Ayudantía 5 - Algoritmos y Complejidad
Dividir y Conquistar

Sassy complexes

1. Introducción

Como todas las otras tecnicas que hemos visto y vamos a ver en el ramo, **Dividir y Conquistar** consiste en transformar un problema dificil en uno facil, aquí lo que hacemos es **dividir** un problema grande y dificil en muchos problemas pequeños y faciles, **juntamos** las soluciones a los problemas faciles para encontrar la solucion al problema dificil. Dos pasos, el nombre es autoexplicativo:

Dividir: el problema P de tamaño n en k subproblemas de tamaño $n_1, n_2, \ldots, n_k$

Conquistar: el problema mezclando las soluciones de los subproblemas.

> "if you have to choose between fight against a single giant monster rat or an army of cute hamsters, what do you choose?"
>
> Mas que una tecnica de diseño de algoritmos, es una filosofia de resolucion de problemas (y de vida)

2. Mergesort[1]

El ejemplo usual de algoritmo *divide & conquer* es **mergesort**, principalmente por que es facil de analizar. Mergesort tiene la estructura de un dividir y conquistar bien explicita, piense en dos operaciones: *Dividir y Ordenar* y *Mezclar y conquistar*. Veamos la mezcla:

Listing 1: Operación Mezcla

```
1  def merge(A, B):
2      ls = [] # Lista vacia para guardar el resultado
3      while len(A) > 0 and len(B) > 0 : # Mientras A y B tengan elementos
4          if A[0] < B[0]: # head(A) < head(B)
5              ls += [ A[0] ] # append head(A) al resultado
6              del A[0] # quita head(A) de A
7          else:
8              # lo mismo pero con B cuando head(A) > head(B)
9              ls += [ B[0] ]
10             del B[0]
11     ls += A[:] # annade lo que sobro de A
12     ls += B[:] # y lo que sobro de B
13     return ls # si A y B estaban ordenados, esto esta en orden ascendente
```

[1] https://youtu.be/es2T6KY45cA

Complejidad de la operación mezcla: $\Theta(A_n + B_n)$

Veamos ahora la otra mitad, ¿que pasa cuando dividimos y dividimos una lista? llegamos a listas de largo 1, y ¿como ordenamos una lista de largo 1? ya está ordenada.

En resumen:

Caso Base: Una lista vacia o con un unico elemento ya esta ordenada

Recursion: Rompe la lista en mitades, aplica el algoritmo a cada lado y luego mezcla los resultados

Listing 2: Dividiendo y conquistando

```
1 def mergesort(ls):
2     if len(ls) <=1:
3         return ls
4     left = mergesort(A[:len(A)//2])
5     right = mergesort(A[len(A)//2:])
6
7     return merge(left, right)
```

¿Cual es su Complejidad?

$$\begin{aligned}
&T(0) = \Theta(1)\\
&T(1) = \Theta(1)\\
&T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n)
\end{aligned}$$

Puede simplificar el analisis haciendo un par de supuestos: Primero, suponga que el largo del arreglo siempre sera potencias de 2, ($n = 1,2,4,8,16,\ldots$), asi nos podemos quitar los floors y los ceiling. Segundo, suponga que $\Theta(1)$ esta acotado superiormente por una constante c, es mas facil de mirar. Estos supuestos simplifican el analisis y no cambian mucho el resultado.

$$\begin{aligned}
&T(1) \leq c\\
&T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n
\end{aligned}$$

Oh! una recurrencia!

3. Recurrencias

3.1. Matematica fea

Empiece a expandir como si tuviera mucho tiempo libre:

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn\\
&\leq 2\left(2T\left(\frac{\frac{n}{2}}{2}\right) + \frac{cn}{2}\right) + cn\\
&= 4T\left(\frac{n}{4}\right) + 2cn\\
&\leq 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn\\
&= 8T\left(\frac{n}{8}\right) + 3cn
\end{aligned}$$

Ya ve para donde va esto?

$$\mathrm{T}(n) \leq 2^k \mathrm{T}\left(\frac{n}{2^k}\right) + kcn$$

Si $n = 1, 2, 4, 8, 16, \ldots = 2^k$ (gracias al supuesto que tomamos)

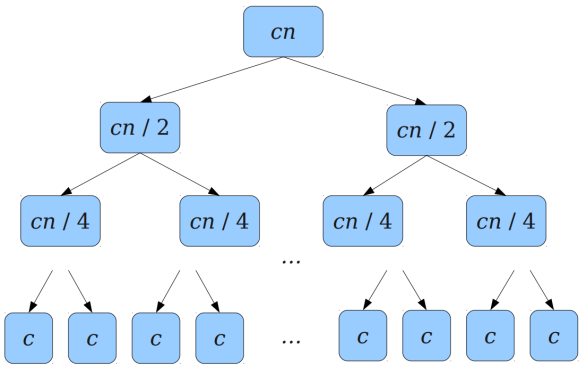
$$n = 2^k \Rightarrow k = \log_2(n) \Rightarrow n/2^k = 1$$

Entonces:

$$\begin{aligned}
\mathrm{T}(n) &\leq 2^k \mathrm{T}\left(\frac{n}{2^k}\right) + kcn \\
&= 2^{\log_2(n)} \mathrm{T}(1) + \log_2(n)cn \\
&= n\mathrm{T}(1) + \log_2(n)cn \\
&= \mathrm{T}(1)n + \log_2(n)cn \\
&\leq cn + \log_2(n)cn \\
\mathrm{T}(n) &= O\left(n\log_2(n)\right)
\end{aligned}$$

3.2. Arboles de recursion

Si no le gusta hacer matematica fea, puede dibujar un arbol:



En cada nivel se hacen $O(n)$ comparaciones, y hay $\log(n)$ niveles. Complejidad total: $O(n\log(n))$

3.3. Funciones generativas

Si se acuerda de *estructuras discretas* puede usar funciones generativas, cosa que no vamos a hacer por que tenemos algo mejor.[2]

[2] La realidad es que ninguno de sus ayudantes se acuerda de esto, pero usted si esta obligado a saberlo, nosotros no

4. Teorema Maestro

Existe una formula magica (y maestra) para resolver recursiones del tipo $F(n) = aF(n/b)+G(n)$. Se llama ***Teorema Maestro***. La idea es comparar $n^{\log_b(a)}$ (cantidad de hojas del árbol) con n^d (costo de mezcla).

4.1. Teorema Maestro *(Version "facil")* (corregida)[3]

Para la recurrencia generica (**Y SOLO ESTA RECURRENCIA GENERICA**[4])

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

n es el tamaño de la entrada, a es el número de subproblemas de la recursión y b es el factor por el que el tamaño del subproblema se reduce en cada recursión. $a > 0, b > 1, n > 0$, $f(n)$ es el costo de unir soluciones.

Definimos $c_{crit} = \log_b(a)$ Entonces:

$$T(n) = \begin{cases} \Theta(n^{c_{crit}}) & \text{if } f(n) = O(n^c) \text{ where } c < c_{crit} \\ \Theta\left(n^{c_{crit}} \log^{\kappa+1}(n)\right) & \text{if } f(n) = \Theta\left(n^{c_{crit}} \log^{\kappa}(n)\right) \text{ for any } \kappa \geq 0 \\ \Theta\left(f(n)\right) & \text{if } f(n) = \Omega(n^c) \text{ where } c > c_{crit} \end{cases}$$

- El primer caso corresponte a los problemas **Leaf Heavy**, donde la complejidad de resolver los subproblemas es *significativamente mayor* que la mezcla.
- El segundo caso corresponte a los problemas **Balanced**, la complejidad de la mezcla y de la resolución son comparables.
- El tercer caso corresponte a los problemas **Root Heavy**, donde la complejidad de unir soluciones es *significativamente mayor* que resolverlas por separado.

Por ejemplo: **mergesort**:

$$\mathtt{T}(0) = \Theta(1)$$
$$\mathtt{T}(1) = \Theta(1)$$
$$\mathtt{T}(n) \leq 2\mathtt{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$$

- ¿Cuales son a, b, y $f(n)$? $a = 2, \quad b = 2, \quad f(n) = \Theta(n^1 \log^0 n), \quad c = 1, \quad \kappa = 0$
- ¿Cuanto es $c_{crit} = \log_b a$? $\log_2 2 = 1$

Por *Master Theorem* estamos en el segundo caso: *Balanced Recurrence*, el costo de la mezcla es comparable con el de resolver subproblemas, $T(n) = \Theta\left(n\log(n)\right)$

4.2. Version *"Normal"*

La versión que aparece en el apunte del profe y en el torpedo oficial es la solución completa a la recurrencia $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ usando funciones generatrices, revise el apunte:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^d) \text{ para } d < \log_b a \\ \Theta(n^{\log_b a}) & f(n) = \Theta(n^{\log_b a} \log^{\kappa} n) \text{ con } \kappa < -1 \\ \Theta(n^{\log_b a} \log\log n) & f(n) = \Theta(n^{\log_b a} \log^{\kappa} n) \text{ con } \kappa = -1 \\ \Theta(n^{\log_b a} \log^{\kappa+1} n) & f(n) = \Theta(n^{\log_b a} \log^{\kappa} n) \text{ con } \kappa > -1 \\ \Theta(f(n)) & f(n) = \Omega(n^d) \text{ con } d > \log_b a \text{ y } af(n/b) < kf(n) \text{ para } n \text{ grande con } k < 1 \end{cases}$$

[3] En ayudantias historicas se mostraba una versión que fallaba para el mismo mergesort y no concordaba con la versión del profe, con ayuda de Wikipedia < 3 la arreglamos

[4] Trate de resolver QuickSort usando MasterTheorem, para que se de cuenta de que no es tan maestro

4.3. Version *"Real"*

Una version mas general del teorema maestro es el **Teorema de Akra-Bazzi**, cosa que ya no se pregunta por que es muy dificil.

5. Ejercicios

1. Para cierto problema dispone de dos algoritmos:

   A: Resuelve el problema dividiéndolo en cinco problemas de la mitad del tamaño, los resuelve recursivamente y combina las soluciones en tiempo lineal.

   R: Leaf-heavy: $\Theta(n^{\log_2 5})$

   C: Divide el problema de tamaño n en nueve problemas de tamaño $\frac{n}{3}$, resuelve los problemas recursivamente y combina las soluciones en tiempo $O(n^2)$.

   R: Balanced problem: $\Theta(n^2 \log n)$

   Elija.

2. Le dan tres algoritmos más:

   D: Cada problema se divide en 8 subproblemas, cada uno con la mitad de los datos de entrada. Unir los subproblemas toma un tiempo de $1000n^2$

   R: Leaf-heavy: $\Theta\left(n^3\right)$ $\left(Soluciónexacta: 1001n^3 - 1000n^2\right)$

   E: Cada problema se divide por la mitad, y la unión toma un tiempo de $10n$

   R: Balanced problem: $\Theta\left(n \log n\right)$ $\left(Soluciónexacta: n + 10n\log_2 n\right)$

   F: Cada problema se divide por la mitad, pero unir las soluciones toma n^2

   R: Root-heavy: $\Theta\left(n^2\right)$ $\left(Soluciónexacta: 2n^2 - n\right)$

3. Dos más:

   G: Ordena los datos dados, divide en tres problemas de un tercio de tamaño y combina en tiempo constante.

   R: Ordenar toma $\Theta(n \log n)$ por que sí[5]. La recurrencia es $T(n) = 3T(n/3) + \Theta(n \log n)$ Balanced ($k = 1$): $\Theta\left(n \log^2 n\right)$

   H: Divide el problema de tamaño n en nueve de tamaño $n/4$ y combina las soluciones en tiempo $O(n^2)$.

   R: La recurrencia es $T(n) = 9T(n/4) + \Theta(n^2)$. Root-heavy: $\Theta\left(n^2\right)$

4. Dos más:

   I: Divido por la mitad, pero solo me quedo con un subproblema, el resto es trabajo constante.

[5] Es un "supuesto educado", un estudiante de algoritmos debería saber por qué $n \log n$ es razonable aquí

R: Apply Master theorem case $c = \log_b a$, where $a = 1, b = 2, c = 0, k = 0$ Balanced-case: $\Theta(\log n)$

J: Divide en dos subproblemas, union constante.

R: Apply Master theorem case $c < \log_b a$ where $a = 2, b = 2, c = 0$ Leaf-heavy: $\Theta(n)$

5. Complejidad de este programa?

Listing 3: StoogeSort

```
1   function stoogesort(A, i = 0, j = L.lenght - 1){
2       if ( A[i] > A[j] ){
3           swap(A[i], A[j])
4       }
5       if ( (j - i + 1) > 2 ) {
6           k = floor( (j - i + 1) / 3 )
7           stoogesort(A, i, j - k)
8           stoogesort(A, i + k, j  )
9           stoogesort(A, i, j - k)
10      return L
11      }
12  }
```

R: Llamemos $S(n)$ al tiempo que toma con un arreglo de largo n. Del programa vemos que las tres llamadas son sobre arreglos de 2/3 del largo, además de trabajo básicamente constante. Esto da la recurrencia:

$$S(n) = 3S\left(2\frac{n}{3}\right) + \Theta(1)$$

$a = 3$, $b = 3/2$, $f(n) = \Theta(1)$, $c = 0$, $c_{crit} = \log_{3/2} 3 = 2{,}709 > c$, Leaf-heavy: $S(n) = \Theta\left(n^{2{,}709}\right)$