Tecnológico de Monterrey

Reflexión Actividad Integradora 1

Jacobo Soffer Levy

A01028653

Campus Santa Fe

Los algoritmos de ordenamiento y búsqueda son de los más importantes y más usados en el mundo de la programación. Hacen muchas tareas mucho más sencillas y eficientes, como por ejemplo, un algoritmo de ordenamiento puede hacer más fácil encontrar un elemento dentro de un arreglo usando un algoritmo de búsqueda como Binary Sort, que solo funciona con arreglos ordenados. Puede hacer más fácil encontrar valores únicos o duplicados en un arreglo, analizar los datos que contiene, presentar la información al usuario, entre otras cosas. Los algoritmos de búsqueda nos permiten encontrar un elemento en un mar de información de forma eficiente. Varios programas dependen de algoritmos de búsqueda, incluidos muchos que usamos día a día. Aunque existe una gran probabilidad que cualquier programa relativamente funcional haga uso de estos algoritmos, algunos ejemplos muy claros son aquellos de un motor de búsqueda o la funcionalidad de búsqueda incluida en muchos programas. Estos programas dependen de estos algoritmos para darnos la información que necesitamos, y de no tenerlos, encontrar la información sería un proceso mucho más tedioso para el usuario. Aunque a primera vista es evidente su uso de algoritmos de búsqueda, los algoritmos de ordenamiento también son esenciales para que al tener la información ordenada, pueda ser más fácil y eficiente encontrarla, así como para ordenar la lista de resultados que es presentada al usuario de una manera que es útil y hace sentido.

Pero no sólo existe un algoritmo de búsqueda o un algoritmo de ordenamiento. Sino que existen varios que abordan el problema de diferentes maneras. Esto no significa que uno sea mejor que otro, sino que son aptos para casos de uso diferentes, y serán eficientes para el adecuado. Por lo tanto, es responsabilidad del programador identificar que algoritmo será mejor para el problema que esta tratando de resolver. Esta fue una consideración importante al momento de diseñar el programa de la Actividad Integradora 1, en la cual se tienen que desplegar registros entre ciertas fechas especificadas por el usuario. Estos registros vienen dados en una bitácora con más de 16,000 registros desordenados, los cuales no vienen dados en un orden particular. Es por esto que hace sentido ordenar estos registros, ya que se tienen que mostrar de esta forma al usuario, y hace mucho más eficiente la tarea de encontrar estos registros al sólo tener que encontrar el registro inicial y final dentro de un rango de fechas.

Al observar este caso, se decidió utilizar el algoritmo de QuickSort, ya que el problema tiene muchos elementos, y al resolver el problema con el enfoque de “divide y vencerás”, es un algoritmo muy efectivo para este caso que puede ser considerado promedio. Este algoritmo fue implementado de forma recursiva, y funciona particionando el arreglo, tomando un elemento como pivote (en este caso el ultimo elemento) y asegurándose de que todos los elementos más pequeños queden del lado izquierdo del pivote, y los más grandes del derecho, de esta forma generando dos particiones. Las particiones son generadas con una función auxiliar que tiene una complejidad temporal y espacial de O(n), ya que itera por todo el arreglo (o la partición que esta siendo utilizada) e intercambia cualquier elemento menor al pivote con el elemento i+1, i siendo el indice del mínimo elemento más pequeño. Cuando se termina la iteración, solo queda intercambiar el pivote con el elemento i+1. Una vez que

tenemos estas particiones el algoritmo se vuelve a llamar de forma recursiva para cada partición.
Gracias a esto sabemos que el algoritmo tiene una complejidad de $O(nlogn)$, ya que la función que participan el arreglo es llamada una vez por cada elemento del arreglo y esta tiene una complejidad de $O(n)$, y al ser un caso promedio, podemos esperar particiones balanceadas, por lo que con cada llamada a la función, el tamaño de n es dividido entre 2, lo cual es equivalente a decir $O(log(n))$. Lo que nos deja con lo siguiente: $O(n) * O(logn) = O(nlogn)$. Aunque en un caso promedio las particiones no siempre van a quedar perfectamente balanceadas (por lo que el tamaño de n no es dividido entre dos con cada partición), la complejidad del problema sigue siendo $O(nlogn)$. Esto se debe a que lo que va a cambiar con la proporción de elementos por partición es la base de $log(n)$ para representar esta nueva proporción. Sin embargo, sabemos que $log_a n = log_b n / log_b a$, por lo que llegamos a que: $log_c n = log_2 n / log_2 c$, c siendo la base del logaritmo y una constante, por lo que el resultado del logaritmo dividiendo a $log_2 n$ es una constante y es eliminado según las reglas del análisis asintótico, por lo que la complejidad temporal para el caso promedio es de $O(nlogn)$.
Un algoritmo de ordenamiento alternativo para este caso sería Merge Sort, que tiene complejidades temporales de $O(nlogn)$ para todos los casos, mientras que la complejidad temporal de QuickSort para el peor caso es de $O(n^2)$. Se opto por usar QuickSort en su lugar ya que para un caso promedio tienen la misma complejidad temporal, pero en complejidad espacial Merge Sort es $O(n)$ mientras que Quick Sort es $O(logn)$, por lo que es la opción más eficiente para este caso, en especial al tener más de 16,000 elementos que tienen que ser ordenados. De igual manera, al hacer pruebas de tiempo para ambos algoritmos de ordenamiento en el programa, Quick Sort tuvo un promedio de tiempo de 150ms, mientras que Merge Sort tuvo un tiempo de ejecución promedio de 6500ms al ser ejecutados en el mismo ambiente, probando lo mencionado anteriormente como correcto. El código con el algoritmo de Merge Sort puede ser encontrado en el branch “merge-sort” del repositorio.
Para el algoritmo de búsqueda se decidió usar Binary Search. Esto es porque fue el algoritmo de búsqueda más eficiente de los vistos en clase, y ya que el arreglo estaba ordenado, podía ser usado. Este algoritmo funciona calculando la variable m, que es igual a la mitad del arreglo. Si A[m] == k, k siendo el valor que se busca, el algoritmo se detiene. Si A[m] > k, el algoritmo se repite, esta vez posicionando m dentro de la primera mitad del arreglo o en la segunda sí A[m] < k. Debido a que se eliminan mitades con cada ejecución de este algoritmo, su complejidad temporal es $O(logn)$.