Treball de Recerca:
Algoritmes d'ordenació

Joan Coma Bages
Tutor: Javier Buj

Curs 2020/2021
IES Gelida

Índex

Pròleg **3**

1 Introducció **6**

2 Què és ordenar un vector? **7**

2.1 Què podem ordenar? 7

2.2 Per què ordenem? 8

I Algoritmes iteratius **9**

3 Selection sort **10**

3.1 Com funciona? 10

3.2 Implementació 10

3.3 Rendiment 11

4 Insertion sort **12**

4.1 Com funciona? 12

4.2 Implementació 12

4.3 Rendiment 13

5 Bubble sort **14**

5.1 Com funciona? 14

5.2 Implementació 14

5.3 Rendiment 15

6 Shell sort **16**

6.1 Com funciona? 16

6.2 Implementació 16

6.3 Rendiment 18

II Algoritmes recursius **19**

7 Recursivitat **20**
7.1 Factorials 20
7.1.1 Implementació no-recursiva 20
7.1.2 Implementació recursiva 21
7.2 Successió de Fibonacci 21
7.2.1 Implementació no-recursiva 21
7.2.2 Implementació recursiva 22
7.3 Torres de Hanoi 22
7.3.1 Implementació recursiva 24

8 Quick sort **26**
8.1 Com funciona? 26
8.2 Implementació 26
8.3 Rendiment 27

9 Merge sort **28**
9.1 Com funciona? 28
9.2 Implementació 28
9.3 Rendiment 30

III Anàlisi i conclusions **31**

10 Comparació dels algoritmes **32**
10.1 Cost Computacional 32
10.1.1 Fórmula 32
10.1.2 Càlcul 34

11 Conclusions **37**

Agraïments **38**

Bibliografia **39**

Pròleg

Estimat fill meu...

Sóc filòleg de formació, amb set anys d'experiència com a docent de secundària, i pagès de vocació, amb 18 anys treballant com a viticultor en una finca familiar agrària. Els mateixos anys que té el meu primer fill, el Joan. Als ulls del qual sempre seré un boomer.

I és que jo, a la seva edat (una expressió que m'avergonyeix escriure perquè pensava que jo mai la utilitzaria per comparar-me amb els meus fills, com van fer els meus pares amb mi) feia els treballs amb una Hispano Olivetti. Quan la tinta s'acabava, aixecaves la tapa i amb el dit feies girar la bobina fins a fer passar tota la cinta d'una banda a l'altra del carret. Si volies afegir alguna imatge als teus treballs, compraves revistes, les retallaves amb tisores i les enganxaves amb Imedio, aprofitant aquell moment que el destapaves per fer unes quantes inspiracions fortes que afegissin una mica d'emoció a les nostres vides, abans de fer forats amb la perforadora i posar un modern fastener o unes clàssiques anelles.

Segurament m'hauria pogut imaginar molt abans que estàvem en dues dimensions irreconciliables, quan amb pocs anys, ja em solucionava els problemes que tenia amb el mòbil. Quan veia que m'acostava massa la pantalla als ulls, em preguntava -Vols que et posi la lletra més gran?. Jo no sabia ni tan sols que es pogués fer. O quan em queixava que no m'hi cabien més fotos i que havia de començar a esborrar-ne em deia, per a gran esgarrifança i temor meu, -Per què les guardes al telèfon? Posa-les al núvol!. I així va seguir. Ja no sabem canviar-nos el mòbil si no el configura ell. No podem veure la televisió si es desconfigura el mòdem de la fibra. Allò de picar la tele o dir que és culpa d'ells està molt mal vist per les noves generacions. Fins i tot la Thermomix avui s'ha de configurar amb la clau wifi, perquè t'actualitzi les receptes.

Sóc usuari actiu de les xarxes. Tinc Facebook (com totes les persones de provecta edat), Twitter, fins i tot, Instagram. A TikTok no hi he entrat, per ara. Diàriament utilitzo el correu electrònic, el Whatsapp i, per si de cas és insegur o ens espia, com asseguren de tant en tant els mems que

rebo, ja m'he descarregat Telegram i Signal, preventivament, per no quedar-me incomunicat. Pràcticament no poso els peus a la meva entitat bancària perquè faig tota l'operativa on-line i ja he entès, que encara que no tingui els diners en l'apunt bancari d'una llibreta, els meus diners són igual de volàtils. I em declaro un analfabet digital.

Estic envoltat i utilitzo una tecnologia que no entenc. I quan he fet algun esforç per entendre-la m'he sentit desplaçat. Crec que la tecnologia i jo ja estem amortitzats mútuament. Ella em facilita uns serveis i jo els consumeixo. És un pacte en el qual tots dos tenim allò que volem. Ella un consumidor submís i jo unes comoditats que em fan creure que puc ser original escrivint un missatge de 140 caràcters i un fotògraf artístic retocant una imatge. Però que no falli res, perquè si surt algun missatge que no m'espero, he de fer venir el Joan per preguntar-li què carai significa.

I aleshores ho soluciona. I ja no prova d'explicar-m'ho perquè la generació Z i els boomers som com l'oli i l'aigua. Però, com a pare, no puc renunciar al mínim control parental que se'ns suposa, que segur que s'ha recollit en els drets de la infància o en qualsevol declaració similar de les Nacions Unides. I miro què fa. Sense entendre res. Perquè allà on jo hi veia un full en blanc, com el que posava a la Hispano Olivetti, i que omplo amb lletres amb la màgica possibilitat d'esborrar-les, corregir-les, canviar-les... ell fa aparèixer un llenguatge ocult, ple de caràcters impossibles i, per descomptat, illegibles. És com un subtext. Com si caigués el decorat d'un teatre i quedés la tramoia al descobert, amb operaris fumant i efectes especials sense glamur. Com si a un cotxe nou i llampant, li traiessis la carrosseria i quedés la mecànica al descobert. Amb la diferència, que aquí la tramoia que aguanta la cosa, la mecànica que fa córrer la màquina, és un llenguatge.

I aquest llenguatge subjacent a les aplicacions, també té un llenguatge que fa les aplicacions interactuïn amb el seu suport i amb altres aplicacions. I per això calen els algoritmes: un conjunt d'instruccions per resoldre un problema o executar una ordre. En aquests algoritmes és on la tecnologia es juga bona part de la seva raó de ser. I jo, sense entendre-ho, sento una admiració profunda pel meu fill que és capaç de fer un treball explicant-ho. Un treball sobre fórmules matemàtiques, àlgebres i números, que són presents

en la nostra quotidianitat i que preferim ignorar però, com fa el Neo a Matrix, el Joan s'ha atrevit a prendre's la pastilla vermella per posar-lo al descobert.

És així, com es passa pàgina i s'avança. Quan la generació que et succeeix és millor que la teva. Enhorabona Joan per la capacitat d'anar més enllà i qüestionar-t'ho tot.

Gil Coma Masana
Pare del Joan

1. Introducció

Quan algú em pregunta de què va el meu TREC i responc amb "Algoritmes d'ordenació" veig davant meu cares d'indiferència o, en el millor dels casos, confusió. No els culpo: quan el Xavi, el meu tutor, em va proposar el tema vaig reaccionar de la mateixa manera. Això va canviar quan vaig començar a indagar, sentia com es despertava dins meu una curiositat cada vegada més intensa. No cal dir que vaig acabar acceptant la seva proposta.

L'objectiu que em plantejo per aquest treball és entendre per tal de poder analitzar i, en última instància, comparar el rendiment de diversos algoritmes d'ordenació.

El document s'estructura en tres grans blocs: dos primers d'algoritmes, uns recursius i uns altres iteratius, on cada algoritme té el seu apartat amb tres seccions: explicació del funcionament, implementació i finalment els temps d'execució. Tots aquests temps d'execució tornen a aparèixer al tercer bloc, on s'analitzen i es comparen.

Els algortimes iteratius analitzats són el *selection sort*, l'*insertion sort*, el *bubble sort* i el *shell sort*. De recursius n'hi ha un parell: *quick sort* i *merge sort*. N'hi ha molts més, tant de recursius com d'iteratius, però tard o d'hora s'ha de posar un límit. Els algoritmes escollits es troben entre els més coneguts i mostren la diversitat del camp, diversitat tant en la manera d'enfocar el problema com en els resultats obtinguts, però la tria no deixa de ser arbitrària.

2. Què és ordenar un vector?

Un vector és un seguit d'elements en un ordre qualsevol. Quan n'ordenem un, estem canviant de posició els elements que conté per tal que estiguin ordenats d'una manera determinada.

En aquest treball els elements a ordenar són números, que ordenarem de forma ascendent.

2.1. Què podem ordenar?

No podem ordenar elements sense una referència que ens indiqui quin va primer. Això ho veiem en números i lletres: primer va l'1 i el segueix el 2, comença la A i a continuació la B.

Si volem ordenar hortalisses, no existeix cap sistema com amb els números o les lletres i se'ns plantegen dues opcions:

a) inventar-nos un sistema de referència: primer les cols, després les pastanagues, i finalment els espinacs o

b) buscar una propietat comuna en totes les nostres verdures i ordenar-les en funció d'aquesta (alfabèticament pel nom, de menor a major pes, etc.)

Ordenem el que ordenem el procediment és el mateix, i els mètodes aquí utilitzats per ordenar números també són vàlids per ordenar paraules o hortalisses.

2.2. Per què ordenem?

Ordenem números pel mateix motiu que ordenem l'armari, l'habitació o la casa sencera: el temps que dediquem ara a ordenar el recuperem a l'hora de buscar.

Si guardem totes les garanties dels electrodomèstics en una carpeta, no patirem (tant) quan se n'espatlli un i necessitem recuperar-les.

L'índex d'aquest document permet navegar-lo amb facilitat, però no seria tan útil si les pàgines no estiguessin ordenades. Si aquesta pàgina és la 8, l'anterior fos la 22 i la següent fos la 3 veuríem que la numeració dels fulls faria més nosa que servei.

De la mateixa manera agraïm que els diccionaris (en paper) ordenin les paraules alfabèticament, si ho fessin aleatòriament vendrien pocs exemplars.

Part I

Algoritmes iteratius

3. Selection sort

3.1. Com funciona?

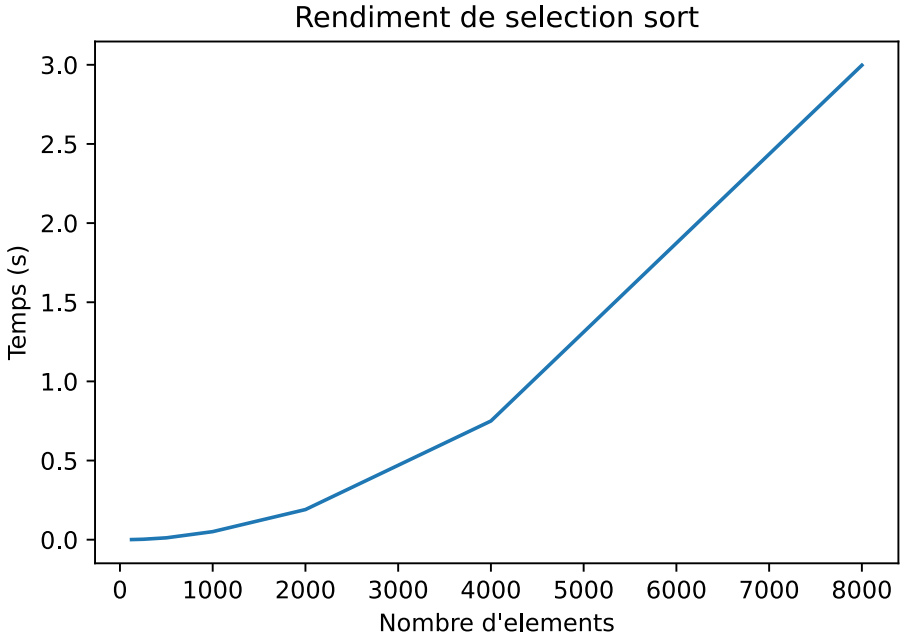
Aquest algoritme itera sobre la llista buscant el nombre més baix, que col·loca al principi. A continuació repeteix el mateix procés buscant el segon nombre més baix, però no des de l'inici sinó des de la segona posició (sabem que en la primera hi trobem el nombre més baix). Aquest procediment es repeteix fins a ordenar tota la llista.

3.2. Implementació

```
#!/usr/bin/env python3
import utils

def sort(array):
    for i in range(len(array[:-1])):
        low = i
        for j in range(i, len(array)):
            if array[low] > array[j]:
                low = j

        array[i], array[low] = array[low], array[i]
    return array

if __name__ == "__main__":
    array = utils.numbers()
    print(sort(array))
```

3.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0007	0.0027	0.0118	0.0508	0.1904	0.7498	2.9973



4. Insertion sort

4.1. Com funciona?

L'*insertion sort* itera sobre tots els elements de la llista i els va comparant amb els anteriors de manera que els elements a la dreta de l'actual quedin ordenats.

Pas a pas això vol dir agafar d'entrada un sol element, el primer, que compararem amb els anteriors. En ser el primer element no n'hi ha d'anteriors, el considerarem ordenat. En la següent iteració agafem el segon element, aquest el compararem progressivament amb els anteriors. El nostre element, que es troba encara en la segona posició, el comparem amb l'anterior. Si l'anterior és major, el nostre passarà davant d'aquest. Si el nostre és menor, quedarà darrere del primer. En les següents iteracions repetim: anem comparant amb els anteriors fins que trobem un element menor al nostre. Aleshores col·locarem el nostre element a continuació del menor. Si després de comparar-lo amb tota la resta no en trobéssim un de menor el deixaríem al principi de la llista.

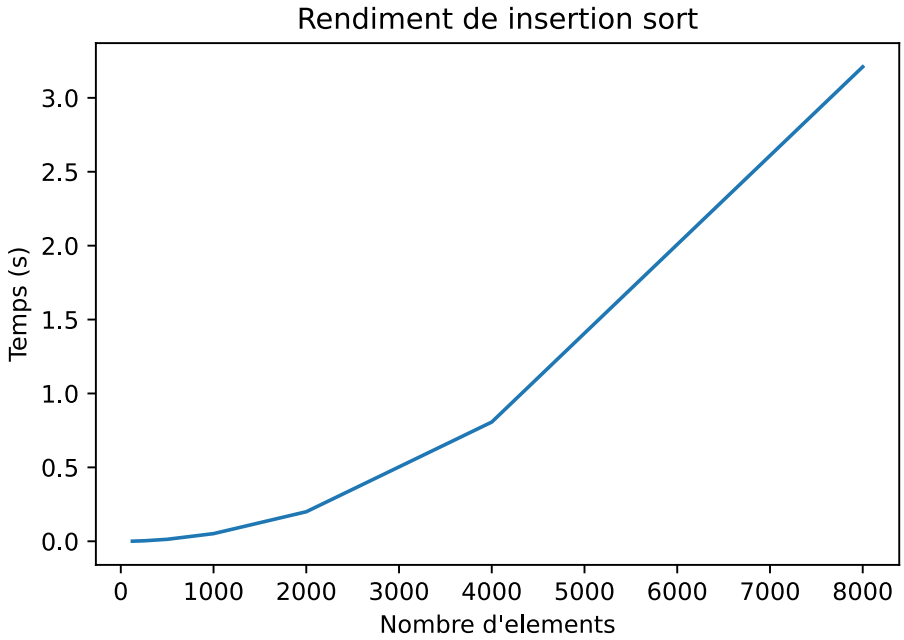
4.2. Implementació

```
#!/usr/bin/env python3
import utils

def sort(array):
    for i in range(len(array)):
        comp = array[i]
        j = i - 1
        while j >= 0 and array[j] > comp:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = comp
    return array

if __name__ == "__main__":
    array = utils.numbers()
```

```
16      print(sort(array))
```

4.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0007	0.0033	0.0134	0.0518	0.1999	0.8066	3.2098



5. Bubble sort

5.1. Com funciona?

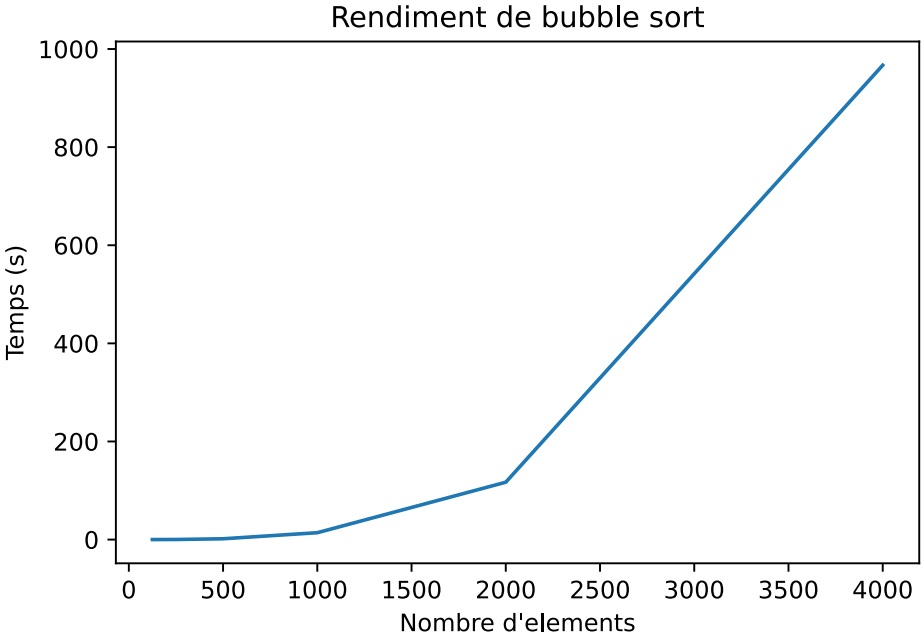
El *bubble sort* compara cada element de la llista amb el següent (d'esquerra a dreta), canviant-ne la posició si el segon és més gran que el primer. Aquest procés es repeteix tantes vegades com faci falta per ordenar la llista. Sempre movem endavant l'element més gran de manera que, progressivament, al final del vector queden ordenats els elements més grans. El nom *bubble sort* li és donat precisament per com els elements més grans floten cap a la superfície, com si es tractés de bombolles. És un algoritme extremadament lent.

5.2. Implementació

```
1  #!/usr/bin/env python3
2  import utils
3
4  def sort(array):
5      while True:
6          for i in range(len(array[:-1])):
7              if array[i] > array[i+1]:
8                  array[i], array[i+1] = array[i+1], array[i]
9                  break
10
11         # aquest else s'executa si el bloc anterior s'ha
       executat sense cap break, és a dir, si mai s'ha complert
       la condició array[i] > array[i+1]
12         else:
13             break
14     return array
15
16 if __name__ == "__main__":
17     array = utils.numbers()
18     print(sort(array))
```

5.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0283	0.2121	1.6764	14.0336	116.9414	966.9290	(-)



6. Shell sort

6.1. Com funciona?

L'ordenació *shell* és un algoritme que es pot considerar una millora de l'*insertion sort*. Recordem que aquest algoritme anava iterant sobre tots els elements i en cada iteració comparava l'element amb els anteriors, fent que tots els elements ja iterats estiguessin en ordre.

És un bon algoritme, però què passaria si trobéssim l'element més gran de tots en la primera posició de la llista? Com que tots els elements restants han de passar davant d'aquest estarem fent moltes comparacions que ens podríem estalviar si aquest element màxim es trobés més endavant.

El *shell sort* aspira a mitigar aquest problema introduint unes quantes iteracions a l'inici de l'algoritme que no comparen l'element amb el següent sinó que ho fan amb un que es troba més enllà. Això permet aproximar a les seves posicions pertinents els elements que se'n trobin lluny.

Si recuperem l'exemple del major element en primera posició el *shell sort* aconsegueix en la primera iteració és situar-lo al bell mig del vector. En la següent iteració el desplaçarà fins a tres quartes parts endins, i així successivament.

Generalitzant això, el *shell sort* comença agafant el primer element, i el compara no amb el segon sinó amb l'enèsim. El que sigui més petit ocupa la posició del primer i continua així amb la resta d'elements fins que arriba al final de la llista. Aleshores torna a començar però fent els salts entre el primer i l'enèsim element cada vegada més petits. L'última iteració es fa amb salts d'u, comparant tots els elements un rere l'altre, tal com ho fa l'*insertion sort*.

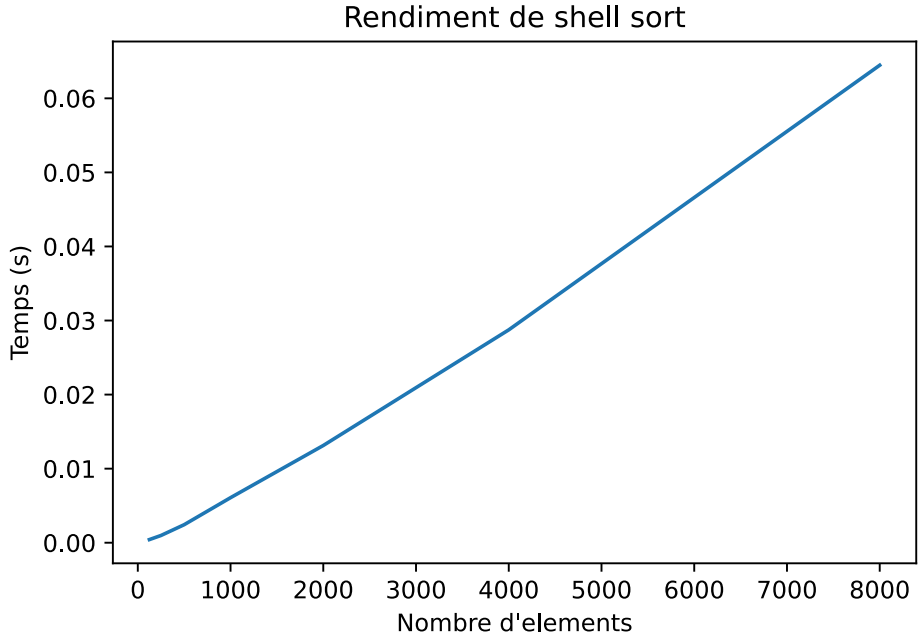
6.2. Implementació

```
1 #!/usr/bin/env python3
2 import utils
3 import insertion
```

```
4
5  def sort(array):
6      step = len(array) // 2
7      while step != 0:
8          for offset in range(step):
9              sorted = insertion.sort(array[offset::step])
10             for i in range(len(sorted)):
11                 array[step*i + offset] = sorted[i]
12         step //= 2
13     return sorted
14
15 if __name__ == "__main__":
16     array = utils.numbers()
17     print(sort(array))
```

6.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0004	0.0010	0.0024	0.0061	0.0131	0.0287	0.0645



Part II

Algoritmes recursius

7. Recursivitat

La recursivitat, en informàtica, és la propietat dels programes que en executar-se es criden a si mateixos.

Problemes que poden ser dividits en parts més petites però iguals a l'original es poden resoldre així. A l'hora de programar cal prestar especial atenció a la condició de sortida, que hi hagi una manera de trencar el bucle de recursió. En cas contrari s'hi quedaria atrapat indefinidament (o fins que el programa es quedés sense memòria i s'aturés).

7.1. Factorials

El factorial d'un nombre (enter i no negatiu) n és la multiplicació successiva de tots els nombres enters, començant per l'u, fins a n. Matemàticament això s'expressa de la següent manera:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Per exemple, el factorial de 4, o 4!, és el següent:

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

7.1.1 Implementació no-recursiva

```
1  #!/usr/bin/env python3
2  def factorial(n):
3      res = 1
4      while n > 0:
5          res *= n
6          n -= 1
7      return res
8
9  n = int(input("Factorial de... "))
10
11 if n < 0:
```

```
    print("El nombre ha de ser positiu.")
else:
    print(f"El factorial de {n} és {factorial(n)}")
```

7.1.2 Implementació recursiva

```
#!/usr/bin/env python3
def factorial(n):
    if n == 0:
        return 1

    return n * factorial(n-1)

n = int(input("Factorial de... "))

if n < 0:
    print("El nombre ha de ser positiu.")
else:
    print(f"El factorial de {n} és {factorial(n)}")
```

7.2. Successió de Fibonacci

La successió de Fibonacci és una successió de nombres enters on els dos primers termes són 1 i a partir del tercer són s la suma dels dos nombres anteriors. Anomenem F_n a l'enèsim nombre de la sèrie, i amb aquesta notació la podem expressar així: $F_1 = F_2 = 1$ i $F_i = F_{i-1} + F_{i-2}$ per a $i > 2$

7.2.1 Implementació no-recursiva

```
#!/usr/bin/env python3
def fibonacci(n):
    res = 0
    pre_res = 1
    for i in range(n):
        t = pre_res
        pre_res = res
        res += t
```

```
    return res


n = int(input("Posició del nombre en la successió de
    Fibonacci: "))

if n < 1:
    print("El nombre ha de ser superior a 0.")
else:
    print(
        f"El nombre en posició {n} de la successió de
    fibonacci és: {fibonacci(n)}")
```

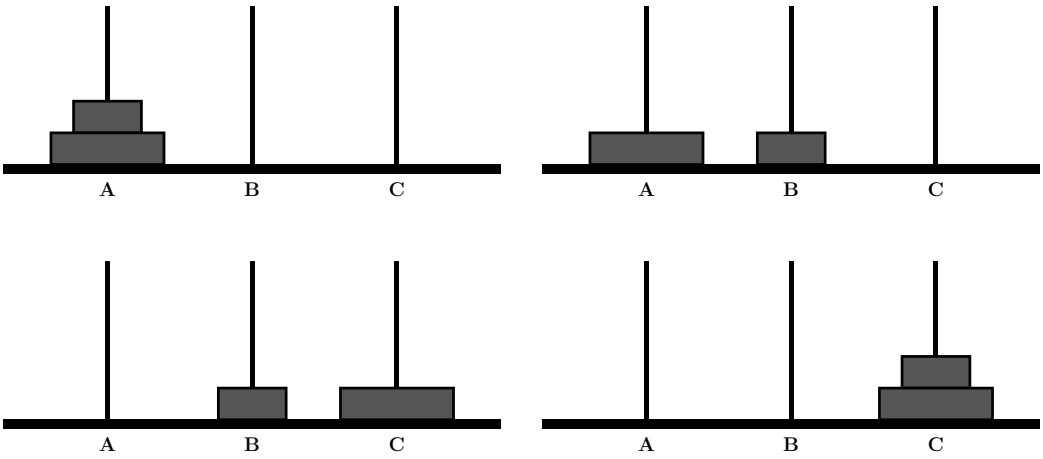
7.2.2 Implementació recursiva

```
#!/usr/bin/env python3
def fibonacci(n):
    if n < 3:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)


n = int(input("Posició del nombre en la successió de
    Fibonacci: "))

if n < 1:
    print("El nombre ha de ser superior a 0.")
else:
    print(f"El nombre en posició {n} de la successió de
    fibonacci és: {fibonacci(n)}")
```

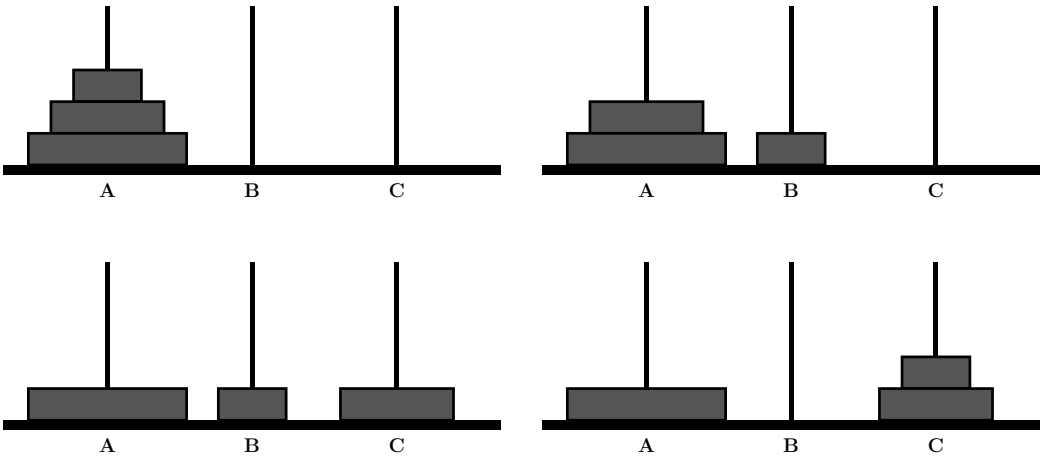
7.3. Torres de Hanoi

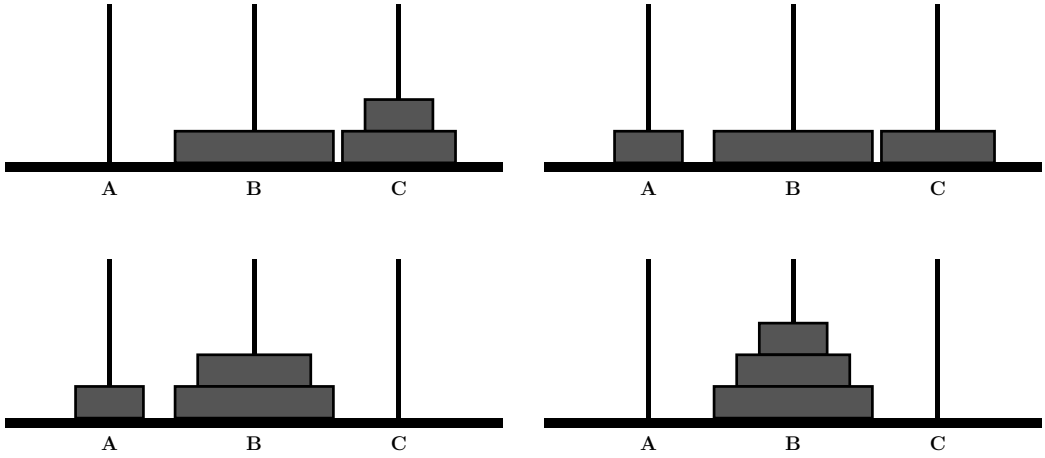
El problema de les Torres de Hanoi consisteix a moure discs apilats de diverses mides d'una fusta a una altra. Hi ha tres fustes i les regles són simples: només es pot canviar de lloc un disc a la vegada i un disc més gran no es pot posar sobre un de més petit.

La resolució amb dos discs és simple. Comencen a la fusta A: movem el disc petit a la fusta B i a continuació movem l'altre a la fusta C. Finalment posem el primer disc sobre el gran, a la fusta C.



Per resoldre el problema amb tres discs primer hem d'aïllar els dos primers, ja que es troben sobre el tercer i, si no ho fem, no podrem moure el tercer disc. Partint doncs de tenir una fusta A amb el tercer disc i una fusta C amb el primer sobre el segon, fem el quart moviment: posar el tercer disc a la fusta B. El que ens queda ara ja ho hem fet abans: tornar a canviar de fusta els primers dos discs. Aquesta vegada partint de la fusta C per acabar a la B, sobre el tercer disc.





Això es va repetint a mesura que augmenta el nombre de discs, i també es pot observar una progressió matemàtica. Si prenem $H(n)$ com el nombre de moviments necessaris per resoldre el problema per un nombre n de discs veiem que per $n = 1$, $H(n) = 1$ doncs només hi ha un sol disc i per moure'l a una altra fusta només cal un moviment.

Amb $n = 2$, $H(n) = 3$, però a què es deu això? Per canviar de fusta dos discs, primer hem de moure el primer ($H(1)$), després canviar el segon i tornar a moure el primer ($H(1)$), per tant:

$$H(2) = H(1) + 1 + H(1) = 2[H(1)] + 1 = 2(1) + 1 = 2 + 1 = 3$$

Amb $n = 3$, $H(n) = 7$. Altra vegada per canviar de fusta tres discs primer hem de moure els dos primers ($H(2)$), després canviar el tercer i tornar a moure els dos primers ($H(2)$) per posar-los sobre el tercer, per tant:

$$H(3) = H(2) + 1 + H(2) = [H(1) + 1 + H(1)] + 1 + [H(1) + 1 + H(1)] =$$
$$= 2[H(1) + 1 + H(1)] + 1 = 2(3) + 1 = 6 + 1 = 7$$

7.3.1 Implementació recursiva

La resolució algorítmica recursiva d'aquest problema és molt interessant i ben senzilla. De fet, l'algoritme és tan senzill i, una vegada entens el raonament, intuïtiu que el problema de les torres sempre es menciona per

explicar la recursió. Si us hi fixeu, ja hem vist que la resolució per un nombre de discs n inclou la resolució per un nombre de discs $n-1$.

```python
#!/usr/bin/env python3
def hanoi(n, start, end, aux):
    if n == 1:
        print(f"Mou 1 de {start} a {end}")
        return
    hanoi(n-1, start, aux, end)
    print(f"Mou {n} de {start} a {end}")
    hanoi(n-1, aux, end, start)


n = int(input("Nombre de discs: "))

if n < 1:
    print("Hi ha d'haver com a mínim un disc.")
else:
    hanoi(n, 'A', 'B', 'C')
```

8. Quick sort

8.1. Com funciona?

El *quick sort* és un algoritme recursiu, que divideix la llista inicial en dos i ordena cada meitat repetint els mateixos passos.

Es pren un element qualsevol que fa de pivot i separa els elements restants en funció de si són més grans o més petits que el pivot. Aquestes dues llistes de nombres menors i majors s'ordenaran de la mateixa manera.

Una vegada les llistes contenen un sol element es comencen a ajuntar: la llista amb els nombres menors passa al davant, seguida del pivot i la dels nombres majors.

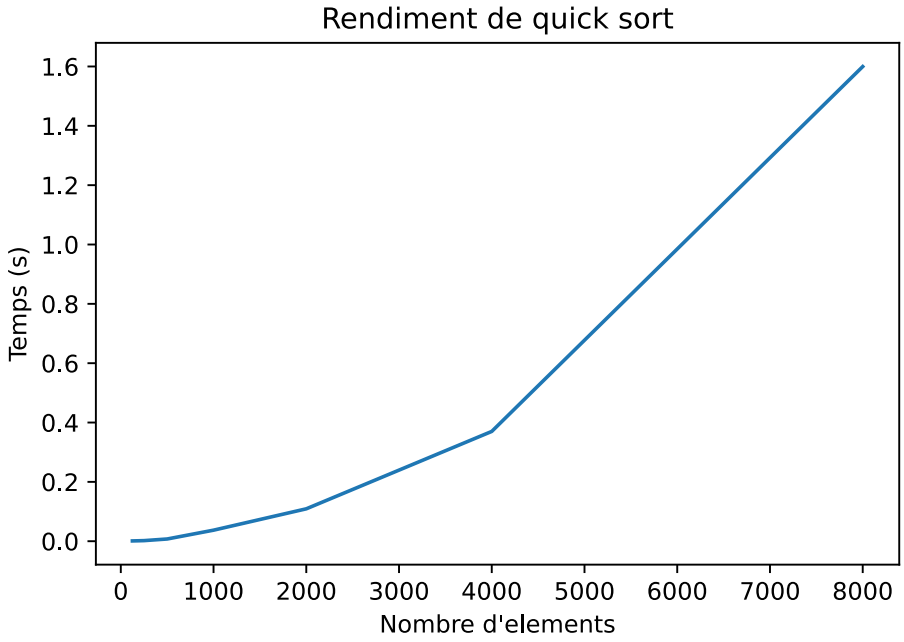
8.2. Implementació

```
#!/usr/bin/env python3
import utils

def sort(array):
    if len(array) < 2:
        return array

    pivot = array[0] # agafo el primer element com a pivot, pot ser qualsevol
    lower, higher = [], []

    for i in array[:-1]:
        if i < pivot:
            lower.append(i)
        else:
            higher.append(i)

    lower = sort(lower)
    higher = sort(higher)

    return lower + [pivot] + higher
```

```
21
22 if __name__ == "__main__":
23     array = utils.numbers()
24     print(sort(array))
```

8.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0010	0.0020	0.0075	0.0372	0.1092	0.3700	1.5997



9. Merge sort

9.1. Com funciona?

Aquest algoritme també és recursiu, divideix la llista en dues i treballa per separat cada meitat. El que fa amb cada meitat és el mateix: dividir-la en dues i repetir fins que es troba amb elements individuals, que compara i ordena formant parelles.

Una vegada tenim les parelles (llistes de dos elements ordenats) podem reconstruir la llista següent: el que fa l'algoritme és agafar l'element més petit d'una llista (el primer) i comparar-lo amb el més petit de l'altra. L'element més petit d'aquests dos es posa a la nova llista. El procés es va repetint fins que una llista es buida, aleshores els elements restants de l'altra es poden afegir al final (sempre respectant l'ordre entre ells). La següent llista aplica el mateix procediment, i així fins que es recupera la llista original, ara amb els elements ordenats.

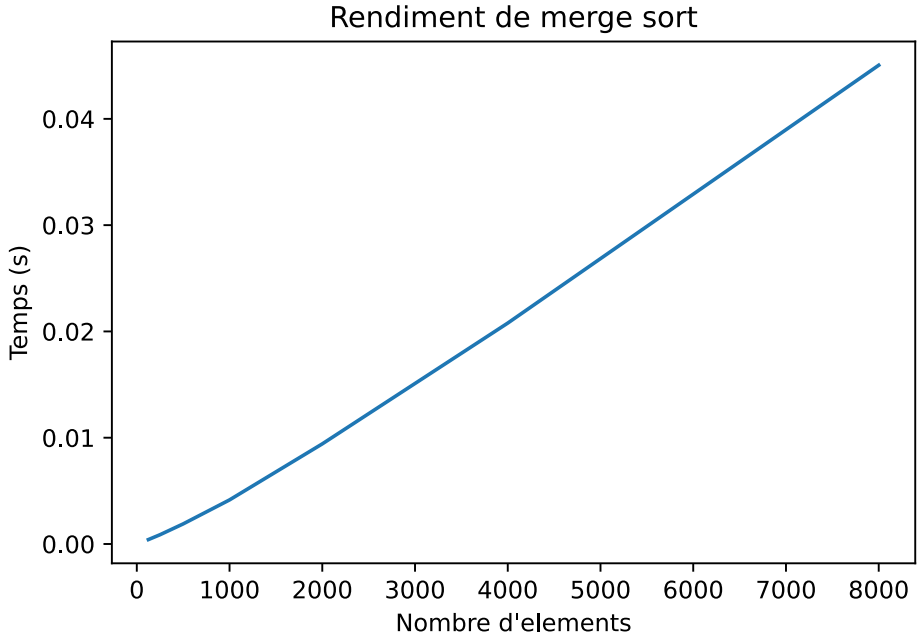
9.2. Implementació

```
1  #!/usr/bin/env python3
2  import utils
3
4  def sort(array):
5      if len(array) < 2:
6          return array
7      mid = len(array)//2
8      left = sort(array[:mid])
9      right = sort(array[mid:])
10
11     merged = []
12     l = 0
13     r = 0
14
15     while l < len(left) and r < len(right):
16         if left[l] < right[r]:
```

```
            merged += [left[l]]
            l += 1
        else:
            merged += [right[r]]
            r += 1

    if l < len(left):
        merged += left[l:]
    if r < len(right):
        merged += right[r:]

    return merged

if __name__ == "__main__":
    array = utils.numbers()
    print(sort(array))
```

9.3. Rendiment

	125	250	500	1000	2000	4000	8000
Temps (s)	0.0004	0.0009	0.0019	0.0041	0.0094	0.0208	0.0450



Part III

Anàlisi i conclusions

10. Comparació dels algoritmes

Hem vist diversos algoritmes que permeten ordenar vectors, però quin és el millor mètode? Com que tots els algoritmes ens retornen el mateix vector, l'ordenat, no hi ha cap algoritme que ens retorni un vector millor ordenat que un altre. Ja que els resultats són tots igual de bons, optarem pel que sigui més eficient.

10.1. Cost Computacional

L'estudi del cost computacional d'un algoritme és l'estudi de la quantitat de recursos que consumeix. Habitualment com a recurs es pren la memòria o, i aquest és el nostre cas, el temps.

Fins ara hem executat tots els algoritmes per saber-ne el temps, però això no sempre és viable. Ho hem vist amb el *bubble sort*: només amb 4000 elements ja s'hi podia estar més d'un quart d'hora, motiu pel qual amb 8000 ni tan sols l'hem executat. Aquest problema apareixerà tard o d'hora amb tots els algoritmes, a algun li passa als 8 mil elements i a un altre li pot passar als 8 milions, per tant és vital trobar la manera d'estimar-ne el temps de computació.

El temps de computació d'un algoritme sempre depèn de la quantitat d'elements, és d'esperar que el que es triga a ordenar deu elements sigui menor que el que es triga a ordenar-ne cent. El temps de computació també depèn del maquinari, però aquest factor el podem eliminar si observem la relació entre dues execucions amb diferent nombre d'elements. És a dir, un algoritme que en duplicar el nombre d'elements duplica el temps d'execució ho farà independentment del maquinari. El que calcularem, doncs, és aquest augment que no depèn del maquinari sinó de l'algoritme.

10.1.1 Fórmula

Comencem posant xifres a l'exemple anterior. Si el temps d'execució de l'algoritme per n elements, que d'ara endavant anomenarem $t(n)$, val també

n, en duplicar el nombre d'elements $t(2n)$ serà $2n$. Veiem que el temps es duplica en duplicar el nombre d'elements amb la següent operació:

$$\frac{t(2n)}{t(n)} = \frac{2\cancel{n}}{\cancel{n}} = 2$$

I què canviaria si en comptes de $t(n)$ fos, en comptes de n, fos de n^2?

$$\frac{t(2n)}{t(n)} = \frac{2^2\cancel{n^2}}{\cancel{n^2}} = 2^2 = 4$$

En aquest cas el temps no s'ha duplicat, s'ha quadruplicat. Passem de multiplicar-lo per 2 en duplicar elements a multiplicar-lo per 2^2.

Generalitzant això veiem que quan el $t(n)$ d'un algoritme és de n^p i se'n dupliquen el nombre d'elements el temps d'execució és de $2^p n^p$. Dividint $t(2n)$ entre $t(n)$ podem obtenir la relació entre ambdós:

$$\frac{t(2n)}{t(n)} = \frac{2^p\cancel{n^p}}{\cancel{n^p}} = 2^p$$

Si seguim generalitzant trobem que per una relació entre les mides dels vectors m la relació entre els temps és m^p:

$$\frac{t(mn)}{t(n)} = m^p$$

Poder aproximar el valor de l'exponent p serà clau per conèixer el temps de computació. La relació entre un vector de 10 elements i un de 10.000 és $\frac{10000}{10} = 1000$. Amb un exponent p d'1 només ens costaria mil vegades més ordenar un vector mil vegades més gran, ja que $1000^1 = 1000$. Si aquest exponent fos 2, el vector mil vegades més gran trigaria un milió de vegades el temps original, $1000^2 = 1.000.000$.

Partint de la fórmula generalitzada $\frac{t(mn)}{t(n)} = m^p$ que acabem d'obtenir

aïllarem p de la següent manera:

$$\frac{t(mn)}{t(n)} = m^p \Rightarrow \log\left(\frac{t(mn)}{t(n)}\right) = \log(m^p) = \log(m) \times p \Rightarrow$$

$$\Rightarrow p = \frac{log\left(\frac{t(mn)}{t(n)}\right)}{log(m)} = \frac{\log(t(mn)) - \log(t(n))}{\log(m)}$$

10.1.2 Càlcul

Si volem calcular l'exponent p aplicant la fórmula que acabem de definir necessitem dos valors: temps d'execució per un nombre d'elements n i el temps d'execució per un nombre d'elements $n \times m$.

No és casualitat que tots els algoritmes els haguem executat amb 125, 250, 500, 1000, 2000, 4000 i 8000 elements. Els més destres amb les xifres podran observar que cada nombre d'aquesta sèrie és el doble que l'anterior. Per calcular l'exponent necessitem els temps d'execució, que estan recollits en la següent taula.

	125	250	500	1000	2000	4000	8000
bubble	0.0283	0.2121	1.6764	14.0336	116.9414	966.9290	(-)
insertion	0.0007	0.0033	0.0134	0.0518	0.1999	0.8066	3.2098
merge	0.0004	0.0009	0.0019	0.0041	0.0094	0.0208	0.0450
quick	0.0010	0.0020	0.0075	0.0372	0.1092	0.3700	1.5997
selection	0.0007	0.0027	0.0118	0.0508	0.1904	0.7498	2.9973
shell	0.0004	0.0010	0.0024	0.0061	0.0131	0.0287	0.0645

El càlcul de l'exponent necessita els temps d'execució amb dos vectors de diferents dimensions. Si calculéssim el del *bubble sort* podríem prendre el temps de 125 i 250 elements, 0.0283 i 0.2121 segons. Com que un és el doble que l'altre, m val 2.

$$p = \frac{\log(t(mn)) - \log(t(n))}{\log(m)} = \frac{\log(t(0.2121)) - \log(t(0.0283))}{\log(2)} = 2.9049$$

Si repetim aquest càlcul amb totes les quantitats d'elements i tots els algoritmes podem començar a omplir la pròxima taula.

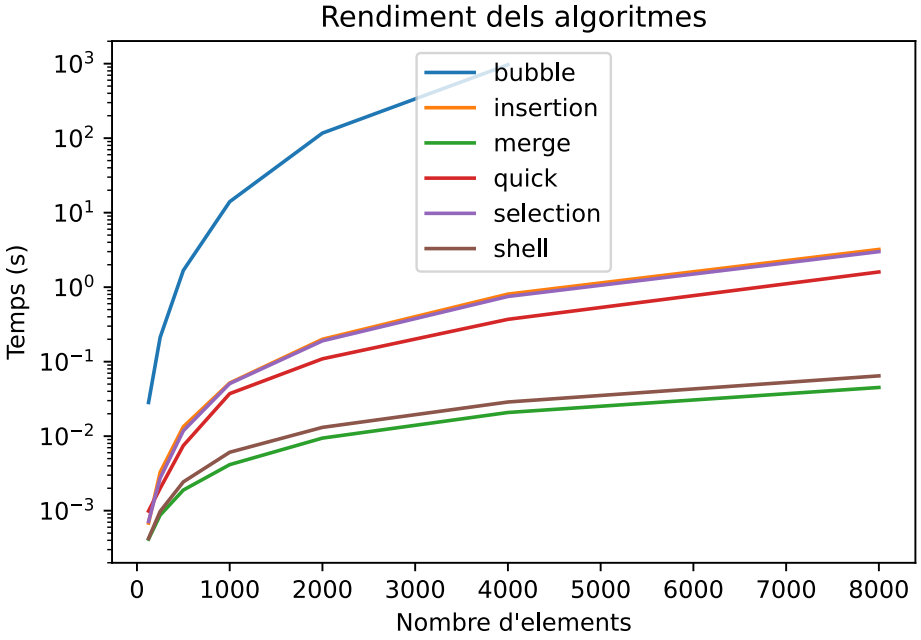
Quan tinguem aquests exponents comprovarem que no n'hi ha cap d'igual. Cal recordar que el temps d'execució depèn del vector, que generem de forma aleatòria, i per tant pot variar. Precisament per això prenem més d'una mesura, fent la mitjana de tots els exponents de cada algoritme podem obtenir un resultat més fiable.

A la nostra taula hi podem afegir una columna més 8000/125 on ttrobem calculat l'exponent p amb els vectors més gran i més petit de cada algoritme. Això vol dir que m val 64 ($\frac{8000}{125} = 64$), excepte amb el *bubble sort*, que és de 32 ($\frac{4000}{125} = 32$, recordem que no l'hem executat amb 8000 elements.

	250/125	500/250	1000/500	2000/1000
bubble	2.9049	2.9823	3.0654	3.0588
insertion	2.2613	2.0320	1.9525	1.9486
merge	1.0684	1.1183	1.1332	1.1834
quick	1.0135	1.9061	2.3159	1.5529
selection	1.9333	2.1087	2.1060	1.9058
shell	1.2038	1.3157	1.3208	1.1100

	4000/2000	8000/4000	mitjana	8000/125
bubble	3.0476	(-)	3.0118	3.0118
insertion	2.0130	1.9925	2.0333	2.0333
merge	1.1418	1.1158	1.1268	1.1268
quick	1.7608	2.1121	1.7769	1.7769
selection	1.9777	1.9991	2.0051	2.0051
shell	1.1305	1.1655	1.2077	1.2077

Observem que el *merge sort* té el menor exponent, $p = 1.1268$, seguit del *shell sort* amb $p = 1.2077$ i el *quick sort* amb $p = 1.7769$. Lleugerament per sobre de 2, els algoritmes de selecció i inserció tenen valors de p de 2.051 i 2.0333. Finalment, molt per sobre la resta, el *bubble sort* amb l'exponent p de 3.0118


Rendiment dels algoritmes
bubble
insertion
merge
quick
selection
shell
Temps (s)
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
0
1000
2000
3000
4000
5000
6000
7000
8000
Nombre d'elements

11. Conclusions

Amb aquest treball tenia com a objectiu entendre i comparar diversos algoritmes d'ordenació.

Considero que no només he assolit els objectius que em vaig plantejar, sinó que a mesura que desenvolupava el treball anava ampliant els meus coneixements sobre programació. Aquest camp sempre m'havia interessat i aquesta no ha estat la primera vegada que m'hi endinso, però ha estat la primera vegada que ho he fet amb un objectiu clar i amb una excusa per passar-me unes quantes hores donant voltes a un algoritme que aleshores encara no entenia.

Durant la recerca i el redactat del treball m'he adonat de com n'és d'important l'efectivitat d'un algoritme d'ordenació. Si amb els coneixements que tenia quan vaig començar aquesta tasca hagués dissenyat un algoritme d'ordenació hauria fet una cosa similar al *bubble sort*, potser a l'*insertion*. Aquests dos algoritmes són els dos algoritmes més lents que he treballat.

Fent aquest treball, a més d'aprendre sobre programació i tenir el meu primer contacte amb l'algorítmica he aprofitat l'avinentesa per tocar altres disciplines, també digitals, de les que vull fer gala. Per exemple, els gràfics de les Torres de Hanoi els he fet a mà en SVG. Les taules, els gràfics de rendiment i el càlcul de l'exponent p no els he fet en cap full de càlcul, ho he fet en Notebooks de Jupyter amb Pandas i Matplotlib. Finalment aquest document no ha passat per un sol processador de text sinó que està fet en LaTeX. Malgrat que d'entrada això pugui semblar un hàndicap, he de confessar que una vegada l'he entès no m'ha portat cap problema, ans el contrari, m'ha facilitat extraordinàriament la tasca.

Ara sí, per concloure d'una vegada per totes aquest treball, només em queda afegir que els recursos que es disposen per fer un TREC són limitats, especialment el temps. Amb això no vull suplicar la misericòrdia del tribunal, sinó obrir la porta a una possible segona part d'aquest treball. No l'escriuré jo, però considero que aquest camp és prou ampli per poder continuar la recerca on l'he deixada jo o bé començar de nou analitzant altres algoritmes o estudiant-ne altres aspectes.

Agraïments

Vull agrair al meu tutor, el Xavi, l'orientació, la paciència i sobretot que em proposés aquest tema.

Al meu pare li dec el pròleg que és, amb diferència, la millor part d'aquest treball i la revisió gramatical.

A la família i els amics el suport donat.

Bibliografia

[1] makigas.es: programming tutorials in Spanish, *Algoritmos de ordenación – 1. Algoritmos de ordenación* [En línea]

https://www.youtube.com/watch?v=nqusvWZiL8M

[Data de consulta: 3 de setembre del 2020]

[2] makigas.es: programming tutorials in Spanish, *Algoritmos de ordenación – 2. Selección* [En línea]

https://www.youtube.com/watch?v=ZMO3Fow05tg

[Data de consulta: 3 de setembre del 2020]

[3] makigas.es: programming tutorials in Spanish, *Algoritmos de ordenación – 3. Inserción* [En línea]

https://www.youtube.com/watch?v=C-fPGWgeYzY

[Data de consulta: 3 de setembre del 2020]

[4] makigas.es: programming tutorials in Spanish, *Algoritmos de ordenación – 4. Burbuja* [En línea]

https://www.youtube.com/watch?v=EQMGabLO_M0

[Data de consulta: 3 de setembre del 2020]

[5] RamiroG, *SHELL SORT, algoritmo de ordenamiento* [En línea]

https://www.youtube.com/watch?v=AeCeFdSoPEM

[Data de consulta: 5 de setembre del 2020]

[6] Alan, *Método de ordenamiento Shell* [En línea]

https://www.youtube.com/watch?v=bv8rxkcZ_-o

[Data de consulta: 5 de setembre del 2020]

[7] Lupo Montero, *Algoritmos - Ordenamiento rápido - Quick Sort* [En línea]

https://www.youtube.com/watch?v=DYmTpUfcyT8

[Data de consulta: 8 de setembre del 2020]

[8] Alexander Arias, *Merge Sort - Algoritmo de Ordenamiento - Programación Avanzada* [En línea]

https://www.youtube.com/watch?v=Xhlf6f26M7Y

[Data de consulta: 9 de setembre del 2020]

[9] Brian Faure, *Merge Sort: Background & Python Code* [En línea]

https://www.youtube.com/watch?v=3aTfQvs-_hA

[Data de consulta: 9 de setembre del 2020]

[10] Wikipedia, *Selection sort* [En línea]

https://en.wikipedia.org/wiki/Selection_sort

[Data de consulta: 18 d'octubre del 2020]

[11] Wikipedia, *Insertion sort* [En línea]

https://en.wikipedia.org/wiki/Insertion_sort

[Data de consulta: 18 d'octubre del 2020]

[12] Wikipedia, *Bubble sort* [En línea]

https://en.wikipedia.org/wiki/Bubble_sort

[Data de consulta: 18 d'octubre del 2020]

[13] Wikipedia, *Shellsort* [En línea]

https://en.wikipedia.org/wiki/Shellsort

[Data de consulta: 20 de novembre del 2020]

[14] Wikipedia, *Quicksort* [En línea]

https://en.wikipedia.org/wiki/Quicksort

[Data de consulta: 21 de novembre del 2020]

[15] Wikipedia, *Merge sort* [En línea]

https://en.wikipedia.org/wiki/Merge_sort

[Data de consulta: 21 de novembre del 2020]

[16] Wikipedia, *Tower of Hanoi* [En línea]
https://en.wikipedia.org/wiki/Tower_of_Hanoi
[Data de consulta: 14 de gener del 2021]