Análisis al algoritmo Merge Sort

Alumnos: Josue Samuel Philco Puma

16 de septiembre de 2024

Resumen

El siguiente documento presenta un análisis propio del algoritmo de ordenanmiento Merge Sort basado en **Lecture 16: MergeSort proof of correctness, and running time (Doina Precup).**

1. Algoritmo Merge Sort

Merge Sort es un algoritmo de ordenamiento que sigue el enfoque de ***Divide y Vénceras***. Este algoritmo funciona de forma recursiva dividiendo el arreglo de entrada en subarreglos más pequeños, los subarreglos se van ordenando y finalmente se fusionan para obtener el arreglo ordenado.

En otras palabras, este proceso que realiza el algoritmo divide el arreglo en dos mitades, ordena cada mitad y luego fusiona las dos mitades ordenadas. Este proceso se va repitiendo hasta ordenar todo el arreglo.

2. Demostrando la Correctitud

Ahora que sabemos que es lo que hace Merge Sort, queremos saber si el algoritmo de Merge Sort funciona correctamente. Para eso, vamos a realizar un análisis de correctitud del algoritmo en sus fases: Merge y MergeSort ya que esto lo va haciendo de forma iterativa y a esto lo llamamos la ***Invariante de Bucle***.

2.1. Correctitud de la función Merge

Primeramente nos vamos a centrar en nuestro arreglo **tmp**, en cualquier iteración $\boldsymbol{k}$, suponemos que los índices en las dos partes del arreglo $\boldsymbol{a}$ son $\boldsymbol{i}$ y $\boldsymbol{j}$. Al hacer esto tenemos que nuestra invariante es:

$$tmp[k] \leq a[l], \forall l \in \{i, ..., m\}$$
$$tmp[k] \leq a[l], \forall l \in \{j, ..., q\}$$

En otras palabras, el elemento que copiamos en la posición **k** va a ser el mínimo de todos los elementos restantes. Pero la pregunta es: ¿Por qué nos ayuda esta condición? Bueno, esto nos ayuda porque en el arreglo **tmp** los elementos se van llenando de izquierda a derecha, y todos los elementos que se están quedando en el arreglo **a** se colocan más adelante, esto en las posiciones:

$$tmp[k+1], ..., tmp[q-p+1]$$

Por lo tanto, esta condición significa que, al final del proceso de *Merge*, se cumple que $tmp[k] \leq tmp[l]$ para todo $k < l$. Esto implica que, después de copiar, el arreglo **a** estará correctamente ordenado entre los índices p y q.

Ahora, tenemos que demostrar los pasos de esta invariante, estos pasos son la *inicialización*, *Mantenimiento* y *Terminación*.

- **Inicialización:** En la inicialización, debemos ver la Condición Inicial, antes de iniciar con el bucle vamos a tener dos subarreglos ordenados **a[p..m]** y **a[m+1..q]**. Las acciones que se toman antes de este bucle es configurar dos punteros:
  - **i** apuntará al primer elemento de la primera subparte **(p)**, y **j** al primer elemento de la segunda subparte **(m+1)**.

- **k** se inicia en 0 para ir llenando el arreglo temporal **tmp**.

La Condición Inicial para la invariante es que el array **tmp** esta vacío, lo que hace que se cumpla de una forma trivial (no existen elementos que estén desordenados).

- **Mantenimiento:** En el mantenimiento, durante cada iteración del bucle, el algoritmo va a ir seleccionando el menor elemento entre **a[i]** y **a[j]**. Cada elemento seleccionado va a ir siendo copiado a **tmp[k]** y el puntero de **i** y **j** va incrementando en cada subparte.
  Para demostrar que la invariante se va manteniendo, primero es en la selección del mínimo. Supongamos el caso en que $a[i] \leq a[j]$. El elemento **a[i]** es seleccionado y se va a copiar a **tmp[k]**. Debido a los subarreglos **a[p..m]** y **a[m+1..q]** están ordenados, **a[i]** es el mínimo de los elementos restantes en esas subpartes.
  El orden en **tmp** al colocar el elemento de **a[i]** en el arreglo **tmp[k]**, el invariante va a garantizar que **tmp[0..k]** va a estar ordenado y como **tmp[k-1]** es menor o igual entre **a[i]** y **a[j]**, garantiza que **tmp** va a ir en forma no decreciente.

- **Terminación:** En la terminación, cuando el bucle finaliza cuando todos los elementos de una de las dos subpartes ya han sido copiadas en **tmp**.

2.2. Correctitud de MergeSort

Ahora que comprobamos que la función Merge funciona correctamente, debemos demostrar que el algoritmo funciona correctamente, y esto lo haremos por prueba de inducción.

- **Base de la Inducción:** Para el caso base, este ocurre si nuestro arreglo de entrada es de tamaño 1 **(n = 1)**. Por lo que en definición está ordenado y simplemente el algoritmo nos va a devolver el arreglo.

- **Hipótesis de la Inducción:** Vamos a asumir que el algoritmo va a funcionar correctamente con todos los arreglos de un tamaño menor que **n**. Eso quiere decir que el algoritmo va a ordenar cualquier arreglo de tamaño **k** donde $\boldsymbol{k < n}$.

- **Paso Inductivo:** Lo que queremos demostrar es que MergeSort va a funcionar correctamente para todo tamaño menor a $\boldsymbol{n}$, lo que da a entender que también va a funcionar en un arreglo de tamaño $\boldsymbol{n}$.

  - **División del problema:** Lo que hace el algoritmo es dividir el arreglo en dos subarreglos de tamaño $\boldsymbol{n/2}$ (cuando $\boldsymbol{n}$ es impar el primer subarreglo será de tamaño $\lceil \boldsymbol{n/2} \rceil$ y el segundo subarreglo será de tamaño $\lfloor \boldsymbol{n/2} \rfloor$).
  - **Llamadas recursivas:** Según nuestra hipótesis, estamos asumiendo que cada llamada recursiva está ordenando los dos subarreglos correctamente, ya que su tamaño en menor que $\boldsymbol{n}$.
  - **Merge:** Cuando ambos subarreglos estén ordenados por las llamadas recursivas, la función Merge los va a combinar en un solo arreglo combinado.

Este proceso nos garantiza que el algoritmo de MergeSort funciona correctamente. Cuando se diseña algoritmos recursivos, debe **poner tu fe en la recursión**, asumiendo que funcionará y especificar el procedimiento que va a seguir.

3. Análisis de la Complejidad

Como se definió anteriormente, el algoritmo Merge Sort sigue el enfoque de ***divide y vencerás***, por lo que vamos a analizar la complejidad de la función Merge y del algoritmo completo.

3.1. Complejidad de Tiempo y Espacio de Merge

Como analizamos la función Merge, este recorre parte del arreglo que esta recibiendo de forma secuencial y luego lo copia. Entonces la complejidad para este paso es de $\boldsymbol{O(q-p+1)}$. Veamos un ejemplo para especificar que es $\boldsymbol{p}$ y $\boldsymbol{q}$:

- **Ejemplo:** Supongamos que tenemos este arreglo **[2, 4, 5, 7, 1, 3, 6, 8]**. Entonces primero hacemos la división del arreglo original:

  - La **primera mitad** es desde el índice **p** hasta **q**. Es decir, **p** empieza en el índice 0 que es el inicio hasta el índice 3 que sería **q** y en la **segunda mitad** empieza en **q+1** que sería el índice 4. Entonces nuestros dos subarreglos quedan de la siguiente forma:

    Primera mitad: $[2, 4, 5, 7]$
    Segunda mitad: $[1, 3, 6, 8]$

  - Entonces, en cada iteración del bucle avanza en uno de los subarreglos, y cada elemento es tocado exactamente una sola vez en el bucle.
  - Ahora, para saber porque es que se puede desglosar todo en **q-p+1** vamos a ver el arreglo original, sabemos que el arreglo tiene un total de 8 elementos lo que empieza desde el índice 0 hasta el índice 7. Entonces **p** asume el valor de 0 y **q** asume el valor de 3, y en la segunda mitad asume el valor de **q+1** que sería 4, esto se va repitiendo hasta que cada subarreglo solo contenga un solo elemento, al momento de tener eso tendríamos a **p** con el valor 0 y a **q** con el valor de 1 y al momento de hacer la fusión tendríamos un total de 2 elementos que se están fusionando y asi va sucesivamente para combinar y ordenar los subarreglos. Ahora en la fórmula es **q-p** porque cuenta cuantos elementos hay desde **p** hasta **q** y sumamos 1 para contar ambos extremos de los índices.

Entonces, ya sabemos la complejidad de tiempo para Merge va a ser de **O(q-p+1)** asegurándonos de contar todos los elementos del subarreglo al momento de fusionarlos.

Ahora veamos la complejidad espacial de Merge, este va a ser igual a la complejidad de tiempo que es **O(q-p+1)**, esto se debe porque al definir el arreglo temporal **tmp** estamos asignando un espacio adicional donde se van a combinar los elementos de los subarreglos antes de poder copiarlos nuevamente al arreglo original, pero esto trae una gran desventaja para el algoritmo Merge Sort para la memoria, esto ocurrirá cuando el arreglo de entrada sea mucho más grande. Por ejemplo, un arreglo de 10 millones de datos requiere una gran cantidad de memoria solo para poder realizar las operaciones de la fusión.

3.2. Complejidad de Merge Sort

Pensemos de forma intuitiva para saber el tiempo de complejidad de este algoritmo, ya vimos que la función Merge procede de forma secuencial sobre la parte del arreglo que recibe y lo va copiando. Merge Sort realiza dos llamadas recursivas, cada una sobre un arreglo que tiene la mitad del arreglo original.

Entonces, supongamos que toda la cantidad de tiempo que toma Merge Sort es ***T(n)*** para un arreglo de tamaño **n**. Con esto podemos definir la siguiente fórmula:

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + T_{\text{merge}}(n)$$
$$T(n) = 2T(\frac{n}{2}) + cn$$

Lo expresamos de esta forma porque hacemos una doble recursión sobre un subarreglo que es de tamaño **n/2** y sumamos un **cn** por el paso de Merge donde **c** será el número de operaciones elementales necesarias.

Para tener una mejor visión de esta fórmula vamos a desglosarlo un poco más:

$$T(n) = 2T(\frac{n}{2}) + cn$$
$$T(n) = 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn$$
$$T(n) = 2^2T(\frac{n}{4}) + 2cn$$

Esto lo podemos seguir haciendo hasta llegar al caso base que es cuando nuestro arreglo es de tamaño 1. Ahora, nos damos cuenta que dividimos el tamaño n siempre a la mitad, por lo que llegaremos a un ***T(1)*** en $\log_2 n$ pasos. Para ese momento lograremos obtener la fórmula final:

$$T(n) = 2^{\log_2 n}T(1) + cn\log_2 n$$

En la fórmula final nos damos cuenta que:

- El primer término $\mathbf{2^{\log_2 n}T(1)}$ es un **O(n)** debido a que $2^{\log_2 n}$ nos da un **n** y el **T(1)** es un constante, por eso queda como un ***O(n)***.
- El segundo término $cn\log_2 n$ es un $O(n\log n)$, esto se debe a que como **c** es una constante nos sale un **n** y el logaritmo es en **n**, por lo que nos queda el $O(n\log n)$

Hasta este punto ya se puede decir que demostramos la complejidad del algoritmo Merge Sort, pero esto que hicimos es para darnos una idea en cuanto al tiempo de ejecución. Para establecer realmente, debemos demostrarlo ahora por inducción para que se demuestra que la recurrencia se está cumpliendo. Esto lo hacemos con el objetivo de:

- Mostrar una recurrencia más simple y directa.
- Aplicar el desenrollado en un contexto más sencillo.
- Contrastar la complejidad $O(n)$ de encontrar el máximo con la complejidad $O(n\log n)$ del algoritmo.

Entonces, vamos a plantear el siguiente problema: Encontrar el número máximo de un arreglo de **n** números.

- Primero, vamos a descomponer el problema de forma recursiva, podemos dividir el problema en encontrar el máximo en los primeros **n-1** elementos.
- Segundo, vamos a denotar el tiempo requerido como ***T(n)*** para encontrar el máximo del arreglo de tamaño **n**, entonces lo podemos expresar en términos de ***T(n-1)***. Esto lo hacemos porque buscamos el máximo en los primeros ***n-1*** elementos y hacer la comparación toma un tiempo constante ***c***. Entonces nuestra recurrencia queda planteado como:

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

- Tercero, aplicamos el desenrollado a la recurrencia quedando como:

$$T(n) = T(n-2) + c + c$$
$$T(n) = T(n-3) + c + c + c$$
$$T(n) = T(n-3) + 3c$$
$$\vdots$$
$$T(n) = T(1) + c(n-1)$$

Al hacerlo, nos está llevando a una complejidad de ***O(n)***.

Entonces, podemos contrastar esto en estos dos tipos de problemas que son de ordenamiento y de encontrar el máximo:

- **Merge Sort,** obtuvimos una recurrencia más compleja y por lo tanto una solución $O(n \log n)$ que es más eficiente para grandes entradas que otros algoritmos de $O(n^2)$ pero más lento que algoritmos de costo lineal.
- **Máximo de un arreglo,** este tiene una solución directa y de forma lineal $O(n)$ que es más simple.

El contraste entre estos dos problemas resalta que diferentes tipos de problemas y algoritmos producen diferentes recurrencias y la forma de resolverlas determinan la complejidad.

4. Conclusión

Como vimos, el algoritmo Merge Sort es eficiente en términos de tiempo y nos garantiza que lo hará de forma correcta con las pruebas de la invariante al momento de fusionar los subarreglos. Pero, este algoritmo es muy costoso en cuanto a memoria debido a la creación de un espacio adicional para ordenar los datos, por lo que utilizarlo en grandes conjuntos de datos no es lo más ideal por la limitación de la memoria. Usar Merge Sort es una buena opción que otros algoritmos que tiene un costo de $O(n^2)$ pero no es mejor que otros algoritmo de costo de $O(n)$.