Programació Avançada i Estructura de Dades

Pràctica 1 del primer semestre - Ordenació

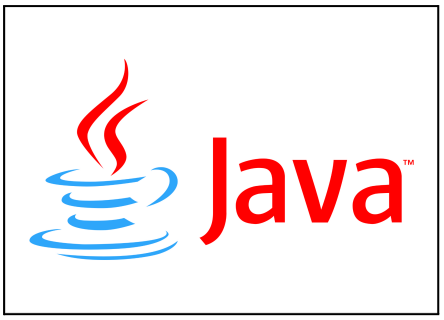
CatTheHobie

Carlos Romero Rodríguez (c.romero)
Marc González Carro (marc.gcarro)
Grup 25
La Salle - Universitat Ramon Llull

23 octubre de 2022

Índex

1. Explicació del llenguatge de programació escollit



VS

Un dels avantatges de Java és la seva compatibilitat multiplataforma. El codi escrit en Java es pot executar a qualsevol plataforma que tingui una màquina virtual de Java instal·lada. Això vol dir que els desenvolupadors poden escriure codi una vegada i executar-lo en múltiples sistemes operatius sense haver de fer canvis addicionals.

Java també ofereix un gran conjunt de biblioteques integrades. Aquestes biblioteques contenen classes i mètodes que els desenvolupadors poden utilitzar per implementar funcionalitats comunes a les seves aplicacions. Això vol dir que els desenvolupadors no han d'escriure tot el codi des de zero, cosa que els permet estalviar temps i esforç.

Java també té una bona seguretat. El llenguatge proporciona mecanismes de seguretat integrats, com ara la seguretat de la màquina virtual de Java, que ajuden a protegir les aplicacions contra atacs externs. A més, Java té un sistema de gestió de memòria automàtic, cosa que significa que els desenvolupadors no s'han de preocupar per administrar manualment la memòria de l'aplicació.

Tot i això, Java també té alguns inconvenients. Un és que la sintaxi del llenguatge pot ser una mica més molesta i complicada que altres llenguatges com .NET o C.

A més, els programes i projectes realitzats amb Java només poden ser executats si es disposa de la màquina virtual de Java, cosa que pot ser un problema per a alguns usuaris.

A més, Java també pot ser menys eficient en termes de rendiment que altres llenguatges com C o C++. Això es deu al fet que Java utilitza una màquina virtual per executar el codi, mentre que altres llenguatges s'executen directament en el sistema operatiu.

Una altra desavantatge pot ser la corba d'aprenentatge una mica més alta en comparació amb altres llenguatges com Python, a causa de la sintaxi més complexa. Això pot fer que sigui una mica més difícil per als nous desenvolupadors d'entendre i utilitzar el llenguatge.

En resum, Java és un llenguatge de programació popular i ampliament utilitzat que ofereix moltes avantatges, com ara compatibilitat multiplataforma, un gran conjunt de biblioteques integrades i seguretat integrada. No obstant, també té alguns inconvenientes, com ara sintaxi complexa i una possible menor eficiència en comparació amb altres llenguatges.

2. Comparativa d'algorismes d'ordenació

2.1. Anàlisis Merge Sort

2.1.1. Teoría

Nota: el cost d'aquest algorisme d'ordenació es divideix en dos parts:

- El cost d'ordenar recursivament 2 mitats d'array (dreta i esquerra).
- El cost de combinar les 2 mitats d'array en 1 array (funció merge).

A partir d'aquí i tenint això en compte:

- Si l'array te "n" objectes, el cost serà C: C(n).
- En cas que l'array sigui imparell, la mitad esquerra tindrà un element extra.

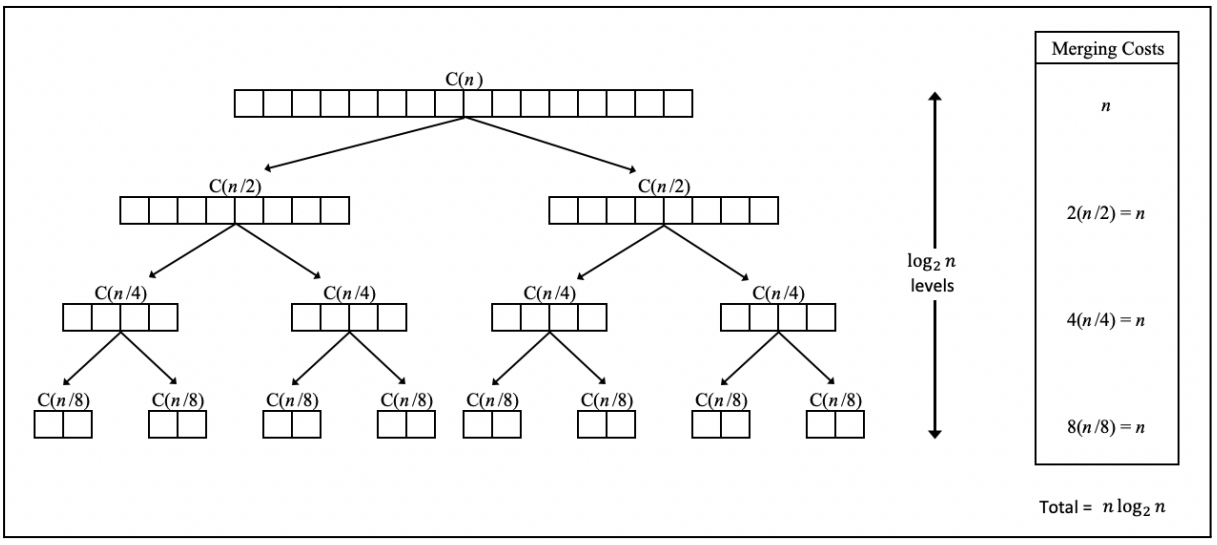
Per tant, el seu cost sería tal que així:

- $C(n) = \underbrace{C([n/2])}_{\text{Array Dreta}} + \underbrace{C([n/2])}_{\text{Array Esquerra}} + n \quad , \quad n > 1 \ \& \ C(1) = 0.$

Simplificant:

- $C(n) = 2C([n/2]) + n \quad , \quad n > 1 \ \& \ C(1) = 0.$

Per deduir el cost de l'algorisme, preferim fer-ho a partir de una imatge il·lustrativa:



Tal i com es pot apreciar a l'imatge, quan les meitats es converteixen en només en únic array, ja no tenim cap cost recursiu o de fusió, ja que l'array ja està ordenat. A l'arbre que es mostra a dalt, només cal tenir en compte 4 nivells de "meitats", ja que el següent nivell consistiría únicament en arrays d'un sol element.

Com es pot veure, cada nivell aporta un total de n assignacions d'array; si l'arbre tingués x nivells, el cost total seria (x * n). Aleshores, per un array de "n = 2x" elements, només podem dividir "n = 2 * x" a la meitat un màxim de "x" vegades, cosa que suggereix que el nombre de nivells és "$x = \log_2(n)$". Per tant, el cost total ve donat per "$C(n) = n * \log_2(n)$".

2.1.2. Ordenació dels vaixells alfabèticament

En quant a la práctica, nosaltres hem volgut utilitzar aquest algorisme per a ordenar alfabèticament els vaixells (3.2 Embarcacions en funció del nom).

Per fer-ho primer hem hagut de llegir el fitxer de vaixells per líneas, parsejar la linea amb la funció ".split()" de java i guardar la informació del vaixell en una classe vaixell la qual té els atributs mencionats al PDF)

```
public class Barco {

    2 usages
    private final int id;
    2 usages
    private final String name;
    2 usages
    private final String type;
    2 usages
    private final double peso;
    2 usages
    private final double slore;
    2 usages
    private final int cap;
    2 usages
    private final int n_comp;
    2 usages
    private final String state;
    2 usages
    private final int v;
    2 usages
    private final String center;
```

Una vegada obtingudes les dades dels vaixells, fem un bucle "for" per a recorrer tots els vaixells guardats i per a cada vaixell ens guardem el nom en un array de "Strings", d'aquesta manera obviem tindre que comparar objectes amb l'interfície Comparable de Java.

```
private static void ordenarAlfabeticamente (ArrayList<Barco> barcos) {
    String[] list = new String[barcos.size()];

    for (int i = 0; i < barcos.size(); i++) { // llenar el array de strings "list"
        Barco aux1 = barcos.get(i);           // con todos los nombres de los barcos
        list[i] = aux1.getName();
    }
```

Una vegada tenim tots els noms emmagatzemats en un array, el passem com a paràmetre d'entrada a la funció *"MergeSort"* la qual a part de rebre aquest parametre, també rep la mesura de l'array (0, longitud array de noms - 1).

```java
public static void mergeSort(String[] a, int i, int j) {
    if (i >= j) {
        return;
    }
    int mid = (i + j) / 2;
    mergeSort(a, i, mid);
    mergeSort(a, i: mid + 1, j);
    merge(a, i, mid, j);
}
```

La primera vegada que vem entregar sobre 10 el document vem cometre l'errada d'utilitzar dos arrays auxiliars (dreta i esquerra) per fer el mergeSort, però posteriorment de veure la solució penjada al PDF i de fer l'entrevista, vem veure que és pot resoldre amb un únic array per realitzar el MergeSort. Per tant, hem cambiat l'algorisme recursiu ja que cal tenir en compte també el cost, el qual també varia fent-ho amb un o dos arrays auxiliars, que és del que tracta la pràctica.

```java
public static void mergeSort(String[] a) {
    // Si el array tiene 1 elemento, ya está ordenado
    if (a.length >= 2) {
        String[] left = new String[a.length / 2];
        String[] right = new String[a.length - (a.length / 2)];

        for (int i = 0; i < left.length; i++) {
            left[i] = a[i];                  // llenar el array izquierdo con los numeros de la mitad izquierda
        }
        for (int i = 0; i < right.length; i++) {
            right[i] = a[i + (a.length / 2)]; // llenar el array derecho con los numeros de la mitad hasta el final
        }

        mergeSort(left);       // ordenar array izquierdo recursivamente
        mergeSort(right);      // ordenar array derecho recursivamente

        merge(a, left, right); // juntar ambos arrays ya ordenados por separado
    }
}
```

2.1.3. Mergesort d'un array auxiliar vs MergeSort de dos arrays auxiliars

L'ús d'un array auxiliar a l'algoritme d'ordenament MergeSort permet fer les operacions de fusió de manera més eficient, reduint el temps d'execució de l'algorisme. L'array auxiliar s'utilitza per emmagatzemar temporalment les dades mentre es realitzen les operacions de fusió, cosa que evita la necessitat de sobreescriure les dades originals a l'array principal i redueix el cost de la complexitat temporal. A més, utilitzar un array auxiliar permet fer les operacions de fusió en un sol pas, cosa que redueix el temps d'execució de l'algorisme i el cost de complexitat temporal.

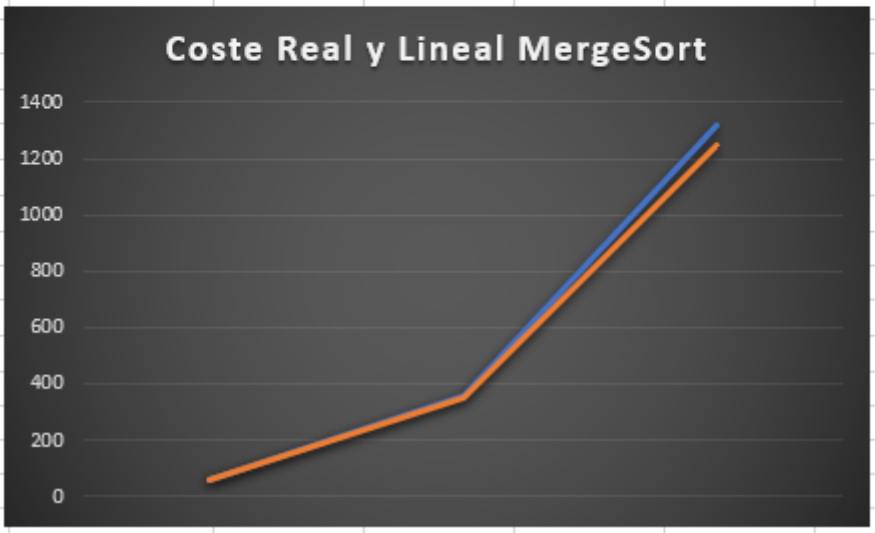
Utilitzar dos arrays en lloc d'un de sol per realitzar l'algoritme d'ordenament MergeSort pot resultar en un ús més gran de la memòria i un temps d'execució més lent a causa de la

necessitat de realitzar múltiples operacions de còpia i fusió, augmentant el cost de complexitat temporal i espacial. En resum, l'ús d'un array auxiliar a l'algoritme d'ordenament MergeSort és més eficient en termes de temps i ús de memòria en comparació amb l'ús de dos arrays, ja que redueix el cost de complexitat temporal i espacial.

*2.1.4. Demostració cost del MergeSort (n*log(n))*

En base al número de iteracions que es crida a l'algoriisme, calculades a partir d'un contador global, hem apuntat el número de operacions per els diferents tamanys de l'array d'entrada de la funció, el qual és directamente al tamany del fitxer de vaixells. Una vegada apuntades el número d'iteracions que fa l'algoriisme de manera recursiva per a cada fitxer que són les següents:

	Demostración coste MergeSort (n*log(n))			
	XS	S	M	L
Iteracions Real	29	59	359	1319
Iteracions Lineal	29	58	348	1246
	x1 (15/15)	x2 (30/15)	x12 (180/15)	x44 (660/15)



En l'imatge podem observar el número d'iteracions de l'algorisme en funció del fitxer de vaixells. Podem observar que el cost real y el cost lineal calculat manualment multiplicant el número de vaixells per el coeficient de relación entre el fitxer obert. D'aquesta manera, una vegada representades les iteraciones reals de l'algorisme i les calculades podems observar que en efecte el cost de l'algorisme és n*log(n) sempre. Això és perquè en cada crida recursiva, l'algorisme divideix l'array en dos subarrays i els ordena recursivament. Com que la mida de l'array es redueix a la meitat en cada trucada recursiva, el nombre total de trucades recursives és log(n).

A més, a cada crida recursiva, l'algorisme realitza una operació de fusió per ordenar els elements als subarrays. Aquesta operació té un cost d'O(n), on n és la mida del subarray.

Com que la mida del subarray és la meitat de la mida de l'array original a cada trucada recursiva, el cost total de les operacions de fusió és n*log(n).

Per tant, el cost total de l'algorisme mergeSort és nlog(n), ja que el nombre total de trucades recursives és log(n) i el cost de les operacions de fusió és nlog(n).

2.2. Anàlisis QuickSort

2.2.1. Teoría

- **Millor cas**
  El millor cas per a l'ordenació ràpida es produeix quan cada partició divideix l'array en dues meitats iguals. Aleshores, el cost d'ordenar "n" elements ve donat per:
  - El cost de partició de "n" elements
  - El cost de l'ordenació recursiva dels dos array de mida "n/2".

  Com que la mida de cada meitat és n/2, la relació de recurrència per a la funció de cost (per a comparacions) està directament relacionada amb l'ordenació del MergeSort:
  - $C(n) = 2C(n/2) + n \quad , \quad n > 1 \ \& \ C(1) = 0.$

  El resultat és similar, amb la funció de cost en termes de comparacions per al QuickSort també: $\mathbf{\Omega (n * \ln_2 (n))}$

- **Pitjor Cas**
  El pitjor cas per el QuickSort es produeix quan la partició no divideix l'array (és a dir, quan un conjunt no té cap element). Això passa quan l'array està ordenat.

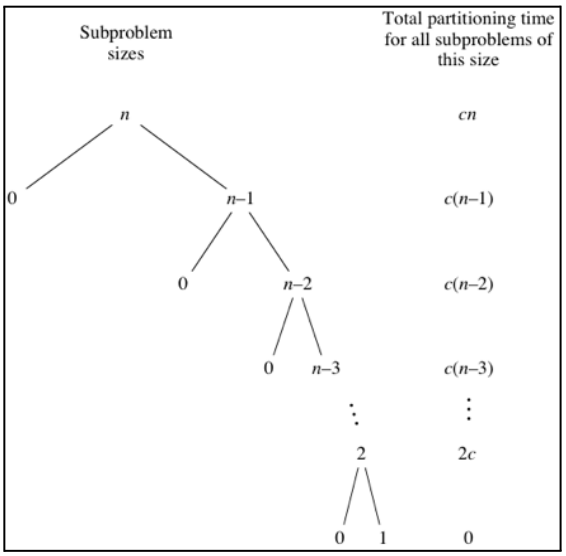
  En aquest cas el cost es resoldria de la següent manera:
  - El cost de dividir un array de mida 1 és 0 i el cost d'un array de n > 1 elements és n+1 (aquest últim prové de comparar el primer element a la dreta del pivot per descobrir que s'ha de "posar a la dreta", i després comparar "n" elements amb el pivot inclòs (inclòs el propi pivot) mentre es recorre l'array cap enrere des del seu final, buscant un element per "posar a l'esquerra").

  - El cost per ordenar recursivament els n−1 elements restants (excloent el pivot) que són tots més grans que el pivot i, per tant, estàn a la seva dreta. El cost per ordenar els zero elements a l'esquerra del pivot és zero, igual que l'altre cost base per ordenar un sol element [C(0)=C(1)=0].

  Per tant, la relació de recurrència en aquest cas és lleugerament diferent:

  $$C(n) = C(n - 1) + n + 1 \quad , \quad C(1) = 0$$

Resolent la relació de recurrència:



C(n) = C(n - 1) + (n + 1) =
= [C(n−2) + n] + (n+1) =
= [C(n−3) + (n−1)] + n + (n + 1) = … =
= C(1) + 3 + ··· + (n−2) + (n−1) + n + (n+1)
= 0 + 3 + ··· + (n−2) + (n−1) + n + (n+1) =
= [1 + 2 + 3 + ··· + (n−2) + (n−1) + n] + (n−2) = (n * (n+1) / 2) + (n−2) =
= n^2 / 2 → **O (n^2)**

Puntualització: Atès que el pitjor dels casos es produeix quan l'array ja està ordenada o molt ordenada, podem garantir probabilísticament que això no succeeixi barrejant de manera intencionada els elements de l'array abans d'utilitzar l'algorisme QuickSort. Bé és cert que això comporta un lleuger cost addicional, però val la pena minimitzar fins al punt de menysprear la possibilitat d'un cost O (n^2).

2.2.2. Ordenació dels vaixells per prestacions

Per a la realització de l'ordenació dels vaixells per prestacions, una vegada s'ha llegit el fitxer de text i s'ha guardat la informació de cada vaixell en un objecte de tipus vaixell el qual té els atributs del PDF:

```
public class Barco {

    2 usages
    private final int id;
    2 usages
    private final String name;
    2 usages
    private final String type;
    2 usages
    private final double peso;
    2 usages
    private final double slore;
    2 usages
    private final int cap;
    2 usages
    private final int n_comp;
    2 usages
    private final String state;
    2 usages
    private final int v;
    2 usages
    private final String center;
```

Hem tingut que pensar una mica per a l'ordenació dels mateixos, ja que tot i que es pot fer el càlcul de manera random sumant, restant, multiplicant o dividint números i ordenar-los en funció d'aquests números, hem volgut considerar l'opció d'ordenar-los bastant-nos en la lògica real. per tant en base al PDF el qual diu el següent "S'haurà de combinar el ***pes***,

*l'**eslora**, la **capacitat** i la **velocitat màxima*** per tal d'obtenir un criteri fiable", hem considerat que el millor vaixell serà el que tingui:

- Un **pes menor**, ja que contra menys pes, més lleuger i ràpid acostuma a ser el vaixell.
- Una **eslora més gran**, ja que interessa que puguin caber diverses persones en ell.
- Una **capacitat major**, tot i que aquesta bé directament relacionada amb l'eslora.
- Una **velocitat màxima major**, ja que contra més ràpid sigui un vaixell més car i millor és, dela mateixa manera que els cotxes o els avions.

Per tant, una vegada considerat en base a que es farà l'ordenació, vam utilitzar una fòrmula matemàtica per a obtenir un valor únic per a cada vaixell en funció de les seves prestacions:

```
prestaciones[i] = (float) (aux1.getPeso()/(aux1.getSlore()+aux1.getCapacity()+aux1.getVelocity()));
```

Cal tenir en compte que contra més petit sigui el valor decimal de l'operació matemàtica, millor serà el vaixell. Ja que contra més petit sigui el pes, i més gran sigui l'eslora, la capacitat i la velocitat, el valor tendirà més cap a 0. En canvi un vaixell amb poca velocitat i poca eslora, capacitat i velocitat tindrà un valor molt més gran que 0, i per tant a l'hora d'ordenar-lo estarà al final ja que serà un dels pitjors.

Per tant, una vegada s'han obtingut els valors únics de cada vaixell en base a les seves prestacions, s'han col·locat en un array de "float" per a passar-ho com a paràmetre d'entrada de l'algorisme QuickSort i ordenar les prestacions de menor a major.

```
private static void ordenarXPrestaciones (ArrayList<Barco> barcos) {
    float[] prestaciones = new float[barcos.size()];

    for (int i = 0; i < barcos.size(); i++) { // llenar el array de strings "prestaciones"
        Barco aux1 = barcos.get(i);           // con todos los valores de la division
        prestaciones[i] = (float) (aux1.getPeso()/(aux1.getSlore()+aux1.getCapacity()+aux1.getVelocity()));
    }
```

Per a la realització de l'algorisme em passat com a paràmetres d'entrada un array de flotants "barcos_id" i els índexs "izq" i "der", que indiquen el rang del subarray a ordenar.

En primer lloc, es pren el primer element de l'array com a pivot i s'inicialitzen dues variables i i j per dur a terme les cerques d'esquerra a dreta i dreta a esquerra, respectivament. L'algorisme aleshores entra en un bucle while en el qual es busquen elements majors que el pivot a la recerca de i, i elements menors que el pivot a la recerca de j. Si existeixen aquests elements, s'intercanvien entre si.

Quan les cerques i i j es creuen, es col·loca el pivot en el seu lloc correcte i es crida recursivament a l'algorisme per ordenar els subarrays a l'esquerra i dreta del pivot. Això es repeteix fins que tot el array estigui ordenat.

```
public static void quickSortFloat(float[] barcos_id, int izq, int der) {
    float pivote = barcos_id[izq]; // tomamos primer elemento como pivote
    int i = izq;                   // i realiza la búsqueda de izquierda a derecha
    int j = der;                   // j realiza la búsqueda de derecha a izquierda
    float aux;

    while (i < j) {                                  // mientras no se crucen las búsquedas
        while (barcos_id[i] <= pivote && i < j) i++; // busca elemento mayor que pivote
        while (barcos_id[j] > pivote) j--;           // busca elemento menor que pivote
        if (i < j) {                                 // si no se han cruzado
            aux = barcos_id[i];                      // los intercambia
            barcos_id[i] = barcos_id[j];
            barcos_id[j] = aux;
        }
    }

    barcos_id[izq] = barcos_id[j]; // se coloca el pivote en su lugar de forma que tendremos los
    barcos_id[j]=pivote;           // números más pequeños a su izquierda y los más grandes su derecha

    if(izq < j-1) {
        quickSortFloat(barcos_id,izq, der: j-1);          // ordenamos subarray izquierdo
    }
    if(j+1 < der)
        quickSortFloat(barcos_id, izq: j+1,der);          // ordenamos subarray derecho
}
```

Pel que fa als costos, l'algorisme de quicksort és de complexitat O(n log n) de mitjana i O(n^2) en el pitjor dels casos, a causa de l'elecció inadequada del pivot. Tot i això, hi ha millores de rendiment que es poden aplicar per evitar el pitjor dels casos, com triar el pivot de manera aleatòria o triar el pivot com l'element mitjà del subarray. També es pot fer servir una tècnica de partició dual pivot, que ajuda a reduir la probabilitat de caure en el pitjor dels casos.

2.2.3. Anàlisis de resultats en funció del pívot escollit

- Pivot com a element mitjà del subarray:

```
public static void quickSortFloat(float[] barcos_id, int izq, int der) {
    int med = (izq+der)/2;
    float pivote = barcos_id[med];
```

- Pivot com a aleatòri:

```
public static void quickSortFloat(float[] barcos_id, int izq, int der) {
    Random rand = new Random();
    int randomIndex = izq + rand.nextInt( bound: der - izq);
    float pivote = barcos_id[randomIndex];
```

- Dual pivot:

```
public static void quickSortFloat(float[] barcos_id, int izq, int der) {
    if (barcos_id[izq] > barcos_id[der]) {
        float aux = barcos_id[izq];
        barcos_id[izq] = barcos_id[der];
        barcos_id[der] = aux;
    }
    float pivot1 = barcos_id[izq], pivot2 = barcos_id[der];
    int i = izq + 1, j = der - 1, k = i;
    float aux;

    while (k <= j) {
        if (barcos_id[k] < pivot1) {
            aux = barcos_id[i];
            barcos_id[i] = barcos_id[k];
            barcos_id[k] = aux;
            i++;
        } else if (barcos_id[k] >= pivot1 && barcos_id[k] <= pivot2) {
            k++;
        } else {
            aux = barcos_id[j];
            barcos_id[j] = barcos_id[k];
            barcos_id[k] = aux;
            j--;
        }
    }
```

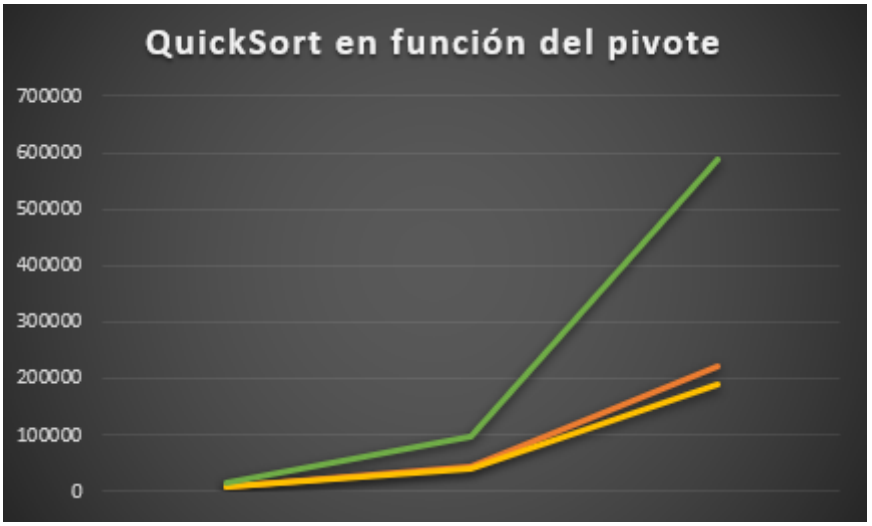
```
int pivotIndex1 = i - 1;
int pivotIndex2 = j + 1;
barcos_id[izq] = barcos_id[pivotIndex1];
barcos_id[pivotIndex1] = pivot1;

barcos_id[der] = barcos_id[pivotIndex2];
barcos_id[pivotIndex2] = pivot2;

if (izq < pivotIndex1 - 1) {
    quickSortFloat(barcos_id, izq, der: pivotIndex1 - 1);
}

if (pivotIndex1 + 1 < pivotIndex2 - 1) {
    quickSortFloat(barcos_id, izq: pivotIndex1 + 1, der: pivotIndex2 - 1);
}

if (pivotIndex2 + 1 < der) {
    quickSortFloat(barcos_id, izq: pivotIndex2 + 1, der);
}
```

Si fem 10 execucions per a cada fitxer amb cadascun dels algorismes obtenim la següent taula:

	QuickSort (Primer element com a pivot)				QuickSort (Element mitjà com a pivot)				QuickSort (Dual pivot)			
	XS	S	M	L	XS	S	M	L	XS	S	M	L
Ejecución 1	4800	10700	42400	185200	4100	6800	40500	185200	6800	20400	102300	734100
Ejecución 2	4400	7500	43800	263900	4400	7700	32000	194800	6200	21300	103200	564700
Ejecución 3	6500	7100	45000	283300	4200	7100	45000	185500	5800	12900	98500	622400
Ejecución 4	4200	7700	40000	272300	3800	7700	40000	201600	5800	13000	99100	578400
Ejecución 5	6200	7000	42400	263800	4500	6200	42400	194800	5700	13100	97600	535100
Ejecución 6	6400	7300	60900	184300	4600	7300	39000	190400	6300	13800	98200	545600
Ejecución 7	6300	7000	41300	188900	5800	7000	41300	188900	6100	13400	103200	575600
Ejecución 8	6300	10900	40700	190400	4300	6300	41400	182300	6300	14200	88900	579600
Ejecución 9	6400	7300	41100	185500	4100	7300	41100	185500	6100	13700	90500	583400
Ejecución 10	4600	6900	41300	182300	4900	6900	41300	182300	6200	19900	88400	563300
Real:	5610	7940	43890	219990	4470	7030	40400	189130	6130	15570	96990	588220

En aquesta taula podem veure les 10 execucions per a cada fitxer (de mida 15, 30, 180 i 660 vaixells) i el temps que ha tardat per a cada fitxer, calculat com la mitjana aritmètica dels 10 temps de cadascuna de les execucions.

En base a aquesta taula, si realitzem una gràfica per a poder analitzar més còmodament els resultats podem veure el següent:



El color verd representa el Dual Pivot, el color taronja el element mitjà com a pivot i el colot groc el primer element com a pivot.
Clarament podem obseervar que el duual pivot triga molt més que si bé agafem com a pivot el pivot mitjà o el primer element, això té una explicació:

L'algorisme QuickSort recursiu amb dual pivot pot trigar més que si s'escull com a pivot l'element mitjà o el primer element a causa de diverses raons:

- Overhead de la selecció de dos pivots: En triar dos pivots en lloc d'un de sol, s'augmenta l'overhead de la selecció dels pivots i la partició del subarray en base a ells. Això pot augmentar el temps dexecució de l algoritme en comparació amb la selecció dun sol pivot.
- Major complexitat a la partició: En dividir el subarray en tres seccions en lloc de dos, augmenta la complexitat de la partició i el temps d'execució de l'algorisme.
- Major probabilitat de caure en casos desfavorables: Encara que la tècnica de partició dual pivot redueix la probabilitat de caure en el pitjor dels casos en comparació amb la selecció d'un sol pivot, encara hi ha més probabilitat de caure en casos desfavorables en comparació amb la selecció d´un sol pivot. Això pot augmentar el temps dexecució del algorisme.

En quant a per què l'algorisme QuickSort recursiu amb l'element mitjà com a pivot pot trigar més que si es tria el primer element com a pivot, hi ha diverses raons:

- Overhead de la selecció del pivot mitjà: En triar l'element mitjà com a pivot, s'augmenta l'overhead de la selecció del pivot mitjà en comparació amb la selecció del primer element.

- Major complexitat a la partició: En triar l'element mitjà com a pivot, augmenta la complexitat de la partició i el temps d'execució de l'algorisme en comparació amb la selecció del primer element.
- Major probabilitat de caure en casos desfavorables: En triar l'element mitjà com a pivot, hi ha més probabilitat de caure en casos desfavorables en comparació amb la selecció del primer element.
-

En resum, l'algorisme QuickSort recursiu amb dual pivot pot trigar més que si s'escull com a pivot l'element mitjà o el primer element a causa de l'overhead addicional de la selecció de dos pivots, la major complexitat a la partició i la major probabilitat de caure en casos desfavorables. Mentre que l'algorisme QuickSort recursiu amb element mitjà com a pivot pot trigar més que si es tria el primer element com a pivot a causa de l'overhead addicional de la selecció del pivot mitjà, la major complexitat en la partició i la major probabilitat de caure en casos desfavorables.

2.3. Anàlisis BucketSort

2.3.1. Teoría

La complexitat del temps a Bucket Sort depèn en gran mesura de la mida de la llista de cubs i també de l'interval sobre el qual s'han distribuït els elements de l'array. És a dir, si els elements de l'array no tenen una diferència matemàtica significativa entre ells, podria resultar que la majoria dels elements s'emmagatzemen al mateix cub. Fet que dificultaria significativament la complexitat de l'algorisme.
Suposem que hi ha un total de c galledes diferents, de manera que el bucle més exterior trigarà almenys O(c) temps.
El bucle intern trigarà almenys O(n) temps en general, ja que hi ha un total de n elements distribuïts per la llista de cubs.
Per tant, podem concloure que la complexitat global de l'ordenació de cubs serà **O(n + c).**

2.3.2. Ordenació dels vaixells per antiguitat

Primer de tot, pero ordenar els ID del vaixells més còmodament i evitar tindre que utilitzar la interfície comparable per a comparar objectes "Vaixell", hem fet un bucle per omplir un array d'enters auxiliar que és el que utilitzarem per a ordenar els ID dels vaixells de major a menor, tal i com apareix explícitament a l'enunciat de la pràctica.

```
//almacenar todos los "id" en un array de enteros
for (int i = 0; i < barcos.size(); i++) {
    Barco aux1 = barcos.get(i);
    barcos_id[i] = aux1.getId();
}
long start = System.nanoTime();
bucketSort(barcos_id, order: 10, buckets); //funcion recursiva
long end = System.nanoTime();
```

Per programar el nostre algoritme BucketSort, hem utilitzat una tècnica de partició estàtica per dividir els elements en cubs, on el rang de cada cub es calcula en funció del màxim i

mínim dels valors a l'array d'entrada. En aquest cas, s'utilitzen 10 cubs, cosa que significa que el nombre de cubs és constant, però al següent apartat analitzarem què passa si es canvia la quantitat de cubs.

```
private static void bucketSort (int[] barcos_id, int order, List<Integer>[] buckets) {
    int numBuckets = 10, bucketRange, index = 0, max = 0, min = 0;
    counterBucketSort++;
    if (order == 10) {
```

D'altra banda per ordenar cada Bucket hem utilitzat un algorisme QuickSort recursiu (aprofitant la base del que s'utilitza per ordenar els vaixells en funció de les seves prestacions, però canviant l'array de float a sencer, ja que els ID dels vaixells són sencers) la complexitat és n*log(n), on n és la grandària del cub.

```
} else{
    //Pasamos la arraylist del cubo seleccionado a un array de enteros para poder manejarlo mejor
    int[] aux = new int[buckets[order].size()];
    for (int k = 0; k < buckets[order].size(); k++){
        aux[k] = buckets[order].get(k);
    }
    //Ordenamos el array mediante un QuickSort
    counterQuickSortForEachBucket++;
    quickSortInt(aux, izq: 0, der: aux.length-1);
    //Asignamos el valor del array ordenado a la Arraylist del cubo seleccionado
    for (int p = 0; p < buckets[order].size(); p++){
        buckets[order].set(p, aux[p]);
    }
    bucketSort(barcos_id, order: order - 1, buckets);
}
```

Per tant, el cost total de l'algorisme BucketSort seria n*log(n), ja que el nombre de cubs és constant i el cost d'ordenar cada cub és n*log(n). No obstant això, en casos on els elements estan distribuïts de manera desequilibrada als cubs, l'algorisme BucketSort pot tenir un rendiment significativament pitjor, ja que alguns cubs podrien estar buits o contenir molt pocs elements, mentre que altres podrien estar plens o contenir un gran nombre d'elements. En aquest cas, el rendiment de l'algorisme QuickSort a cada cub es veuria afectat i el rendiment global de l'algorisme BucketSort també es veuria afectat.

2.3.3. Anàlisi de rendiment de l'algorisme BucketSort amb diferents números de cubs

En base a probar amb diferents quantitats de Buckets (1, 10, 100, 700) i veure el temps d'execució i el número d'iteracions de l'algorisme hem pogut concluir el següent:

[La raó per la que hem triat 1, 10, 100 i 700 buckets no és arbitrària, si no perquè un bucket pot ser interessant analitzar-ho, 10 buckets perquè és el número que teniem de buckts i ens era molt equilibrat, 100 buckets perquè ens era interessant analitzar-ho amb una gran quantitat de buckets i 700 buckets perquè com en el fitxer més gran hi ha 660 vaixells, voliem que quedessin buckets buits per analitzar el rendiment de cara a l'anàlisis final.]

El nombre de buckets a l'algorisme BucketSort recursiu té un impacte significatiu en el rendiment de l'algorisme. Quan s'utilitzen més buckets, es divideix el conjunt de dades en més segments, cosa que permet una distribució més equilibrada de les dades entre els buckets. Això redueix la mida mitjana dels buckets i, per tant, redueix el temps d'ordenació necessari per a cada bucket. Tot i això, també augmenta el nombre de buckets que han de ser ordenats, la qual cosa pot augmentar el temps total d'execució de l'algorisme.

D'altra banda, quan s'utilitzen menys buckets, es divideix el conjunt de dades en menys segments, cosa que pot causar una distribució desequilibrada de les dades entre els buckets. Això augmenta la mida mitjana dels buckets i, per tant, augmenta el temps d'ordenament necessari per a cada bucket. Tot i això, també redueix el nombre de buckets que han de ser ordenats, la qual cosa pot disminuir el temps total d'execució de l'algorisme.

En resum, el canvi del nombre de buckets a l'algorisme BucketSort recursiu té un impacte directe en la complexitat de l'algorisme. A mesura que augmenta el nombre de buckets, la complexitat de l'algorisme s'acosta a O(n), ja que els elements es distribueixen de manera més equitativa entre els buckets, cosa que redueix el nombre d'iteracions necessàries per ordenar els elements. No obstant això, també es requereix un major ús de memòria a causa de laugment en el nombre de buckets. D'altra banda, si es redueix el nombre de buckets, la complexitat de l'algorisme augmenta a causa de la major quantitat d'elements que s'han d'ordenar a cada bucket, cosa que implica un nombre més gran d'iteracions i un menor ús de memòria. Per tant, l'equilibri adequat entre el nombre de buckets i la complexitat de l'algorisme és essencial per obtenir el millor rendiment en termes de temps i de memòria.

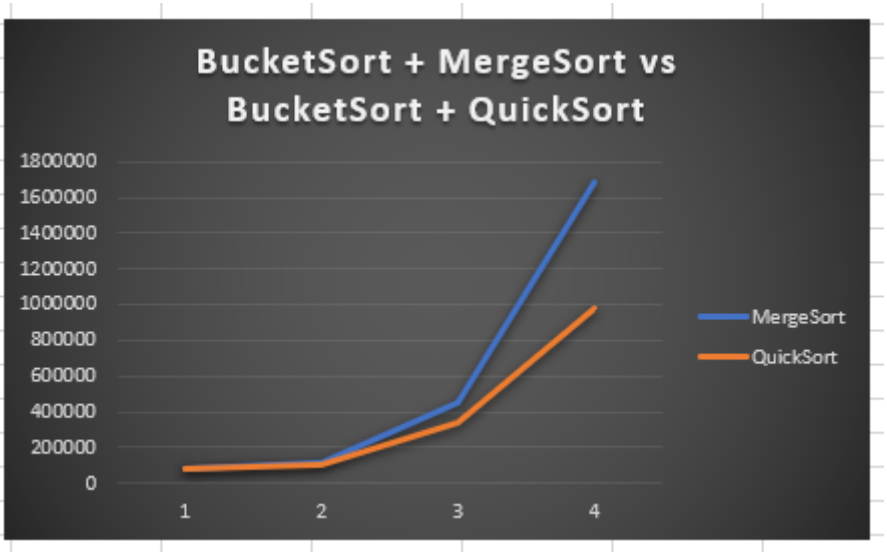
2.3.4. Ordenació dels buckets MergeSort vs QuickSort

En base a 10 opertures de fichers de vaixell de cada tipus (XS, S, M i L), hem recopilat tots els resultats en un excel. Per evitar errors en els càlculs hem realitzat 10 opertures per a cada ficher, en total 80 opertures, y hem fet la mitjana aritmètica per a cada ficher, tant per el Bucket Sort amb MergeSort, com amb BucketSort amb QuickSort.
La taula queda tal que així: (Els temps están en nanosegons per a més precisió).

	BucketSort + MergeSort				BucketSort + QuickSort			
	XS	S	M	L	XS	S	M	L
Ejecución 1	63200	92600	506400	1719500	77600	91200	347300	1092000
Ejecución 2	73300	118700	409300	1620300	86600	98900	325600	1007200
Ejecución 3	79600	103900	442800	1574900	103200	103900	325500	1041500
Ejecución 4	104200	101400	456600	1745100	86600	112300	339500	932100
Ejecución 5	74900	126800	411200	1677100	67800	106800	373500	1005700
Ejecución 6	101300	95700	419600	1493500	86500	113800	289900	825800
Ejecución 7	104100	93800	428000	2075800	65500	109200	364100	927200
Ejecución 8	96400	164100	587300	1616700	77600	83100	316000	963700
Ejecución 9	71200	116700	362500	1598200	74600	106500	392800	928000
Ejecución 10	63500	128900	442800	1729900	61100	86200	370100	1010900
Mitjana:	83170	114260	446650	1685100	78710	101190	344430	973410
Num Iteracions	17	18	22	23	17	18	22	23

Una vegada recopilats tots els temps, els hem graficat per a veure amb més precisió la diferència entre el BucketSort que utilitza un QuickSort per a ordenar cada bucket, com el BucketSort que utilitza un MergeSort per a ordenar cada bucket.

La gràfica queda tal que així:



En base a la gràfica podem observar que el temps d'execució és més gran quan s'utilitza l'algoritme d'ordenament MergeSort en lloc de QuickSort, això es deu a diversos factors. En primer lloc, el MergeSort és un algorisme d'ordenament "divideix i venceràs", la qual cosa significa que es divideix l'arranjament original en diverses subllistes/subproblemes fins que cadascuna només tingui un element, i després es combinen de manera ordenada. Això requereix un nivell més gran de complexitat i, per tant, pot requerir més temps per executar.

En comparació, el QuickSort és un algorisme d'ordenament "divideix i conquesta" que es basa en la selecció d'un element pivot i la partició de l'arranjament en dos subllistes, una amb elements menors al pivot i una altra amb elements més grans al pivot. Tot seguit s'ordenen recursivament les subllistes. El QuickSort és considerat un algorisme més eficient en temps d'execució en comparació del MergeSort a causa de la seva estratègia de particionar l'array i la seva elecció de l'element pivot.

A més, el QuickSort és un algorisme in-place, el que significa que utilitza una estructura de dades per emmagatzemar les dades a ordenar, en lloc de crear una nova estructura de dades per emmagatzemar les dades ordenades. Això també contribueix a la seva eficiència en temps d'execució.

En resum, el BucketSort és un algorisme eficient en temps d'execució, però en canviar l'algoritme d'ordenament utilitzat a cada cub (de QuickSort a MergeSort) s'augmenta la complexitat i per tant el temps d'execució. És important tenir en compte aquests factors en triar l'algorisme d'ordenament adequat per a una tasca específica.

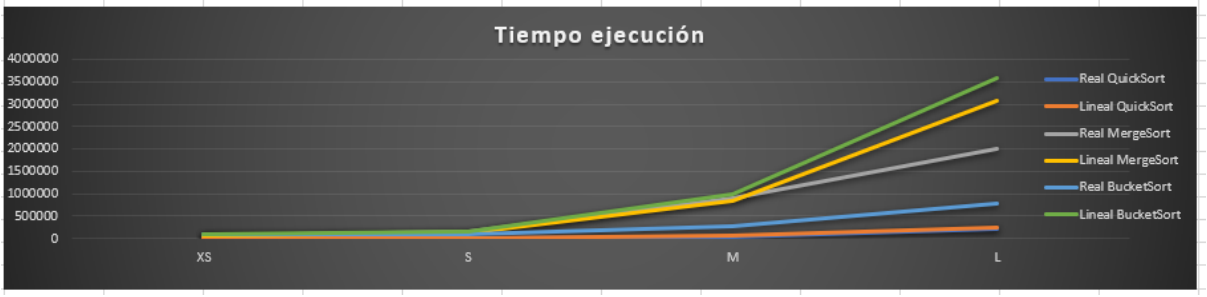
3. Anàlisi general dels tres algorismes d'ordenació

Una vegada programats els algorismes i una vegada entenem la finalitat, els costos i els pros i contras de cadascun d'ells, podem analitzar els 3 juntament amb una opinió més detallada i profunda.

En primer lloc començarem analitzant una gràfica que hem realitzat a partir de recopilar tots els temps dels 3 algorismes per separat, fent 10 opertures de cadascun del fichers i apuntant els valors manualment en Excel per tal de poder fer la posterior gràfica. Una vegada recopilats tots els temps la taula queda tal que així:

	QuickSort				MergeSort				BucketSort			
	XS	S	M	L	XS	S	M	L	XS	S	M	L
Ejecución 1	4800	10700	42400	185200	75200	143600	790700	1967900	85400	115400	221700	846300
Ejecución 2	4400	7500	43800	263900	66400	205200	781800	1951500	66400	90400	260100	884900
Ejecución 3	6500	7100	45000	283300	69400	180400	958600	2145900	66500	85400	230400	958800
Ejecución 4	4200	7700	40000	272300	70600	130400	865700	2018000	107200	92600	226900	698500
Ejecución 5	6200	7000	42400	263800	68400	125600	783800	1901000	95400	91400	255400	776000
Ejecución 6	6400	7300	60900	184300	71100	126600	917600	1971100	88100	92700	228800	732900
Ejecución 7	6300	7000	41300	188900	69000	129700	907100	2291600	65500	85000	365800	768100
Ejecución 8	6300	10900	40700	190400	67200	130800	1050700	1881900	84200	63200	226200	775600
Ejecución 9	6400	7300	41100	185500	75200	186400	906800	2066400	79500	109200	369900	724200
Ejecución 10	4600	6900	41300	182300	65000	187400	942400	1889900	75100	85500	221700	744500
Real:	5610	7940	43890	219990	69750	154610	890520	2008520	81330	91080	260690	790980
Lineal:	5.610	11.220	67.320	246840	69750	139500	837000	3069000	81330	162660	975960	3578520

Cal destacar que el cost Real és el cost Real és el cost calculat a partir de la mitjana aritmètica dels temps de la columna correlativa, i el cost lineal és el temps calculat a partir de multiplicar el temps de la mitjana aritmètica del primer ficher x2, x12 i x44 respectivament. Això es deu a que el ficher XS té 15 vaixells, el ficher S 30 (x2 del primer), el ficher M 180 (x12 del primer) i el ficher L 660 (x44 del primer). Aquest segon calcul és de manera teòrica per analitzar a banda dels temps dels algorismes entre ells, els temps dels mateixos amb el cost Lineal per a saber si el seu temps és lineal o no.
Una vegada gràficats els valors queden de la següent manera:



A partir d'aquest gràfic podem veure i concluir el següent:

En primer lloc, el BucketSort és un algorisme d'ordenament que es basa en la distribució dels elements en diferents "cubs", amb l'objectiu de després ordenar-los de manera individual dins de cada cub. Aquest algorisme té una complexitat temporal d'O(n + k), on n és el nombre d'elements a l'arranjament i k és el nombre de cubs. Encara que aquesta complexitat és similar a la de l'algoritme d'ordenament de comptatge, el BucketSort

requereix un ús més gran de memòria a causa de la necessitat de crear i emmagatzemar els cubs.

El MergeSort, d'altra banda, és un algorisme d'ordenament "divideix i venceràs" que es divideix en subllistes fins que cadascuna només té un element, i després es combinen de manera ordenada. Aquest algorisme té una complexitat temporal d'O(n log n) i una complexitat espacial d'O(n), cosa que significa que requereix un ús moderat de memòria. Encara que és menys eficient en termes de temps d'execució que el BucketSort, el MergeSort és més estable i no requereix la creació d'estructures addicionals com els cubs del BucketSort.

Finalment, QuickSort és un algorisme d'ordenament "divideix i conquesta" que es basa en la selecció d'un "pivot" per dividir l'arranjament i ordenar els elements al seu voltant. Aquest algorisme té una complexitat temporal d'O(n log n) en el millor i pitjor cas, i una complexitat espacial d'O(log n) en el millor cas i O(n) en el pitjor cas. Encara que pot ser menys estable que el MergeSort, QuickSort és l'algorisme més eficient en termes de temps d'execució a causa de la seva eficiència en la selecció del pivot i la divisió de l'arranjament.

Cadascun d'aquests algoritmes d'ordenament té els seus propis avantatges i desavantatges quant a temps d'execució i ús de memòria. El BucketSort és el que més triga a causa de la complexitat de la creació i emmagatzematge dels cubs, el MergeSort és el segon que més triga a causa de la complexitat de la divisió i combinació de les subllistes, i el QuickSort és el que menys triga degut eficiència en la selecció del pivot i la divisió de l'arranjament. Cadascú d'aquests algorismes es pot utilitzar en diferents situacions i contextos, i l'elecció de l'algorisme adequat dependrà de les necessitats específiques del problema que cal resoldre.
D'altra banda, també hem comptat el número d'iteracions de cada algoritme d'ordenació i els hem situat en una taula d'excel:

	QuickSort			
	XS	S	M	L
Numero iteracions	9	19	112	443
	MergeSort			
	XS	S	M	L
Numero iteracions	29	59	359	1319
	BucketSort			
	XS	S	M	L
Numero iteracions	17 + k	18 + k	22 + k	23 + k

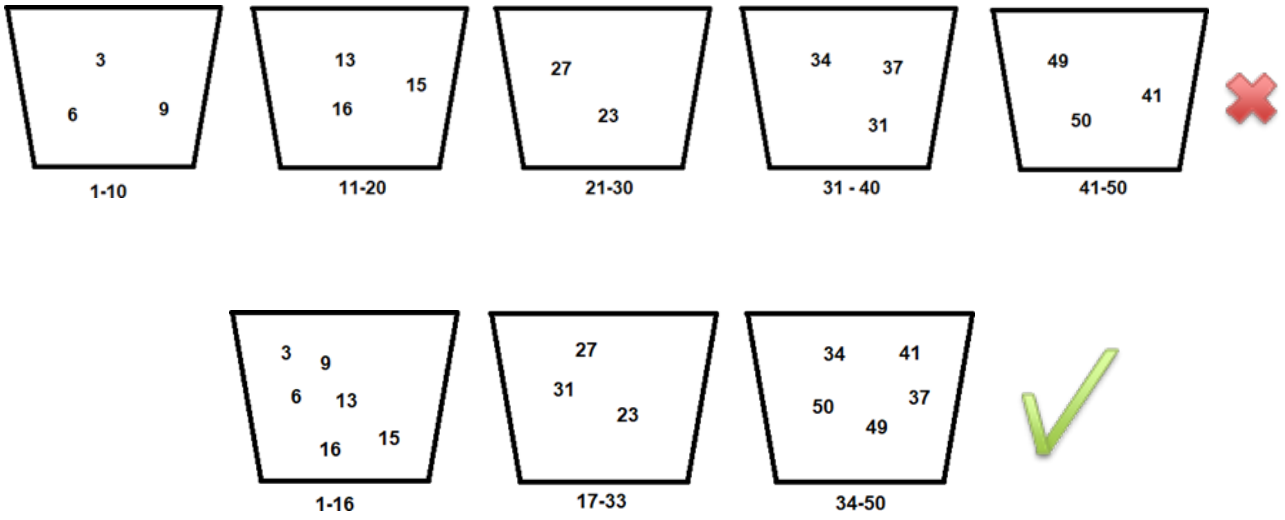
Amb aquesta taula també podem justificar el temps que triguen i perquè al gràfic anterior el QuickSort era el que menys trigaba, després el MergeSort i finalment el BucketSort. Això es deu al número de vegades que es crida el mateix algorisme recursivament, ja que contra més vegades un algorisme es crida a si mateix, més temps triga en resoldre el problema
Per a més precisió, hem utilitzat la funció System.nanoMillis() ja implementada a Java ja que d'aquesta manera tots els gràfics quedaven representats amb moltíssima més precisió. Ja que a l'informe que vem entregar sobre 10, al utilitzar la funció System.millis() moltes vegades el temps donava 0, i això representat en una gràfica no és que sigui molt precís. Es per això quue posteriorment a investigar vem trobar aquesta funció que ofereix moltíssima més precisió.

4. Problemes observats

Hem trobat alguns problemes durant la implementació dels algoritmes. Això era degut a la utilització d'un algoritme que feia moltes iteracions i provocava que la memòria del IDE s'acabés.

Ho hem solucionat utilitzant, per exemple en el bucket sort, menys cubs. En comptes d'utilitzar 100 cubs, hem fet servir 10. Això fa que hi hagi més elements dins de cada cub, però que s'hagin d'ordenar posteriorment menys quantitat de cubs.

Per exemple:



Gracies a aquesta solució no només l'algoritme ja funcionava sinó que a més a més anava més ràpid.

Un altre problema que ens a sorprès era a l'hora d'implementar el bucket sort vam acabar fent un counting sort ja que vam entendre malament a l'hora de buscar informació sobre l'algoritme. Vam poder resoldre-ho preguntant al professor.

Hi van haver molts més problemes però eren errors al programar i la majoria no eren importants. Per exemple a l'hora de manejar ArraysLists o utilitzar funcions especifiques d'alguna llibreria de java.

5. Conclusions

En conclusió, els algoritmes d'ordenació BucketSort, MergeSort i QuickSort són tècniques valuoses que tenen usos i aplicacions diferents en funció de les necessitats específiques d'un problema.

El BucketSort és un algorisme que es basa en la distribució dels elements en diferents cubs o lleixes, cosa que el fa adequat per treballar amb conjunts de dades amb una distribució coneguda o limitada.

El MergeSort és un algorisme "divideix i venceràs" que es divideix en subllistes fins que cadascuna només té un element, i després es combinen de manera ordenada, és un algorisme estable i precís.

El QuickSort és un algorisme "divideix i conquesta" que es basa en la selecció d'un pivot per dividir l'arranjament i ordenar els elements al seu voltant, és un algorisme eficient en termes de temps d'execució.

En conclusió, la complexitat de cadascun dels tres algorismes d'ordenació és un factor important que cal tenir en compte en triar l'adequat per a un problema específic.

El BucketSort té una complexitat temporal d'O(n+k) i requereix un major ús de memòria a causa de la creació i emmagatzematge dels cubs, però és adequat per a conjunts de dades amb una distribució coneguda o limitada.

El MergeSort té una complexitat temporal d'O(n log n) i una complexitat espacial d'O(n) i és més estable i precís, però és menys eficient en termes de temps d'execució que el BucketSort.

Per acabar, el QuickSort té una complexitat temporal d'O(n log n) i una complexitat espacial d'O(log n) en el millor cas i O(n) en el pitjor cas, és eficient en termes de temps d'execució però menys estable que el MergeSort.

És important considerar cadascuna d'aquestes complexitats i pros i contres de cada algorisme per triar l'adequat per a cada problema.

6. Bibliografia

Where developers learn, share, & build careers. Stack Overflow. (n.d.). Retrieved November 18, 2022, from https://stackoverflow.com/

Upadhyay, S. (2022, March 28). *Bucket sort algorithm: Time complexity & pseudocode: Simplilearn*. Simplilearn.com. Retrieved November 23, 2022, from https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm

Merge sort algorithm. Studytonight.com. (n.d.). Retrieved November 25, 2022, from https://www.studytonight.com/data-structures/merge-sort

Khan Academy. (n.d.). *Analysis of Quicksort (article) | quick sort*. Khan Academy. Retrieved November 26, 2022, from https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Insertion sort. GeeksforGeeks. (2022, October 18). Retrieved November 22, 2022, from https://www.geeksforgeeks.org/insertion-sort/

Counting sort. GeeksforGeeks. (2022, November 17). Retrieved November 22, 2022, from https://www.geeksforgeeks.org/counting-sort/

Dineshpathak, A. (2022, February 19). *Algorithmic complexity*. Devopedia. Retrieved November 19, 2022, from https://devopedia.org/algorithmic-complexity