UNIVERSIDAD SIMÓN BOLÍVAR
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Resuelto por:
Christopher Gómez

Tarea 3: Divide y vencerás

1. Para las siguientes recurrencias, use el teorema maestro visto en clase para hallar una fórmula cerrada para $T(n)$:

   a) $T(n) = 3T(\frac{n}{4}) + \frac{3(n^2-1)}{2}$

   Se tiene que $a = 3$, $b = 4$ y $g(n) = \frac{3(n^2-1)}{2} \in \Theta(n^2)$. Así, $k = 2$ y $a < b^k$, por lo que $T(n) \in \Theta(n^2)$.

   b) $T(n) = 7T(\frac{n}{7}) + 2n - 3$

   Se tiene que $a = 7$, $b = 7$ y $g(n) = 2n - 3 \in \Theta(n)$. Así, $k = 1$ y $a = b^k$, por lo que $T(n) \in \Theta(n \log n)$.

   c) $T(n) = 5T(\frac{n}{2}) + 2n$

   Se tiene que $a = 5$, $b = 2$ y $g(n) = 2n \in \Theta(n)$. Así, $k = 1$ y $a > b^k$ por lo que $T(n) \in \Theta(n^{\log_2 5}) \approx \Theta(n^{2.321})$.

   d) $T(n) = \frac{\sum_{i=1}^{n}(T(\frac{n}{2})+i)}{n}$

   Si desarrolamos la recurrencia, obtenemos:

$$\begin{aligned} T(n) &= \frac{\sum_{i=1}^{n}(T(\frac{n}{2}) + i)}{n} \\ &= \frac{nT\left(\frac{n}{2}\right) + \sum_{i=1}^{n} i}{n} \\ &= T\left(\frac{n}{2}\right) + \frac{n(n+1)}{2n} \\ &= T\left(\frac{n}{2}\right) + \frac{n+1}{2} \end{aligned}$$

   De esta forma, se tiene que $a = 1$, $b = 2$ y $g(n) = \frac{n+1}{2} \in \Theta(n)$. Así, $k = 1$ y $a < b^k$, por lo que $T(n) \in \Theta(n)$.

2. Definimos los números de Perrin con la siguiente recurrencia:

$$P(n) = \begin{cases} 3 & \text{si } n = 0 \\ 0 & \text{si } n = 1 \\ 2 & \text{si } n = 2 \\ P(n-2) + P(n-3) & \text{si } 3 \leq n \end{cases}$$

Se desea que diseñe un algoritmo que tome tiempo $\Theta(\log n)$ para hallar el valor de $P(n)$. Puede suponer que todas las operaciones aritméticas involucradas (sumas, multiplicaciones, etc.) toman tiempo $\Theta(1)$.

De manera similar a como se construye la matriz de Fibonacci, se construye una matriz que nos permita calcular el valor de $P(n)$ en tiempo logarítmico, esta matriz es:

$$P = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Veamos que:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} P(n-1) \\ P(n-2) \\ P(n-3) \end{pmatrix} = \begin{pmatrix} P(n-2) + P(n-3) \\ P(n-1) \\ P(n-2) \end{pmatrix} = \begin{pmatrix} P(n) \\ P(n-1) \\ P(n-2) \end{pmatrix}$$

Sea V el vector con los casos base de la recurrencia, es decir:

$$V = \begin{pmatrix} P(2) \\ P(1) \\ P(0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 3 \end{pmatrix}$$

Se tiene que podemos obtener el valor de cada $P(n)$ a partir de $n = 3$ haciendo multiplicaciones sucesivas por la izquierda de la matriz P, con lo que llegamos a la relación:

$$P^{(n-2)}V = \begin{pmatrix} P(n) \\ P(n-1) \\ P(n-2) \end{pmatrix}$$

Así, podemos obtener el valor de $P(n)$ en tiempo logarítmico utilizando potenciación rápida de matrices, en la que se ejecutan $\Theta(\log n)$ multiplicaciones de matrices, que se pueden tomar como $\Theta(1)$ porque el tamaño de las matrices a multiplicar no crece con n (aunque luego en implementación no sea del todo cierto cuando los números se hacen muy grandes). El algoritmo entonces es el siguiente:

```
def perrin(n):
    if n == 0:
        return 3
    if n == 1:
        return 0
    if n == 2:
        return 2

    P = [[0, 1, 1],
         [1, 0, 0],
         [0, 1, 0]]

    P = P^(n-2) # Usando exponenciación rápida

    return 2P[0, 0] + 3P[0, 2]
```

Donde el último paso viene de que solo nos interesa el primer valor de $P^{(n-2)}V$, por lo que no necesitamos calcular toda la multiplicación.

Una implementación de este algoritmo en Go se puede encontrar aquí, además de un benchmark que compara el tiempo de ejecución de este algoritmo con el de una implementación iterativa ingenua que corre en tiempo lineal. Se puede ver cómo el algoritmo ingenuo resulta más rápido para n hasta 100, pero luego el de exponenciación rápida se distancia hasta ser alrededor de 60 veces más rápido para $n = 10^6$.

3. Dada una cadena de caracteres $S[1..n]$ compuesta únicamente de paréntesis que abren y que cierran, queremos un árbol de segmentos para hacer consultas de la forma $maxBP(i, j)$ que debe devolver la longitud de la sub-cadena (no necesariamente contigua) bien parentizada más larga, comprendida en el rango $[i..j]$.

   Por ejemplo, si $S = [(,),), (, (,),), (, (,),), (]$ y solicitamos $maxBP(3, 10)$, el rango corresponde a $S' =], (, (,),), (, (,)]$.

   La sub-cadena bien parentizada de mayor longitud en S' sería $[(, (,),)]$ (de longitud 4).

   Describa cómo sería el proceso con el cuál construiría el valor precalculado de cada nodo, incluyendo el caso base (las hojas) y el caso recursivo (nodos intermedios).

   Para construir el árbol de segmentos, cada nodo (representando un rango de valores) estará asociado a una tupla de 3 enteros (l, r, wp), donde l y r representan el número de paréntesis que abren y cierran sin balancear en el rango, respectivamente, y wp es la longitud de la sub-cadena bien parentizada más larga en el rango.

   Con esta información, la cadena de entrada se divide por la mitad y se construyen los nodos hijos con los valores correspondientes a los rangos que representan. Se presentan dos casos:

   - **Caso base:** Si el rango es de tamaño 1, el nodo hoja tendrá valores $(1, 0, 0)$ si el caracter es `'('`, y $(0, 1, 0)$ si es `')'`.
   - **Caso recursivo:** Los valores de los nodos hijos se combinan para obtener el valor del nodo padre de la siguiente manera:
     - $l = l_1 + l_2 - \text{mín}(l_1, r_2)$
     - $r = r_1 + r_2 - \text{mín}(l_1, r_2)$
     - $wp = wp_1 + wp_2 + 2\,\text{mín}(l_1, r_2)$
     - Donde (l_1, r_1, wp_1) y (l_2, r_2, wp_2) son los valores de los nodos izquierdo y derecho, y (l, r, wp) es el valor del nodo padre.

     El cálculo descrito viene justificado por que el padre toma los paréntesis que abren del lado izquierdo y los que cierran del lado derecho, añade a wp aquellos que se balancean al unir los dos rangos, y resta a l y r aquellos que siguen quedando sin balancear luego de la unión.

   Con esto, el árbol se construye en tiempo lineal (porque la cantidad de nodos está acotada por $4n$) y se pueden hacer consultas de $maxBP(i, j)$ en tiempo logarítmico, utilizando el mismo caso recursivo para ir bajando por el árbol y combinando los valores de los nodos que se encuentran en el camino de la raíz a los nodos que representan el rango $[i..j]$. El resultado de la consulta será el valor de wp calculado.

   Una implementación en Haskell de la construcción del árbol de segmentos y su consulta se puede encontrar aquí.