Ayudantia I1

Divide and Conquer

Divide and Conquer:

- Estrategia algorítmica que nos permite dividir problemas en **N** subproblemas de menor o igual dificultad

- Cuando usar divide and conquer:
  - Cuando resolver un problema iterativamente es muy caro*
  - Cuando la dificultad del problema disminuye en cada división.
  - Cuando se los solicitamos en una interrogación 👀

- Un ejemplo claro es Merge y Quick sort.

* OJO, La recursión es cara en la vida real, por lo que si se mantiene el runtime en contexto de un algoritmo. Deberíamos intentar optar por la implementación iterativa

Divide and Conquer: Ejemplo 1. Fibonacci

```
def f(n):
    a, b = 0, 1
    for i in 0...n do
        a, b = b, a+b
    return a
```

Iterativo:
+ Eficiente
- Facil de implementar

```
def f(n):
    if n < 2: return n
    return f(n-1) + f(n-2)
```

Recursivo:
- Eficiente
+ Facil de implementar

Divide and Conquer: Ejemplo 1. Fibonacci

Existirá una forma eficiente y fácil de implementar?

Divide and Conquer: Ejemplo 1. Fibonacci

Existirá una forma eficiente y fácil de implementar?

Spoiler: **SI**, lo veremos en el último capítulo del curso

$\boldsymbol{MergeSort}$ utiliza la estrategia "dividir para conquistar" dividiendo los datos en 2 y luego resolviendo el problema recursivamente. Considera una variante de $\boldsymbol{MergeSort}$ que divide los datos en 3 y los ordena recursivamente, para luego combinar todo en un arreglo ordenado usando una variante de $\boldsymbol{Merge}$ que recibe 3 listas.

a. Escribe la recurrencia $\boldsymbol{T(n)}$ del tiempo que toma este nuevo algoritmo para un arreglo de $\boldsymbol{n}$ datos. ¿Cuál es su complejidad, en notación asintótica?

b. Generaliza esta recurrencia a $\boldsymbol{T(n,k)}$ para la variante de $\boldsymbol{MergeSort}$ que divida los datos en $\boldsymbol{k}$. ¿Cuál es la complejidad de este algoritmo en función de $\boldsymbol{n}$ y $\boldsymbol{k}$? Considera que la cantidad de pasos que toma $\boldsymbol{Merge}$ para $\boldsymbol{k}$ listas ordenadas, de $\boldsymbol{n}$ elementos en su totalidad, es $\boldsymbol{n \cdot \log_2(k)}$. Por ejemplo, si $\boldsymbol{k = 2}$, $\boldsymbol{Merge}$ toma $\boldsymbol{n}$ pasos, ya que $\boldsymbol{\log_2(2) = 1}$.

   Finalmente, ¿Qué sucede con la complejidad del algoritmo cuando $\boldsymbol{k}$ tiende a $\boldsymbol{n}$?

A.

$$T(n) = \begin{cases} 1 & \text{if n} = 1 \\ T\left(\lceil \frac{n}{3} \rceil\right) + T\left(\lfloor \frac{n}{3} \rfloor\right) + T\left(n - \lceil \frac{n}{3} \rceil - \lfloor \frac{n}{3} \rfloor\right) + n & \text{if n} > 1 \end{cases}$$

$$T(n) \leq \begin{cases} 1 & \text{if n} = 1 \\ 3 * T\left(\lceil \frac{n}{3} \rceil\right) + n & \text{if n} > 1 \end{cases}$$

A.

$$T(n) = \begin{cases} 1 & \text{if n} = 1 \\ T\left(\lceil \frac{n}{3} \rceil\right) + T\left(\lfloor \frac{n}{3} \rfloor\right) + T\left(n - \lceil \frac{n}{3} \rceil - \lfloor \frac{n}{3} \rfloor\right) + n & \text{if n} > 1 \end{cases}$$

$$T(n) \leq \begin{cases} 1 & \text{if n} = 1 \\ 3 * T\left(\lceil \frac{n}{3} \rceil\right) + n & \text{if n} > 1 \end{cases}$$

k tal que $n \leq 3^k < 3n$

$$T(n) \leq T(3^k) = \begin{cases} 1 & \text{if k} = 0 \\ 3^k + 3 \cdot T(3^{k-1}) & \text{if k} > 0 \end{cases}$$

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

A.

$$\text{cuando } i = k \longrightarrow T(3^{k-i}) = 1$$

$$T(n) \leq k \cdot 3^k + 3^k \cdot 1$$

$$3^k < 3n$$

$$T(n) \leq k \cdot 3^k + 3^k < \log_3(3n) \cdot 3n + 3n$$

$$T(n) \in \mathcal{O}\big(n \cdot \log_3(n)\big) = \mathcal{O}\big(n \cdot \log(n)\big)$$

B.

$$T(n,k) \leq \underbrace{log_2(k) \cdot n}_{\text{Costo de realizar merge para k arreglos ordenados}} + \overbrace{T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right) + ... + T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)}$$

Y para n= 1

$$T(1,k) = 1$$

Ahora bien, esto es equivalente a decir

$$T(n,k) \leq log_2(k) \cdot n + k \cdot T\left(\left\lceil \frac{n}{k} \right\rceil, k\right)$$

Si se reemplaza n por $n \leq k^y < k \cdot n$ quedara

$$T(n,k) \leq T(k^y,k) = log_2(k) \cdot k^y + k \cdot T\left(k^{y-1}, k\right)$$

Y de manera recursiva quedara

$$T(k^y,k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot T\left(k^{y-2}, k\right))$$

B.

Quedando finalmente

$$T(k^y, k) = log_2(k) \cdot k^y + k \cdot (log_2(k) \cdot k^{y-1} + k \cdot (log_2(k) \cdot k^{y-2} + ... + (k^{y-y} \cdot log_2(k) + k \cdot T\,(1, k))))$$

Que en otras palabras es

$$T(k^y, k) = y \cdot k^y \cdot log_2(k) + k^y$$

Y por la condicion que se establecio en la definición de k^y, notar que

$$k^y < k \cdot n / log_k$$

$$y < log_k(k \cdot n) = \frac{log(kn)}{log(k)}$$

Por tanto quedara

$$T(n, k) \leq T(k^y) < \left(\frac{log_2(n)}{log_2(k)} + 1\right) \cdot n \cdot k \cdot log_2(k) + n \cdot k$$

Reordenando

$$T(n, k) < log_2(n) \cdot n \cdot k + n \cdot k \cdot (log_2(k) + 1)$$

A partir de esto se puede concluir que

B.

$$T(n,k) \in O(k \cdot n \cdot log(n \cdot k))$$

Para el caso de la complejidad del algoritmo para el caso que k tienda a n, es claro que la complejidad tendera a converger a $O(n \cdot log(n))$. Es claro si se reemplaza en la ecuación de recursión T(n,n).

a) ¿Qué valor de q (o j) devuelve `partition` cuando todos los elementos del arreglo `A[p..r]` son iguales? Justifica.

a) ¿Qué valor de q (o j) devuelve `partition` cuando todos los elementos del arreglo `A[p..r]` son iguales? Justifica.

Respuesta

Devuelve $q = (p+r-1)/2$. Los `while`'s internos no se ejecutan, ya que los elementos del arreglo son iguales al pivote, no mayores ni menores. Por lo tanto, el efecto del `while` externo es que en cada iteración `j` se incrementa en 1 y `k` se decrementa en 1 (y el `exchange` dentro del `while` intercambia de posición dos elementos iguales). Así, `j` se "encuentra" con `k` en la mitad del rango `p`, ..., `r`−1 (ya que `k` parte en `r`−1), y en ese punto se detiene el `while`. El `exchange` fuera del while no cambia el valor de `j`.

b) ¿Cuál es el tiempo de ejecución de `quickSort` cuando todos los elementos del arreglo `A` son iguales? Justifica.

b) ¿Cuál es el tiempo de ejecución de `quickSort` cuando todos los elementos del arreglo `A` son iguales? Justifica.

Respuesta

$O(n \log n)$. Como vimos en clase, cuando `partition` produce dos particiones de similar tamaño, `quickSort` tiene su mejor desempeño. Y de **a)**, este es el caso cuando todos los elementos del arreglo son iguales.

c) Considera la siguiente versión de `quickSort`, que sólo hace la primera llamada recursiva:

```
quickSort'(A, p, r)
while (p < r)
   q = partition(A, p, r)
   quickSort'(A, p, q-1)
   p = q+1
```

Explica por qué `quickSort'` ordena correctamente el arreglo `A`.

c) Considera la siguiente versión de `quickSort`, que sólo hace la primera llamada recursiva:

```
quickSort'(A, p, r)
while (p < r)
   q = partition(A, p, r)
   quickSort'(A, p, q-1)
   p = q+1
```

Explica por qué `quickSort'` ordena correctamente el arreglo `A`.

Respuesta

`quickSort'` hace la misma partición que `quickSort`, y luego también se llama recursivamente con parámetros `A`, `p` y `q-1`. La diferencia es que en este punto `quickSort` se llama recursivamente por segunda vez con parámetros `A`, `q+1` y `r`; en cambio, `quickSort'` asigna `p = q+1` y hace otra iteración del `while`. Sin embargo, esto ejecuta las mismas operaciones que la segunda llamada recursiva, con `A`, `q+1` y `r`, ya que en ambos casos `A` y `r` tienen los mismos valores que antes y `p` tiene el valor antiguo de `q+1`.

Recomendaciones de Estudio

1. Mas que estudiar los algoritmos, estudien las estrategias algoritmicas por detras
2. Estudiar bien cómo y porqué funcionan bien Quicksort y Mergesort
3. Suelten la mano con correctitud. El compilado esta lleno de ejercicios

The End

Recomendaciones de Estudio

1. Mas que estudiar los algoritmos, estudien las estrategias algoritmicas por detras
2. Estudiar bien cómo y porqué funcionan bien Quicksort y Mergesort
3. Suelten la mano con correctitud. El compilado esta lleno de ejercicios

Cual compilado? Parece que alguien no lee los anuncios

Recomendaciones de Estudio

1. Mas que estudiar los algoritmos, estudien las estrategias algoritmicas por detras
2. Estudiar bien cómo y porqué funcionan bien Quicksort y Mergesort
3. Suelten la mano con correctitud. El compilado esta lleno de ejercicios

Cual compilado? Parece que alguien no lee los anuncios

4. Ademas de ver las clases, es recomendable estudiar el libro guia (Cormen)