Índice

1. Parejas de primos **2**

2. Brute Force TSP **6**

3. Precision **11**
3.1. Explanation 1: 11
3.2. Explicación 2 11

4. Supeditado Contest 2020 **13**

5. Poketoms **16**

6. Cripto **21**

7. Organizando los satélites! **22**

8. DOCE COFRES de IPA **23**

9. Dados escalonados **25**

10. Todo esto es irracional **26**
10.1. Opción 1: Dos casos (Solución de Facundo Martín Gutiérrez) 26
10.2. Opción 2: Cardinalidad (Solución de Federico Felguer) 26

11. Super mega repeticons **27**

1. Parejas de primos

Propuesto por: Sebastián Marin, Marcos Kolodny y Mateo Carranza Vélez

Observemos que se pueden formar 1010 parejas haciendo por ejemplo $(1, 6)$, $(2, 5)$, $(3, 4)$ y agrupando i con $2027 - i$ para $i \geq 7$.

Una observación clave para resolver este problema es que como $1 + 2 > 2$, y 2 es el único primo par, todos los primos formados serán impares, y por lo tanto deberán ser la suma de un número par y uno impar. De esta manera, podemos formar un grafo bipartito con los números pares de un lado, los impares del otro (1010 números en cada conjunto), y conectar con una arista un par con un impar cuando su suma es un número primo. Además, el peso de esa arista será naturalmente el primo obtenido como suma.

En estas condiciones, lo que necesitamos calcular es el matching perfecto de mayor producto, y el de menor producto. Podemos calcular esto en forma exacta utilizando el algoritmo húngaro, cambiando sumas por productos, restas por divisiones, y el valor neutro 0 por el valor 1. Las comparaciones quedan igual. Los números involucrados pueden ser muy grandes y el cómputo muy lento e ineficiente, pero haciendo las cuentas exactas (por ejemplo con `Fraction` de python) podemos estar completamente seguros de que el cálculo es exacto. El código python que se muestra más adelante computa el producto máximo en aproximadamente 7 horas (en hardware razonable), y como puede ejecutarse en paralelo (corriendo los dos programas) el cálculo del mínimo y el del máximo (que es casi idéntico pero invirtiendo comparaciones, valores centinela y/o signos), se resuelve en forma completamente exacta el problema en ese tiempo.

Notemos también que minimizar/maximizar el producto de los primos es equivalente a minimizar/maximizar la suma de los logaritmos de dichos primos. Si aplicamos húngaro directamente sobre los logaritmos en double (o long double), la implementación será mucho más simple y rápida, pero tenemos solamente 15 o 18 dígitos decimales de precisión, mientras que los números enteros involucrados como solución tienen miles de dígitos, razón por la cual no es evidente a priori que sus logaritmos permitan distinguirlos con la precisión necesaria. Resulta que, en la matriz particular de este problema, la precisión de double con los logaritmos es suficiente, y se obtiene la misma solución que al ejecutar el algoritmo exacto con racionales y precisión arbitraria.

Ninguno de los autores o participantes consultados a la fecha de escribir estas líneas conoce ninguna demostración o análisis de error razonable para este resultado: es simplemente una observación empírica al ejecutar los algoritmos en la matriz de esta instancia específica. No es particularmente difícil construir manualmente otras matrices de costos basadas en logaritmos de enteros hasta 1000, en las cuales sí se requieran miles de dígitos de precisión para distinguir dos posibles soluciones, con lo cual este enfoque no tendría suficiente precisión para una matriz de costos arbitraria con cotas similares.

Resolviendo el problema de esta forma, como la respuesta que obtendremos es un double en lugar de un entero, lo que haremos será multiplicar (módulo $10^9 + 7$) los primos involucrados en el emparejamiento solución obtenido con el algoritmo húngaro.

Código python exacto (pero **muy** lento) para computar el máximo producto:

```
from fractions import Fraction

INFTO = 2050**1020

def isprime(n):
    if n < 2:
        return False
    for i in range(2,n):
        if n % i == 0:
            return False
    return True

n = 1010
```

```
def rowval(i):
    return 1 + 2*i

def colval(j):
    return 2 + 2*j

# Matriz de costo del problema.
# Cuando la suma no es primo, utilizamos un valor extremadamente
# chico para que jamas convenga multiplicar por el.

mt = [[Fraction(rowval(i) + colval(j),1) if isprime(rowval(i) + colval(j)) else Fraction(1,INFTO)
                for j in range(n)] for i in range(n)]

# De aqui en mas, es una implementacion del algoritmo hungaro

lx =  [Fraction(1,1) for i in range(n)]
ly =  [Fraction(1,1) for i in range(n)]

slk = [Fraction(INFTO,1) for i in range(n)]
slkx = [0 for i in range(n)]


def updtree(x):
    for y in range(n):
        if lx[x] * ly[y] / mt[x][y] < slk[y]:
            slk[y] = lx[x] * ly[y] / mt[x][y]
            slkx[y] = x

class Queue:
    def __init__(self):
        self.q = []
        self.nextIndex = 0

    def empty(self):
        return self.nextIndex == len(self.q)

    def push(self, x):
        self.q.append(x)

    def pop(self):
        ret = self.q[self.nextIndex]
        self.nextIndex += 1
        return ret

def hungar():
    global slk, slkx, ly, lx, mt
    for i in range(n):
        ly[i] = Fraction(1,1)
        lx[i] = max(mt[i])
    xy = [-1 for i in range(n)]
```

```
yx = [-1 for i in range(n)]
for m in range(n):
    S = [0 for i in range(n)]
    T = [0 for i in range(n)]
    prv  = [-1 for i in range(n)]
    slk = [Fraction(INFTO,1) for i in range(n)]
    q = Queue()
    def bpone(e,p):
        q.push(e)
        prv[e] = p
        S[e] = 1
        updtree(e)
    for i in range(n):
        if xy[i] == -1:
            bpone(i, -2)
            break
    x=0
    y=-1
    while (y==-1):
        while (not q.empty() and y==-1):
            x = q.pop()
            for j in range(n):
                if (mt[x][j] == lx[x] * ly[j] and not T[j]):
                    if (yx[j] == -1):
                        y = j
                        break
                    T[j] = 1
                    bpone(yx[j], x)
        if (y!=-1):
            break
        dlt = Fraction(INFTO,1)
        for j in range(n):
            if (not T[j]):
                dlt = min(dlt, slk[j])
        for k in range(n):
            if (S[k]):
                lx[k] /= dlt
            if (T[k]):
                ly[k] *= dlt
            if (not T[k]):
                slk[k] /= dlt
        for j in range(n):
            if (not T[j] and slk[j] == Fraction(1,1)):
                if (yx[j] == -1):
                    x = slkx[j]
                    y = j
                    break
                else:
                    T[j] = 1
                    if (not S[yx[j]]):
                        bpone(yx[j], slkx[j])
```

```
        if (y!=-1):
            p = x
            while p != -2:
                yx[y] = p
                ty = xy[p]
                xy[p] = y
                y = ty
                p = prv[p]
        else:
            break
    res = Fraction(1,1)
    for i in range(n):
        res *= mt[i][xy[i]]
    return res


print(hungar())
```

2. Brute Force TSP

Propuesto por: Marcelo Fornet

The most important observation to solve this problem is noticing that since $19! > 10^{17}$, then across all the 10^{17} permutations checked by Fito, only the last 19 elements are being permuted, the first 81 elements (elements from 1 to 81) remains in the same place.

So if we fix the first 81 numbers, we should solve a TSP instance with the last 19 elements in the graph. There is a standard technique to do this (`https://www.quora.com/How-do-I-solve-the-traveling-salesman-problem-using-dynamic-programming`).

This alone is not enough, since $10^{17} < 19!$. All permutations of the 19 last elements shouldn't be considered. To solve this problem you should find the k-th permutation (`https://stackoverflow.com/questions/31216097/given-n-and-k-return-the-kth-permutation-sequence`) and try to find the solution over all the smaller permutations similar to how you would solve a digits dp (`https://codeforces.com/blog/entry/53960`).

The given graph is not that random, the seed was chosen such that the solution for different k are the following:

k	shortest path
18!	43259711
10^{17}	42798818
19!	42754163

This is the full solution in c++ for more details:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

using namespace std;

const int maxn = 100;
const int oo = 0x3f3f3f3f;

int graph[maxn][maxn];

// Return if n! > k
bool big_fak(int n, long long k)
{
    long long answer = 1;

    for (int i = 1; i <= n; ++i)
    {
        if (answer > k / i)
        {
            return true;
        }

        answer *= i;
    }
```

```
    return answer > k;
}

// Return n!
long long fak(long long n)
{
    long long answer = 1;
    for (int i = 2; i <= n; ++i)
    {
        answer *= i;
    }
    return answer;
}

// Check if the nth bit of mask is set
bool test(long long mask, int n)
{
    return mask >> n & 1;
}

// Find the position of the biggest bit of mask
int __lg(long long mask)
{
    int n = -1;
    while (mask)
    {
        ++n;
        mask >>= 1;
    }
    return n;
}

// Solve the problem for a graph of n nodes and at most k permutations
int solve(int n, long long k)
{
    k--;
    int num = 0;
    while (!big_fak(num, k))
        num++;

    num = min(num, n);
    int start = n - num;

    vector<vector<int>> dp = vector<vector<int>>(num, vector<int>(1 << num, oo));

    for (int mask = 1; mask < (1 << num); ++mask)
    {
        if ((mask & (mask - 1)) == 0)
        {
            dp[__lg(mask)][mask] = 0;
```

```
    }
    else
    {
        for (int i = 0; i < num; ++i)
        {
            if (test(mask, i))
            {
                auto n_mask = mask ^ (1 << i);
                for (int j = 0; j < num; ++j)
                {
                    if (test(n_mask, j))
                    {
                        dp[i][mask] = min(dp[i][mask],
                                          dp[j][n_mask] + graph[i + start][j + start]);
                    }
                }
            }
        }
    }
}

vector<bool> used(n);
int answer = oo;
int last = -1;
int prefix = 0;

for (int i = 0; i < n; ++i)
{
    int u = 0;
    while (true)
    {
        while (used[u])
            ++u;

        if (big_fak(n - i - 1, k))
        {
            used[u] = true;
            if (last != -1)
            {
                prefix += graph[last][u];
            }
            last = u;
            break;
        }

        int mask = 0;
        for (int j = 0; j < num; ++j)
        {
            if (!used[start + j])
            {
                mask |= 1 << j;
```

```
                }
            }

            int edge = last != -1 ? graph[last][u] : 0;
            int tail = dp[u - start][mask];
            int cost = prefix + edge + tail;
            answer = min(answer, cost);

            k -= fak(n - i - 1);
            u++;
        }
    }

    answer = min(answer, prefix);
    return answer;
}

// Random setup
long long seed = 2122071420;
long long mult = 1103515245;
long long inc = 12345;
long long mod = (1LL << 31) - 1;

long long next_rand()
{
    return seed = (seed * mult + inc) & mod;
}

// Generate the graph
void generate(int n)
{
    long long checksum = 0;

    for (int i = 0; i < n; ++i)
    {
        for (int j = i + 1; j < n; ++j)
        {
            graph[j][i] = next_rand() % 999999 + 1;
            graph[i][j] = graph[j][i];
            checksum = (checksum + graph[i][j]) % 1000000007;
        }
    }

    cout << "checksum: " << checksum << endl;
}

int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
```

```
    int n = 100;
    long long k = 100000000000000000;

    generate(n);
    cout << solve(n, k) << endl;

    return 0;
}
```

3. Precision

Propuesto por: Quimey Vivas y Federico Felguer

3.1. Explanation 1:

There is a pdf with the statement of the problem called `libre.pdf`. To that pdf we compress it using gzip and then base64 encode it.

We split the result in chunks of 70 characters and write that to the pdf called `enunciado.pdf` using white text.

You can get back to the original problem by reversing these steps.

The original problem is a classic IMO problem (1980) with a twist and can be solved by reading section 14.4 of problem solving strategies by Engel or looking for solution of project euler 356.

This is the sage code we used to produce the answer:

```
def compute(n):
    assert n % 2 == 0
    M = Matrix(Zmod(10 ** 10), [[5, 12], [2, 5]])
    a, _ = M ** (n // 2) * vector([1, 0])
    return (2 * a - 1) % 10 ** 10

def compute_raw(n):
    return int(RealField(500)(expand((sqrt(2) + sqrt(3)) ** n))) % 10 ** 10

for i in range(10):
    print(compute(2 * i), compute_raw(2 * i))

print(compute(12345678))
```

```
(1, 1)
(9, 9)
(97, 97)
(969, 969)
(9601, 9601)
(95049, 95049)
(940897, 940897)
(9313929, 9313929)
(92198401, 92198401)
(912670089, 912670089)
292841609
```

3.2. Explicación 2

Como en la explicación anterior, primero llegamos al auténtico enunciado del problema, que pide calcular los últimos 10 dígitos de la parte entera de $(\sqrt{2}+\sqrt{3})^{12345678}$. Es muy conveniente elevar al cuadrado aprovechando el exponente par, para obtener $(\sqrt{2}+\sqrt{3})^2 = 5+2\sqrt{6}$, de modo que ahora buscamos los últimos 10 dígitos de la parte entera de $(5+2\sqrt{6})^{6172839}$.

Si numéricamente vamos calculando las potencias de $(5+2\sqrt{6})$, veremos algo muy curioso, que es que están muy muy cerca de ser entero, cada vez más. La razón es que si miramos la secuencia de números

$a_n = (5 + 2\sqrt{6})^n + (5 - 2\sqrt{6})^n$, esta secuencia siempre produce números enteros (pues al desarrollar los binomios de Newton, las potencias pares producen enteros, y las impares quedan con signos distintos y se cancelan), para todo n, pero el segundo término tiende a 0 a medida que n tiende a infinito (ya que $0 < 5 - 2\sqrt{6} < 1$).

Entonces la idea es entender la secuencia de números enteros a_n, porque de lo anterior sabemos que lo que queremos son los últimos 10 dígitos de su término 6172839 (menos 1). Esta forma se parece mucho a la fórmula para la sucesión de fibonacci, y de hecho si λ es raíz de un polinomio, la sucesión $a_n = \lambda^n$ es solución de una recurrencia lineal asociada al polinomio. En este caso, $5+2\sqrt{6}$ y $5-2\sqrt{6}$ son las raíces del polinomio $X^2-10X+1$, cuya recursión asociada es $a_{n+2} = 10a_{n+1} - a_n$. Podemos entonces implementar esta recursión, directamente, y mirar el término 6172839. Sabemos evaluando que inicialmente $a_0 = 2$ y $a_1 = 10$.

El siguiente one-liner de Haskell implementa esta idea:

```
Prelude> iterate (\(a,b) -> (b, (10*b-a) `mod` 10^10)) (2,10) !! 6172839
(292841610,2151360002)
```

Con lo cual $a_{6172839}$ termina en 292841610. Ya vimos que debemos restar uno, así que la respuesta es 292841609 (con 9 dígitos porque omitimos un cero a la izquierda)

4. Supeditado Contest 2020

Propuesto por: Agustín Santiago Gutiérrez

En este problema, es posible dar un algoritmo de programación dinámica exponencial, basado en un estado que calcule $p(S)$, la probabilidad de que el juego se arruine dado que en el bolillero todavía quedan las bolillas del conjunto S. Notar que $|S|$ determina a quién le toca sacar bolilla a continuación, y según los elementos de S podemos computar a qué nuevo estado se salta, y con qué probabilidad.

Esto no será suficiente para resolver el problema para 200, como se pide, pero sí para valores pequeños. Si lo implementamos en python:

```
from fractions import Fraction

fp = dict() # (remainingSet) --> fail_probability_from_that_start

def sortuple(s):
    return tuple(sorted(s))

def fail_prob_from(s):
    assert len(s) > 0
    st = sortuple(s)
    if st not in fp:
        if len(s) == 1:
            if next(iter(s)) == 0:
                ret = Fraction(1,1)
            else:
                ret = Fraction(0,1)
        else:
            ret = Fraction(0,1)
            elements = list(s)
            current = len(s) - 1
            total_options = len(s) - (1 if current in s else 0)
            for x in elements:
                if x != current:
                    s.remove(x)
                    ret += fail_prob_from(s)
                    s.add(x)
            ret /= total_options
        fp[st] = ret
    return fp[st]

def calculate_fail_chance(n):
    s = set()
    for i in range(n):
        s.add(i)
        print(i, fail_prob_from(s))

def main():
    calculate_fail_chance(16)

if __name__ == "__main__":
    main()
```

Podemos obtener rápidamente resultados:

```
0 1
1 0
2 1/4
3 5/36
4 19/144
5 203/1800
6 4343/43200
7 63853/705600
8 58129/705600
9 160127/2116800
10 8885501/127008000
11 1500518539/23051952000
12 404156337271/6638962176000
13 16040576541971/280496151936000
14 1694200740145637/31415569016832000
15 24047240650458731/471233535252480000
```

Y aquí tenemos secuencias de números enteros. Siempre que tenemos secuencias de enteros, podemos ir a `https://oeis.org/` y probar suerte. En este caso, si ponemos los numeradores que estamos viendo, caemos en `https://oeis.org/A102262`, y si ponemos los denominadores caemos en `https://oeis.org/A102263`. Allí se dan fórmulas cuadráticas mediante programación dinámica para este problema, que es más que suficiente para resolverlo en forma exacta para $n = 200$.

Sin embargo, copiando las fórmulas sin entender nos perdemos razonar cómo llegar a ellas, ya que allí no están bien explicadas. Y las fórmulas concretas que se dan allí son un poco feas y poco intuitivas, por lo que preferimos dar un algoritmo de programación dinámica cuadrático diferente (aunque en cierta forma muy similar a esas fórmulas).

Hay un equipo especial, el último (digamos, el n) tal que nos interesa realizar $n-1$ pasos **sin haber nunca seleccionado** al equipo especial. La probabilidad buscada es exactamente eso: la probabilidad de realizar $n-1$ pasos, sin seleccionar al n.

Estas dos cosas son generalizables: podemos plantear $p(i,k)$, la probabilidad de realizar i pasos del proceso, sin haber nunca seleccionado a **ninguno** de k equipos especiales predefinidos (pero que **no** sean de los primeros i equipos). La respuesta al problema es simplemente $p(n-1,1)$ con esta definición, así que computar p es suficiente para resolver el problema. Y p admite una fórmula recursiva:

```
from fractions import Fraction

n = 200

p = [[Fraction(1,1) for j in range(n+1)] for i in range(n+1)]
for i in range(1,n):
    for k in range(1, n-i+1):
        p[i][k] = ( p[i-1][k+1] * Fraction(n-i-k, n-i) +
                    (p[i-1][k] - p[i-1][k+1]) * Fraction(n-i+1-k, n-i+1)
                  )

print(p[n-1][1])
```

Esto porque para haber realizado los primeros i pasos sin haber seleccionado a alguno de los k particulares (que podemos imaginar para fijar ideas que son los equipos $n, n-1, \cdots, n-k+1$, pero por simetría, la probabilidad es exactamente la misma sin importar quiénes sean exactamente los que prefijamos entre los equipos restantes luego de los primeros i pasos), hay solo dos posibilidades disjuntas:

- En el paso i, **estaba todavía** en el bolillero el nombre del equipo i. En este caso, quiere decir que tras haber hecho los primeros $i-1$ pasos, no se habían seleccionado a ninguno de los k especiales, **pero tampoco al** i. Por la simetría antes mencionada, la probabilidad de que eso pase es exactamente $p(i-1, k+1)$, pues es como considerar que el i también era especial, ya que había que evitar seleccionarlo para que esté disponible en este paso. Dado que llegamos a esta situación, como el nombre del i es uno de los $n-i+1$ nombres que quedan para elegir, hay solo $n-i$ opciones viables equiprobables, pero solamente $n-i-k$ de ellas son las que no seleccionan uno de los k valores especiales que se debe esquivar. Por lo tanto finalmente, la probabilidad en estecaso es $p(i-1, k+1) \cdot \frac{n-i-k}{n-i}$
- En el paso i, **ya no estaba** en el bolillero el nombre del equipo i. En este caso, hemos llegado a realizar los primeros $i-1$ pasos evitando los k especiales, pero sin embargo no hemos evitado el i. La probabilidad de que esto ocurra es $p(i-1, k) - p(i-1, k+1)$, pues $p(i-1, k)$ es la probabilidad de esquivar los k especiales sin importar lo que hagamos con el i, y $p(i-1, k+1)$ es como en el caso anterior, la probabilidad de **además** de evitar los k especiales, evitar también al i. Ahora, dado que llegamos a esta situación, como el nombre del i **no** es uno de los $n-i+1$ nombres que quedan para elegir, tenemos todas las $n-i+1$ posibles elecciones como válidas, todas equiprobables, y de ellas, solamente $n-i+1-k$ evitan los k especiales. Por lo tanto finalmente, la probabilidad de lograr el objetivo de realizar los primeros i pasos evitando los k especiales es en este subcaso, $(p(i-1, k) - p(i-1, k+1)) \cdot \frac{n-i+1-k}{n-i+1}$.

La probabilidad total para $p(i, k)$ será la suma de las dos anteriores, y esto es lo que computa el algoritmo.

5. Poketoms

Propuesto por: Agustín Santiago Gutiérrez

En este problema simplemente tenemos que buscar los datos y computar lo que pide el enunciado:

- Conseguir un listado de los 100 primeros pokemones, ordenados por número de pokemón.
- Conseguir un listado de los 100 primeros elementos químicos, ordenados por número atómico.
- Escribir un programa que recorra los números de 1 a 100 y agregue a la respuesta el elemento de la lista correspondiente, según sea o no primo.

Los listados son fáciles de encontrar buscando en internet. Un detalle sutil que el autor no tuvo en cuenta pero puede traer problemas es que se debe usar la escritura "Aluminum" para ese metal, en lugar de "Aluminium", ya que admite las dos escrituras y la lista que utilizó el autor lo escribía así. Otro detalle es que los Nidoran contienen habitualmente caracteres no ascii para distinguir las versiones masculina y femenina, que se deben borrar.

Código del autor:

```
elements = [
"Hydrogen",
"Helium",
"Lithium",
"Beryllium",
"Boron",
"Carbon",
"Nitrogen",
"Oxygen",
"Fluorine",
"Neon",
"Sodium",
"Magnesium",
"Aluminum",
"Silicon",
"Phosphorus",
"Sulfur",
"Chlorine",
"Argon",
"Potassium",
"Calcium",
"Scandium",
"Titanium",
"Vanadium",
"Chromium",
"Manganese",
"Iron",
"Cobalt",
"Nickel",
"Copper",
"Zinc",
"Gallium",
"Germanium",
```

```
"Arsenic",
"Selenium",
"Bromine",
"Krypton",
"Rubidium",
"Strontium",
"Yttrium",
"Zirconium",
"Niobium",
"Molybdenum",
"Technetium",
"Ruthenium",
"Rhodium",
"Palladium",
"Silver",
"Cadmium",
"Indium",
"Tin",
"Antimony",
"Tellurium",
"Iodine",
"Xenon",
"Cesium",
"Barium",
"Lanthanum",
"Cerium",
"Praseodymium",
"Neodymium",
"Promethium",
"Samarium",
"Europium",
"Gadolinium",
"Terbium",
"Dysprosium",
"Holmium",
"Erbium",
"Thulium",
"Ytterbium",
"Lutetium",
"Hafnium",
"Tantalum",
"Tungsten",
"Rhenium",
"Osmium",
"Iridium",
"Platinum",
"Gold",
"Mercury",
"Thallium",
"Lead",
"Bismuth",
```

```
"Polonium",
"Astatine",
"Radon",
"Francium",
"Radium",
"Actinium",
"Thorium",
"Protactinium",
"Uranium",
"Neptunium",
"Plutonium",
"Americium",
"Curium",
"Berkelium",
"Californium",
"Einsteinium",
"Fermium",
]


pokemons=[
"Bulbasaur",
"Ivysaur",
"Venusaur",
"Charmander",
"Charmeleon",
"Charizard",
"Squirtle",
"Wartortle",
"Blastoise",
"Caterpie",
"Metapod",
"Butterfree",
"Weedle",
"Kakuna",
"Beedrill",
"Pidgey",
"Pidgeotto",
"Pidgeot",
"Rattata",
"Raticate",
"Spearow",
"Fearow",
"Ekans",
"Arbok",
"Pikachu",
"Raichu",
"Sandshrew",
"Sandslash",
"Nidoran",
"Nidorina",
```

```
"Nidoqueen",
"Nidoran",
"Nidorino",
"Nidoking",
"Clefairy",
"Clefable",
"Vulpix",
"Ninetales",
"Jigglypuff",
"Wigglytuff",
"Zubat",
"Golbat",
"Oddish",
"Gloom",
"Vileplume",
"Paras",
"Parasect",
"Venonat",
"Venomoth",
"Diglett",
"Dugtrio",
"Meowth",
"Persian",
"Psyduck",
"Golduck",
"Mankey",
"Primeape",
"Growlithe",
"Arcanine",
"Poliwag",
"Poliwhirl",
"Poliwrath",
"Abra",
"Kadabra",
"Alakazam",
"Machop",
"Machoke",
"Machamp",
"Bellsprout",
"Weepinbell",
"Victreebel",
"Tentacool",
"Tentacruel",
"Geodude",
"Graveler",
"Golem",
"Ponyta",
"Rapidash",
"Slowpoke",
"Slowbro",
"Magnemite",
```

```
"Magneton",
"Farfetch'd",
"Doduo",
"Dodrio",
"Seel",
"Dewgong",
"Grimer",
"Muk",
"Shellder",
"Cloyster",
"Gastly",
"Haunter",
"Gengar",
"Onix",
"Drowzee",
"Hypno",
"Krabby",
"Kingler",
"Voltorb",
]


def isprime(n):
    if n < 2:
        return False
    for i in range(2,n):
        if n % i == 0:
            return False
    return True

ret = ""
for i in range(1,101):
    if isprime(i):
        ret += elements[i-1]
    else:
        ret += pokemons[i-1]

print(ret)
```

6. Cripto

Propuesto por: Agustín Santiago Gutiérrez

El secreto es tomar la i-ésima letra de la i-ésima palabra. Es decir, la primera letra de la primera, la segunda de la segunda, la tercera de la tercera, y así siguiendo. Como las palabras pueden ser cortas, esto se toma cíclicamente: por ejemplo, si la sexta palabra tiene 5 letras, la que viene después de la quinta letra vuelve a ser la primera, así que tomaríamos la primera letra. Similarmente, si la séptima palabra tiene 5 letras, tomamos su segunda letra.

Al concatenar todas estas letras elegidas, se forma la respuesta al problema, que es:
`LARESPUESTAALPROBLEMAESESTAMISMACADENATODAENMAYUSCULAS`

7. Organizando los satélites!

Propuesto por: Juan Ignacio Cantarella, Gastón Fontenla Núñez y David Lescano

Lo primero que hay que observar es que una conexión sin cortocircuitos de nuestros satélites es equivalente a una cadena bien parenteseada. Decimos que una cadena de tamaño par que consiste solo de caracteres '(' y ')' está bien parenteseada si es posible obtener una expresión aritmética correcta insertando caracteres '1' y '+' en esta cadena. Por ejemplo "()" y "(())" son dos cadenas bien parenteseadas, mientras que ")())" no lo es.

Es conocido que la cantidad de cadenas bien parenteseadas de tamaño $2n$ es $C(n)$, donde $C(n)$ es el n-ésimo número de Catalán. A su vez, es sabido que $C(n)$ es igual a $\frac{1}{n+1}\binom{2n}{n}$.

Como la respuesta puede ser muy grande (mayor a $2^{63}-1$ para n igual a 33), necesitamos imprimir la respuesta módulo 10^9+7. Para hacer esto, necesitamos calcular $\binom{2n}{n} \bmod 10^9+7$ de forma eficiente. Esto se puede hacer con exponenciación binaria, ya que esta permite el cálculo de inversos modulares en $O(\log n)$. A continuación, dejamos el código que escribimos: `https://pastebin.com/1vZ6jwJa`

8. DOCE COFRES de IPA

Propuesto por: Facundo Martín Gutiérrez

Notar que "doce cofres de IPA" es anagrama de "API de code forces". Con esto ya podemos resolver el problema:

```
#####################################################
# SOLUCION: DOCE COFRES de IPA -> CODE FORCES API #
#####################################################
#                                                   #
#  La idea es simplemente hacer algo aunque sea   #
# bruto pero no a mano. En particular, usando la  #
#   API de Codeforces se puede resolver con el    #
#    código Python que pongo a continuación.      #
#                                                   #
#####################################################

import requests,time

def api_url():
    return 'https://codeforces.com/api/'

def user_information():
    return 'user.info?handles='

def contest_list():
    return 'contest.list?gym=false'

def contest_standings(contest_id):
    return 'contest.standings?contestId=' + str(contest_id)

def api_query(query):
    time.sleep(0.25) # Remeber, no more than 5 requests per second
    return requests.get(url = api_url() + query).json()

def get_all_contests_ids():
    data = api_query(contest_list())
    return [contest['id'] for contest in data['result']]

def get_standings_from_contest_id(contest_id):
    data = api_query(contest_standings(contest_id))
    #~ print(contest_id,data['status'])

    if data['status'] != 'FAILED': # Private contest or upcoming contest
        return data['result']['rows']
    else:
        return []

def get_users_handles_from_contest_id(contest_id):
    data = get_standings_from_contest_id(contest_id)
    return set([member['handle'] for row in data for member in row['party']['members']])
```

```
def get_all_handles(id_from,id_to):
    return list(set([handle
            for contest_id in [contest_id
                    for contest_id in get_all_contests_ids() if id_from <= contest_id <= id_to]
            for handle in get_users_handles_from_contest_id(contest_id)]))

def valid(user):
    return ('country' in user.keys()) and ('Argentina' in user['country'])


def get_users_for_problem(handle_list,chunk,problem_answer):
    for start in range(len(handle_list)//chunk+1):
        handle_chunk = handle_list[start*chunk:(start+1)*chunk]
        #~ print(handle_chunk)
        data = api_query(user_information() + ';'.join(map(str,handle_chunk)))
        if data['status'] == 'FAILED' and chunk > 1:
            get_users_for_problem(handle_chunk,max(1,chunk//2),problem_answer)
        elif data['status'] != 'FAILED':
            for user in data['result']:
                if valid(user):
                    problem_answer.add(user['handle'].lower())
                    #~ print(user['handle'])
        else:
            # User changed his name cannot find him through API ?
            print('COULD NOT FIND INFORMATION FOR USER:',handle_chunk)

def main():
    problem_answer = set()
    valid_users = get_users_for_problem(get_all_handles(1190,1199),400,problem_answer)
    ans = ''.join(sorted(list(problem_answer)))
    print(ans)
    assert(20 <= len(ans) <= 1000)
    for x in list(map(ord,ans)):
        assert(33 <= x <= 126)

if __name__ == '__main__':
    main()
```

9. Dados escalonados

Propuesto por: Agustín Santiago Gutiérrez

Se transcribe a continuación la solución enviada por Mateo Carranza.

Vamos a ver que no es posible. Supongamos que lo es, luego calculemos el valor de p. Tenemos que $p + 2p + \cdots + 11p = \frac{11 \cdot 12}{2} p = 66p = 1$, luego $p = \frac{1}{66}$.

En lugar de pensar que tenemos dados con números del 1 al 6 y que queremos formar los números del 2 al 12, pensemos que nuestros dados tienen los números del 0 al 5 y hay que formar del 0 al 10. Sean p_i, q_i las probabilidades de sacar el número i en el primer y segundo dado respectivamente. Observemos que estos números tienen que ser reales y estar en el intervalo $[0, 1]$.

Sean $P(x) = \sum_{i=0}^{5} p_i x^i$ y $Q(x) = \sum_{i=0}^{5} q_i x^i$. Luego, la probabilidad de sacar m como suma de ambos dados es $\sum_{i=0}^{m} p_i q_{m-i}$ (donde si un número es imposible de obtener con un dado entonces su probabilidad es 0), que es igual al coeficiente m del polinomio $P(x)Q(x)$. Luego $P(x)Q(x) = \frac{1}{66} \sum_{i=0}^{10} (i+1)x^i$.

Veamos que para x real, $\sum_{i=0}^{10} (i+1)x^i = (x+1)^2(1+2x^2+3x^4+4x^6+5x^8)+6x^{10} > 0$. Para $x = 0$ tenemos que el valor de esta expresión es 1, y si $x \neq 0$ entonces $6x^{10} > 0$ y $(x+1)^2(1+2x^2+3x^4+4x^6+5x^8) \geq 0$ por lo que este polinomio siempre toma valores positivos para x real y por lo tanto no tiene raíces reales.

Luego $P(x)Q(x)$ tampoco tiene, pero tanto P como Q tienen coeficientes reales y son de grado 5 (ya que $p_5 q_5 = \frac{11}{66}$ por lo que ninguno de ellos puede ser 0), por lo tanto cada uno debería tener una raíz real. Pero esto es una contradicción porque $P(x)Q(x)$ no tiene raíces reales. Luego no existen tales dados.

10. Todo esto es irracional

Propuesto por: Agustín Santiago Gutiérrez

Demostrar que números particulares son irracionales es en líneas generales algo muy difícil. El primer número que se probó que es irracional es $\sqrt{2}$ (y en general, todas las raíces de enteros que no sean justo una potencia perfecta, ya que todas se demuestran de la misma manera elemental mirando la factorización).

La irracionalidad de otros números como e, π, $\ln 2$ y demás fueron demostradas más de mil años más tarde, y utilizan prácticamente siempre técnicas analíticas mucho más avanzadas. Por lo tanto, sería ideal si podemos dar una demostración sencilla de lo que queremos que no requiera asumir la irracionalidad de números, más allá de las raíces de enteros.

10.1. Opción 1: Dos casos (Solución de Facundo Martín Gutiérrez)

La respuesta es que sí existen tales a y b. Para demostrarlo vamos a exhibir un ejemplo. Voy a usar que $\sqrt{2}$ es irracional.

Consideremos $a_1 = 2\sqrt{2}$ y $b_1 = \sqrt{2}$. Si calculamos $a_1^{b_1}$ hay dos casos:

- $a_1^{b_1} = 2^{\sqrt{2}}\sqrt{2}^{\sqrt{2}}$ es racional. Listo, la respuesta al problema es afirmativa tomando $a = a_1$ y $b = b_1$, pues son irracionales y distintos.
- $a_1^{b_1} = 2^{\sqrt{2}}\sqrt{2}^{\sqrt{2}}$ es irracional. Tomamos $a_2 = 2^{\sqrt{2}}\sqrt{2}^{\sqrt{2}}$ (enfatizo que es irracional por hipótesis en este caso) y $b_2 = \sqrt{2}$ también irracional. Entonces, $a_2^{b_2} = 2^{\sqrt{2}\sqrt{2}}\sqrt{2}^{\sqrt{2}\sqrt{2}} = 2^2\sqrt{2}^2 = 8$, que es racional. La respuesta al problema es afirmativa, tomando $a = a_2$ y $b = b_2$. Notar que $a_2 \neq b_2$, pues como $\sqrt{2} \geq 1$, $a_2 = 2^{\sqrt{2}}\sqrt{2}^{\sqrt{2}} \geq 2\sqrt{2} > \sqrt{2} = b_2$

En cualquiera de los casos, podemos encontrar a y b distintos que cumplen, así que existen.

10.2. Opción 2: Cardinalidad (Solución de Federico Felguer)

Esta solución requiere maquinaria más abstracta y avanzada, no tan elemental como la anterior (en concreto, nociones de cardinalidad), pero es elegante y simple, así que merece una mención especial aquí.

La idea es considerar el conjunto de pares $\{(a, b) : a^b = 10, \ \ 1 < a < 2 \text{ y } a \text{ es irracional } \}$

Este conjunto tiene exactamente un par por cada a irracional entre 1 y 2, ya que al ser $a^b = 10$ solo puede ser $b = \log_a 10 = \frac{\ln 10}{\ln a}$. Pero los irracionales entre 1 y 2 son un conjunto no numerable, mientras que los racionales son numerables, por lo cual alguno de todos los b involucrados debe ser necesariamente irracional. Por lo tanto, para ese b tendremos $a^b = 2$, que es racional, con ambos a y b irracionales. Finalmente, no puede ser $a = b$ porque entonces sería $a^a = 10$ pero $a^a \leq 2^2 = 4$.

11. Super mega repeticons

Propuesto por: Agustín Santiago Gutiérrez

La propiedad clave que podemos observar es que dados dos enteros $x, y \geq 2$, y es un repeticon de x si y solo si el producto $x \cdot y$ está formado todo por dígitos 9.

Esto surge de la forma de pasar de decimal a fracción que se enseña en la escuela, donde se pone el período dividido tantos 9 como tenga el período. Como ejemplo de por qué es correcto este método, si un número x es de la forma $0,123123123123\cdots$, por ejemplo, $1000x = 123,123123123\cdots$, luego $1000x - x = 123$, luego $999x = 123$ y queda $x = \frac{123}{999}$. Así que para que y sea un repeticon de x tiene que ser $\frac{1}{x} = \frac{y}{999}$ con alguna cantidad de 9 (en este ejemplo ponemos 3), y de aquí que $xy = 999$.

Esto es interesante porque si bien parece asimétrica a primera vista la condición de ser repeticon, resulta ser simétrica: si x es repeticon de y, y es repeticon de x.

Observado esto, resulta que entonces para que los números formen un super mega repeticon lo único necesario es que el producto de todos sea 9, o 99, o 999, o en general $10^n - 1$ para cierto n. Buscamos entonces el menor n tal que el número formado con esa cantidad de 9 se puede escribir como producto de 14 números diferentes, todos mayores que 1.

Notemos que, para que un número se pueda escribir como producto de 14 diferentes, en la factorización tiene que tener unos cuantos primos, o exponentes grandes. En particular, si un número tiene t factores primos contando con multiplicidad (por ejemplo, $12 = 2^2 \cdot 3$ tendría $t = 3$, pues sus factores son $2, 2, 3$), como máximo puede escribirse como producto de t números diferentes. Esto simplemente porque cada número del producto "consume" como mínimo un factor primo de la factorización. Como necesitamos 14 diferentes, para tener oportunidad tiene que ser sí o sí $t \geq 14$.

Sin más, abrimos una consola de linux y empezamos a factorizar los números con nueves, ejecutando los comandos `factor 9`, `factor 99`, `factor 999`, etc. Los resultados que vemos en la consola son los siguientes:

```
9: 3 3
99: 3 3 11
999: 3 3 3 37
9999: 3 3 11 101
99999: 3 3 41 271
999999: 3 3 3 7 11 13 37
9999999: 3 3 239 4649
99999999: 3 3 11 73 101 137
999999999: 3 3 3 3 37 333667
9999999999: 3 3 11 41 271 9091
99999999999: 3 3 21649 513239
999999999999: 3 3 3 7 11 13 37 101 9901
9999999999999: 3 3 53 79 265371653
99999999999999: 3 3 11 239 4649 909091
999999999999999: 3 3 3 31 37 41 271 2906161
9999999999999999: 3 3 11 17 73 101 137 5882353
99999999999999999: 3 3 2071723 5363222357
999999999999999999: 3 3 3 3 7 11 13 19 37 52579 333667
9999999999999999999: 3 3 111111111111111111
99999999999999999999: 3 3 11 41 101 271 3541 9091 27961
999999999999999999999: 3 3 3 37 43 239 1933 4649 10838689
9999999999999999999999: 3 3 11 11 23 4093 8779 21649 513239
99999999999999999999999: 3 3 1111111111111111111111
999999999999999999999999: 3 3 3 7 11 13 37 73 101 137 9901 99990001
9999999999999999999999999: 3 3 41 271 21401 25601 182521213001
99999999999999999999999999: 3 3 11 53 79 859 265371653 1058313049
999999999999999999999999999: 3 3 3 3 3 37 757 333667 440334654777631
9999999999999999999999999999: 3 3 11 29 101 239 281 4649 909091 121499449
99999999999999999999999999999: 3 3 3191 16763 43037 62003 77843839397
999999999999999999999999999999: 3 3 3 7 11 13 31 37 41 211 241 271 2161 9091 2906161
```

Este último es el primero que cumple que $t \geq 14$, así que es el primer candidato que tenemos. De hecho es posible obtener 14 factores en este caso, y la única forma es tomar cada primo individualmente, y el $9 = 3^2$ restante, como los factores. Todos estos números forman entonces un super mega repeticon, y es el único posible para obtener este producto mínimo de $999999999999999999999999999999 = 10^{30} - 1$.

Como hay que dar la respuesta ordenada lexicográficamente, el orden ideal es el siguiente, que constituye la respuesta final:

`3,7,9,11,13,31,37,41,211,241,271,2161,9091,2906161`