Resolución de Tarea 3 - Divide y Vencerás (Fecha: 16 de Octubre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 3 (9 puntos)

Indice

Pregunta 1

Teniendo en cuenta la siguiente version simplificada del **Teorema Maestro**

$$\text{Para } T(n) = aT(\frac{n}{b}) + g(n)$$

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k log(n)) & \text{si } a = b^k \\ \Theta(n^{log_b(a)}) & \text{si } a > b^k \end{cases}$$

Se tiene para los siguientes problemas que

- $T(n) = 3T(\frac{n}{4}) + \frac{7(n^2-1)}{3}$

  Tenemos $g(n) = \frac{7(n^2-1)}{3} \in O(n^2)$, por lo tanto

  $$\begin{array}{l} a = 3 \\ b = 4 \\ k = 2 \end{array} \implies 3 < 4^2 \implies T(n) \in \Theta(n^2)$$

- $T(n) = 5T(\frac{n}{5}) + 7n - 4$

  Tenemos $g(n) = 7n - 4 \in O(n)$, por lo tanto

  $$\begin{array}{l} a = 5 \\ b = 5 \\ k = 1 \end{array} \implies 5 = 5^1 \implies T(n) \in \Theta(nlog(n))$$

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

  Tenemos $g(n) = 2n \in O(n)$, por lo tanto

  $$\begin{array}{l} a = 5 \\ b = 2 \\ k = 1 \end{array} \implies 5 > 2^1 \implies T(n) \in \Theta(n^{log_2(5)})$$

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

  Manipulando esta expresión un poco, tenemos

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

  Por lo que, $g(n) = \frac{(n+1)}{2} \in O(n)$, por lo tanto

  $$\begin{array}{l} a = 1 \\ b = 2 \\ k = 1 \end{array} \implies 1 < 2^1 \implies T(n) \in \Theta(n)$$

Pregunta 2

La recurrencia de Perrin está definida como:

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

Justificación

Usaremos el principio de "divide y vencerás" para calcular la n-ésima potencia de una matriz de transición en tiempo logarítmico.

Dado que la recurrencia se define por los tres términos anteriores ($P(n-2)$ y $P(n-3)$), se necesita una matriz de transición de dimensión 3×3.

Definimos el vector de estado S_n para el paso n:

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

Para encontrar S_n a partir de S_{n-1}, necesitamos una matriz de transición M tal que $S_n = M \cdot S_{n-1}$.

Las relaciones que definen M son: 1. $P(n) = 0 \cdot P(n-1) + 1 \cdot P(n-2) + 1 \cdot P(n-3)$ 2. $P(n-1) = 1 \cdot P(n-1) + 0 \cdot P(n-2) + 0 \cdot P(n-3)$ 3. $P(n-2) = 0 \cdot P(n-1) + 1 \cdot P(n-2) + 0 \cdot P(n-3)$

La matriz de transición M es:

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

Podemos relacionar el estado S_n con el estado base S_2:

$$S_n = M^{n-2} \cdot S_2 \quad \text{para } n \geq 2$$

Donde el vector base es:

$$S_2 = \begin{pmatrix} P(2) \\ P(1) \\ P(0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 3 \end{pmatrix}$$

El tiempo de ejecución está dominado por la exponenciación de la matriz M^{n-2}. Dado que la multiplicación de dos matrices 3×3 toma tiempo $O(3^3) = O(1)$ (asumiendo que las operaciones aritméticas elementales son $O(1)$, como se establece en el problema), y la exponenciación se realiza mediante la técnica de divide y vencerás, el tiempo total es $\Theta(\log n)$.

Implementación en C++

A continuación se presenta el código completo en C++ que implementa la lógica anterior.

Utilizamos `long long` para manejar los valores de $P(n)$, ya que el problema no especifica un límite superior para n, y los números de Perrin crecen exponencialmente.

El archivo funcional se encuentra en Perrin.cpp

```cpp
typedef long long ll;
typedef vector<vector<ll>> Matrix;

// Tamaño de la matriz para la recurrencia de Perrin (3x3)
const int K = 3;

Matrix multiply(const Matrix& A, const Matrix& B) {
    Matrix C(K, vector<ll>(K, 0));
    for (int i = 0; i < K; ++i) {
        for (int j = 0; j < K; ++j) {
            for (int k = 0; k < K; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}

Matrix power(Matrix M, ll n) {
    // Matriz identidad para inicializar el resultado
    Matrix R(K, vector<ll>(K, 0));
    for (int i = 0; i < K; ++i) R[i][i] = 1;

    while (n > 0) {
        if (n & 1) R = multiply(R, M); // Si n es impar
        M = multiply(M, M);
        n >>= 1; // n = n / 2
    }
    return R;
}

ll perrin(ll n) {
    // Casos base (n=0, 1, 2)
    if (n == 0) return 3;
    if (n == 1) return 0;
    if (n == 2) return 2;
```

```
    // 1. Definir la matriz de transición M
    Matrix M = {
        {0, 1, 1},
        {1, 0, 0},
        {0, 1, 0}
    };

    // 2. Calcular M^(n-2)
    Matrix M_power = power(M, n - 2);

    // 3. Vector base S_2 = {P(2), P(1), P(0)} = {2, 0, 3}
    vector<ll> S_base = {2, 0, 3};

    // 4. Calcular S_n = M_power * S_2. P(n) es el primer elemento.
    // P(n) = M_power*P(2) + M_power*P(1) + M_power*P(0)
    ll P_n = 0;
    for (int i = 0; i < K; ++i) {
        P_n += M_power[0][i] * S_base[i];
    }

    return P_n;
}
```

Pregunta 3

El problema $maxBP(i, j)$ requiere encontrar la longitud de la subcadena bien parentizada más larga dentro del rango $S[i..j]$.

Para resolver esto cada nodo debe almacenar información que permita **maximizar el número de pares válidos formados entre sus hijos, respetando la regla de precedencia** (un `(` debe venir antes de su `)` correspondiente).

Cada nodo del Árbol de Segmentos representará un segmento de la cadena S y almacenará tres componentes esenciales:

Componente	Definición
`M (matched_pairs)`	El número de pares `()` perfectamente cerrados y maximizados dentro del subsegmento cubierto por este nodo.
`O (unmatched_open)`	El número de paréntesis de apertura `(` que quedan sin emparejar.
`C (unmatched_close)`	El número de paréntesis de cierre `)` que quedan sin emparejar.

La longitud final en un segmento es $2 \times M$.

Caso Base (Hojas)

Los nodos hoja representan un único carácter $S[i]$ de la cadena.

1. Si $S[i] = '('$:
   - `M` $= 0$
   - `O` $= 1$ (Apertura pendiente)
   - `C` $= 0$
2. Si $S[i] = ')'$:
   - `M` $= 0$
   - `O` $= 0$
   - `C` $= 1$ (Cierre pendiente)

Caso Recursivo (Nodos Intermedios)

Un nodo padre U se forma combinando la información de su hijo izquierdo L y su hijo derecho R. La operación crucial aquí es identificar los **nuevos pares** que se pueden formar en la frontera entre L y R.

1. **Cálculo de Nuevos Pares (M_{new}):** Los nuevos pares solo se pueden formar si un paréntesis abierto que sobró en el lado izquierdo ($L.O$) encuentra un paréntesis de cierre que sobró en el lado derecho ($R.C$).

$$\mathbf{M_{new}} = \min(\mathbf{L.O}, \mathbf{R.C})$$

2. **Combinación de Pares ($U.M$):** El total de pares resueltos en el nodo padre es la suma de los pares resueltos en los hijos, más los nuevos pares formados en la frontera.

$$\mathbf{U.M} = \mathbf{L.M} + \mathbf{R.M} + \mathbf{M_{new}}$$

3. **Combinación de Paréntesis Abiertos ($U.O$):** La cuenta de `O` es la suma de los abiertos de ambos hijos, menos aquellos que se utilizaron exitosamente para formar M_{new}.

$$\mathbf{U.O} = \mathbf{L.O} + \mathbf{R.O} - \mathbf{M_{new}}$$

4. **Combinación de Paréntesis Cerrados ($U.C$):** De manera simétrica, la cuenta de `C` es la suma de los cierres de ambos hijos, menos aquellos que se utilizaron para formar M_{new}.

$$\mathbf{U.C} = \mathbf{L.C} + \mathbf{R.C} - \mathbf{M_{new}}$$

Proceso de Consulta ($maxBP(i, j)$)

Para una consulta de rango $[i..j]$:

1. Se realiza un recorrido estándar del Segment Tree, combinando (utilizando la operación `Merge` descrita anteriormente) los valores de todos los nodos que cubren exactamente o parcialmente el rango $[i..j]$.
2. La función de consulta devuelve un único objeto `Node` resultante, U_{total}.
3. La longitud de la subcadena bien parentizada más larga en el rango $S[i..j]$ es simplemente $2 \times U_{total}.M$.

Dado que la construcción inicial del árbol toma tiempo $O(n)$ (tiempo lineal, ya que el *merge* es $O(1)$) y cada consulta requiere solo $O(\log n)$, este algoritmo resuelve el problema.

Implementación en C++

El archivo funcional se encuentra en seq_paren.cpp

```cpp
// Estructura para almacenar la información de subcadenas bien parentizadas
struct Node {
    int matched_pairs;   // M: Pares coincidentes totales (longitud = 2*M)
    int unmatched_open;  // O: Paréntesis abiertos sin coincidir (disponibles a la derecha)
    int unmatched_close; // C: Paréntesis cerrados sin coincidir (disponibles a la izquierda
};

class MaxBPSubsequenceSegmentTree {
private:
    int n;
    string S;
    vector<Node> tree;

    // Función auxiliar para combinar los resultados de dos nodos hijos (Left y Right)
    Node merge(const Node& L, const Node& R) {
        Node result;

        // 1. Calcular nuevas coincidencias formadas por L.O y R.C
        int new_matches = min(L.unmatched_open, R.unmatched_close);

        // 2. Total de pares coincidentes
        result.matched_pairs = L.matched_pairs + R.matched_pairs + new_matches;

        // 3. Unmatched Open: Suma de los hijos menos los que se acaban de emparejar
        result.unmatched_open = L.unmatched_open + R.unmatched_open - new_matches;

        // 4. Unmatched Close: Suma de los hijos menos los que se acaban de emparejar
        result.unmatched_close = L.unmatched_close + R.unmatched_close - new_matches;
```

```
    return result;
}

// Construcción recursiva del Segment Tree
void build(int v, int tl, int tr) {
    if (tl == tr) {
        // Caso Base: Nodo Hoja
        if (S[tl] == '(') {
            tree[v] = {0, 1, 0}; // M=0, O=1, C=0
        } else if (S[tl] == ')') {
            tree[v] = {0, 0, 1}; // M=0, O=0, C=1
        } else {
            // Si la cadena solo tiene paréntesis, esto es por seguridad
            tree[v] = {0, 0, 0};
        }
    } else {
        // Caso Recursivo: Nodos Intermedios
        int tm = (tl + tr) / 2;
        build(2 * v, tl, tm);          // Hijo izquierdo
        build(2 * v + 1, tm + 1, tr); // Hijo derecho

        // Combinar los resultados de los hijos
        tree[v] = merge(tree[2 * v], tree[2 * v + 1]);
    }
}

// Función de consulta recursiva para obtener la información de un rango [l, r]
Node query_recursive(int v, int tl, int tr, int l, int r) {
    // Inicializar un nodo nulo (cero coincidencias y cero paréntesis pendientes)
    if (l > r || tl > tr) {
        return {0, 0, 0};
    }

    if (l == tl && r == tr) {
        // El nodo actual cubre exactamente el rango de consulta [l, r]
        return tree[v];
    }

    int tm = (tl + tr) / 2;

    // Consultar y combinar los resultados de las partes que se superponen con [l, r]
    Node L_result = query_recursive(2 * v, tl, tm, l, min(r, tm));
    Node R_result = query_recursive(2 * v + 1, tm + 1, tr, max(l, tm + 1), r);

    // La combinación debe realizarse si ambos lados retornaron datos válidos
    if (L_result.matched_pairs == 0
```

```
            && L_result.unmatched_open == 0
            && L_result.unmatched_close == 0) {
            return R_result;
        }
        if (R_result.matched_pairs == 0
            && R_result.unmatched_open == 0
            && R_result.unmatched_close == 0) {
            return L_result;
        }

        return merge(L_result, R_result);
    }

public:
    // S_input es la cadena, asumimos índices base 0 internamente.
    MaxBPSubsequenceSegmentTree(const string& input_S) : S(input_S) {
        n = S.length();
        // Redimensionar para 4*n (tamaño estándar para Segment Trees)
        tree.resize(4 * n + 1);
        if (n > 0) {
            // Construir desde la raíz (v=1), cubriendo [0, n-1]
            build(1, 0, n - 1);
        }
    }

    // Función para realizar la consulta maxBP(i, j)
    // i y j se asumen como índices base 1 (como en S[1..n]).
    // Internamente usamos base 0.
    int maxBP(int i, int j) {
        if (i < 1 || j > n || i > j || n == 0) return 0;

        // Convertir a índices base 0: [i-1, j-1]
        Node result = query_recursive(1, 0, n - 1, i - 1, j - 1);

        // La longitud de la subcadena bien parentizada es 2 * M
        return 2 * result.matched_pairs;
    }
};
```