

Facultad de Ingeniería

Ingeniería en Computación e informática

Estructuras de Datos

Unidad 4

**“BÚSQUEDA DE UN MÉTODO DE ORDENAMIENTO QUE CUMPLA CON LOS PARÁMETROS DE EFICACIA EN TIEMPO Y USO DE MEMORIA”**

**Autores**

Alexis Sáez [a.saezparra@uandresbello.edu](mailto:a.saezparra@uandresbello.edu)

Leticia Palazuelos [l.palazuelosgutirrez@uandresbello.edu](mailto:l.palazuelosgutirrez@uandresbello.edu)

José Carlos García Padrino [josecgarcia233@gmail.com](mailto:josecgarcia233@gmail.com)

Índice

[Introducción](#_xbd79m3s242b)

[Descripción del Problema](#_f99v4g9vhjph)

[Solución](#_5skfngg9l7em)

[Análisis del Problema](#_2yypbhqei3l4)

[Fundamentos Teóricos](#_xu3z2odktrwa)

[Solución del Problema](#_b1wkcri7t6nf)

[Método de solución:](#_bm0553y1nnu3)

[Implementación](#_2q0g8d7mgrdq)

[Modo de Uso](#_r6s5hdjck42v)

[Resultados](#_obpxjm8szv1l)

[Conclusiones](#_tg1pjdbkawr7)

Anexos

# Introducción

## Descripción del Problema:

La Real Academia Española describe la ordenación como la *“colocación de las cosas en el lugar que les corresponde”*. En la computación, adecuamos este concepto para que bajo un criterio establecido podamos ubicar archivos o elementos de un vector en el orden que este disponga. Para los efectos de la documentación, la ordenación es una de las operaciones básicas en la que las computadoras emplean gran cantidad de tiempo, es por ello que surge la necesidad de encontrar el método más adecuado y eficiente que cumpla con el criterio que definamos para nuestro algoritmo.

Por consiguiente, la búsqueda de un método de ordenamiento que cumpla con los parámetros de eficacia en tiempo y uso de memoria pasa a ser un inconveniente por ejemplo, para diversos sistemas o softwares de empresas con bases de datos, pues mientras mayor sea el tiempo de ejecución mayor será el costo y requiere una solución inmediata. Específicamente, el problema que se aborda en este informe radica en determinar cuál de tres métodos conocidos (bubble sort, selection sort e insertion sort) ordena de forma más eficiente los elementos de un vector.

## **Solución**:

Para encontrar el método indicado se realizarán tres pruebas con entradas de datos de distintos tamaños y distribuidas de diferente forma a cada algoritmo, cuyos resultados se colocarán en 3 archivos distintos con las gráficas en excel correspondientes a cada uno en anexos para hacer una comparación y establecer cuál es el más eficiente. En la primera prueba, se debe contar el número de comparaciones de un arreglo generado con números aleatorios, en la segunda prueba se debe tomar el arreglo resultante de la primera prueba, es decir el arreglo con orden ascendente y repetir el procedimiento contando el número de comparaciones. Finalmente, para la tercera y última prueba se utiliza un arreglo ordenado con elementos de menor a mayor. En cada implementación de las 3 técnicas de ordenación, debe haber un contador de comparaciones y este número de comparaciones almacenadas en el contador serán mostradas en los archivos de salida.

# 

# **Análisis del Problema**

**Detalle del problema:**

La investigación tiene por objeto identificar el método de ordenamiento de vectores que sea más eficaz en tiempo y uso de memoria. La eficiencia estará dada por el menor tiempo que emplea un método en realizar el ordenamiento de los elementos de un vector y además ocupe la menor cantidad de memoria, por tanto, debemos escoger entre tres diferentes métodos (burbuja, inserción, selección) aquel que para diferentes tamaños de conjuntos y criterios de ordenamiento sea el que cumpla con nuestros parámetros de eficiencia. El tiempo de ejecución en estos métodos de ordenamiento directo depende de la cantidad de comparaciones que cada uno realice para lograr el objetivo, es por ello que el problema se incrementa con el aumento de la cantidad de elementos o datos a ordenar.

**Supuestos:**

Se puede suponer que el método más adecuado será aquel en el que no hay que comparar todos los elementos del arreglo entre sí, pues mientras menor sean las comparaciones, menor será el tiempo de ejecución del algoritmo. Además, también se puede suponer que mientras la distribución de elementos del vector se acerque a la clave de orden establecida, el tiempo de ejecución y el número de comparaciones será menor.

**Metodología:**

La metodología se divide en fases diferentes para lograr la solución adecuada al problema. Primeramente, se utilizará un programa construido con el lenguaje de programación C usando el IDE Visual Studio Code. Se realizarán tres pruebas a cada método de ordenación con las siguientes entradas de datos distintas:

1.- Conjunto con elementos generados aleatoriamente (elementos desordenados).

2.- Conjunto resultante de la Prueba 1, con elementos ordenados de menor a mayor.

3.- Conjunto resultante de la Prueba 1, con los elementos ordenados de mayor a menor.

En cuanto a las etapas de desarrollo y programación, para la primera, se creará una estructura estática, en este caso un arreglo, en el cual los elementos se generarán aleatoriamente. Esto se obtendrá a través de la función *“rand()”*. Para evitar que los elementos se repitan, se usará la función “*srand(time(NULL))*”, a la que se le pasará de parámetro la fecha/hora del sistema que se utilizará como número inicial para las cuentas. Para la ejecución del programa en lenguaje C, se incluirán las librerías “stdio.h”, ”stdlib.h”, “time.h”, “string.h”.

Para la segunda etapa, al programa principal se le agregan los algoritmos de los métodos de ordenación a estudiar. Este programa principal generará tres archivos con las salidas que se analizarán posteriormente. Además, se crearán 3 programas separados denominados ‘bubble.c’, ‘insertion.c’, ‘selection.c’; cada uno contendrá un algoritmo de ordenamiento diferente, un contador de comparaciones y tendrá como salida un archivo con su nombre respectivo. En cada algoritmo de ordenación se agrega un contador que acumula y se incrementa cada vez que existe una comparación. Este contador se usará posteriormente como dato duro para el análisis de resultados.

La tercera. etapa consistirá en analizar las diferentes salidas en los archivos. La forma de análisis será comparativa con gráficos de Comparaciones vs Tamaños, tablas con números de comparaciones en el peor caso y mejor caso y también una tabla de comparación de los tiempo de ejecución de cada algoritmo. Se harán ecuaciones de correlación en los gráficos para obtener conclusiones a partir de sus coeficientes R.

**Situaciones de borde:**

Es importante precisar que para este informe se usarán vectores o arreglos de enteros cuyos valores (no repetidos) están en el rango [0,100.0000] y cuyos tamaños están entre 10.000 y 90.000. Dado un contexto mayor los datos de este informe podrían no ser significativos. Igualmente se debe considerar que existen más algoritmos de ordenamiento y en esta implementación solamente fueron considerados 3.

# 

# **Fundamentos Teóricos**

Existen diversos métodos de ordenación de vectores en computación, sin embargo este informe estará enfocado en abordar tres de ellos solamente, **intercambio o burbuja (bubble sort), selección (selection sort) e inserción (insertion sort);** todos métodos de ordenación directa.

El **bubble sort** es el algoritmo de ordenamiento más simple y funciona al intercalar elementos adyacentes repetidas veces hasta que el vector esté ordenado de acuerdo a una clave dada. Por ejemplo, si tomamos el siguiente arreglo : {5, 1, 4, 2, 8} el algoritmo toma los primeros dos elementos, el 5 y el 1 y los intercambia dejando el arreglo en el siguiente orden: {1 5 4 2 8}. Luego pasa a comparar el 5 y el 4 que son el siguiente par de elementos y como el 5 es mayor que el 4, vuelve a hacer el intercambio. De esta forma y repitiendo el proceso de comparar pares de elementos, el arreglo termina ordenado de menor a mayor {1, 2, 4, 5, 8}. Una ejemplificación adicional se encuentra en los anexo, mostrando cada paso realizado por el algoritmo. Cabe destacar que el algoritmo depende de n, el número de elementos en el arreglo y funciona más eficientemente dado un arreglo que está relativamente ordenado.

En el mejor de los casos este algoritmo tiene un orden de ejecución de O(n), es decir el tiempo de ejecución aumenta lentamente en relación al aumento de sus datos de entrada. En el peor caso tiene un orden de O() que es cuando el arreglo está ordenado a la inversa. En la Tabla 1.1 se encuentran las comparaciones de tiempos de ejecución de cada algoritmo.

El **selection sort** tiene un orden de ejecución de O(); es un algoritmo de clasificación por comparación en sitio. Su funcionamiento para vectores es el siguiente: Se selecciona el elemento más pequeño del arreglo y este es intercambiado con el elemento que está en la primera posición del arreglo y ese elemento luego forma parte del arreglo ordenado. Este proceso se repite, recorriendo en cada iteración una posición a la derecha. En la figura 1.2 del anexo se puede ver la implementación en pseudocódigo.

Finalmente, el **insertion sort** cumple la misma función que los dos anteriores solamente que el proceso de funcionamiento varía. Este algoritmo, al igual que el selection sort se basa en la clasificación por comparación en sitio y mantiene una sub-lista que está ordenada en donde inserta un elemento. Por ejemplo, tomamos el siguiente arreglo: {14, 33, 27, 10, 35, 19, 42, 44}. El primer paso es comparar los primeros dos elementos que serían el 14 y el 33; debido a que están en el orden correcto para este caso (orden ascendente) el 14 pasa a la sub-lista. A continuación se avanza una posición y se compara los siguientes dos elementos que serían el 33 y el 27 y como el 33 es mayor, los intercambia y de esta forma el 27 pasa a la sublista de elementos ordenados. Este proceso se repite varias veces hasta que el arreglo este completamente ordenado, dando como resultado : {10, 14, 19, 27, 33, 35, 42, 44}.

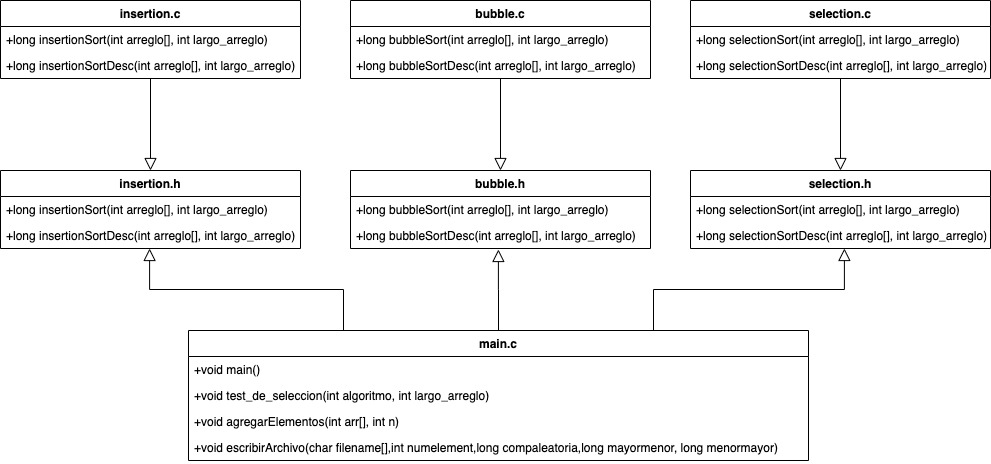
# **Solución del Problema**

## **Método de solución:**

1. Se investiga los métodos de ordenamiento requeridos (bubble, insertion, selection sort), con lo cual se obtienen los algoritmos de estos.
2. Se selecciona el lenguaje y el entorno de desarrollo para la implementación de estos algoritmos.
3. Se diseña un programa que con los requerimientos del taller para lograr analizar el comportamiento de estos algoritmos en arreglos de 10.000, 50.000 y 90.000 elementos.
4. Para lograr obtener los datos requeridos se implementa un menú que permite seleccionar el algoritmo que se desea analizar y el largo del arreglo a ordenar, donde una vez terminada la ejecución se genera un archivo con los resultados por cada prueba.
5. Se toman los resultados desde los archivos y se genera un gráfico de dispersión el cual muestra la comparativa del rendimiento de cada algoritmo con los distintos arreglos.
6. Con lo anterior se sacan las conclusiones.

## Implementación:

Se codifica el programa que va a testear los algoritmos de acuerdo al siguiente diseño:



El lenguaje seleccionado es c, y se codifica Visual Studio Code que es un IDE desarrollado por Microsoft y se puede adquirir de manera gratuita. Además, para facilitar el trabajo de codificación colaborativo, se utiliza el plugin de Microsoft llamado Live Share.

## Modo de Uso:

Dentro del archivo main.c del programa hay un menú que facilita el uso y ejecución del programa. A través del main, el usuario puede elegir entre las opciones seleccionadas si desea realizar la prueba con el bubble, insertion o selection sort. Tras elegir el método el programa solicitará al usuario el tamaño que desea colocar al arreglo. Una vez determinado, se ejecutarán las 3 pruebas y de acuerdo a los resultados guardados en el contador de comparaciones, se imprimirán los archivos de texto de salida con los diferentes resultados.

Para la compilación del programa se utiliza el comando :

|  |
| --- |
| gcc main.c bubble.c selection.c insertion.c -o app.o |

Posteriormente se debe colocar el siguiente comando:

|  |
| --- |
| ./app.out |

# Resultados

Las pruebas realizadas fueron 9 y cada una con un caso diferente. Para los 3 algoritmos los casos de prueba fueron los mismos. Para cada uno de los 3 algoritmos se realizaron 3 pruebas cada una con un arreglo de tamaño 10.000, 50.000, 90.000 elementos. El Primer paso de la prueba fue llenar el arreglo de tamaño seleccionado generando números aleatorios. Para el segundo paso, se usa el arreglo que ordenado, devuelto por la primera prueba y este está ordenado de mayor a menor de tamaño Xmil. En la tercera prueba se genera un arreglo ordenado de menor a mayor de tamaño Xmil.

Los resultados generados son los siguientes:

En la Tabla 1.3 se muestran los 3 resultados de los 3 diferentes tamaños aplicados y los números decompraciones respectivas para cada método. A grandes rasgos se confirmó que el bubble sort es el de menor rendimiento en todas las pruebas dado que los números de comparaciones doblan a los otros dos métodos. Igualmente en los Gráficos 1.1, 1.2 y 1.3 se puede ver los mismos resultados expresados en gráficos. En los gráficos se observa que el número de comparaciones de los algoritmos en función del tamaño crece en forma polinómica de grado 2, esto es demostrable con el coeficiente de correlación que para los 3 gráficos es 1 en las ecuaciones asociadas a los diferentes métodos.

Igualmente se realizó una implementación de bubble sort mas optimizada que la que fue usada en las pruebas oficiales, y para poder ver los resultados de estas dos implementaciones sobre el mismo método se adjunto la Tabla 1.2. En los gráficos 1.4 y 1.5 se puede ver que la función polinómica,ya no es de grado 2, es una recta con una pendiente muy grande, es decir crecer con mayor lentitud que el bubble normal.

## Conclusiones

# Conociendo y comprendiendo el funcionamiento de cada método y sus diferentes aspectos, vemos que los resultados de la implementación de cada uno son distintos. Como anteriormente se describieron los tiempos de ejecución de cada algoritmo, incluso antes de la implementación pudimos confirmar los supuestos reflejados en este informe.

# Pese a que tanto el bubble sort como el insertion sort tienen un tiempo de complejidad de O(N2), generalmente el insertion sort supera a bubble sort. Esto se debe al número de intercambios que realizan ambos algoritmos; el número de intercambios en bubble sort es mayor, sin embargo la simplicidad de bubble sort reduce las líneas de escritura de código. En cuanto a la comparación de insertion sort y bubble sort, el primero es muy eficiente para ordenar listas de diferentes tamaños

Al comparar los métodos de selection e insertion podemos ver que cuando el arreglo tiene un mayor tamaño, en este caso 90 mil elementos, ambos tienen igual rendimiento. La diferencia entre los dos se nota en la segunda prueba cuando se realiza el arreglo ordenado en forma ascendente ya que el insertion sort supera al selection sort. Finalmente, en cuanto a la primera prueba, insertion sort prueba ser más eficiente que el selection sort, superando casi en un 50 % de rendimiento. Como se había supuesto el bubble sort, siendo el método más simple de ordenamiento es el menos eficaz, siendo su número de comparaciones en todas las pruebas casi el doble que las de los demás métodos. Sin embargo, al ocupar el método de bubble optimizado, el número de comparaciones disminuye considerablemente en comparación con el original, pero aún así sigue siendo más lento que los otros estudiados. Por ende, nuestra búsqueda del mejor algoritmo de ordenamiento

## Anexos

Figura 1.1 Ejemplo del funcionamiento del bubble sort

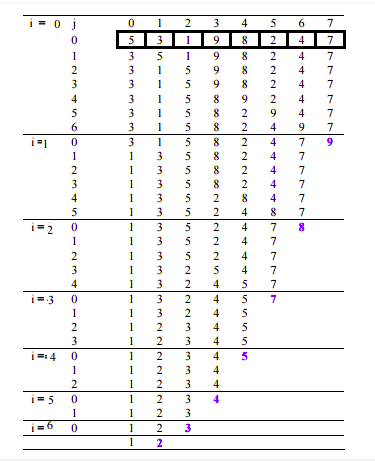


Figura 1.2 Ejemplo del procedimiento del Selection sort

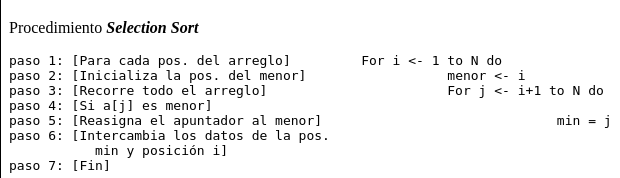


Figura 1.3 Ejemplo del funcionamiento del insertion sort

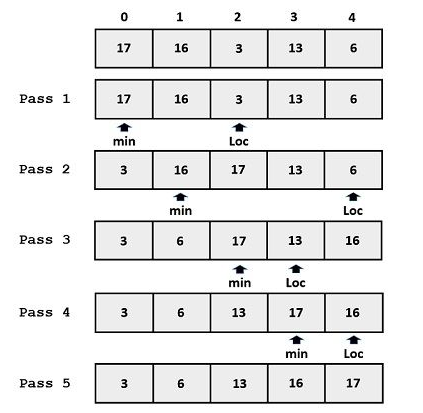


Tabla 1.1 Comparación de los tiempos de ejecución de cada algoritmo en notación Big O

|  |  |  |  |
| --- | --- | --- | --- |
| Comparación de los tiempos de Ejecución en Notación Big O | | | |
|  | Mejor caso | Caso Promedio | Peor Caso |
| Bubble Sort | **O(N2)** | **O(N2)** | **O(N2)** |
| Selection Sort | **O(N2)** | **O(N2)** | **O(N2)** |
| Insertion Sort | **O(N)** | **O(N2)** | **O(N2)** |

Tabla 1.2 Comparación de un bubble sort mas optimizado versus uno que no esta optimizado(el menos optimizado fue usado en la implementación).

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10,000 | 50,000 | 90,000 |
| Bubble Sort | numCompElemAleat 100000000  numCompMenorMayor  100000000  numCompMayorMenor  100000000 | numCompElemAleat2500000000  numCompMenorMayor  2500000000  numCompMayorMenor  2500000000 | numCompElemAleat  8100000000  numCompMenorMayor 8100000000  numCompMayorMeno: 8100000000 |
| Bubble Sort Optimizado | numCompElemAleat  20157554  numCompMenorMayor  0  numCompMayorMenor  50005000 | numCompElemAleat  415832228  numCompMenorMayor  0  numCompMayorMenor  1250025000 | numCompElemAleat  825671944  numCompMenorMayor  0  numCompMayorMeno:  2250225000 |

Tabla 1.3 Comparación de los 3 metodos de ordenamiento correspondientes a los resultados de la implementación.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10,000 | 50,000 | 90,000 |
| Bubble Sort | numCompElemAleat 100000000  numCompMenorMayor  100000000  numCompMayorMenor  100000000 | numCompElemAleat2500000000  numCompMenorMayor  2500000000  numCompMayorMenor  2500000000 | numCompElemAleat  8100000000  numCompMenorMayor 8100000000  numCompMayorMeno: 8100000000 |
| Insertion Sort | numCompElemAleat  24982343  numCompMenorMayor  0  numCompMayorMenor  49995000 | numCompElemAleat 624897929  numCompMenorMayor  0  numCompMayorMenor 1249975000 | numCompElemAleat 2026428645  numCompMenorMayor  0  numCompMayorMenor 4049955000 |
| Selection sort | numCompElemAleat  49995000  numCompMenorMayor 49995000  numCompMayorMenor 49995000 | numCompElemAleat  1249975000  numCompMenorMayor 1249975000  numCompMayorMenor 1249975000 | numCompElemAleat 4049955000  numCompMenorMayor 4049955000  numCompMayorMenor 4049955000 |

Gráfico 1.1

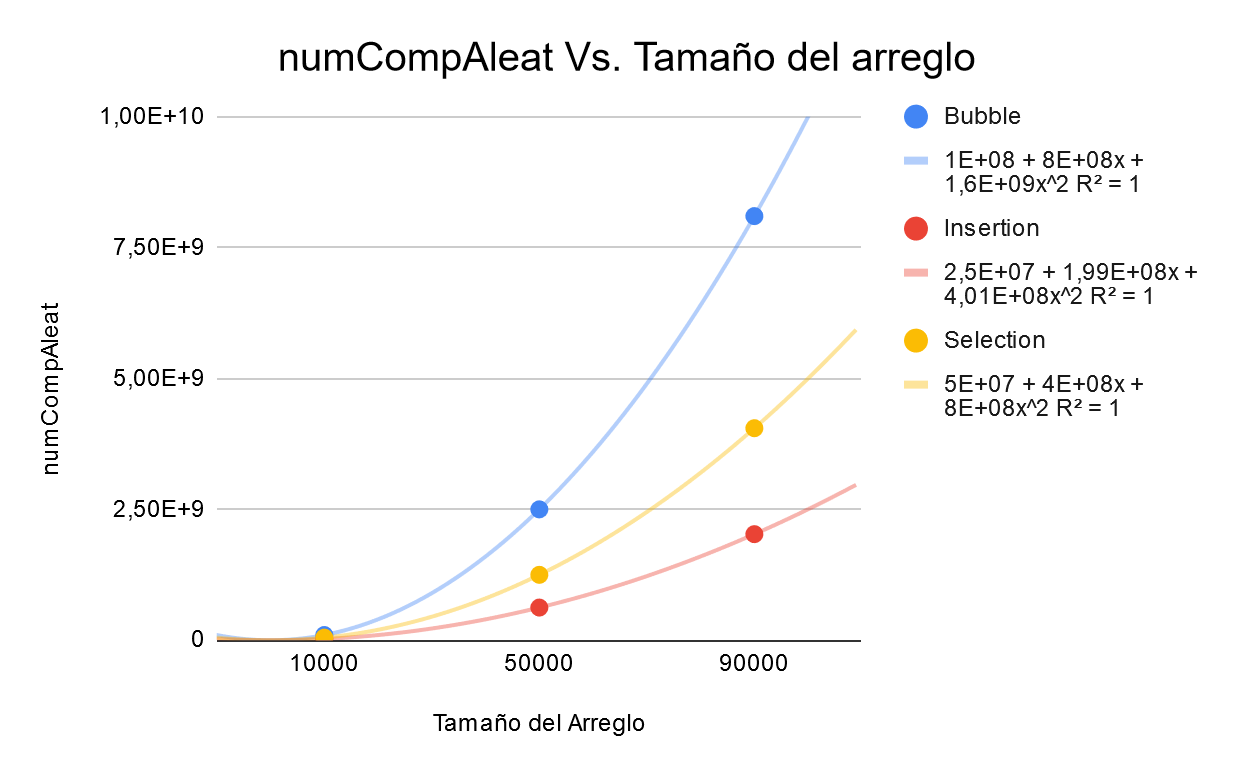


Gráfico 1.2

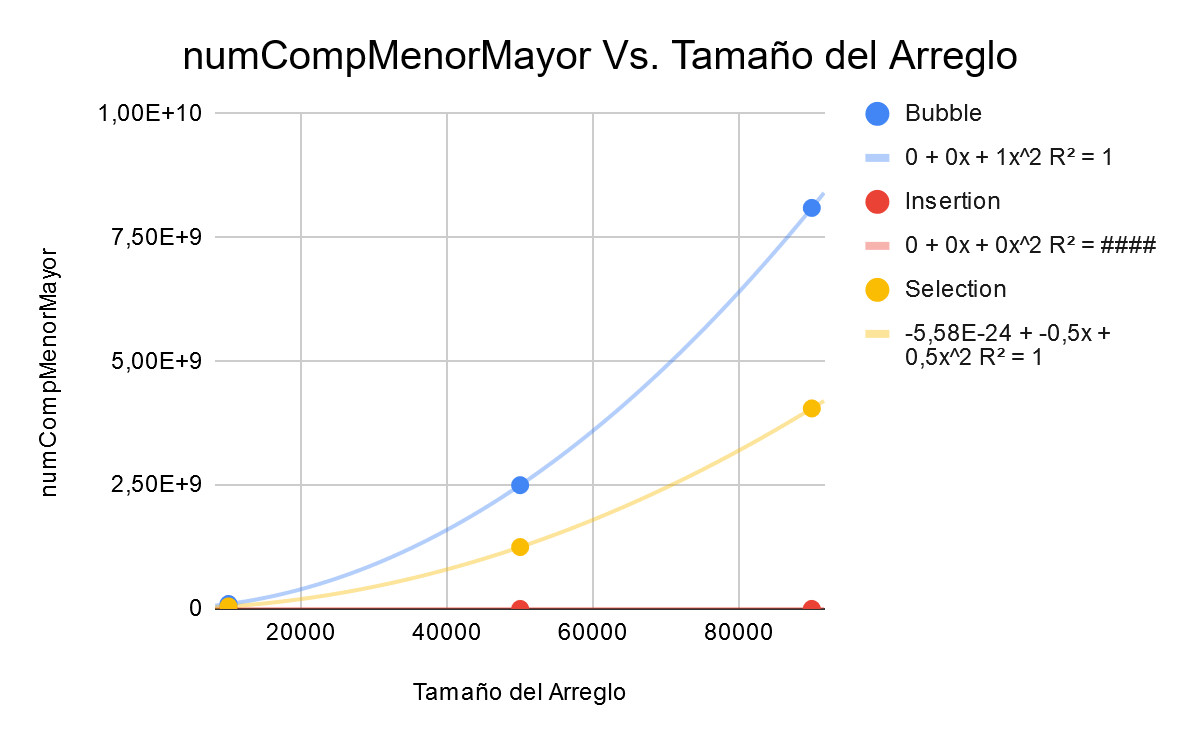


Gráfico 1.3

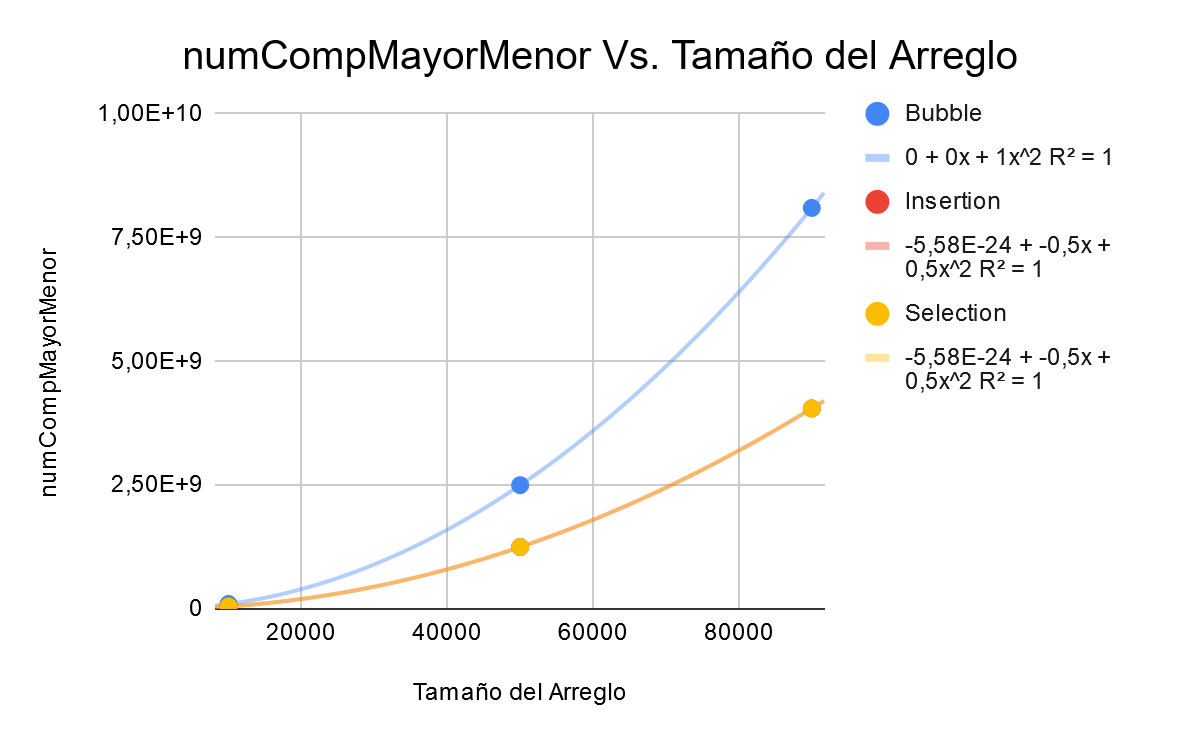


Gráfico 1.4

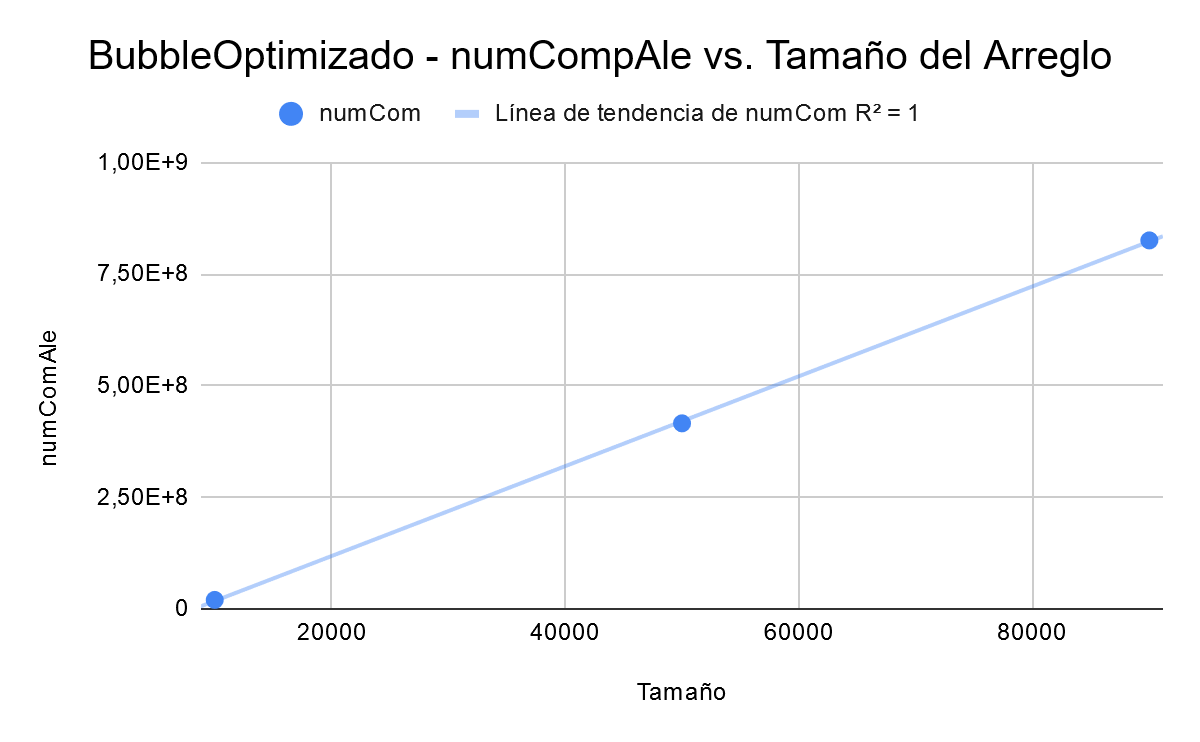


Gráfico 1.5

