Ordenació recursiva

Pràctica 1 - RunnersLS

Grup 14

Pol Piñol Castuera (pol.pinol)
Roger Casas Aisa (roger.casas)

Índex

1 Introducció

Aquest treball, RunnersLS, és un projecte fictici creat per la universitat La Salle amb l'objectiu de gestionar les dades dels atletes que participen en competicions en el campus. A la pràctica tracta sobre la implementació de tres algorismes d'ordenació recursius. Aquests algorismes són: Quicksort, Merge Sort i Bucket Sort. Cadascun d'aquests algorismes haurà d'ordenar segons un criteri determinat que en les següents seccions explicarem.

El llenguatge escollit per la implementació del algorismes recursius ha estat Java (versió 15.0.1) i el IDE utilitzat ha estat IntelliJ IDEA Ultimate 2020.2.3 x64. La raó per la qual hem escollit aquest IDE ha estat principalment per la facilitat que representa programar amb IntelliJ, ja que és un programa molt user-friendly el qual ajuda a l'usuari a corregir errors de codi i/o recomanacions de com optimitzar dit codi.

L'elecció del llenguatge de programació Java ha estat perquè la modularitat que ofereix el llenguatge ens ha semblat molt més atractiva que la linealitat que presenten altres llenguatges com ara C o C++. El fet de treballar per mòduls, fent una classe per a cada bloc de la pràctica i així poder debugar de forma molt més senzilla ens ha semblat la millor opció per a realitzar la pràctica.

2 Explicació dels algorismes

En aquesta secció explicarem com s'han codificat els tres algorismes d'ordenació recursius. Abans d'entrar en detall, explicarem els criteris que ens don l'enunciat de la pràctica per utilitzar en els algorismes.

1. Ordenar la llista de clubs segons la mitjana d'edat dels seus atletes, de menor a major
2. Ordenar alfabèticament la llista de atletes segons la seva nacionalitat, de la A fins la Z. En cas que existeixi més d'un atleta amb la mateixa nacionalitat, es desempatarà segons el seu nom.
3. Ordenar alfabèticament la llista de atletes segons el seu estat físic.

2.1 Bucket Sort

Respecte al Bucket Sort, hem decidit usar-lo per a l'ordenació dels clubs segons la mitjana d'edat dels atletes ja que al saber que la mitjana d'edat oscil·laria entre els 10 i 100 anys, sabíem que llavors només faria falta treballar amb 10 buckets inicialment i això reduïa el cost de l'algorisme respecte als altres casos, en els quals podien haver-hi molts més. La lògica darrere aquest algorisme és que creem inicialment 10 buckets, en cada un dels quals guardarem les diferents mitjanes segons la seva primera xifra i, si en un bucket s'hi troben més d'un element, creem 10 més de forma recursiva (si abans miraven les desenes, ara les unitats) i anem repetint aquest procés fins que tenim tots els elements col·locats de forma individual en el seu respectiu bucket. En general, hem fet servir informació de [1], [2] i [3].

2.2 Merge Sort

Respecte al Merge Sort, hem decidit usar-lo per a l'ordenació alfabèticament de la llista de atletes segons la seva nacionalitat i, si tenen la mateixa nacionalitat, segons el seu nom de forma també alfabètica. Pel que fa al algorisme programat hem seguit principalment els apunts de l'assignatura, i aplicat el pseudocodi al llenguatge Java. En aquest algorisme dividim els arrays per la meitat fins que ens quedem amb cel·les individuals, que llavors és quan cridem *merge*() per a que comenci a ordenar les cel·les segons el criteri utilitzat. En general, hem fet servir informació de [4].

2.3 Quick Sort

Respecte al Quick Sort, hem decidit usar-lo per a l'ordenació de la llista de atletes segons l'estat físic de cada un d'ells.

De cara a calcular l'estat físic (ordenació per combinació de prioritats) volíem donar especial importància a la distància recorreguda per l'atleta respecte al temps en el qual ha recorregut dita distància. Inicialment hem considerat que la distància es inversament proporcional al temps. Per tant teníem tres possibles opcions de fórmula.

La primera opció era la següent:

$$estatFisic = \frac{d^3}{t} \quad \text{on d és la distància i t el temps.}$$

Amb aquesta fórmula, vam executar el algorisme Quick Sort (explicat més endavant) i vam obtenir els següents resultats.

	Nom	Estat Físic	Distància	Temps
Posició 1	Emilio Barton	342950.0	19.0	0.02
Posició 2	Herma Okuneva	151407.7	27.0	0.13
Posició 3	Judith Murazik	100000.0	10.0	0.01

La segona opció era la següent:

$$estatFisic = \frac{d^2}{t} \quad \text{on d és la distància i t el temps.}$$

Amb aquesta fórmula, vam executar el algorisme Quick Sort i vam obtenir els següents resultats.

	Nom -	Estat Físic	- Distància	- Temps
Posició 1	Emilio Barton	- 342950.0	- 19.0	- 0.02
Posició 2	Judith Murazik	- 100000.0	- 10.0	- 0.01
Posició 3	Herma Okuneva	- 151407.7	- 27.0	- 0.13

La tercera opció era la següent:

$$estatFisic = \frac{d}{t} \quad \text{on d és la distància i t el temps.}$$

Amb aquesta fórmula, vam executar el algorisme Quick Sort i vam obtenir els següents resultats.

	Nom -	Estat Físic	- Distància	- Temps
Posició 1	Judith Murazik	- 100000.0	- 10.0	- 0.01
Posició 2	Emilio Barton	- 342950.0	- 19.0	- 0.02
Posició 3	Herma Okuneva	- 151407.7	- 27.0	- 0.13

Analitzant el top 3 de les posicions amb les diferents opcions de fórmules ens va semblar mes encertada les posicions obtingudes a partir de la primera opció ja que, per exemple, valorem més la distància recorreguda que el temps, per això el Herma Okuneva està en la posició 2, però encara que el Emilio Barton tingui menys distància recorreguda, si calculem la velocitat mitjana observem que quasi quadriplica al Herma Okuneva. Per tant, hem considerat que elevar al cub era més adequat per a les nostres prioritats.

Per això hem decidit implementar finalment la següent funció:

$$estatFisic = \frac{d^3}{t} \quad \text{on d és la distància i t el temps.}$$

En altres paraules, podríem expressar la fórmula de la següent forma:

$$estatFisic = v_m d^2 \quad \text{on d és la distància i } v_m \text{ la velocitat mitjana.}$$

Pel que fa al algorisme programat hem seguit principalment els apunts de l'assignatura, i aplicat el pseudocodi al llenguatge Java. En aquest algorisme simplement col·loquem un pivot a la meitat del array, definint així la meitat esquerra on hi hauran els valors més petits i la meitat dreta on hi hauran els valors més grans. Anem movent els punters de forma recursiva i anem canviant el pivot segons el valor comparant amb els punters els estats físics. En general, hem fet servir informació de [5].

3 Comparativa teòrica

En aquesta secció explicarem l'ordre de complexitat darrera de cada algorisme d'ordenació recursiu i els resultats esperats pels diferents temps d'execució dels algorismes implementats a la nostra pràctica.

En el primer algorisme que ens centrarem és el Quick Sort. Aquest té un cost asimptòtic de $O(n^2)$ en el pitjor dels casos, un cost $O(nlogn)$ en el millor dels casos i un cost mitjà de $O(nlogn)$. Es tracta d'un algorisme que treballa de forma bastant ràpida excepte amb petits volums de dades, en el qual els altres dos algorismes, teòricament, són una mica més ràpids.

L'algorisme Merge Sort sempre té un cost asimptòtic de $O(nlogn)$, el mateix que el Quick Sort. Tot i això, l'anterior algorisme (Quick Sort) és considerat més ràpid ja que, mentre el Merge Sort sempre serà de cost $O(nlogn)$, el Quick Sort pot oscil·lar de cost asimptòtic segons si hi ha "sort" o no. Per a "sort"entenem el fet que depenent d'on es col·loqui el pivot els temps d'execució pot variar de forma considerable.

Per acabar, el Bucket Sort té un cost asimptòtic (en els pitjors de casos) de $O(n^2)$. En general té un cost asimptòtic $O(n + \frac{n^2}{k} + k)$ on k és el número de buckets que es fa servir. Com el nostre cas tenim 10 buckets, seria més exacte dir que tenim un cost $O(n + \frac{n^2}{10} + 10) \approx O(n^2)$.

En general, hem vist com segons els costos de cada algorisme, teníem la previsió de que el Quick Sort seria l'algorisme més ràpid, seguit del Merge Sort i el Bucket Sort com a últim. Teòricament el pitjor algorisme hauria de ser el Bucket Sort conforme anem analitzant més dades.

4 Anàlisi de resultats

En aquesta secció compararem els diferents algorismes d'ordenació recursius a través dels reulstats obtinguts i calculats. Mostrarem gràfiques obtingudes a partir de *gnuplot* [6] i analitzarem els resultats obtinguts amb els esperats (teòrics).

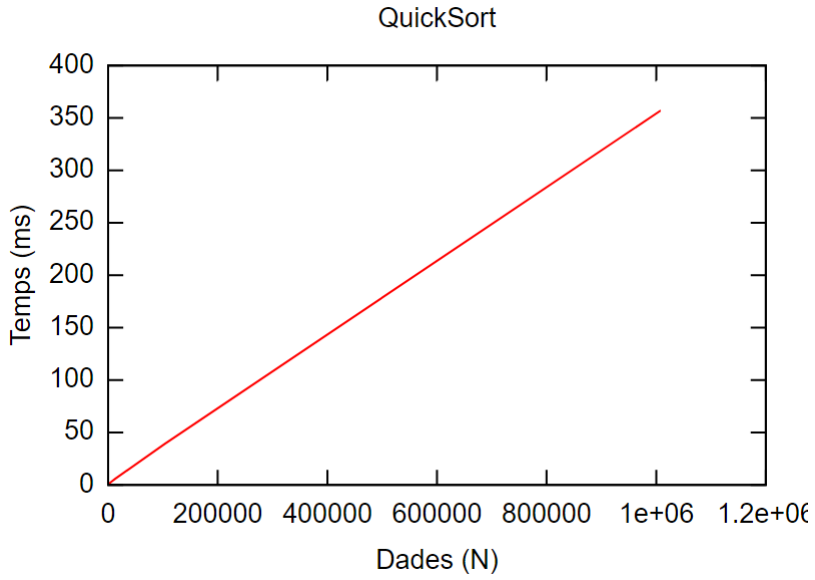

Figura 1: Temps d'execució del algorisme Quick Sort per diferents dataseets

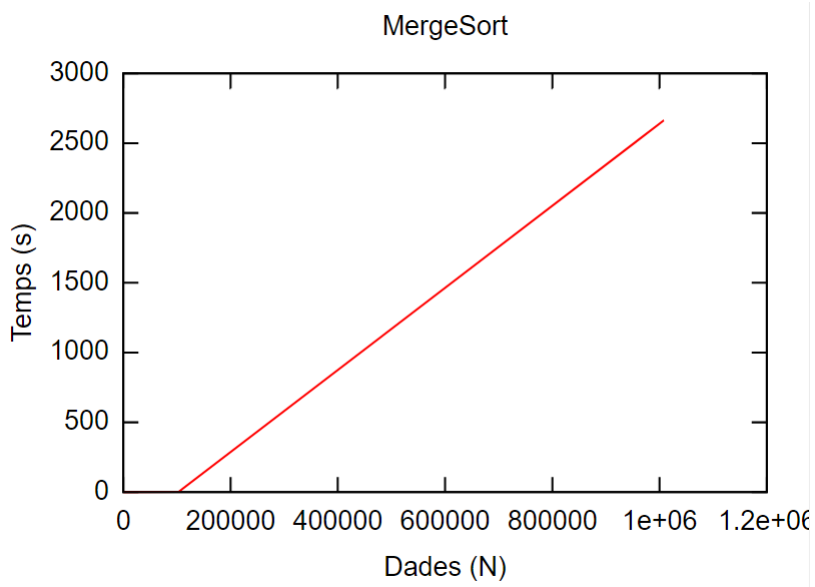

Figura 2: Temps d'execució del algorisme Merge Sort per diferents dataseets

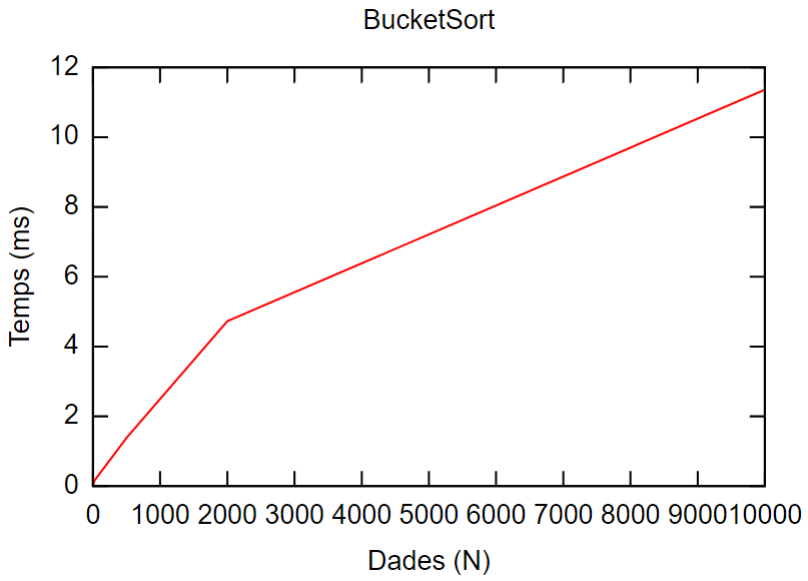

Figura 3: Temps d'execució del algorisme Bucket Sort per diferents dataseets

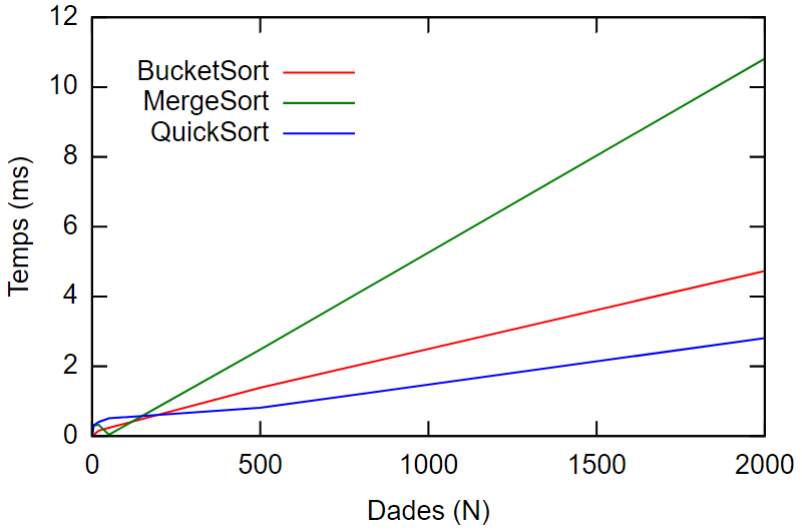

Figura 4: Temps d'execució dels tres algorismes per diferents dataseets

El que ens ha sorprès principalment en la obtenció de resultats és que l'algorisme Bucket Sort ha rendit molt millor que el Merge Sort, resultat del qual esperàvem exactament el contrari. Això creiem que es deu a que com hem optimitzat molt l'algorisme Bucket Sort i aconseguit reduir el número de buckets a només 10, ha passat de cost $O(n^2)$ a assemblar-se a $O(n * logn)$, és a dir a quedar gairebé paral·lel al Quick Sort però evidentment bastant més lent. A més, hem escollit el criteri de comparar un conjunt de dades que podem esbrinar des de un principi com seran (les mitjanes d'edat haurien de variar entre els 10 anys i els 100 anys).

El Merge Sort, llavors, ha passat a ser el més lent dels tres, i com a petita curiositat hem vist com el Quick Sort és l'algorisme més lent dels tres quan es treballa amb volums molt petits de dades, però com més augmenta el volum, també ho fa la diferència entre temps d'execució entre el Quick Sort i la resta.

5 Mètode de proves

El mètode de proves emprat per a aquesta pràctica ha estat possible gràcies a haver escollit Java i IntelliJ, ja que hem creat una sèrie de procediments els quals fem un assert entre els resultats obtinguts pels algorismes implementats (les llistes ordenades) amb el mètode d'ordenació ja implementat dins de IntelliJ anomenat sort().

Cal remarcar que, com desconeixem el funcionament intern de la funció sort(), comparem els dos arrays ordenats utilitzant com a criteri els valors obtinguts, i no els noms dels clubs o atletes. Per exemple, si en la mitjana d'edat obtenim a les dues últimes posicions: Club A – 20.5 anys i tot seguit Club B – 20.5 anys, mirarem que en les dues últimes posicions de l'array ordenat per sort() hi hagi 20.5 i a continuació 20.5, ja que podria ser que posessin el Club B per davant el Club A per raons de funcionament intern. Mentre els valors que tinguin els dos arrays siguin iguals, no mirem de forma tan exhaustiva si els noms coincideixen ja que en aquest cas particular es podrien donar errors.

A més, afegir que caldria acabar de demostrar-ho tot analíticament, però amb aquesta comprovació hem obtingut resultats convincents de que s'han ordenat de forma correcta.

6 Problemes observats

Un dels principals problemes que ens hem trobat i notificat al professor ha estat que no hem fet ús de l'eina de repositori, al no estar acostumats a utilitzar-la. Per raons de comoditat compartíem el codi mitjançant eines convencionals que ara mirant enrere són evidentment contraproduents.

Un altre problema trobat ha estat de cara al Bucket Sort; saber controlar i diferenciar mitjanes de 10-19 i de 100-109, ja que al llegir sempre la primera posició de la cadena de caràcters havíem de poder inserir cada mitjana al seu bucket corresponent. També hem hagut de crear zeros "artificials" per tal de poder comparar números amb un nombre diferent de decimals (per exemple poder comparar 30.1 amb 30.108 sense que directament seleccionem 30.1 com al major valor).

Inicialment amb el Bucket Sort teníem problemes a nivell de decidir com anar comparant els diferents dígits de la mitjana, ignorant els punts i comparant decimals un per un. És llavors quan hem decidit comparar-los utilitzant la mitjana com una cadena de caràcters i així simplement anar fent un $cadena.charAt(i)$ de cada casella de la cadena.

A més a més ens hem trobat que quan calculàvem el temps d'execució amb un portàtil, els valors variaven dràsticament que quan executàvem el programa amb un ordinador de torre o fins i tot un altre portàtil. Llavors amb tal de poder mostrar gràfiques simples i que mostrin els resultats de forma senzilla, hem realitzat totes aquestes mesures amb un sol ordinador de torre.

En general el algorisme Bucket Sort és el que més problemes ens ha donat, ja que mentre que per als altres dos disposàvem de pseudocodis i apunts per a consultar, no teníem molta documentació respecte aquest i ha resultat un repte interessant de cara a anar dissenyant des de zero el seu funcionament. En general ha estat on hem dedicat més temps de tota la pràctica.

Per acabar, el que seria el problema més important que ens hem trobat seria la lectura dels fitxers JSON, ja que amb volums de dades molt elevats el procediment utilitzat habitualment en altres assignatures per llegir un fitxer (llegir línies del fitxer i fer un $text+ = linia$ fins que les línies llegides no siguessin *null*) no funcionava de forma òptima ja que ocupava molta memòria. Per a donar una idea aproximada, només llegir el fitxer JSON ja ens ocupava més de dues hores i mitja amb el $dataseetXL.json$. Al final hem decidit utilitzar un $StringBuilder$, el qual es diferencia en que funciona com un $ArrayList <>$ i cada línia que llegim es fa un append() dins d'aquesta llista. En comparació, ara la lectura dels fitxers $dataseetXL.json$ i $dataseetXXL.json$ s'executa en pocs segons. Indicar que hem fet servir informació de [7] i [8] per obtenir informació sobre el $StringBuilder$.

7 Conclusions

Com a conclusió, aquesta pràctica ens ha resultat una presa de contacte gradual amb la codificació d'algorismes recursius, ja que els dos primers algorismes (Quick Sort i Merge Sort) havien estat explicats a classe de forma exhaustiva mentre que no s'havia explicat el Bucket Sort. Això ha causat que els primers dos algorismes fossin implementats de forma ràpida i amb confiança amb l'ajut dels apunts, mentre que el tercer ha estat un veritable repte. Al ser un algorisme poc usat i per tant amb poca documentació trobada a Internet, hem hagut de dissenyar-lo des de zero i sense ajuda, amb tots els conceptes que hem vist prèviament, i sense saber si anàvem pel camí correcte o no.

Hem vist com l'algorisme Quick Sort treballa de forma molt ràpida amb volum de dades grans i una mica més lent amb volums de dades molt petits. Per altra banda, el Merge Sort és el més ràpid dels tres amb volums molt petits de dades, però quan augmenta una mica és el més lent amb diferència; i el Bucket Sort en general es manté en valors de creixement paral·lels al Quick Sort però bastant més lent en execució, tot i que molt més ràpid que el Merge Sort.

Un altre aspecte que hem notat és com hi ha algorismes que s'adeqüen a certs casos o problemes millor que altres, com és el cas del Bucket Sort, en el qual hem vist que estava millor preparat per a treballar amb marges de treball coneguts prèviament (en el nostre cas els dígits de la mitjana d'edat, que anaven de 0 a 9 i per tant necessitàvem només 10 buckets) mentre que amb altres tipus de dades el disseny semblava menys aparent i possiblement més complicat.

En resum, hem comprovat la dràstica diferència de temps que pot suposar utilitzar un cert algorisme recursiu sense comprovar si s'adequa als requeriments del problema i al volum de dades utilitzat, en comptes de fer una disseny preventiu i escollir l'algorisme indicat per al nostre problema o programa.

Referències

[1] Bucket sort. (2020). Retrieved 4 December 2020, from https://en.wikipedia.org/wiki/Bucket_sort: :text=Bucket%20sort%2C%20or%20bin%20sort, applying%20the%20bucket%20sorting%20algorithm.

[2] Bucket Sort - GeeksforGeeks. (2020). Retrieved 4 December 2020, from https://www.geeksforgeeks.org/bucket-sort-2/

[3] Bucket Sort Algorithm. (2020). Retrieved 4 December 2020, from https://www.programiz.com/dsa/bucket-sort

[4] Merge sort. (2020). Retrieved 4 December 2020, from https://en.wikipedia.org/wiki/Merge_sort

[5] Quicksort. (2020). Retrieved 4 December 2020, from https://en.wikipedia.org/wiki/Quicksort

[6] Huettig, C. (2020). Gnuplot online - BETA. Retrieved 4 December 2020, from http://gnuplot.respawned.com/

[7] StringBuilder (Java Platform SE 7). (2020). Retrieved 4 December 2020, from https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

[8] Caules, C. (2020). File to String Java 8 y manejo de ficheros. Retrieved 4 December 2020, from https://www.arquitecturajava.com/file-to-string-java8-y-manejo-de-ficheros/