Treball de Recerca: Algoritmes d'ordenació

Joan Coma Bages

Curs 2020/2021

Índex

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 11 i la 3 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?

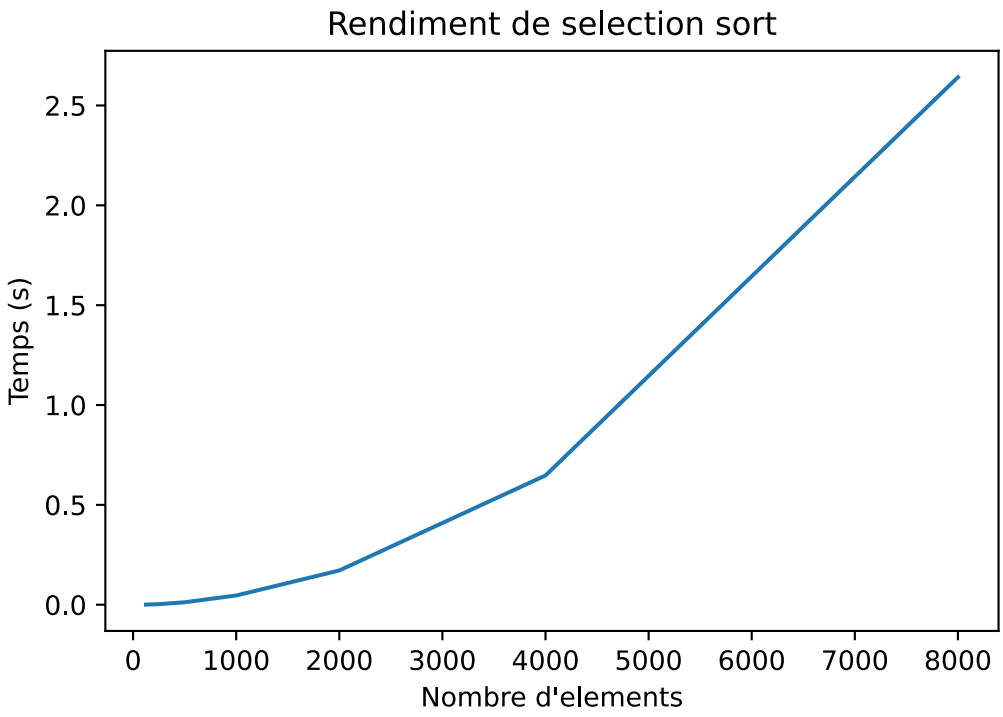
Aquest algoritme itera sobre la llista buscant el nombre més baix, que col·loca al principi. A continuació repeteix el mateix procés buscant el segon nombre més baix, però no des de l'inici sinó des de la segona posició (sabem que en la primera hi trobem el nombre més baix). Aquest procediment es repeteix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

3.2. Pseudocodi

3.3. 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))
```

3.4. Rendiment



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. Al ser el primer element no n'hi ha d'anteriors, el considerarem ordenat. En la seguent 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. Quan el trobem el nostre element el col·locarem a continuació del menor. Si després de comparar-lo amb tota la resta en trobéssim un de menor el deixariem al principi de la llista.

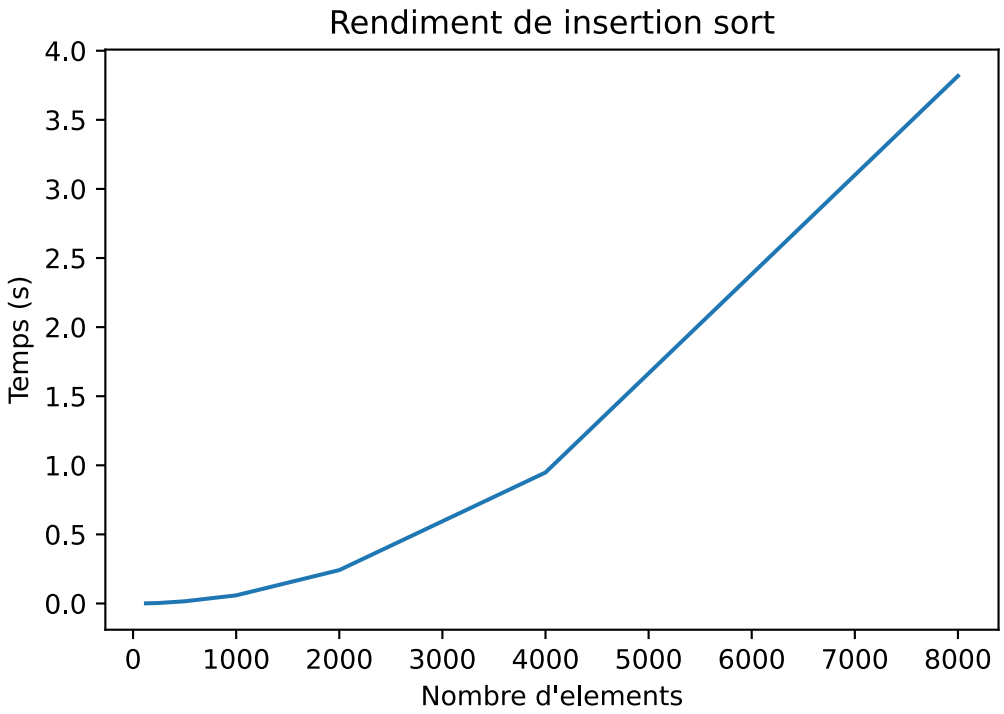
4.2. Pseudocodi

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



5. Bubble sort

5.1. Com funciona?

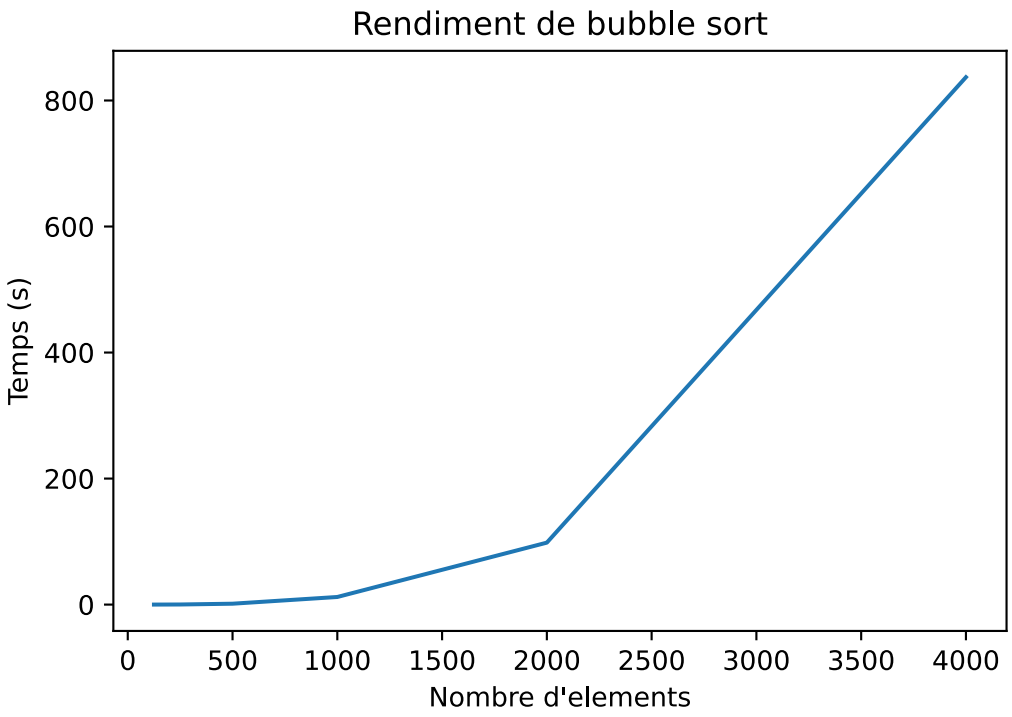
El bubble sort compara cada element de la llista amb el següent, canviant-ne la posició si el segon és més gran que el primer. Aquest procés es repeteix tantes vegades com faci falta per ordenar la llista.

5.2. Pseudocodi

5.3. Implementació

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

def sort(array):
    while True:
        for i in range(len(array[:-1])):
            if array[i] > array[i+1]:
                array[i], array[i+1] = array[i+1], array[i]
                break

        # 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]
        else:
            break
    return array

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

5.4. Rendiment



6. Shell sort

6.1. Com funciona?

L'ordenació shell és un algoritme que abans de comparar els elements que es troben un al costat de l'altre per tal d'anar creant una llista ordenada progressivament el que fa és compartimentar la tasca agrupant nombres similars.

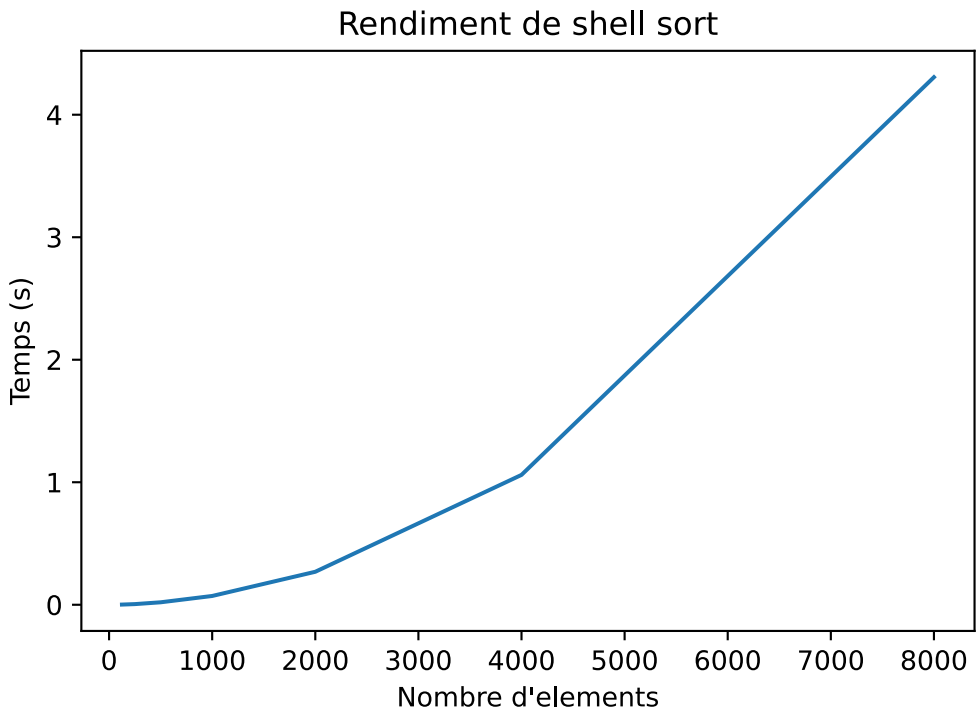
Comença agafant el primer element, i no el compara amb el segon sinó amb l'enèsim, el que sigui més petit ocupa la posició del primer. Continua així fins que arriba al final de la llista, quan torna a començar però fent salts (entre el primer i l'enèsim element) més petits. L'última iteració es fa amb passos d'u.

6.2. Pseudocodi

6.3. Implementació

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

def sort(array):
    step = len(array) // 2
    while step != 0:
        for offset in range(step):
            sorted = insertion.sort(array[offset::step])
            for i in range(len(sorted)):
                array[step*i + offset] = sorted[i]
        step //= 2
    return sorted

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

6.4. Rendiment



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:

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

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

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

7.1.1 Implementació no-recursiva

```
#!/usr/bin/env python3
def factorial(n):
    res = 1
    while n > 0:
        res *= n
        n -= 1
    return res

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)}")
```

7.1.2 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 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

```
#!/usr/bin/env python3
def fibonacci(n):
    res = 1
    pre_res = 0
    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 < 0:
    print("El nombre ha de ser positiu.")
else:
    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 puzle 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 puzle 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 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 puzle és molt interessant i ben senzilla. Si us hi fixeu, ja hem vist que la resolució per un nombre de discs n inclou la resolució per un nombre de discs $n - 1$.

```
#!/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.

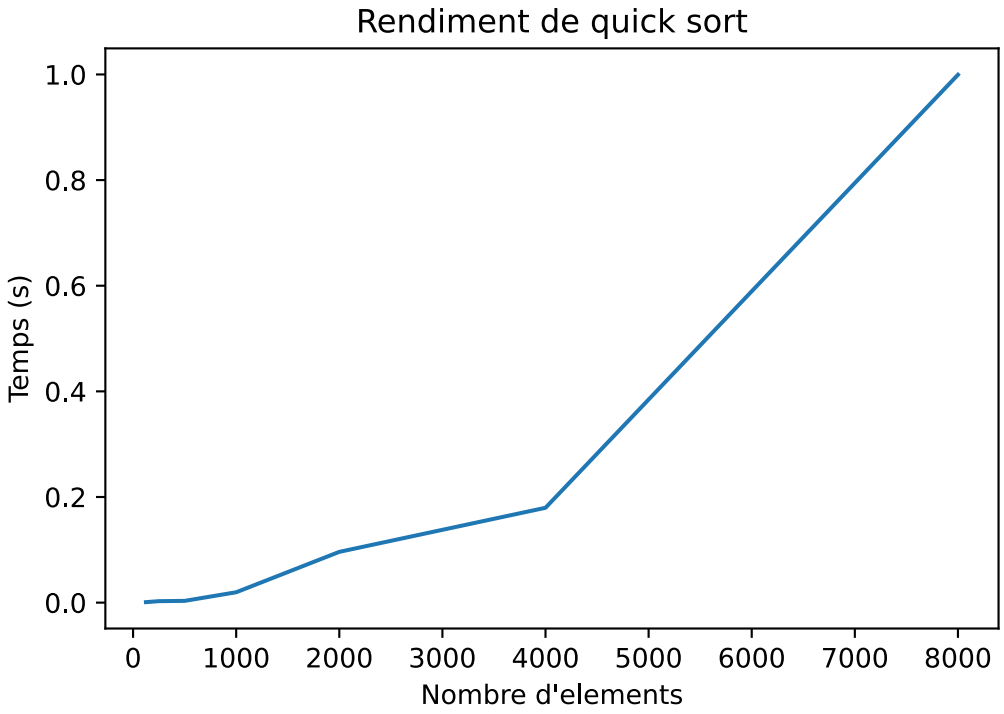
8.2. Pseudocodi

8.3. 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)
```

```
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



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 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.

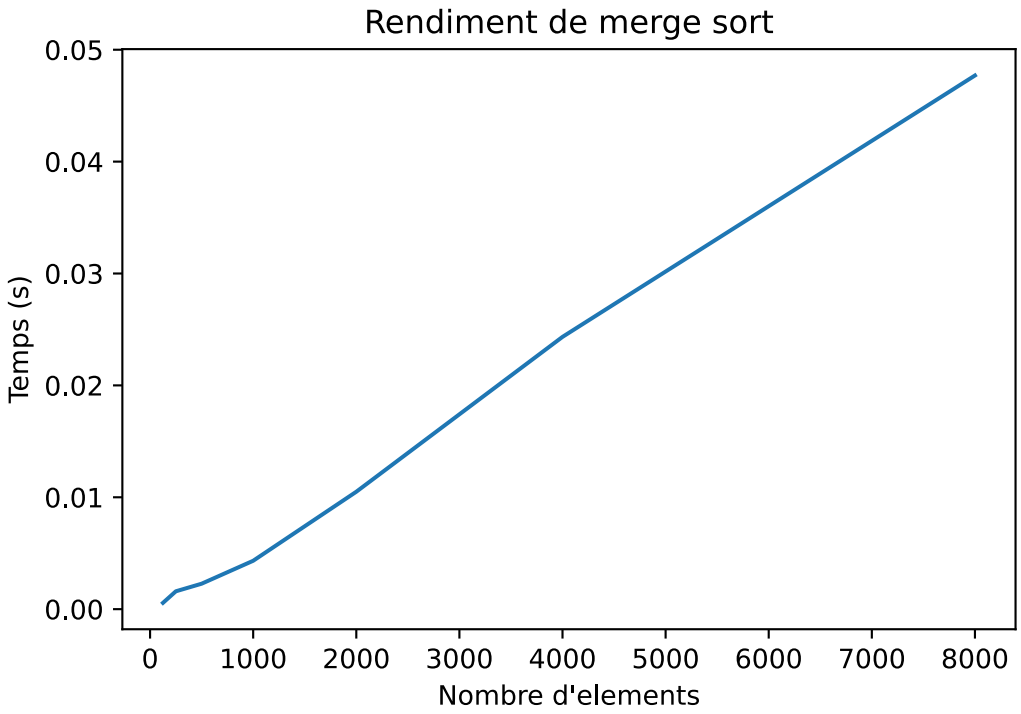
9.2. Pseudocodi

9.3. Implementació

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

def sort(array):
    if len(array) < 2:
        return array
    mid = len(array)//2
    left = sort(array[:mid])
    right = sort(array[mid:])

    merged = []
    l = 0
    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



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

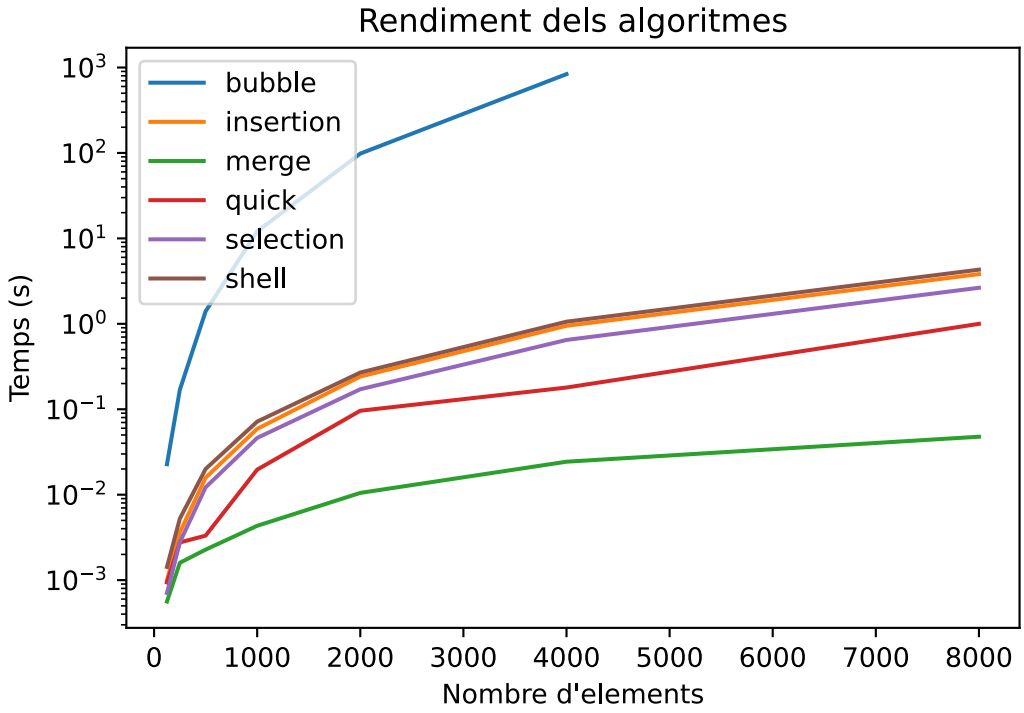
L'estudi del cost computacional d'un algoritme és l'estudi de la quantitat de recursos que consumeix. Habitualment com a recurs es pren la memòria o, i aquest és el nostre cas, el temps.

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.

Mentre el temps de computació d'un algoritme depèn del maquinari que l'executa la relació entre dues execucions amb diferent nombre d'elements no. Un algoritme que en duplicar el nombre d'elements duplica el temps d'execució ho farà independentment del maquinari. Això es deu al fet que cada instrucció (una comparació de dos elements o un canvi de posició) triga pràcticament el mateix, una fracció de mil·lisegon, en una mateixa màquina. Per tant quan es mesura el temps realment es mesura el nombre de passos.

10.1.1 Càlcul

Comencem amb l'exemple d'abans, un algoritme que amb un nombre n d'elements que en duplicar el nombre d'elements duplica el temps, és a dir, el nombre de passos.


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