Treball de Recerca:  
Algoritmes d’ordenació

Joan Coma Bages

Curs 2020/2021

**Table of Contents**

[1. Introducció 3](#__RefHeading___Toc1554_401841249)

[2. Què és ordenar un vector? 3](#__RefHeading___Toc1556_401841249)

[2.1. Què podem ordenar? 3](#__RefHeading___Toc1558_401841249)

[2.2. Per què ordenem? 3](#__RefHeading___Toc1560_401841249)

[3. Selection sort 4](#__RefHeading___Toc1562_401841249)

[3.1. Com funciona? 4](#__RefHeading___Toc1564_401841249)

[3.2. Pas a pas 4](#__RefHeading___Toc1566_401841249)

[3.3. Implementació 4](#__RefHeading___Toc1568_401841249)

[3.4. Rendiment 4](#__RefHeading___Toc1570_401841249)

[4. Insertion sort 4](#__RefHeading___Toc1572_401841249)

[4.1. Com funciona? 4](#__RefHeading___Toc1574_401841249)

[4.2. Pas a pas 5](#__RefHeading___Toc1576_401841249)

[4.3. Implementació 5](#__RefHeading___Toc1578_401841249)

[4.4. Rendiment 5](#__RefHeading___Toc1580_401841249)

[5. Bubble sort 5](#__RefHeading___Toc1582_401841249)

[5.1. Com funciona? 5](#__RefHeading___Toc1584_401841249)

[5.2. Pas a pas 5](#__RefHeading___Toc1586_401841249)

[5.3. Implementació 5](#__RefHeading___Toc1588_401841249)

[5.4. Rendiment 6](#__RefHeading___Toc1590_401841249)

[6. Shell sort 6](#__RefHeading___Toc1592_401841249)

[6.1. Com funciona? 6](#__RefHeading___Toc1594_401841249)

[6.2. Pas a pas 6](#__RefHeading___Toc1596_401841249)

[6.3. Implementació 6](#__RefHeading___Toc1598_401841249)

[6.4. Rendiment 7](#__RefHeading___Toc1600_401841249)

[7. Recursivitat 7](#__RefHeading___Toc1602_401841249)

[7.1. Factorials 7](#__RefHeading___Toc1604_401841249)

[7.1.1. Implementació no-recursiva 7](#__RefHeading___Toc1606_401841249)

[7.1.2. Implementació recursiva 7](#__RefHeading___Toc1608_401841249)

[8. Successió de Fibonacci 8](#__RefHeading___Toc1610_401841249)

[8.1.1. Implementació no-recursiva 8](#__RefHeading___Toc1612_401841249)

[8.1.2. Implementació recursiva 8](#__RefHeading___Toc1614_401841249)

[8.2. Torres de Hanoi 9](#__RefHeading___Toc1616_401841249)

[8.2.1. Implementació recursiva 9](#__RefHeading___Toc1618_401841249)

[9. Quick sort 10](#__RefHeading___Toc1620_401841249)

[9.1. Com funciona? 10](#__RefHeading___Toc1622_401841249)

[9.2. Pas a pas 10](#__RefHeading___Toc1624_401841249)

[9.3. Implementació 10](#__RefHeading___Toc1626_401841249)

[9.4. Rendiment 11](#__RefHeading___Toc1628_401841249)

[10. Merge sort 11](#__RefHeading___Toc1630_401841249)

[10.1. Com funciona? 11](#__RefHeading___Toc1632_401841249)

[10.2. Pas a pas 11](#__RefHeading___Toc1634_401841249)

[10.3. Implementació 11](#__RefHeading___Toc1636_401841249)

[10.4. Rendiment 12](#__RefHeading___Toc1638_401841249)

[11. Comparació dels algoritmes 12](#__RefHeading___Toc1640_401841249)

[11.1. Cost Computacional 12](#__RefHeading___Toc1642_401841249)

[11.1.1. Càlcul 12](#__RefHeading___Toc1644_401841249)

# Introducció

# 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.

## 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:

1. inventar-nos un sistema de referència: primer les cols, després les pastanagues, i finalment els espinacs o
2. 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.

## 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 es trobés entre la + 6 i la - 2 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.

# Selection sort

## Com funciona?

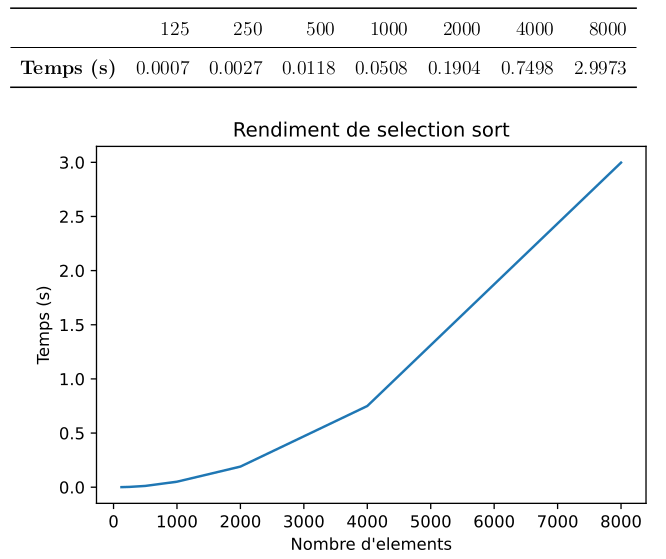
Aquest algoritme itera sobre la llista buscant el nombre més baix, que col·loca al principi. A continuació repeteix el mateix procés buscant el segon nombre més baix, però no des de l’inici sinó des de la segona posició (sabem que en la primera hi trobem el nombre més baix). Aquest procediment es repeteix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

## Pas a pas

## Implementació

v#!/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))

## Rendiment



# Insertion sort

## 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.

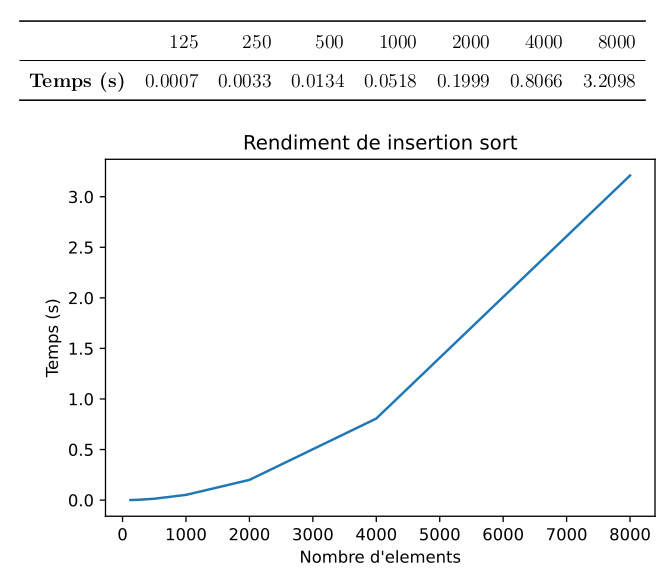
Pas a pas això vol dir agafar d’entrada un sol element, el primer, que compararem amb els anteriors. En ser el primer element no n’hi ha d’anteriors, el considerarem ordenat. En la següent iteració agafem el segon element, aquest el compararem progressivament amb els anteriors. El nostre element, que 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. 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 deixaríem al principi de la llista.

## Pas a pas

## 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()  
 print(sort(array))

## Rendiment



# Bubble sort

## Com funciona?

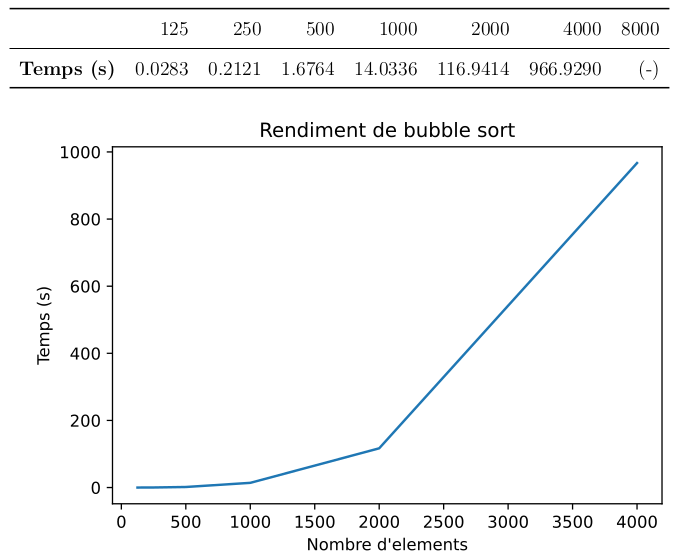
El bubble sort compara cada element de la llista amb el següent, canviant-ne la posició si el segon és més gran que el primer. Aquest procés es repeteix tantes vegades com faci falta per ordenar la llista.

## Pas a pas

## 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))

## Rendiment



# Shell sort

## 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.

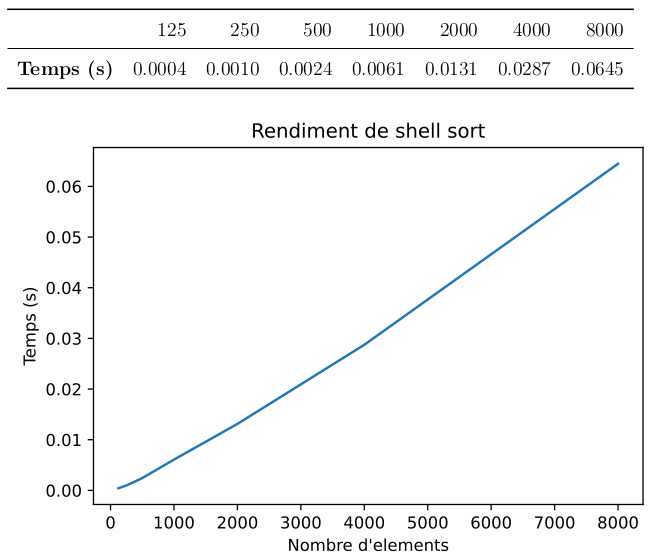
Comença agafant el primer element, i no el compara amb el segon sinó amb l’enèsim, el que sigui més petit ocupa la posició del primer. Continua així fins que arriba al final de la llista, quan torna a començar però fent salts (entre el primer i l’enèsim element) més petits. L’última iteració es fa amb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

## Pas a pas

## 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))

## Rendiment



# 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).

## Factorials

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

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

### 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)}")

### 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)}")

## Successió de Fibonacci

La successió de Fibonacci és una successió de nombres enters en què un nombre en posició 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 , el tercer , el quart , etc.

### 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(  
 f"El nombre en posició {n} de la successió de fibonacci és: {fibonacci(n)}")

### Implementació recursiva

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

## 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 com el nombre de moviments necessaris per resoldre el problema per un nombre de discs veiem que per , doncs només hi ha un sol disc i per moure’l a una altra fusta només cal un moviment.

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

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

### 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 inclou la resolució per un nombre de discs .

#!/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')

# Quick sort

## 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.

## Pas a pas

## 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  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 array = utils.numbers()  
 print(sort(array))

## Rendiment

# Merge sort

## 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.

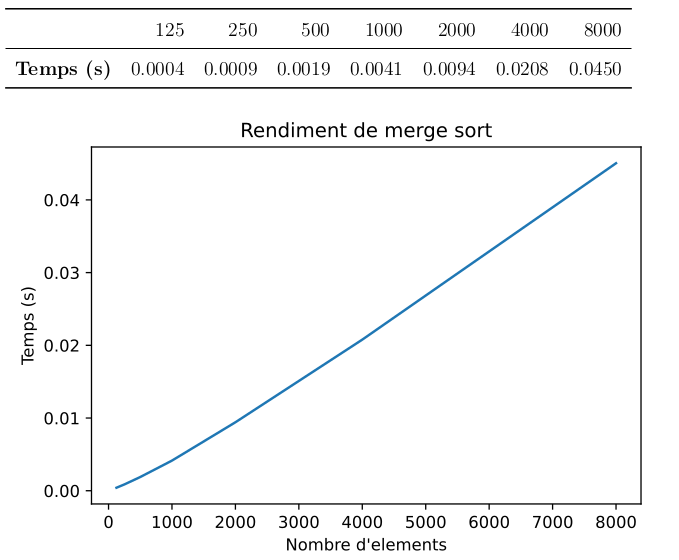
Una vegada tenim les parelles (llistes de dos elements ordenats) podem reconstruir la llista següent: el que fa l’algoritme és agafar l’element més petit de una llista (el primer) i comparar-lo amb el més petit de l’altra. L’element més petit d’aquests dos es posa a la nova llista. El procés es va repetint 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.

## Pas a pas

## 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  
  
 while l < len(left) and r < len(right):  
 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))

## Rendiment



# 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.

## Cost Computacional

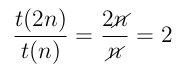
L’estudi del cost computacional d’un algoritme és l’estudi de la quantitat de recursos que consumeix. Habitualment com a recurs es pren la memòria o, i aquest és el nostre cas, el temps.

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.

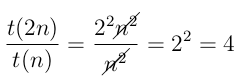
El temps de computació d’un algoritme sempre depèn de la quantitat d’elements, és d’esperar que el que es triga a ordenar deu elements sigui menor que el d’ordenar-ne cent. El temps de computació també depèn del maquinari, però aquest factor el podem eliminar si observem la relació entre dues execucions amb diferent nombre d’elements no. 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.

### Càlcul

Comencem posant xifres a l’exemple anterior. Si el temps d’execució de l’algoritme per elements, és a dir el , és val també , en duplicar el nombre d’elements serà . Veiem que el temps es duplica en duplicar el nombre d’elements amb la següent operació:

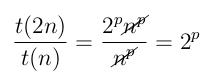


I què canviaria si en comptes de fos, en comptes de , fos de ?



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

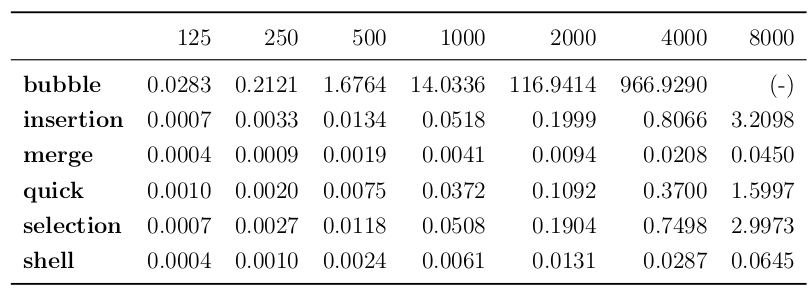
Generalitzant això veiem que quan el d’un algoritme és de i se’n dupliquen el nombre d’elements el temps d’execució és de . Dividint entre podem obtenir la relació entre ambdós:



Si seguim generalitzant trobem que per una relació entre les mides dels vectors la relació entre els temps és .

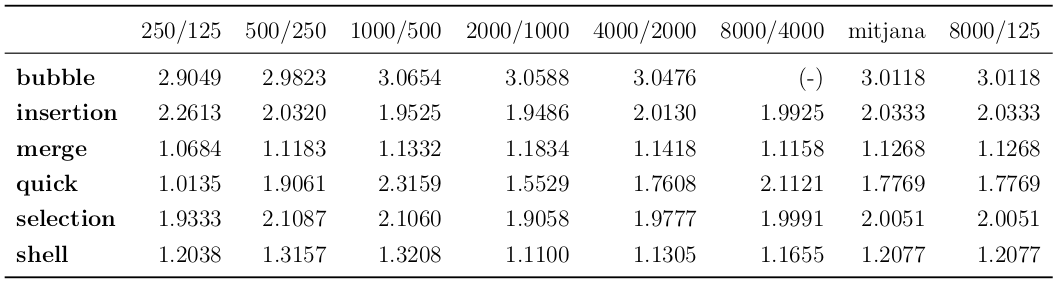
Poder aproximar el valor de l’exponent serà clau per conèixer el temps de computació. La relació entre un vector de 10 elements i un de 10000 és . Amb un exponent d’1 només ens costaria mil vegades més ordenar un vector mil vegades més gran, ja que . Si aquest exponent fos 2, el vector mil vegades més gran trigaria un milió de vegades el temps original, .

Aïllarem de la següent manera:



Prenent els temps d’aquesta taula podem calcular per qualsevol algoritme. Si substituir per una quantitat d’elements i per 2 obtenim entre el nombre d’elements entre el gran i el petit per calcular l’exponent de dos vectors.

Ara, amb els temps d’execució de cada algoritme, podem calcular el valor de . Prenem com a el nombre d’elements i com a 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 calculat amb els vectors més petit i més gran. en aquest cas té valor de 64 (), excepte amb el bubble sort, que és de 32 (, no l’hem executat amb 8000 elements).



Observem que el merge sort té el menor exponent, , seguit del shell sort amb i el quick sort amb . Lleugerament per sobre de 2, els algoritmes de selecció i inserció tenen valors de de i . Finalment, molt per sobre la resta, el bubble sort amb l’exponent de