Cálculo de T(n) de algoritmos recursivos

Objetivo

La idea es disponer de un documento resumido que sirva de repaso y referencia.

Deducción de T(n)

Frente a un algoritmo recursivo, lo primero a resolver es la función de recurrencia para T(n)
Los siguientes algoritmos a analizar, fueron extraídos de la práctica.

i)

```
int rec2 (int n) {
    if (n <= 1)
        rec2 = 1;
    else {
       rec2 =  2 * rec2 (n-1);
    }
    return rec2;
}
```

ii)

```
int rec1 (int n) {
    if (n <= 1)
        rec1 = 1;
    else  {
      rec 1 =  rec1 (n-1) + rec1 (n-1);
    }
    return rec1;
}
```

iii)

```
int  f (int n) {
   if  ( n = 0 )
      f = 0;
   else {
      if ( n = 1 )
          f = 1;
      else {
          f = f (n-2) x f (n-2);
      }
   }
   return f;
}
```

Todos obviamente tienen uno o dos casos bases y uno o dos llamados recursivos. Para resolver las funciones recurrentes es muy importante tener presente cuantas llamadas recursivas se realizan en cada caso ya que cada una tiene un costo computacional, y esto es precisamente lo que estamos calculando:

```
caso i)      rec2 =  2 * rec2 (n-1);                    1 llamada     →   rec2(n-1)

caso ii)     rec 1 =  rec1 (n-1) + rec1 (n-1);          2 llamadas    →   2 * rec1(n-1)

caso iii)    f = f (n-2) x f (n-2);                     2 llamadas    →   2 * f(n-2)
```

Por otro lado, las operaciones que se realizan con las llamadas recursivas no es importante, ya que si sumo dos llamadas recursivas o las multiplico o elevo una a la otra debería ser indistinto en el calculo del T(n) puesto que todas representan (desde el punto de vista del costo computacional del algoritmo) operaciones constantes.
Por lo antes expuesto podemos concluir que los T(n) para los algoritmos presentados son:

```
Caso i)      T(n)   =      c1                  si n <= 1
                           T(n-1) + c2         si n > 1
```

```
Caso ii)     T(n)   =      c1                  si n <= 1
                           2T(n-1) + c2        si n > 1
```

```
Caso iii)    T(n)   =      c1                  si n = 0
                           c2                  si n = 1
                           2T(n-2) + c3        si n > 1
```

En general, si en el algoritmo, tenemos los siguientes llamados recursivos, entonces en el T(n) se escribirían así:

```
f1 = f1(n-1)                      T(n) = T(n-1) + c1    (la constante implica la asignación)
f2 = f2(n-1) ^ 2                  T(n) = T(n-1) + c1    (la constante también incluye el ^2)
f3 = f3(n-2) * f3(n-2)            T(n) = 2T(n-2) + c1   (en este caso implica el = y el *)
```

Resolución de T(n)

Supongamos que tenemos que resolver el caso ii)

```
int rec1 (int n) {
    if (n <= 1)
        rec1 = 1;
    else  {
      rec 1 =  rec1 (n-1) + rec1 (n-1);
    }
    return rec1;
}
```

Entonces como vimos anteriormente, la función de recurrencia es:

```
Caso ii)     T(n)   =      c1                  si n <= 1
                           2T(n-1) + c2        si n > 1
```

La idea para resolverlo es suponer que se tiene que resolver T(n) para un "n" muy grande de modo que siempre se utiliza el llamado recursivo. Es muy importante también, tener presente la cantidad de veces que se utilizó la función recurrente, de modo de poder calcular el caso genérico.
Veamos:

1

T(n) = **2T(n-1) + c2** (es el resultado de utilizar la recursión 1 vez)

Continuamos resolviendo aplicando nuevamente la recursión:

T(n) = 2 (2T(n-1-1) + c2) + c2 (notar que lo marcado es el resultado de T(n-1) que era lo único que se puede resolver del paso anterior.

Además, lo marcado es el parámetro de la función recurrente. Esto es muy importante porque si mi función recurrente fuese de la forma:

```
T(n)  =     c1                  si n <= 1
            2T(n-1) + (n/2)^2   si n > 1
```

al momento de resolver la función recurrente, habría que reemplazar en cada lugar donde aparece "n", por el parámetro de la función. Por ejemplo:

```
T(n-1) =  2T(n-1 - 1) + ((n-1)/2)^2  =  2T(n-2) + (n-1)^2/4
```

Continuando con nuestro ejercicio original, tenemos que:

$T(n) = 2T(n-1)+c2$

$T(n) = 2(2T(n-1-1)+c2)+c2 = 4T(n-2)+3c2$

$T(n) = 4(2T(n-3)+c2)+3c2 = 8T(n-3)+4c2+3c2 = 8T(n-3)+7c2$

Podemos seguir resolviendo la recursión, suponiendo un "n" grande, hasta tanto nos demos cuenta como sería el caso general que se dará en el paso i
Por lo tanto, de la observación de los pasos anteriores podemos deducir que:

$T(n) = 2^i T(n-i)+(2^i-1)c2$

Ahora, nuestro T(n) depende de dos variables, de "n" y de "i". Además está expresado en forma recursiva. Para eliminar la recursión, podemos pensar en que es lo que pasa cuando $T(n-i)$ alcanza su caso base, es decir cuando $n-i=1$ o sea que $i=n-1$

Con esta igualdad, además de deshacernos del llamado recursivo, como efecto colateral, tambien podemos reemplazar las "i" en función de "n". Veamos como quedaría:

$T(n)=2^i T(n-i)+(2^i-1)c2$ dado que $i=n-1$ entonces, reemplazando nos queda:

$T(n)=2^{(n-1)}T(n-(n-1))+(2^{(n-1)}-1)c2=2^{(n-1)}T(1)+2^{(n-1)}c2-c2$

Utilizando el caso base que establece que $T(1)=c1$ entonces T(n) queda

$$T(n)=2^{(n-1)}c1+2^{(n-1)}c2-c2=2^{(n-1)}(c1+c2)-c2$$

Por lo tanto, $T(n)=2^{(n-1)}(c1+c2)-c2$ es $O(2^n)$

Demostración del orden de ejecución (BigO)

Para demostrarlo podemos encontrar un par de constantes “c0” y “n0” de modo que validen la definición de BigO: $T(n) \leq c0 * 2^n \forall\, n \geq n0$

Para ello lo mas fácil es separar en términos T(n) y analizarlos por separado

$2^{(n-1)}(c1+c2) \leq (c1+c2) * 2^n \forall\, n \geq 1$ Notar que utilizamos las constantes c0=c1+c2 y n0=1

Para el otro termino de T(n) hacemos el mismo cálculo:

$-c2 \leq 2^n \forall\, n \geq 0$ En este caso c0=1 y n0=0

Si ahora sumamos ambos lados de la desigualdad, la suma de lo de la izquierda debería ser menor o igual que la suma de lo de la derecha.

$2^{(n-1)}(c1+c2)+(-c2) \leq (c1+c2) * 2^n + 2^n \forall\, n \geq 1$ Notar que n0 sale de la intersección de los n0 utilizados anteriormente.

Resolviendo un poco nos queda que:

$T(n) \leq (c1+c2+1) * 2^n \forall\, n \geq 1$ por lo tanto, encontramos c0=c1+c2+1 y n0=1 tal que $T(n) \leq c0 * 2^n \forall\, n \geq n0$, por lo tanto T(n) es $O(2^n)$

Nota1: También hubiese sido valido acotar solamente la parte positiva del T(n).

Demostración del orden de ejecución alternativa (Lopital)

Alternativamente se podría haber demostrado el orden, analizando $\lim n \to \infty \frac{T(n)}{2^n}$

En caso que el resultado sea una constante, nos muestra que el orden de crecimiento coincide. Si por el contrario, el limite tiende a infinito, significa que nuestro T(n) esta creciendo mas rápidamente que el orden propuesto. Finalmente, si tiende a cero, significa que nuestro T(n) crece mas lento que el orden propuesto y al ser el denominador del cociente mas grande cada vez, el resultado es cero.

$$\lim n \to \infty \frac{T(n)}{2^n} = \lim n \to \infty \frac{2^{(n-1)}(c1+c2)+(-c2)}{2^n} = \lim n \to \infty \frac{2^{(n-1)}(c1+c2)}{2^n} + \frac{-c2}{2^n}$$

$$\lim n \to \infty \frac{2^{(n-1)}(c1+c2)}{2^n} + \lim n \to \infty \frac{-c2}{2^n} = \lim n \to \infty \frac{c1+c2}{2} + \lim n \to \infty \frac{-c2}{2^n} = \frac{c1+c2}{2} + 0$$

Como el resultado es una constante, $\frac{c1+c2}{2}$, entonces podemos decir que la función $T(n)$ tiene un crecimiento menor o igual que 2^n . Por lo tanto T(n) es $O(2^n)$.