Münchausen számok hatékony keresése

Készítette:
Vitkos Bence
PTI

Mik a Münchausen számok?

A **Münchausen** szám, vagy **PDDI** (perfect digit to digit invariant), egy olyan **természetes** szám, amelyre igaz, hogy számjegyeinek önmagukkal egyenlő hatványára emelt értékéinek összege (ezt a továbbiakban **Münchausen összegnek** nevezzük), az eredeti számmal egyenlő. Minden számrendszerben értelmezettek, mi most a decimálist fogjuk vizsgálni.

Példa:
10-es számrendszerben a **3435** Münchausen szám ugyanis:
3^3+4^4+3^3+5^5 = 27+256+27+3125 = 3435
Tehát a szám Münchausen összege 3435, vagyis önmaga.

A nulla kérdése

A **Münchausen** számoknál előkerül egy érdekes művelet, minden számjegyet magával egyenlő hatványra kell emelni, így a nullára végre kell hajtani a 0^0 műveletet.

Matematikában a 0^0–nak nincs egyértelmű eredménye, különböző ágazatok különbözően értelmezik. Eredménye lehet 0 vagy 1, de olyan matematikai ágazat is van amiben nincs értelmezve a művelet.

Mi most a **0^0 = 0** értelmezést fogjuk használni

A feladat

Maga a feladat az, hogy kódot írjunk ami képes 10-es számrendszerben, 0^0=0 értelmezéssel megtalálni az összes Münchausen számot. (a feladat megadja, hogy a legnagyobb, 440 milliónál kisebb)

A triviális megoldás:

```
def main():
    for i in range(440_000_000):
        munchausen_sum = 0
        for d in str(i):
            n = int(d)
            if(n != 0):
                munchausen_sum += n**n
        if(munchausen_sum == i):
            print("Munchausen number found:",i)
```

A triviális megoldás hatékonysága

A triviális megoldásban egyszerűen végig megyünk minden természetes számon 440 millióig, vesszük egyesével számjegyeiket egy string és utána több int cast-olással, hatványozzuk ezen számokat, összeadjuk őket és hasonlítjuk az eredeti számhoz az összeget.

Ennek költsége:

- 440 millió int készítése egy range-ből
- 440 millió string cast
- 3 848 888 890 int cast
- 3 848 888 890 hatványozás
- 440 millió int hasonlítás

Van ezen mit javítani…

Optimalizáció: hatványozás

A hatványozás egy költséges művelet, így ajánlatos lenne kiküszöbölni. Mivel csupán 10 különböző hatványozást végzünk (ebből a 0^0-t valójában el se végezzük), érdemes lenne ezeket előre elvégezni, tárolni az eredményeket, majd csak ezekre hivatkozni.

```python
def main():
    powers_list = [0]+[i**i for i in range(1,10)]
    for i in range(440_000_000):
        munchausen_sum = 0
        for d in str(i):
            n = int(d)
            munchausen_sum += powers_list[n]
        if(munchausen_sum == i):
            print("Munchausen number found:",i)
```

Az optimalizáció hatékonysága

Listát generálunk a 10 lehetséges hatványozáshoz, ami elenyésző időbe kerül. Ezután az összes hatványozás helyett ami már az 5-ös számjegy esetén is 5 darab szorzást jelentene, egyetlen darab számot kell a listából kiolvasni.

Költség változása:

- 440 millió int készítése egy range-ből
- 440 millió string cast
- 3 848 888 890 int cast
- 3 848 888 890 hatványozás -> 3 848 888 890 listaelem elérés
- 440 millió int hasonlítás

Optimalizáció: castolás

A castolás sok nyelvben költséges, és képességeinek csak egy töredéke kell nekünk: 10 darab egy számjegyet tartalmazó string castolása újra és újra. Általánosságban két opció van ennek a kikerülésére: A karakterek kódjának lekérése majd ebből egy olyan x kivonása, hogy a számot kapjuk meg, vagy egy Map (Dict) használata, hogy string-int párokat adjunk meg. Pythonban a legjobb megoldás:

```
def main():
    powers_list = [0] + [i**i for i in range(1,10)]
    cast_dict = {"0": 0}
    for i in range(1,10):
        cast_dict[str(i)] = i
    for i in range (440*(10**6)):
        munchausen_sum = 0
        for d in str(i):
            munchausen_sum += powers_list[cast_dict[d]]
        if(munchausen_sum == i):
            print("Munchausen number found:",i)
```

Az optimalizáció hatékonysága

A castolás ez esetben mindenképp a legrosszabb lehetőség, egyszerűen nem 1 karakterre, hanem egy egész string-re van tervezve. A kód használata (ord) jobb, sok nyelvben valószínűleg ez nyerne. Gondolhatunk még esetleg egy olyan dict-re is ami a string számjegyeket, direkt a hatványaikhoz párosítja, de a hash-elést a tárolt elemek mérete és vagy távolsága lelassítja. Így a helyes megoldás: kis dict a castolás helyett és a hatványlista használata.

Költség változása:

- 440 millió int készítése egy range-ből
- 440 millió string cast
- 3 848 888 890 int cast -> 3 848 888 890 dict elem elérése
- 3 848 888 890 listaelem elérés
- 440 millió int hasonlítás

Optimalizáció: azonos összegek

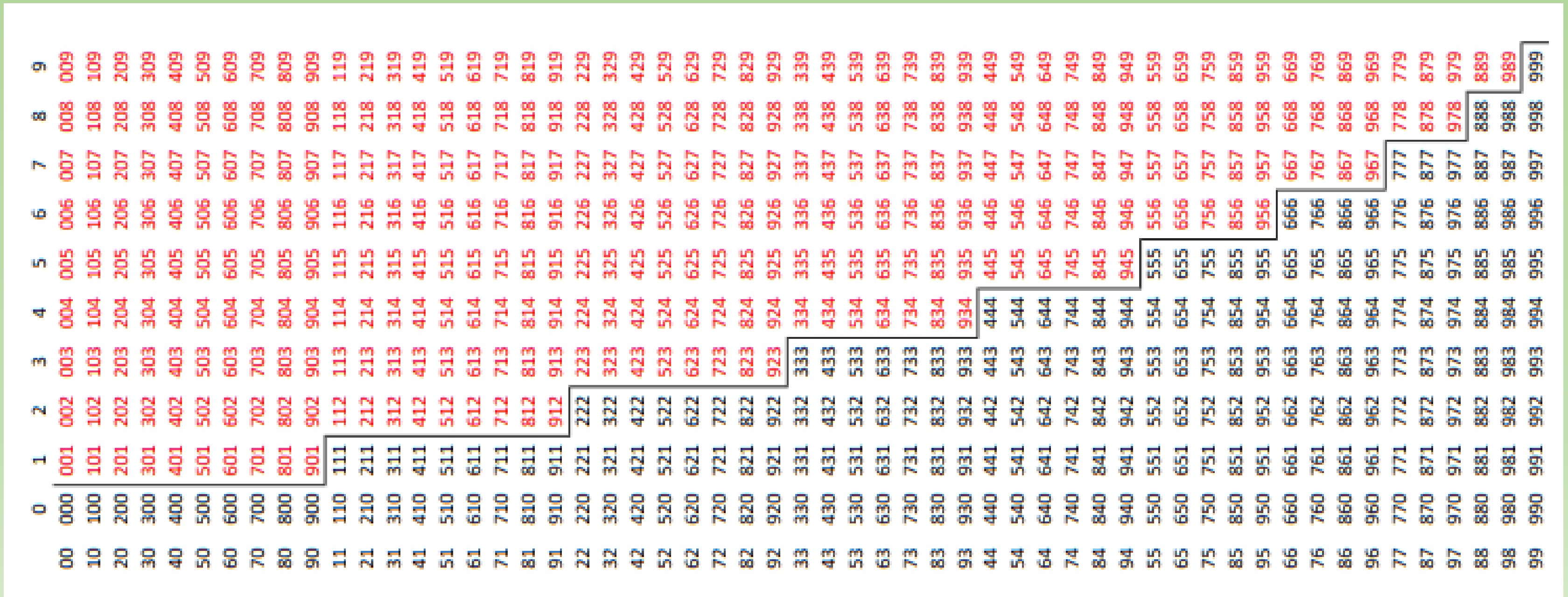
Beláthatjuk azt, hogy ha egy szám Münchausen szám, akkor Münchausen összeg is. Egy szám Münchausen összegére pedig igaz, hogy azonos számjegyekből álló számoknál azonos, a számjegyek sorrendjétől függetlenül. Ezért minden szám bejárása helyett, járjuk be csak a különböző számjegy kombinációkat. Ha minden kombinációt csak egyszer nézünk, az 2 számjegynél a következő módon néz ki:

	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	10	11	12	13	14	15	16	17	18	19
2	20	21	22	23	24	25	26	27	28	29
3	30	31	32	33	34	35	36	37	38	39
4	40	41	42	43	44	45	46	47	48	49
5	50	51	52	53	54	55	56	57	58	59
6	60	61	62	63	64	65	66	67	68	69
7	70	71	72	73	74	75	76	77	78	79
8	80	81	82	83	84	85	86	87	88	89
9	90	91	92	93	94	95	96	97	98	99

Fekete:
Egyedi kombináció
Piros:
Kihagyott duplikátum

Optimalizáció: azonos összegek

A hatékonyság több számjegynél egyre jobban nő, egy előzőhöz hasonló szemléltetés 3 számjegyre így néz ki. Mint látható, az előzőnél a kombinációk kevesebb mint felét, itt pedig már több mint a felét hagyjuk el.



Optimalizáció: azonos összegek

Megvalósítás

Először fel kell venni az egyjegyű számok számjegyei négyzetének összegeit, ugyanis minden új összeget a korábbi összegekre fogunk építeni. Ennek a kezdőértéke azonos a hatványok listájával.

```
prev_sums = powers_list.copy()
```

Ezeket a fő program loop nem fogja vizsgálni, ezt külön tesszük meg

```
for munchausen_sum in prev_sums:
    testsum = 0
    for char in str(munchausen_sum):
        testsum+=powers_list[cast_dict[char]]
    if(munchausen_sum == testsum):
        print(munchausen_sum)
```

Optimalizáció: azonos összegek

Megvalósítás

Vegyünk fel egy listát amely azt fogja tartalmazni, hogy adott számjegy esetén mely előző összegeket nem kell már növelni, a duplikátumok elkerülése végett. Ennek kezdőértékei:

```
duplicate_avoidance = [0,1,2,3,4,5,6,7,8,9]
```

Vegyünk továbbá még fel egy változót, amivel azt vizsgálhatjuk, hogy egy összeg kevesebb számjegyű-e mint az őt generáló szám, mert ez esetben vizsgálni se kell.

```
min_sum = 9
```

Optimalizáció: azonos összegek

Megvalósítás

A fő loop a számjegyek mennyiségén iterál, ez legyen [2;10[:

```
for n in range(2,10):
```

Ezen belül kell két segédváltozó, először rögzítsük, hogy jelenleg mennyi a korábbi összegek száma:

```
sums_to_increase = len(prev_sums)
```

Ezután pedig rögzítjük, hogy mennyi szám lett „nullával növelve”, mert ezeket következő körben nem növeljük semmivel, duplikátumok elkerülése végett.

```
next_dup_avoid = len(prev_sums)
```

Optimalizáció: azonos összegek

Megvalósítás

A loop lényege, az új összegek generálása, és azonnali ellenőrzése:

```
for i in range(1,10):
    for j in range(duplicate_avoidance[i],sums_to_increase):
        munchausen_sum = prev_sums[j]+powers_list[i]
        if(min_sum<munchausen_sum and 440_000_000>munchausen_sum):
            testsum = 0
            for char in str(munchausen_sum):
                testsum+=powers_list[cast_dict[char]]
            if(munchausen_sum == testsum):
                print(munchausen_sum)
        prev_sums.append(munchausen_sum)
    duplicate_avoidance[i],next_dup_avoid = next_dup_avoid,next_dup_
```

Optimalizáció: azonos összegek

Megvalósítás

1-9 hatványaival növeljük az előző összegeket.

```
for i in range(1,10):
```

Minden számjegynél csak azon előző összegeket növeljük amiket a duplikátum kerülés enged, illetve már a jelenlegi ciklusban is léteztek.

```
for j in range(duplicate_avoidance[i],sums_to_increase):
```

Előállítjuk az összeget egy előző összeg és a jelenlegi számjegy hatványának összegeként.

```
munchausen_sum = prev_sums[j]+powers_list[i]
```

Optimalizáció: azonos összegek

Megvalósítás

Ellenőrizzük, hogy az összeg megfelelő számjegyű, és 440 millió alatt van-e. Ha igen, ellenőrizzük Münchausen szám-e.

```
if(min_sum<munchausen_sum and 440_000_000>munchausen_sum):
    testsum = 0
    for char in str(munchausen_sum):
        testsum+=powers_list[cast_dict[char]]
    if(munchausen_sum == testsum):
        print(munchausen_sum)
```

A feltételektől függetlenül hozzáadjuk az előző összegek listájához.

```
prev_sums.append(munchausen_sum)
```

Optimalizáció: azonos összegek

Megvalósítás

Rakjuk a következő körre már kiszámolt duplikátum elkerülési pontot a listába, és számoljuk ki az azt követőt (mostani pont + mennyi korábbi összeg lett növelve a vizsgált számjeggyel.

```
duplicate_avoidance[i],next_dup_avoid =
```

```
= next_dup_avoid,next_dup_avoid+sums_to_increase-duplicate_avoidance[i]
```

Ezzel vége az új összegeket előállító loopnak. A ciklus végén már csak növelni kell a számot amely megadja a lekisebb megfelelő számjegyű szám előtti számot a következő ciklusra

```
min_sum = (min_sum+1)*10-1
```

Optimalizáció hatékonysága

Ahogy korábban szemléltettem, ezzel a módszerrel minél több számot vizsgálunk, annál kisebb részét kell ténylegesen megvizsgálni, hisz annál több szám áll azonos számjegyekből.

A hatékonyság a következő módon alakul:

100:	55
1 000:	220
10 000:	715
100 000:	2002
1 000 000:	5005
10 000 000:	11440
100 000 000:	24310
1 000 000 000:	48620

Ez 1 milliárd számnál kb. 20000-szer hatékonyabb!

Optimalizáció hatékonysága

Az előbb említett hatékonyság jól látható a program futási idején is:

```
The Munchausen numbers in decimal:
0
1
3435
438579088
Execution time in seconds: 0.0305984020233l543
```

Az eredeti futás ehhez képes több tucat percet is igénybe vett!

Köszönöm a figyelmet!