

Treball de Recerca:  
Algoritmes d'ordenació

Joan Coma Bages

Curs 2020/2021

# Índex

<b>1</b>	<b>Introducció</b>	<b>3</b>
<b>2</b>	<b>Què és ordenar un vector?</b>	<b>4</b>
2.1	Què podem ordenar? . . . . .	4
2.2	Per què ordenem? . . . . .	5
<b>3</b>	<b>Selection sort</b>	<b>6</b>
3.1	Com funciona? . . . . .	6
3.2	Pas a pas . . . . .	6
3.3	Implementació . . . . .	6
3.4	Rendiment . . . . .	7
<b>4</b>	<b>Insertion sort</b>	<b>8</b>
4.1	Com funciona? . . . . .	8
4.2	Pas a pas . . . . .	8
4.3	Implementació . . . . .	8
4.4	Rendiment . . . . .	9
<b>5</b>	<b>Bubble sort</b>	<b>10</b>
5.1	Com funciona? . . . . .	10
5.2	Pas a pas . . . . .	10
5.3	Implementació . . . . .	10
5.4	Rendiment . . . . .	11
<b>6</b>	<b>Shell sort</b>	<b>12</b>
6.1	Com funciona? . . . . .	12

6.2	Pas a pas . . . . .	13
6.3	Implementació . . . . .	13
6.4	Rendiment . . . . .	14
<b>7</b>	<b>Recursivitat</b>	<b>15</b>
7.1	Factorials . . . . .	15
7.1.1	Implementació no-recursiva . . . . .	15
7.1.2	Implementació recursiva . . . . .	16
7.2	Successió de Fibonacci . . . . .	16
7.2.1	Implementació no-recursiva . . . . .	16
7.2.2	Implementació recursiva . . . . .	17
7.3	Torres de Hanoi . . . . .	18
7.3.1	Implementació recursiva . . . . .	19
<b>8</b>	<b>Quick sort</b>	<b>20</b>
8.1	Com funciona? . . . . .	20
8.2	Pas a pas . . . . .	20
8.3	Implementació . . . . .	20
8.4	Rendiment . . . . .	22
<b>9</b>	<b>Merge sort</b>	<b>23</b>
9.1	Com funciona? . . . . .	23
9.2	Pas a pas . . . . .	23
9.3	Implementació . . . . .	23
9.4	Rendiment . . . . .	25
<b>10</b>	<b>Comparació dels algoritmes</b>	<b>26</b>
10.1	Cost Computacional . . . . .	26
10.1.1	Fórmula . . . . .	27
10.1.2	Càlcul . . . . .	28

# 1. Introducció

## 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.

L'índex d'aquest document permet navegar-lo amb facilitat, però no faria servei si les pàgines no estiguessin ordenades. Si aquesta pàgina, la 5 es trobés entre la 7 i la 17 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.

## 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. Pas a pas

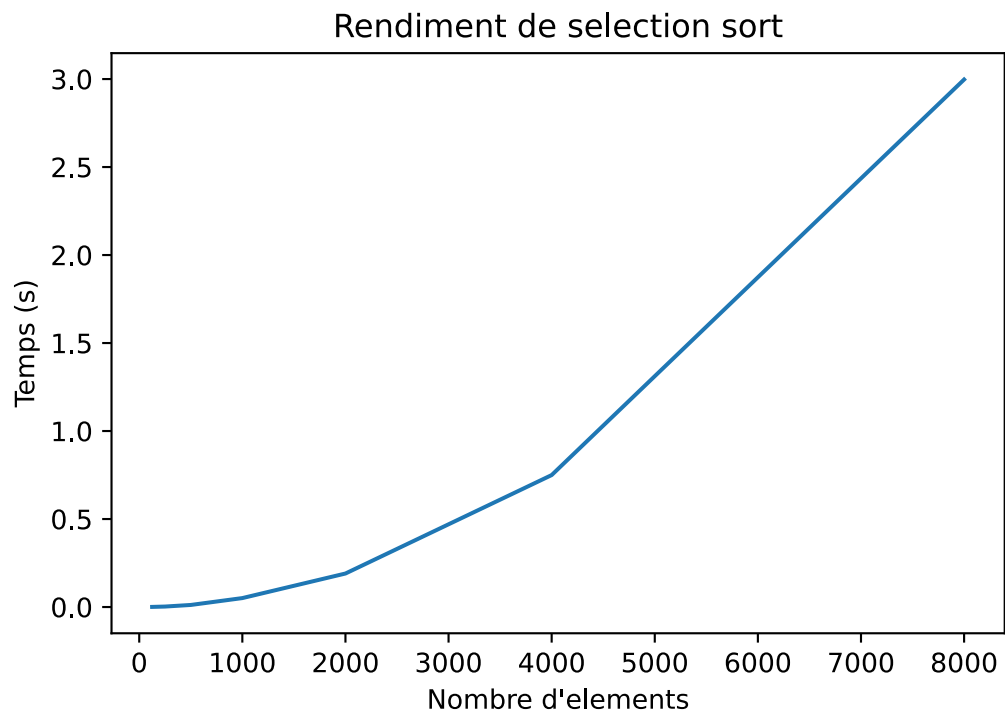
### 3.3. Implementació

```
1  #!/usr/bin/env python3
2  import utils
3
4  def sort(array):
5      for i in range(len(array[: -1])):
6          low = i
7          for j in range(i, len(array)):
8              if array[low] > array[j]:
9                  low = j
10
11         array[i], array[low] = array[low], array[i]
12     return array
13
14 if __name__ == "__main__":
15     array = utils.numbers()
```

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

### 3.4. 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 deixariem al principi de la llista.

### 4.2. Pas a pas

### 4.3. Implementació

```
1 #!/usr/bin/env python3
2 import utils
3
4 def sort(array):
5     for i in range(len(array)):
```

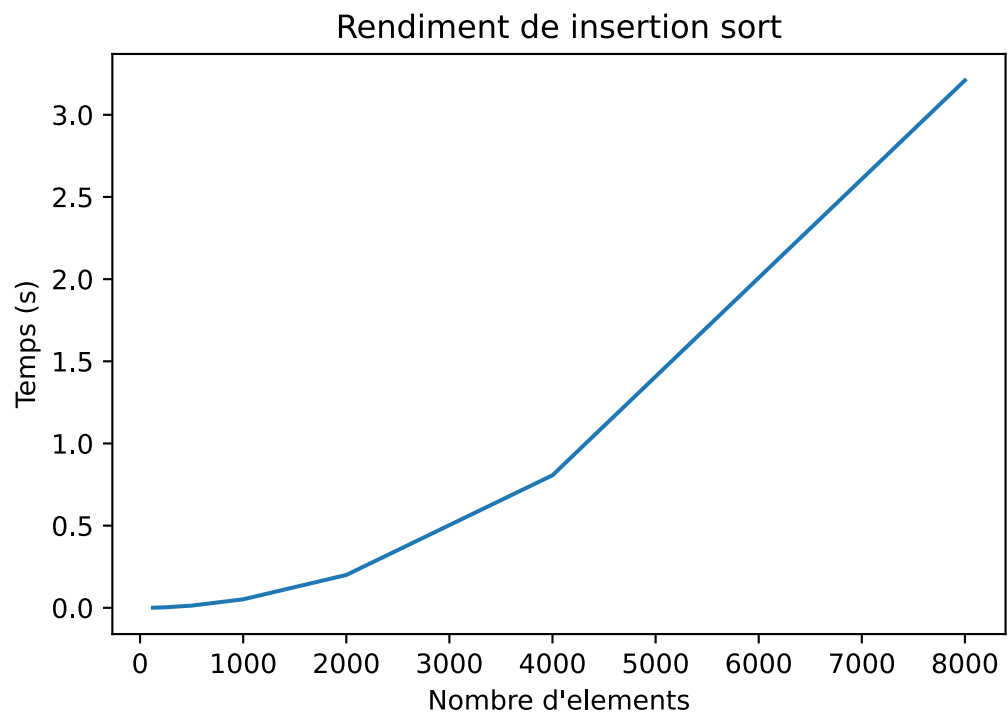
```

6         comp = array[i]
7         j = i - 1
8         while j >= 0 and array[j] > comp:
9             array[j + 1] = array[j]
10            j -= 1
11            array[j + 1] = comp
12    return array
13
14 if __name__ == "__main__":
15     array = utils.numbers()
16     print(sort(array))

```

## 4.4. 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, canviant-ne la posició si el segon és més gran que el primer. Com que sempre movem endavant l'element més gran Aquest procés es repeteix tantes vegades com faci falta per ordenar la llista.

### 5.2. Pas a pas

### 5.3. 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
12         # executat sense cap break, és a dir, si mai s'ha complert
13         # la condició array[i] > array[i+1]
14         else:
```

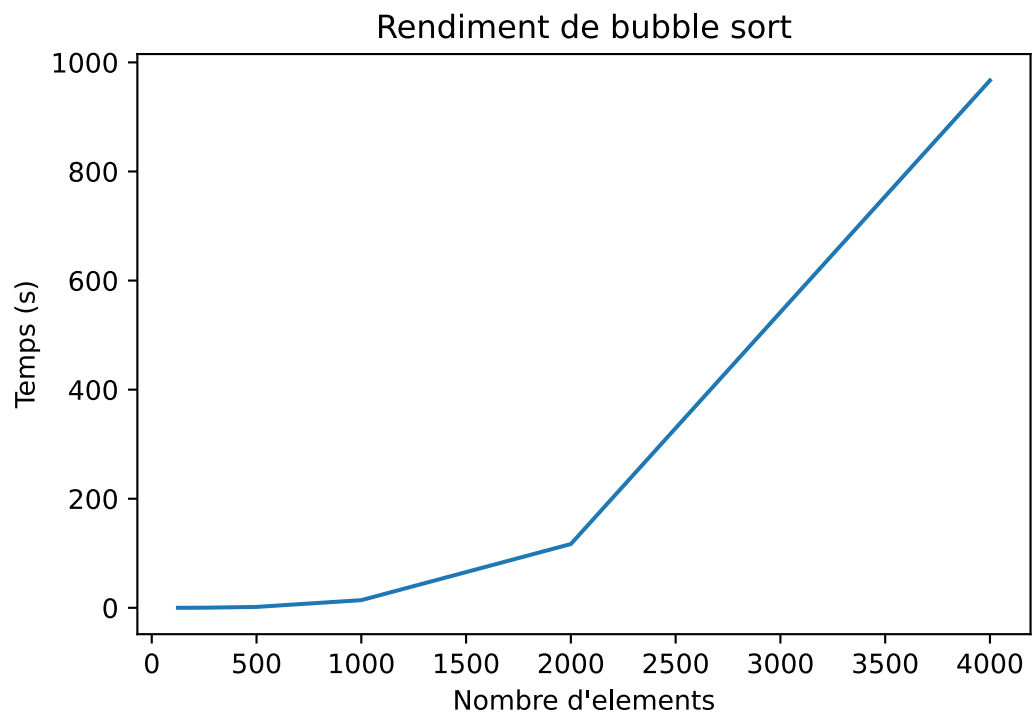
```

13         break
14     return array
15
16 if __name__ == "__main__":
17     array = utils.numbers()
18     print(sort(array))

```

## 5.4. 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 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.

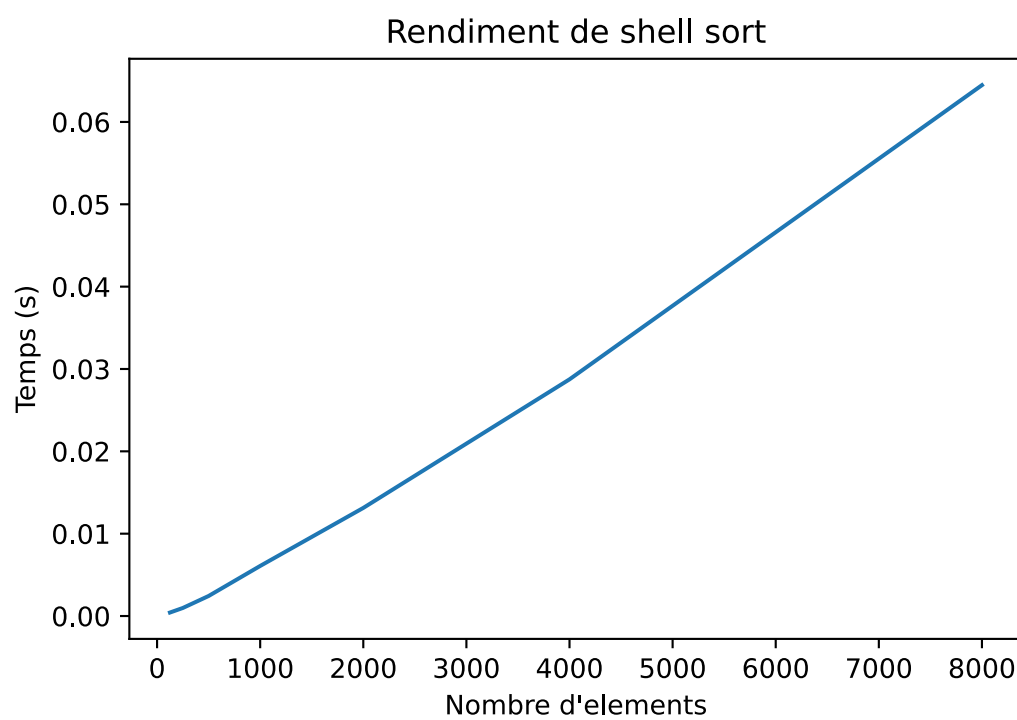
## 6.2. Pas a pas

## 6.3. 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))
```

## 6.4. Rendiment

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



## 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:

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

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

```

### 7.1.2 Implementació recursiva

```

1 #!/usr/bin/env python3
2 def factorial(n):
3     if n == 0:
4         return 1
5
6     return n * factorial(n-1)
7
8 n = int(input("Factorial de... "))
9
10 if n < 0:
11     print("El nombre ha de ser positiu.")
12 else:
13     print(f"El factorial de {n} és {factorial(n)}")

```

## 7.2. Successió de Fibonacci

La successió de Fibonacci és una successió de nombres enters en què un nombre en posició  $n$  és la suma dels dos anteriors. Per definició, els nombres en primera i segona posició són l'1. Els següents ja es poden calcular: el segon  $1 + 1 = 2$ , el tercer  $1 + 2 = 3$ , el quart  $2 + 3 = 5$ , etc.

### 7.2.1 Implementació no-recursiva

```

1 #!/usr/bin/env python3
2 def fibonacci(n):

```

```

3     res = 1
4     pre_res = 0
5     for i in range(n):
6         t = pre_res
7         pre_res = res
8         res += t
9     return res
10
11
12 n = int(input("Posició del nombre en la successió de
    Fibonacci: "))
13
14 if n < 0:
15     print("El nombre ha de ser positiu.")
16 else:
17     print(
18         f"El nombre en posició {n} de la successió de
        fibonacci és: {fibonacci(n)}")

```

## 7.2.2 Implementació recursiva

```

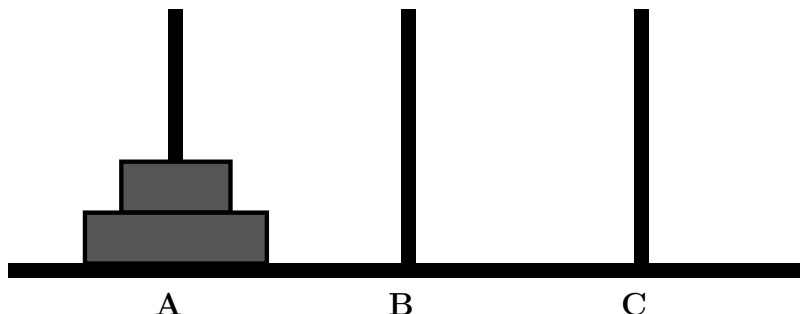
1 #!/usr/bin/env python3
2 def fibonacci(n):
3     if n < 2:
4         return 1
5     return fibonacci(n-1) + fibonacci(n-2)
6
7
8 n = int(input("Posició del nombre en la successió de
    Fibonacci: "))
9
10 if n < 0:
11     print("El nombre ha de ser positiu.")
12 else:
13     print(f"El nombre en posició {n} de la successió de
        fibonacci és: {fibonacci(n)}")

```

### 7.3. Torres de Hanoi

Les Torres de Hanoi és un problema que 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, si no, 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:

$$\begin{aligned} H(3) &= H(2) + 1 + H(2) = [H(1) + 1 + H(1)] + 1 + [H(1) + 1 + H(1)] = \\ &= 2[H(1) + 1 + H(1)] + 1 = 2(3) + 1 = 6 + 1 = 7 \end{aligned}$$

### 7.3.1 Implementació recursiva

La resolució algorítmica recursiva d'aquest problema és molt interessant i ben senzilla. De fet aquest problema sempre s'utilitza en parlar de recursivitat, perquè l'ús d'aquesta permet que . Si us hi fixeu, ja hem vist que la resolució per un nombre de discs  $n$  inclou la resolució per un nombre de discs  $n - 1$ .

```

1  #!/usr/bin/env python3
2  def hanoi(n, start, end, aux):
3      if n == 1:
4          print(f"Mou 1 de {start} a {end}")
5          return
6      hanoi(n-1, start, aux, end)
7      print(f"Mou {n} de {start} a {end}")
8      hanoi(n-1, aux, end, start)
9
10
11 n = int(input("Nombre de discs: "))
12
13 if n < 1:
14     print("Hi ha d'haver com a mínim un disc.")
15 else:
16     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. Pas a pas

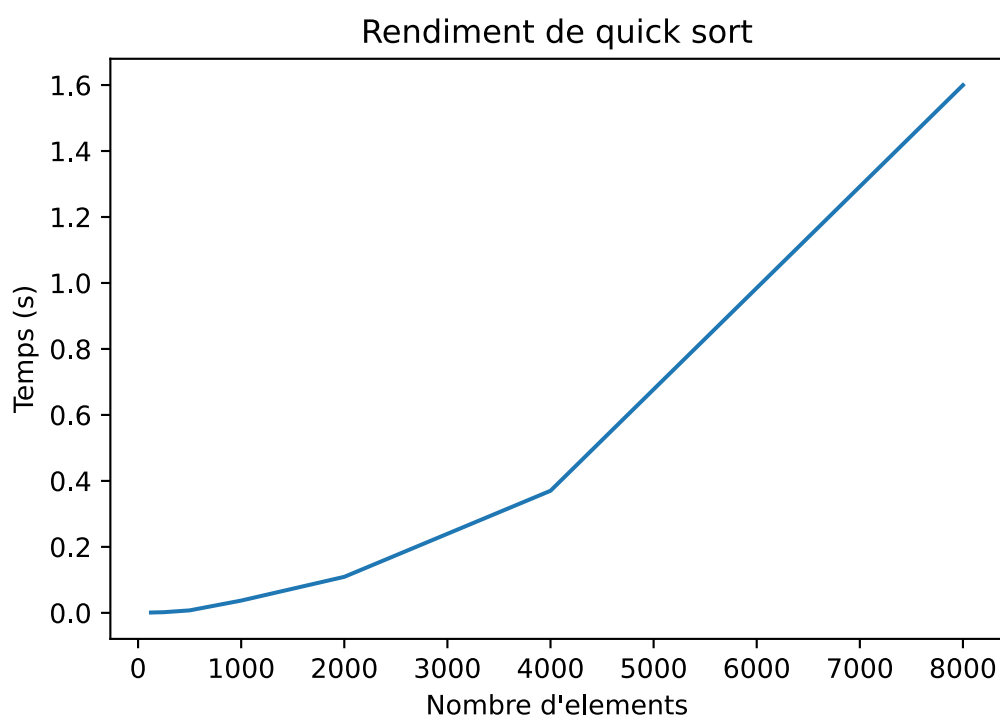
### 8.3. Implementació

```
1  #!/usr/bin/env python3
2  import utils
3
4  def sort(array):
5      if len(array) < 2:
6          return array
7
8      pivot = array[0] # agafo el primer element com a pivot,
9      # pot ser qualsevol
10     lower, higher = [], []
```

```
11     for i in array[:-1]:
12         if i < pivot:
13             lower.append(i)
14         else:
15             higher.append(i)
16
17     lower = sort(lower)
18     higher = sort(higher)
19
20     return lower + [pivot] + higher
21
22 if __name__ == "__main__":
23     array = utils.numbers()
24     print(sort(array))
```

## 8.4. 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 (l·listes de dos elements ordenats) podem reconstruir la llista següent: el que fa l'algoritme és agafar l'element més petit de 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. Pas a pas

### 9.3. Implementació

```
1 #!/usr/bin/env python3
2 import utils
3
4 def sort(array):
5     if len(array) < 2:
6         return array
```



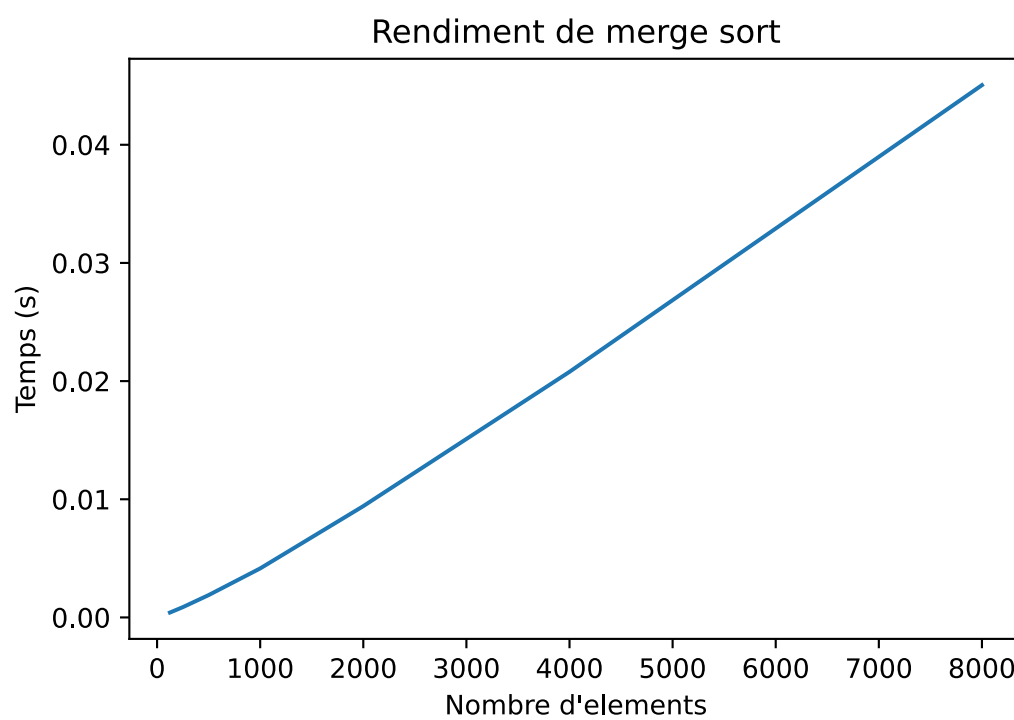
```

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]:
17             merged += [left[l]]
18             l += 1
19         else:
20             merged += [right[r]]
21             r += 1
22
23     if l < len(left):
24         merged += left[l:]
25     if r < len(right):
26         merged += right[r:]
27
28     return merged
29
30 if __name__ == "__main__":
31     array = utils.numbers()
32     print(sort(array))

```

## 9.4. Rendiment

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



## 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, un ho fa als 8 mil elements i un altre ho pot fer 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 d'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, és a dir el  $t(n)$ , és 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:

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

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

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

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:

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

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 10000 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,  $1000^2 = 1000000$ .

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

$$\begin{aligned} \frac{t(mn)}{t(n)} = m^p &\Rightarrow \log\left(\frac{t(mn)}{t(n)}\right) = \log(m^p) = \log(m) \times p \Rightarrow \\ &\Rightarrow p = \frac{\log\left(\frac{t(mn)}{t(n)}\right)}{\log(m)} = \frac{\log(t(mn)) - \log(t(n))}{\log(m)} \end{aligned}$$

### 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 hàgim 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

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

Prenent els temps d'aquesta taula podem calcular  $p$  per qualsevol algorisme. Si substituir  $n$  per una quantitat d'elements i  $m$  per 2 obtenim  $p$  entre el nombre d'elements entre el gran i el petit per calcular l'exponent  $p$  de dos vectors.

Ara, amb els temps d'execució de cada algorisme, podem calcular el valor de  $p$ . Prenem com a  $n$  el nombre d'elements i com a  $m$  2 per calcular la

relació amb la següent quantitat d'elements per completar la següent taula. La mitjana l'obtenim amb aquests valors i podem comprovar que té el mateix valor que l'exponent  $p$  calculat amb els vectors més petit i més gran.  $m$  en aquest cas té valor de 64 ( $\frac{8000}{125} = 64$ ), excepte amb el bubble sort, que és de 32 ( $\frac{4000}{125} = 32$ , no l'hem executat amb 8000 elements).

	250/125	500/250	1000/500	2000/1000	4000/2000	8000/4000	mitjana	8000/125
<b>bubble</b>	2.9049	2.9823	3.0654	3.0588	3.0476	(-)	3.0118	3.0118
<b>insertion</b>	2.2613	2.0320	1.9525	1.9486	2.0130	1.9925	2.0333	2.0333
<b>merge</b>	1.0684	1.1183	1.1332	1.1834	1.1418	1.1158	1.1268	1.1268
<b>quick</b>	1.0135	1.9061	2.3159	1.5529	1.7608	2.1121	1.7769	1.7769
<b>selection</b>	1.9333	2.1087	2.1060	1.9058	1.9777	1.9991	2.0051	2.0051
<b>shell</b>	1.2038	1.3157	1.3208	1.1100	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

