SPD - Exercici 2: Teoria de números / aritmètica modular

Autor: Francesc Xavier Bullich Parra

A: Càlcul del MCD amb algoritme mcd_euclides

Veure mcd_euclides a utils.py Veure fitxer 1_euclides.py

L'algoritme d'euclides és molt simple. Es va dividint el dividend pel divisor i es compara el residu amb 0.

- Si el residu és 0 llavors el resultat del mcd es el divisor actual.
- Si el residu és diferent de 0 llavors el divisor passa a ser el divident, i el residu passa a ser el proper divisor.

Utilitzant la funcio mcd_input veiem alguns exemples de funcionament:

```
python 1_euclides.py
entreu un dividend natural positiu: 48
entreu un divisor natural positiu: 36
Dividend:  48
Divisor:  36
MCD:  12
2
Temps invertit: 0.0000202000 segons
```

```
python 1_euclides.py
entreu un dividend natural positiu: 5158215025
entreu un divisor natural positiu: 15684911025
Dividend:  5158215025
Divisor:  15684911025
MCD:  25
Temps invertit: 0.0000244000 segons
```

Prova de complexitat nombre de digits/temps_final

Per provar la complexitat d'aquest mètode vaig provant de buscar el MCD de diferents nombres generats aleatoriament. Es realitzen varies iteracións en la que cada una duplica el nombre de digits de l'anterior. Començant per 2 dígits i fins uns 160.000 dígits. Veure funció mcd_rep del fitxer 1_euclides.py.

Es mostra el gràfic resultant:

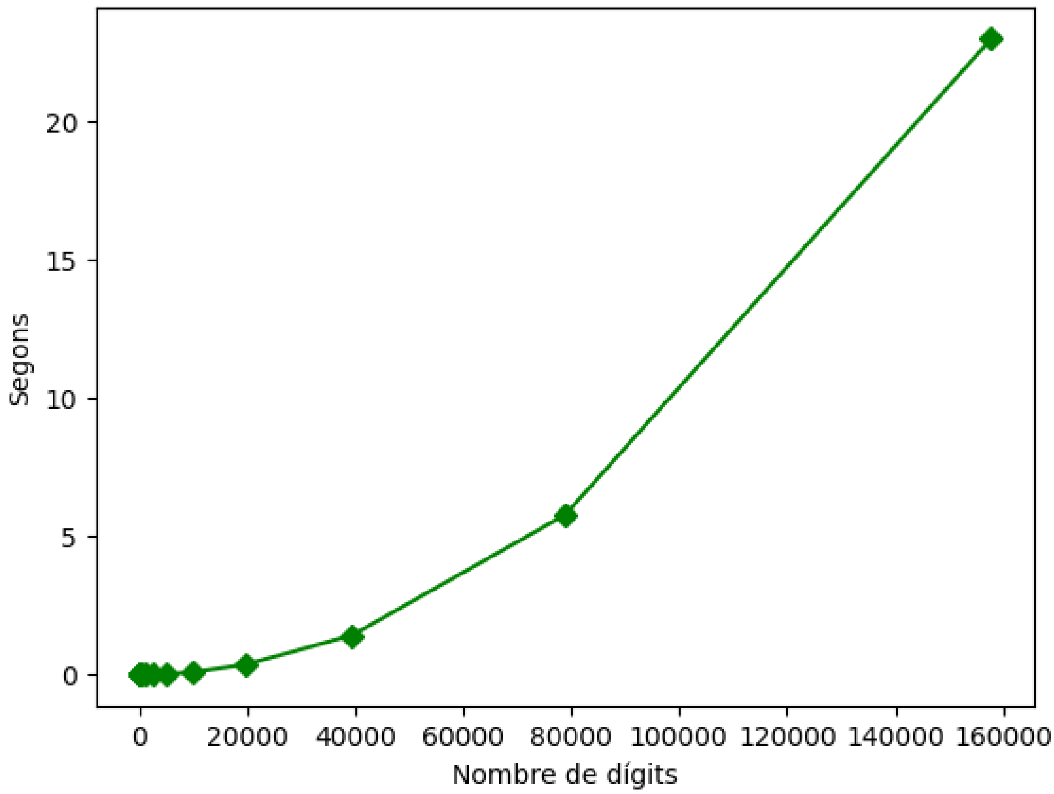

Figure 1: Grafic mcd

Les dades més significatives son: - n = 40.000, t = 1,2 - n = 80.000, t = 5,3 - n = 160.000, t = 22,11

Es pot observar que per cada cop que es duplica el nombre de digits la relació en temps creix mes o menys en un factor de 4. Podem determinar que la complexitat d'aquest métode entra dins de la complexitat polinòmica.

B: Càlcul del MCD amb l'algoritme de la identitat de Bezoud.

Veure bezoud a utils.py Veure fitxer 2_bezoud.py

La identitat de Bezoud determina que el MCD de 2 valors es pot expressar com a una combinació lineal d'enters. Ens diu que existeixen 2 nombres enters que satisfant la formula MCD = $rdividend + s$divisor

Si dividend i divisor tenen un MCD diferent de 1 aquests números seran 1 i -1 o viceversa. Altrament ens donarà uns nombres que podem utilitzar posteriorment per el calcul d'operacions d'aritmètica modular. L'algoritme utilitza el mateix esquema que el d'Euclides, afegint més variables que controlen els valors de r i s. Per tant pel mateix preu d'executar l'algoritme d'euclides podrem trobar els nombres de l'identitat de Bezoud.

r i s S'obtenen amb una finestra de 2 nombres anteriors que comencen

- rAnt = 0, rAnt2 = 1
- sAnt = 1, sAnt2 = 0

s actual s'obte del quocient de la divisió actual –> quocient * sAnt + sAnt2 r actual s'obte del quocient de la divisió actual –> quocient * rAnt + rAnt2

També s'ha de tenir en compte el nombre de la iteració actual. Si es parell s sera negatiu, si es senar r sera negatiu.

Prova d'execució de l'algoritme de Bezoud:

```
python 2_bezout.py
entreu un dividend natural positiu: 48
entreu un divisor natural positiu: 36
Dividend:  48
Divisor:  36
MCD: mcd = 12, r = 1, s=-1
Comprovacio 12 = 1 * 48 + -1*36
Temps invertit: 0.0000262000 segons
```

```
python 2_bezoud.py
entreu un dividend natural positiu: 508
entreu un divisor natural positiu: 355
Dividend:  508
Divisor:  355
MCD: mcd = 1, r = -58, s=83
Comprovacio 1 = -58 * 508 + 83*355
Temps invertit: 0.0000811000 segons
```

Complexitat digits/temps

Com he comentat l'algoritme de Bezout segueix el mateix esquema que l'algoritme d'euclides per el que la gràfica serà similar al del mètode anterior. Per tant es pot dir que la complexitat també és polinomica.

Test de primalitat amb algorimte fins sqr(n)

Veure fitxer 3_test_primalitat.py

Com hem vist a classe si comparem amb la xifra sqrt(n) podriem deduir que aquest algorimte podria ser polinomic, pero si ho mirem per nombre de bits resulta que la complexitat és exponencial.

Veiem algun exemple

```
python 3_test_primalitat.py
Provant si 1000000000100011 es primer
True
15.3277976 segons
```

Complexitat digits/temps

Com podem observar en el seguent gràfic, per a uns 9 digits el temps que tarda es prou bo. A la que tenim nombres una mica més grans el test de primalitat es dispara en el temps. Com s'aprecia en la prova anterior de 16 dígits ja arriba a tardar uns 15 segons. Això es 75 cops més que el de 12 xifres. Per tant podem concluir que efectivament el test de primalitat té una complexitat exponencial.

En aquest cas per provar la complexitat es generen nombres aleatoriament i es van provant. Si no son primers no es guarda el temps que ha tardat i es prova amb un altre nombre random del mateix nombre de digits. Així podem trobar sempre el pitjor cas.

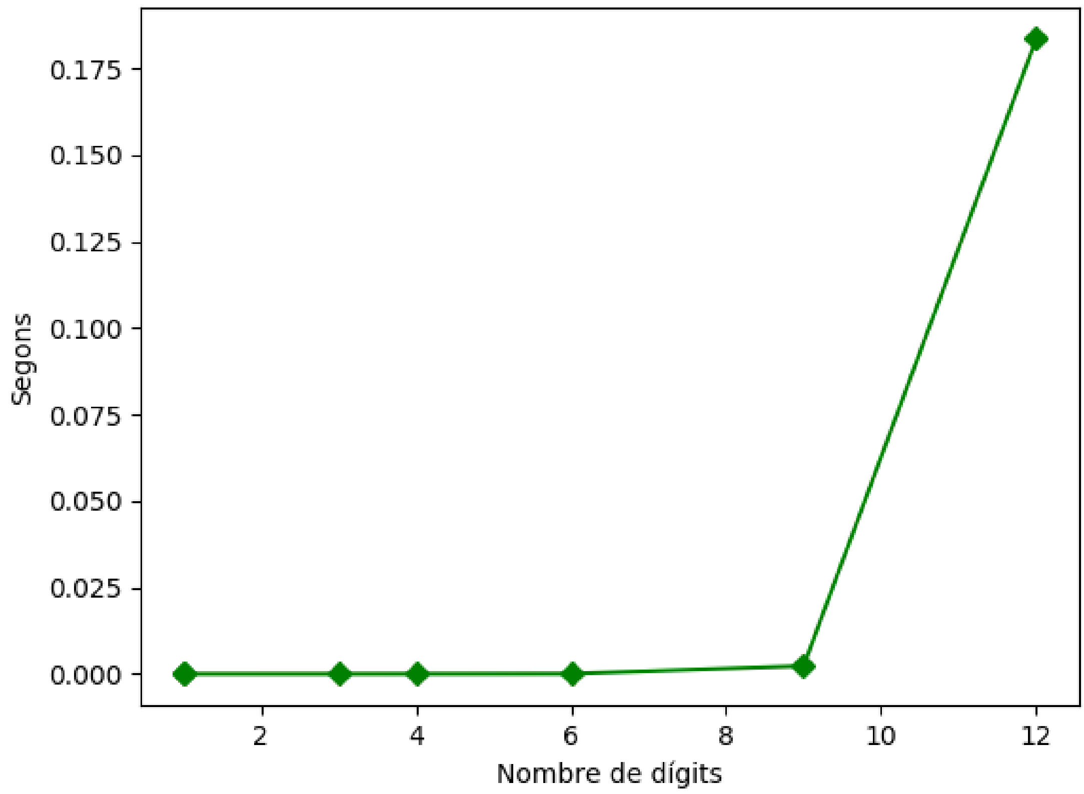

Figure 2: Grafic test primalitat

```
pandoc README.md -o README.pdf --from markdown --template eisvogel --listings
```