Grupo Backslash:

- Ricardo Bernal
- Diego Bulla

Tarea 2: dividir y conquistar

1. Implemente en el lenguaje de su preferencia tres de los ejercicios 2.15 al 2.18 del libro de Dasgupta. Demuestre en su informe que sus algoritmos tienen la complejidad solicitada.
    - Ejercicio 2.16 - BinSearchInfinity.java: Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in O(log n) time.

      Se utiliza búsqueda binaria para dividir el arreglo en dos mitades en cada recursión. Como se trata de un llamado recursivo (ver Ejercicio 2.17), esto se realiza en tiempo log n por lo que su complejidad temporal es O(log n).

```
private static int binSearchInfinite(int[] arr, int i) {
    int pivote = (int)(arr.length/2);

    if (arr[pivote] == i) {
        return arr[pivote];
    }else if (arr[pivote] < i) {
        return binSearchInfinite(Arrays.copyOfRange(arr, pivote+1, arr.length), i);
    }else {
        return binSearchInfinite(Arrays.copyOfRange(arr, 0, pivote), i);
    }
```

    - Ejercicio 2.15 - SplitAlgorithm.java: Show how to implement this split operation in place, that is, without allocating new memory.

      La complejidad espacial en este algoritmo es O (1) porque solo se utiliza una variable extra (la variable swap) para realizar el intercambio entre los valores del arreglo en cada uno de los loops (for).

```
private static int[] split(int[] arr1, int pos
    // TODO Auto-generated method stub
    int x = 0, n = arr1.length, i;

    for ( i = 0; i < arr1.length; i++) {
        if (arr1[i] < pos) {
            int swap = arr1[i];
            arr1[i] = arr1[x];
            arr1[x] = swap;
```

    - Ejercicio 2.17 - BinSearchAlgo217.java: Given a sorted array of distinct integers A[1, . . . , n], you want to find out whether there is an index i for which A[i] = i. Give a divide-and-conquer algorithm that runs in time O (log n).

      La complejidad temporal se realiza en O (log n) ya que en cada recursión se divide la secuencia n a la mitad por lo que se tiene n/2, n/4, n/8, … hasta que solo quedamos con

1 elemento para lo cual $\log_2 2 = 1$. Por lo que para una secuencia de tamaño n, se demora log n.

```
private static int binSearch(int[] arr1, int begin, int end, int buscar) {
    // TODO Auto-generated method stub
    int mid = (begin + end) / 2;
    if (end < begin) {
        return -1;
    }
    if (arr1[mid] == buscar) {
        return mid;
    }
    if (arr1[mid] > buscar) {
        return binSearch(arr1, begin, mid-1, buscar);
    }else if(arr1[mid] < buscar) {
        return binSearch(arr1, mid+1, end, buscar);
    }
    return -1;
}
```

2. Implemente los algoritmos burbuja, mergesort y quicksort. Para todos los algoritmos, estudie sus tiempos de ejecución para instancias de tamaños variados de la manera que lo hicimos en la tarea anterior. Incluya en su informe la solución del problema 2.24 del libro de Dasgupta.

   Los tiempos de ejecución se midieron con tamaños de instancias que son potencia de 10 (10,100, 1000, …) y en la medición del tiempo de cada instancia, se registraron 5 tiempos con los cuales se obtiene un tiempo promedio del tiempo de ejecución por cada instancia en los distintos algoritmos de ordenamiento (Ver Anexo Excel). A continuación, se muestra algunos de los tiempos registrados:

   - **BubbleSort**

     Complejidad temporal: En cada iteración, se realiza n-1 comparaciones (siendo n el número de elementos de la secuencia) por lo que tenemos:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \ldots + 1 \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

     Complejidad espacial: Como se trata de un algoritmo in situ, no hace uso de espacio auxiliar considerable por lo que su complejidad es O (1).

     i. **Instancia tamaño 100**

```
<terminated> BubbleSort_V1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw
Tamaño de la instancia: 100
Secuencia Original: [356, 754, 478, 758, 811, 890, 769, 307, 462, 14,
Secuencia Ordenada: [8, 14, 21, 29, 37, 52, 54, 54, 60, 84, 106, 115,
Tiempo de ejecucion: 1344900.0 ns
```

     ii. **Instancia tamaño 1000**

```
<terminated> BubbleSort_V1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 1000
Secuencia Original: [789, 657, 366, 358, 474, 542, 401, 692, 302, 664, 78
Secuencia Ordenada: [0, 1, 1, 2, 2, 3, 3, 5, 5, 6, 8, 10, 11, 11, 11, 15,
Tiempo de ejecucion: 1.47057E7 ns
```

iii. Instancia tamaño 10000

```
<terminated> BubbleSort_V1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 10000


Tiempo de ejecucion: 3.147945E8 ns
```

- **MergeSort**

Complejidad temporal: Por teorema maestro, tenemos que la complejidad temporal para mergeSort es:

$T(n) = 2T\left(\frac{2}{n}\right) + O(n)$ donde $2T\left(\frac{2}{n}\right)$ corresponde al llamado recursivo para los dos sub-arreglos y *O(n)* para merge.

$$d = log_2 2 = 1$$

$$O(n \log n)$$

Complejidad espacial: Como se trata de un algoritmo no *in situ,* estamos creado secuencias o arreglos temporales en cada llamado recursivo (una secuencia para la parte izquierda 'l' y otro para la derecha 'r') por lo que sería $S(n) = l+r$ => $O(l+r) = O(max(l,r)) = O(n)$.

i. Instancia tamaño 100

```
<terminated> MergeSort_V2 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (
Tamaño de la instancia: 100
Secuencia Original: [405, 972, 825, 461, 857, 109, 886, 830, 925, 670, 375
Secuencia Ordenada: [1, 2, 3, 23, 34, 60, 90, 91, 109, 126, 164, 169, 192,
Tiempo de ejecucion: 224000.0 ns
```

ii. Instancia tamaño 1000

```
<terminated> MergeSort_V2 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 1000
Secuencia Original: [786, 435, 553, 271, 695, 765, 390, 862, 133, 831, 38
Secuencia Ordenada: [1, 1, 2, 3, 3, 5, 8, 8, 9, 9, 9, 12, 12, 13, 13, 14,
Tiempo de ejecucion: 1608100.0 ns
```

iii. Instancia tamaño 10000

```
<terminated> MergeSort_V2 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 10000


Tiempo de ejecucion: 1.15556E7 ns
```

- **Quicksort**

  Complejidad temporal: Si el pivote resulta estar ubicado cerca del principio o final del arreglo, podemos considerar:

$$T(n) = T(n-1) + O(n)$$

$$= O(n^2)$$

  El mejor caso depende si el pivote es ubicado en la mitad de la secuencia o cerca de esta. Si es asi, su complejidad seria:

  $T(n) = 2T\left(\frac{2}{n}\right) + O(n)$ donde $2T\left(\frac{2}{n}\right)$ corresponde al llamado recursivo para los dos sub-arreglos y *O(n)* para partition.

$$d = log_2 2 = 1$$

$$O(n \log n)$$

Complejidad espacial: Debido a la llamada recursiva que realiza el algoritmo sobre sí mismo, el espacio requerido para ejecutarse es $\mathrm{S(n) = O\ (log\ n)}$.

i. Instancia tamaño 100

```
<terminated> QuickSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 100
Secuencia Original: [36, 30, 86, 32, 89, 14, 22, 5, 17, 77, 69, 30, 90
Secuencia Ordenada: [1, 2, 2, 2, 3, 5, 5, 8, 10, 10, 12, 14, 16, 16, 1
Tiempo de ejecucion: 87900.0 ns
```

ii. Instancia tamaño 1000

```
<terminated> QuickSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 1000
Secuencia Original: [50, 95, 83, 27, 48, 98, 45, 48, 44, 45, 8, 54, 64
Secuencia Ordenada: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1
Tiempo de ejecucion: 1050600.0 ns
```

iii. Instancia tamaño 10000

```
<terminated> QuickSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (
Tamaño de la instancia: 10000


Tiempo de ejecucion: 1.46094E7 ns
```

3. Implemente dos de los algoritmos de ordenamiento en tiempo O(n) vistos en clase. Mida su tiempo de ejecución con las mismas instancias usadas en el punto anterior. (Note que las instancias de ejemplo deben servir para todos los algoritmos).

- **CountingSort**

  Complejidad temporal: con una complejidad temporal de O(n+k), siendo **n** la cantidad de datos a ordenar y **k** el tamaño del vector auxiliar (máximo - mínimo).

  Complejidad espacial: Como se trata de un algoritmo in situ, el espacio utilizado es S(n) = O(k+1) n donde *n es el número de elementos y k es el rango o espacio auxiliar utilizado*.

  i. Instancia tamaño 100

```
<terminated> CountingSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (
CountingSort
Tamaño de la instancia: 100
Tiempo de ejecucion: 48600.0 ns
```

  ii. Instancia tamaño 1000

```
<terminated> CountingSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
CountingSort
Tamaño de la instancia: 1000
Tiempo de ejecucion: 237000.0 ns
```

  iii. Instancia tamaño 10000

```
<terminated> CountingSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (
CountingSort
Tamaño de la instancia: 10000
Tiempo de ejecucion: 2570700.0 ns
```

- **BucketSort**

  Complejidad temporal: Debido a que su complejidad temporal depende el algoritmo de ordenamiento utilizado sobre cada casillero, se puede decir que su complejidad es $O(n^2)$ la cual es dada por el algoritmo Insertionb Sort para el peor de los casos (un arreglo ordenado al revés).

  Complejidad espacial: Como se trata de un algoritmo in situ, el espacio utilizado es S(n) =O(n+k) donde *k* corresponde al número de casilleros y *n* es el número de elementos.

  i. Instancia tamaño 100

```
<terminated> BucketSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 100
Tiempo de ejecucion: 2980700.0 ns
```

ii. Instancia tamaño 1000

```
<terminated> BucketSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (
Tamaño de la instancia: 1000
Tiempo de ejecucion: 1.02187E7 ns
```

iii. Instancia tamaño 10000

```
<terminated> BucketSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 10000
Tiempo de ejecucion: 3.08423E7 ns
```

4. Para un tamaño de entrada grande, prediga el tiempo de ejecución de los cinco algoritmos. Mida y registre.

   Como se describe en el numeral dos se registraron distintos tiempos para cada instancia de tamaño diferente (ver Anexo Excel) y estos tiempos se promediaron. Con estos tiempos, se realizó una gráfica en la cual realizamos un análisis de regresión y así poder obtener la ecuación de la gráfica para poder determinar el valor del tiempo para una instancia de tamaño mas grande (100000 para estos casos)

   - **BubbleSort**
     - **i. Tiempo Experimental:**

```
<terminated> BubbleSort_V1 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
Tamaño de la instancia: 100000
Tiempo de ejecucion: 2.88280108E10 ns
Tiempo de ejecucion: 28.8280108 s
```

     ii. **Tiempo Teórico (Predicción):** t (100000) = 2899500000 con un error del 89,94%

   - MergeSort
     i. **Tiempo Experimental:**

```
<terminated> MergeSort_V2 [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
MergeSort
Tamaño de la instancia: 100000
Tiempo de ejecucion: 8.36392E7 ns
Tiempo de ejecucion: 0.0836392 s
```

     ii. **Tiempo teorico(prediccion) :** t(1000000) = 114957445 con un error del 37,44%

   - Quicksort

i. **Tiempo Experimental:**

```
<terminated> QuickSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe
QuickSort
Tamaño de la instancia: 100000
Tiempo de ejecucion: 1.2636E8 ns
Tiempo de ejecucion: 0.12636 s
```

ii. **Tiempo teorico(prediccion)**: t(100000) = 138698206 con un error 9.76

- CountingSort

i. **Tiempo Experimental:**

```
<terminated> CountingSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (2
CountingSort
Tamaño de la instancia: 100000
Tiempo de ejecucion: 1.55206E7 ns
```

ii. **Tiempo teorico(prediccion)**: t(100000) = 20838806 con un error 34,27%

- BucketSort

i. **Tiempo Experimental:**

```
<terminated> BucketSort [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (28/09/2020 09:06:28 PM – 09:06:31 PM)
Tamaño de la instancia: 100000
Tiempo de ejecucion: 2.016171E8 ns
```

Ii. **Tiempo teorico(prediccion)**: t(100000) = 284970000 con un error del 41,34%

5. Conclusiones.
   - Para instancias de menor tamaño, podemos concluir que el algoritmo Quicksort es el que presenta la mayor eficiencia en su tiempo de ejecución debido a que su orden de complejidad temporal es O (n log n). Aunque Counting Sort presente un orden de complejidad O(n), gasta mayor tiempo en la definición del espacio auxiliar necesario para su ejecución mientras que Quicksort evade esto de forma recursiva (algoritmo no in situ).
   - Por otro lado, en instancias de mayor tamaño los algoritmos que presentan mayor eficiencia son aquellos algoritmos que utilizan o requieren de un espacio auxiliar también como algoritmos in situ (in place). Esto permite que los accesos en memoria sean más rápidos al estar ya definidos y, además, la mayoría de estos algoritmos se ejecutan de forma lineal como es el caso de BucketSort y CountingSort.
   - BubbleSort (Burbuja) es el algoritmo que presente el peor desempeño o eficiencia tanto para instancias de tamaño pequeño como las de tamaño grande ya que su orden de complejidad es mayor que el todo los demás, siendo la cota más alta en este caso.