Tarea 1: Comparación de Algoritmos de Ordenación y Algoritmos de Búsqueda

Integrantes:

Santiago Martinez

Profesores:

Jose Manuel Saavedra

30 de Agosto del 2024

Resumen:

La investigación se centra en evaluar la eficiencia de diferentes algoritmos de búsqueda y ordenación para determinar la existencia de elementos en cinco diccionarios de distintos tamaños. Los algoritmos de búsqueda analizados son la búsqueda secuencial y la búsqueda binaria, mientras que los algoritmos de ordenación seleccionados son Selection Sort y Radix Sort.

La búsqueda secuencial examina cada elemento de la lista de manera lineal hasta encontrar el dato deseado, mientras que la búsqueda binaria divide repetidamente el arreglo en segmentos, lo que permite localizar el elemento de manera más eficiente.

Los algoritmos de ordenación primero el selection sort va recorriendo la lista e intercambiando el menor valor encontrado al principio de la lista así ordenando la lista de manera creciente, en cambio el radix sort ocupa la filosofía del counting sort el cual llena una lista de largo de la cantidad de caracteres posibles y va rellenando en los espacio la cantidad de ocurrencias, luego con esa lista rellena otra lista la cual se va ordenando según el valor del carácter que se está analizando.

Los resultados sugieren que la combinación de la búsqueda binaria con Radix Sort es la más eficiente para manejar grandes volúmenes de datos, debido a su comportamiento lineal en términos de tiempo de ejecución, lo que le otorga una ventaja significativa.

En conclusión, es fundamental elegir el algoritmo adecuado según las necesidades específicas de cada situación, evaluando cuándo es conveniente invertir tiempo en optimizar el código para lograr un mejor rendimiento en la ejecución.

Introducción

En la ciencia de la computación, la construcción de algoritmos eficientes es fundamental para optimizar el uso de tiempo y recursos. Este informe se enfoca en la evaluación de distintos algoritmos de búsqueda aplicados a diccionarios de tamaños variables, con el objetivo de determinar cuáles son más óptimos en diferentes escenarios.

Los datos consisten en cinco diccionarios, cada uno conteniendo palabras en formato de string, con tamaños de 10,000, 50,000, 100,000, 200,000, y 400,000 elementos respectivamente. Además, se utiliza un arreglo adicional llamado *‘queries’*, compuesto por 1 millón de datos, para realizar búsquedas en estos diccionarios. El objetivo es medir la eficiencia de distintos algoritmos en términos de tiempo de ejecución y analizar los problemas que puedan surgir en la búsqueda.

Desarrollo de algoritmos

#### **Algoritmos de Búsqueda**

Los algoritmos de búsqueda son los responsables de localizar la palabra en cuestión dentro de los diccionarios. Los algoritmos utilizados en este estudio son los siguientes:

* **Búsqueda Secuencial:** Este es el método más sencillo para encontrar un elemento. El algoritmo recibe el elemento que se desea buscar y recorre el arreglo desde el principio hasta encontrarlo. Si no lo encuentra, recorrerá el arreglo completo sin detenerse.

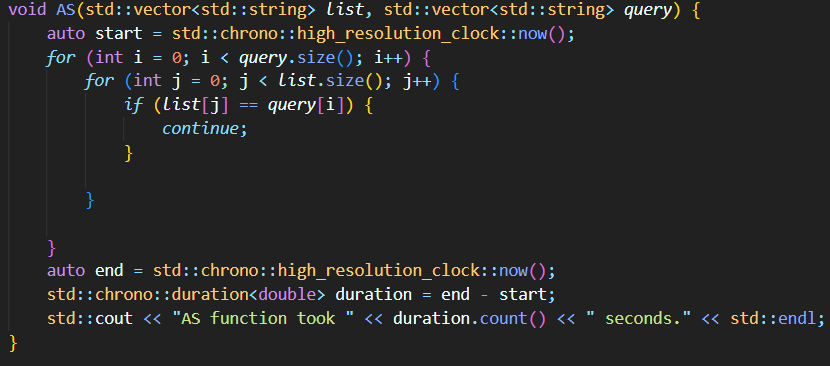


Imagen N°1: Búsqueda Secuencial

Elaboración propia

Como podemos observar en la Imagen N°1, el algoritmo toma los valores *list* y *query*, donde el primero representa el diccionario y el segundo es el arreglo de elementos a buscar. Luego, se inicia un cronómetro para comparar los tiempos de ejecución. El algoritmo principal comienza en el bucle que recorre la lista de elementos a buscar (*query*). Dentro de este bucle, se encuentra otro bucle que recorre el diccionario (*list*). En cada iteración, se evalúa una condición if que verifica si el elemento actual de *query* es igual al elemento actual del diccionario. Si se cumple esta condición, el algoritmo pasa al siguiente elemento de *query*, lo que indica que el elemento ha sido encontrado.

* **Búsqueda Binaria:** Este algoritmo requiere que el arreglo esté previamente ordenado. Para ejecutar la búsqueda binaria, se deben generar tres variables: el índice del principio del arreglo, el índice final, y el índice de la mitad. Si el elemento buscado coincide con el valor en la mitad, se considera encontrado. Si no, el algoritmo determina si el elemento se encuentra en el rango entre el principio y la mitad, o entre la mitad y el final. Dependiendo del caso, ajusta el rango y repite el proceso hasta encontrar el elemento o concluir que no está presente.

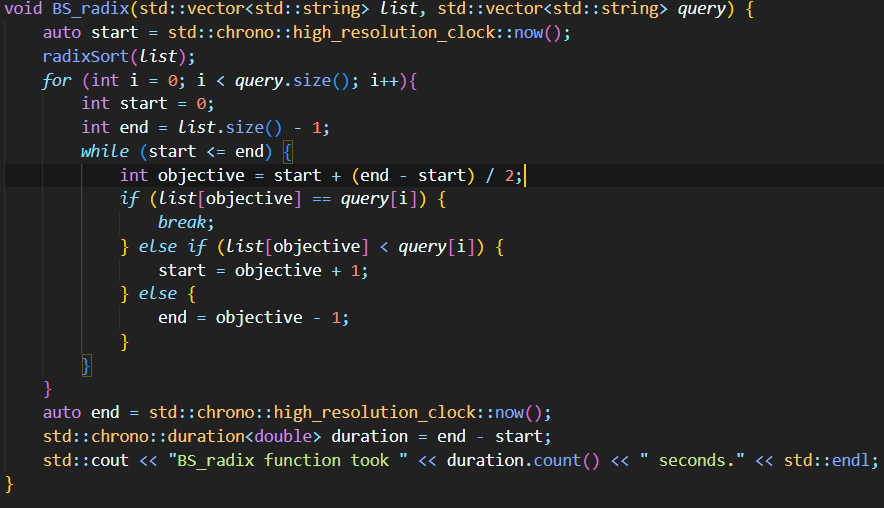


Imagen N°2: Búsqueda Binaria

Elaboración propia

La imagen anterior muestra la búsqueda binaria utilizada para completar la tarea. En este caso, podemos observar que se requieren los mismos parámetros que en la búsqueda secuencial. Antes de analizar el algoritmo en detalle, es importante mencionar que se llama a una función llamada *`radixSort`*, que será analizada más adelante. Esta función se utiliza para ordenar la lista *`query`.*

El algoritmo comienza con un bucle que recorre la lista *`query`*. A continuación, se definen los parámetros *`start`* y *`end`*. Luego, el algoritmo entra en un bucle que se detiene cuando `start` es mayor que *`end`*. Dentro de este bucle, se calcula el índice objetivo como la mitad de *`start`* y *`end`.*

Hay tres condiciones principales en el algoritmo:

1. Si el valor en el índice objetivo es igual al número deseado, significa que el elemento ha sido encontrado en el diccionario.

2. Si el valor en el índice objetivo es menor que el número deseado, se actualiza el parámetro *`start`* al valor del índice objetivo más uno.

3. En caso contrario, se actualiza el parámetro *`end`* a la mitad menos uno.

#### **Algoritmos de Ordenación**

Sabemos que la búsqueda binaria requiere que el arreglo esté ordenado. Para ello, se implementaron dos tipos de algoritmos de ordenación, y se evaluó cuál es más eficiente para optimizar el tiempo de búsqueda:

* **Ordenación por Selección:** Es un algoritmo sencillo de aplicar. Lo que hace es recorrer el arreglo y, durante el recorrido, guarda el índice del elemento de menor valor. Una vez que llega al penúltimo puesto del arreglo, intercambia este elemento con el primer puesto de la lista y luego repite el proceso, intercambiándolo con el segundo, hasta terminar con todo el arreglo.

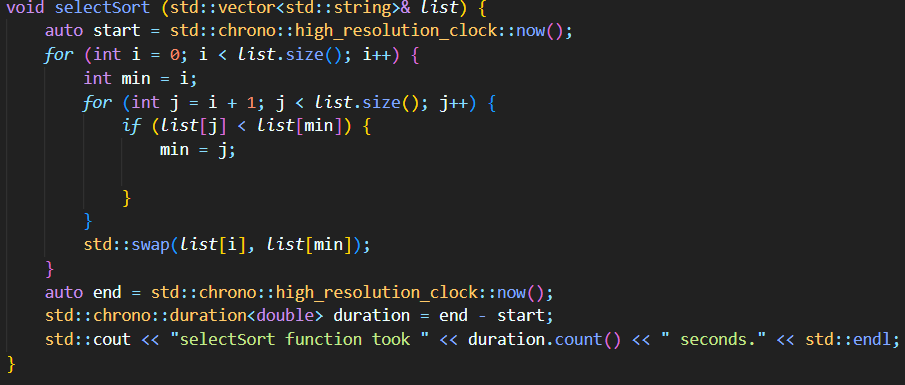


Imagen N°3: Select Sort

Elaboración propia

En este algoritmo, se recibe un solo arreglo, el cual será ordenado. El algoritmo comienza con un bucle que recorre la lista de manera secuencial. Inicialmente, se define el mínimo como el valor en la posición `i`. Luego, se inicia otro bucle que también recorre la lista, pero este comienza un valor más adelante que la posición inicial del primer bucle.

Dentro del segundo bucle, se evalúa una condición que verifica si el elemento actual es menor que el valor mínimo que tenemos definido. Si esta condición se cumple, se actualiza el valor mínimo. Una vez que el segundo bucle termina, se intercambia el valor mínimo con el elemento en la posición `i` de la lista. Este proceso se repite hasta que el primer bucle haya recorrido toda la lista.

Se utilizó este método de ordenación porque es sencillo de aplicar, como hemos visto, y no requiere muchas líneas de código. Lo interesante aquí es observar cómo va perdiendo eficacia al interactuar con arreglos de gran tamaño, lo que nos muestra que lo más simple o corto en términos de código no siempre es lo más eficiente en términos de tiempo.

* **Radix Sort:** Este algoritmo es más complejo que la ordenación por selección. Su funcionamiento se basa en el largo del elemento más largo del arreglo, definido como la "posición deseada". Se crea una lista del largo correspondiente a la cantidad de caracteres posibles (por ejemplo, para comparar números, de 0 a 9). Luego, se recorre el arreglo tomando en cuenta cada carácter de la posición deseada. Si un número tiene el valor 18 y otro el 19, se toman los dígitos 8 y 9, respectivamente. Se cuenta cada instancia ocurrida en la lista creada previamente, que llamaremos **cuenta**, y si el largo del elemento es menor que la posición deseada, se deja en la primera posición de la lista. Después de recorrer el arreglo completo, la lista **cuenta** se hace acumulativa, sumando la cantidad de la posición anterior a la actual. Posteriormente, se crea una nueva lista llamada **output**, y se recorre el arreglo desde el final hacia el principio, colocando los elementos en la lista **output** según los valores acumulados. Luego de colocar el elemento en su posición en la lista **output**, el valor acumulado en la lista **cuenta** se reduce en uno. Este proceso se repite hasta que se ordenan todos los elementos, reduciendo la "posición deseada" en cada iteración.

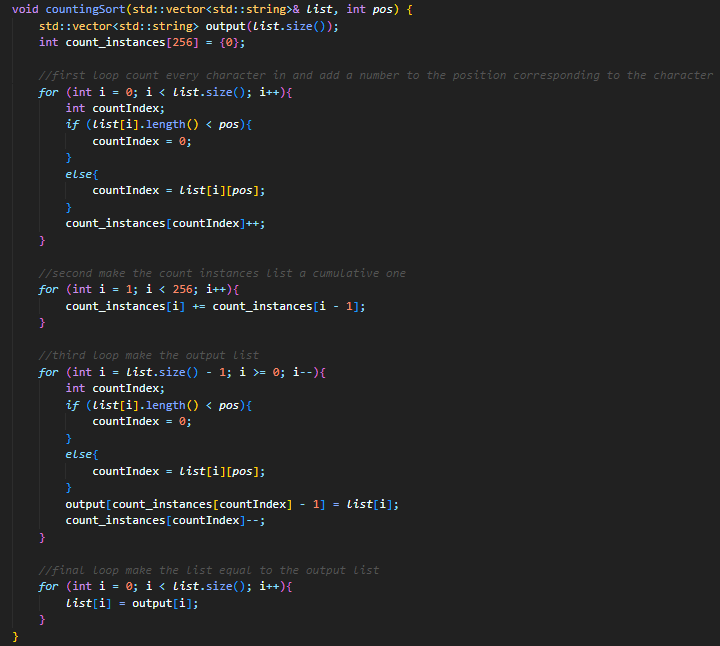


Imagen N°4: Count Sort for Radix Sort

Elaboración propia

En esta sección se lleva a cabo la mayor parte del proceso de Radix Sort. En primer lugar, se crean dos arreglos: uno llamado `output`, que almacenará temporalmente los elementos ordenados, y otro llamado `count\_instances`, que tiene una longitud de 256 (correspondiente al número de caracteres posibles) y en el que cada posición empieza con el valor 0. Este último arreglo se utiliza para contar la cantidad de veces que aparece cada carácter.

El primer bucle recorre la lista de elementos y crea una variable llamada `countIndex`. Luego, se evalúa una condición que verifica si la longitud del elemento es menor que `pos`, la cual determina qué posición del elemento se está observando, comenzando siempre por el valor del elemento más largo. Si se cumple la condición, `countIndex` se establece en cero; de lo contrario, `countIndex` toma el valor del carácter que se está examinando. A continuación, se incrementa en uno el valor de la posición `countIndex` en el arreglo `count\_instances`, y este proceso se repite hasta recorrer toda la lista de elementos.

El segundo bucle convierte el arreglo `count\_instances` en acumulativo, sumando el valor anterior al seleccionado en cada posición.

El tercer bucle llena la lista `output` de una manera particular. Primero, es importante notar que la lista de elementos se recorre de atrás hacia adelante. Luego, se aplica una condición similar a la del primer bucle. Lo que hace este bucle es seleccionar el valor en la lista `count\_instances` en la posición `countIndex` y asignar a esa posición el elemento correspondiente de la lista original, rellenando así la lista `output` hasta que los elementos se ordenan según cada carácter.

Finalmente, el último bucle hace que la lista de elementos sea igual a la lista `output` recién creada.

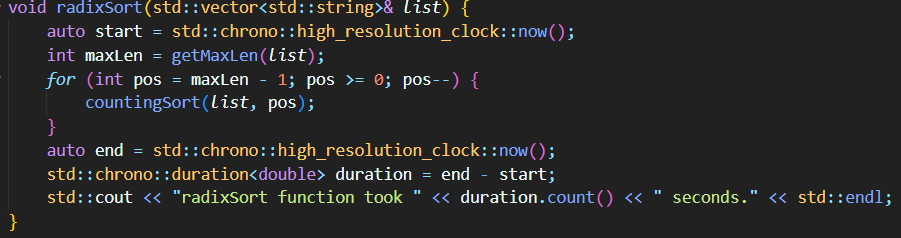


Imagen N°5: Radix Sort

Elaboración propia

Como podemos observar, la función `radixSort` comienza definiendo la longitud del elemento más largo dentro del arreglo. Luego, entra en un bucle que llama repetidamente a la función `countingSort`, la cual ya hemos analizado previamente. El bucle recorre los elementos desde el valor máximo de la longitud hasta el mínimo, decrementando en cada iteración. De esta manera, se analizan los caracteres de los elementos desde el final hacia el inicio.

Como podemos observar, el algoritmo de Radix Sort es bastante complejo en términos de código y extenso, con varios bucles y condiciones. Sin embargo, resulta ser el mejor método para trabajar con arreglos de gran tamaño y elementos de tipo string. Además, es un método que intriga debido a que no es comúnmente aplicado en la vida diaria y, debido a su eficiencia, sólo una máquina podría realizarlo a tal velocidad.

Resultados y Discusiones:

Las características del computador utilizado para la obtención de resultados fueron las siguientes: un Lenovo LOQ 15IRH8, con una CPU Intel i7-13620H, 16GB de RAM y una tarjeta gráfica Nvidia GeForce RTX 4060 de 8 GB.

Los resultados obtenidos a continuación son bastante interesantes, ya que permiten apreciar cómo los distintos métodos de ordenación se ven afectados por la cantidad de datos. La brecha en el rendimiento entre ellos se vuelve cada vez más notoria a medida que aumenta el tamaño de los datos.

Tener en consideración que los resultados mostrados estarán en segundos

| Tamaño diccionario | Búsqueda por selección | Búsqueda Binaria | Búsqueda binaria con Radix Sort | Búsqueda binaria con Selection Sort |
| --- | --- | --- | --- | --- |
| 10K | 87.2701 | 0.3761 | 0.4157 | 0.7842 |
| 50K | 435.082 | 0.4559 | 0.5365 | 9.8631 |
| 100K | 859.855 | 0.5437 | 0.6872 | 50 |
| 200K | 1719.47 | 0.6503 | 1.0826 | 195.838 |
| 400K | 2520.08 | 0.5296 | 1.2022 | 613.213 |

Tabla N°1: Tiempos

Elaboración propia

Como podemos ver, las diferencias en los resultados son notables. A primera vista, se puede concluir que la mejor opción para la búsqueda es la combinación de búsqueda binaria con ordenación Radix Sort, mientras que la peor es la búsqueda por selección. Un dato sorprendente es el rendimiento de la búsqueda binaria con 400,000 datos, ya que tomó menos tiempo en comparación con las búsquedas en los diccionarios previos. Esto podría deberse a que los elementos a buscar pasaron por menos bucles en esta configuración.

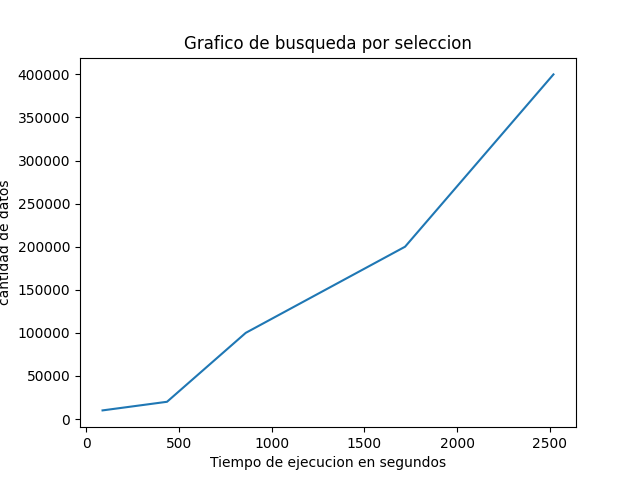


Imagen N°6: Tiempo de búsqueda por selección

Elaboración propia

Lo que se observa en el gráfico es que el algoritmo de búsqueda por selección comienza con una pendiente aceptable, pero está pendiente aumenta significativamente con el tamaño de los datos, a excepción del tramo entre 100,000 y 200,000 datos. En la última tanda de datos, con 400,000 elementos, el tiempo final alcanza varios minutos, lo que indica que este método de búsqueda es bastante ineficiente al analizar mucha cantidad de datos. El tiempo de ejecución, que siempre supera un minuto, es inaceptable para búsquedas de datos en aplicaciones prácticas.

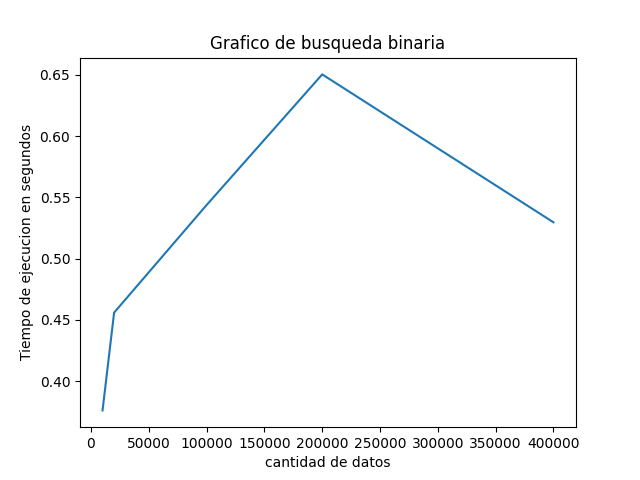


Imagen N°7: Tiempo de búsqueda binaria

Elaboración propia

En el gráfico de búsqueda binaria, se observa que el tiempo de ejecución se mantiene siempre por debajo de un segundo, lo que representa una mejora significativa en comparación con la búsqueda por selección. El tiempo máximo alcanzado es de 0.65 segundos, lo que demuestra su alta eficiencia. Además, se puede notar un decrecimiento en el tiempo de ejecución en la última tanda de datos, el cual ya ha sido explicado previamente.

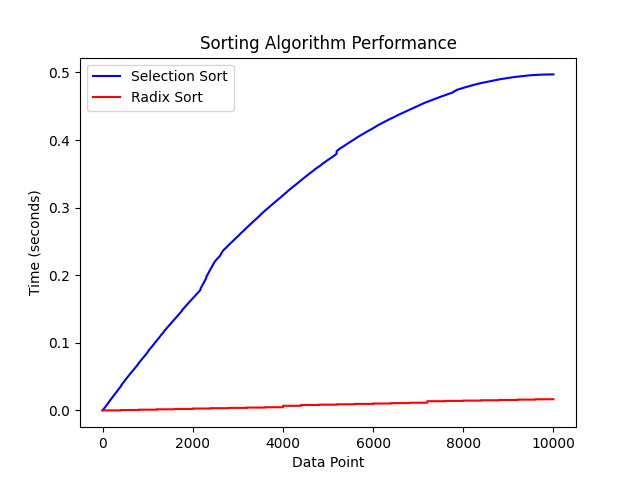


Imagen N°8: Selection v/s Radix 10K

Elaboración propia

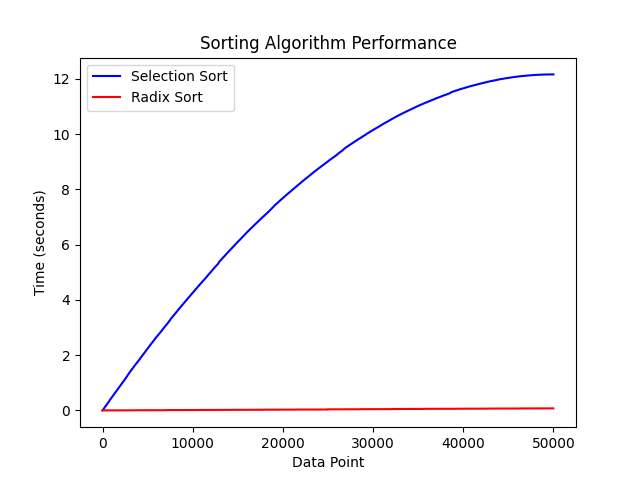


Imagen N°9: Selection v/s Radix 50K

Elaboración propia

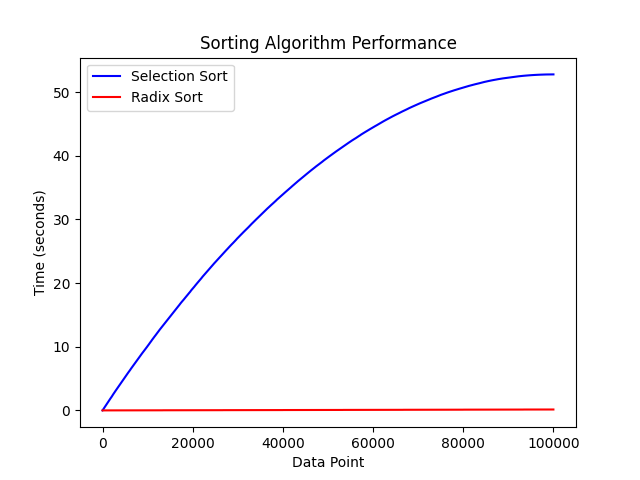


Imagen N°10: Selection v/s Radix 100K

Elaboración propia

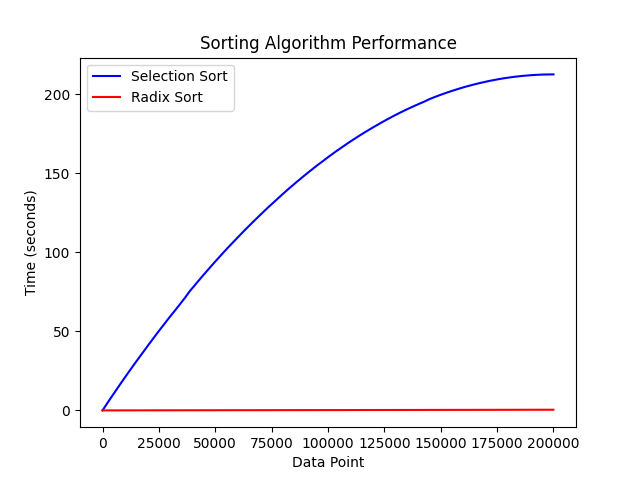


Imagen N°11: Selection v/s Radix 200K

Elaboración propia

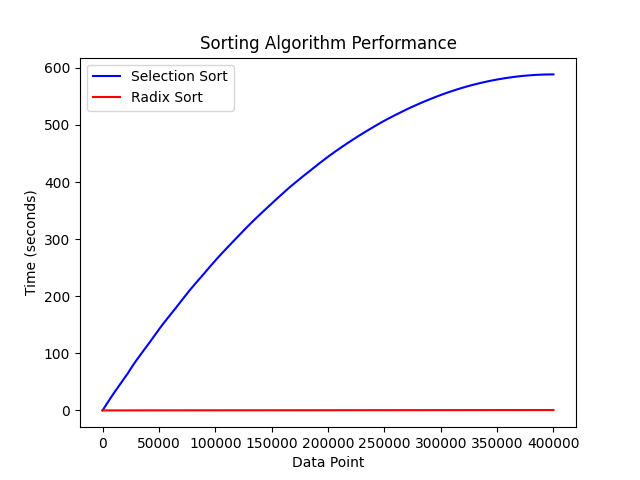


Imagen N°12: Selection v/s Radix 400K

Elaboración propia

En los gráficos, se puede apreciar que la ordenación por selección tiene una forma de parábola. Esto se debe a que, a medida que se procesan más datos, el algoritmo recorre menos camino hasta el final del arreglo. En contraste, el Radix Sort mantiene lo que parece ser una línea horizontal cerca del valor 0, ya que los tiempos de ejecución son extremadamente bajos, lo que indica que el Radix Sort tiene un tiempo casi despreciable. La forma de los gráficos es similar en todos los casos, manteniendo una pendiente consistente para ambos métodos. Esto sugiere que, al aumentar la cantidad de datos en el Selection Sort, el tiempo de ejecución aumentará significativamente debido a su naturaleza parabólica creciente. Por otro lado, el Radix Sort mantiene una linealidad, lo que lo hace muy eficiente para analizar grandes cantidades de datos, ya que el tiempo de ejecución crece de manera lineal y se mantiene cercano a 0, lo que es ideal para estos casos.

Conclusión:

En conclusión, podemos afirmar que la mejor estrategia de búsqueda es la búsqueda binaria, y el algoritmo de ordenación más eficiente es, sin duda, el Radix Sort. Para arreglos de tamaño pequeño, la diferencia de tiempo entre ambos métodos es casi imperceptible. Sin embargo, cuando se deben realizar búsquedas en bucles que manejan millones de números, como en este caso, es crucial implementar un algoritmo que optimice significativamente el tiempo de ejecución, independientemente de su complejidad para ser comprendido o programado.

Un hallazgo interesante es que el tiempo de búsqueda binaria en el último tramo fue inferior al de los dos tramos anteriores. Esto se debe a que, evidentemente, realizó menos bucles, lo que explica la optimización en el tiempo sin necesidad de modificar el código.

Además, es relevante mencionar que, para manejar grandes volúmenes de datos, es beneficioso utilizar algoritmos similares al Bucket Sort, como el Radix Sort, que tienen un comportamiento lineal en el tiempo de ordenación de datos.