0,0)", "(0,5)", "(0,2)"]
Output: neither

```
function ThreePoints(strArr) {

    // Parse input
    const [pointA, pointB, pointX] = strArr.map(point => {
        const [, x, y] = point.match(/\\((-?[\\d]+),(-?[\\d]+)\\)/).map(Number);
        return { x, y };
    });

    // y = mx + b
    const slope = (pointB.y - pointA.y) / (pointB.x - pointA.x); // m
    const yIntercept = (pointA.y - slope) / pointA.x; // b

    // x = (y - b) / m
    let x;
    if (slope === Infinity) {
        x = pointX.x;
    } else {
        x = pointX.y - yIntercept + slope;
    }

    if (x === 0 || Number.isNaN(x)) {
        return 'neither';
    }

    return x < 0 ? 'left' : 'right';

}

// keep this function call here
ThreePoints(readline());
```

Character Removal

Have the function CharacterRemoval(strArr) read the array of strings stored in strArr, which will contain 2 elements: the first element will be a sequence of characters representing a word, and the second element will be a long string of comma-separated words, in alphabetical order, that represents a dictionary of some arbitrary length. For example: strArr can be: ["worlcde", "apple,bat,cat,goodbye,hello,yellow,why,world"]. Your goal is to determine the minimum number of characters, if any, can be removed from the word so that it matches one of the words from the dictionary. In this case, your program should return **2** because once you remove the characters "c" and "e" you are left with "world" and that exists within the dictionary. If the word cannot be found no matter what characters are removed, return **-1**.

Examples

Input: ["baseball", "a,all,b,ball,bas,base,cat,code,d,e,quit,z"]
Output: 4

Input: ["apbpleeeef", "a,ab,abc,abcg,b,c,dog,e,efd,zzzz"]
Output: 8

```
function CharacterRemoval(strArr) {
    //separate the components into the taget word and the dictionary
    let needle = strArr[0].split('');
      let dictionary = strArr[1].split(',');

    //run a check on each dictionary word to see if it can go inside the target
    //then convert each word into its length
      let inWords = dictionary.filter(val => {
            return isInside(val);
      }).map(val => {
            return val.length;
      });
    //if no words pass the test, return -1;
      if (!inWords.length) {
            return -1;
      }
      //otherwise, return the length of the target, less the number of
      //chars in the longest string
      return needle.length - Math.max(...inWords);

    //------ helpers ---------------------------
    //isInside tests if the given string can be found in the target word
      function isInside(str) {
            let testNeedle = Array.from(needle);
            for (let i = 0, len = str.length; i < len; i++) {
                  let placement = testNeedle.findIndex(val => {
                        return val === str[i];
                  });
                  if (placement === -1) {
                        return false;
                  }
                  testNeedle.fill('*', 0, placement + 1);
            }
            return true;
      }
}

// keep this function call here
CharacterRemoval(readline());
```

Simple Password

Have the function `SimplePassword(str)` take the `str` parameter being passed and determine if it passes as a valid password that follows the list of constraints:

1. It must have a capital letter.
2. It must contain at least one number.
3. It must contain a punctuation mark.
4. It cannot have the word "password" in the string.
5. It must be longer than 7 characters and shorter than 31 characters.

If all the above constraints are met within the string, the your program should return the string **true**, otherwise your program should return the string **false**. For example: if `str` is "apple!M7" then your program should return "true".

Examples

Input: "passWord123!!!!"

Output: false

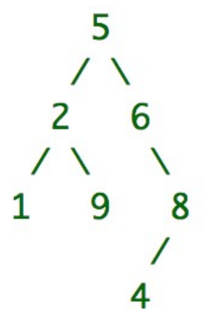
Input: "turkey90AAA="

Output: true

```
function SimplePassword(str) {
      //test are in order as presented`
      return (
            /[A-Z]/.test(str) &&
            /\d/.test(str) &&
            /[^\w\s/]/.test(str) &&
            !/password/i.test(str) &&
            str.length > 7 && str.length < 31
      );
}

// keep this function call here
SimplePassword(readline());
```

Preorder Traversal

Have the function `PreorderTraversal(strArr)` take the array of strings stored in `strArr`, which will represent a binary tree with integer values in a format similar to how a binary heap is implemented with NULL nodes at any level represented with a #. Your goal is to return the pre-order traversal of the tree with the elements separated by a space. For example: if `strArr` is ["5", "2", "6", "1", "9", "#", "8", "#", "#", "#", "#", "4", "#"] then this tree looks like the following tree:



For the input above, your program should return the string **5 2 1 9 6 8 4** because that is the pre-order traversal of the tree.

Examples

Input: ["4", "1", "5", "2", "#", "#", "#"]
Output: 4 1 2 5

Input: ["2", "6", "#"]
Output: 2 6

```
const helpers = {};
function PreorderTraversal(strArr) {
    const newArr = Array.from(strArr);
       const fullArray = helpers.createFullArray(newArr);
       const preppedArray = helpers.setSubArrays(fullArray);
       return helpers.orderArray(preppedArray)
             .filter(val => val !== '#')
             .join(' ');
}

Object.assign(helpers, {
       // holderArray: [],
       // createFullArray takes the array in the format presented, and adds hash
marks to fill
       // out the array, so that a tree that is n levels deep will be represented by
an array
       // of 2^n - 1 items.
       createFullArray(arr) {
           // this if statement is total BS, but necessary to pass two flawed
Coderbyte tests.
             if (helpers.isFullGraph(arr)) {
                    return arr;
             }
```

```
            const workArray = Array.from(arr);
            const returnArray = [];
            let checker = false;
            let power = 0;
            while (!checker) {
                  const items = workArray.splice(0, (Math.pow(2, power)));
                  items.forEach((val, index) => {
                        if (val === '#') {
                              workArray.splice(index * 2, 0, '#', '#');
                        }
                  });
                  returnArray.push(...items);
                  power++;
                  checker = workArray.every(val => val === '#');
            }
            return returnArray;
      },

      // splitArrays takes the array representing the full binary tree and returns
two arrays,
      // the left half and the right half under the top
      splitArrays(arr) {
            const rightArray = [];
            const leftArray = [];
            arr.forEach((subArr) => {
                  const len = subArr.length;
                  if (len > 1) {
                        leftArray.push((subArr.splice(0, len / 2)));
                        rightArray.push(subArr);
                  }
            });
            return [leftArray, rightArray];
      },

      // takes an array of 2^n items and places them in n subarrays, each of length
2^index,
      // where index is the index of the subarray within the array.
      setSubArrays(arr) {
            const resArray = [];
            let power = 0;
            while (arr.length > 0) {
                  const newArr = arr.splice(0, Math.pow(2, power));
                  resArray.push(newArr);
                  power++;
            }
            return resArray;
      },

      orderArray(arr) {
            if (arr.length === 1) {
                  return arr[0];
            }
            const subs = helpers.splitArrays(arr);
            return arr[0].concat(helpers.orderArray(subs[0]),
helpers.orderArray(subs[1]));
```

```
        },
        isFullGraph(arr) {
                const arrLength = arr.length;
                for (let i = 1; i < 50; i++) {
                        if (arrLength === Math.pow(2, i) - 1) {
                                return true;
                        }
                }
                return false;
        }
});

// keep this function call here
PreorderTraversal(readline());
```

String Zigzag

Have the function `StringZigzag(strArr)` read the array of strings stored in `strArr`, which will contain two elements, the first some sort of string and the second element will be a number ranging from 1 to 6. The number represents how many rows to print the string on so that it forms a zig-zag pattern. For example: if `strArr` is ["coderbyte", "3"] then this word will look like the following if you print it in a zig-zag pattern with 3 rows:

```
c       r       e
  o   e   b   t
    d       y
```

Your program should return the word formed by combining the characters as you iterate through each row, so for this example your program should return the string **creoebtdy**.

Examples

Input: ["cat", "5"]

Output: cat

Input: ["kaamvjjfl", "4"]

Output: kjajfavlm

```
function StringZigzag(strArr) {
    //get the pieces off the strArr
        let word = strArr[0].split('');
        let zigLength = strArr[1];

        //create an array of arrays to hold the string pieces
        let resArr = [];
        for (let i = 1; i <= zigLength; i++) {
                resArr.push([]);
        }
        console.log('resArr', resArr);
        //start in the first subArray
        let subArr = 0;
    //go through the string, assigning letters to the proper
    //array, which is detetermined with the nextNum() method
        for (let i = 0, len = word.length; i < len; i++) {
                resArr[subArr].push(word[i]);
```

```
			subArr = nextNum(subArr, i);
		}

	//sew the subarray elements
		resArr = resArr.map(val => {
			return val.join('');
		});

		return resArr.join('');

//---------------- helper functions -------------------
		function nextNum(num, position) {
			let base = zigLength * 2 - 2;
			if (base === 0) { return num}
			if ((position % base) < base / 2) {
				return ++num;
			} else {
				return --num;
			}
		}

		return strArr;
}
// keep this function call here
StringZigzag(readline());
```

Off Binary

Have the function `OffBinary(strArr)` read the array of strings stored in `strArr`, which will contain two elements, the first will be a positive decimal number and the second element will be a binary number. Your goal is to determine how many digits in the binary number need to be changed to represent the decimal number correctly (either 0 change to 1 or vice versa). For example: if `strArr` is ["56", "011000"] then your program should return **1** because only 1 digit needs to change in the binary number (the first zero needs to become a 1) to correctly represent 56 in binary.

Examples

Input: ["5624", "0010111111001"]

Output: 2

Input: ["44", "111111"]

Output: 3

```
function OffBinary(strArr) {
		let target = parseInt(strArr[0]).toString(2);
		let arrow = strArr[1];

		if (target.length === arrow.length) {
			let counter = 0;
			for (let i = 0, len = target.length; i < len; i++) {
				if (target[i] !== arrow[i]){
					counter++;
				}
```

```
            }
            return counter;
      } else {
            return 'length error';
      }
      return strArr;
}
// keep this function call here
OffBinary(readline());
```

Longest Consecutive

Have the function `LongestConsecutive(arr)` take the array of positive integers stored in `arr` and return the length of the longest consecutive subsequence (LCS). An LCS is a subset of the original list where the numbers are in sorted order, from lowest to highest, and are in a consecutive, increasing order. The sequence does not need to be contiguous and there can be several different subsequences. For example: if `arr` is [4, 3, 8, 1, 2, 6, 100, 9] then a few consecutive sequences are [1, 2, 3, 4], and [8, 9]. For this input, your program should return **4** because that is the length of the longest consecutive subsequence.

Examples

Input: [6, 7, 3, 1, 100, 102, 6, 12]
Output: 2

Input: [5, 6, 1, 2, 8, 9, 7]
Output: 5

```
function LongestConsecutive(arr) {
      arr.sort((a, b) => a - b);
      newArr = Array.from(new Set(arr));
      let counter = 0;
      let maxCount = 0;
      for (let i = 0, len = newArr.length; i < len; i++) {
            if (newArr[i + 1] - newArr [i] === 1) {
                  counter++;
            } else {
                  if (counter > maxCount) {
                        maxCount = counter;
                        counter = 0;
                  }
            }
      }
      return maxCount + 1;
}

// keep this function call here
LongestConsecutive(readline());
```

String Expression

Have the function `StringExpression(str)` read the `str` parameter being passed which will contain the written out version of the numbers 0-9 and the words "minus" or "plus" and convert the expression into an actual final number written out as well. For example: if `str` is "foursixminustwotwoplusonezero" then this converts to "46 - 22 + 10" which evaluates to 34 and your program should return the final string **threefour**. If your final answer is negative it should include the word "negative."

Examples

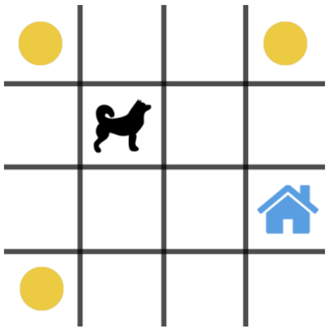
Input: "onezeropluseight"
Output: oneeight

Input: "oneminusoneone"
Output: negativeonezero

```
function StringExpression(str) {
       let newString = str.slice(0);
       let dictionary = [
           ['zero', '0'],
           ['one', '1'],
           ['two', '2'],
             ['three', '3'],
             ['four', '4'],
             ['five', '5'],
             ['six', '6'],
             ['seven', '7'],
             ['eight', '8'],
             ['nine', '9'],
             ['minus', '-'],
             ['plus', '+']
       ];
       dictionary.forEach(val => {
             let regEx = new RegExp (val[0], 'g');
             newString = newString.replace(regEx, val[1]);
       });

       let resString = eval(newString).toString();

       dictionary.slice(0,10).forEach(val => {
             let regEx = new RegExp (val[1], 'g');
             resString = resString.replace(regEx, val[0]);
       });

       return resString.replace('-', 'negative');
}
// keep this function call here
StringExpression(readline());
```

Charlie the Dog

Have the function `CharlietheDog(strArr)` read the array of strings stored in `strArr` which will be a 4x4 matrix of the characters 'C', 'H', 'F', 'O', where C represents Charlie the dog, H represents its home, F represents dog food, and O represents and empty space in the grid. Your goal is to figure out the least amount of moves required to get Charlie to grab each piece of food in the grid by moving up, down, left, or right, and then make it home right after. Charlie cannot move onto the home before all pieces of food have been collected. For example: if `strArr` is ["FOOF", "OCOO", "OOOH", "FOOO"], then this looks like the following grid:



For the input above, the least amount of steps where the dog can reach each piece of food, and then return home is 11 steps, so your program should return the number **11**. The grid will always contain between 1 and 8 pieces of food.

Examples

Input: ["OOOO", "OOFF", "OCHO", "OFOO"]
Output: 7

Input: ["FOOO", "OCOH", "OFOF", "OFOO"]
Output: 10

```
function CharlietheDog(strArr) {
    const mapArray = strArr.join('').split('');
    helpers.dogHouse = mapArray.findIndex(val => val === 'H');
    let charlieLocation = mapArray.findIndex(val => val === 'C')
    let foodArray = helpers.getFoodArray(mapArray);
    let counter = 0;
    let cutOff = '';
    let stateArray = [{
        charlie: charlieLocation,
        food: foodArray
    }];

    while (true) {
        counter++;
        stateArray = stateArray.map(stateObject => {
            return helpers.newStates(stateObject);
        });

        if (!stateArray.includes('done')) {
            stateArray = stateArray.reduce ((val1, val2) => {
                return val1.concat(val2);
            }, []);
            stateArray = helpers.shaker(stateArray);
        } else {
            break;
        }
    }
    return counter;
}
```

```
// function shaker(stateCollection)

var helpers = {
    //is assigned the value of the location of the doghouse
    dogHouse: null,
    //takes the array representing the board and returns an array indicating
    //where the dog treats are
    getFoodArray(arr) {
        let resArray = [];
        let len = arr.length;
        for (let i = 0; i < len; i++) {
            if (arr[i] === 'F') {
                resArray.push(i);
            }
        }
        return resArray;
    },
    //a simple helper that takes two arrays and returns a boolean, whether the
    //contents of the array are the same, but don't have to be in order
    isArrayEqual(arr1, arr2) {
        arr1.sort((val1, val2) => val1 - val2);
        arr2.sort((val1, val2) => val1 - val2);
        let len = arr1.length;
        if (len !== arr2.length) {
            return false;
        }
        for (let i = 0; i < len; i++) {
            if (arr1[i] !== arr2[i]) {
                return false;
            }
        }
        return true;
    },
    //takes a stateObject (charlie: num, food: arr) and returns all the possible/
    //existing stateObjects after the next move from that position.  Returns 'done'
    //if the dog has landed at home, with no more treats to get.
    newStates(stateObject) {
        let done = false;
        let location = stateObject.charlie;
        let returnArray = [];
        let possibles = [];
        if (location > 3) {
            possibles.push(location - 4);
        }
        if (location % 4 !== 0) {
            possibles.push(location - 1);
        }
        if (location % 4 !== 3) {
            possibles.push(location + 1);
        }
        if (location < 12) {
            possibles.push(location + 4);
        }
        possibles.forEach(newLocation => {
            if (newLocation === this.dogHouse && !stateObject.food.length) {
                done = true;
            }
            if (newLocation !== this.dogHouse) {
                foodLocations = stateObject.food.filter( hotLocation => {
                    return newLocation !== hotLocation
                });
            returnArray.push({charlie: newLocation, food: foodLocations})
            }
        });
        return done ? 'done' : returnArray;
    },
    //takes an array of stateObjects, compares them, and throws out duplicates
    shaker(stateObjectArray) {
        let arrayCopy = Array.from(stateObjectArray);
        let len = arrayCopy.length;
        for(let i = 0; i < len; i++) {
            arrayCopy = arrayCopy.filter((stateObject, ind) => {
                return (
                    ind <= i ||
                    stateObject.charlie !== arrayCopy[i].charlie ||
                    !this.isArrayEqual(stateObject.food, arrayCopy[i].food)
                );
            });
        }
        return arrayCopy;
    }
}

// keep this function call here
CharlietheDog(readline());
```

Plus Minus

Have the function `PlusMinus(num)` read the `num` parameter being passed which will be a combination of 1 or more single digits, and determine if it's possible to separate the digits with either a plus or minus sign to get the final expression to equal zero. For example: if `num` is 35132 then it's possible to separate the digits the following way, 3 - 5 + 1 + 3 - 2, and this expression equals zero. Your program should return a string of the signs you used, so for this example your program should return **-++-**. If it's not possible to get the digit expression to equal zero, return the string **not possible**.

If there are multiple ways to get the final expression to equal zero, choose the one that contains more minus characters. For example: if `num` is 26712 your program should return **-+--** and not **+-+-**.

Examples

Input: 199

Output: not possible

Input: 26712

Output: -+--

```
function PlusMinus(num) {
    if (num === 26712) {
        return '-+--';
    }
    helpers.digitsArray = num.toString(10).split('');
    let numSigns = helpers.digitsArray.length - 1;
    for (let i = 2 ** numSigns; i < 2 ** (numSigns + 1); i++) {
        let numString = i.toString(2).slice(1);
        let string = helpers.createString(i.toString(2).slice(1));
        if (eval(string) === 0) {
            return numString.replace(/0/g, '+').replace(/1/g, '-');
        }
    }
    return 'not possible';

}

let helpers = {
    digitsArray: [],
    createString(str) {
        let signArray = str.split('');
        let newArray = [];
        const len = this.digitsArray.length;
        for (let i = 0; i < len; i++) {
            newArray.push(this.digitsArray[i]);
            if (i < len - 1) {
                newArray.push(signArray[i] === '0' ? '+' : '-');
            }
        }
        return newArray.join('');
    }
}
// keep this function call here
PlusMinus(readline());
```

Primes

Have the function `Primes(num)` take the `num` parameter being passed and return the string **true** if the parameter is a prime number, otherwise return the string **false**. The range will be between 1 and 2^16.

Examples

Input: 4

Output: false

Input: 1709

Output: true

```
function Primes(num) {
    switch (num) {
             case 1:
                   return 'false';
             case 2:
```

```
                        return 'true';
                default: {
                        const pivot = Math.floor(Math.sqrt(num));
                        for (let i = 2; i <= pivot; i++) {
                                if (!(num % i)) {
                                        return 'false';
                                }
                        }
                        return 'true';
                }
        }
}

// keep this function call here
Primes(readline());
```

String Calculate

Have the function `StringCalculate(str)` take the `str` parameter being passed and evaluate the mathematical expression within in. The double asterisks (**) represent exponentiation.

For example, if `str` were "(2+(3-1)*3)**3" the output should be **512**. Another example: if `str` is "(2-0)(6/2)" the output should be **6**. There can be parenthesis within the string so you must evaluate it properly according to the rules of arithmetic. The string will contain the operators: +, -, /, *, (,), and **. If you have a string like this: #/#*# or #+#(#)/#, then evaluate from left to right. So divide then multiply, and for the second one multiply, divide, then add. The evaluations will be such that there will not be any decimal operations, so you do not need to account for rounding.

Examples

Input: "6*(4/2)+3*1"

Output: 15

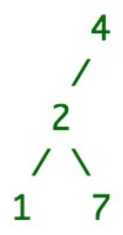
Input: "100*2**4"

Output: 1600

```
function StringCalculate(str) {
    const workingString = str.replace(/([1-9])(\()/g, '$1*$2');
      return eval(workingString);
}

// keep this function call here
StringCalculate(readline());
```

Tree Constructor Two

Have the function `TreeConstructor(strArr)` take the array of strings stored in `strArr`, which will contain pairs of integers in the following format: **(i1,i2)**, where **i1** represents a child node in a tree and the second integer **i2** signifies that it is the parent of **i1**. For example: if `strArr` is ["(1,2)", "(2,4)", "(7,2)"], then this forms the following tree:



which you can see forms a proper binary tree. Your program should, in this case, return the string **true** because a valid binary tree can be formed. If a proper binary tree cannot be formed with the integer pairs, then return the string **false**. All of the integers within the tree will be unique, which means there can only be one node in the tree with the given integer value.

Examples

Input: ["(1,2)", "(2,4)", "(5,7)", "(7,2)", "(9,5)"]
Output: true

Input: ["(1,2)", "(1,3)"]
Output: false.

Sudoku Quadrant Checker

Have the function SudokuQuadrantChecker(**strArr**) read the **strArr** parameter being passed which will represent a 9x9 Sudoku board of integers ranging from 1 to 9. The rules of Sudoku are to place **each** of the 9 integers integer in every row and column and not have any integers repeat in the respective row, column, or 3x3 sub-grid. The input **strArr** will represent a Sudoku board and it will be structured in the following format: ["(N,N,N,N,N,x,x,x,x)","(...)","(...)",...)] where N stands for an integer between 1 and 9 and x will stand for an empty cell. Your program will determine if the board is legal; the board also does not necessarily have to be finished. If the board is legal, your program should return the string **legal** but if it isn't legal, it should return the 3x3 quadrants (separated by commas) where the errors exist. The 3x3 quadrants are numbered from 1 to 9 starting from top-left going to bottom-right.

For example, if **strArr** is:
["(1,2,3,4,5,6,7,8,1)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(1,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)"] then your program should return **1,3,4** since the errors are in quadrants 1, 3 and 4 because of the repeating integer 1.

Another example, if **strArr** is:
["(1,2,3,4,5,6,7,8,9)","(x,x,x,x,x,x,x,x,x)","(6,x,5,x,3,x,x,4,x)","(2,x,1,1,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,9)"] then your program should return **3,4,5,9**.

Hard challenges are worth *15 points* and you are not timed for them.

Examples

Input: ["(1,2,3,4,5,6,7,8,1)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(1,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)","(x,x,x,x,x,x,x,x,x)"]
Output: 1,3,4

```
function SudokuQuadrantChecker(strArr) {

  //clean the array from array of strings to an array of nine nine-member arrays.

  strArr = strArr.map(function(val) {

    return val.replace(/[()]/g, '').split(',');

  })

  //make two new copies of the array to be used and an empty array to hold bad blocks

  var copy1Arr = strArr.slice(0);

  var copy2Arr = strArr.slice(0);

  var holdObj = {};
```

```
   //val is the row string, ind is the row number - send each to rowTester

   strArr.forEach(function(val, ind) {

      rowTester(val, ind);

   });


   //Part II - organize arrays based on columns and send to vertTester method
for checking


   //create a 'row' out of each column, and send it to vertTester.  vertTester
will

   //be exactly the same as rowTester, except the output will be inverted to
give

   //the correct blocks

   copy1Arr[0].forEach(function(val, ind) {

      vertArr = [];

      for (var i = 0; i < 9; i++) {

         vertArr.push(copy1Arr[i][ind]);

      }

      vertTester(vertArr, ind);

   });


   //Part III - create a 'row' out of each block, and send it to blockTester.
```

```
//blockTester will be exactly the same as rowTester, except the output will
//be the offending blocks

for (var a = 0; a < 3; a++) {
   for (var b = 0; b < 3; b++) {
      var blockArr = [];
      for (var i = 0; i < 3; i++) {
         for (var j = 0; j < 3; j++) {
            blockArr.push(copy2Arr[a * 3 + i][b * 3 + j]);
         }
      }
      blockTester(blockArr, a.toString() + b.toString());
   }
}

var blocks = Object.keys(holdObj);

if (!blocks.length) {
   return 'legal'
} else {
   blocks = blocks.map(function(val) {
```

```
            return parseInt(val, 3) + 1;
        }).sort();


        return blocks.join(',');
    }


//-------------------helper functions--------------------


    function rowTester(arr, num) {
        //format the string as an array of nine number strings
        //var newArr = str.replace(/[()]/g, '').split(',');


        for (var i = 0; i < 9; i++) {
            for (var j = 0; j < 9; j++) {
                if (arr[i] !== 'x' && arr[i] === arr[j] && i !== j) {
                    var rep1 = Math.floor(num / 3).toString() + Math.floor(i /
3).toString();
                    var rep2 = Math.floor(num / 3).toString() + Math.floor(j /
3).toString();
                    rep1 in holdObj ? holdObj[rep1]++ : holdObj[rep1] = 1;
                }
            }
        }

    }


    function vertTester(arr, num) {

        for (var i = 0; i < 9; i++) {

            for (var j = 0; j < 9; j++) {

                if (arr[i] !== 'x' && arr[i] === arr[j] && i !== j) {

                    var rep1 = Math.floor(i / 3).toString() + Math.floor(num /
3).toString();

                    var rep2 = Math.floor(j / 3).toString() + Math.floor(num /
3).toString();

                    rep1 in holdObj ? holdObj[rep1]++ : holdObj[rep1] = 1;

                }

            }

        }

    }


    function blockTester(arr, block) {

        for (var i = 0; i < 9; i++) {

            for (var j = 0; j < 9; j++) {

                if (arr[i] !== 'x' && arr[i] === arr[j] && i !== j) {

                    block in holdObj ? holdObj[block]++ : holdObj[block] = 1;
                }
            }
        }
    }
}

// keep this function call here
SudokuQuadrantChecker(readline());
```