Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
Nicolás Lahsen (nflahsen@uc.cl)
Florencia Ferrer (fpferrer@uc.cl)
Lothar Droppelmann (ldroppelmann@uc.cl)

IIC2133 — Estructuras de datos y algoritmos — 2' 2020

Ayudantía 2

1 Merge Sort

Merge Sort es un algoritmo de ordenación utilizando el método *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 = MergeSort$(A, i, \lfloor m \rfloor)$
    A_2 = MergeSort$(A, \lceil m \rceil, f)$ ▷ Ordena los 2 subarreglos recursivamente
    A[i,f] = **Merge(**A_1**,**A_2**)** ▷ Uno los subarreglos ordenadamente
  end if
  return A[i,f]
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 Binary Search

Binary Search es un algoritmo de búsqueda que encuentra la posición de un valor en un arreglo ordenado.

```
function BinarySearch(A,i,f,value):
    if f < i then
        return false
    else
        mid = (f+i)//2
        if A[mid] > value then
            return BinarySearch(A, i, mid-1, value)
        else if A[mid] < value then
            return BinarySearch(A, mid+1, f, value)
        else
            return mid
        end if
    end if
end function
```

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

3 Teorema Maestro

*Basado en la explicación del curso Matemáticas Discretas IIC1253, Diéguez y Suárez, 2019.

El teorema maestro es un método matemático utilizado para calcular complejidades en algoritmos del estilo *Divide & Conquer*, en términos de la **notación asintótica**. Estos algoritmos se analizan a partir de un umbral n_0, a partir del cuál se resuelve recursivamente el problema (Desde n_0 en adelante, el problema se comporta de tal forma, mínima cantidad de elementos para dividir $= \frac{b}{b-1}$). Por lo general, estos algoritmos dividen el input en una constante b, y luego aproximan a un entero (utilizando $\lceil\ \rceil$ o $\lfloor\ \rfloor$), haciendo a_1 y a_2 llamadas recursivas en cada caso. Además, por lo general se realiza un procesamiento adicional, antes o después de las llamadas recursivas, que llamaremos $f(n)$, la cual hace $c \cdot n^d$ pasos. Siguiendo lo anterior, el teorema enuncia lo siguiente:

Si a_1, a_2, b, c, c_0, d, $\in \mathbb{R}^+$, y $b > 1$, entonces para una recurrencia de la forma:

$$T(n) = \begin{cases} c_0 & 0 \leq n < n_0 \\ a_1 \cdot T(\lceil \frac{n}{b} \rceil) + a_2 \cdot T(\lfloor \frac{n}{b} \rfloor) + c \cdot n^d & n \geq n_0 \end{cases}$$

se cumple que

$$T(n) \in \begin{cases} \Theta(n^d) & a_1 + a_2 < b^d \\ \Theta(n^d \cdot log(n)) & a_1 + a_2 = b^d \\ \Theta(n^{log_b(a_1+a_2)}) & a_1 + a_2 > b^d \end{cases}$$

Utilizando el teorema, calcule la complejidad asintótica de los siguientes algoritmos:

- Binary-Search
- MergeSort

Soluciones

1 MergeSort

1.1 Correctitud

Debemos demostrar que el algoritmo es finito y que entrega el resultado correcto.

El algoritmo es finito

Para demostrar que el algoritmo es finito, debemos demostrar que este siempre termina, independiente del largo del arreglo. Para demostrarlo podemos usar inducción:

- **Caso Base:** Consideramos el caso base con un input de $n = 1$. Se cumple que $i = 0$ y $f = 0$ inicialmente (porque hay un solo elemento), por lo que el algoritmo termina en una sola llamada.
- **Hipótesis Inductiva:** Consideremos que el algoritmo termina para inputs de tamaño n o menores.
- **Por demostrar:** El algoritmo termina 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 $n + 1$ (para todo $n > 0$), y por lo tanto, menores o iguales a n. Podemos ver, por la **hipótesis inductiva**, que si llamamos *MergeSort* con estos arreglos como input, estas llamadas terminan.

  Luego, llamamos a *merge* con ambos arreglos ordenados como input, y como asumimos que el algoritmo *merge* es correcto (y por lo tanto finito), concluimos que la llamada incial de *MergeSort* con un arreglo de $n + 1$ también termina. Por lo tanto, podemos concluir que *MergeSort* es finito.

El algoritmo entrega el resultado correcto

Para demostrarlo usaremos inducción.

- **Caso Base:** Consideramos el caso con un input 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 $\lfloor \frac{n}{2} \rfloor$ y $\lceil \frac{n}{2} \rceil$, para después unirlas con *Merge*. Por lo tanto, la ecuación de recurrencia quedaría:

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

Para resolver esta recurrencia buscamos un k tal que $n \leq 2^k < 2n$. Como sabemos que la funcion T(n) es estrictamente no decreciente, 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:

$$2^k < 2n$$

$$k < log_2(2n)$$

Tenemos entonces que

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

Por lo tanto

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

2 BinarySearch

2.1 Correctitud

Debemos demostrar que el algoritmo es finito y que entrega el resultado correcto.

El algoritmo es finito

Para demostrar que el algoritmo es finito, debemos demostrar que este siempre termina, independiente del largo del arreglo. En este caso, el algoritmo puede terminar de dos formas; retornando *False* cuando el elemento que se busca no está en el arreglo, o retornando el índice correspondiente al encontrarlo. Para demostrarlo podemos usar inducción:

- **Caso Base:** Consideramos el caso base con un input de $n = 1$. Se cumple que $i = 0$ y $f = 0$ inicialmente (porque hay un solo elemento). Si el único elemento que hay coincide con el valor buscado ($A[mid] = value$), entonces el algoritmo termina en la primera llamada y retorna mid.

  Si $A[mid] > value$, se llamará $BinarySearch(A, 0, -1, value)$, la cuál terminaria en un ciclo, ya que $f < i$, y se retorna $False$. Similarmente, si $A[mid] < value$, se llamará $BinarySearch(A, 1, 0, value)$, la cuál igualmente terminaría en un ciclo ya que $f < i$, retornando $False$. Por lo tanto $BinarySearch$ siempre termina para un input de largo 1.

- **Hipótesis Inductiva:** Consideremos que el algoritmo termina (es finito) para inputs de tamaño n o menores.

- **Por demostrar:** El algoritmo termina para un input de tamaño $n + 1$.

  Si el valor buscado coincide con el elemento en la posicion mid ($A[mid] = value$), entonces el algoritmo termina en la primera llamada y retorna mid.

  Si no es así, entonces se llamara a $BinarySearch(A, i, mid - 1, value)$ ó $BinarySearch(A, mid + 1, f, value)$, dependiendo de si $A[mid]$ es mayor o menor que $value$.

  En cualquiera de los dos casos, el largo del input sobre el cual se hace la llamada recursiva es menor o igual a $\lfloor \frac{n+1}{2} \rfloor$. Ahora, sabemos que $\lfloor \frac{n+1}{2} \rfloor$ es siempre menor a $n + 1$ (para todo $n > 0$), y por lo tanto, menor o igual a n. Entonces, por la **hipótesis inductiva**, sabemos que si llamamos *BinarySearch* con estos arreglos como input, estas llamadas terminan. Por lo tanto, concluimos que la llamada incial de *BinarySearch* con un arreglo de $n+1$ también termina. Por lo tanto, podemos concluir que el algoritmo de *BinarySearch* es finito.

El algoritmo entrega el resultado correcto

Para demostrarlo usaremos inducción. Sea n el largo del arreglo, donde $n = f - i + 1$.

- **Caso Base:** Cuando $n = 1$, el arreglo solo tiene un elemento. Si A[mid] = value, se retorna mid (mid = 0). Si no es así, se llama recursivamente con un arreglo vacio ($f < i$) y la función retorna $false$.

- **Hipótesis Inductiva:** Consideramos que para arreglos de tamaño $< k$, con $k > 1$, BinarySearch retorna mid si $value$ está en el arreglo y $false$ en otro caso.

- **Por demostrar:** El algoritmo en un arreglo de k elementos retorna mid si $value$ está en el arreglo y $false$ en otro caso.
  Al comparar A[mid] con $value$ hay 3 casos:
  CASO 1: A[mid] $< value$: como A es un arreglo ordenado, $value$ debe estar entre las posiciones $i + 1$ y f, lo que se busca de manera recursiva y retorna mid o $false$ de manera correcta por la hipótesis inductiva.
  CASO 2: A[mid] $> value$: similar al caso de arriba, $value$ debe estar entre las posiciones i y $mid - 1$, que se busca correctamente de forma recursiva.
  CASO 3: A[mid] $= value$: el algoritmo retorna mid, lo cual es correcto porque $value$ claramente está en el arreglo.

2.2 Complejidad

Sabemos que *Binary Search* funciona en $\Theta(1)$ para un solo elemento, y que para un input n, llamará recursivamente a *BinarySearch* con inputs de largo no mayores a $\lfloor \frac{n}{2} \rfloor$. Por lo tanto, la ecuación de recurrencia quedaría:

$$T(n) \leq \begin{cases} 1 & \text{if n} = 1 \\ T\Big(\lfloor \frac{n}{2} \rfloor\Big) + 1 & \text{if n} > 1 \end{cases}$$

Pasos de la recursión:

$$T(n) \leq T\Big(\lfloor \frac{n}{2} \rfloor\Big) + 1 \tag{7}$$

$$T(n) \leq T\Big(\lfloor \frac{n}{4} \rfloor\Big) + 2 \tag{8}$$

$$T(n) \leq T\Big(\lfloor \frac{n}{8} \rfloor\Big) + 3 \tag{9}$$

$$... \tag{10}$$

$$T(n) \leq T\Big(\lfloor \frac{n}{2^i} \rfloor\Big) + i \tag{11}$$

Para resolver esta recurrencia buscamos un k tal que $n \leq 2^k < 2n$. Como sabemos que la función T(n) es estrictamente no decreciente, se cumple que $T(n) \leq T(2^k)$.

Cuando $i = k$, por el caso base tenemos:

$$T(n) \leq T(2^k) \leq T(1) + k \tag{12}$$

$$T(n) \leq T(2^k) \leq 1 + k \tag{13}$$

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

$$2^k < 2n$$

$$k < log_2(2n)$$

Tenemos entonces que

$$T(n) \leq T(2^k) \leq 1 + k < 1 + log_2(2n)$$

Por lo tanto

$$T(n) \in O\big(\log(n)\big)$$

3 Teorema Maestro

3.1 Binary-Search

En binary-search, la recursión es

$$T(n) \leq T(\lceil \frac{n}{2} \rceil) + 1$$

por lo tanto, tenemos que $a_1 + a_2 = 1$, $b = 2$ y $d = 0$, por lo tanto $a_1 + a_2 = b^d$. Es decir, estamos en el segundo caso, por lo que la complejidad del algoritmo es $\theta(n^0 \cdot log(n)) = \theta(log(n))$

3.2 MergeSort

En mergesort, la recursión es

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + c \cdot n$$

por lo tanto, tenemos que $a_1 + a_2 = 2$, $b = 2$ y $d = 1$, por lo tanto $a_1 + a_2 = b^d$. Es decir, estamos en el segundo caso, por lo que la complejidad del algoritmo es $\theta(n^1 \cdot log(n)) = \theta(n \cdot log(n))$