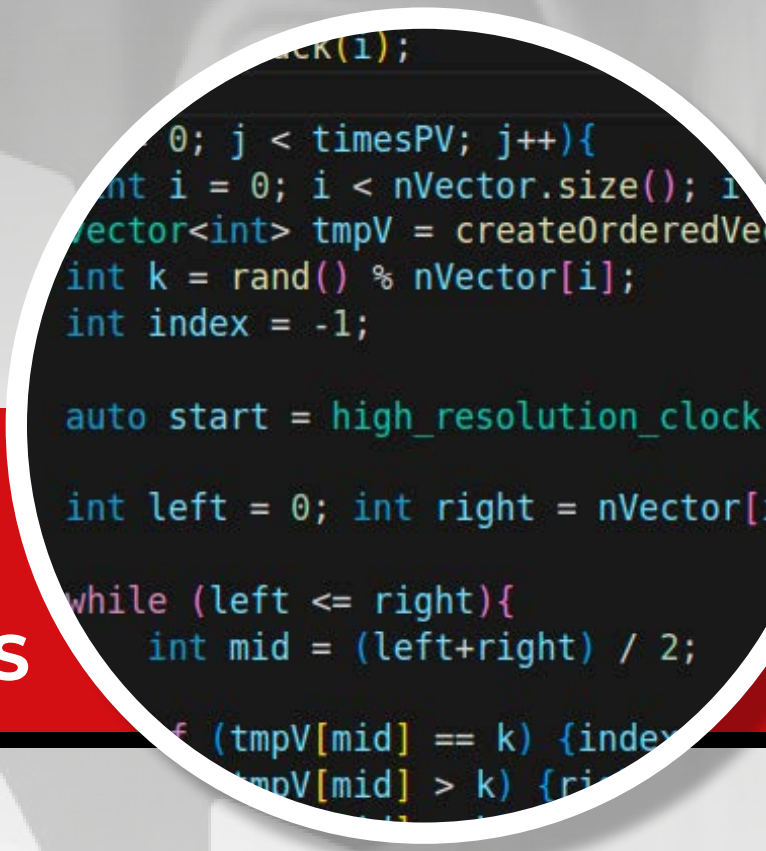
LA CERCA DE L'EFICIÈNCIA EN LA GESTIÓ DE DADES



Institut Els Pallaresos
Jordina Gavaldà Rovira
2n Bat B

Tutor: Jordi Irazuzta
Curs 2022-2023

ÍNDEX

ÍNDEX DE FIGURES

ABSTRACT

Technology is evolving very rapidly and competition between companies is fierce. They want their devices to be the safest, fastest and most efficient. In order to achieve that, they need very good hardware and software. The programs, procedures and routines that make up the software must be developed very carefully to be as efficient as possible. To further analyze these programs and procedures, we need to study the process they perform without considering the implementation. In other words, we need to study the algorithm they use to solve a particular computational problem.

In this paper we will study what algorithms are and how their time efficiency is analyzed. Then, two search algorithms and two sorting algorithms will be analyzed theoretically and practically and the results obtained will be compared. We will also briefly talk about algorithmic analysis when using nonlinear data structures. Finally, to visualize the algorithms, I have created two programs: one for sorting and searching algorithms and the other for pathfinding algorithms.

Keywords: algorithm, array, BFS, Big O notation, binary search, bubble sort, complexity, DFS, Dijkstra, efficiency, graphs, hardware, input, latex, linear search, merge sort, output, practical analysis, program, running time, searching, software, sorting, theoretical analysis, visualization.

INTRODUCCIÓ

Justificació del treball

Vivim en una societat en la qual utilitzem dispositius electrònics per a gairebé tot; per a comunicar-nos, treballar, oci, estudiar, jugar, comprar...

Les empreses del sector informàtic pretenen millorar aquests dispositius per fer-los més segurs, estables, ràpids, eficients... per destacar-se de la competència i guanyar quota de mercat.

Hi ha diverses formes de millorar els dispositius. Utilitzant millors components s'obtindrà un millor dispositiu. També és molt important la programació d'aquest, tant del sistema operatiu com de les aplicacions. No serveix de res tenir el millor maquinari (hardware) si a l'hora de realitzar qualsevol tasca el programari (software) no és eficient.

Per això, aquest treball se centra a analitzar una petita part del programari i implementar-lo, amb la intenció de mesurar teòricament i pràcticament el cost d'aquests procediments per poder comparar-los i determinar que tan eficients són. Analitzar l'eficiència dels procediments serveix per predir quin resoldrà una tasca més eficientment i també per optimitzar-los. Finalment, el resultat d'optimitzar el programari serà un dispositiu més eficient, i, per tant, més ràpid.

Personalment, des de fa un parell d'anys participo en concursos de programació, aquests con-

sisteixen a resoldre problemes de matemàtiques en què la solució és un programa. Aquest ha de resoldre el problema de la forma més eficient possible, i gairebé sempre s'utilitzen algoritmes.

En aquest treball ens centrarem en la part d'analitzar i programar alguns algoritmes, i per simplificar-ho i entendre bé els conceptes, utilitzaré problemes dels concursos com a exemples.

Objectius

En aquest treball podem definir dos tipus d'objectius, els que fan referència al contingut del treball:

- Estudiar i definir què és l'algorísmia i per a què serveix.
- Estudiar com s'analitza l'eficiència dels algoritmes.
- Estudiar de forma teòrica alguns algoritmes.
- Realitzar un programa per visualitzar els algoritmes estudiats.
- Implementar aquells algoritmes i comparar els resultats pràctics amb els teòrics.

I els que fan referència a la realització del treball:

- Programar el treball en LaTeX.
  - LaTeX és un sistema de composició de textos, orientat especialment a la creació de llibres, documents científics i tècnics que continguin fórmules matemàtiques.

Metodologia

Per assolir els nostres objectius, partirem el treball en dues parts teòriques i dues parts pràctiques.

Primer definirem alguns conceptes bàsics, explicarem que són els algoritmes, els programes, com s'analitzen i perquè és important fer-ho.

Tot seguit, veurem exemples més concrets d'algoritmes i els analitzarem com haurem explicat al capítol 1.

A continuació, programarem els exemples que haurem analitzat al capítol 2 i compararem els resultats teòrics (capítol 2) amb els resultats pràctics (capítol 3). També he implementat una visualització per entendre bé el funcionament d'aquests algoritmes.

Per últim, explicarem breument que és LaTeX , quines diferències hi ha entre un processador de textos i aquest llenguatge de programació, i perquè l'he utilitzat per redactar aquest treball.

CAPÍTOL 1

CONCEPTES BÀSICS

1.1 Com funcionen els dispositius electrònics

Els dispositius electrònics com els telèfons mòbils, tauletes, ordinadors, caixers automàtics, consoles, impressores, rentaplats, televisors, rentadores... tots resolen tasques diferents i tenen utilitats diferents, però, tots tenen els components necessaris per executar un programa.

1.1.1 Com interactuem amb els dispositius electrònics

Els dispositius són màquines que responen a accions com ara tocar la pantalla tàctil, clicar amb el ratolí, escriure amb el teclat, parlar pel micròfon, prémer algun botó del comandament a distància... i, el dispositiu genera una resposta en funció de l'acció de l'usuari: pot canviar el contingut de la pantalla, reproduir un so a través dels altaveus, enviar un senyal a un altre dispositiu, canviar el canal del televisor, retirar una quantitat de diners...

La informació que l'usuari dona al dispositiu l'anomenarem **entrada**. I al resultat o resposta que genera l'anomenarem **sortida**.

Per exemple, quan premem el botó d'incrementar el volum del comandament del televisor, generem una entrada. El televisor la processa i genera una sortida: incrementa el volum. Si

l'entrada fos diferent i premem el botó de canviar de canal, la sortida també seria diferent, i s'hauria canviat el canal en lloc d'incrementar el volum. Per tant, podem deduir que la sortida depèn completament de l'entrada. Si canviem l'entrada, la sortida també serà diferent.

El maquinari dels dispositius per si mateixos no poden processar les entrades i generar sortides, necessiten que algun element rebi la informació (entrada), la processi i generi el resultat (sortida). Aquí és on entra la programació. Els programes reben, llegeixen i processen la informació de l'entrada i generen la sortida.

Per exemple, premem el botó d'engegar del mòbil (**entrada**), l'entrada arriba al **programa**, aquest processa la informació i genera una **sortida**, la qual consisteix a encendre la pantalla.

Figura 1.1: Diagrama d'entrada, programa i sortida. Font: elaboració pròpia.

Totes les tasques que podem resoldre utilitzant un programa les anomenarem problema. Ens referim a tots els exemples d'aquest punt com a un problema que hem resolt amb un programa.

Quan parlem d'un problema matemàtic com ara sumar dos nombres, la solució és el resultat de la suma, l'equivalent a la sortida. Però quan parlem de problemes de programació, com els d'aquest treball, la solució és el procediment, el qual és diferent de la sortida, ja que la sortida és el resultat de finalitzar el procediment.

1.1.2 Què són els algoritmes i en què es diferencien dels programes

Un algoritme és un procediment, un seguit de passos finits i ordenats que resolen un problema concret. Els algoritmes ofereixen formes teòriques de resoldre un problema.

En canvi, un programa és tot allò que un ordinador pot entendre i executar. Un programa pot implementar[1] o "traduir" un algoritme. Els programes ofereixen formes pràctiques de resoldre un problema.

[1]La paraula 'implementar' s'utilitza en aquest context per expressar l'acció de passar alguna solució/procediment/algoritme a un programa. La implementació d'un algoritme és un programa que resolgui la mateixa tasca que l'algoritme implementat.

És a dir, els algoritmes són els procediments o els passos a seguir. I els programes són la traducció dels algoritmes de manera que els dispositius els puguin executar.

Posem com a exemple el següent problema que volem resoldre amb un programa: definim tres nombres com a x, y, z i volem trobar $(x + y) \cdot z$. Aquest problema està generalitzat perquè no tenim valors fixos per l'entrada i, per tant, no podem saber quina és la sortida. És a dir, farem un algoritme i programa que funcionin per a qualsevol nombre, no només per a un.

Per solucionar matemàticament aquest problema, primer cal resoldre la suma, i després la multiplicació. I per fer un programa que el resolgui, podríem fer el següent:

ALGORITME:

1. Llegir els nombres amb l'ordre adequat.
2. Sumar els dos primers nombres.
3. Multiplicar el resultat del pas 2 pel tercer nombre.
4. La solució és el resultat del pas 3.

PROGRAMA:

```
1  x = int(input())    # agafar l'entrada i assignar-li el valor x
2  y = int(input())    # agafar l'entrada i assignar-li el valor y
3  z = int(input())    # agafar l'entrada i assignar-li el valor z
4
5  a  = x + y          # pas 2 de l'algoritme
6  b = a * z           # pas 3 de l'algoritme
7  print(b)            # escriure la solució
```

Figura 1.2: Programa que resol l'operació[2]. Font: elaboració pròpia.

Com podem veure, l'algoritme és un seguit de passos i el programa permet executar-lo en un dispositiu, ja que l'escriu de forma que aquests el poden executar.

[2]El programa està escrit en Python. El propòsit és fer-se una idea de com és un programa, ja que en parlarem durant tot el treball. No cal entendre la sintaxi del llenguatge de programació. Després dels coixinets hi ha un comentari explicatiu que no afecta el funcionament.

Veiem com independentment de les entrades que tinguem, el procediment sempre serà el mateix. En un cas generalitzat com aquest, no tenim entrades i sortides concretes, però podem especificar un cas i d'aquesta manera podem comprovar el correcte funcionament del programa. Posaré un cas concret en què $x = 2, y = 3, z = 4$. En aquest cas, les entrades són 2, 3 i 4 i la sortida seria $(2 + 3) \cdot 4 = 20$.

1.1.3 Com se solucionen els problemes més complexos

En els exemples anteriors hem resolt problemes molt senzills, però quan hem d'abordar problemes més complexos, ens trobem amb altres dificultats. Pot passar que hi hagi moltes solucions diferents que resolen el problema, i hi haurà solucions poc eficients que haurem d'identificar i descartar. És molt important analitzar l'eficiència de les solucions per saber si és profitós implementar-les i executar-les.

Precisament en aquest treball ens centrarem a trobar diverses solucions per a un problema i analitzar-les per saber la seva complexitat en funció de la quantitat de dades de l'entrada. Ho explicaré més detalladament en el següent apartat. Ara per fer-nos una idea que un problema té diverses solucions unes més eficients que les altres, posaré un exemple:

El problema consisteix a fer un programa que trobi la solució a qualsevol sudoku[3]. Hi ha moltes estratègies que es poden seguir per resoldre'n un, seguidament en proposo dues.

La primera estratègia és posar tots els nombres a l'atzar i comprovar cada vegada si la solució és vàlida, i repetir el procés fins a trobar la combinació correcta. Quan apliquem aquest algoritme en un programa, aquest serà poc eficient, perquè hi ha moltes combinacions possibles, i inclús podríem repetir combinacions que ja hem provat.

La segona estratègia és la següent: Posem un nombre escollit de forma ordenada en una casella buida. Seguidament, comprovem que la fila, columna, i el quadre ens ho permet, en el cas que sigui vàlid, repetim el procés amb la següent casella buida, i, si no és vàlid, provem amb un nombre diferent. Amb aquesta solució estem provant nombres de forma ordenada i sense

[3]Aquest és un trencaclosques en què l'objectiu és omplir amb nombres de l'1 al 9 una quadrícula de 9x9 caselles dividida en quadres de 3x3. Les úniques restriccions són que no es poden repetir nombres en cap filera, columna o quadre 3x3. Inicialment, el trencaclosques et proporciona alguns nombres col·locats a la seva casella corresponent i l'objectiu és trobar-ne la resta.

repetir-los. Aquesta solució la podem generalitzar de la següent manera: si funciona avancem un pas i repetim el procés, si no funciona retrocedim i repetim el procés provant el següent nombre. Aquest tipus de solució és un algoritme de *backtracking*. La implementació d'aquesta solució es pot veure a l'annex 6.1.

1.2 Quina és la millor solució?

Quan podem solucionar un mateix problema de diverses maneres, el més lògic és resoldre'l de la manera més eficient possible. Si no, totes les tasques tardarien molt més en acabar i el resultat seria un dispositiu poc eficient. Per evitar això hem d'analitzar l'eficiència dels algoritmes.

1.2.1 Introducció a l'anàlisi de l'eficiència dels algoritmes

L'eficiència dels algoritmes o complexitat algorítmica és una mesura per determinar el cost d'un algoritme o programa. Amb aquesta mesura podem comparar els algoritmes o programes entre ells i determinar quin és més convenient d'utilitzar. La complexitat algorítmica depèn únicament de dos paràmetres que s'estudien per separat:

1. L'**eficiència espacial** (la quantitat de memòria que ocupa).
2. L'**eficiència temporal** (la velocitat de l'algoritme).

L'eficiència espacial fa referència a l'espai que ocupen les dades, les dades d'entrada s'han d'emmagatzemar perquè el programa les pugui utilitzar (bases de dades), i l'algoritme pot requerir emmagatzemar-ne més. Com més dades s'hagin d'emmagatzemar diem que l'algoritme té una pitjor eficiència espacial o major complexitat. Però en aquest treball no ens centrarem en aquesta part de l'estudi de l'eficiència.

Ens centrarem en l'eficiència temporal, la qual mesura el cost que té un algoritme o programa en funció de la mida de l'entrada.

Per exemple: un dentista tarda 30 minuts per visitar cada pacient. Per tant, per saber quan acabarà la jornada només hem de multiplicar el nombre de pacients per 30 minuts. Si anomenem n el nombre de pacients, podem saber que el dentista treballarà $30 \cdot n$ minuts. D'aquesta manera

estem expressant en funció del nombre de pacients (n) el temps que treballarà el dentista en una jornada.

L'eficiència temporal mesura el mateix. En funció de la quantitat de dades de l'entrada (n), expressa quant tardarà l'algoritme o programa en acabar.

L'eficiència temporal es pot analitzar de dues formes:

1. De forma **matemàtica**. Consisteix a analitzar la quantitat d'operacions que fa l'algoritme.
2. De forma **empírica**. Consisteix a implementar l'algoritme i mesurar el temps que tarda el programa. A partir d'aquestes mesures s'intenta predir el comportament del programa, i, per tant, de l'algoritme, per les mesures no realitzades.

Ha de quedar clar que l'eficiència temporal d'un algoritme no depèn en absolut de l'ordinador que utilitzem, sinó de l'algoritme en si mateix, és a dir, en la quantitat d'operacions o comparacions que faci. Un ordinador més potent podrà fer les operacions més ràpidament que un de més lent, però els dos faran la mateixa quantitat d'operacions, és a dir, serà més ràpid, però no més eficient.

Tornant a l'exemple del dentista, si un dentista B tarda 5 minuts per pacient en lloc del dentista A que en tarda 30. El dentista B tardarà $5 \cdot n$ minuts a atendre n pacients. El dentista B atén més ràpidament els pacients, però els dos realitzen el mateix procediment. Per tant, el dentista B és més ràpid, però, en termes d'anàlisi d'eficiència tots dos són igual d'eficients.

L'anàlisi empírica és poc fiable, ja que depèn de les dades d'entrada i de l'entorn de programació utilitzat. En canvi, en l'anàlisi matemàtica podem extreure conclusions basant-nos únicament en l'algoritme i no en les implementacions d'aquest. A més, un dels objectius principals d'analitzar l'eficiència dels algoritmes és determinar si és profitós implementar-lo, i això únicament es pot fer de forma teòrica analitzant l'algoritme.

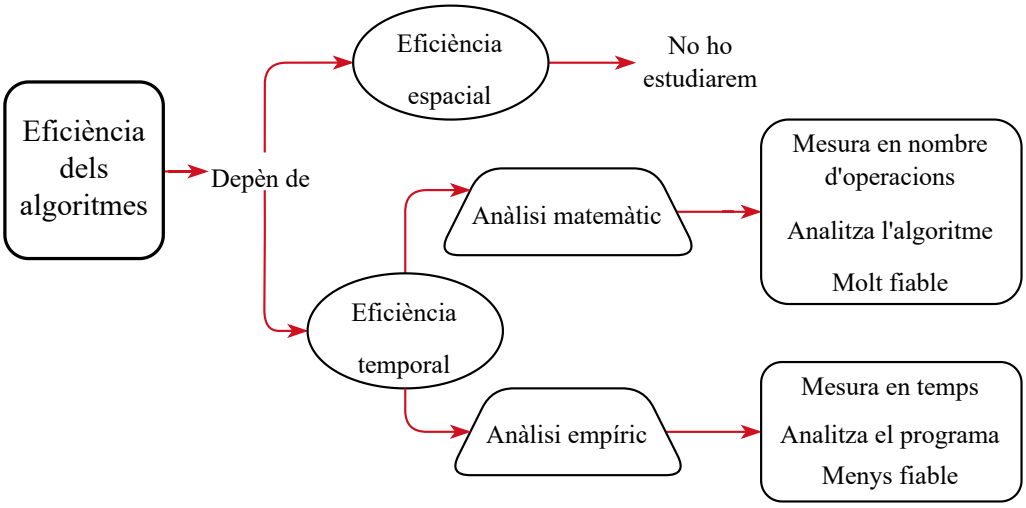


Figura 1.3: Diagrama de l'eficiència dels algoritmes. Font: elaboració pròpia.

1.2.2 Notació de Landau

Hi ha diverses maneres de mesurar la complexitat algorítmica respecte de l'entrada. La notació de Landau engloba tres notacions, $O()$, $\theta()$ i $o()$. En aquest treball només definirem i estudiarem $O()$.

La notació de Landau és una notació asimptòtica que proporciona una forma matemàtica d'expressar la complexitat d'un algoritme en funció d'n quan n pren valors molt grans. Aquest anàlisis és totalment independent de la implementació de l'algoritme.

Definim $O(g)$ com el conjunt de funcions que per a valors molt grans d'n (asimptòticament) creixen de manera proporcional o inferior a la funció g. Per tant, $f \in O(g) \iff f = O(g)$.

Per exemple, $f(n) = 5n^2 - 2n$ com que $f(n) \in O(n^2)$ podem afirmar que $5n^2 - 2n = O(n^2)$.

Si ens hi fixem, estem perdent informació de la funció $g(n)$, ja que $O(n^2)$ engloba totes les funcions de creixement inferior i proporcional a n^2. Això ens permet ignorar detalls com les constants o les funcions de menor ordre que no són rellevants en l'anàlisi de l'eficiència dels algoritmes.

Propietats d'$O()$ (només les necessàries per aquest treball):

1. $O(f) \cdot O(g) = O(f \cdot g)$
2. $O(f) + O(g) = O(f + g) = O(Max\{f, g\})$

A l'hora d'analitzar l'eficiència dels algoritmes, ens interessa saber de quina manera creix l'algoritme en funció a la mida de l'entrada. També ens interessa analitzar el pitjor cas possible, d'aquesta manera podem saber que en qualsevol altre cas, l'algoritme tindrà un cost menor o igual que en el cas analitzat.

El pitjor cas possible fa referència a l'entrada que perjudica més el funcionament d'aquell algoritme concret. El pitjor cas possible és diferent per a cada algoritme i n'hi ha que no en tenen.

Cal remarcar que $O(g)$ no és el nombre d'operacions que realitza un algoritme en el pitjor cas possible. Si no que a partir d'analitzar el nombre d'operacions f, $f = O(g) \iff f \in O(g)$.

Seguidament, explicaré les quatre complexitats més comunes que són les que tenen afectació en l'elaboració d'aquest treball.

Constant $O(1)$

En aquest cas a l'interior dels parèntesis no hi ha cap n, això vol dir que les operacions no depenen de la mida de l'entrada i sempre, independentment de la seva mida l'algoritme farà la mateixa quantitat d'operacions.

Per exemple, si agafem un llibre d'un prestatge d'n llibres i saps que vols el primer. No caldrà que el busquis, podràs agafar el primer fent una única operació. No importa quants llibres hi hagi, sempre faràs una sola operació. Per això, el nombre d'operacions és constant i no depèn d'n (la mida de l'entrada).

A l'hora d'analitzar un algoritme trobarem moltes operacions constants (comparacions, assignacions o increments), com que $O(1)$ sempre és inferior a qualsevol altre cost, molts cops no el tenim en compte, ja que quan simplifiquem l'expressió, sempre ens quedem amb el cost més gran (propietat 2). A més, les constants depenen de l'ordinador on s'executin.

Si fem un gràfic d'aquesta complexitat i posem a l'eix d'abscisses la mida de l'entrada (n), i a

l'eix d'ordenades el nombre d'operacions, obtenim la figura 1.4.

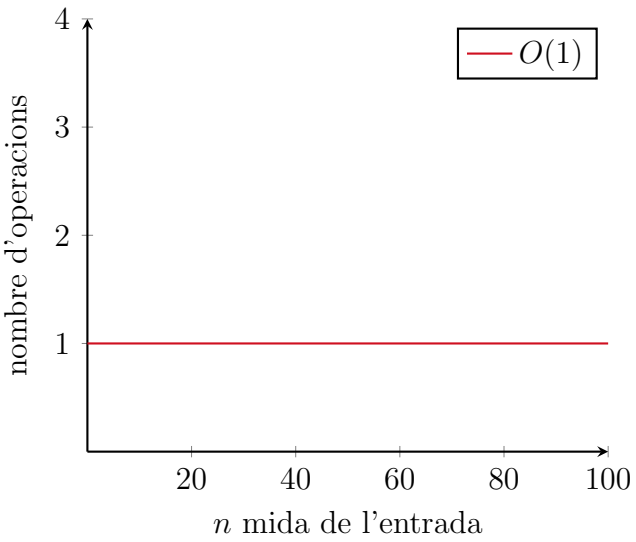


Figura 1.4: Gràfic de complexitat constant. Font: elaboració pròpia.

Podem veure com a mesura que l'entrada es va fent cada vegada més gran, el nombre d'operacions es manté constant.

Per tant, si hem d'utilitzar un algoritme de complexitat constant per a $n = 1.000.000$, farà la mateixa quantitat d'operacions que per a $n = 10$. I la velocitat d'aquestes operacions dependrà completament de factors externs a l'algoritme (ordinador, implementació, etc.).

Lineal $O(n)$

A mesura que incrementa n el temps també incrementa de forma directament proporcional.

Per exemple, si busques un llibre en un prestatge, hauràs de comprovar cada llibre fins a trobar el que busques. En el pitjor cas possible el llibre estarà situat l'últim del prestatge, i hauràs de comprovar-los tots fins a trobar-lo. Si hi ha n llibres, trobar el teu llibre t'haurà costat n iteracions com a màxim. Per això, aquest procediment té complexitat $O(n)$, ho veurem amb més detall al següent capítol.

L'exemple del dentista també té complexitat lineal, ja que fa n visites. Ens és igual si tarden $30 \cdot n$ minuts, $5 \cdot n$ minuts, o qualsevol constant per n, ja que per definició tots aquests casos pertanyen a $O(n)$.

En el gràfic 1.5 podem veure que a mesura que incrementa la mida de l'entrada, també incrementa el nombre d'operacions en la mateixa mesura. Per a una mida d'entrada petita el nombre d'operacions incrementa igual de ràpid que per a mides d'entrada més grans.

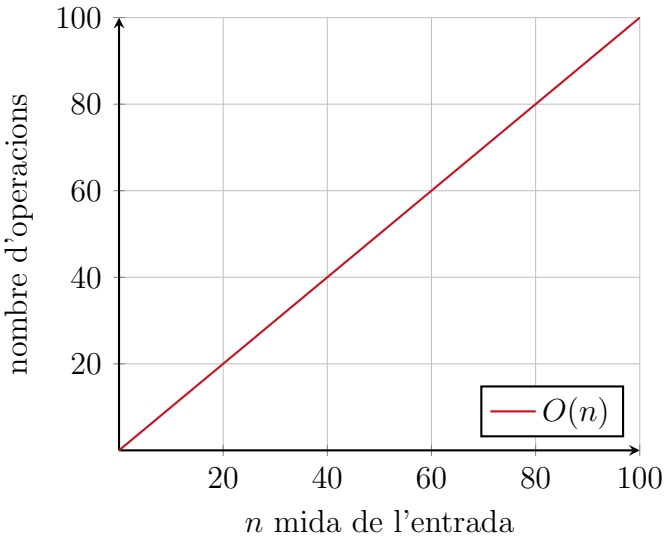


Figura 1.5: Gràfic de complexitat lineal. Font: elaboració pròpia.

Quadràtic $O(n^2)$

En aquest tipus d'algoritmes el nombre d'operacions és proporcional al quadrat d'n.

Per exemple, estem al supermercat i tenim una llista de la compra amb n productes. Però abans d'anar a pagar volem comprovar que no ens hàgim deixat res. Així que llegim el primer producte de la llista i el busquem a la nostra cistella, després fem el mateix fins a comprovar els n productes. Cada vegada que busquem el producte a la cistella, estem fent màxim n operacions, és a dir, buscar el producte a la cistella té un cost d'$O(n)$. Però estem fent aquest procés tantes vegades com productes apuntats a la llista (n). Així que estem fent n iteracions n vegades o $n \cdot n = n^2$ iteracions. Per tant, aquest procediment faria n^2 iteracions com a màxim.

En el gràfic de la figura 1.6 podem veure com aquesta complexitat és menys eficient que les altres, ja que el nombre d'operacions incrementa molt ràpidament. A diferència de la complexitat anterior, aquesta quan la mida de l'entrada és més petita incrementa més lentament, però a mesura que l'entrada és més gran, el nombre d'operacions incrementa cada vegada més ràpidament.

Recordem que sempre estem analitzant el pitjor cas possible, així que un algoritme mai farà més operacions de les previstes, però generalment en farà menys.

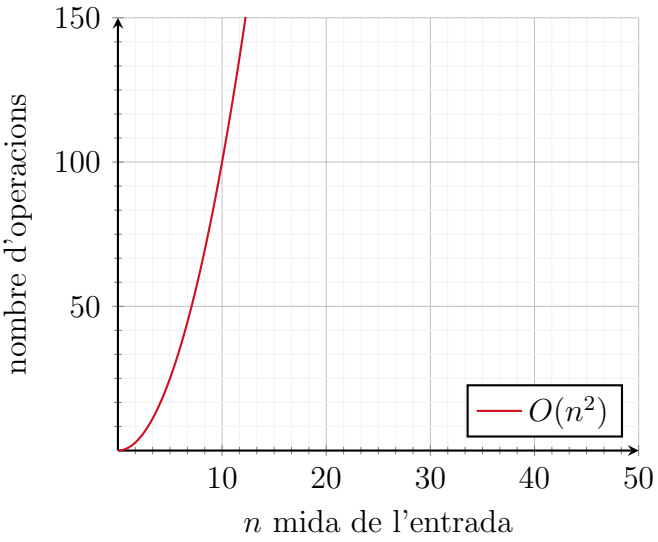


Figura 1.6: Gràfic de complexitat quadràtica. Font: elaboració pròpia.

Logarítmica $O(\log_2 n)$

Un algoritme té complexitat logarítmica quan la quantitat d'operacions és proporcional al logaritme d'n.

Per exemple: quan busquem una paraula en un diccionari en paper. Per buscar-la primer obrim el diccionari més o menys per la meitat i per ordre alfabètic deduïm si la paraula queda a les pàgines anteriors o posteriors de la partició. Després fem exactament el mateix amb la meitat que conté la paraula, i a poc a poc anem fent més petit el rang on podem trobar la paraula. Aquest procediment té complexitat logarítmica o $O(\log_2 n)$.

Els logaritmes són la inversa de l'exponencial (una constant elevada a n, com 2^n), i responen a quantes vegades (v) la base (b) es multiplica a ella mateixa fins a arribar a l'argument (n). És a dir $\log_b n = v \iff b^v = n$, per exemple: $\log_2 16 = 4 \iff 2^4 = 16$.

En l'exemple del diccionari per a $n = 16$ partim les pàgines per la meitat, després fem la meitat de la meitat... Així: $16 \to 8 \to 4 \to 2 \to 1$. Si ens hi fixem, aquesta seqüència és la mateixa que 2^4 però invertida: $2^4 = 2 \cdot 2^3 = 4 \cdot 2^2 = 8 \cdot 2 = 16$.

El procediment del diccionari parteix les pàgines exponencialment però de forma inversa, i el logaritme és la inversa de l'exponencial. Per això aquest procediment té complexitat logarítmica.

Una altra forma de resoldre un logaritme és dividir l'argument entre la base v vegades fins a obtenir el quocient d'1. En la figura 1.7 podem veure un exemple amb $\log_2 8 = 3$. És a dir, es necessiten 3 passos per partir 8 dades en 2 meitats fins a obtenir grups d'una dada.

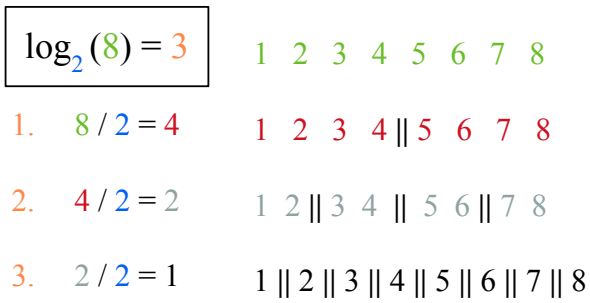


Figura 1.7: Relació entre partir nombres i els logaritmes. Font: elaboració pròpia.

Si el diccionari té n pàgines i $n = 128$, seguint el procediment de l'exemple anterior, haurem de fer $\log_2 n = \log_2 128 = 7$ passos per arribar a partir-lo en pàgines individuals. Per això diem que la complexitat d'aquest algoritme és $O(\log_2 n)$.

Si representem aquesta complexitat en el gràfic 1.8, podem veure com quan la mida de l'entrada és més petita, el nombre d'operacions creix més ràpidament, però a mesura que les n són més grans, cada vegada el nombre d'operacions creix més a poc a poc.

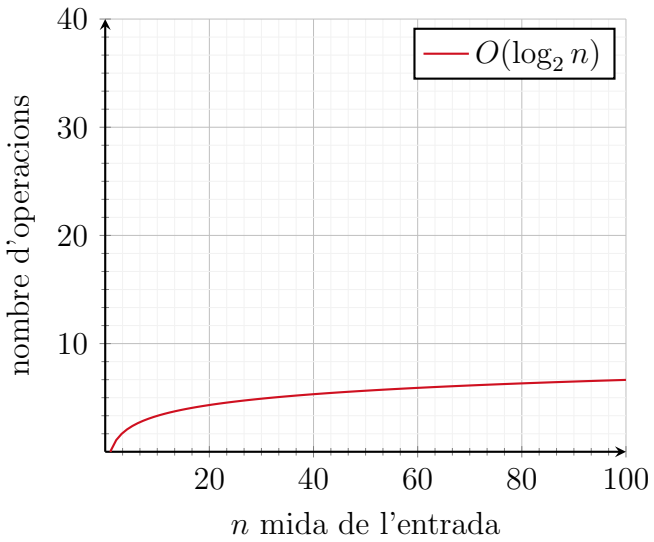


Figura 1.8: Complexitat logarítmica. Font: elaboració pròpia.

Per tant, per a $n = 10$, farà $\log_2 n = \lceil \log_2 10 \rceil = 4$ operacions i per a $n = 10^6$ farà $\log_2 n = \lceil \log_2 10^6 \rceil = 20$ operacions.

Totes les complexitats

Finalment, per comparar les diferents complexitats entre elles, podem fer un gràfic que les representi totes i una taula amb algunes dades per comparar-les. D'aquesta manera podem veure com la complexitat quadràtica té el cost més gran de les que hem explicat, i la complexitat logarítmica té menys cost excloent-hi la constant.

Els algoritmes de complexitat constant només es poden utilitzar per a problemes que no depenguin d'n, els quals són molt poc comuns i molt senzills, com per exemple operacions matemàtiques.

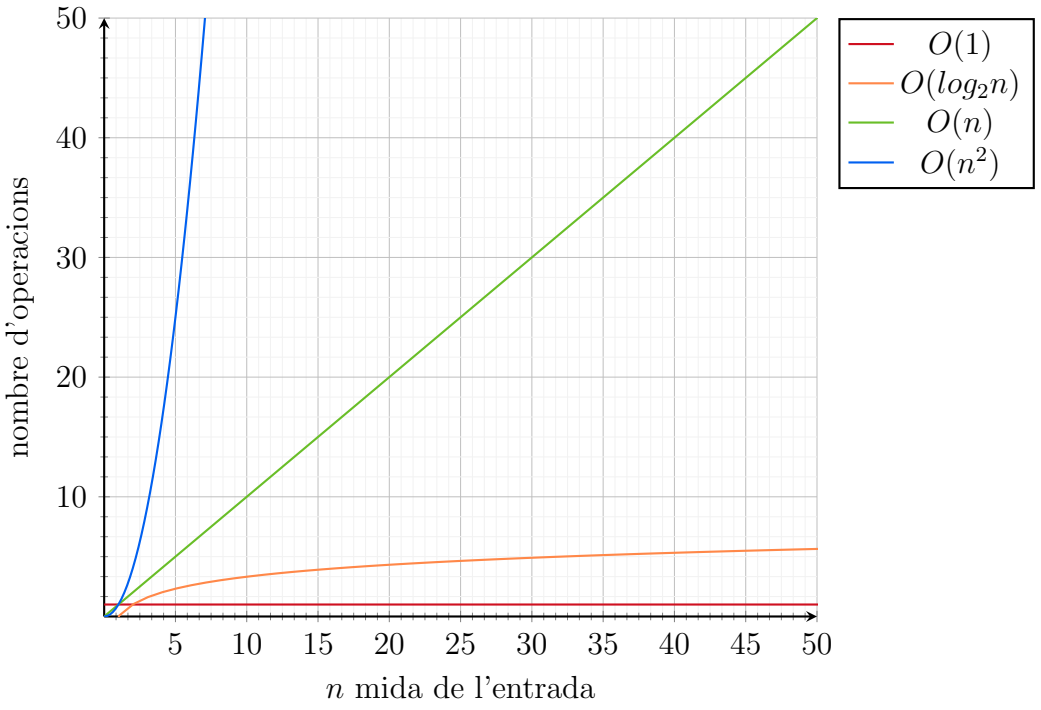


Figura 1.9: Gràfic amb totes les complexitats. Font: elaboració pròpia.

Complexitats \ n	10	1.000	10.000	50.000	10^6
$O(1)$	1	1	1	1	1
$O(\log_2 n)$	4	10	14	16	20
$O(n)$	10	1.000	10.000	50.000	10^6
$O(n^2)$	100	10^6	10^8	$25 \cdot 10^8$	10^{12}

Figura 1.10: Taula amb exemples del nombre d'operacions per a cada complexitat i mida d'entrada. Font: elaboració pròpia.

A més de les quatre complexitats elementals que hem explicat, n'hi ha més com la complexitat exponencial ($O(2^n)$) i la complexitat factorial ($O(n!)$) que tenen un cost molt superior a totes les altres. No les expliquem perquè no cal per als objectius d'aquest treball.

1.3 Anàlisi d'algoritmes per a estructures de dades no lineals

Les estructures de dades són diferents formes d'organitzar informació en un ordinador. En funció del programa que calgui implementar s'utilitzarà l'estructura de dades que sigui més convenient.

Depenent de l'estructura on guardem les dades per executar un algoritme, expressarem la complexitat amb diferents variables, ja que podem tenir diversos elements diferents com a entrada.

Les estructures de dades es classifiquen en dos tipus:

1. Lineals. Per exemple: llistes, cues, piles i cues de prioritat.
2. No lineals. Per exemple: grafs, arbres i taules hash.

En aquest treball hem explicat tots els exemples utilitzant llistes, unes estructures lineals en què es poden guardar seqüències de nombres o paraules. Però hi ha altres algoritmes per a resoldre problemes més complexos en què és necessari utilitzar estructures no lineals.

Per exemple, un problema molt important en la informàtica, és trobar el camí més curt entre dos punts. Guardar aquestes dades en una llista no és possible, per això hem d'utilitzar altres estructures de dades, en aquest cas un graf.

Un graf és un conjunt de vèrtex i arestes connectats entre elles. Els vèrtexs es representen amb cercles, i les arestes uneixen determinats vèrtexs. Es pot anar d'un vèrtex a un altre sempre que estiguin connectats per una aresta. El cost de passar per una aresta és 1 (excepte quan és un graf amb pesos a les arestes) i es pot anar en ambdues direccions si és un graf no dirigit com el de la figura 1.11, si no, s'anomena graf dirigit i les arestes es representen amb fletxes que indiquen el sentit.

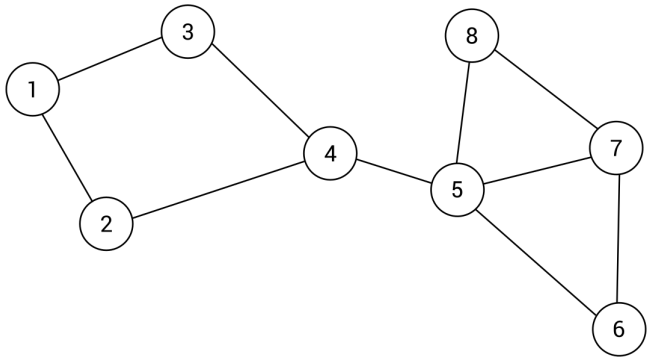


Figura 1.11: Exemple de graf no dirigit i sense pesos a les arestes. Font: https://www.oreilly.com/library/view/c-data-structures/9781788833738/0309b525-cfe4-429d-bdb9-53b25f94bbcd.xhtml.

Per exemple, Google maps utilitza grafs per calcular el camí més curt entre dos llocs. Les interseccions són els vèrtexs i les carreteres les arestes. En aquest cas són grafs dirigits amb pesos a les arestes i calen algoritmes molt ràpids per recalcular el recorregut a l'instant.

També s'utilitzen grafs a les xarxes socials, els usuaris són vèrtex i els seguidors s'uneixen amb arestes, d'aquesta manera les xarxes socials poden suggerir nous seguidors.

No entrarem en detall, però problemes en què estructurem les dades en una estructura de dades no lineal, se segueixen analitzant en funció de la mida d'entrada, però la mida d'entrada no la podem definir com a n, ja que necessitem més variables.

En el cas dels grafs, analitzem la complexitat en funció de la quantitat nombre d'arestes (A) i vèrtexs (V) que hi ha. A diferència de les llistes, utilitzarem dues variables per definir la mida de l'entrada.

Hi ha molts algoritmes de grafs diferents per a resoldre problemes diferents, però hi ha dos algoritmes bàsics per a recórrer o travessar un graf sense pesos a les arestes, aquests són el breadth-first search o BFS (complexitat temporal d'$O(V+A)$) i depth-first search o DFS (complexitat temporal d'$O(V)$). I per grafs amb pesos l'algoritme bàsic és el Dijkstra (complexitat temporal d'$O((V+A)\log V)$). Les implementacions d'aquests tres algoritmes les podeu trobar als annexos 6.2.

Aquests tres algoritmes són formes de recórrer el graf, és a dir, formes de visitar tots els vèrtexs de manera ordenada. Això és molt útil perquè variant una mica aquests algoritmes en funció del problema que es vulgui resoldre, es poden obtenir solucions a molts problemes diferents. Per això, tot i ser només formes d'iterar un graf, són algoritmes fonamentals per resoldre una gran varietat de problemes.

He fet una visualització de l'algoritme BFS implementat per a trobar el camí més curt entre dos punts en una matriu i el DFS implementat per indicar si hi ha un camí possible entre els dos punts de la matriu. Podeu trobar el programa en aquest enllaç: https://github.com/JordinaGR/TDR-visualitzacio-BFS-DFS o escanejant el codi de la figura 1.12.

Figura 1.12: Programa de la visualització d'algoritmes de grafs. Font: elaboració pròpia.

He fet un vídeo que mostra el funcionament del programa, es pot trobar en el següent enllaç: https://drive.google.com/file/d/1wqpY9PLo06W36MiOt74nirxrB-yl8TjB/view?usp=sharing o escanejant el codi de la figura 1.13.

Figura 1.13: Vídeo de la visualització d'algoritmes de grafs Font: elaboració pròpia.

CAPÍTOL 2

ANÀLISI MATEMÀTIC

Hi ha dos problemes bàsics i essencials que han de poder resoldre tots els dispositius. Aquests són: la cerca i l'ordenació. La cerca consisteix a trobar la posició d'una dada en un conjunt. I l'ordenació consisteix a arranjar un conjunt de dades seguint un criteri determinat.

Com tots els problemes hi ha moltes formes de solucionar-los, per això explicaré dos algoritmes per cada problema i els analitzarem per determinar el més eficient.

2.1 Cerca

Aquest problema consisteix a trobar la posició d'una dada en un conjunt. En aquest treball, per simplicitat, utilitzarem un conjunt d'n nombres enters. Per tant, aquest problema té dues entrades: el nombre que cerquem, que anomenarem k, i una llista o seqüència d'n nombres.

Com que la sortida del problema és la posició o índex de k a la llista, hem de definir quin índex té cada nombre. En programació normalment es comença a indexar des de zero, així que el primer element té índex 0, el segon té índex 1 així fins a $n-1$. Per exemple:

Veiem com la llista té $n = 4$ elements, així que l'últim element té l'índex d'$n - 1 = 4 - 1 = 3$.

Llista	48	5	12	83
Índex	0	1	2	3

Figura 2.1: Llistes i índex.

Per exemple, si les entrades d'un problema de cerca són la llista de la figura 2.1, i $k = 5$. Hem de trobar la posició de $k = 5$ en la $Llista = \{48, 5, 12, 83\}$. De manera que la sortida seria 1. Ja que la posició de $k = 5$ en la llista té índex = 1. També ho podríem expressar com $Llista_1 = 5$, ja que a l'índex 1 de la llista hi ha l'element 5.

2.1.1 Cerca lineal

L'algoritme de cerca lineal consisteix en mirar cada element un per un i comprovar si coincideixen amb el que estem buscant (k). Aquest algoritme acaba quan trobem l'element o hem comprovat tota la llista.

Per exemple, si tenim una $Llista = \{6, 2, 1, 8, 4\}$ i $k = 8$. L'algoritme farà el següent:

Anomenem i l'índex del nombre que estem comprovant (i incrementa a cada pas). Així, quan es compleix $Llista_i = k$, la sortida serà igual a i.

1. Comprova el primer element ($Llista_0 = 6$), com que aquest no és igual a $k = 8$, passa al següent, $i = i + 1$.
2. $Llista_1 = 2$, com que $2 \neq k$, passa al següent.
3. Després comprova $i = 2$ com que $Llista_2 \neq k$ comprova el següent.
4. Finalment $Llista_3 = k$, la sortida és $i = 3$ i acaba l'algoritme.

Ho podem representar gràficament amb la figura 2.2.

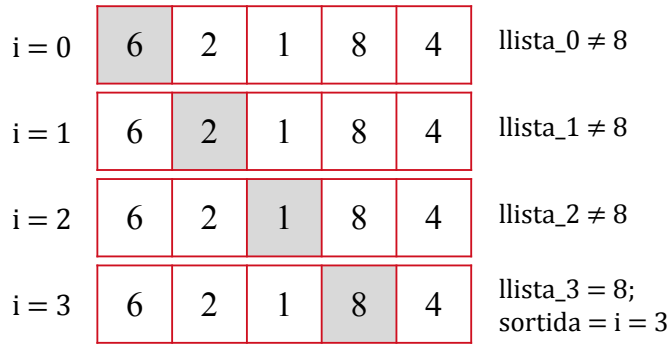


Figura 2.2: Exemple. Cerca lineal. Font: elaboració pròpia.

Pot passar que l'element no estigui a la llista, en aquest cas comprovaríem els n elements i quan acabés la llista, acabaria l'algoritme.

El pitjor cas possible de la cerca lineal és que l'element se situï en l'última posició o que no hi sigui a la llista. En ambdós casos hauríem d'iterar els n elements, i en cada iteració comprovar si l'element coincideix i si és així assignar l'índex a una variable. Iterar tots els elements suposa n operacions i per cada iteració cal fer 2 operacions. Així que aquest algoritme fa $2n$ operacions i $2n = O(n)$. Així que aquest algoritme té complexitat lineal.

Per veure en més detall el funcionament de l'algoritme, hi ha una implementació a l'annex 6.2.

2.1.2 Cerca dicotòmica

L'algoritme de la cerca dicotòmica resol el mateix problema que la cerca lineal, però amb el requisit que la llista ha d'estar ordenada. Aquest algoritme és el mateix que el de l'exemple del diccionari de la complexitat logarítmica.

Aquest algoritme consisteix a acotar el rang on podem trobar l'element fins que només en queda un i coincideix amb el que busquem. Per acotar el rang partim la llista per la meitat, i com que està ordenada, podem saber en quina meitat es troba k, de manera que a cada pas podem descartar la meitat dels elements que ens quedaven.

Per exemple, si tenim la $Llista = \{2, 4, 7, 12, 24, 38, 51, 56, 62, 65, 71, 83, 89, 98, 99\}$, $n = 15$ i $k = 62$. L'algoritme fa el següent:

1. k es troba en el rang 0 - 14 ambdós inclosos, per tant, calculem en quin índex queda la dada central, així, $\frac{0+14}{2} = 7$. Com que k és més gran que la dada de l'índex 7, podem descartar totes les posicions més petites a l'índex 7 inclòs. És a dir, com que $Llista_7 = 56 < k$ podem assegurar que k es troba en el rang 8 - 14 ambdós inclosos.
2. Repetim el pas 1 però en el rang 8 - 14 ambdós inclosos. La dada central es troba a l'índex $\frac{8+14}{2} = 11$. Com que $Llista_{11} = 83 > k$, k es troba a l'esquerra de l'índex 11, en el rang 8 - 10 ambdós inclosos.
3. Trobem la dada central de 8 - 10, $\frac{8+10}{2} = 9$, $Llista_9 = 65 > k$. Per tant, k es troba en el rang 8 - 8.
4. La dada central de 8 - 8 es troba en $\frac{8+8}{2} = 8$, $Llista_8 = 62 = k$. Per tant, acaba l'algoritme i la sortida és 8.

Ho representem en la figura 2.3. Els elements en verd representen els que no han estat descartats, els que estan en blau representen les dades centrals, i en vermell representen k. El pitjor

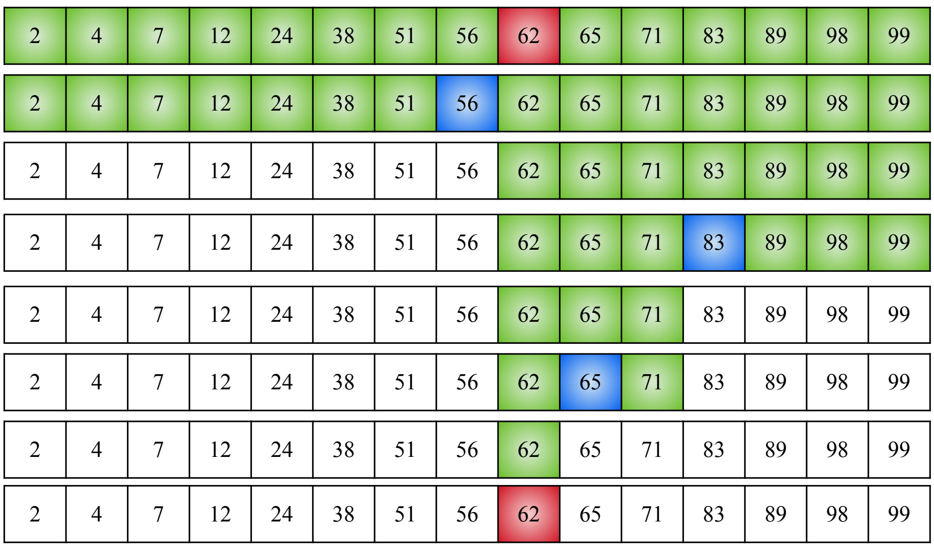


Figura 2.3: Exemple. Cerca dicotòmica. Font: elaboració pròpia.

cas possible és que k no es trobi a la llista, ja que hauríem d'arribar a dividir les dades d'una en una per assegurar-nos que no hi estigui. Això també ha passat en l'exemple de la figura 2.3, encara que k estigués a la llista. En canvi, si haguéssim escollit $k = 83$, haguésim acabat l'algoritme partint les dades una sola vegada.

Com hem explicat al punt 1.2.2, aquest algoritme té complexitat logarítmica, ja que $\lceil \log_2 n \rceil$ és la quantitat necessària de passos per partir les dades de la manera com ho fa la cerca dicotòmica. En la figura 1.7 es pot veure com es parteixen les dades igual que en aquest algoritme, i la relació que té amb els logaritmes.

Com hem vist, a cada iteració s'ha de buscar la dada central i en el pitjor dels casos fer tres comparacions (si $Llista_i = k, Llista_i > k, Llista_i < k$) i retornar l'índex. Així que més concretament aquest algoritme fa $\log_2 n \cdot 5$ operacions. Com que $5 \cdot \log_2 n = O(\log_2 n)$ aquest algoritme té complexitat logarítmica.

Per veure en més detall el funcionament de l'algoritme, hi ha la implementació a l'annex 6.3.

2.2 Ordenació

En aquest treball ordenarem ascendentment una llista de nombres enters. L'entrada d'aquest problema és una llista d'n elements. I la sortida consisteix en els n elements ordenats ascendentment.

Hi ha molts algoritmes diferents per resoldre aquest problema, i tots tenen avantatges i inconvenients. En aquest treball ens centrem només en l'eficiència temporal, però hi ha molts altres factors que també afecten l'eficiència de l'algoritme i depenent de la situació cal tenir-ho en compte. N'hi ha que són molt eficients temporalment, però ocupen molta memòria, en canvi, n'hi ha que tenen una menor eficiència temporal, però ocupen menys memòria. N'hi ha que són més convenients si hi ha moltes dades repetides, n'hi ha que són específics per a estructures de dades determinades (llistes, grafs, taules hash...).

2.2.1 Ordenació de bombolla

Aquest algoritme consisteix a intercanviar dos elements adjacents si no estan en ordre.

Per exemple, si tenim una $Llista = \{7, 2, 5, 3, 11\}$ i $n = 5$, l'algoritme farà el següent:

1. Comprova el primer i segon element (7 i 2), com que $7 > 2$ els intercanvia. I ens queda $Llista = \{2, 7, 5, 3, 11\}$.

2. Comprova el segon i tercer element. Com que $7 > 5$, els intercanvia. I queda $Llista = \{2, 5, 7, 3, 11\}$.
3. Comprova el tercer i quart element. Com que $7 > 3$, els intercanvia. I queda $Llista = \{2, 5, 3, 7, 11\}$.
4. Comprova el quart i cinquè element. Com que $7 < 11$, no els intercanvia. I la llista queda igual $Llista = \{2, 5, 3, 7, 11\}$.
5. Ara ja ha acabat la llista, i ha de repetir el mateix procediment n vegades en total.
   Comprova el primer i segon element. Com que $2 < 5$, queden igual. I queda $Llista = \{2, 5, 3, 7, 11\}$.
6. Comprova el segon i tercer element. Com que $5 > 3$, els intercanvia. I queda $Llista = \{2, 3, 5, 7, 11\}$.
7. Ara la llista ja està ordenada, però els ordinadors no ho poden saber si no ho comproven. I no ho comproven perquè això faria l'algoritme poc eficient, ja que cada comprovació té complexitat lineal[1]. Això comporta que en un ordinador aquest algoritme segueixi funcionant fins a repetir aquest procés un total d'n vegades, encara que no es canviarà d'ordre cap dada. De manera que no hi ha un pitjor cas possible, ja que en tots els casos es farà la mateixa quantitat d'operacions.

Aquest algoritme repeteix n vegades el mateix procediment. Aquest procediment itera fins a $n - 1$ i a cada iteració fa tres operacions per intercanviar dos elements. Per tant, l'ordenació de bombolla té una complexitat d'$n \cdot (n-1) \cdot 3 = 3n^2 - 3n = O(n^2)$. Per això aquest algoritme té complexitat quadràtica.

Per veure en més detall el funcionament de l'algoritme, hi ha la implementació a l'annex 6.4.

2.2.2 Ordenació per barreja

El funcionament d'aquest algoritme el podríem separar en dues parts: en la primera part divideix les dades igual que en la cerca dicotòmica (figura 2.4), i en la segona part les uneix fins que queden en una sola llista ordenada (figura 2.5).

[1]Per comprovar que la llista ha quedat ordenada, caldria iterar una vegada els n elements i comprovar que l'element actual és més petit al següent. Aquest procediment té un cost lineal $O(n)$

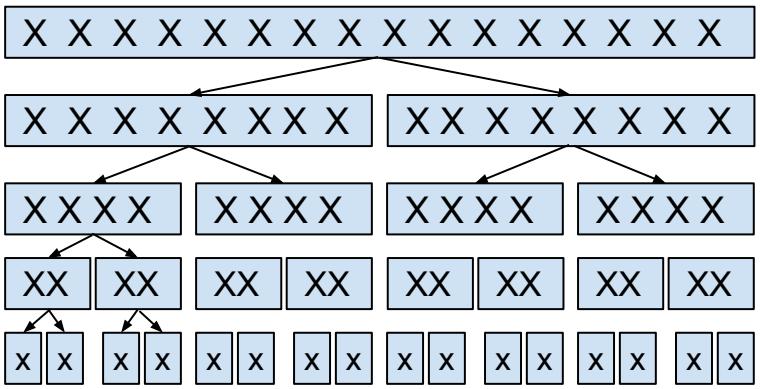


Figura 2.4: Divisió de les dades. Font: elaboració pròpia.

En la figura 2.4 podem veure com cada llista es divideix per la meitat fins a quedar dades individuals. Igual que en la cerca dicotòmica, i, per tant, també té complexitat logarítmica.

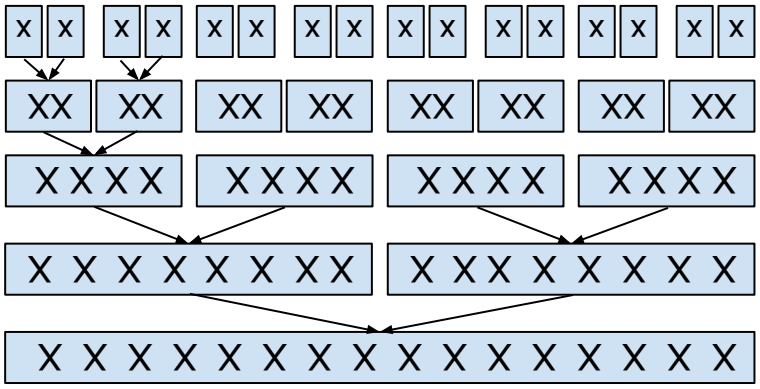


Figura 2.5: Unir les dades. Font: elaboració pròpia.

Unir les dades és molt simple. Com que comencem amb dades individuals sempre són grups de dades ordenades, per això les podem ordenar tan ràpidament. Primer mirem la dada més petita dels dos grups que volem unir, és a dir, la primera. Després posem la més petita de les dues a la següent llista i avancem una posició a la llista d'on hem tret la dada. Ho podem representar gràficament amb la figura 2.6.

Aquest procediment es repeteix cada vegada que s'ajunten dues llistes.

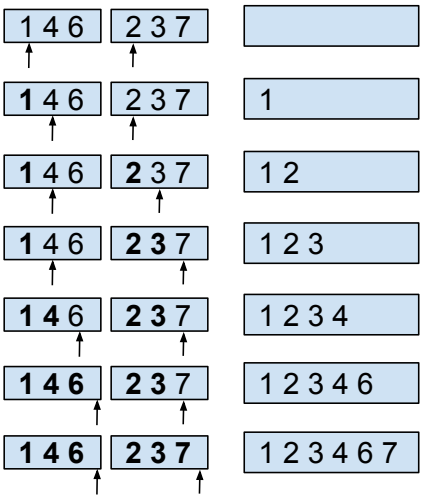


Figura 2.6: Exemple d'unió de les dades. Font: elaboració pròpia.

Podem posar un exemple d'aquest algoritme per a $n = 8$ i $Llista = \{5, 2, 4, 7, 1, 3, 2, 6\}$ en la figura 2.7.

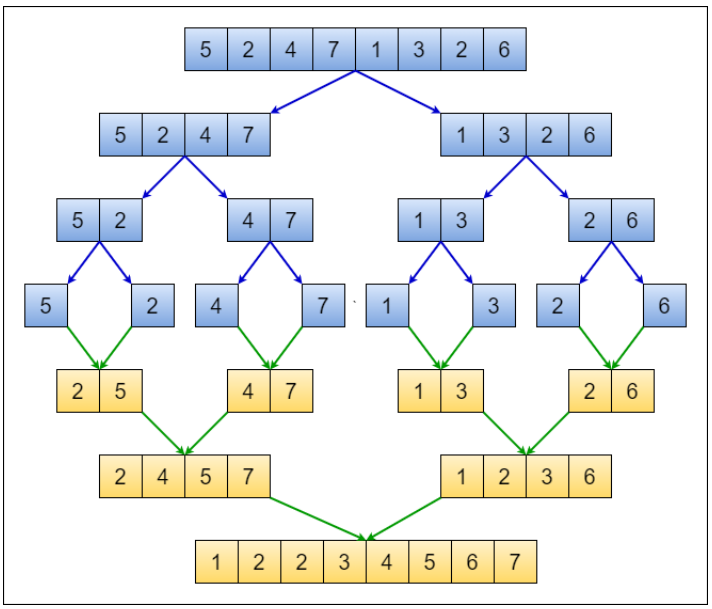


Figura 2.7: Exemple d'ordenació per barreja. Font: https://levelup.gitconnected.com/visualizing-designing-and-analyzing-the-merge-sort-algorithm-cf17e3f0371f.

En la primera part de l'algoritme (part blava) només es parteixen les dades, i en la segona part (part groga) s'ajunten com hem vist a la figura 2.6 fins a obtenir una única llista.

Analitzarem la complexitat d'aquest algoritme separant la part blava de la groga, i després sumarem els dos resultats. La part blava com ja hem vist anteriorment té complexitat logarítmica. I la part groga té complexitat $O(n \cdot log_2 n)$.

En la figura 2.6 hem ajuntat dues llistes de 3 elements per obtenir una llista d'$n = 6$, i hem fet n operacions, ja que hem recorregut cada llista una vegada. De manera que el procediment d'ajuntar dues llistes té complexitat lineal.

En la part groga de la figura 2.7 hi ha les mateixes particions que en la part blava però invertides. És a dir, hi ha $\log_2 n$ passos a la part groga, i a cada pas ajuntem els n elements de forma lineal. Els ajuntem en llistes diferents de mides diferents, però la suma d'aquestes subllistes tenen mida n. Així que en total fem un procediment lineal $\log_2 n$ vegades, $O(n \cdot \log_2 n)$.

Si ho ajuntem tot ens queda una complexitat de $O(\log_2 n)+O(n\cdot\log_2 n) = O(\log_2 n+n\cdot\log_2 n) = O(n \cdot \log_2 n)$, ja que en l'últim pas podem eliminar el primer $\log_2 n$ ja que és un terme de creixement més lent (propietat 2).

CAPÍTOL 3

COMPROVACIÓ EMPÍRICA

L'objectiu d'aquest capítol és entendre millor els algoritmes que hem explicat amb una visualització del seu funcionament i implementar-los en un llenguatge de programació per demostrar de forma pràctica les complexitats que hem analitzat al capítol anterior.

3.1 Visualització dels algoritmes

El programa de la figura 3.1 és una visualització dels algoritmes que hem explicat. Aquest representa gràficament les dades en rectangles proporcionals, i executa un algoritme. Els gràfics s'actualitzen a cada pas de l'algoritme, i d'aquesta manera podem veure perfectament el funcionament de cada un.

També mesura quant temps tarda cada un en executar-se i guarda les dades en un document .csv. Aquestes dades no ens serveixen per comprovar la complexitat dels algoritmes, ja que els gràfics alenteixen molt el funcionament de cada algoritme. Ja que estan implementats per ser visualitzats, no per ser molt eficients.

Aquest programa l'he fet en Python, ja que és un llenguatge de programació que té bones biblioteques gràfiques.

El programa està en aquest enllaç:
https://github.com/JordinaGR/TDR-visualitzacio-algoritmes o escanejant el codi de la figura 3.1.

Figura 3.1: Codi QR que porta a una pàgina web amb el codi del programa. Font: elaboració pròpia.

He fet un vídeo que mostra el funcionament del programa, es pot trobar en el següent enllaç:
https://drive.google.com/file/d/124msWkaHTGWdloKxFPQzvh6XZ1bwt1jO/view?usp=sharing
o escanejant el codi de la figura 3.2.

Figura 3.2: Codi QR que porta a vídeo amb el funcionament del programa. Font: elaboració pròpia.

3.2 Anàlisi empíric dels algoritmes proposats

En aquest apartat implementarem els algoritmes que hem explicat, els executarem per a mides d'entrada diferents i farem un gràfic amb els resultats. Més tard compararem el gràfic que obtinguem amb el gràfic de la complexitat analitzada matemàticament en el capítol 2.

Aquest cop he implementat els algoritmes en C++, ja que és un llenguatge de programació més ràpid que Python.

El programa en C++ executa l'algoritme i escriu les dades en un document .csv. Després he fet un altre programa en Python per calcular la mitjana del temps que tarden els algoritmes amb la mateixa mida d'entrada utilitzant la biblioteca Pandas, i generar un gràfic amb la biblioteca Matplotlib.

A l'hora d'executar el programa per recollir les mostres per fer els gràfics, m'he assegurat que l'ordinador no estigués fent altres tasques alhora. De manera que el rendiment de l'ordinador sempre sigui el mateix i minimitzar al màxim els factors externs a l'execució del programa.

El programa està en aquest enllaç: https://github.com/JordinaGR/TDR-complexitat-algs o escanejant el codi de la figura 3.3.

Figura 3.3: Codi QR que porta a una pàgina web amb el codi del programa. Font: elaboració pròpia.

Anàlisi empíric de la cerca lineal

He executat aquest algoritme un total de 60.000 vegades. Per evitar errors he executat 20 vegades cada mida d'entrada diferent i he calculat la mitjana. Les mides d'entrada les he augmentat en 20 unitats en el rang de 10 a 60.000.

Després he representat les dades en un gràfic de dispersió, i he obtingut el següent resultat:

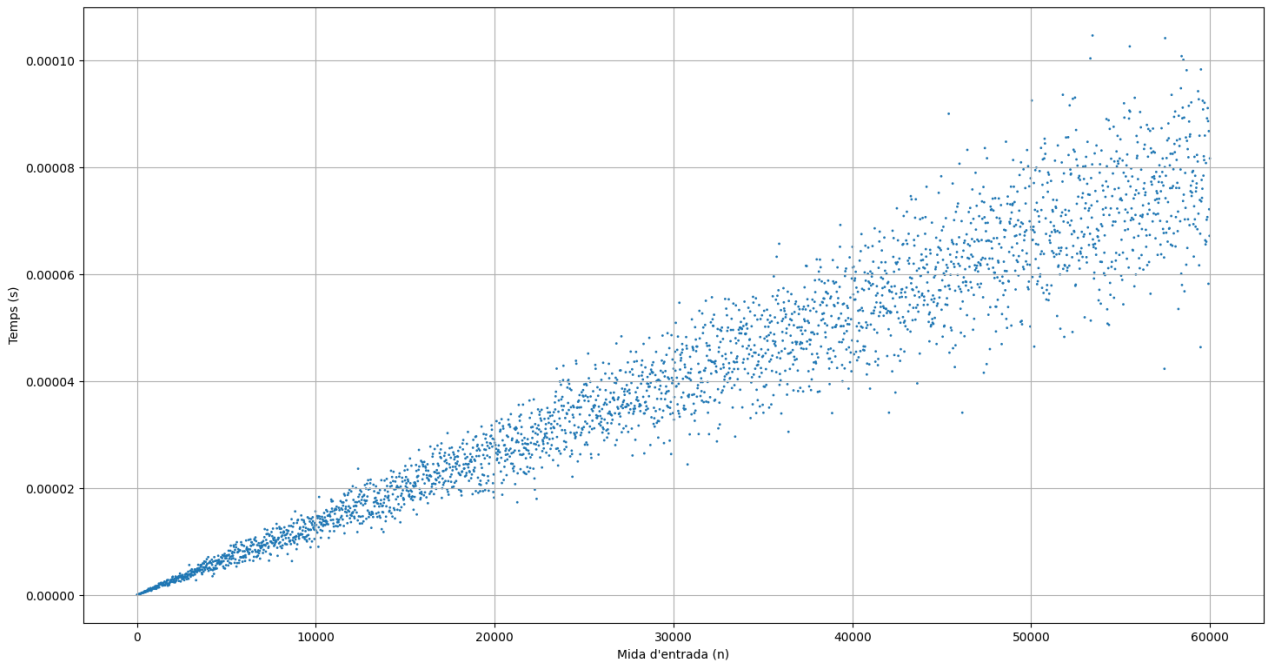


Figura 3.4: Gràfic de dispersió de la cerca lineal. Font: elaboració pròpia.

Podem veure en el gràfic de la figura 3.4 una certa relació entre les dues variables. No podem trobar una fórmula que determini el comportament d'aquest programa, ja que hi ha molts factors que afecten els resultats (ordinador, compilador, la dada a cercar...). En canvi, el que podem fer és buscar una correlació entre les dues variables.

En el gràfic 3.4 hi ha una correlació lineal, ja que el gràfic s'aproxima a una recta. Aquesta recta la podem trobar programant amb la biblioteca SciPy. I obtenim el gràfic 3.5:

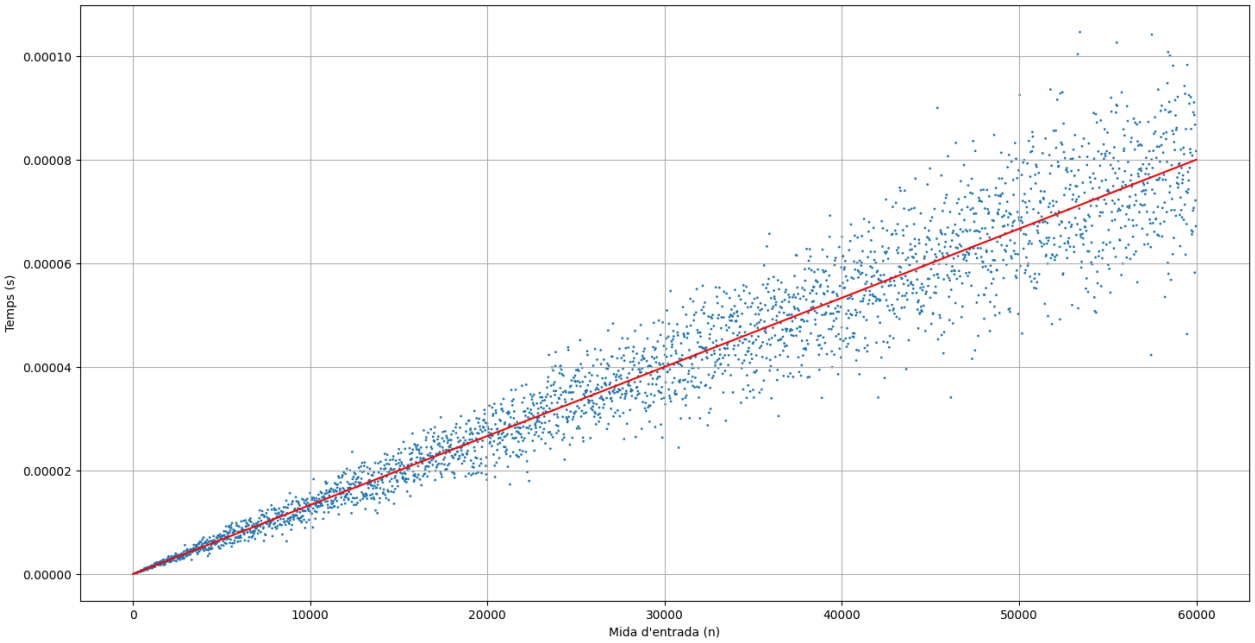


Figura 3.5: Gràfic de dispersió de la cerca lineal amb la recta de regressió lineal. Font: elaboració pròpia.

La recta en vermell és la recta que més s'acosta a tots els punts del gràfic ($r = 0.966$).

Si comparem els resultats teòrics que concloïen que la cerca lineal té complexitat $O(n)$ i tornem a fer un gràfic per comparar els resultats. Podem veure com els dos gràfics són lineals i, per tant, analitzant la cerca lineal de forma matemàtica i empírica hem arribat al mateix resultat.

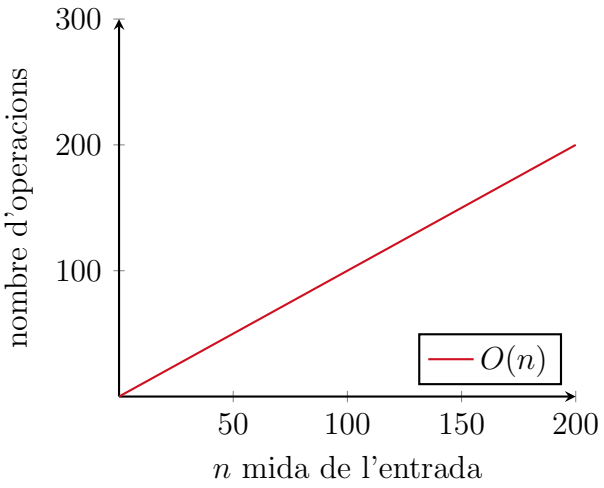


Figura 3.6: Gràfic de complexitat lineal. Font: elaboració pròpia.

En el gràfic 3.4 podem veure que com més creix la mida de l'entrada, les mostres s'allunyen més de la recta. Això és perquè com més gran és la mida de l'entrada, la dada que busquem k pot prendre més valors diferents. És a dir, el programa pot fer entre 1 i n operacions, i com més gran és n més gran és el rang, i per això tenim resultats molt diferents.

Per comprovar que això és cert, podem executar les mateixes 60.000 mostres per al mateix algoritme, però sent k l'últim element de la llista en tots els casos. D'aquesta manera hem obtingut els següents gràfics:

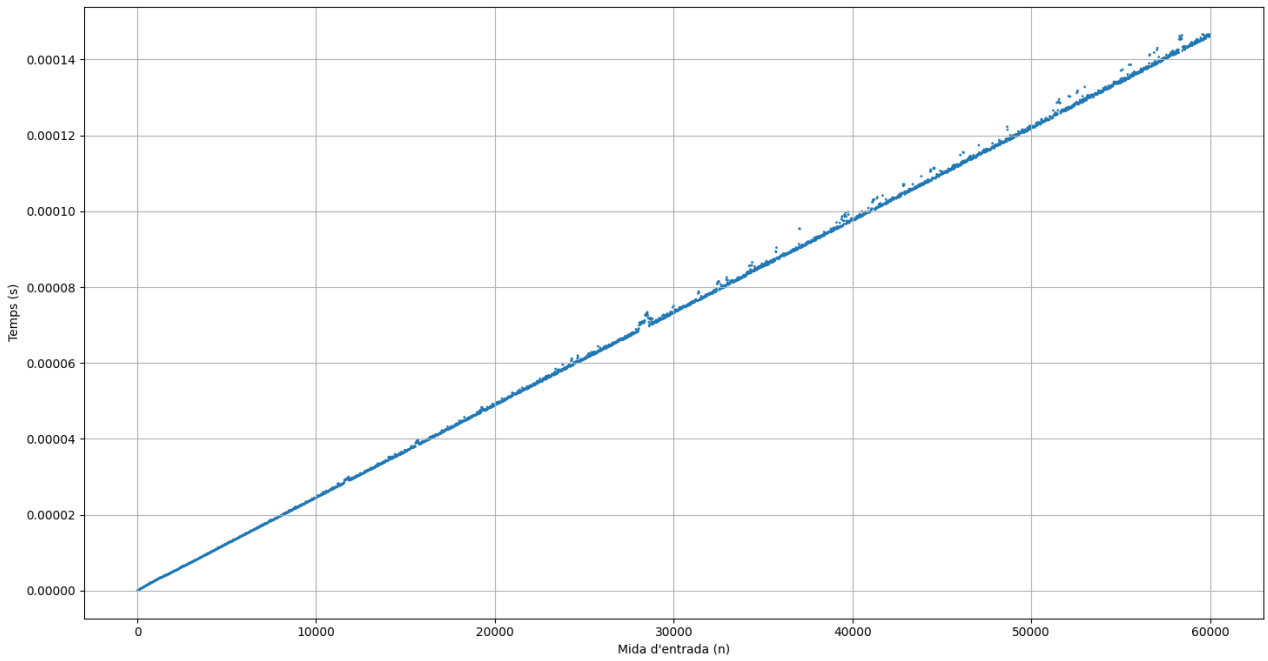


Figura 3.7: Gràfic de dispersió de la cerca lineal sent k l'últim element. Font: elaboració pròpia.

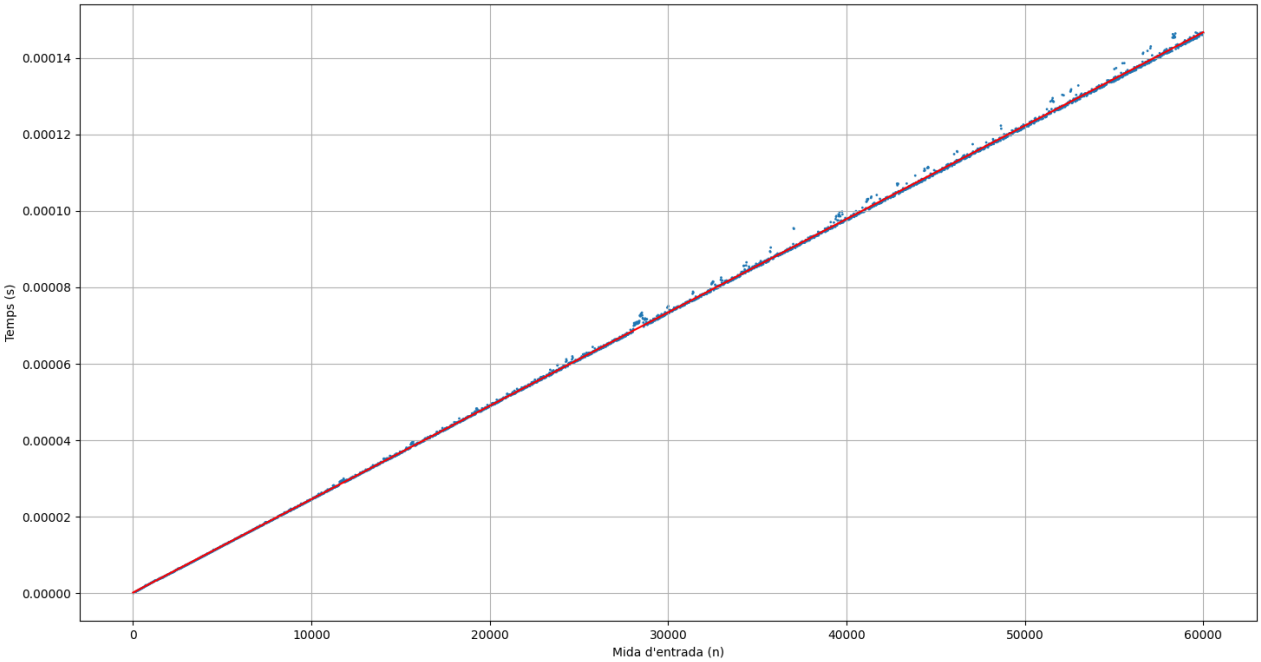


Figura 3.8: Gràfic de dispersió de la cerca lineal amb la recta de regressió lineal i sent k l'últim element. Font: elaboració pròpia.

En aquests gràfics podem veure de forma més clara que són lineals. A més, la recta de regressió lineal encaixa perfectament amb les mostres que hem obtingut ($r = 0.9999$). Tanmateix, totes les mostres que s'allunyen de la recta, queden sempre per sobre i mai per sota. Això demostra que quan hi ha hagut algun error, el programa ha tardat més en executar-se i això pot ser degut a factors de programari o maquinari.

Si comparem els pendents de les rectes, la recta de regressió lineal del gràfic de la figura 3.5 té un pendent d'$1.333 \cdot 10^{-9}$ i la recta del gràfic de la figura 3.8 té un pendent de $2.442 \cdot 10^{-9}$. És correcte que el pendent de la recta que hem obtingut en l'algoritme en què hem executat el pitjor cas possible sigui superior que en el cas en què k era aleatori.

Anàlisi empíric de la cerca dicotòmica

Ja que aquest algoritme és més eficient que el de la cerca lineal, he pogut recopilar més mostres. He executat l'algoritme 150.270 vegades. L'he executat 30 vegades per cada mida d'entrada entre 10 i 50.000 en intervals de 10 unitats i de 2 a 10 en intervals d'una unitat.

Finalment he obtingut els següents gràfics:

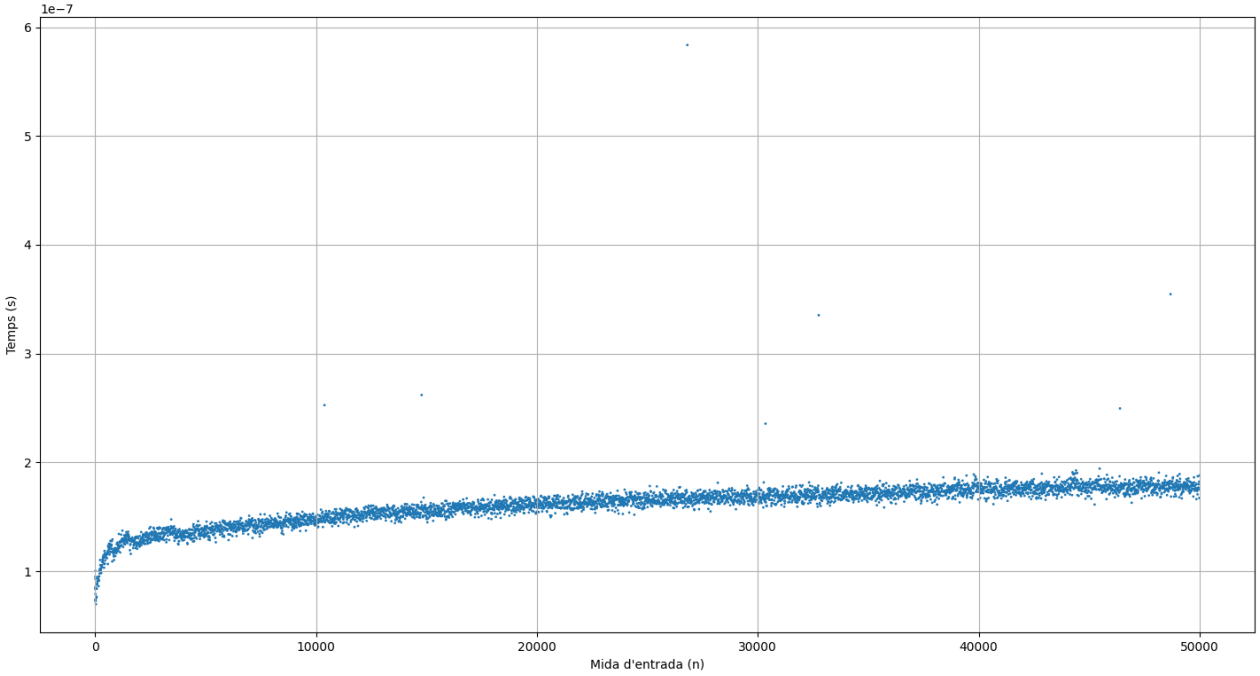


Figura 3.9: Gràfic de dispersió de la cerca dicotòmica. Font: elaboració pròpia.

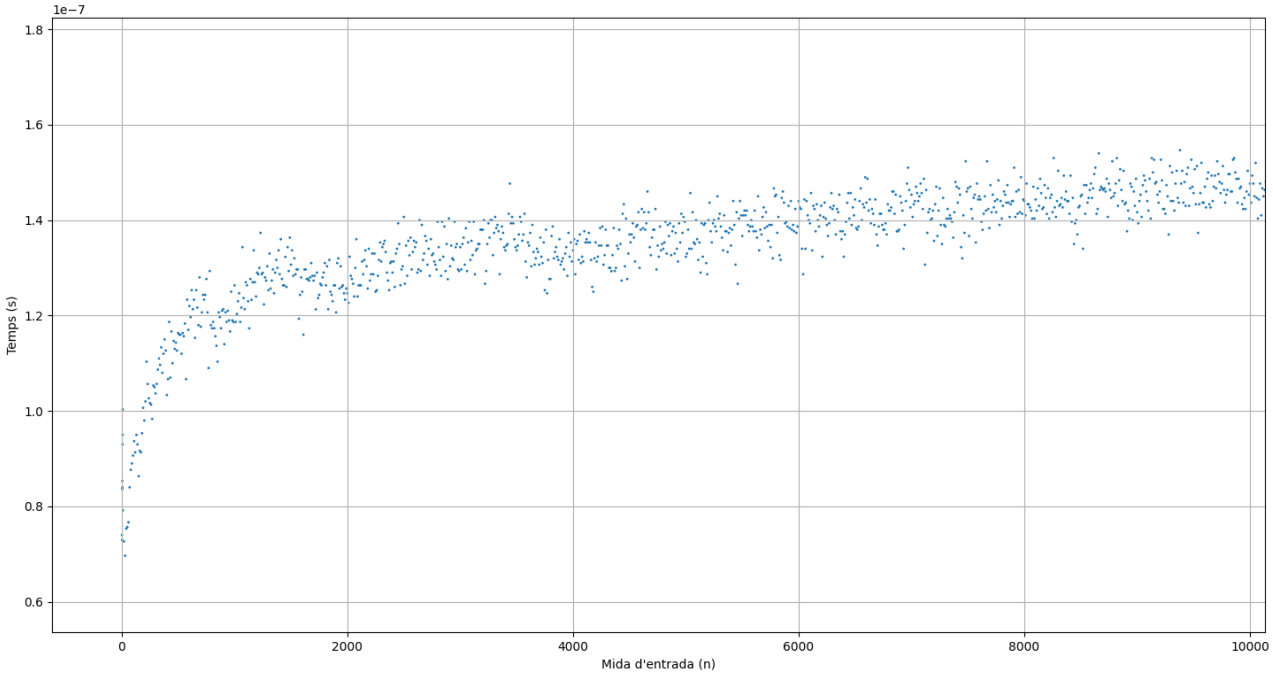


Figura 3.10: Gràfic de dispersió de la cerca dicotòmica entre l'interval 1 - 10.000. Font: elaboració pròpia.

Podem veure que aquests gràfics tenen la mateixa forma que la funció logarítmica.

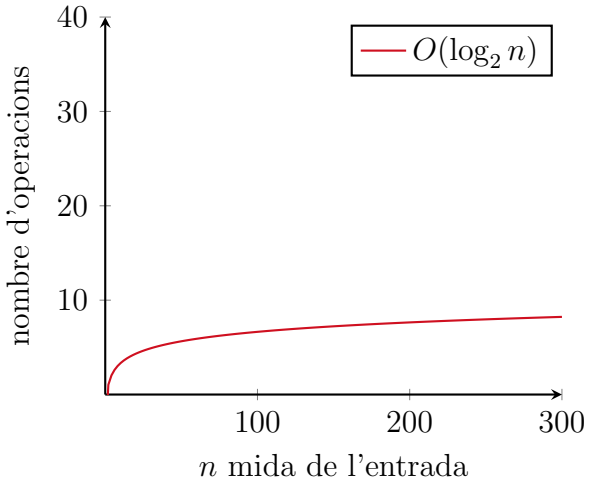


Figura 3.11: Complexitat logarítmica. Font: elaboració pròpia.

Si comparem els gràfics de les figures 3.9 i 3.11 podem veure que els dos tenen la mateixa forma. Per tant, hem demostrat de forma pràctica i teòrica que la cerca dicotòmica té complexitat logarítmica.

També podem veure que en aquests gràfics les mostres no s'allunyen tant com en la cerca lineal, així que podem arribar a la conclusió que la posició de k no afecta gaire els resultats, tampoc quan la mida de l'entrada és molt gran.

Anàlisi empíric de l'ordenació de bombolla

Aquest algoritme l'he executat 15.837 vegades. L'he executat 10 vegades per cada mida d'entrada entre 1 i 8.000 en intervals de deu unitats, i 10 vegades entre 8.000 i 10.000 en intervals de 50 unitats.

He obtingut els següents gràfics:

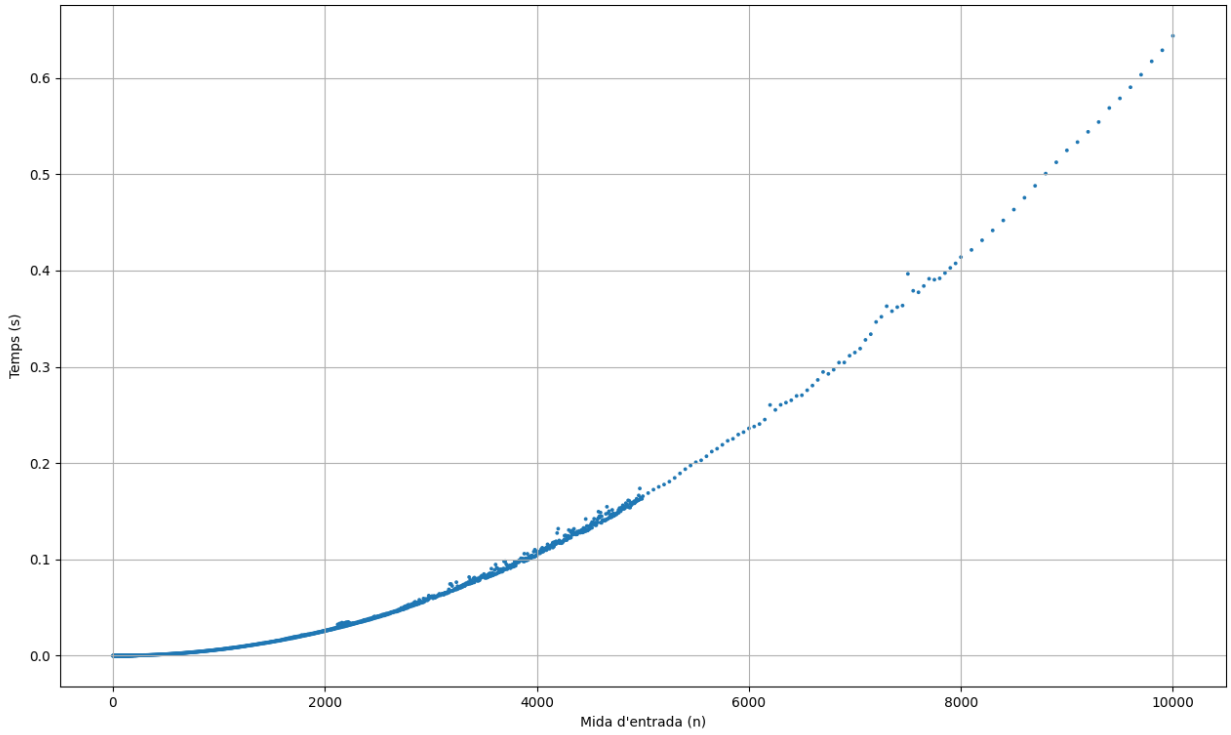


Figura 3.12: Gràfic de dispersió de l'ordenació bombolla. Font: elaboració pròpia.

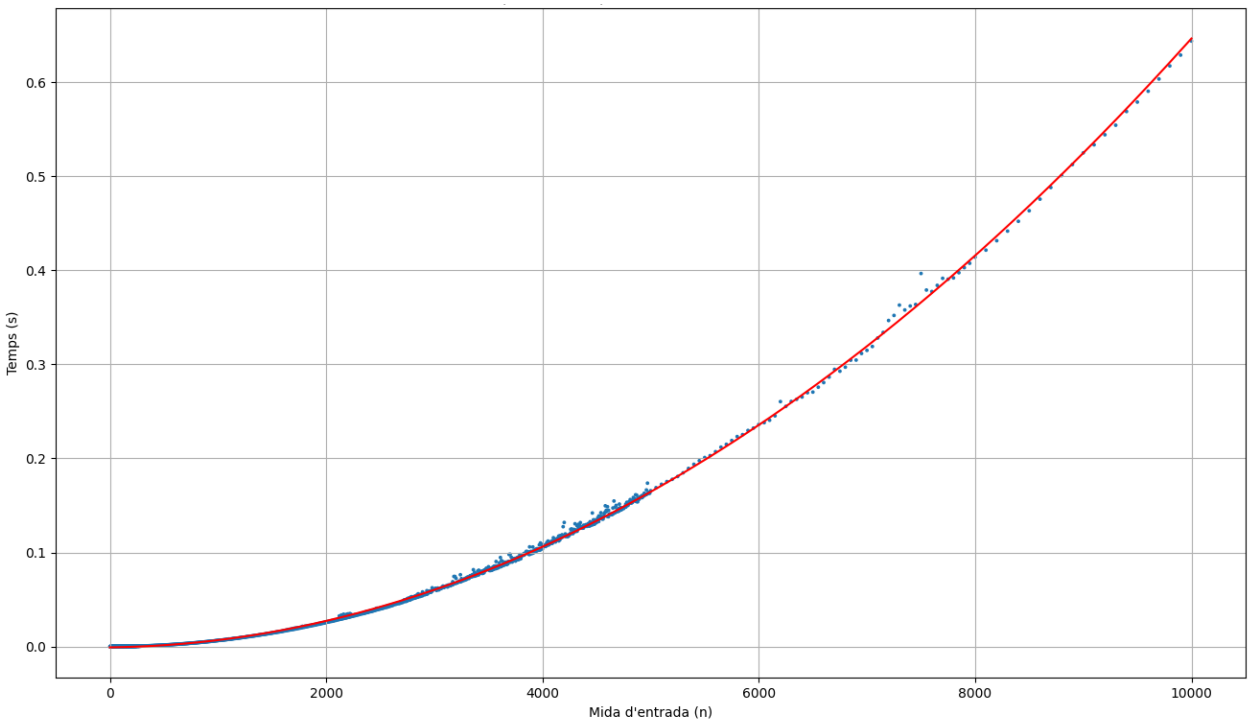


Figura 3.13: Gràfic de dispersió de l'ordenació bombolla amb la corba de regressió $f(x) = 6.33 \cdot 10^{-9}x^2 + 1.4 \cdot 10^{-6}x - 0.0008552$. Font: elaboració pròpia.

Podem veure clarament que la forma que té el gràfic de la figura 3.13 s'assembla molt a la funció quadràtica. A més, encaixa perfectament amb la corba de regressió polinòmica.

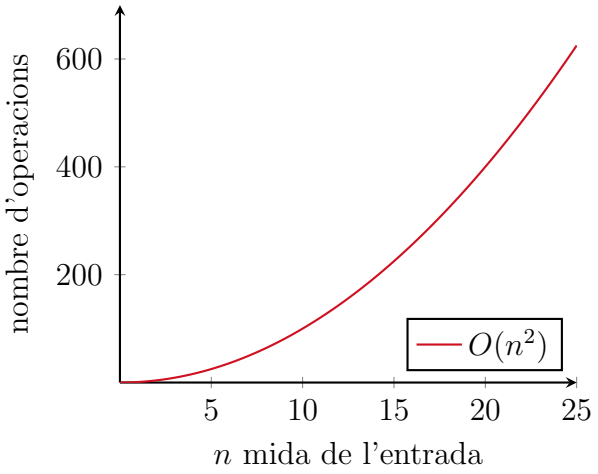


Figura 3.14: Gràfic de complexitat quadràtica. Font: elaboració pròpia.

Per tant, amb l'estudi matemàtic i empíric hem arribat a la mateixa conclusió: l'ordenació bombolla té complexitat quadràtica.

Anàlisi empíric de l'ordenació per barreja

Aquest algoritme l'he executat 54.260 vegades. Ho he fet 10 vegades per cada mida d'entrada d'entre 10 i 30.000 en intervals de 10 unitats, de 30.050 a 100.000 en intervals de 50 unitats i de 100.100 a 200.000 en intervals de 100 unitats. També he reforçat amb més mostres les parts del gràfic que s'allunyaven més de la resta de mostres.

Finalment he obtingut els següents gràfics:

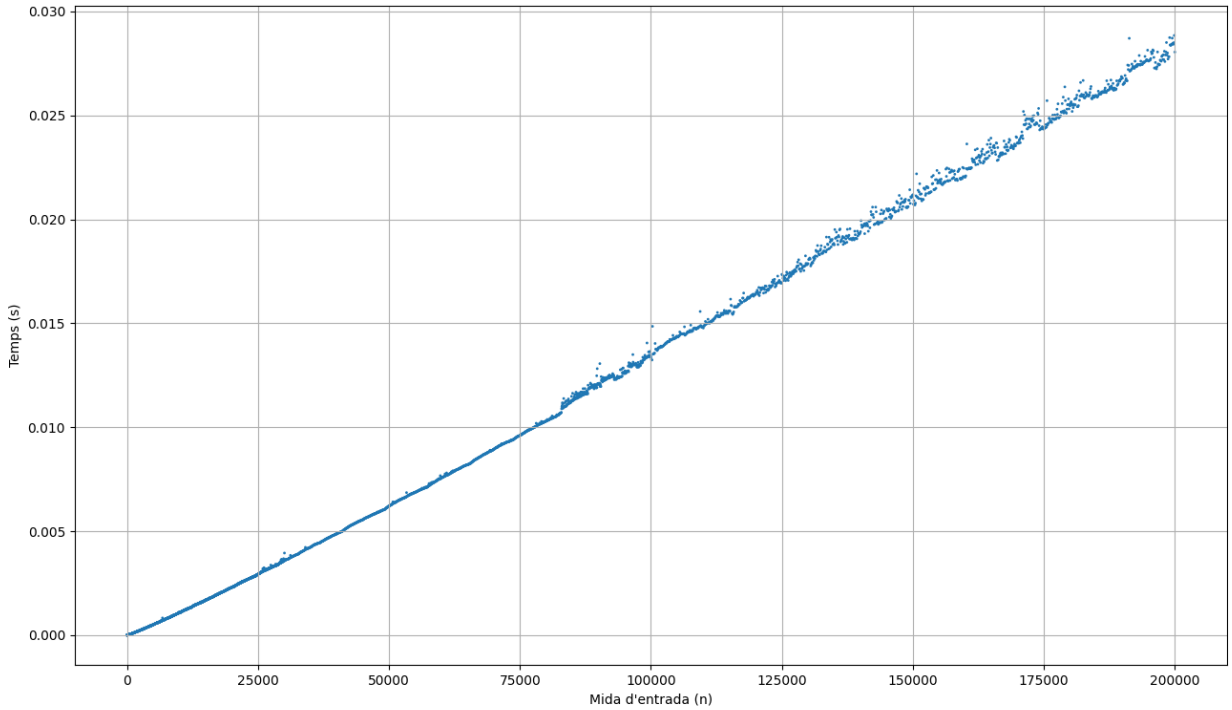


Figura 3.15: Gràfic de dispersió de l'ordenació per barreja. Font: elaboració pròpia.

Sembla una recta, així que podem dibuixar al gràfic una recta de regressió:

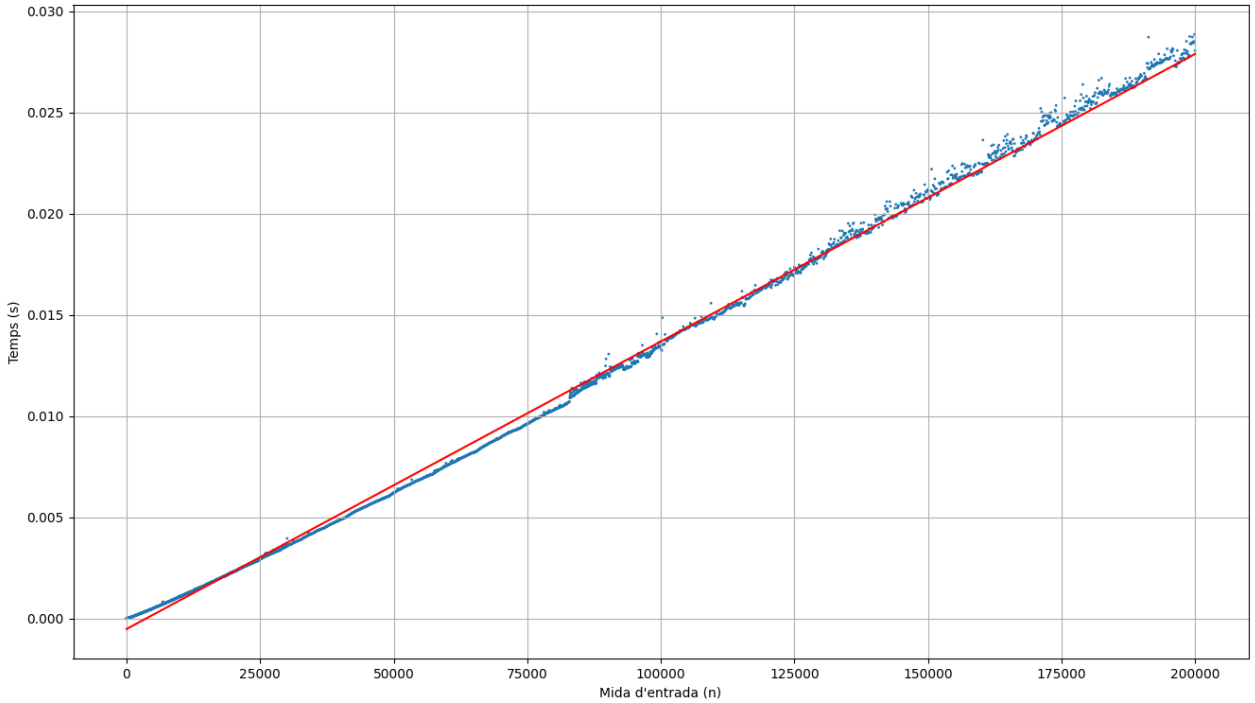


Figura 3.16: Gràfic de dispersió de l'ordenació per barreja amb la recta de regressió lineal. Font: elaboració pròpia.

Si comparem la recta vermella amb les dades que hem obtingut, podem veure que no acaba d'encaixar.

En l'anàlisi matemàtic hem arribat a la conclusió que aquest algoritme té complexitat $O(n \cdot \log_2 n)$. Així que podem comparar les figures 3.15 i 3.17.

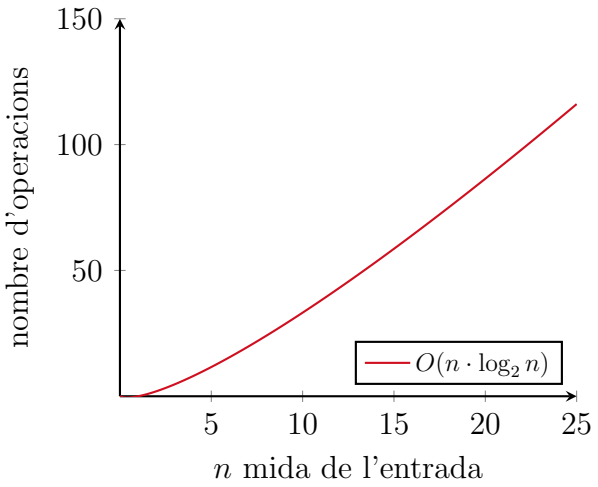


Figura 3.17: Gràfic de complexitat $O(n \cdot \log_2 n)$. Font: elaboració pròpia.

Podem veure com els gràfics s'assemblen, encara que no tant com en els altres tres algoritmes.

Sobre la mida d'entrada 80.000 veiem com la funció passa a ser menys acurada comparat amb els valors anteriors. Inicialment, he pensat que hi podria haver un error així que he tornat a executar les mesures i a fer un altre gràfic. En els dos casos he obtingut els mateixos resultats en la mateixa mida d'entrada. No sé explicar per què passa això en aquest punt.

CAPÍTOL 4

LaTeX

Finalment, he escrit tot aquest treball programant-ho en LaTeX i utilitzant l'editor Overleaf.

LaTeX és un sistema de composició de textos orientat a l'àmbit científic i matemàtic. No és un processador de textos, sinó que és un llenguatge de programació orientat a la generació de text.

En els processadors de textos s'escriu directament al document, i es pot editar l'estil amb el ratolí o es pot col·locar les imatges directament sobre el document. En canvi, LaTeX és un llenguatge de programació que es basa en etiquetes. Per editar qualsevol aspecte del document cal escriure una etiqueta i per visualitzar el document cal compilar el codi cada vegada.

Aquest llenguatge de programació és molt utilitzat en àmbits acadèmics de ciències, tecnologia, enginyeria, matemàtiques, física... i també es requereix per segons quines publicacions en revistes científiques, sobretot de matemàtiques i física.

Per exemple, per crear una figura i inserir una imatge, cal escriure el següent:

```latex
\begin{figure}[H]
    \centering
    \includegraphics[width=.15\textwidth]{imatge.png}
    \caption[Peu de foto.]{Peu de foto. Font: elaboració pròpia.}
    \label{fig:my_label}
\end{figure}
```

Figura 4.1: Exemple de codi en LaTeX. Font: elaboració pròpia.

La primera línia de la figura 4.1, crea la figura i la situa seguida del text. Després la centra, inclou la imatge basant-se en el directori on està guardada, es determina la mida i el peu de foto.

Igual que en la resta de llenguatges de programació s'han d'incloure biblioteques per a poder tenir més funcionalitats. Per exemple, per determinar l'espaiat, el format dels títols, capçalera i peus de pàgina, per incloure les figures, utilitzar colors...

En aquest enllaç trobareu tots els fitxers que formen aquest treball: https://github.com/JordinaGR/TdR-Latex o escanejant el codi QR de la figura 4.2:

Figura 4.2: Codi QR que porta a una pàgina web amb el codi del programa. Font: elaboració pròpia.

CAPÍTOL 5

CONCLUSIONS

Finalment, he assolit els objectius plantejats a l'inici d'aquest treball:

- Estudiar i definir què és l'algorísmia i per a què serveix.

En el capítol 1 hem definit els conceptes més importants per entendre bé què són i per a què serveixen els algoritmes. També hem posat alguns exemples.

- Estudiar com s'analitza l'eficiència dels algoritmes.

Al final del capítol 1 he explicat perquè és tan important analitzar l'eficiència dels algoritmes, i com es pot fer. He definit la notació que s'acostuma a utilitzar i he posat alguns exemples. També he comparat les complexitats més bàsiques per veure que efectivament hi ha molta diferència de creixement entre algunes complexitats.

- Estudiar de forma teòrica alguns algoritmes.

En el capítol 2 he explicat quatre algoritmes. Dos dels quals resolen un problema de cerca i els altres dos d'ordenació. En aquest capítol hem vist el funcionament d'aquests algoritmes i també els hem analitzat. En analitzar-los, hem pogut determinar quin dels dos és més eficient en termes d'eficiència temporal.

- Realitzar un programa per visualitzar els algoritmes estudiats.

He pogut implementar el programa sense gaires dificultats. Com que és un programa llarg i amb diversos programes en diferents fitxers que s'acaben ajuntant en un, hi ha la dificultat d'estructurar bé el programa per fer més fàcil la lectura i poder solucionar errors i depuracions amb més facilitat.

- Implementar aquells algoritmes i comparar els resultats pràctics amb els teòrics.

En el capítol 3 he implementat els algoritmes que hem estudiat al capítol 2 i he fet gràfics per comprovar si l'anàlisi pràctica i teòrica coincideixen.

En aquest capítol m'he trobat amb algunes dificultats. A l'hora de fer els gràfics en alguns casos hi havia dades que no coincidien amb els resultats esperats, així que vaig analitzar perquè passava això, i vaig arribar a la conclusió que l'ordinador estava executant algun procés en segon pla. Inicialment, executava 10 vegades seguides cada mida d'entrada, això provocava que quan hi havia un procés en segon pla les 10 mostres i les mides d'entrada del voltant es veien afectades. I encara que fes la mitjana, totes les mostres de la mateixa mida d'entrada estaven afectades. Així que vaig repetir els gràfics, però aquesta vegada executant una vegada totes les mides d'entrada i ho vaig repetir 10 vegades. D'aquesta manera, només canviant l'ordre en què s'executen les mostres, en cas que s'executi algun procés en segon pla, només afectarà una mostra de mides d'entrada diferents i a l'hora de fer la mitjana, afecta molt menys el resultat.

D'aquesta manera vaig aconseguir uns gràfics molt acurats.

- Programar el treball en LaTeX.

Com hem explicat al capítol 4, LaTeX és un llenguatge de programació i requereix temps aprendre'l.

Per fer aquest treball, primer vaig fer tot el format del document: capçalera, marges, tipus de lletra, espaiat, peus de pàgina, numeració de pàgina, l'estil dels enllaços, l'estil dels gràfics, taules, figures... També vaig crear un document diferent per a cada capítol per importar-los tots al document principal. I després de crear tota l'estructura del treball, vaig començar a

redactar el contingut.

Va ser difícil fer l'estructura del document, però considero que ha valgut la pena. Perquè a l'hora d'escriure el contingut m'ha resultat més senzill que amb qualsevol altre editor de text, i el resultat considero que és molt bo.

En conclusió, he assolit tots els objectius que m'he marcat en aquest treball. M'hauria agradat analitzar algoritmes més complexos i que tinguessin una aplicació pràctica més directa i interessant, com alguns algoritmes de grafs o conceptes com la programació dinàmica. Però, he prioritzat explicar les nocions bàsiques per fer més fàcil l'enteniment del treball i a causa del límit de pàgines no he pogut arribar a explicar alguns conceptes i algoritmes que m'hagués agradat.

Finalment, vull agrair la feina del meu tutor Jordi Irazuzta per totes les propostes i idees que has aportat en aquest treball i tot el temps que hi has dedicat.

També vull agrair la paciència i l'esforç del meu pare Jordi Gavaldà per llegir i rellegir el treball i corregir tots els errors ortogràfics i gramaticals.

CAPÍTOL 6

REFERÈNCIES

BIBLIOGRAFIA:

Roughgarden, Tim. *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures*, Soundlikeyourself, 2018.

REFERÈNCIES:

[1] "Sections and chapters." Consulta 1 d'abril 2022. https://es.overleaf.com/learn/latex/Sections_and_chapters

[2] Admin. "Your Guide to Fancy Chapters in LaTeX." LaTeX-Tutorial.Com (blog). Consulta 12 d'abril 2022. https://latex-tutorial.com/fancy-chapters-latex/

[3] TeX - LaTeX Stack Exchange. "Glenn Fncychap + Color of the Box." Consulta 13 d'abril 2022. https://tex.stackexchange.com/questions/405297/glenn-fncychap-color-of-\the-box

[4] "Using Colours in LaTeX." Consulta 13 d'abril 2022. https://www.overleaf.com/learn/latex/Using_colours_in_LaTeX

[5] "¿Qué Es Un Algoritmo y Qué Es Un Programa?" Consulta 18 de juny 2022. http://www.edu4java.com/es/conceptos/que-es-un-algoritmo-que-es-un-programa.html

[6] "Lista Completa de Símbolos En LaTeX — Manualdelatex.Com." Consulta 18 de juny 2022. https://manualdelatex.com/simbolos#chapter8

[7] –"ZoteroBib: Fast, Free Bibliography Generator - MLA, APA, Chicago, Harvard Citations." Consulta 18 de juny 2022. https://zbib.org/#

[8] ¿Qué Es Eso Del Problema P versus NP? Consulta 22 de juny 2022. https://www.youtube.com/watch?v=UR2oDYZ-Sao

[9] freeCodeCamp.org. "Beginners Guide to Big O Notation,". Consulta 23 de juny 2022. https://www.freecodecamp.org/news/my-first-foray-into-technology-c5b6e83fe8f1/

[10] Mejia, Adrian. "How to Find Time Complexity of an Algorithm?" Adrian Mejia Blog. Consulta 23 de juny 2022. https://adrianmejia.com/how-to-find-time-complexity-of-an-\algorithm-code-big-o-notation/

[11] Roura, Salvador. "Eficiència d'algorismes". UPC. Consulta 23 de juny 2022. https://www.cs.upc.edu/~roura/eficiencia.pdf

[12] Big O Notation - Full Course. Consulta 6 de juliol 2022. https://www.youtube.com/watch?v=Mo4vesaut8g

[13] arvindpdmn, dineshpathak. "Algorithmic Complexity." Consulta 7 de juliol 2022. https://devopedia.org/algorithmic-complexity

[14] "Real-World Example of Exponential Time Complexity." Forum post. Stack Overflow, Consulta 7 de juliol 2022. https://stackoverflow.com/q/7055652

[15] Lovett, Pamela. "Big-O Notation: A Simple Explanation with Examples." Consulta 7 de juliol 2022. https://betterprogramming.pub/big-o-notation-a-simple-explanation-\with-examples-a56347d1daca

[16] "Hide an Entry from Toc in Latex." Forum post. Stack Overflow. Consulta 12 de juliol 2022. https://stackoverflow.com/q/2785260

[17] Alraja, Humam Abo. "Logarithms & Exponents in Complexity Analysis." Consulta 15 de juliol 2022. https://towardsdatascience.com/logarithms-exponents-in-complexity-analysis-b8071979e847

[18] MIT. "Big O notation". Consulta 24 de juliol 2022. https://web.mit.edu/16.070/www/lecture/big_o.pdf

[19] GeeksforGeeks. "Bubble Sort Algorithm," Consulta 24 de juliol 2022. https://www.geeksforgeeks.org/bubble-sort/

[20] LaTeX-Tutorial.com. "How to Make Clickable Links in LaTeX." Consulta 27 de juliol 2022. https://latex-tutorial.com/tutorials/hyperlinks/

[21] Ljosa, Vebjorn. "Remove Ugly Borders around Clickable Cross-References and Hyperlinks." Forum post. TeX - LaTeX Stack Exchange, April 13, 2017. Consulta 27 de juliol 2022. https://tex.stackexchange.com/q/823

[22] MrD. "Answer to 'How to Allow Line Break in a Long Hyperlink in a PDF Compiled by Latex-Dvips-Ps2pdf?'" TeX - LaTeX Stack Exchange, April 10, 2013. Consulta 27 de juliol 2022. https://tex.stackexchange.com/a/108003

[23] user133284. "Pagebreak for Minted in Figure." Forum post. TeX - LaTeX Stack Exchange, May 9, 2017. Consulta 27 de juliol 2022. https://tex.stackexchange.com/q/368864

[24] Simplilearn.com. "What Is DFS (Depth-First Search): Types, Complexity & More — Simplilearn." Consulta 26 d'agost 2022. https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm

[25] "Example; Undirected and Unweighted Edges - C# Data Structures and Algorithms [Book]." Consulta 28 d'agost 2022. https://www.oreilly.com/library/view/c-data-structures/9781788833738/0309b525-cfe4-429d-bdb9-53b25f94bbcd.xhtml

[26] Quora. "Who Uses TeX, besides LaTeX?" Consulta 29 d'agost 2022. https://www.quora.com/Who-uses-TeX-besides-LaTeX

[27] Ivorra, Carlos. "Introducción al LATEX". Consulta 29 d'agost 2022. https://www.uv.es/~ivorra/Latex/LaTeX.pdf

CAPÍTOL 7

ANNEX

7.1 Una solució al sudoku amb gràfics

Podeu trobar el programa en aquest enllaç: https://github.com/JordinaGR/sudoku o escanejant el codi de la figura 6.1.

Figura 7.1: Programa que resol els sudokus. Font: elaboració pròpia.

He fet un vídeo que mostra el funcionament del programa, es pot trobar en el següent enllaç: https://drive.google.com/file/d/125IrdwNMtVojWc54ez3fvlFfB3J_Ozhe/view?usp=sharing o escanejant el codi de la figura 6.2.

Figura 7.2: Vídeo del programa que resol els sudokus. Font: elaboració pròpia.

7.2 Implemetació del BFS, DFS i Dijkstra

```cpp
// Implementació del BFS en C++
// Entrada: número de vèrtex i arestes, i les arestes (no dirigides)
// Sortida: llista de V elements amb la distància mínima per anar del primer vèrtex
// a l'i-èssim.

#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, m; cin >> n >> m;

    vector<vector<int>> graph(n);
    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        u--; v--;

        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    vector<int> ans(n);

    for (int i = 1; i < n; i++){
        bool flag = false;

        vector<int> visited(n, -1);
        queue<int> q;

        visited[0] = 0;
        q.push(0);

        while (!q.empty() and !flag) {
            int p = q.front();
            q.pop();

            for (int nei : graph[p]){
                if (visited[nei] == -1){
                    visited[nei] = visited[p] + 1;
                }
                if (nei == i){
                    ans[i] = visited[nei];
                    flag = true;
```

```
                    break;
                }
                q.push(nei);
            }
        }
    }
    for (auto x : ans){
        cout << x << ' ';
    } cout << endl;
}
```

Figura 7.3: Implementació del BFS en C++ en un graf no dirigit. Font: elaboració pròpia.

```
// Implementació del DFS en C++
// Entrada: número de vèrtex i arestes, i les arestes (no dirigides)
// Sortida: si és possible arribar des del primer a l'últim vèrtex

#include <bits/stdc++.h>
using namespace std;

string dfs(int start, int end, vector<bool>& visited, vector<vector<int>>& graph){
    visited[start] = true;

    if (start == end){
        return "SI";
    }

    for (int nei : graph[start]){
        if (!visited[nei]){
            return dfs(nei, end, visited, graph);
        }
    }
    return "NO";
}
int main(){
    int n, m; cin >> n >> m;

    vector<vector<int>> graph(n);
    vector<bool> visited(n, false);

    for (int i = 0; i < m; i++){
        int u, v;
        cin >> u >> v;
        u--; v--;

```

```
            graph[u].push_back(v);
            graph[v].push_back(u);
        }

        cout << dfs(0, n-1, visited, graph) << endl;
    }
```

Figura 7.4: Implementació del DFS en C++ en un graf no dirigit. Font: elaboració pròpia.

```
    // Implementació del Dijkstra en C++
    // Entrada: número de vèrtex i arestes, i les arestes amb pesos (graf no dirigit)
    // Sortida: llista de V elements amb la distància mínima per anar del primer vèrtex
    // a l'i-èssim.

    #include <bits/stdc++.h>
    using namespace std;
    const int INF = 1 << 30;

    int main(){
        ios::sync_with_stdio(false);
        cin.tie(NULL);

        int n, m;
        cin >> n >> m;
        vector<vector<pair<int, int>>> adj(n);
        for (int i = 0; i < m; ++i){
            int u, v, w;
            cin >> u >> v >> w;
            u--; v--;

            adj[u].emplace_back(w, v);
            adj[v].emplace_back(w, u);
        }

        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        vector<int> dist(n, INF);
        pq.emplace(0, 0);
        dist[0] = 0;

        for (int i = 1; i < n; i++){
            pq.emplace(INF, i);
        }

        while (pq.size()){
            auto [c, v] = pq.top();
```

```
            pq.pop();
            if (c != dist[v]) continue;
            for (auto [w, u] : adj[v]){
                if (c + w < dist[u]){
                    dist[u] = c + w;
                    pq.emplace(dist[u], u);
                }
            }
        }

        for (int d : dist){
            cout << d << ' ';
        }
        cout << endl;
    }
```

Figura 7.5: Implementació del Dijkstra en C++ en un graf no dirigit. Font: elaboració pròpia.

7.3 Implementació de la cerca lineal

```
// Implementació en C++

#include<bits/stdc++.h>
using namespace std;

int main(){
        int n, k; cin >> n >> k;
        int ans = -1;

        for (int i = 0; i < n; i++){
                int x; cin >> x;
                if (x == k) ans = i;
        }

        cout << ans << endl;

}
```

Figura 7.6: Implementació de cerca lineal en C++. Font: elaboració pròpia.

```python
# Implementació en Python

n, k = map(int, input().split())
ans = -1
array = list(map(int, input().strip().split()))

for i in range(n):
    if array[i] == k:
        ans = i

print(ans)
```

Figura 7.7: Implementació de cerca lineal en Python. Font: elaboració pròpia.

7.4 Implementació de la cerca dicotòmica

```cpp
// Implementació en C++

#include <bits/stdc++.h>
using namespace std;

int binary(int l, int r, int k, vector<int> v){
    while (l <= r){
        int mid = (l+r)/2;

        if (v[mid] == k) return mid;
        else if (v[mid] > k) {
            r = mid-1;
        } else if (v[mid] < k) {
            l = mid+1;
        }
    }
    return -1;
}
int main(){

    int n, k; cin >> n >> k;
    vector<int> v(n);
    for (int i = 0; i < n; i++) cin >> v[i];

    int x = binary(0, n-1, k, v);

    cout << x << endl;
}
```

Figura 7.8: Implementació de la cerca dicotòmica en C++. Font: elaboració pròpia.

```python
# Implementació en Python

def binary(l, r, arr):
    while l <= r:
        mid = (l+r) // 2

        if arr[mid] == k:
            return mid
        elif arr[mid] > k:
            r = mid-1
        elif arr[mid] < k:
            l = mid+1

    return -1

n, k = map(int, input().split())
array = list(map(int, input().strip().split()))

x = binary(0, n-1, array)

print(x)
```

Figura 7.9: Implementació de la cerca dicotòmica en Python. Font: elaboració pròpia.

7.5 Implementació de l'ordenació de bombolla

```cpp
// Implementació en C++

#include <bits/stdc++.h>
using namespace std;

int main(){
    int n; cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++){
        cin >> v[i];
    }
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n-1; j++){
            if (v[j] > v[j+1]){
                int tmp = v[j];
                v[j] = v[j+1];
                v[j+1] = tmp;
            }
        }
    }
    for (auto x : v){
        cout << x << ' ';
    } cout << endl;
}
```

Figura 7.10: Implementació de l'ordenació de bombolla en C++. Font: elaboració pròpia.

```python
# Implementació en Python

n = int(input())
arr = list(map(int, input().strip().split()))

for i in range(n):
    for j in range(0, n-1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]

print(*arr)
```

Figura 7.11: Implementació de l'ordenació de bombolla en Python. Font: elaboració pròpia.

7.6 Implementació de l'ordenació per barreja

```cpp
// Implementació en C++

#include <bits/stdc++.h>
using namespace std;

void merge(vector<int>& v, int l, int mid, int r){
    vector<int> vec;
    int i = l; int j = mid+1;

    while (i <= mid and j <= r){
        if (v[i] < v[j]){
            vec.push_back(v[i]);
            i++;
        } else {
            vec.push_back(v[j]);
            j++;
        }
    }
    while (i <= mid){
        vec.push_back(v[i]);
        i++;
    }
    while (j <= r){
        vec.push_back(v[j]);
        j++;
    }
    int y = 0;
    for (int q = l; q <= r; q++){
        v[q] = vec[y];
        y++;
    }
}
void mergeSort(vector<int>& v, int l, int r){
    if (l < r){
        int mid = (l+r) / 2;
        mergeSort(v, l, mid);
        mergeSort(v, mid+1, r);

        merge(v, l, mid, r);
    }
}
int main(){
    int n; cin >> n;
```

```
44          vector<int> v(n);
45
46          for (int i = 0; i < n; i++) cin >> v[i];
47
48          mergeSort(v, 0, n-1);
49
50          for (auto x : v){
51              cout << x << ' ';
52          } cout << endl;
53      }
```

Figura 7.12: Implementació de l'ordenació per barreja en C++. Font: elaboració pròpia.

```
1   # Implementació en Python
2
3   def merge(arr, l, mid, r):
4       b = []
5       i, j = l, mid+1
6
7       while i <= mid and j <= r:
8           if arr[i] < arr[j]:
9               b.append(arr[i])
10              i += 1
11          else:
12              b.append(arr[j])
13              j += 1
14
15      while i <= mid:
16          b.append(arr[i])
17          i += 1
18
19      while j <= r:
20          b.append(arr[j])
21          j += 1
22
23      y = 0
24      for q in range(l, r+1):
25          arr[q] = b[y]
26          y += 1
27
28  def mergeSort(arr, l, r):
29      if (l < r):
30          mid = (l+r) // 2
31          mergeSort(arr, l, mid)
32          mergeSort(arr, mid+1, r)
```

```
33
34              merge(arr, l, mid, r)
35
36
37      n = int(input())
38      arr = list(map(int, input().strip().split()))
39
40      mergeSort(arr, 0, n-1)
41
42      print(*arr)
```

Figura 7.13: Implementació de l'ordenació per barreja en Python. Font: elaboració pròpia.