UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PROYECTO #1 – COMPLEJIDAD COMPUTACIONAL EN LOS ALGORITMOS DE ORDENAMIENTO

CARRERA: INGENIERÍA EN COMPUTACIÓN

ESTRUCTURAS DE DATOS Y ALGORITMOS II

GRUPO: 07

PROFESOR: EDGAR TISTA GARCÍA

EQUIPO: 4

INTEGRANTES:

- HERNÁNDEZ GALLARDO DANIEL ALONSO
- JIMÉNEZ ORTIZ SEBASTIÁN
- LÓPEZ FLORES ISAAC

SEMESTRE 2024-1

FECHA DE ENTREGA: 17 DE SEPTIEMBRE DEL 2023

Contenido

Objetivo: Que el alumno observe la complejidad computacional de los algoritmos de ordenamiento para comparar su eficiencia de ejecución en grandes volúmenes de información.

Introducción:

Los algoritmos de ordenamiento son muy importantes en la ciencia de la computación y en la programación, ya que en la vida real la mayoría de las veces la información que necesitamos analizar está desordenada, y para poder trabajar con ella debemos ordenarla y facilitar la búsqueda con algoritmos como la binaria, pues requiere que la colección a manipular se encuentre ordenada.

Hay dos aspectos fundamentales al analizar la eficiencia de un algoritmo: tiempo y recursos computacionales (memoria y procesamiento). En este proyecto realizamos una comparativa (algo burda e inocente) entre los algoritmos de ordenamiento que vimos en clase, utilizando como punto de comparación el número de operaciones que realiza cada uno.

Nuestras expectativas al iniciar con el proyecto eran que no iba a ser complicado completarlo, después de todo ya teníamos la implementación de la mayoría de los algoritmos, solo restaba implementar los faltantes y elegir como contaríamos sus operaciones.

Una vez que empezamos a meternos de lleno, nos dimos cuenta de que la tarea iba a ser un poco más complicada de lo anticipado. Las muchas variantes del algoritmo *Quick Sort* (decidir cómo se elige el pivote) nos quitaron más tiempo del esperado. El otro aspecto que nos complicó el inicio fue decidir como manejaríamos el conteo de operaciones para analizar la complejidad.

Lo anterior se debe a que existen varias metodologías para analizar la complejidad de un algoritmo, dado que en el proyecto se define que vamos a contar las operaciones, eso nos deja con dos metodologías viables a escoger para este proyecto: *operation count* y *step count*.

Operation count (conteo de operaciones) se basa en decidir que operaciones contribuyen más al tiempo/complejidad del algoritmo. Por otro lado, *Step count* (conteo de pasos) consiste en contar todos los pasos o instrucciones presentes en el algoritmo.

Para los algoritmos de ordenamiento las principales operaciones son comparación, intercambio, intercalación e inserción. Así que podríamos haber decidido solo contar esas en cada algoritmo. Sin embargo, algunos algoritmos dependen de otras operaciones como el *Counting Sort* o el *Radix Sort*. Esta es la causa por la que consideramos que sería más sencillo seguir la metodología *step count*.

Desarrollo:

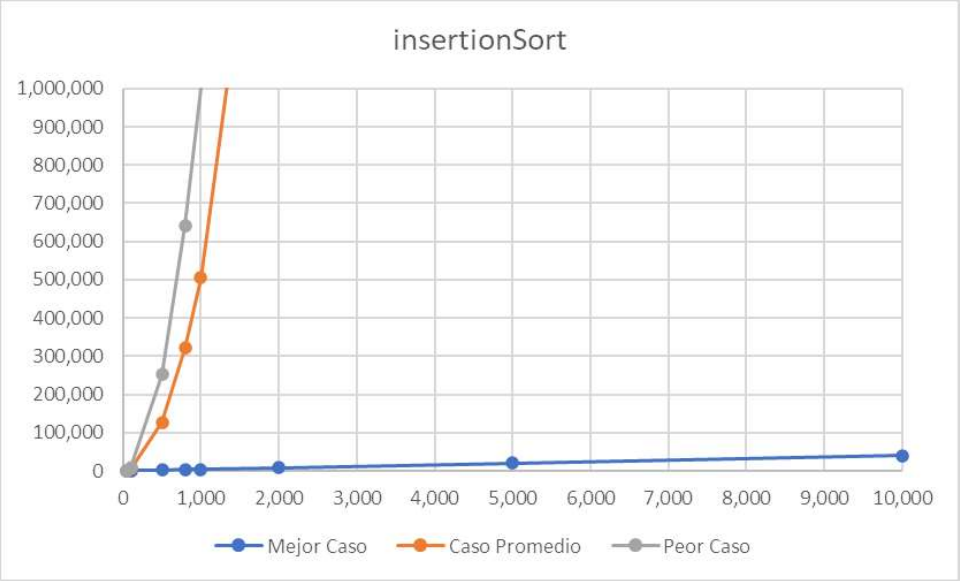
A cada algoritmo de ordenamiento se le agregó una variable de tipo entero llamada *operations* para contabilizar el número de operaciones, la cual a su vez fue inicializada con el valor cero para que posteriormente regresar el valor. Esto implico también modificar todos los algoritmos que originalmente se tenían.

```java
public static int selectionSort(int[] arr){
    int operations = 0;
    int n = arr.length;
    operations = operations+2;
    for (int i = 0; i < n - 1; i++){
        int min = i;
        for (int j = i + 1; j < n; j++){
            if (arr[j] < arr[min]){
                min = j;
                operations = operations+1;
            }
            operations = operations+1;
        }
        Utilerias.swap(arr, i,min);
        operations = operations+5;
    }
    return operations;
}
```

Equipo 4. (2023). *Ejemplo del conteo de operaciones en el algoritmo Selection Sort* [PNG]. Herramienta de recortes.

Insertion Sort:

El algoritmo de I*nsertion Sort* no sufrió grandes cambios, puesto que la implementación se dejó tal y como nos fue proporcionada.

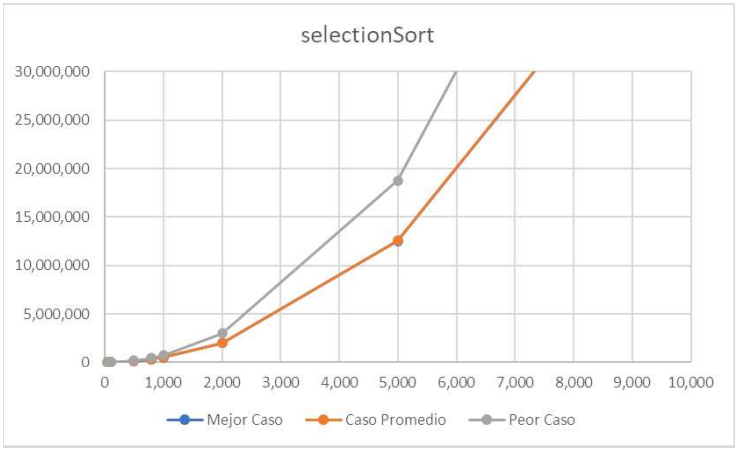


Equipo 4. (2023). *Gráfica de la complejidad temporal de Insertion Sort* [PNG]. Excel.

En esta grafica se nota como las complejidades del I*nsertion Sort* van cambiando según el caso que haya sido escogido, tomando en cuenta el mejor caso como el arreglo ya ordenado, *Insertion Sort* queda con una complejidad lineal, y para el peor caso y el caso promedio *Insertion Sort* termina con una complejidad cuadrática, lo cual coincide con los elementos vistos en la teoría dando veracidad a los datos recopilados.

Selection Sort:

El algoritmo de *Selection Sort* no tuvo cambios considerables, ya que el algoritmo funcionaba de manera óptima con la implementación que nos fue brindada.

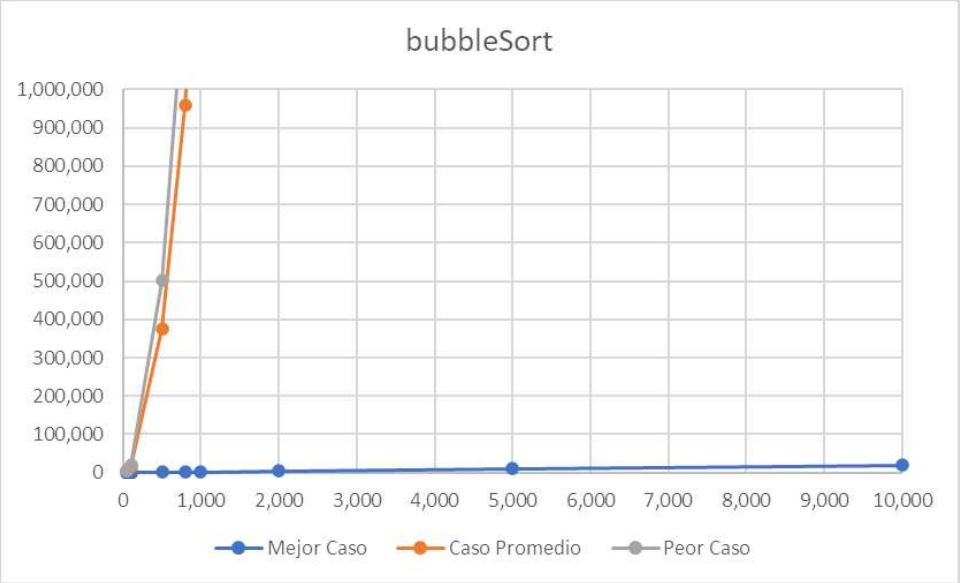


Equipo 4. (2023). *Gráfica de la complejidad temporal de la primera implementación de Selection Sort* [PNG]. Excel.

En base a la gráfica se ve como en todos los casos *Selection Sort* mantiene la complejidad cuadrática e incluso el mejor caso es muy similar al caso promedio, pero sigue siendo el peor caso el que realiza más operaciones. Por ende, los datos coinciden plenamente con las complejidades de este algoritmo, pues la complejidad cuadrática en todos los casos se mantiene.

Bubble Sort:

El algoritmo de *Bubble Sort* es una versión mejorada de este, en la cual su mejor caso termina teniendo una complejidad temporal que es lineal. Sin embargo, al ser una implementación con la que ya contábamos no se le modifico alguna otra instrucción, porque anteriormente ya había sido probada.

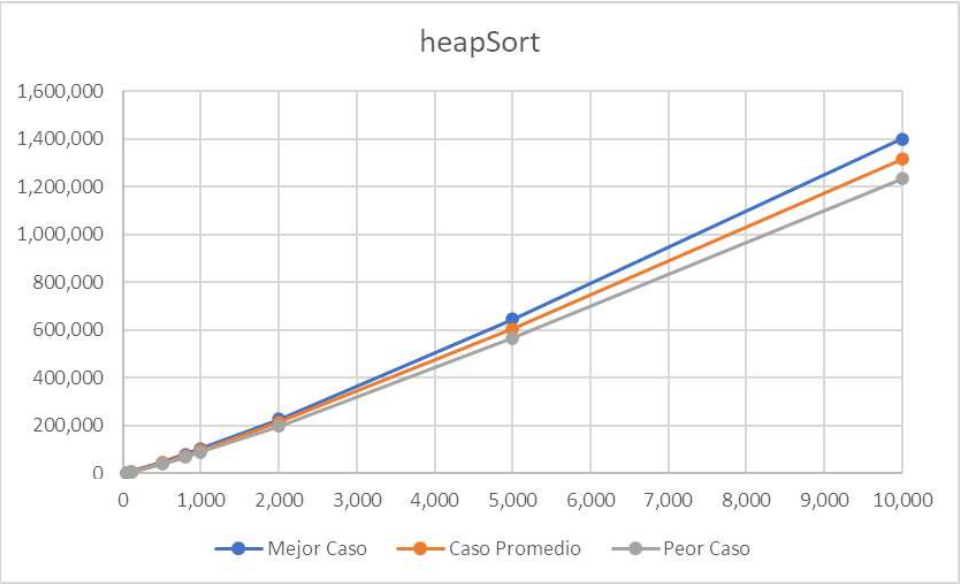


Equipo 4. (2023). *Gráfica de la complejidad temporal de la primera implementación de Bubble Sort* [PNG]. Excel.

El *Bubble Sort* se denota con una complejidad lineal en el mejor caso, y cuadrática en el peor caso y caso promedio, ya que en la gráfica se nota como el caso promedio y peor caso elevan su cantidad de operaciones de una manera muy significativa con respecto al mejor caso, el cual es que la lista ya está ordenada. Por lo anterior, se cumple de manera acertada con los elementos teoricos vistos en clase.

Heap Sort:

Este algoritmo se compone de dos funciones *heapSort* y *heapify*, donde para poder hacer de manera correcta el conteo, agregamos como parámetro de la función *heapify* a la variable operations y al final la función regresa la variable con las sumas que le hizo y este valor se asigna al punto original en el que se llamó la función.



Equipo 4. (2023). *Gráfica de la complejidad temporal de Heap Sort* [PNG]. Excel.

El *Heap Sort* tiene resultados muy similares con respecto a las operaciones realizadas según la cantidad de datos introducidos, dando razón a que en todos los casos su complejidad es la misma O (n log n).

Quick Sort:

El algoritmo de *Quick Sort* que nos fue proporcionado arrojaba grandes resultados en cuanto al número de operaciones realizadas, por lo que, se optó por investigar a qué se debían estos resultados y al mismo tiempo se cambió esta implementación por dos diferentes implementaciones para demostrar que la posición del pivote si afecta enormemente el número de operaciones. De ahí que en la implementación donde el pivote siempre es el último elemento de manera directa no sea la mejor, pues el mejor caso donde la lista ya se encuentra ordenada, termina convirtiéndose en el peor caso. Entonces, en las otras dos implementaciones seleccionadas el pivote se selecciona de maneras particulares.

A su vez, el algoritmo de *Quick sort* tuvo otros problemas relacionados con la construcción del código, dado que en las dos implementaciones nos vimos obligados a buscar alternativas para contabilizar las operaciones. Por ello, utilizamos un arreglo que almacena el *pivot index* y que al mismo tiempo va contabilizando las operaciones realizadas. Gracias a lo anterior, evitamos la mala práctica de usar variables globales para contar todas las operaciones que se llevaban a cabo en los distintos métodos usados por el algoritmo.

Quick Sort (Pivot Median of Three):

En la primera implementación del algoritmo *Quick sort* se sigue una estrategia para escoger el pivote denominada *Median of Three*. La estrategia consiste en escoger el pivote más cercano al valor de la mitad en un arreglo ordenado, pero como no está ordenado, podemos conseguir una aproximación, que se determina tomando el valor de en medio, el primero y el último. Luego se escoge el valor que quede entre dos de los tres valores escogidos, logrando así una optimización en la forma en la que se selecciona el pivote para empezar con el ordenamiento.

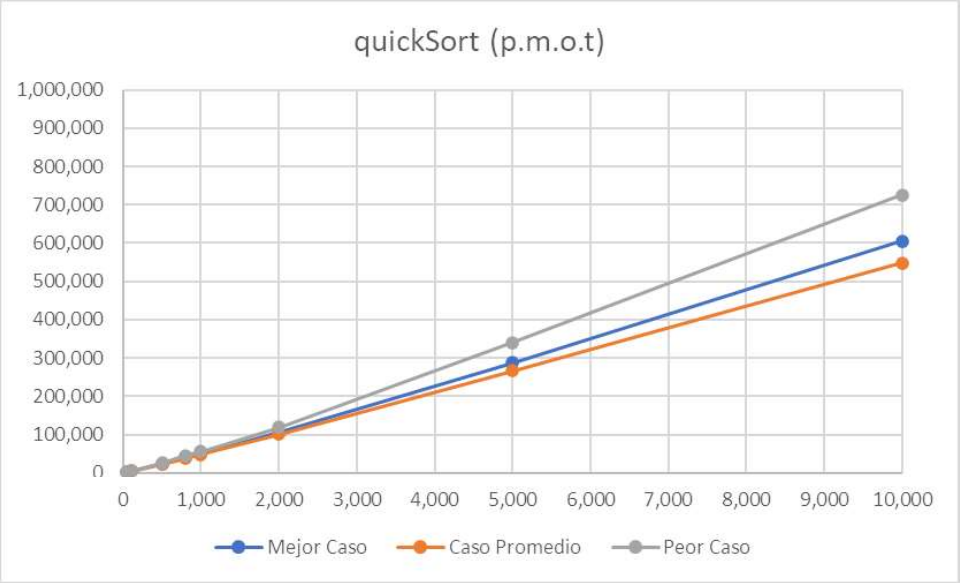
En nuestro código fuente la implementación de dicha estrategia quedo de la siguiente manera:

```
public static int choosePivot(int arr[], int low, int high,  int pi_operations[]) {
    int mid = low + (high - low) / 2;
    int a = arr[low];
    int b = arr[mid];
    int c = arr[high];
    pi_operations[1]+=4;
    if ((a <= b && b <= c) || (c <= b && b <= a)) {
            pi_operations[1]+=4;
        return mid; // Median is b
    } else if ((b <= a && a <= c) || (c <= a && a <= b)) {
            pi_operations[1]+=8;
        return low; // Median is a
    } else {
            pi_operations[1]+=8;
        return high; // Median is c
    }
}
```

Equipo 4. (2023). *Implementación en Java de la estrategia Pivot Median of Three* [PNG]. Herramienta de Recortes.

Como parte de sus instrucciones, realiza comparaciones entre valores con una variable auxiliar y, según estas comparaciones, devuelve el índice ubicado en la mediana de esos valores. Nótese que incluso estas

instrucciones añadidas también son contabilizadas porque se realizan como parte del algoritmo. Lo anteriormente mencionado se ve reflejado en el análisis de complejidad.



Equipo 4. (2023). *Gráfica de la complejidad temporal de la primera implementación de Quick Sort* [PNG]. Excel.

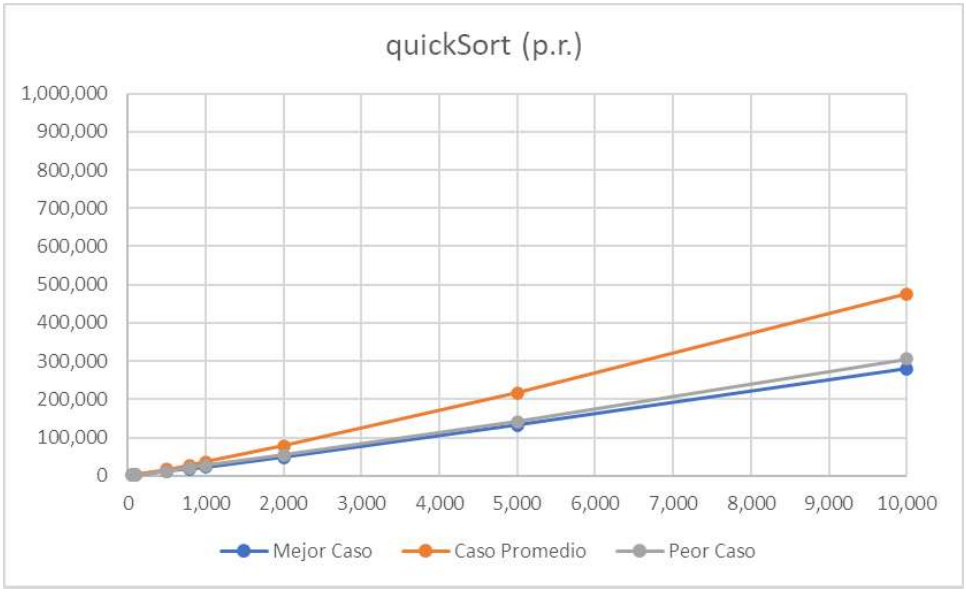
En la gráfica anterior se aprecia como el *Quick Sort* mantiene su complejidad O (n log n) en el caso promedio y en el mejor caso, y se nota como en el peor caso la complejidad se aleja mucho más de lo que se alejan el mejor y el caso promedio entre sí. Lo anterior se debe a la forma en que se toma el pivote, ya que la volatilidad entre el caso promedio y el mejor es notoria, haciendo que incluso en el caso promedio tengan mejores resultados, pues al cambiar los datos se puede escoger un mejor pivote.

Quick Sort (Pivote final aleatorio):

Con esta estrategia el pivote se selecciona de manera aleatoria entre los valores del *high index* y *low index*, para intercambiar el índice aleatorio con el último elemento del arreglo y seguir con la lógica de que el pivote es el último elemento. Sin embargo, con esta modificación se obtiene una mejora enorme en todos los casos, ya que el número de operaciones se ve reducido.

```
int pivotIndex = new Random().nextInt(highIndex - lowIndex) + lowIndex;
int pivot = array[pivotIndex];
Utilerias.swap(array, pivotIndex, highIndex);
```

Equipo 4. (2023). *Implementación en Java de la estrategia Pivote final aleatorio* [PNG]. Herramienta de Recortes.



Equipo 4. (2023). *Gráfica de la complejidad temporal de la segunda implementación de Quick Sort* [PNG]. Excel.

Esta es una gran demostración de como el pivote afecta la complejidad según los datos que se tengan, pues como se guía por la aleatoriedad el caso promedio termina siendo el menos inconsistente, el cual incluso se asemeje a una complejidad cuadrática, pero siendo esto lo único que cambia *Quick Sort* en esta implementación aún mantiene sus complejidades lineales-logarítmicas y cuadrática.

Merge Sort:

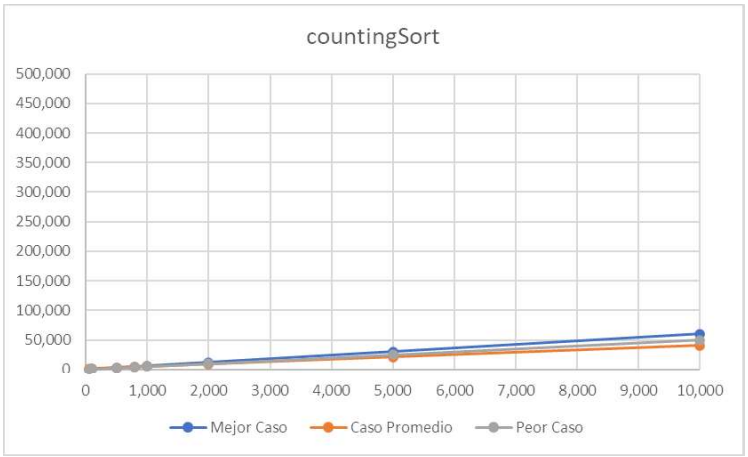
A este algoritmo no se le realiza una modificación en el prospecto de cómo funciona y organiza los elementos, pero si se estableció una manera diferente en la que se contabilizan las operaciones, pues aprovechando la recursividad del algoritmo, cada que termina un proceso se devuelve a si mismo la cantidad de operaciones contabilizadas, y así hasta llegar al caso base que termina por devolver las operaciones al método main para ser recolectadas.

Equipo 4. (2023). *Gráfica de la complejidad de Merge Sort* [PNG]. Excel.

El *Merge Sort* mantiene una gran consistencia en sus casos pues en todos mantiene su complejidad O (n log n), siendo el caso promedio el que se aleja un poco, pero esto sucedería por la variación en los del arreglo datos.

Counting Sort:

En este caso si se realizó una modificación al *Counting Sort,* pues este funciona sin que el usuario este condicionado a un rango o que lo condicione antes de acceder al algoritmo, con las funcionalidades agregadas el algoritmo busca el rango, es decir, encuentra el valor mayor de entre todos los valores haciendo un recorrido lineal y lo fija como el máximo valor encontrado, una vez encontrado procede con el procedimiento común de la realización del counting sort; esta modificación tiene un leve impacto en la complejidad del algoritmo, ya que esta pasa de O(n + k) a O(2n + k), dado que recorre la lista una vez más en busca del elemento máximo para establecer el rango del arreglo de cuentas.



Equipo 4. (2023). *Gráfica de la complejidad de Counting Sort* [PNG]. Excel.

En todos los casos se nota como mantiene su misma complejidad con pequeñas variaciones entre sí. Asimismo, se nota como llega a ser casi lineal, pues solo aumenta un poco siendo esto por el elemento máximo, ya que el mejor y el peor caso siempre contienen el elemento máximo entre todos los rangos de valores. Por ende, sube un poco más la cantidad de operaciones con respecto al caso promedio.

Radix Sort:

En la práctica donde se trabajaba con este algoritmo se manejaba un caso particular en el que los valores de la colección a ordenar estaban delimitados en cierto rango (4 dígitos que solo podían usar los números 1, 2, 3 y 4). Para poder utilizar el algoritmo en este proyecto, tuvimos que adaptarlo para que funcionara con cualquier cantidad de dígitos y sin restricciones en los números a utilizar.

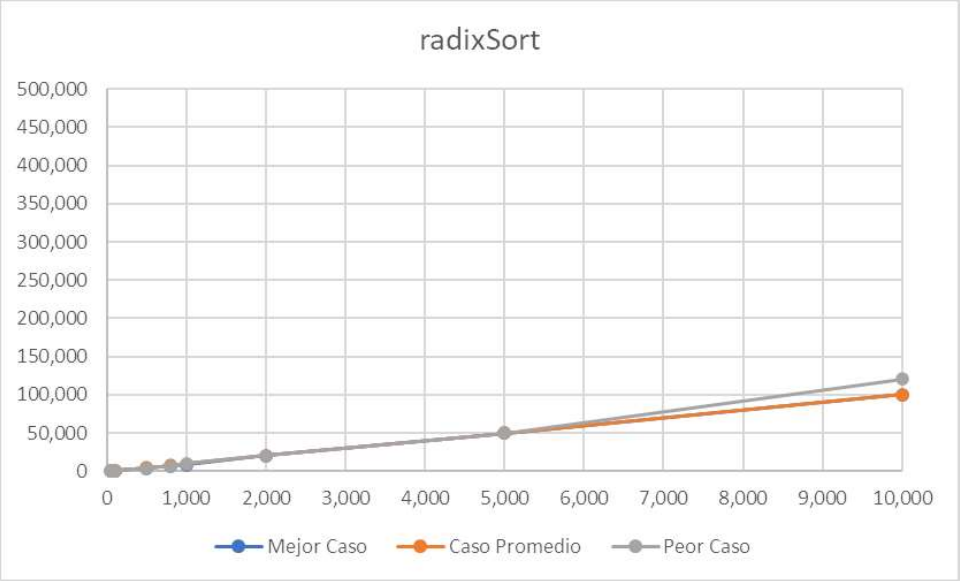
Se creo un arreglo de **colas** de tamaño 10, pues son 10 los posibles dígitos diferentes que puede tener un número. Luego, para poder trabajar con cualquier número se utilizó un ciclo *while* con un ciclo *for* que va recorriendo el arreglo original y para cada elemento se realiza lo siguiente:

- Se divide el numero entre 10^n, siendo n una variable que se inicializa en 0 y que aumenta en uno por cada ciclo del algoritmo. Esto nos permite ir eliminando los dígitos desde la derecha que ya analizamos.
- Posteriormente se obtiene el módulo 10 de este resultado $10^n mod 10$, al hacer esto nos quedamos únicamente con el digito que queremos analizar en ese ciclo.

```
while(cola0Size < arr.length){
    for(int i : arr){
        queues[(i/(int)Math.pow(a:10, n))%10].add(i);
        operations+=1;
    }
```

Equipo 4. (2023). *Implementación en Java de la estrategia Radix Sort* [PNG]. Herramienta de Recortes.

En combinación estas operaciones nos permiten obtener únicamente el valor que queremos analizar en cada ciclo y es este el que utilizamos para decidir en qué cola (*queue*) ingresarlo. El *while* se repite hasta que todos los elementos quedan en la cola 0, lo que implica que ya terminamos de ordenar la colección.

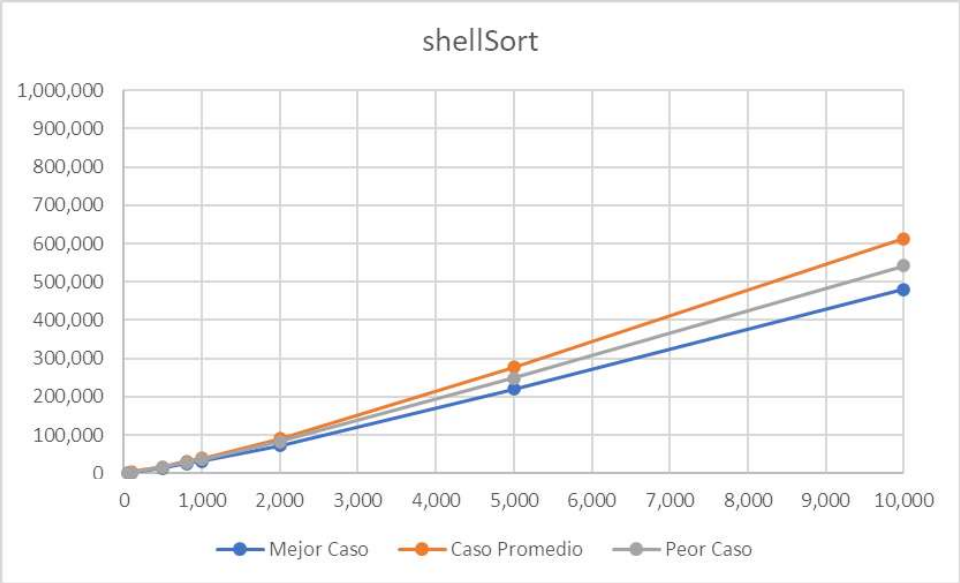


Equipo 4. (2023). *Gráfica de la complejidad de Radix Sort* [PNG]. Excel.

Como se ve *Radix Sort* se acerca mucho a la complejidad lineal, pero aumenta un poco más debido a que según la cantidad de dígitos diferentes serán las veces que se aumentara, denotándose la complejidad como O (n * k), la cual es una complejidad más alta que la de *Counting Sort,* pero sin llegar a ser tan alta como lás demás.

Shell Sort:

Este algoritmo de ordenamiento consiste en ir dividiendo el arreglo en intervalos de varios elementos para organizarlos después por medio de la lógica que sigue el algoritmo de *Insertion Sort*. Este proceso se repite, pero con intervalos cada vez más pequeños ya que se va dividiendo entre 2, de tal manera que al final, el ordenamiento se haga en un intervalo de una sola posición, similar al algoritmo de ordenamiento *Insertion Sort*, donde la diferencia entre ambos es qué, al final, en el algoritmo *Shell Sort* los elementos ya están casi ordenados.



Equipo 4. (2023). *Gráfica de la complejidad de Shell Sort* [PNG]. Excel.

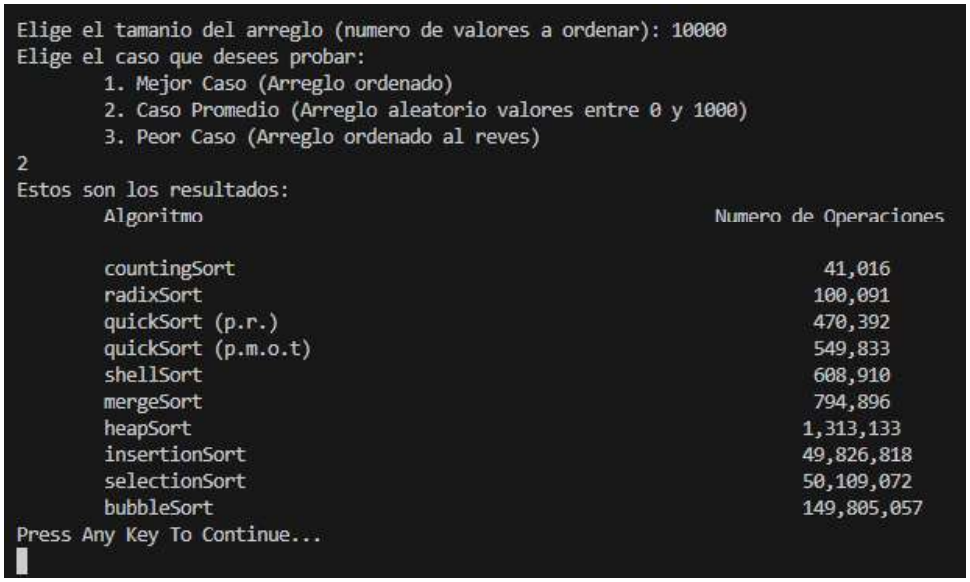
En la gráfica anterior, se aprecia cómo varía la complejidad temporal de este algoritmo, puesto queda a la deriva de cómo se vayan haciendo los intervalos en cada iteración, es decir, los gaps, ya que a veces el peor caso se puede comportar como promedio, el mejor o incluso hasta el peor. Todo depende de la colección que se tenga y como se van construyendo los intervalos. No obstante, la tendencia es O (n log n) para el mejor caso y caso promedio, y O(n^2) para el peor caso.

Recolección de datos e impresión:

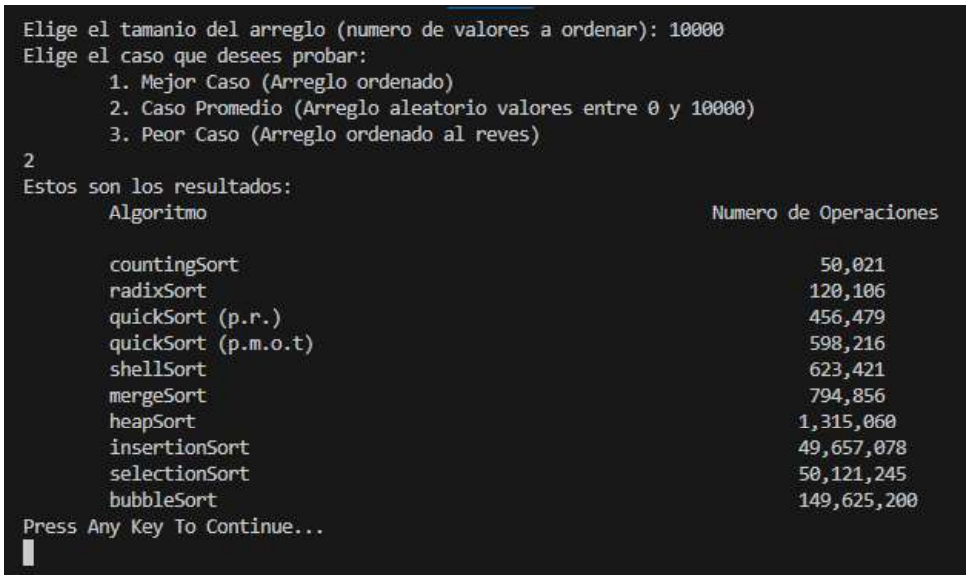
Para la recolección de datos se utilizó un arreglo de enteros con la capacidad de la cantidad de algoritmos de ordenamiento. Estos se van llamando uno a uno y como todos los algoritmos de ordenamiento devuelven la cantidad de operaciones estas se guardan en el arreglo. Por otra parte, para tener una mejor visualización de los datos se decidió ordenar la cantidad de operaciones de la menor a la mayor cantidad de operaciones, y para ello se creó un *treemap* (lo que es un *sorted map* en Java o lo mismo que un *binary search tree*) que ordena las k*eys.* Por ende, en este *treemap* se utilizó como *key* la cantidad de operaciones recolectadas (para que asi el treemap las ordene) y como valor asociado a esa *key* se utilizó el nombre del algoritmo que tiene esa cantidad de operaciones, donde una vez recolectados los datos en el *treemap*, estos se van imprimiendo. Dicho de otra forma, es como si hubiéramos usado un *Heap sort* que tiene como raíz al valor más pequeño y la vamos eliminando sucesivamente para imprimirla, hasta acabar con todos los valores.

Rango de los valores dentro de los arreglos

Para los casos promedios se rellenó el arreglo con números aleatorios entre 0 y 1000, donde no tenemos una razón específica para elegir este rango. Se podría llegar a pensar que, al hacer entradas de 10 000 elementos, el que el rango sea solo de 0 a 1000 generaría muchos valores repetidos que podrían afectar la eficiencia de los algoritmos. Sin embargo, realizamos una prueba para comprobar si este era el caso:



Equipo 4. (2023). *Ejecución del programa con elementos entre 0 y 1000* [PNG]. Programa en Java.

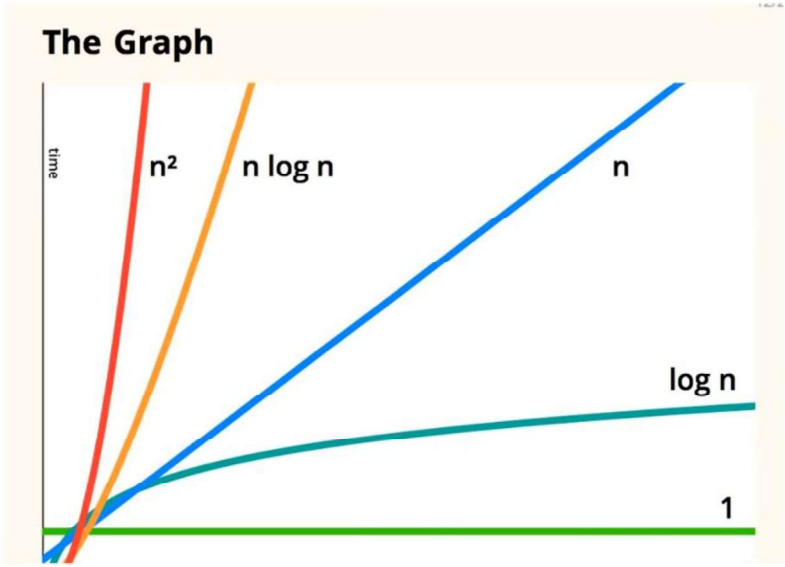


Equipo 4. (2023). *Ejecución del programa con elementos entre 0 y 10 000* [PNG]. Programa en Java

Como se puede observar, la variación en el rango de los elementos dentro de la colección tiene un impacto mínimo en el número de operaciones de los algoritmos, afectando principalmente a los algoritmos *Counting Sort* y *Radix Sort*, esto por la forma en que trabajan (creación de un arreglo en base al elemento más grande y división de los elementos entre 10^n respectivamente).

Comparando la complejidad (operaciones) de los algoritmos

Las expectativas previas a la realización del proyecto eran que los algoritmos se comportaran de la misma forma que habíamos visto en la teoría:

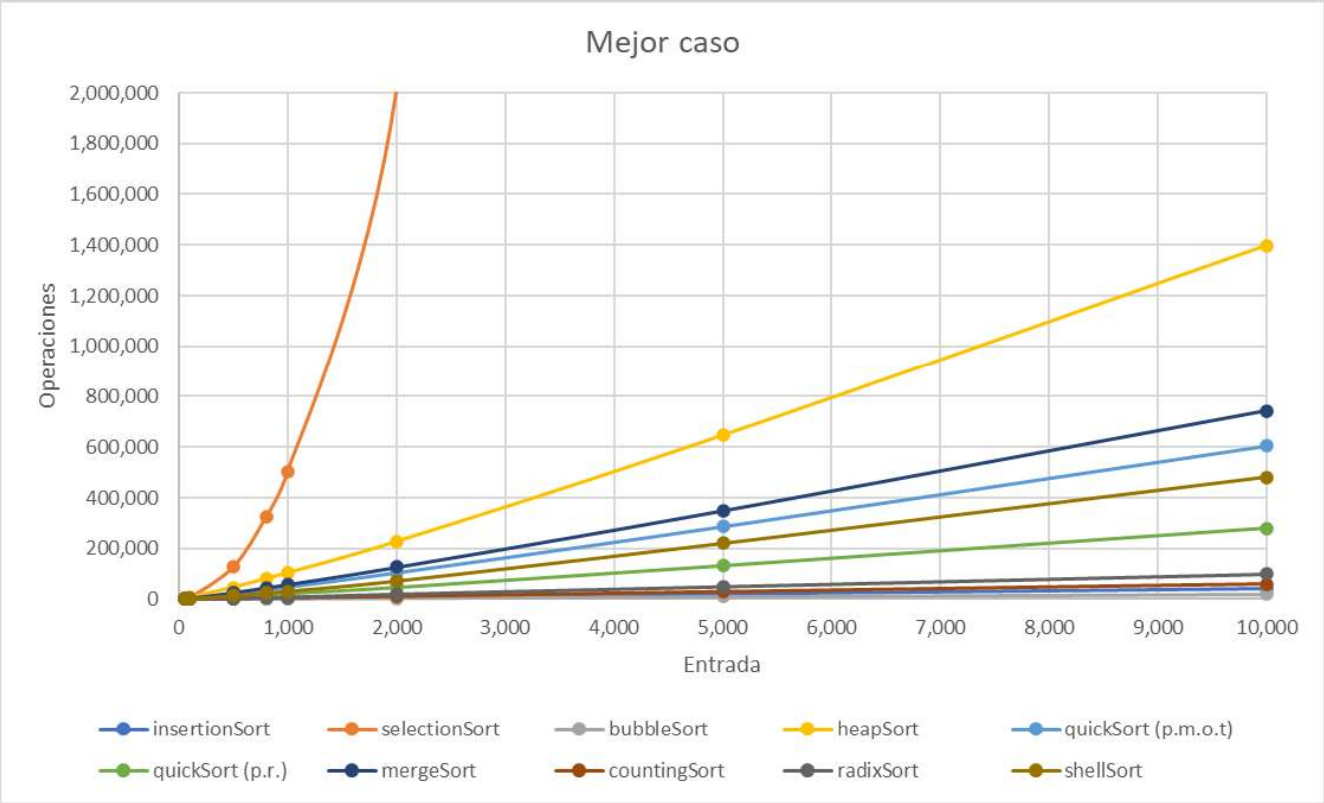


Prof. Edgar Tista. (2020). *Complejidad temporal asintótica* [PNG]. YouTube. https://www.youtube.com/watch?v=hju7wj8EfAg&ab_channel=Prof.EdgarTista

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Select Sort	O(n^2)	O(n^2)	O(n^2)
Counting Sort	O(n+k)	O(n+k)	O(n+k)
Radix Sort	O(nk)	O(nk)	O(nk)

Tista. G. E. (2020). *Tema 1 Algoritmos de ordenamiento* [PNG]. EDUCAFI.

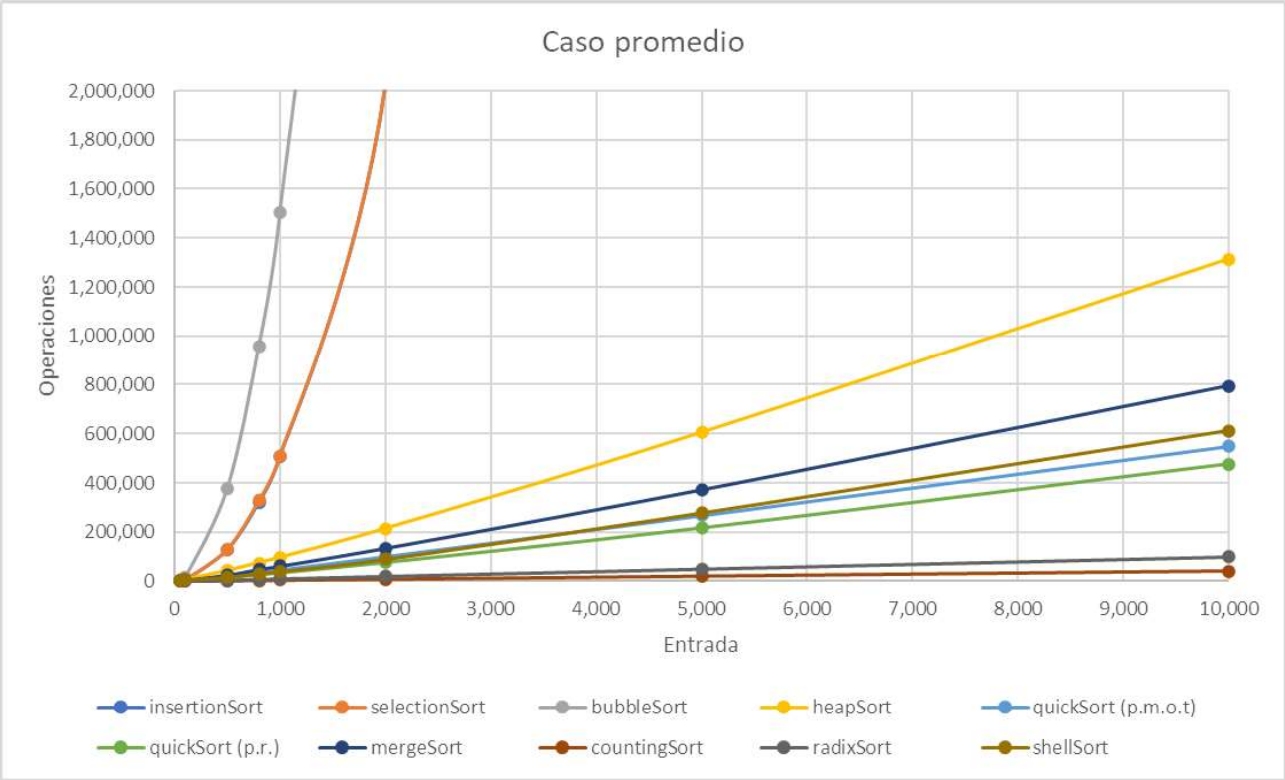
Mejor caso:



Equipo 4. (2023). *Gráfica de la complejidad en el mejor caso para todos los algoritmos estudiados* [PNG]. Excel.

Para el mejor caso se nota de una manera abismal las diferencias de complejidad entre los distintos algoritmos, el *Insertion sort* al ser el único algoritmo de complejidad cuadrática se desplaza rápidamente hacia arriba, mientras le consecuan los algoritmos de complejidad lineal logarítmica que son H*eap Sort*, M*erge Sort*, *Quick sort* y *Shell sort* manteniéndose *Quick Sort* como el más rápido de este grupo, *Radix Sort* y *Counting Sort* se mantienen por encima de *Bubble Sort* e *Insertion Sort*, pues estos si son completamente lineales en su mejor caso. La diferencia es mínima, pues para *Counting Sort* simplemente se suma **k** que corresponde con el número máximo de elementos, y al ser una suma queda por debajo de *Radix Sort*, dado que este es una multiplicación n * k. Por lo anterior, con esta comparativa se demuestra completamente como se mantienen los resultados *a priori* con los resultados *a posteriori*.

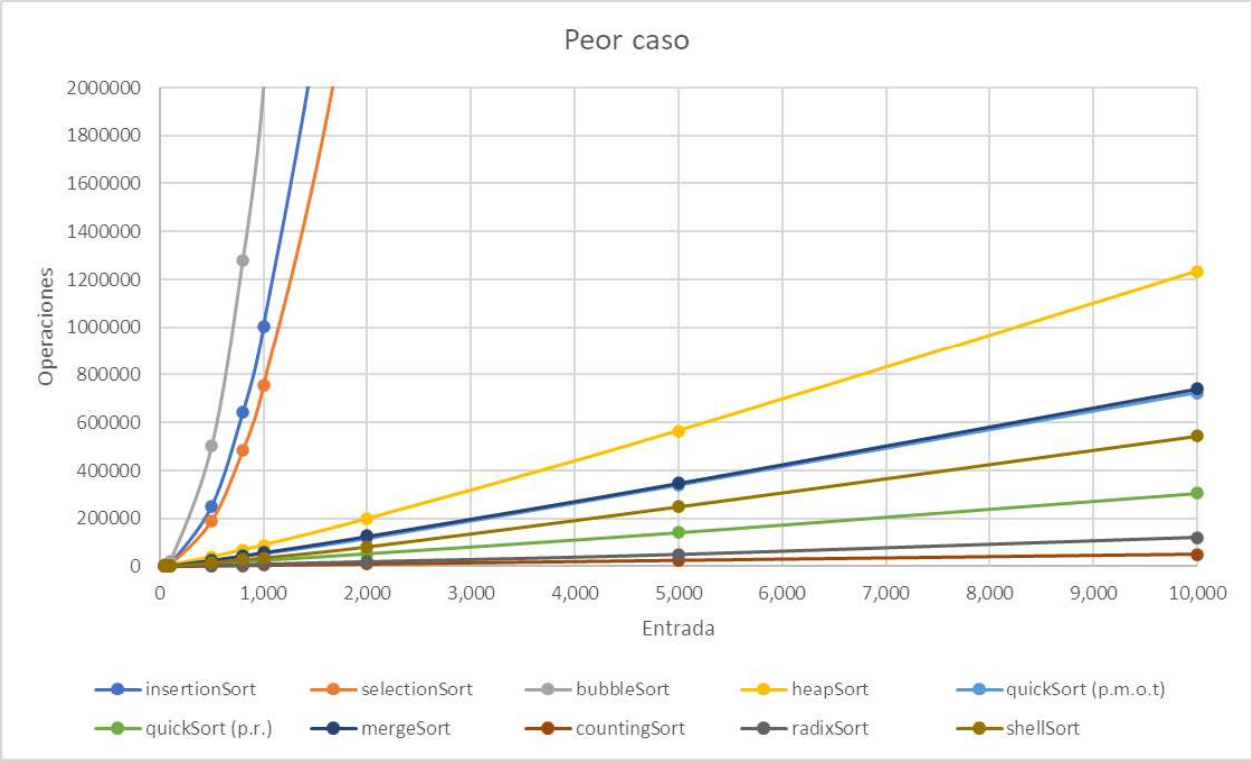
Caso promedio:



Equipo 4. (2023). *Gráfica de la complejidad para el caso promedio en todos los algoritmos estudiados* [PNG]. Excel.

En el caso promedio ya se va notando como los algoritmos que no estaban en su caso ideal cambian drásticamente su forma en la gráfica, especialmente el *Insertion Sort* y el *Bubble Sort* porque pasan a tener complejidades cuadráticas, donde el *Bubble Sort* es incluso peor que el *Selection Sort* para el caso promedio. Los demás algoritmos más consistentes se mantienen en posiciones similares, ya que su complejidad es la misma. Empeoro, con entradas más grandes en conjunto con un rango de valores aleatorios mayor, podría obtenerse un orden diferente para algunos algoritmos.

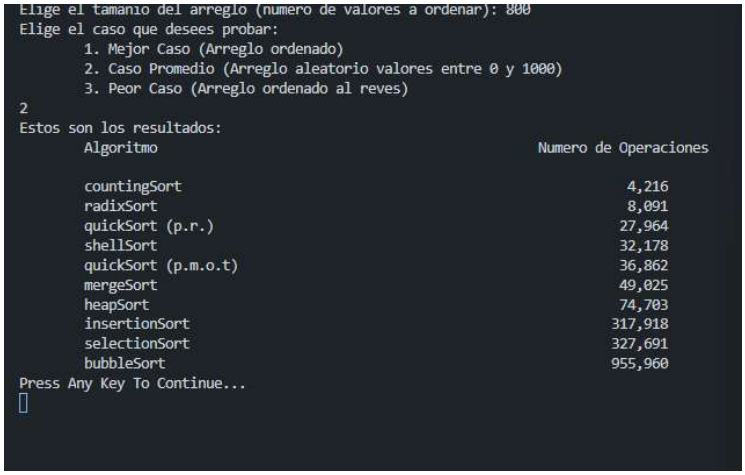
Peor caso:



Equipo 4. (2023). *Gráfica de la complejidad en el peor caso para todos los algoritmos estudiados* [PNG]. Excel.

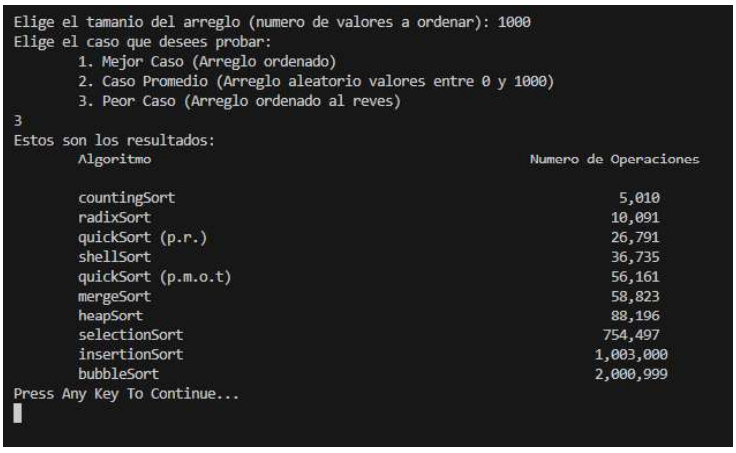
En el peor caso no se tiene mucha diferencia con el caso promedio, pues a partir de ese caso no había una complejidad distinta, las consistentes se mantienen igual y las cuadráticas ocupan posiciones similares, simplemente es que las operaciones aumentan en más cantidad, pero sin salirse de sus limitaciones de complejidad. Algo curioso a denotar es que el *Selection Sort* a pesar de que en los mejores casos es muy malo es el algoritmo más consistente, ya que su forma en esta grafica no es muy distante en comparación a las demás, mientras que *Bubble Sort* llega a ser el peor algoritmo en el peor caso y en el caso promedio. Nuevamente, hay que tener en consideración que existen intentos de mejora para los algoritmos, con el fin de reducir su complejidad en alguno de los casos, pero en esto también se podrían ver comprometidos los otros dos casos mejorándolos o empeorándolos, en base a la entrada y valores que se tengan.

Algunas de las ejecuciones para la recolección de datos:

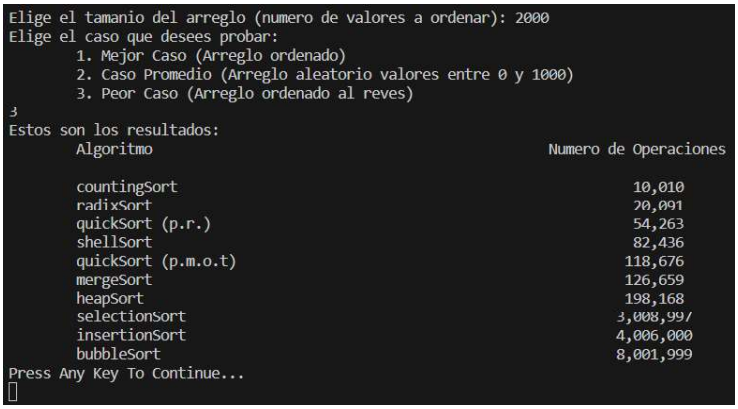


Equipo 4. (2023). *Caso promedio con una entrada de 800* [PNG]. Herramienta de recortes.

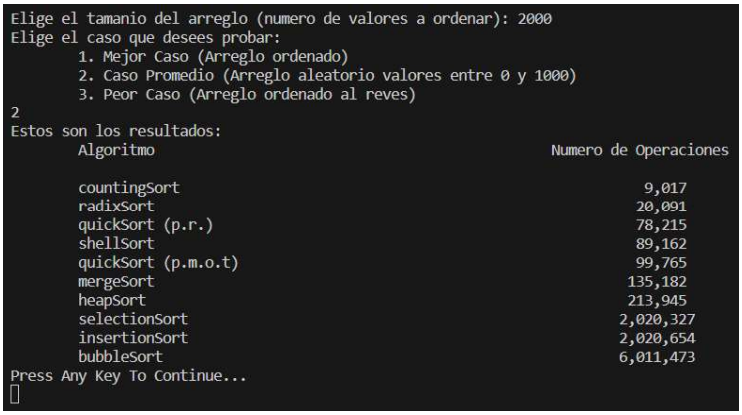
Equipo 4. (2023). *Caso promedio con una entrada de 1000*[PNG]. Herramienta de recortes.



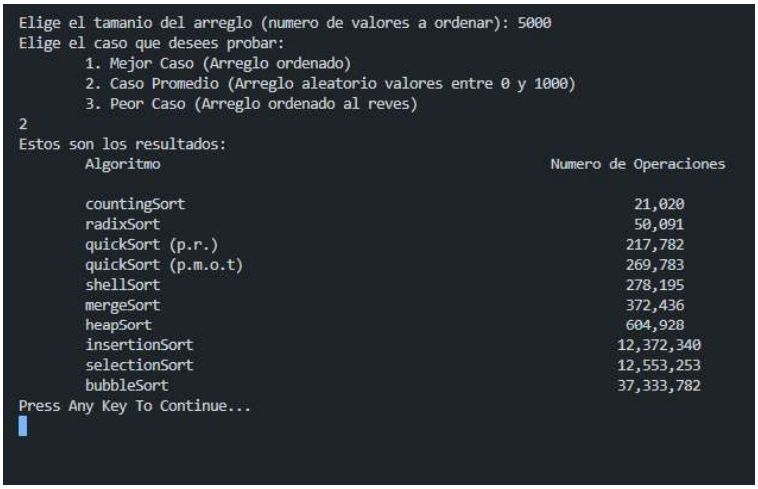
Equipo 4. (2023). Peor caso *con una entrada de 1000* [PNG]. Herramienta de recortes.



Equipo 4. (2023). *Peor caso con una entrada de 2000* [PNG]. Herramienta de recortes.



Equipo 4. (2023). *Caso promedio con una entrada de 2000* [PNG]. Herramienta de recortes.



Equipo 4. (2023). *Caso promedio con una entrada de 5000* [PNG]. Herramienta de recortes.

Conclusiones

Hernández Gallardo Daniel Alonso

Los algoritmos de ordenamiento representan una parte fundamental para el mundo que manejamos, pues con la era digital y cientos de miles de datos en información es importante mantener un orden que nos permita acceder a esa información de una manera rápida. Con la tecnología actual parece ser que los algoritmos de ordenamiento se han quedado estancados y no se ve la llegada de uno ideal, pues los fáciles de implementar toman mucho tiempo, los complejos de implementar solo alcanzan un estándar de O (n log n) y los que llegan a ser lineales poseen la desventaja de ocupar mucha memoria, por lo que no hay un algoritmo optimo o que se considere el mejor y se tiene que adaptar a las necesidades de la información que se va a ordenar.

Este proyecto fue muy útil para entender estos algoritmos a fondo e identificar rápidamente cual algoritmo de ordenamiento podría ser útil según lo que necesitáramos. Asimismo, considero que el visualizar las operaciones es algo muy útil y que nos da mucha más información que visualizar el tiempo de respuesta, dado que las operaciones se pueden ajustar a la máquina, lenguaje y demás componentes que puede volatizar el tiempo de respuesta.

De esta forma, este fue un excelente proyecto para detallar conocimientos, y aunque es algo tardado de realizar, hacerlo en equipo ayuda a entender y compartir conocimientos haciéndolo más ameno. Además, se contó con el tiempo suficiente para desarrollo. Finalmente, el proyecto cumplió con el objetivo, dado que entendí completamente la complejidad de cada uno de los algoritmos y por qué llegan a fluctuar.

Jiménez Ortíz Sebastián

Tras terminar la practica 3 sentía que dominaba los algoritmos de ordenamiento, pues entendía cómo funcionaba cada uno y podía implementarlos en código. Después de terminar este proyecto y la tarea 1 me parece que aún tengo mucho que aprender sobre los algoritmos de ordenamiento y sobre el análisis de complejidad. La investigación realizada para completar estos trabajos me hizo conocer una variedad más grande de algoritmos aparte de los que hemos visto hasta ahora, muchos de ellos variaciones o mejoras de estos, pero algunos otros utilizan técnicas más avanzadas como los algoritmos paralelos.

También pude darme cuenta de que analizar la complejidad de un algoritmo no es tan trivial como podría parecer, ya que para hacer un análisis exacto hay que cuidar cómo se medirá la eficiencia, incluso la metodología de contar cada paso (instrucción) para todos los algoritmos, puede no ser tan justa como había supuesto. No todas las instrucciones “valen” lo mismo, ni requieren la misma cantidad de tiempo en ejecutarse.

Aun asi me parece que los resultados obtenidos se aproximan lo suficiente a los teoricos que esperábamos, por lo que creo que los objetivos iniciales del proyecto se cumplieron. Lo aprendido en este trabajo me hace reafirmar la idea de que no se puede denominar a un algoritmo de ordenamiento como superior absoluto a los demás. Dependerá del contexto en el que nos encontremos:

- La cantidad de elementos a ordenar
- El rango de los elementos
- La variable que queremos minimizar (tiempo o memoria)
- El estado en el que se encuentre la colección

Todos estos elementos influirán en la elección del algoritmo de ordenamiento adecuado.

Por último y talvez algo fuera de relación con los objetivos del proyecto, tras batallar con la creación de las gráficas, es evidente la importancia de saber presentar la información recolectada para interpretar correctamente los resultados.

Me gusto el objetivo del proyecto, el tema de algoritmos de ordenamiento quizá es mi preferido entre los temas de programación visto desde que empecé la carrera y las actividades requeridas junto con las practicas, me permitieron entenderlo.

López Flores Isaac

Después de finalizar este proyecto, considero que el objetivo se cumplió, pues logré observar la complejidad temporal asintótica de los algoritmos de ordenamiento vistos en la clase, lo que me permitió comparar su eficiencia al probarlos con grandes volúmenes de información, en este caso para números enteros. Por otra parte, el proyecto contribuyo al aprendizaje de varios de los conceptos vistos previamente, porque ordenar datos es una operación usual en muchas de las situaciones con las que convivimos día a día, pues el ordenamiento nos permite manejar grandes volúmenes de información, como se vio a través de las diversas ejecuciones, insertar elementos fácilmente, jerarquizar elementos fácilmente, analizar u procesar algún tipo de información, y por último encontrar rápidamente algún elemento, es decir, buscarlo. De ahí que, a pesar de la extensión de este primer tema de la asignatura, su importancia se tan grande, ya que en un futuro podemos tener una mejor noción para determinar que algoritmo de ordenamiento nos puede convenir para cumplir ciertos requerimientos.

En cuanto a las ventajas del proyecto, encontré que esto contribuye a entender por qué los algoritmos varían su número de operaciones según la entrada y tipo de información que se quiera ordenar, ya que ahorita solo probamos algoritmos internos, pero en el futuro si probamos estos combinando con algoritmos externos, los resultados podrían variar aún más, porque en realidad los volúmenes de información que usamos cotidianamente son enormes y no bastara con usar un algoritmo de ordenamiento interno como los puestos a prueba en este proyecto. No obstante, también hubo algunas desventajas como poder determinar qué operaciones contabilizar y cómo hacerlo, a causa de que en los algoritmos recursivos la cosa se complicaba un poco. Además, tuvimos que investigar y modificar el algoritmo de *Quick Sort* para determinar si los resultados eran similares, lo cual nos llevó a identificar que para este algoritmo el cómo se selecciona el pivote juega un papel muy importante. Finalmente, pese a tener una dificultad aceptable, se reforzaron los conceptos del tema 1 complementados con otros de la asignatura de Programación Orientada a Objetos, e incluso se adquirieron conocimientos para futuras codificaciones.

Por lo anterior, no cambiaría ni modificaría ninguna parte del proyecto, porque desarrollarlo en equipo y poder compartir diferentes conocimientos entre nosotros fue una experiencia agradable, donde la solución se logró gracias a conocimientos de asignaturas pasadas junto con las prácticas, temas de esta asignatura e investigación. Será interesante ver que algoritmos de ordenamiento surgen en el futuro, puesto que la computación es un tema que está en constante evolución, donde además el almacenamiento y memoria sigue aumentando su capacidad. Finalmente, a pesar de que la lógica e implementación de los algoritmos más eficientes no es tan sencilla de comprender en un primer acercamiento, con la investigación y pruebas necesarias, sin duda alguna esa brecha puede desaparecer.