Universidad Simón Bolívar
División de Ciencias Físicas y Matemáticas
Departamento de Computación y Tecnología de la Información
CI5651: Diseño de Algoritmos I

Profesor: Ricardo Monascal. **Estudiante**: Astrid Alvarado, 18-10938.

Tarea 3 (9 %):

1. **Pregunta 1** Se tiene que las recurrencias $T(n)$ para ser aplicadas con el *Teorema Maestro* deben presentar la siguiente forma:

$$T(n) = aT\left(\frac{n}{b}\right) + g(n),\ \text{donde } g(n) \in O(n^k)$$

Por lo que se tendrá:

a) $a = 3, b = 4$ y $g(n) \in O(n^2)$. Dado que $a < b^k$ $(3 < 16)$, según el *Teorema Maestro* la recursión se reduce a $\Theta(n^2)$

b) $a = 5, b = 5$ y $g(n) \in O(n)$. Dado que $a = b^k$ $(5 = 5)$, según el *Teorema Maestro* la recursión se reduce a $\Theta(n\ log(n))$

c) $a = 5, b = 2$ y $g(n) \in O(n)$. Dado que $a > b^k$ $(5 > 2)$, según el *Teorema Maestro* la recursión se reduce a $\Theta(n^{log_2 5})$

d) Para esta recursión, se debe hacer la siguiente manipulación:

$$\begin{aligned}
T(n) &= \frac{\sum_{i=1}^{n}(T(\frac{n}{2}) + i)}{n} \\
&= \frac{nT(\frac{n}{2}) + \sum_{i=1}^{n} i}{n} \leftarrow T \text{ es constante en la sumatoria} \\
&= \frac{\cancel{n}T(\frac{n}{2})}{\cancel{n}} + \frac{\frac{n(n+1)}{2}}{n} \leftarrow \text{ separación de fracciones; suma notable} \\
&= T\left(\frac{n}{2}\right) + \frac{\cancel{n}(n+1)}{2\cancel{n}} \\
&= T\left(\frac{n}{2}\right) + \frac{(n+1)}{2}
\end{aligned}$$

Con esto, se identifica $a = 1, b = 2$ y $g(n) \in O(n)$. Dado que $a < b^k$ $(1 < 2)$, según el *Teorema Maestro* la recursión se reduce a $\Theta(n)$

2. **Pregunta 2**
Este problema se resolverá de forma similar a la solución propuesta para *Fibonacci*. Para esto, se tiene que buscar la matriz P y el vector V_P acorde a la secuencia de Perrin. Note que:

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

Por lo tanto, la matriz P será

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

Y V_P será el vector con los casos base, de la forma:

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

Con esto, se puede ver que:

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

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} P(3) \\ P(2) \\ P(1) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 2 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} P(4) \\ P(3) \\ P(2) \end{pmatrix}$$

$$\vdots$$

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

Gracias a esta generalización, se puede calcular el $n-$ésimo número de Perrin al exponenciar la matriz P a $n-2$ y posteriormente multiplicarlo con el vector V_P. Para el algoritmo, se aprovechará el uso de accesos a arreglos, sumas y multiplicaciones (que toman tiempo $O(1)$) para hacer una función auxiliar `multiply`, que multiplicará dos matrices de tamaño 3×3:

function MULTIPLY(A: matriz 3×3, B: matriz 3×3)
  $C \leftarrow$ matriz(3×3)

  $C[0][0] \leftarrow A[0][0] * B[0][0] + A[0][1] * B[1][0] + A[0][2] * B[2][0]$
  $C[0][1] \leftarrow A[0][0] * B[0][1] + A[0][1] * B[1][1] + A[0][2] * B[2][1]$
  $C[0][2] \leftarrow A[0][0] * B[0][2] + A[0][1] * B[1][2] + A[0][2] * B[2][2]$
  $C[1][0] \leftarrow A[1][0] * B[0][0] + A[1][1] * B[1][0] + A[1][2] * B[2][0]$
  $C[1][1] \leftarrow A[1][0] * B[0][1] + A[1][1] * B[1][1] + A[1][2] * B[2][1]$
  $C[1][2] \leftarrow A[1][0] * B[0][2] + A[1][1] * B[1][2] + A[1][2] * B[2][2]$
  $C[2][0] \leftarrow A[2][0] * B[0][0] + A[2][1] * B[1][0] + A[2][2] * B[2][0]$
  $C[2][1] \leftarrow A[2][0] * B[0][1] + A[2][1] * B[1][1] + A[2][2] * B[2][1]$
  $C[2][2] \leftarrow A[2][0] * B[0][2] + A[2][1] * B[1][2] + A[2][2] * B[2][2]$

  return C
end function

Con esta función, se podrá entonces modificar el algoritmo visto en clase de exponenciación, pero aplicado a matrices 3×3:

```
function EXP_MATRIX(A: matriz 3 × 3, n: entero)
    if n = 0 then
        return Id                                   ▷ Devolver matriz identidad 3 × 3
    end if

    half ← EXP_MATRIX(A, n/2)

    if n es par then
        return MULTIPLY(half, half)
    else
        temp ← MULTIPLY(half, half)
        return MULTIPLY(temp, A)
    end if
end function
```

Con estas dos funciones auxiliares, se podrá entonces proponer el siguiente algoritmo para calcular el $n-$ésimo número de Perrin:

```
function PERRIN(n: entero)
    V_P ← (2, 0, 3)
    if n = 0 then
        return V_P[2]
    end if

    if n = 1 then
        return V_P[1]
    end if

    if n = 2 then
        return V_P[0]
    end if

    P ← ((0, 1, 1), (1, 0, 0), (0, 1, 0))
    result ← EXP_MATRIX(P, n − 2)

    return V_P[0] ∗ result[0][0] + V_P[1] ∗ result[0][1] + V_P[2] ∗ result[0][2]
end function
```

De esta forma, el algoritmo propuesto toma $O(log(n-2))$ gracias al algoritmo de exponenciación de matrices.

Este programa fue implementado en `C++` y se encuentra disponible en el siguente enlace. Leer `README.md` para la ejecución del mismo.

3. **Preguntа 3** Para armar el árbol de segmentos, se debe tener los siguientes atributos para cada nodo:

   - `balanced`: la cantidad de pares de paréntesis correctamente cerrados, es decir, la sub-cadena bien parentizada más larga para el rango representado por el nodo.
   - `open`: la cantidad de paréntesis que abren (`"("`) que no han sido emparejados en el rango representado por el nodo.

- `close`: la cantidad de paréntesis que cierran (`")"`) que no han sido emparejados en el rango representado por el nodo.

Con esta información, los valores de los nodos vendrán siendo:

i. **Caso base** (nodos hoja)
Los nodos hoja deberán representar el paréntesis encontrado en la posición i de la cadena de paréntesis. En este caso, un nodo de rango $[i, j]$ con $i = j$ tendrá como valores:
   - `balanced` $= 0$ dado que un solo paréntesis no es una sub-cadena bien parentizada
   - `open` $= 1$ y `close`$=0$ si el paréntesis en la posición i es `"("`
   - `open` $= 0$ y `close`$=1$ si el paréntesis en la posición i es `")"`

ii. **Caso recursivo** (nodos intermedios)
Los nodos intermedios se deberán calcular a partir de sus nodos hijos, y para actualizar los atributos se debe calcular la cantidad de nuevos pares formados como sigue:

```
new_pairs = min(left_node.open, right_node.close)
```

Note que este cálculo se realiza tomando en cuenta la cantidad de `"("` que están en el nodo izquierdo y la cantidad de `")"` que están en el nodo derecho, dado que así se forman sub-cadenas bien parentizadas. Además, se utiliza `min` para garantizar que no se intenta formar más pares de los que permite el recurso más escaso de un lado o del otro. A partir de estos pares formados, se procede a calcular los atributos como:
   - `balanced = left_node.balanced + right_node.balanced + 2*new_pairs`
   - `open = left_node.open + right_node.open - new_pairs`
   - `close = left_node.close + right_node.close - new_pairs`

Así, se tiene acumulados los valores y además se tiene en cuenta los nuevos pares formados en el rango $[i, j]$ con $i < j$. Para mayor facilidad en la explicación siguiente, a este proceso descrito se le denominará *combinar* nodos.

Con esto, se podrá construir el árbol de segmentos correspondiente a la candena de paréntesis de la entrada. Luego, para consultar por la longitud de la sub-cadena bien parentizada más larga en un rango determinado $[p, q]$ se deberá entonces:

- Para los casos bases:
  a) En caso de que el rango del nodo cumpla con $[i, j] = [p, q]$, se devuelve el valor de `balanced` precalculado.
  b) En caso de que el rango $[i, j]$ del nodo sea tal que $i > q$ o $j < p$ (es decir, se está fuera del rango de consulta) devolver un nodo con todos sus valores en 0 para que no aporte al los cálculos realizados al *combinar* nodos. Esto facilita el "delegar" el trabajo a uno de los nodos hijos.
- Para el caso recursivo, se delega el trabajo a ambos hijos y se *combinan* los nodos.

Este programa fue implementado en `C++` y se encuentra disponible en el siguente enlace. Leer `README.md` para la ejecución del mismo.