Resolución de Tarea 7 - Cadenas de Caracteres y Geometría Computacional (Fecha: 17 de Noviembre 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 7 (9 puntos)

Indice

- Resolución de Tarea 7 - Cadenas de Caracteres y Geometría Computacional (Fecha: 17 de Noviembre de 2025)
- Indice
- Pregunta 1
  - (a) Construcción del Árbol de Sufijos para w
  - (b) Construcción del Arreglo de Sufijos (SA)
  - (c) Cálculo de PLCP[k] y LCP[k]
    - 1. Cálculo de Phi
    - 2. Cálculo de PLCP[k] (Prefijo Común Permutado más Largo)
    - 3. Cálculo de LCP[i] (Prefijo Común más Largo)
- Pregunta 2
  - Implementación en C++
- Pregunta 3
  - Implementación en C++

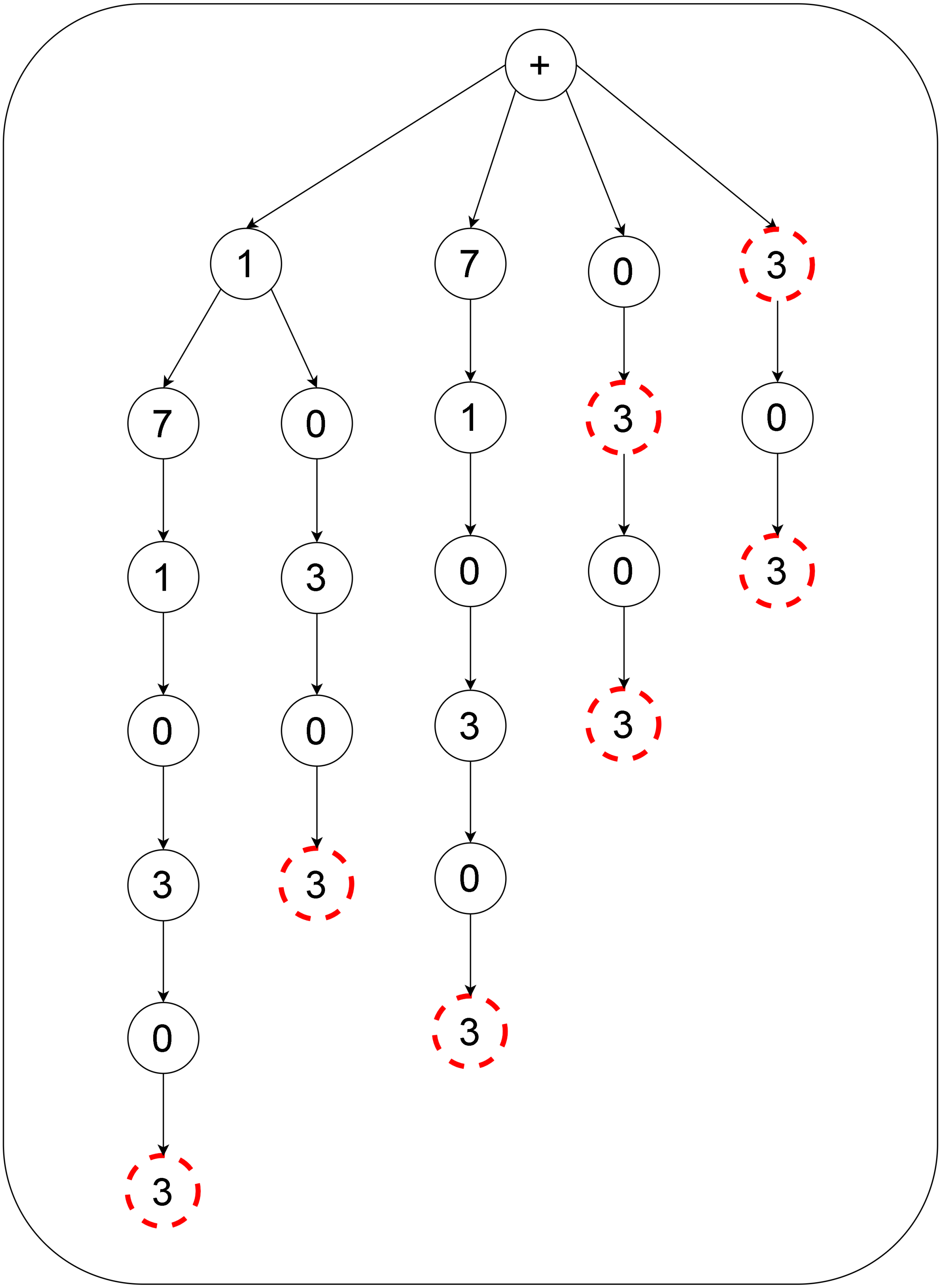
Pregunta 1

El carné a utilizar es “17-10303”. La cadena n a considerar, sin el guión, es $w =$ "1710303". La longitud de la cadena es $N = 7$.

(a) Construcción del Árbol de Sufijos para w

Un árbol de sufijos se construye a partir de la cadena w al organizar todos sus sufijos en una estructura de árbol que aprovecha los prefijos compartidos, similar a un Trie. En este caso, la estructura contendría todos los sufijos de $w =$ “1710303”:

1. 1710303
2. 710303
3. 10303
4. 0303
5. 303
6. 03
7. 3


+
1
7
0
3
7
0
1
3
0
1
3
0
3
0
0
3
3
0
3
3
0
3
3

(b) Construcción del Arreglo de Sufijos (SA)

El arreglo de sufijos (SA) almacena los índices de inicio de los sufijos de w en el orden en que aparecerían si estuvieran ordenados lexicográficamente.

Los sufijos ordenados de $w =$ "1710303" son:

Índice	Sufijo
7	λ
3	0303
5	03
2	10303
0	1710303
4	303
6	3
1	710303

El Arreglo de Sufijos (SA) es:

$$SA = [7, 3, 5, 2, 0, 4, 6, 1]$$

(c) Cálculo de PLCP[k] y LCP[k]

Para calcular los arreglos de Longitud de Prefijo Común (LCP) y Prefijo Común Permutado más Largo (PLCP), primero se define el arreglo auxiliar Φ (Phi).

1. Cálculo de Phi

El arreglo Φ almacena, para el sufijo que empieza en la posición k, el índice de inicio del sufijo que lo precede inmediatamente en el arreglo de sufijos ordenado (SA).

$$\Phi[SA[i]] = SA[i-1]$$

El caso base es $SA = 7$ (el sufijo λ), para el cual Φ se define como λ o -1.

Rank (i)	$SA[i]$ (Índice)	Sufijo $w[SA[i]..]$	$SA[i-1]$ (Predecesor)	[**SA**[**i**]]
0	7	λ	N/A	-1
1	3	0303	7	7
2	5	03	3	3
3	2	10303	5	5
4	0	1710303	2	2
5	4	303	0	0
6	6	3	4	4
7	1	710303	6	6

Arreglo Φ (Indexado por posición k de 0 a 7):

$$\Phi = [2, 6, 5, 7, 0, 3, 4, -1]$$

2. Cálculo de PLCP[k] (Prefijo Común Permutado más Largo)

$PLCP[k]$ es la longitud del prefijo común más largo entre el sufijo que comienza en k ($w[k..]$) y el sufijo que comienza en $\Phi[k]$ ($w[\Phi[k]..]$).

k	$\Phi[k]$	Sufijo $w[k..]$	Sufijo $w[\Phi[k]..]$	**PLCP[k]**
0	2	**1**710303	**1**0303	1
1	6	710303	3	0
2	5	10303	03	0
3	7	0303	λ	0
4	0	303	1710303	0
5	3	**03**	**03**03	2
6	4	**3**	**3**03	1
7	-1	λ	N/A	0

Arreglo PLCP (Indexado por posición k de 0 a 7):

$$PLCP = [1, 0, 0, 0, 0, 2, 1, 0]$$

3. Cálculo de LCP[i] (Prefijo Común más Largo)

$LCP[i]$ es la longitud del prefijo común más largo entre el sufijo de rango i ($SA[i]$) y el sufijo de rango $i-1$ ($SA[i-1]$). Se calcula utilizando el arreglo PLCP mediante la fórmula $LCP[i] = PLCP[SA[i]]$.

LCP se define como 0.

Rank (i)	$SA[i]$	Sufijo $w[SA[i]..]$	Sufijo $w[SA[i-1]..]$	$PLCP[SA[i]]$	**LCP[i]**
0	7	λ	N/A	N/A	**0**
1	3	0303	λ	$PLCP = 0$	**0**
2	5	**03**	**03**03	$PLCP = 2$	**2**
3	2	10303	03	$PLCP = 0$	**0**
4	0	**1**710303	**1**0303	$PLCP = 1$	**1**
5	4	303	1710303	$PLCP = 0$	**0**
6	6	**3**	**3**03	$PLCP = 1$	**1**
7	1	710303	3	$PLCP = 0$	**0**

Arreglo LCP (Indexado por rango i de 0 a 7):

$$LCP = [0, 0, 2, 0, 1, 0, 1, 0]$$

Pregunta 2

Este problema es una aplicación directa del algoritmo de pre-procesamiento utilizado en Knuth-Morris-Pratt (KMP), el cual calcula la longitud del Prefijo Propio más Largo (LPS, por sus siglas en inglés) que también es un sufijo para cada prefijo de la cadena.

El algoritmo KMP utiliza una tabla de saltos (b o π) que, para una cadena x, almacena en la posición i la longitud del prefijo propio más largo de $x[0..i]$ que también es un sufijo de $x[0..i]$.

Si aplicamos este algoritmo de pre-procesamiento a la cadena S, el último elemento de la tabla LPS calculará precisamente la longitud L del prefijo propio más largo de S que es también un sufijo de S.

La complejidad temporal de este pre-procesamiento es lineal, $O(N)$, lo que satisface el requerimiento de eficiencia del problema.

Implementación en C++

El archivo funcional se encuentra en lps.cpp

```cpp
// Función para calcular la longitud del Longest Proper Prefix (LPS) array para KMP.
// La complejidad es O(n).
string compute_lps_array(const std::string &S) {
    int N = S.length();
    // La subcadena más grande debe ser propia (T != S), así que si N <= 1,
    // solo podemos retornar la cadena vacía.
    if (N <= 1)
        return "";

    // lps[i] almacenará la longitud del prefijo propio más largo de S[0...i]
    // que también es un sufijo de S[0...i].
    vector<int> lps(N, 0);

    int len = 0; // Longitud del prefijo más largo anterior que es también sufijo
    int i = 1;

    // Lazo que calcula lps[i] para i = 1 a N-1
    while (i < N) {
        if (S[i] == S[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            // Desigualdad de caracteres
            // Retrocedemos a la longitud LPS anterior (len - 1)
            if (len != 0)
                len = lps[len - 1];
            else {
                // Si len es 0, no hay un prefijo común, lps[i] = 0
                lps[i] = 0;
                i++;
            }
        }
    }
    // La longitud L de la subcadena T deseada es el último elemento del arreglo LPS.
    // Esto se debe a que lps[N-1] almacena la longitud del prefijo más largo
    // de S[0...N-1] que también es un sufijo de S[0...N-1].
    int L = lps[N - 1];

    // Extraemos la subcadena de longitud L.
    return S.substr(0, L);
}
```

Pregunta 3

Este problema requiere la aplicación iterativa de un algoritmo de Cáscara Convexa (Convex Hull, CH) para determinar cuántas "capas" concéntricas de puntos existen en un conjunto dado. El algoritmo de **Graham Scan** es una técnica de **barrido geométrico** adecuada para este fin.

Graham Scan es un algoritmo eficiente para calcular el $CH(P)$ en tiempo $O(n \log n)$ para n puntos.

1. Se identifica el punto p_0 que tiene la coordenada y mínima (y la coordenada x más a la derecha en caso de empate). Este punto siempre pertenece al Convex Hull.
2. Los puntos restantes se ordenan en sentido antihorario basándose en el ángulo polar que forman con p_0. Este paso domina la complejidad con $O(n \log n)$. Las comparaciones angulares se realizan utilizando

el **producto cruz** de los vectores. El producto cruz determina si tres puntos forman un **giro a la izquierda** o un **giro a la derecha**.

3. Se utiliza una pila para construir la envolvente, asegurando que solo los puntos que mantienen consistentemente un giro a la izquierda permanezcan en ella.
   - Si el punto actual p_i forma un **giro no-izquierdo** (es decir, colineal o a la derecha) con los dos puntos superiores de la pila, el punto superior se elimina (se saca de la pila) hasta que se restablezca un giro a la izquierda.
   - El punto p_i se agrega a la pila.

Debido a que Graham Scan toma $O(n \log n)$ y se aplica para cada punto (n) tenemos entonces $O(n^2 \log n)$.

Implementación en C++

El archivo funcional se encuentra en graham_scan.cpp

```cpp
// Utilizamos long long para las coordenadas para asegurar precisión en el producto cruz.
struct Point {
    long long x, y;
    int id; // Identificador único para rastrear el punto original.
};

// Función de orientación (producto cruz). >0: CCW, <0: CW, =0: colineal
long long ccw(Point p, Point q, Point r) {
    return (q.x - p.x) * (r.y - p.y) - (q.y - p.y) * (r.x - p.x);
}

// Distancia al cuadrado (auxiliar, opcional)
long long distSq(Point p1, Point p2) {
    return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
}

// Implementación robusta del Convex Hull.
// Conserva los puntos colineales en el borde (útil para "onion peeling" / capas).
vector<Point> graham_scan(vector<Point> &P) {
    int n = P.size();
    if (n <= 2)
        return P;

    // Ordenar por x, luego por y
    sort(P.begin(), P.end(), [](const Point &a, const Point &b)
         {
        if (a.x != b.x)
            return a.x < b.x;
        return a.y < b.y; });

    vector<Point> lower, upper;

    // Construir la mitad inferior (permitimos colinealidad en el borde: pop solo si giro < 0)
    for (int i = 0; i < n; ++i) {
        while (lower.size() >= 2 && ccw(lower[lower.size() - 2], lower.back(), P[i]) <
            lower.pop_back();
        lower.push_back(P[i]);
    }

    // Construir la mitad superior
    for (int i = n - 1; i >= 0; --i) {
        while (upper.size() >= 2 && ccw(upper[upper.size() - 2], upper.back(), P[i]) <
            upper.pop_back();
        upper.push_back(P[i]);
```

```
    }

    // Concatenar lower + upper (excluyendo los últimos elementos porque se repiten)
    // Si todos los puntos son colineales, this will produce duplicates; we handle that gracefu
    lower.pop_back();
    upper.pop_back();

    vector<Point> ch = lower;
    ch.insert(ch.end(), upper.begin(), upper.end());

    return ch;
}

// 4. Función principal: Contar capas de Convex Hull
int count_layers(const vector<Point> &original_P) {
    if (original_P.size() < 3)
        return 0;

    vector<Point> P = original_P;
    int layer_count = 0;

    // Repetimos mientras queden suficientes puntos para formar una capa
    while (P.size() >= 3) {
        // 1. Calcular el Convex Hull (se modifica el orden de P)
        vector<Point> hull_points = graham_scan(P);

        // Si el CH no es un polígono, el proceso termina
        if (hull_points.size() < 3)
            break;

        layer_count++;

        // 2. Identificar y eliminar los puntos de la capa actual

        // Usamos un mapa para marcar eficientemente los IDs de los puntos del hull
        map<int, bool> is_on_hull;
        for (const auto &hp : hull_points)
            is_on_hull[hp.id] = true;

        vector<Point> next_P;
        next_P.reserve(P.size() - hull_points.size());

        // 3. Construir el nuevo conjunto de puntos P' (los puntos no eliminados)
        for (const auto &p : P) {
            if (is_on_hull.find(p.id) == is_on_hull.end())
                next_P.push_back(p);
        }

        P = next_P;
    }

    return layer_count;
}
```