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

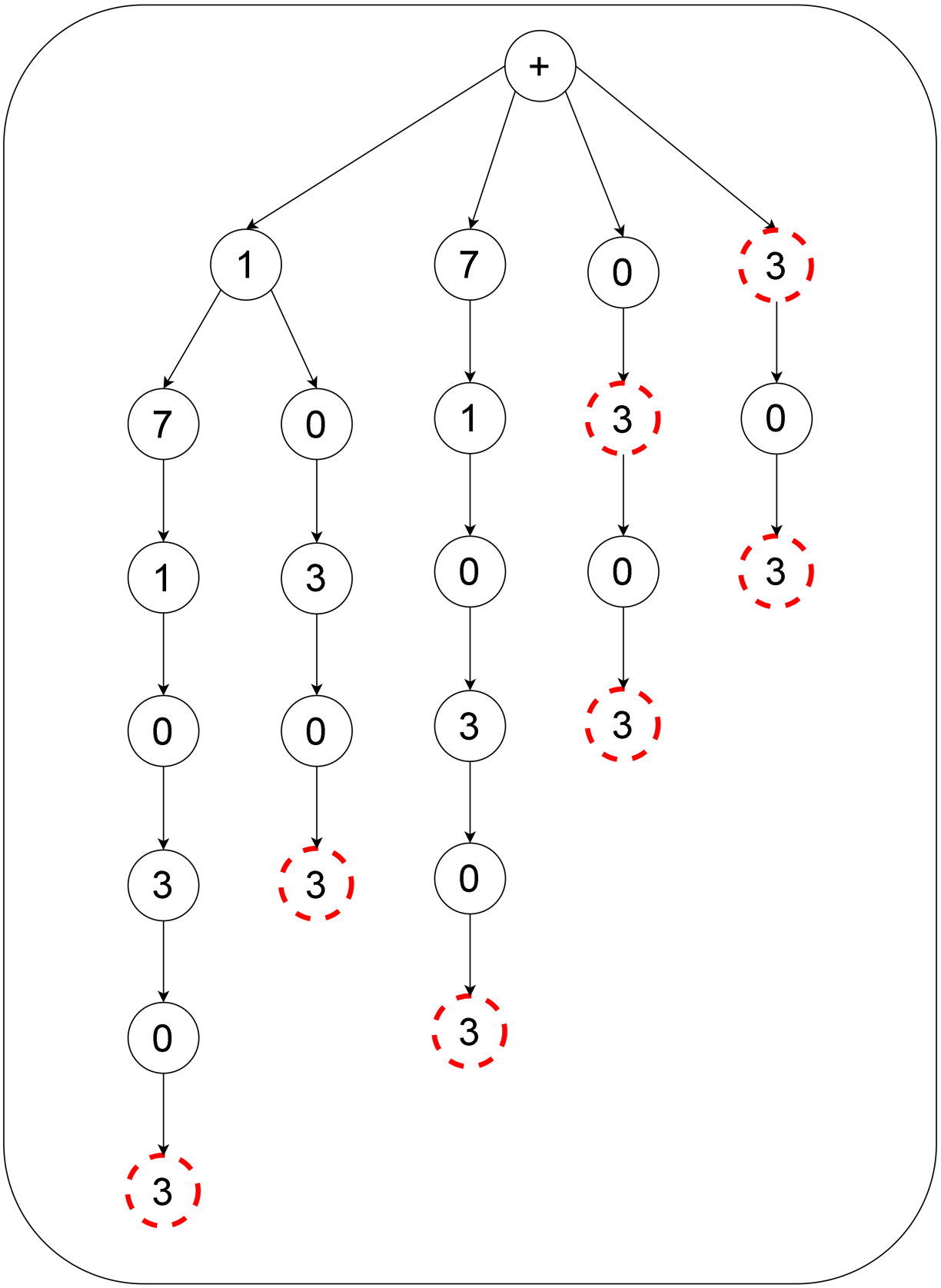
Pregunta 1

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
7
1
3
0
0
3
1
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 Φ

   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;
}
```

Pregunta 4

Para un círculo óptimo de radio R (fijo), existe un centro C tal que el círculo resultante pasa por al menos un punto $p_i \in P$. Esto nos permite reducir el espacio continuo de búsqueda a un conjunto discreto de posiciones.

Si el círculo pasa por un punto p_i, el centro C debe residir en la circunferencia C_i de radio R centrado en p_i. La solución óptima debe estar en alguna de estas N circunferencias candidatas.

La solución se basa en explotar esta propiedad iterando sobre cada punto p_i como si fuera el punto que define el círculo óptimo:

1. **Iteración Principal ($O(N)$):** Seleccionamos un punto $p_i \in P$. Consideramos la circunferencia C_i (centro p_i, radio R) como el foco de los posibles centros óptimos C.
2. **Definición de Arcos ($O(N)$):** Para cada otro punto $p_j \in P$ ($j \neq i$), queremos saber qué centros C en C_i cubren también a p_j.
   - Si la distancia entre p_i y p_j es mayor que $2R$ (más allá del diámetro), p_j nunca puede ser cubierto por un círculo de radio R que toque p_i en su frontera, por lo que no contribuye a un arco.
   - Si p_i y p_j están lo suficientemente cerca, los centros C en C_i que cubren p_j forman un **arco** $(\alpha_{start}, \alpha_{end})$. Este arco está centrado en la dirección del punto medio de $p_i p_j$ y tiene una apertura angular determinada por R.
3. **Barrido y Conteo ($O(N \log N)$):**
   - Recopilamos los $O(N)$ puntos finales de los arcos (eventos angulares: inicio (+) o fin (-)).
   - Ordenamos estos $O(N)$ eventos angulares en el rango $[0, 2\pi)$ ($O(N \log N)$ tiempo).
   - Recorremos la lista de eventos. Mantenemos un contador . Cada evento de inicio aumenta el contador, y cada evento de fin lo disminuye. El máximo valor alcanzado por en el barrido, más 1 (por p_i que ya está cubierto), es la cobertura máxima posible para el círculo que toca p_i.
4. **Resultado:** El máximo entre las coberturas calculadas en las N iteraciones es la respuesta.

En este sentido, para la complejidad tenemos

- Hay N iteraciones principales ($O(N)$).
- Dentro de cada iteración, calculamos $O(N)$ arcos ($O(N)$).
- Ordenamos $O(N)$ eventos angulares ($O(N \log N)$).
- El barrido angular toma $O(N)$.
- La complejidad total es: $N \times (O(N) + O(N \log N)) = O(N^2 \log N)$.

Implementación en C++

El archivo funcional se encuentra en angular_sweep.cpp

```cpp
// Constante de tolerancia para comparaciones con punto flotante
const double EPS = 1e-9;
const double PI = acos(-1.0);

// Estructura para representar un punto
struct Point {
    double x, y;
    int id; // Para seguimiento
};

// Estructura para representar un evento en el barrido angular
struct Event {
    double angle; // Angulo en radianes (0 a 2*PI)
    int type;     // 1 para inicio de arco (+1 cobertura), -1 para fin de arco (-1 cobertura)
};

// Estructura para el resultado completo
struct OptimalResult {
```

```
    int max_count = 0;
    Point optimal_center = {0.0, 0.0};
    vector<Point> covered_points;
};

// Función de comparación para ordenar los eventos.
bool compare_events(const Event &a, const Event &b) {
    if (fabs(a.angle - b.angle) > EPS)
        return a.angle < b.angle;

    return a.type > b.type;
}

// Calcula la distancia euclidiana
double dist(Point p1, Point p2) {
    return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}

// Funcion para calcular el centro C basado en un punto Pi y un ángulo polar theta
Point calculate_center(Point pi, double R, double theta) {
    // El centro C está a distancia R de Pi, en la dirección angular theta
    return {
        pi.x + R * cos(theta),
        pi.y + R * sin(theta)};
}

/**
 * Funcion principal que implementa el algoritmo O(N^2 log N) y encuentra el centro óptimo.
 */
OptimalResult max_points_covered_n2_log_n(vector<Point> &P, double R) {
    int N = P.size();
    if (N == 0)
        return {};

    // Inicializar IDs para depuración y unicidad
    for (int i = 0; i < N; ++i)
        P[i].id = i;

    OptimalResult result;

    // Variables para rastrear la mejor solución encontrada
    // Inicializamos con 1 (al menos el propio pivote)
    result.max_count = 1;
    int best_pivot_idx = 0;
    double best_angle = 0.0;

    // O(N) iteraciones principales.
    for (int i = 0; i < N; ++i) {
        vector<Event> events;

        // O(N) para generar eventos.
        for (int j = 0; j < N; ++j) {
            if (i == j)
                continue;

            double d = dist(P[i], P[j]);

            // Si la distancia es mayor que 2R, P[j] no puede ser cubierta por C(P[i], R).
            if (d > 2.0 * R + EPS)
```

```
                continue;

            // Calculamos el ángulo alpha del vector P[i] -> P[j]
            double alpha = atan2(P[j].y - P[i].y, P[j].x - P[i].x);

            // Calculamos el ángulo beta de apertura del arco: beta = acos(d / (2*R))
            double ratio = min(1.0, d / (2.0 * R)); // Asegura que el argumento de acos no exc
            double beta = acos(ratio);

            double angle_start = alpha - beta;
            double angle_end = alpha + beta;

            // Normalización de ángulos a [0, 2*PI) para el barrido.
            // Para la aritmética de eventos, es más sencillo trabajar con [-PI, PI) o similar
            // y luego mapear a [0, 2PI) si es necesario. Aquí usamos la detección de cruce de

            // Convertir a [0, 2*PI)
            auto normalize = [](double angle) {
                while (angle < 0)
                    angle += 2.0 * PI;
                while (angle >= 2.0 * PI)
                    angle -= 2.0 * PI;
                return angle;
            };

            angle_start = normalize(angle_start);
            angle_end = normalize(angle_end);

            // Generamos los eventos de inicio y fin.
            if (angle_start <= angle_end + EPS) {
                // El arco no cruza el eje 0 (2*PI).
                events.push_back({angle_start, 1});
                events.push_back({angle_end, -1});
            } else {
                // El arco cruza 2*PI (wrap around). Segmento 1: [angle_start, 2*PI). Segmento
                events.push_back({angle_start, 1});
                events.push_back({2.0 * PI - EPS, -1}); // Fin cerca de 2*PI
                events.push_back({0.0, 1});             // Inicio en 0
                events.push_back({angle_end, -1});
            }
        }

        // 3. Barrido Angular

        // O(N log N) para ordenar.
        sort(events.begin(), events.end(), compare_events);

        int current_coverage = 1; // P[i] siempre está cubierto

        // Iteración O(N)
        for (const auto &event : events) {
            current_coverage += event.type;

            // Si encontramos una cobertura superior, actualizamos el máximo.
            // Utilizamos el ángulo del evento (si es de inicio o inmediatamente después de un
            // como el centro óptimo candidato.
            if (current_coverage > result.max_count) {
                result.max_count = current_coverage;
                best_pivot_idx = i;
```

```
                best_angle = event.angle;

                // Si el evento era de fin (-1), significa que la cobertura máxima fue justo
                // antes. Usamos este ángulo como una aproximación del inicio del plateau de c
                // Sin embargo, para simplicidad y robustez, usamos el ángulo del evento que
                // nos llevó a este nuevo máximo.
            }
        }
    }

    // 4. Calcular el Centro Óptimo Final
    result.optimal_center = calculate_center(P[best_pivot_idx], R, best_angle);

    // 5. Generar la lista de puntos cubiertos por el centro óptimo.
    for (const auto &p : P) {
        // dist devuelve la distancia euclidiana; comparar con R (no R^2)
        if (dist(p, result.optimal_center) <= R + EPS)
            result.covered_points.push_back(p);
    }

    return result;
}
```