Resolución de Tarea 4 - Programación Dinámica (Fecha: 20 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 4 (9 puntos)

Indice

Pregunta 1

Mi fecha de naimiento es: 8 de **Junio** de 1999 y mi dia de nacimiento es **Martes**. Verificar acá.

La tabla de la distancia de edición entre **Martes** → **Junio** es

Index	1	2	3	4	5	6
Día	M	A	R	T	E	S
Mes	J	U	N	I	O	

- En la inicialización de fila 0 y columna 0 tenemos:

0	1	2	3	4	5
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0

- Tabla para $i = 1$

0	1	2	3	4	5
1	1	2	3	4	5
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0

- Tabla para $i = 2$

0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0

- Tabla para $i = 3$

0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
3	3	3	3	4	5
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0

- Tabla para $i = 4$

0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
3	3	3	3	4	5
4	4	4	4	4	5
5	0	0	0	0	0
6	0	0	0	0	0

- Tabla para $i = 5$

0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
3	3	3	3	4	5
4	4	4	4	4	5
5	5	5	5	5	5
6	0	0	0	0	0

- Tabla para $i = 6$

0	1	2	3	4	5
1	1	2	3	4	5
2	2	2	3	4	5
3	3	3	3	4	5
4	4	4	4	4	5
5	5	5	5	5	5
6	6	6	6	6	$\boxed{6}$

Por lo tanto, a distancia de edición entre las cadenas es 6.

> [!NOTE] Dado que no existe al menos una letra en común entre los estados iniciales de las palabras notamos que la distancia es de 6 igual a la longitud de la palabra mas larga (martes). Esto básicamente simula que se debe cambiar todos los caracteres para conseguir el objetivo. Acá el código en C++ con el algoritmo respectivo para realizar pruebas: distancia_edicion.cpp.

Pregunta 2

Definimos $DP[i][j]$ como la longitud del par de subarreglos familiares más largo, donde **el primer subarreglo termina en el índice i y el segundo subarreglo termina en el índice j**, asumiendo $i < j$.

Para calcular $DP[i][j]$, consideramos la posibilidad de extender un par familiar anterior, $DP[i-1][j-1]$.

Sea $L' = DP[i-1][j-1]$.

1. **Condición de Coprimos:** Los elementos $A[i]$ y $A[j]$ deben ser coprimos (es decir, $\gcd(A[i], A[j]) = 1$).
2. **Condición de Disyunción:** El nuevo par de longitud $L = L' + 1$ debe ser disjunto. Dado que $i < j$ y la longitud L es $L' + 1$, el primer subarreglo va desde $i - L + 1$ hasta i, y el segundo va desde $j - L + 1$ hasta j. Para que sean disjuntos, el índice final del primero (i) debe ser estrictamente menor que el índice inicial del segundo ($j - L + 1$).

$$i < j - L + 1 \quad \Rightarrow \quad L \leq j - i$$

Si esta condición se cumple, los subarreglos son disjuntos (o adyacentes si $L = j - i$).

La **recurrencia** es:

$$DP[i][j] = \begin{cases} DP[i-1][j-1] + 1 & \text{si } \begin{cases} \gcd(A[i], A[j]) = 1 \\ \wedge \\ DP[i-1][j-1] + 1 \leq j - i \end{cases} \\ 0 & \text{en otro caso} \end{cases}$$

El valor final buscado es el máximo valor encontrado en toda la tabla DP.

Complejidad

La dependencia $DP[i][j] \leftarrow DP[i-1][j-1]+1$ es puramente diagonal. Esto significa que, al calcular la fila i, solo necesitamos los resultados de la fila anterior $i-1$.

- **Complejidad Temporal**
  - **Doble Bucle:** El algoritmo utiliza dos bucles (uno para el índice final del primer subarreglo, i, y otro para el índice final del segundo, j). Estos bucles anidados recorren aproximadamente $n(n-1)/2$ pares de índices, lo que resulta en una complejidad de $O(n^2)$.
  - **Operaciones Constantes:** Dentro del bucle, la operación crucial (verificar la condición de coprimos mediante el Máximo Común Divisor y actualizar el valor de DP) se realiza en tiempo constante, $O(1)$, según las condiciones del problema.
- **Complejidad Espacial**

  Dado que la dependencia es local (solo de la diagonal anterior), podemos aplicar la técnica de "ahorro de espacio". En lugar de un arreglo 2D de $O(n^2)$, utilizaremos un arreglo 1D auxiliar de tamaño $O(n)$ para almacenar solo los valores de la fila anterior, reduciendo la memoria adicional a $O(n)$.

Implementación en C++

La solución se implementa en C++. Dada la restricción de $O(1)$ para las operaciones aritméticas, se incluye una implementación simple del Máximo Común Divisor (GCD) basada en el algoritmo de Euclides.

El archivo funcional se encuentra en family_array.cpp

```cpp
int calculate_gcd(int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}

int longest_familiar_subarrays(const vector<int> &A, int n) {
    if (n < 2) return 0;

    // Usamos A...A[n]. El vector A indexado a 0 debe tener tamaño n+1.
    // Creamos un vector interno que representa A[1..n].
    // Creamos una copia A_1_based para facilitar la lógica de A[i] donde i >= 1.
    vector<int> A_1_based(n + 1);
```

```cpp
for (int i = 1; i-1 < n; ++i) {
    A_1_based[i] = A[i - 1];
}

// DP[j] almacenará el valor de DP[i-1][j] (de la fila anterior).
// DP_row[j] es la longitud del par familiar más largo que termina en (i-1, j).
// Inicialmente, todos son 0. Usamos tamaño n+1 para indexar 1 a n.
// Esto cumple con la restricción de memoria adicional O(n).
vector<int> DP_row(n + 1, 0);
int max_length = 0;

// Iteramos 'i', el índice de terminación del primer subarreglo. (i = 1 a n)
for (int i = 1; i <= n; ++i) {
    // Variable temporal para almacenar DP[i-1][j-1] (dependencia diagonal).
    // Se inicializa a 0, que es DP[i-1][i-1].
    int prev_diag_val = 0;

    // Iteramos 'j', el índice de terminación del segundo subarreglo.
    // (j = i+1 a n) Forzamos j > i para asegurar que los
    // pares (i, j) sean únicos y en orden.
    for (int j = i + 1; j <= n; ++j) {
        // 1. Guardamos el valor actual de DP_row[j] (que es DP[i-1][j]).
        //    Este valor se convertirá en la dependencia diagonal
        //    (DP[i-1][j]) para la próxima iteración de j (j+1) en la fila i
        //    (i.e., j_new = j+1, i_new=i+1).
        int current_diag_val = DP_row[j];
        int new_length = 0;

        // 2. Condición de Coprimidad: Chequeamos si A[i] y A[j] son coprimos.
        if (calculate_gcd(A_1_based[i], A_1_based[j]) == 1) {
            // L' es la longitud del par familiar que terminaba en (i-1, j-1).
            int L_prime = prev_diag_val;
            int potential_length = L_prime + 1;

            // 3. Condición de Disyunción: El nuevo par de longitud L debe
            //    ser disjunto. El subarreglo termina en i. Si la longitud
            //    es L, comienza en i - L + 1. Para que los subarreglos
            //    sean disjuntos, L debe ser a lo sumo la distancia entre
            //    ellos: L <= j - i
            if (potential_length <= j - i) {
                new_length = potential_length;
            }
        }

        // 4. Actualizamos el máximo global.
        max_length = max(max_length, new_length);
```

```
            // 5. Almacenamos el nuevo valor DP[i][j] en el arreglo DP_row[j]
            //    para la siguiente fila.
            DP_row[j] = new_length;

            // 6. Actualizamos prev_diag_val para la próxima iteración del
            // bucle j. DP[i-1][j] se convierte en DP[i-1][(j+1)-1].
            prev_diag_val = current_diag_val;
        }
    }
    return max_length;
}
```

Pregunta 3

El programa se encuentra en el siguiente enlace: virtual_inicialization.cpp

Pregunta 4

Se establece que transitar entre dos puntos a y b toma un tiempo igual al **cuadrado de la distancia cartesiana** entre ellos.

Si tenemos dos puntos $P_a = (x_a, y_a)$ y $P_b = (x_b, y_b)$, la distancia Euclidiana $d(P_a, P_b)$ se define como:

$$d(P_a, P_b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Esta fórmula se basa en el Teorema de Pitágoras.

El costo (tiempo) T de viajar entre P_a y P_b es el **cuadrado** de esta distancia:

$$T(P_a, P_b) = d(P_a, P_b)^2 = (x_a - x_b)^2 + (y_a - y_b)^2$$

Así, al calcular el costo en la dirección opuesta, $T(P_b, P_a)$, obtenemos:

$$T(P_b, P_a) = (x_b - x_a)^2 + (y_b - y_a)^2$$

Dado que elevar al cuadrado elimina el efecto del signo (es decir, $(k)^2 = (-k)^2$), tenemos que:

$$\text{En x } \rightarrow (x_a - x_b)^2 = (x_b - x_a)^2$$

$$\text{En y } \rightarrow (y_a - y_b)^2 = (y_b - y_a)^2$$

Por lo tanto, la función de costo es **simétrica**:

$$T(P_a, P_b) = T(P_b, P_a)$$

Por lo tanto, realizar un viaje que cumpla el siguiente camino 1. Viaje $P_0 \to P_i \to P_j \to P_0$ 2. Viaje $P_0 \to P_j \to P_i \to P_0$

Ambos costos, al sumarse individualmente, son: * Costo 1: $T(P_0, P_i)+T(P_i, P_j)+T(P_j, P_0)$ * Costo 2: $T(P_0, P_j) + T(P_j, P_i) + T(P_i, P_0)$

Debido a la simetría del costo individual, el Costo 1 y el Costo 2 serán siempre idénticos en este problema.

Utilizamos una **máscara de bits** (Bitmask) para representar el conjunto de maletas recogidas. Una máscara S es un entero donde el i-ésimo bit encendido (1) indica que la maleta i ha sido recogida.

- **Estado:** $DP[S]$ es el tiempo mínimo acumulado para recoger todas las maletas cuyo índice de bit está encendido en la máscara S.
- **Caso Base:** $DP = 0$ (cero tiempo si no se ha recogido ninguna maleta).
- **Objetivo:** Calcular $DP[(1 \ll n) - 1]$, que es la máscara con todos los bits encendidos.

Para calcular $DP[S]$, identificamos el costo del **último viaje** que condujo al estado S. Fijamos una maleta de referencia i que debe haber sido recogida en el último viaje (por ejemplo, la maleta con el índice de bit más bajo).

- Sea i la maleta de menor índice en S que fue recogida en el último viaje.
- Sea P_k la posición de la maleta k, y P_0 la posición del avión $(0, 0)$.
- Se define "apagar un bit" con la operacion XOR tal que S xor $\{i, j\}$ es aplicar S xor $i \wedge S$ xor j.

La recurrencia busca minimizar el tiempo entre dos posibles escenarios para el último viaje:

1. **Opción 1: Viaje de una maleta.** Solo se recogió i.

$$Costo_1 = DP[S \text{ xor } \{i\}] + T(P_0, P_i, P_0)$$

Donde $T(P_0, P_i, P_0) = d(P_0, P_i)^2 + d(P_i, P_0)^2$.

2. **Opción 2: Viaje de dos maletas.** Se recogió i junto con otra maleta j. Debemos iterar sobre todas las maletas $j \in S, j \neq i$.

$$Costo_2 = DP[S \text{ xor } \{i, j\}] + T(P_0, P_i, P_j, P_0)$$

Donde $T(P_0, P_i, P_j, P_0) = d(P_0, P_i)^2 + d(P_i, P_j)^2 + d(P_j, P_0)^2$.

Debido a que el costo de transitar es simétrico (es el cuadrado de la distancia cartesiana), el tiempo total de la ruta es independiente del orden en que se recogen i y j si el viaje empieza y termina en P_0. Por lo tanto, solo calculamos una vez el costo del viaje completo:

$$T_{i,j} = \text{Dist}(P_0, P_i)^2 + \text{Dist}(P_i, P_j)^2 + \text{Dist}(P_j, P_0)^2 = T_{j,i}$$

La recurrencia final es:

$$DP[S] = \min(Costo_1, Costo_2)$$

Complejidad

- **Número de Estados:** 2^n estados.
- **Costo de Transición:** Para cada estado, se calcula el costo de un viaje simple ($O(1)$) y se itera sobre $O(n)$ posibles parejas j para el viaje doble. La operación del Bitmasking y el cálculo de la distancia (asumido $O(1)$) es rápido.
- **Complejidad Total:** $O(n \cdot 2^n)$ tiempo, cumpliendo con la restricción.
- **Memoria:** $O(2^n)$ para la tabla de memoización.

Implementación en C++

A continuación, se presenta la implementación del algoritmo usando C++ Top-Down DP (Memoización)

El archivo funcional se encuentra en airplane_bags.cpp

```cpp
/**
 * @brief Estructura para representar las coordenadas de un punto.
 */
struct Point {
    int x, y;
};

// Variables globales para el contexto del problema
int N_Bags;          // Número de maletas (n)
const int INF = 1e9; // Constante para infinito
vector<Point> P;     // Puntos del aeropuerto. P[0] es el avión.
vector<int> memo;    // Tabla de memoización DP[S]

/**
 * @brief Calcula el cuadrado de la distancia cartesiana (tiempo de tránsito).
 *
 * Se asume que esta operación es O(1).
 * @param p1 Primer punto.
 * @param p2 Segundo punto.
 * @return ll El cuadrado de la distancia.
 */
int squared_distance(Point p1, Point p2) {
    return pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2);
}

/**
```

```
 * @brief Calcula el tiempo mínimo para recoger las maletas definidas por la máscara S.
 *
 * Implementa la Programación Dinámica Top-Down con Bitmasking.
 * Complejidad temporal total: O(N * 2^N).
 *
 * @param S Máscara de bits que representa el subconjunto de maletas ya recogidas (1-indexe
 * @return ll Tiempo mínimo.
 */
int solve_dp(int S) {
    // Caso Base: Si no quedan maletas por recoger (S=0), el tiempo es 0.
    if (S == 0)
        return 0;

    // Consulta la tabla de memoización (Memoization)
    if (memo[S] != -1)
        return memo[S];

    int min_time = INF;

    // 1. Identificar la maleta de referencia (i).
    // Buscamos el bit menos significativo encendido, que corresponde a la
    // maleta de menor índice que aún no ha sido considerada en el subproblema.
    // Esto asegura que cada estado S solo se resuelva una vez con una maleta
    // de inicio fija, manteniendo la complejidad en O(N * 2^N).

    int LSB_mask = S & -S;
    // log2(LSB_mask) nos da el índice del bit (0-based)
    int i_bit_idx = __builtin_ctz(LSB_mask);

    // i_maleta es el índice real de la maleta (1-based index)
    int i_maleta = i_bit_idx + 1;

    // Máscara sin la maleta i
    int S_without_i = S ^ LSB_mask;

    // Costo de ida y vuelta al avión (P) para la maleta i
    int cost_0_i = squared_distance(P[0], P[i_maleta]);

    // 2. Opción 1: Recoger la maleta i sola. (Viaje: P -> P[i] -> P)
    // Costo total = tiempo anterior + costo del viaje 0->i->0
    int cost_single = cost_0_i + cost_0_i;
    min_time = min(min_time, solve_dp(S_without_i) + cost_single);

    // 3. Opción 2: Recoger la maleta i junto con otra maleta j (j > i)
    // Recorremos todas las otras maletas j cuyo bit esté encendido en S_without_i.
```

```
    for (int j_bit_idx = 0; j_bit_idx < N_Bags; ++j_bit_idx) {
        // Comprobar si el bit j está encendido en la máscara S_without_i
        // y si j_bit_idx es mayor que i_bit_idx (para evitar simetría y
        // redundancia en el bucle)
        if ((S_without_i & (1 << j_bit_idx))) {
            int j_maleta = j_bit_idx + 1;

            // Máscara sin la maleta i ni la maleta j
            int S_without_i_j = S_without_i ^ (1 << j_bit_idx);

            int cost_0_j = squared_distance(P[0], P[j_maleta]);
            int cost_i_j = squared_distance(P[i_maleta], P[j_maleta]);

            // Costo del viaje doble: P -> P[i] -> P[j] -> P.
            // Debido a la simetría de la distancia al cuadrado, el orden
            // de recogida (i->j o j->i) no altera el costo total del circuito.
            int cost_double_trip = cost_0_i + cost_i_j + cost_0_j;

            min_time = min(min_time, solve_dp(S_without_i_j) + cost_double_trip);
        }
    }

    // Almacenar el resultado (Memoization) y retornarlo.
    return memo[S] = min_time;
}

// Función principal para resolver el problema (implementación de la Tarea 4, P4)
int solve_problem_4(const vector<Point> &bag_positions) {
    int n = bag_positions.size();
    if (n == 0)
        return 0;

    N_Bags = n;

    // Punto (0, 0) es el avión.
    P.clear();
    P.push_back({0, 0});

    // P a P[n] son las maletas.
    P.insert(P.end(), bag_positions.begin(), bag_positions.end());

    int num_states = 1 << N_Bags;

    // Inicialización de la tabla de memoización con -1 (INF si usamos Bottom-Up)
    memo.assign(num_states, -1);
```

```
    // El estado final es cuando todos los bits están encendidos
    int final_mask = num_states - 1;

    return solve_dp(final_mask);
}
```