Programació Avançada i Estructura de Dades

Pràctica 1

Grup 12

Roger Miranda Pérez (roger.miranda)

Guillem Serra Cazorla (guillem.serra)

1. INDEX

2. Llenguatge de programació escollit

Durant les primes fases de disseny de la practica ens vam plantejar la utilització de diferents llenguatges de programació, vam valorar fer-ho amb C, C++ i Java.

Una de les avantatges que ens proporcionaves C era que teníem molta més experiència i probablement era la opció més eficient. El problema el qual ens vam trobar va ser que alhora de fer la lectura del JSON amb les dades proporcionades per la pràctica, es complicaria molt.

Una altra opció que vàrem estar valorant va ser la de la utilització de C++ ja que ens facilitava la lectura de JSONs tot i que mai ho havíem fet. Finalment el factor determinant que ens va fer decidir-nos per Java va ser que ja havíem llegit fitxers i ens havia sortit amb relativa facilitat en altres projectes.

3. Algoritmes de Ordenació

En aquesta pràctica hem implementat els algoritmes QuickSort (amb i sense multithreading), MergeSort, InsertionSort, i BucketSort.

Per tal de ordenar els Atletes hem creat la funció actitud que ens ajuda a saber quin atleta té més prioritat:

$$Aptitud = velocitat^2 * distancia$$

3.1 MergeSort

El algorisme MergeSort es un conegut algoritme de ordenació ja que proporciona una ordenació estable en quan a cost.

El seu funcionament es basa en la divisió del array en meitats fins que ens trobem en arrays de longitud 1 o 2. Un cop totes les parts es troben en aquest punt s'aplica el merge que les ordena i les junta. Molts cops aquest algorisme es programa amb recursivitat per facilitar tant la implementació com l'enteniment i elegància.

$$Pitjor\ cas\ \ O(nlog(n))$$

$$Rendiment\ mitjà\ \ O(nlog(n))$$

3.2 QuickSort

El algorisme QuickSort es un dels algorismes més eficients que tractarem. Una de les característiques que té es que es un dels que requereixen menys memòria per tal de fer la ordenació. El seu funcionament es semblant al de "divide and conquer" que es va donar a classe.

$$Pitjor\ cas\ \ O(n^2)$$

$$Rendiment\ mitjà\ \ O(n^2)$$

3.2.1 ParalelQuickSort

Com a algoritme extra també hem implementat el algoritme de QuickSort utilitzant threads per tal de millorar la eficiència en CPUs més potents.

3.3 InsertionSort

InsertionSort és un algorisme de ordenació on un Objecte s'ordena cada cop col·locant-lo a la posició que li toca i desplaçant els altres. Mentre que es una implementació senzilla no té una eficiència gaire bona.

$$Pitjor\ cas\ \ O(n^2)$$

$$Rendiment\ mitjà\ \ O(nlog(n))$$

3.4 BucketSort

BucketSort també es un algorisme de ordenació que te com a característica que fa “buckets” es a dir agrupacions de valors, per exemple el primer “bucket” agafa tots els valors de 0 a 100. Un cop ha fet la agrupació crida una funció com pot ser InsertionSort o QuickSort per ordenar lo que està dins del “bucket”. Aquest algoritme té una limitació important i es que per tal de ser eficient s’ha de prioritzar que la quantitat de Objectes a comparar estigui equitativament distribuït.

Ens ha semblat interessant comparar els costos de temps aplicant diferents algoritmes de ordenació per ordenar cadascun dels buckets.

$$Pitjor\ cas \quad O(n^2)$$

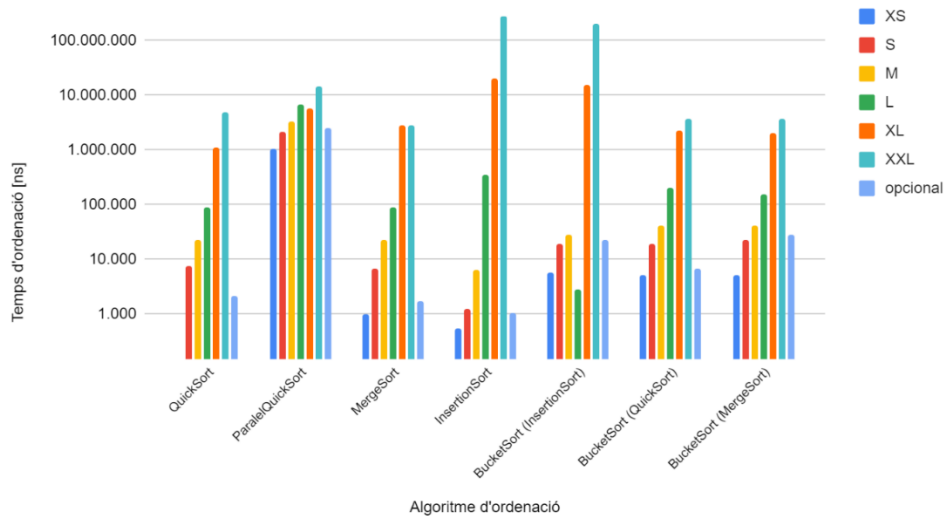
$$Rendiment\ mitjà \quad O(n + \frac{n^2}{numBuckets} + numBuckets)$$

4. Comparativa dels algorismes

4.1 Clubs

Es pot observar els diferents costos de temps dels diferents algorismes al ordenar els Clubs. Cal destacar el rendiment del InsertionSort el qual veiem que augmenta moltíssim com alhora que augmentem la quantitat de dades.

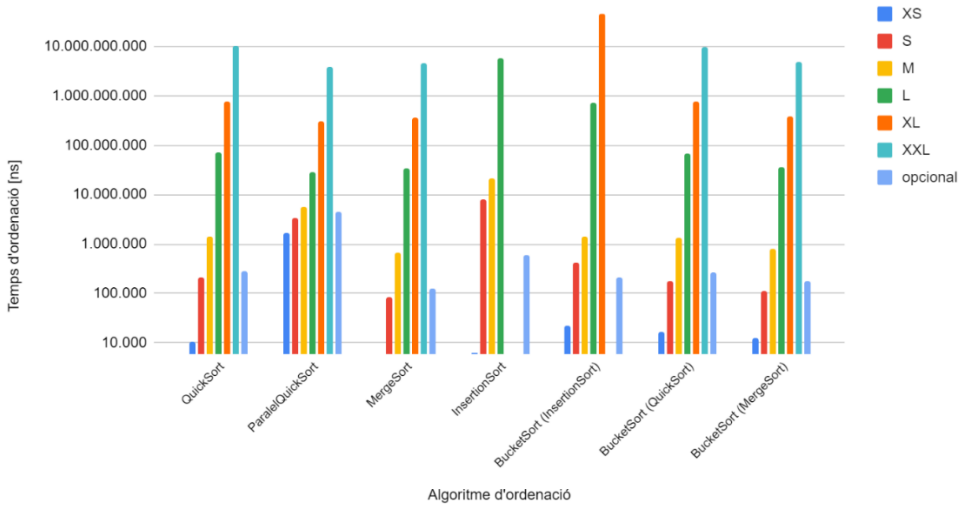
Cal destacar que per l'apartat opcional el InsertionSort es el més ràpid i òptim.



Il·lustració 1 Dades al Annex 10.1

4.2 Atletes per nacionalitat

En la gràfica següent veiem representats els costos temporals al ordenar els Atletes segons la seva nacionalitat i en el cas que siguin equivalents per el seu nom.

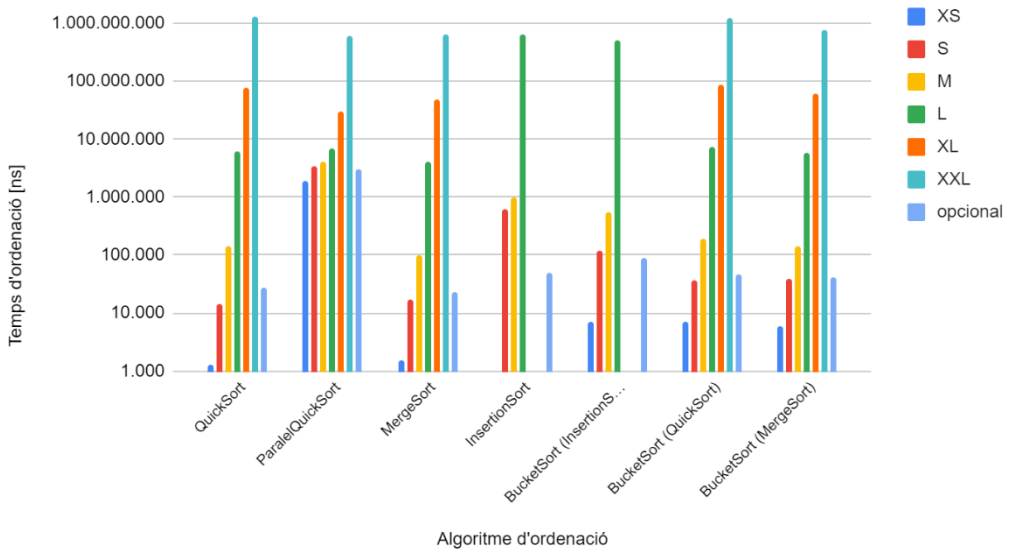


Il·lustració 2 Dades al Annex 10.2

4.3 Atletes per aptitud

Observem com en les anteriors el cost dels diferents algorismes al ordenar Atletes segons la aptitud, recordem que la funció que hem gastat per a definir la aptitud es:

$$Aptitud = velocitat^2 * distancia$$



Il·lustració 3 Dades al Annex 10.3

5. Resultats

5.1 Ordenació de Clubs

Els resultats mostren que per a quantitats de dades petites interessa el InsertionSort (de XS-M) metre que quan s'ha de ordenar dades més grans interessa el MergeSort.

Al ordenar el fitxer "opcional.json" observem que el més òptim és el InsertionSort ja que al només tenir un Club desordenat te cost O(n).

5.2 Ordenació de Atletes per nacionalitat

En aquesta ordenació també trobem una relació directa entre la quantitat de dades i els algoritmes que obtenen costos diferents, podem observar que en aquest cas el Algorisme més òptim en tots els casos es el de MergeSort.

5.3 Ordenació de Atletes per aptitud

Tot i que considerem que la operació és més complexa no trobem un augment representatiu del cost temporal, de fet trobem una disminució del temps comparat amb la ordenació de Atletes per nacionalitat.

En aquesta ordenació el algorisme més òptim és en XS el InsertionSort i en S el QuickSort i en la resta el MergeSort és el més ràpid.

6. Mètode de proves

Al inicial la pràctica una de les coses que vam intentar pensar i implementar primer va ser estandarditzar el sistema de testos dels diferents algoritmes. Un mètode que cridem al finalitzar la ordenació que ens comprovi que la ordenació s'ha fet correctament.

Una de les primeres idees que vam tenir va ser utilitzar un altre algoritme de ordenació que sapiguéssim que funcionava correctament i comparar els resultats que obtenia amb la ordenació que havíem implementat. Aquesta idea va quedar descartada ja que no sabíem si podíem fer crides de funcions de ordenació de Java ja que podia ser invàlid per la pràctica.

La següent idea que se'ns va ocórrer va ser la que tenim implementada a la Pràctica, realitzar una funció que recorri el Array mirant si es compleix que el següent és més gran que el actual fins que finalitza, la qual cosa implica que està ordenat. Com a mètode de informació al sistema en el cas que el algorisme no ordeni correctament fem un *"throw"* de *NotSortedExeption*.

7. Problemes Observats

Un dels problemes que ens hem trobat ha estat alhora de treballar en el projecte des de una altre ordinador ja que utilitzem una llibreria per a poder fer el *@NotNull* que modifica el funcionament de la instanciació de les variables a nivell de compilador. Ens hem trobat que hi havia casos que el IntelliJ detectava que no teníem instal·lada la llibreria i ens la ficava i d'altres amb la opció de ajuda i d'altres que teníem que "forçar" la instal·lació amb un link extern.

Alhora de realitzar la implementació hem trobat dificultats per entendre el pivot en el QuickSort ja que era complicat d'entendre i debugar si fallava el funcionament. Ens hem trobat que el pivot fallava quan dos elements eren equivalents i la seva solució ens ha semblat complicada de extreure.

8. Conclusions

Els resultats que havíem previst encaixen amb els obtinguts en general ja que la relació que hi ha en quant al cost temporal afecta molt a el rendiment del programa.

Una característica que no havíem previst ha estat que hem trobat que el InsertionSort en fitxers petits ha tret rendiments millors que el MergeSort, mentre que ja esperàvem que els rendiments quan teníem fitxers grans fos molt lent, no havíem considerat que pogués ser més ràpid que MergeSort en ningun cas.

Com a extra, observem que el ParalelQuickSort (QuickSort utilitzant paralelisme) només mereix la pena tenint fitxers molt grans, ja que el cost de crear un thread en fitxers petits supera al cost computacional d'ordenar-lo.

9. Bibliografia

Wikipedia [data de consulta: 28 de Novembre de 2020] Disponible en: https://en.wikipedia.org/wiki/Quicksort

Wikipedia [data de consulta: 1 de Desembre de 2020] Disponible en: https://en.wikipedia.org/wiki/Merge_sort

Wikipedia [data de consulta: 2 de Desembre de 2020] Disponible en: https://en.wikipedia.org/wiki/Bucket_sort

GeeksForGeeks [data de consulta: 3 de Desembre de 2020] Disponible en: https://www.geeksforgeeks.org/insertion-sort/

GeeksForGeeks [data de consulta: 29 de Novembre de 2020] Disponible en: https://www.geeksforgeeks.org/quick-sort/

GeeksForGeeks [data de consulta: 2 Desembre de 2020] Disponible en: https://www.geeksforgeeks.org/merge-sort/

GeeksForGeeks [data de consulta: 2 de Desembre de 2020] Disponible en: https://www.geeksforgeeks.org/bucket-sort-2/

Universitat d'Alacant [online] [data de consulta: 5 de Desembre de 2020] Disponible en: http://werken.ubiobio.cl/html/downloads/ISO_690/Guia_Breve_ISO690-2010.pdf

10. Annex

10.1 Dades il·lustració 1

	XS [ms]	S [ms]	M [ms]	L [ms]	XL [ms]	XXL [ms]	opcional [ms]
QuickSort	150	7.553	22.398	89.676	1.106.003	4.887.640	2.086
ParalelQuickSort	1.034.240	2.107.390	3.331.000	6.736.790	5.594.660	14.311.710	2.458.570
MergeSort	1.005	6.810	22.291	90.258	2.761.364	2.795.239	1.718
InsertionSort	550	1.250	6.300	341.000	20.182.100	273.922.150	1.050
BucketSort (InsertionSort)	5.850	19.350	27.950	2.800	14.952.050	205.972.750	22.350
BucketSort (QuickSort)	5.100	19.300	41.700	199.650	2.306.500	3.729.950	6.650
BucketSort (MergeSort)	5.250	22.200	42.100	151.100	2.038.700	3.769.850	27.500

10.2 Dades il·lustració 2

	XS [ms]	S [ms]	M [ms]	L [ms]	XL [ms]	XXL [ms]	opcional [ms]
QuickSort	10.545	212.085	1.435.585	72.188.920	782.466.450	10.263.205.825	283.165
ParalelQuickSort	1.697.580	3.336.220	5.785.820	27.984.900	311.013.340	3.856.829.500	4.378.820
MergeSort	5.930	84.510	667.500	34.247.755	355.036.905	4.595.767.970	123.970
InsertionSort	6.250	7.964.800	21.011.250	5.716.465.250	-	-	599.600
BucketSort (InsertionSort)	21.800	413.075	1.419.900	703.471.800	45.195.049.400	-	205.100
BucketSort (QuickSort)	16.550	178.450	1.309.400	69.040.350	742.056.150	9.571.469.700	267.350
BucketSort (MergeSort)	12.550	113.500	817.050	36.013.250	370.748.150	4.702.510.850	180.250

10.3 Dades il·lustració 3

	XS [ms]	S [ms]	M [ms]	L [ms]	XL [ms]	XXL [ms]	opcional [ms]
QuickSort	1.315	14.645	138.630	6.231.045	76.456.925	1.299.702.160	27.090
ParalelQuickSort	1.877.900	3.325.180	3.987.180	7.013.500	29.567.540	616.944.840	2.967.760
MergeSort	1.530	16.895	98.250	4.095.500	47.330.540	646.237.695	23.010
InsertionSort	950	624.650	977.400	654.944.200	-	-	49.000
BucketSort (InsertionSort)	6.950	120.700	556.650	506.701.350	-	-	87.100
BucketSort (QuickSort)	7.050	37.100	186.800	7.225.600	85.384.650	1.230.858.800	46.050
BucketSort (MergeSort)	5.800	39.050	143.600	5.839.100	61.504.350	777.167.600	41.650