Tarea 2 - Análisis de Algoritmos

Cristian Ignacio Reyna Méndez

20 de Agosto del 2024

1 Complejidad Asintótica - Algoritmo recursivo

1.1 Árbol de recursión

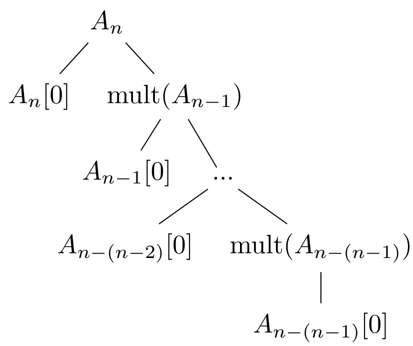
Sea el algoritmo de la tarea 1 ”mult”:

```
1 def mult(A):
2     if(len(A) == 1):
3         return A[0]
4     else:
5         return A[0] * mult(A[1:])
```

El cuál devuelve dado una lista de ”n” elementos, la multiplicación de estos de manera recursiva; es decir, multiplica A[0] * A[1] * ... * A[n]. Analizando el comportamiento, nos damos cuenta que en la primera vuelta por líneas 2-3, comparará primero si la lista es de longitud 1; para posteriormente devolver la cabeza de la lista y multiplicar por el algoritmo nuevamente, o dicho de otra manera, ahora realizaremos el algoritmo para A[1:], esto por la línea 5.

De este modo, tendremos en proceso la multiplicación pendiente de A[0] * mult(A[:1]), y a su vez la multiplicación de la cabeza de A_{n-1} (por términos prácticos, llamaremos A_{n-1} al resto de la lista de A, es decir, sin la cabeza, mismo caso para A_{n-2} y así sucesivamente) * mult(A_{n-1}[:1]) hasta llegar al caso base.

Por lo que, el árbol de recursión para una lista de longitud ”n” se vería tal que:



1.2 Costo interno y complejidad

Ahora, sea una lista A con "n" elementos, argumentaremos línea por línea el costo de cada una de ellas:

Línea 2) Realizamos una comparación sobre la longitud de A, la cuál tendría *O(1)*.

Línea 3) En caso de que sea verdadero, retornamos la cabeza de la lista A, el cuál su costo sería *O(1)*.

Línea 5) Caso contrario, realizamos una multiplicación sobre el primer elemento de la lista A y volvemos a realizar el algoritmo mult, ahora sobre la lista A_{n-1} y así sucesivamente.

A tener en cuenta: En la misma línea 5; encontramos un proceso muy curioso, el cual es "A[1:]". Este mismo tiene 2 variantes, las cuales son de orden constante y orden de n; por practicidad, tomamos en este caso que el proceso es constante, ya que en caso contrario, estaríamos realizando un proceso de copiado en la lista A, en la lista A_{n-1} y así sucesivamente, la cuál se realizaría n-1 veces y al mismo tiempo realizaríamos n-1 veces llamadas recursivas. Por lo tanto en caso contrario de tomarlo como orden de n, su complejidad asintótica sería de $(n-1)^2$, es decir $O(n^2)$.

De este modo, mientras que la lista A_{n-1} está ejecutandose, también lo hará la lista A_{n-2} hasta $A_{n-(n-1)}$. Sin embargo, las llamadas recursivas serán de *O(n-1)* ya que en el caso base, que es cuando la lista consta de 1 elemento, este mismo es el que retorna; así finalmente se pueden realizar las multiplicaciones las cuales son de *O(1)*. Por lo tanto, lo que pesa en el orden de este algoritmo o mejor dicho, su complejidad asintótica fueron las llamadas recursivas, por lo cual es de *O(n)*.

2 Complejidad Asintótica - Algoritmo iterativo

2.1 Costo interno y complejidad

Sea el algoritmo "multi" de la tarea 1:

```
1 def multi(A):
2     acc = 1
3     for v in A:
4         acc *= v
5     return acc
```

Vamos a realizar el análisis para verificar su complejidad asintótica en el caso de una lista de "n" elementos:

Línea 2) Primeramente, realizamos la asignación de una variable llamada acc, la cuál nos servirá para acumular la multiplicación de los elementos de la lista. Se ejecuta 1 sola vez.

Línea 3-4) Se ejecutaran un total de n veces de donde solamente se realizarán operaciones elementales, las cuáles solo son multiplicaciones de los elementos de la lista de n elementos por la variable acc.

De este modo, en el peor y mejor de los casos, solamente estaremos multiplicando la variable acc por los elementos de la lista "n" veces, siendo un costo de $O(n)$, por lo tanto su complejidad será de ese mismo orden.