Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
Nicolás Lahsen (nflahsen@uc.cl)

IIC2133 — Estructuras de datos y algoritmos — 1' 2020

Ayudantía 0: Selection Sort, Insertion Sort y Merge Sort

1 Merge Sort

Merge Sort es un algoritmo de ordenación utilizando el meodo *divide and conquer*, que involucra la división recursiva del arreglo por la mitad, ordenándolas por separado y luego haciendo un *merge* de ellas.

function MergeSort(A,i,f):
    if $f - i \geq 1$ **then** ▷ Si es que puedo volver a dividir
        $m = \frac{i+f}{2}$
        $A_1 = \text{MergeSort}(A, i, \lfloor m \rfloor)$
        $A_2 = \text{MergeSort}(A, \lceil m \rceil, f)$ ▷ Ordena los 2 subarreglos recursivamente
        A[i,f] = **Merge(**A_1**,**A_2**)** ▷ Uno los subarreglos ordenadamente
        return A[i,f]
    end if
end function

La funcion *Merge* recibe como parámetro los dos subarreglos ordenados, y devuelve la unión ordenada de ambos. Puedes asumir que *Merge* es correcta y toma tiempo $\Theta(n)$.

1. Determina la correctitud del algoritmo
2. Calcula su complejidad

2 Comparación de rendimientos

Tenemos los números 1, 3, 5, 42 ordenados de las siguientes formas:

[1, 3, 5, 42]   [42, 5, 3, 1]   [3, 1, 42, 5]

Ordénelos usando los métodos de ordenación vistos en clases (Selection Sort, Insertion Sort y Merge Sort). Cuente el tiempo (en pasos) que requiere cada método.

1. Para Selection Sort, ¿Qué diferencia habría entre trabajar con listas ligadas o con arreglos?
2. Para Insertion Sort, ¿Cuál fue el peor caso? ¿Por qué?
3. Para Merge Sort, ¿En cuál hubo que hacer mayor número de comparaciones?

3 Algoritmo desconocido

*Recuperado de: *Introduction to Algorithms*, Cormen, Leiserson, Rivest & Stein.

Describe un algoritmo, de orden $\Theta\big(n{\cdot}\log(n)\big)$ que, dado un arreglo A de n enteros y otro entero cualquiera x, determine si existen en el arreglo dos enteros cuya suma sea exactamente x.

Soluciones

1 MergeSort

1.1 Correctitud

Para demostrarlo usaremos inducción.

- **Caso Base:** Consideramos el caso con un imput de $n = 1$. En este caso, el algoritmo simplemente retorna el elemento, por lo tanto, devuelve un arreglo ordenado.
- **Hipótesis Inductiva:** Consideremos que el algoritmo funciona correctamente para inputs de tamaño n o menores, es decir, retorna un arreglo ordenado.
- **Por demostrar:** El algoritmo es correcto para un input de tamaño $n + 1$.

  Primero, el algoritmo divide el input de $n + 1$ en dos arreglos de tamaño $\lfloor \frac{n+1}{2} \rfloor$ y $\lceil \frac{n+1}{2} \rceil$. Ahora, sabemos que $\lfloor \frac{n+1}{2} \rfloor$ y $\lceil \frac{n+1}{2} \rceil$ son ambos menores a $n + 1$, y por lo tanto, menores o iguales a n. Podemos ver, por la **hipótesis inductiva**, que si usamos *MergeSort* con estos arreglos como input, nos retornará un arreglo ordenado para cada uno.

  Luego, si llamamos a *merge* con ambos arreglos como input, al ser dos arreglos ordenados y considerando que el algoritmo *merge* es correcto, tenemos que retornará correctamente la lista unificada ordenada. Por lo tanto, para un imput de $n + 1$ el algoritmo funcionará. En particular, para cualquier n, el algoritmo funciona.

1.2 Complejidad

Sabemos que *MergeSort* funciona en $\Theta(1)$ para un solo elemento, y que para un input n, llamará recursivamente a *MergeSort* con inputs de largo $\lfloor \frac{n+1}{2} \rfloor$ y $\lceil \frac{n+1}{2} \rceil$, para después unirlas con *Merge*. Por lo tanto, la ecuación de recurrencia quedaría:

$$T(n) = \begin{cases} 1 & \text{if n = 1} \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & \text{if n > 1} \end{cases}$$

Para resolver esta recurrencia buscamos un k tal que $n \leq 2^k < 2n$. Se cumple que $T(n) \leq T(2^k)$, y además las funciones techo y piso de potencias de 2 son iguales (a excepción de 2^0). Podemos entonces, reescribir la recurrencia de la siguiente forma:

$$T(n) \leq T(2^k) = \begin{cases} 1 & \text{if k = 0} \\ 2^k + 2 \cdot T(2^{k-1}) & \text{if k > 0} \end{cases}$$

Expandiendo la recursión:

$$\begin{aligned} T(n) \leq T(2^k) &= 2^k + 2 \cdot [2^{(k-1)} + 2 \cdot T(2^{k-2})] && (1) \\ &= 2^k + [2^k + 2^2 \cdot T(2^{k-2})] && (2) \\ &= 2^k + 2^k + 2^2 \cdot [2^{k-2} + 2 \cdot T(2^{k-3}) && (3) \\ &= 2^k + 2^k + 2^k + 2^3 \cdot T(2^{k-3}) && (4) \\ &... && (5) \\ &= i \cdot 2^k + 2^i \cdot T(2^{k-i}) && (6) \end{aligned}$$

cuando $i = k$, por el caso base tenemos que $T(2^{k-i}) = 1$, con lo que nos queda

$$T(n) \leq k \cdot 2^k + 2^k \cdot 1$$

Ahora, tenemos que volver a nuestra variable inicial n. Por construcción de k:

$$2^k < 2n$$

Tenemos entonces que

$$T(n) \leq k \cdot 2^k + 2^k < \log(2n) \cdot 2n + 2n$$

Por lo tanto

$$T(n) \in \Theta\big(n \cdot \log(n)\big)$$

2 Comparación de rendimientos

Para una comprensión didáctica de los algoritmos, sugiero vean los siguientes videos.

- *Selection Sort*
- *Insertion Sort*
- *Merge Sort*

1. No hay ninguna diferencia. Iterar sobre la lista entera es $\Theta(n)$ tanto para arreglos como para listas ligadas, y la operación de inserción al final de una lista o arreglo es $\Theta(1)$, conociendo la posición en que se desea insertar.
2. Para Insertion Sort, el peor caso fue el segundo. Esto se debe a la cantidad de **inversiones** que debe realizar el algoritmo, alcanzando su mejor rendimiento con 0 inversiones, y el peor con $\frac{n^2-n}{2}$ inversiones, que es este caso.
3. Para Merge Sort, hubo que hacer la misma cantidad de comparaciones en todas, ya que el algoritmo primero separa el arreglo recursivamente, independiente del orden inicial, y luego hace merge de estas.

3 Algoritmo desconocido

Primero, nos conviene ordenar el arreglo, y el mejor método para esto es utilizar *MergeSort*, que funciona con $\Theta(n \cdot \log(n))$. Ahora comienza la búsqueda, y la una buena alternativa para buscar en un arreglo ordenado es utilizar *Binary Search*, que funciona con $\Theta\big(\log(n)\big)$.

Lo que debemos hacer ahora, es tomar cada elemento i de nuestro arreglo ordenado, y buscar con *Binary Search* si existe en el arreglo el valor $x - i$. De esta forma, si están simultáneamente i y $x - i$ en el arreglo, tendremos un par de elementos que suman x. Esta búsqueda la debemos realizar para cada elemento del arreglo, por lo tanto ejecutamos n veces *Binary Search*, por lo que la ultima parte del algoritmo funcionará con $\Theta(n \cdot \log(n))$. El algoritmo quedaría de la siguiente manera:

function AlgoritmoDesconocido(A, x):
    A_o = MergeSort($A, 0, n$)
    for $i \in 0, ..., n$ **do**
        if BinarySearch($A_o, x - A_o[i]$) **then**
            return `True`
        end if

    end for
    return `False`
end function