**Project Euler**

### Problem 63

## Powerful digit counts

“The 5-digit number, 16807=75, is also a fifth power. Similarly, the 9-digit number, 134217728=89, is a ninth power.

How many *n*-digit positive integers exist which are also an *n*th power?”

**Çözüm**

const begin = process.hrtime()

let c = 0

for(let i=1,n=-1,k=1,d=0;i<4\*10\*\*10;i+=k){

let b = Math.floor(i\*\*(1/Math.ceil(Math.log10(i))))

if(n!==b){

k = i-d

d = i

c++

}n=b}const end = process.hrtime(begin)

console.info('Result %d (Execution time: %ds %dms)',c, end[0], end[1] / 1000000)

**Cevap**

**49**

**Problem 123**

## Prime square remainders

“Let *p*n be the *n*th prime: 2, 3, 5, 7, 11, ..., and let *r* be the remainder when (*p*n−1)*n* + (*p*n+1)*n* is divided by *p*n2.

For example, when *n* = 3, *p*3 = 5, and 43 + 63 = 280 ≡ 5 mod 25.

The least value of *n* for which the remainder first exceeds 109 is 7037.

Find the least value of *n* for which the remainder first exceeds 1010.”

**Çözüm**

function Problem123(limit)

{

var primes=[2,3]

var sum = 5

var n = 5

while (n <= limit){

if (isPrime(n)){

primes.push(n)}

n = n+2

if ((n <= limit)&& (isPrime(n))) {primes.push(n)}

n = n+4

}

var res=primes.map((a, i) => sqrem(i+1,a))

for(var k=0;k<=res.length;k=k+2){

if(res[k]>10000000000){

var index=k+1

break;

}

}

console.log(index)

}

Problem123(500000)

function isPrime(num) {

if (num <= 1) {

return false

} else if (num <= 3) {

return true

} else if (num%2 === 0 || num%3 === 0) {

return false

}

let i = 5

while (i\*i <= num) {

if (num%i === 0 || num%(i+2) === 0) {

return false

}

i += 6

}

return true

}

function sqrem(n,p){

if(n%2==1){

twins=(2\*n\*p)

twins%=(p\*p);

}

if(n%2==0){

twins=2

}

return twins;

}

console.timeEnd();

**Cevap**

21035

**Problem 73**

## Counting fractions in a range

“Consider the fraction, *n/d*, where *n* and *d* are positive integers. If *n*<*d* and HCF(*n,d*)=1, it is called a reduced proper fraction.

If we list the set of reduced proper fractions for *d* ≤ 8 in ascending order of size, we get:

1/8, 1/7, 1/6, 1/5, 1/4, 2/7, 1/3, **3/8, 2/5, 3/7**, 1/2, 4/7, 3/5, 5/8, 2/3, 5/7, 3/4, 4/5, 5/6, 6/7, 7/8

It can be seen that there are 3 fractions between 1/3 and 1/2.

How many fractions lie between 1/3 and 1/2 in the sorted set of reduced proper fractions for *d* ≤ 12,000?”

**Çözüm**

function main(){

let count = 0;

let D = 12000;

for(let i = 2;i<=(D/2);i++){

if(i%100 === 0)console.log(i);

let upper = 2\*i,lower =

Math.min(3\*i,D);

for(let d =

upper+1;d<=lower;d++){

if(GCD(i,d) === 1){

count++;

if(i >

5990)console.log(i+"/"+d);

}

}

}

console.log(count);

}

**Cevap**

**7295372**

### **Problem 3**

## Largest prime factor

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

**Çözüm**

let i = 0;

let primeMagic = 600851475143;

let storedNumbers = [];

let themMultiplied = 0;

for (i = 0; themMultiplied < primeMagic; i++) {

if (primeMagic % i === 0) {

storedNumbers.push(i);

themMultiplied = storedNumbers.reduce(multiplyThem);

function multiplyThem(value, total) {

return value \* total;

}

if (themMultiplied === primeMagic) {

console.log(storedNumbers[storedNumbers.length-1]);

}

}

}

**Cevap**

**6857**

**Problem 23**

## Non-abundant sums

A perfect number is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of 28 would be 1 + 2 + 4 + 7 + 14 = 28, which means that 28 is a perfect number.

A number *n* is called deficient if the sum of its proper divisors is less than *n* and it is called abundant if this sum exceeds *n*.

As 12 is the smallest abundant number, 1 + 2 + 3 + 4 + 6 = 16, the smallest number that can be written as the sum of two abundant numbers is 24. By mathematical analysis, it can be shown that all integers greater than 28123 can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.

Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

**Çözüm**

function divisors(n) {

var divs = []

for (let i = 1; i <= Math.floor(n/2); i++) {

if (n%i == 0)

divs.push(i)

}

return divs

}

var abun = []

for (let i = 1; i <= 28123; i++) {

var divs = divisors(i)

var sumDivs = 0

for (let y = 0; y < divs.length; y++)

sumDivs += divs[y]

if (sumDivs > i)

abun.push(i)

}

var negatives = {}

for (let y = 0; y < abun.length; y++) {

for (let z = 0; z < abun.length; z++) {

var sum = abun[y] + abun[z]

if (sum <= 28123)

negatives[sum] = true

}

}

var sumPositives = 0

var countPositives = 0

var maxPositives = 12

for (let i = 1; i <= 28123; i++)

if (!negatives[i]) {

if (i > maxPositives) maxPositives = i

countPositives++

sumPositives += i

}

console.log('Max: '+maxPositives)

console.log('Count: '+countPositives)

console.log('Sum: '+sumPositives)

**Leetcode**

1313

**Decompress Run-Length Encoded List**

We are given a list nums of integers representing a list compressed with run-length encoding.

Consider each adjacent pair of elements [freq, val] = [nums[2\*i], nums[2\*i+1]] (with i >= 0).  For each such pair, there are freq elements with value val concatenated in a sublist. Concatenate all the sublists from left to right to generate the decompressed list.

Return the decompressed list.

**Çözüm**

var decompressRLElist = function(nums) {

let result = [];

for(let i=0; i<nums.length; i+=2) {

let j = 1;

let temp = [];

while(j <= nums[i]) {

temp.push(nums[i+1]);

j++;

}

result = result.concat(temp);

}

return result;

};

**1323**

**Maximum 69 Number**

Given a positive integer num consisting only of digits 6 and 9.

Return the maximum number you can get by changing ****at most**** one digit (6 becomes 9, and 9 becomes 6).

**Çözüm**

const maximum69Number = num => {

let numStr = String(num).split('')

let idx = numStr.findIndex(num => num === '6');

if (!numStr.includes('6')) {

return num;

}

numStr[idx] = '9'

return Number(numStr.join(''));

}

**1403**

**Minimum Subsequence in Non-Increasing Order**

“Given the array nums, obtain a subsequence of the array whose sum of elements is ****strictly greater**** than the sum of the non included elements in such subsequence.

If there are multiple solutions, return the subsequence with ****minimum size**** and if there still exist multiple solutions, return the subsequence with the ****maximum total sum**** of all its elements. A subsequence of an array can be obtained by erasing some (possibly zero) elements from the array.

Note that the solution with the given constraints is guaranteed to be ****unique****. Also return the answer sorted in ****non-increasing**** order.”

**Çözüm**

var minSubsequence = function(nums) {

nums.sort((a, b) => b - a);

let total = nums.reduce((a,b) => a + b), sum = 0;

const res = [];

for(let i = 0; i < nums.length; i++) {

sum += nums[i];

total -= nums[i];

res.push(nums[i]);

if(sum > total) return res;

}

};

**933**

**Number of Recent Calls**

Write a class RecentCounter to count recent requests.

It has only one method: ping(int t), where t represents some time in milliseconds.

Return the number of pings that have been made from 3000 milliseconds ago until now.

Any ping with time in [t - 3000, t] will count, including the current ping.

It is guaranteed that every call to ping uses a strictly larger value of t than before.

**Çözüm**

const limit = 3000;class RecentCounter {

constructor() {

this.queue = [];

}

/\*\*

\* @param {number} t

\* @return {number}

\*/

ping(t) {

const { queue } = this;

while (queue[0] < t - limit) {

queue.shift();

}

return queue.push(t);

}

}

**893**

**Groups of Special-Equivalent Strings**

You are given an array A of strings.

A move onto S consists of swapping any two even indexed characters of S, or any two odd indexed characters of S.

Two strings S and T are special-equivalent if after any number of moves onto S, S == T.

For example, S = "zzxy" and T = "xyzz" are special-equivalent because we may make the moves "zzxy" -> "xzzy" -> "xyzz" that swap S[0] and S[2], then S[1] and S[3].

Now, a group of special-equivalent strings from A is a non-empty subset of A such that:

1. Every pair of strings in the group are special equivalent, and;
2. The group is the largest size possible (ie., there isn't a string S not in the group such that S is special equivalent to every string in the group)

Return the number of groups of special-equivalent strings from

**Çözüm**

/\*\*

\* @param {string[]} A

\* @return {number}

\*/var numSpecialEquivGroups = function(A) {

const groupSet = new Set();

A.forEach(a => groupSet.add(transform(a)));

return groupSet.size;

};

const transform = S => {

const even = S.split('').filter((\_, i) => i % 2 === 0);

const odd = S.split('').filter((\_, i) => i % 2 === 1);

even.sort();

odd.sort();

return `${even.join('')}${odd.join('')}`;

};