

Resolución de Tarea 8 - FFT y Optimización para Programación Dinámica (Fecha: 24 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 8 (9 puntos)

Indice

- Resolución de Tarea 8 - FFT y Optimización para Programación Dinámica (Fecha: 24 de Noviembre de 2025)
- Indice
- Pregunta 1
 - Nivel 1
 - Nivel 2
 - Nivel 3 (Cálculos de base)
 - Nivel 2 (Continuación)
 - Nivel 1 (Continuación)
 - Resultado
- Pregunta 2
 - Implementación en C++
- Pregunta 3
 - Fase I: Reducción Geométrica y Linealización
 - Fase II: Programación Dinámica con Convex Hull (CH)
 - Complejidad Total
 - Implementación en C++

Pregunta 1

El carné es **17-10303** y por tanto el polinomio derivado es:

$$P(x) = 1 + 7x + x^2 + 0x^3 + 3x^4 + 0x^5 + 3x^6$$

Teniendo en cuenta que FFT es un algoritmo (un método) que se utiliza para calcular la DFT siempre y cuando el tamaño de la entrada N sea una potencia de dos. Entonces, el grado de ligadura (o tamaño del vector de coeficientes) debe ser estrictamente mayor que el grado del polinomio, generalmente $N = d + 1$. Por lo tanto, el grado de ligadura mínimo sería $N = 7$ en nuestro caso.

Dado que N debe ser una potencia de dos para aplicar FFT eficientemente, el tamaño N debe aumentarse a la siguiente potencia de dos, que es $N = 8$ (ya que $2^3 = 8$). Así, nuestro polinomio se completa con 0's.

$$P(x) = 1 + 7x + x^2 + 0x^3 + 3x^4 + 0x^5 + 3x^6 + 0x^7$$

Se tiene:

- **Coeficientes:** $a = \langle 1, 7, 1, 0, 3, 0, 3, 0 \rangle$.
- **Grado de Ligadura (N):** Dado que el grado máximo es 6, y se pide usar las raíces octavas de la unidad, el grado de ligadura es $N = 8$.

- **Raíz Primitiva de la Unidad (ω_8):** Usamos la N -ésima raíz primitiva de la unidad, $\omega_8 = e^{2\pi i/8} = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$.

El algoritmo FFT se basa en la descomposición de $P(x)$ en polinomios de coeficientes pares e impares.

Nivel 1

Se tiene $N = 8$, $\omega = \omega_8$

- **División:**
 - $P_{par}(x)$: Coeficientes pares: $\langle a_0, a_2, a_4, a_6 \rangle = \langle 1, 1, 3, 3 \rangle$.
 - $P_{impar}(x)$: Coeficientes impares: $\langle a_1, a_3, a_5, a_7 \rangle = \langle 7, 0, 0, 0 \rangle$.
- **Recursión:** Calculamos la DFT de P_{par} y P_{impar} en las $\frac{N}{2} = 4$ raíces de la unidad, $\omega_4 = (\omega_8)^2 = i$.

Nivel 2

Se tiene $N = 4$, $\omega = \omega_4 = i$

- **1. DFT de $P_{par}(x) = 1 + x + 3x^2 + 3x^3$:**
 - **División:**
 - * $P_{pp}(x)$: $\langle 1, 3 \rangle$.
 - * $P_{pi}(x)$: $\langle 1, 3 \rangle$.
 - **Recursión:** Calculamos DFT de P_{pp} y P_{pi} en las $\frac{4}{2} = 2$ raíces, $\omega_2 = (\omega_4)^2 = -1$.
- **2. DFT de $P_{impar}(x) = 7$:**
 - **División:**
 - * $P_{ip}(x)$: $\langle 7, 0 \rangle$.
 - * $P_{ii}(x)$: $\langle 0, 0 \rangle$.
 - **Recursión:** Calculamos DFT de P_{ip} y P_{ii} en $\omega_2 = -1$.

Nivel 3 (Cálculos de base)

Se tiene $N = 2$, $\omega = \omega_2 = -1$

- **1. DFT de $P_{pp}(x) = 1 + 3x$ (y P_{pi}):**
 - Base (FFT de coeficientes $\langle 1 \rangle$ y $\langle 3 \rangle$).
 - Combinación: $j = 0, 1$. Raíces: $\omega_2^0 = 1$, $\omega_2^1 = -1$.
 - * $P_{pp}(1) = 1 + 1 \cdot 3 = 4$.
 - * $P_{pp}(-1) = 1 + (-1) \cdot 3 = -2$.
 - **Resultado S_{pp} y S_{pi} :** $\langle 4, -2 \rangle$.
- **2 DFT de $P_{ip}(x) = 7$:**
 - Base (FFT de coeficientes $\langle 7 \rangle$ y $\langle 0 \rangle$).
 - Combinación: $j = 0, 1$. Raíces: $\omega_2^0 = 1$, $\omega_2^1 = -1$.
 - * $P_{ip}(1) = 7 + 1 \cdot 0 = 7$.
 - * $P_{ip}(-1) = 7 - 1 \cdot 0 = 7$.
 - **Resultado S_{ip} :** $\langle 7, 7 \rangle$.
- **3. DFT de $P_{ii}(x) = 0$:**
 - **Resultado S_{ii} :** $\langle 0, 0 \rangle$.

Nivel 2 (Continuación)

Se tiene combinación $N = 4$

- **1. Combinación de $P_{par}(x)$:**
 - S_{par} : $S_{pp} = \langle 4, -2 \rangle$, $S_{pi} = \langle 4, -2 \rangle$.
 - Raíces: $\omega_4^0 = 1$, $\omega_4^1 = i$.

- Utilizamos la fórmula de recombinación: $r_j = S_j + \omega^j S'_j$ y $r_{j+N/2} = S_j - \omega^j S'_j$.

j	ω_4^j	S_j	S'_j	$P_{par}(\omega_4^j)$	$P_{par}(\omega_4^{j+2})$
0	1	4	4	$4 + 1(4) = 8$	$4 - 1(4) = 0$
1	i	-2	-2	$-2 + i(-2) = -2 - 2i$	$-2 - i(-2) = -2 + 2i$

Resultado S_{par} : $\langle 8, -2 - 2i, 0, -2 + 2i \rangle$.

- **2. Combinación de $P_{impar}(x)$:**
 - S_{impar} : $S_{ip} = \langle 7, 7 \rangle$, $S_{ii} = \langle 0, 0 \rangle$.
 - Raíces: $\omega_4^0 = 1, \omega_4^1 = i$.

j	ω_4^j	S_j	S'_j	$P_{impar}(\omega_4^j)$	$P_{impar}(\omega_4^{j+2})$
0	1	7	0	$7 + 1(0) = 7$	$7 - 1(0) = 7$
1	i	7	0	$7 + i(0) = 7$	$7 - i(0) = 7$

Resultado S_{impar} : $\langle 7, 7, 7, 7 \rangle$.

Nivel 1 (Continuación)

Se tiene combinación final $N = 8$

- **1. Combinación Final de $P(x)$:**
 - S_{par} y S_{impar} .
 - Raíz: $\omega_8 = e^{\pi i/4}$.
 - Utilizaremos ω_8 y sus potencias para $j = 0, 1, 2, 3$:
 - * $\omega_8^0 = 1$
 - * $\omega_8^1 = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$
 - * $\omega_8^2 = i$
 - * $\omega_8^3 = -\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$
- Sea $C = 3.5\sqrt{2} \approx 4.95$.

j	ω_8^j	$S_{par,j}$	$S_{impar,j}$	$P(\omega_8^j) = S_{par,j} + \omega_8^j S_{impar,j}$	$P(\omega_8^{j+4}) = S_{par,j} - \omega_8^j S_{impar,j}$
0	1	8	7	$8 + 1(7) = 15$	$8 - 1(7) = 1$
1	ω_8^1	$-2 - 2i$	7	$-2 - 2i + 7\omega_8^1 = (-2 + C) + i(-2 + C)$	$-2 - 2i - 7\omega_8^1 = (-2 - C) + i(-2 - C)$
2	i	0	7	$0 + i(7) = 7i$	$0 - i(7) = -7i$
3	ω_8^3	$-2 + 2i$	7	$-2 + 2i + 7\omega_8^3 = (-2 - C) + i(2 + C)$	$-2 + 2i - 7\omega_8^3 = (-2 + C) + i(2 - C)$

Resultado

La Transformada Discreta de Fourier del polinomio $P(x)$ en las raíces octavas de la unidad es el vector Y :

$$Y = \langle P(\omega_8^0), P(\omega_8^1), P(\omega_8^2), P(\omega_8^3), P(\omega_8^4), P(\omega_8^5), P(\omega_8^6), P(\omega_8^7) \rangle$$

Donde $C = \frac{7\sqrt{2}}{2}$:

$$\begin{aligned} Y = & \langle 15, \\ & (-2 + C) + i(-2 + C), \\ & 7i, \\ & (-2 - C) + i(2 + C), \\ & 1, \\ & (-2 - C) + i(-2 - C), \\ & -7i, \\ & (-2 + C) + i(2 - C) \rangle \end{aligned}$$

Pregunta 2

Definimos $F(Y)$ como la cantidad de formas en que un entero positivo Y puede escribirse como el producto de dos enteros positivos $a \cdot b$. En otras palabras, $F(Y)$ es el número de divisores de Y .

El valor $\text{decomp}(X)$ es entonces el número de pares (Y_1, Y_2) tales que $Y_1 + Y_2 = X$, donde Y_1 se puede descomponer en $a \cdot b$ de $F(Y_1)$ maneras, y Y_2 se puede descomponer en $c \cdot d$ de $F(Y_2)$ maneras. Esto es la definición de una **convolución**:

$$\text{decomp}(X) = \sum_{Y_1 + Y_2 = X, Y_1, Y_2 > 0} F(Y_1) \cdot F(Y_2)$$

Esta operación de convolución puede ser modelada como la multiplicación de un polinomio $P(x)$ por sí mismo, $C(x) = P(x) \times P(x)$, donde el coeficiente de x^Y en $P(x)$ es $F(Y)$:

$$P(x) = \sum_{Y=1}^N F(Y)x^Y$$

El coeficiente C_X del polinomio producto $C(x)$ nos da exactamente $\text{decomp}(X)$.

Así se tiene que

- Se calculan todos los valores $F(Y)$ hasta N utilizando un método similar al tamiz (iterando sobre múltiplos) en tiempo $O(N \log N)$.
- Se aplica la FFT al vector de coeficientes F para pasar a la representación puntual. Esto se logra mediante el algoritmo recursivo de Divide y Vencerás, $T(n) = 2T(\frac{n}{2}) + O(n)$, cuya solución es $\Theta(n \log n)$.
- Se multiplican los valores de la transformada término a término.
- Se aplica la FFT inversa (que también se ejecuta en $O(N \log N)$) para obtener los coeficientes finales $\text{decomp}(X)$.

Dado que todas las fases dominantes se ejecutan en $O(N \log N)$, el algoritmo total cumple con la restricción de complejidad requerida.

Implementación en C++

El archivo funcional se encuentra en [decomp.cpp](#)

```
1 // Definición de tipos para el FFT
2 using CD = complex<long double>;
3 const long double PI = acos(-1.0L);
4
5 /**
6  * Precalcula F[i], el número de divisores de i, para 1 <= i <= N.
7  * Esta es la cantidad de formas de escribir i = a * b (a, b > 0).
8  * Complejidad: O(N log N)
9  */
10 vector<int> calculate_divisor_counts(int N) {
11     vector<int> F(N + 1, 0);
12     // Itera sobre los posibles factores 'a' y marca sus múltiplos 'Y'
13     for (int a = 1; a <= N; ++a) {
14         for (int Y = a; Y <= N; Y += a)
15             F[Y]++;
16     }
17     return F;
18 }
19
20 /**
21  * Implementación de la Transformada Rápida de Fourier (FFT).
22  * time: O(N log N)
23  */
24 void fft(vector<CD> &a, bool invert) {
25     int n = a.size();
26     if (n == 1)
27         return;
28
29     // Implementación sin recursión explícita (bit reversal permutation)
30     for (int i = 1, j = 0; i < n; i++) {
31         int bit = n >> 1;
32         for (; j & bit; bit >>= 1)
33             j ^= bit;
34         j ^= bit;
35         if (i < j)
36             swap(a[i], a[j]);
37     }
38
39     for (int len = 2; len <= n; len <<= 1) {
40         long double ang = 2 * PI / len * (invert ? -1 : 1);
41         CD wlen(cos(ang), sin(ang));
42         for (int i = 0; i < n; i += len) {
43             CD w(1);
44             for (int j = 0; j < len / 2; j++) {
45                 CD u = a[i + j];
46                 CD v = a[i + j + len / 2] * w;
47                 a[i + j] = u + v;
48                 a[i + j + len / 2] = u - v;
49                 w *= wlen;
50             }
51         }
52     }
53
54     if (invert) {
55         for (CD &x : a)
56             x /= n;
```

```

57     }
58 }
59
60 /**
61  * Realiza la multiplicación de polinomios P * P (convolución).
62  * Retorna el vector C, donde C[X] = decomp(X).
63  * time: O(N log N)
64  */
65 vector<long long> multiply_polynomial_self(const vector<int> &F, int N) {
66     // F es el vector de coeficientes [F, F, ..., F[N]]
67
68     // El grado de ligadura N_prime debe ser al menos 2N (para P*P)
69     int N_coeffs = N + 1; // Cantidad de coeficientes reales (0 a N)
70     int N_prime = 2 * N + 1;
71     int M = 1;
72     while (M < N_prime)
73         M *= 2; // Siguiendo potencia de 2 para FFT
74
75     // 1. Padding y Mapeo a Complejos
76     vector<CD> P_coeffs(M, 0);
77     for (int i = 0; i <= N; ++i)
78     {
79         // F[i] es el coeficiente de x^i
80         P_coeffs[i] = F[i];
81     }
82
83     // 2. FFT Forward
84     fft(P_coeffs, false);
85
86     // 3. Multiplicación Puntual (P * P)
87     for (int i = 0; i < M; ++i)
88         P_coeffs[i] *= P_coeffs[i];
89
90     // 4. FFT Inversa (Interpolación)
91     fft(P_coeffs, true);
92
93     // 5. Extracción de Coeficientes Reales
94     // C[X] almacena decomp(X)
95     // Usamos round para manejar errores de punto flotante
96     vector<long long> C(N_prime);
97     for (int i = 0; i < N_prime; ++i)
98         C[i] = round(P_coeffs[i].real());
99
100     return C;
101 }
102
103 /**
104  * Encuentra el máximo valor de decomp(X) en el rango [1, N].
105  * Retorna un par {Max_X, Max_Decomposition_Count}.
106  */
107 pair<int, long long> solve_max_decomposition(int N) {
108     if (N < 1)
109         return {0, 0};
110
111     // 1. Precomputar el número de divisores F(Y)
112     vector<int> F = calculate_divisor_counts(N);
113
114     // 2. Calcular la convolución C(x) = P(x) * P(x)
115     vector<long long> C = multiply_polynomial_self(F, N);

```

```

116
117 // 3. Encontrar el máximo de comp(X) para 1 <= X <= N
118 long long max_val = 0;
119 int max_X = 0;
120
121 // La convolución C tiene elementos hasta 2N, pero solo nos interesa hasta N.
122 // Además, C siempre será 0 ya que a, b, c, d > 0.
123 int limit = min((int)C.size() - 1, N);
124
125 for (int X = 1; X <= limit; ++X) {
126     if (C[X] > max_val) {
127         max_val = C[X];
128         max_X = X;
129     }
130 }
131
132 return {max_X, max_val};
133 }

```

Pregunta 3

El objetivo es minimizar el costo total de la partición, donde el costo de un subconjunto C está definido por $\text{costo}(C) = \max(h) \times \max(w)$. Entonces, se presentan las siguientes fases para solventar el problema

Fase I: Reducción Geométrica y Linealización

El problema se presenta sobre un conjunto desordenado R , pero la Programación Dinámica es inherentemente lineal. Debemos imponer un orden que preserve la optimalidad.

La primera observación es que muchos rectángulos son **redundantes**. Si un rectángulo $r_j = (h_j, w_j)$ domina a $r_i = (h_i, w_i)$ ($h_j \geq h_i$ y $w_j \geq w_i$), entonces r_i nunca determinará el costo máximo de altura o ancho en ningún subconjunto que contenga a r_j .

- Filtramos el conjunto R para retener solo los rectángulos **maximales** R' .
- Esto se logra ordenando los rectángulos por ancho (w) decreciente. Luego, en una pasada de barrido, se mantiene la altura máxima vista. Cualquier rectángulo con una altura menor o igual a esa altura máxima es descartado.
- Esta fase requiere un ordenamiento ($O(N \log N)$) y un barrido lineal ($O(N)$).

Los rectángulos maximales restantes R' deben ordenarse para permitir la segmentación lineal (subsegmentos contiguos). Ordenamos R' por **altura** (h) **creciente**. Debido al filtrado previo:

- La altura h_i es no-decreciente.
- El ancho w_i es estrictamente decreciente (para alturas iguales, no importa el orden, ya que el ancho máximo será el primero).

Al imponer este orden a R' , un segmento contiguo $R'[j+1..i]$ tendrá un costo simplificado:

$$\text{costo}(R'[j+1..i]) = \max_{k=j+1}^i (h_k) \times \max_{k=j+1}^i (w_k) = h_i \times w_{j+1}$$

Fase II: Programación Dinámica con Convex Hull (CH)

Definimos $DP[i]$ como el costo mínimo de particionar los primeros i rectángulos de la secuencia ordenada R' .

$$DP[i] = \min_{0 \leq j < i} \{DP[j] + h_i \cdot w_{j+1}\}$$

Reescribimos la recurrencia para que encaje en la forma $m \cdot x + b$:

$$DP[i] = \min_{0 \leq j < i} \{ (w_{j+1}) \cdot (h_i) + (DP[j]) \}$$

Aquí:

- **Pendiente:** $m_j = w_{j+1}$ (depende solo del punto de división j).
- **Punto de Consulta:** $x_i = h_i$ (depende solo del estado actual i).
- **Intercepto:** $b_j = DP[j]$ (depende solo del punto de división j).

Dado que la secuencia R' fue ordenada:

- $x_i = h_i$ es **monótono no-decreciente** (las consultas avanzan de izquierda a derecha).
- $m_j = w_{j+1}$ es **monótono decreciente** (las pendientes se hacen cada vez menos inclinadas).

Estas propiedades permiten aplicar la versión más eficiente de la CH, utilizando una estructura de datos tipo cola de doble terminación (deque) que mantiene la envolvente convexa óptima, lo que reduce la fase de búsqueda de mínimos a $O(1)$ **amortizado** por estado.

Complejidad Total

Fase	T. Dominante	Complejidad
I. Filtrado Maximal	Ordenamiento	$O(N \log N)$
I. Ordenamiento para DP	Ordenamiento	$O(N \log N)$
II. DP con CH	CH (Deque)	$O(N)$ amortizado
Total		$O(N \log N)$

El uso de memoria es $O(N)$ para almacenar los rectángulos iniciales, los rectángulos maximales y las tablas DP y CH.

Implementación en C++

El archivo funcional se encuentra en [partition_rectangles.cpp](#)

```

1 // Usamos long long para manejar costos grandes (Area = H * W)
2 using ll = long long;
3
4 struct Rectangle {
5     ll h; // alto
6     ll w; // ancho
7 };
8
9 /**
10  * Filtra el conjunto R para obtener solo rectángulos maximales R'.
11  * Ordena por ancho descendente y luego filtra por altura.
12  * Complejidad: O(N log N)
13  */
14 vector<Rectangle> filter_maximal_rects(vector<Rectangle> &R) {
15     if (R.empty())
16         return {};
17
18     // 1. Ordenar por ancho (w) descendente. Si hay empate, por alto (h) ascendente.
19     // Esto es crucial para la lógica de barrido.
20     sort(R.begin(), R.end(), [](const Rectangle &a, const Rectangle &b)
21         {
22             if (a.w != b.w) return a.w > b.w;

```

```

23     return a.h < b.h; });
24
25     vector<Rectangle> R_prime;
26     ll max_h_so_far = 0;
27
28     // 2. Barrido para eliminar redundantes
29     for (const auto &r : R) {
30         // Si ya encontramos un rectángulo con mayor o igual altura Y mayor ancho,
31         // este rectángulo actual es dominado.
32         // Como estamos ordenados por ancho decreciente, solo necesitamos verificar la altura.
33         if (r.h > max_h_so_far) {
34             R_prime.push_back(r);
35             max_h_so_far = r.h;
36         }
37     }
38     return R_prime;
39 }
40
41 /**
42  * Función auxiliar para verificar la envolvente convexa de 3 líneas j1, j2, j3.
43  * Comprueba si la línea j2 es redundante (no formará parte de la envolvente).
44  * Usamos la comparación de pendiente/intersección.
45  * Dado que las pendientes (m_j) son decrecientes, necesitamos que las intersecciones
46  * sean crecientes (convex hull inferior).
47  * La intersección entre j1 y j2 debe ser ANTES de la intersección entre j2 y j3.
48  *
49  * La función retorna true si (j2, j3) es "menos empinada" que (j1, j2)
50  *
51  * Fórmula: (b2 - b1) / (m1 - m2) <= (b3 - b2) / (m2 - m3)
52  * Multiplicando por denominadores positivos (m1 > m2 > m3), evitamos divisiones
53  * y mantenemos precisión con long long.
54  */
55 bool is_redundant(int j1, int j2, int j3, const vector<ll> &DP, const vector<Rectangle>
56     // Extracción de pendientes m y coordenadas y b
57     ll m1 = R_prime[j1 - 1].w;
58     ll b1 = DP[j1];
59
60     ll m2 = R_prime[j2 - 1].w;
61     ll b2 = DP[j2];
62
63     ll m3 = R_prime[j3 - 1].w;
64     ll b3 = DP[j3];
65
66     // Chequeo de convexidad usando cross product/pendiente (evitando floats)
67     // (b2 - b1) * (m2 - m3) <= (b3 - b2) * (m1 - m2)
68     // True si j2 debe ser eliminado (no contribuye a la convex hull inferior)
69
70     // Nota: Se debe manejar la indexación. DP está indexado de 0 a N'. R_prime de 0 a N'-1.
71     // j=0 es el caso base/ficticio.
72
73     // Caso especial para j=0 (DP=0, m_0 = infinito). Usamos un rectángulo ficticio
74     // con m_0 muy grande. Esto se simplifica forzando j1 > 0.
75
76     // Para simplificar, asumimos que los índices j1, j2, j3 ya son índices válidos
77     // de la tabla DP (0 a N').
78
79     // j1 es la línea ficticia con m_0 = MAX.
80     // En este caso, j2 y j3 deben ser los primeros dos puntos reales.
81     // La primera línea no puede ser redundante hasta que haya 3 líneas.

```

```

82     if (j1 == 0)
83         return false;
84
85     // Si m1 < m2, la envolvente inferior falla, lo que es imposible por el pre-ordenamiento.
86     // En nuestro caso m1 > m2 > m3, ya que w es estrictamente decreciente.
87
88     // Si las pendientes (w) son iguales, la línea con DP[j] más pequeño gana
89     if (m1 == m2)
90         return b1 <= b2;
91     if (m2 == m3)
92         return b2 <= b3;
93
94     // La fórmula original (b2 - b1) / (m1 - m2) <= (b3 - b2) / (m2 - m3)
95     return (b2 - b1) * (m2 - m3) >= (b3 - b2) * (m1 - m2);
96 }
97
98 /**
99  * Algoritmo principal: Partición óptima de rectángulos usando CH.
100  * Complejidad: O(N log N) debido al pre-ordenamiento.
101  */
102 ll solve_optimal_partition(vector<Rectangle> &R)
103 {
104     // Paso 1: Filtrado O(N log N)
105     vector<Rectangle> R_prime = filter_maximal_rects(R);
106     int N_prime = R_prime.size();
107     if (N_prime == 0)
108         return 0;
109
110     // IMPORTANTE: No volvemos a reordenar R_prime por altura aquí porque el
111     // filtrado garantiza que los anchos están en orden no creciente y las
112     // propiedades para tomar R_prime[j].w como ancho máximo del segmento se mantienen.
113
114     // DP[i] = mínimo costo para cubrir primeros i rectángulos (0..i-1)
115     const ll INF = numeric_limits<ll>::max() / 4;
116     vector<ll> DP(N_prime + 1, INF);
117     DP[0] = 0;
118
119     for (int i = 1; i <= N_prime; ++i) {
120         ll H_i = R_prime[i - 1].h;
121         ll best = INF;
122         for (int j = 0; j < i; ++j) {
123             // ancho máximo en el segmento [j..i-1] es R_prime[j].w por la ordenación previa
124             ll W_j = R_prime[j].w;
125             ll cand = DP[j] + H_i * W_j;
126             if (cand < best)
127                 best = cand;
128         }
129         DP[i] = best;
130     }
131
132     return DP[N_prime];
133 }

```