Universidad Simón Bolívar
División de Ciencias Físicas y Matemáticas
Departamento de Computación y Tecnología de la Información
CI5651: Diseño de Algoritmos I

Profesor: Ricardo Monascal. **Estudiante**: Astrid Alvarado, 18-10938.

Tarea 4 (9 %):

1. Pregunta 1

Se procede a construir la tabla para decidir la distancia de edición entre **martes** y **mayo** (fecha de nacimiento: **07 de mayo de 2002**). Para esto, se tendrá que armar una tabla 4×6 de la siguiente forma:

i \ j	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	2	3	4	5	6
1	1						
2	2						
3	3						
4	4						

A = m a r t e s
1 2 3 4 5 6

B = m a y o
1 2 3 4

Note que los valores en la misma corresponde a la incialización de tabla con `d[i][0] = i` y `d[0][j] = j`. Ahora, se procede a calcular los valores como:

- Primera iteración: `i = 1`
  - `d[1][1] = d[0][0] = 0 (A[1] = B[1])`
  - `d[1][2] = 1 + min(d[0][2], d[1][1], d[0][1]) = 1 + 0 = 1`
  - `d[1][3] = 1 + min(d[0][3], d[1][2], d[0][2]) = 1 + 1 = 2`
  - `d[1][4] = 1 + min(d[0][4], d[1][3], d[0][3]) = 1 + 2 = 3`
  - `d[1][5] = 1 + min(d[0][5], d[1][4], d[0][4]) = 1 + 3 = 4`
  - `d[1][6] = 1 + min(d[0][6], d[1][5], d[0][5]) = 1 + 4 = 5`

  Resultando en la siguiente tabla:

i \ j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2						
3	3						
4	4						

A = m a r t e s   B = m a y o
1 2 3 4 5 6   1 2 3 4

- Segunda iteración: `i = 2`
  - `d[2][1] = 1 + min(d[1][1], d[2][0], d[1][0]) = 1 + 0 = 1`
  - `d[2][2] = d[1][1] = 0` (`A[2] = B[2]`)
  - `d[2][3] = 1 + min(d[1][3], d[2][2], d[1][2]) = 1 + 0 = 1`
  - `d[2][4] = 1 + min(d[1][4], d[2][3], d[1][3]) = 1 + 1 = 2`
  - `d[2][5] = 1 + min(d[1][5], d[2][4], d[1][4]) = 1 + 2 = 3`
  - `d[2][6] = 1 + min(d[1][6], d[2][5], d[1][5]) = 1 + 3 = 4`

  Resultando en la siguiente tabla:

i \ j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3						
4	4						

A = m a r t e s   B = m a y o
1 2 3 4 5 6   1 2 3 4

- Tercera iteración: `i = 3`
  - `d[3][1] = 1 + min(d[2][1], d[3][0], d[2][0]) = 1 + 1 = 2`
  - `d[3][2] = 1 + min(d[2][2], d[3][1], d[2][1]) = 1 + 0 = 1`
  - `d[3][3] = 1 + min(d[2][3], d[3][2], d[2][2]) = 1 + 0 = 1`
  - `d[3][4] = 1 + min(d[2][4], d[3][3], d[2][3]) = 1 + 1 = 2`

- `d[3][5] = 1 + min(d[2][5], d[3][4], d[2][4]) = 1 + 2 = 3`
- `d[3][6] = 1 + min(d[2][6], d[3][5], d[2][5]) = 1 + 3 = 4`

Resultando en la siguiente tabla:

i \ j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3	2	1	1	2	3	4
4	4						

A = m a r t e s
1 2 3 4 5 6   B = m a y o
1 2 3 4

- Cuarta iteración: `i = 4`
  - `d[4][1] = 1 + min(d[3][1], d[4][0], d[3][0]) = 1 + 2 = 3`
  - `d[4][2] = 1 + min(d[3][2], d[4][1], d[3][1]) = 1 + 1 = 2`
  - `d[4][3] = 1 + min(d[3][3], d[4][2], d[3][2]) = 1 + 1 = 2`
  - `d[4][4] = 1 + min(d[3][4], d[4][3], d[3][3]) = 1 + 1 = 2`
  - `d[4][5] = 1 + min(d[3][5], d[4][4], d[3][4]) = 1 + 2 = 3`
  - `d[4][6] = 1 + min(d[3][6], d[4][5], d[3][5]) = 1 + 3 = 4`

  Resultando en la siguiente tabla:

i \ j	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	0	1	2	3	4
3	3	2	1	1	2	3	4
4	4	3	2	2	2	3	4

A = m a r t e s
1 2 3 4 5 6   B = m a y o
1 2 3 4

Con esta tabla resultante, se puede determinar que la distancia de edición para **martes** y **mayo** es `d[4][6] = 4`

2. Pregunta 2

Para resolver este problema se planteará una implementación *Bottom-up*. Para esto, se define la tabla `DP` ($n \times n$) en donde se cumpla lo siguiente:

$$\texttt{DP[i][j]} = \textbf{la longitud del par de subarreglos familiares}$$
$$\textbf{más largos que comienza en las posiciones } i \textbf{ y } j$$

El objetivo entonces es encontrar el valor máximo en esta tabla. El cálculo de cada celda `DP[i][j]` sigue la siguiente lógica:

a) **Casos Base (No Familiares):**

- Si $i = j$, los subarreglos $A[i..i]$ y $A[j..j]$ no son disjuntos. Por lo tanto, $\texttt{DP[i][j]} = 0$.
- Si $mcd(\texttt{A[i]}, \texttt{A[j]}) \neq 1$, los elementos iniciales no son coprimos, por lo que no pueden iniciar un par familiar. Por lo tanto, $\texttt{DP[i][j]} = 0$.

b) **Caso Recursivo (Coprimos y Disjuntos):**
Si $mcd(\texttt{A[i]}, \texttt{A[j]}) = 1$ y $i \neq j$, entonces se puede formar *al menos* un par familiar de longitud 1. Con esto, se puede intentar extender un par existente que comienza en `DP[i+1][j+1]`, dando una longitud *potencial* $k = 1 + \texttt{DP[i+1][j+1]}$.

Sin embargo, a los subarreglos resultantes $A[i..i+k-1]$ y $A[j..j+k-1]$ se debe **verificar la condición de ser disjuntos** para poder extender la longitud:

- Si los subarreglos **son disjuntos**, la longitud es k, por lo tanto $\texttt{DP[i][j]} = k$.
- Si los subarreglos **NO son disjuntos**, entonces no se puede intentar extender la longitud. Sin embargo, como $mcd(\texttt{A[i]}, \texttt{A[j]}) = 1$, se sabe que el par $A[i..i]$ y $A[j..j]$ sí es un par familiar válido de longitud 1. Por lo tanto, $\texttt{DP[i][j]} = 1$.

Note que para determinar si $A[i..i+k-1]$ y $A[j..j+k-1]$ son disjuntos, basta con ver que $i+k-1 < j$ o $j+k-1 < i$. Además, dado que para este punto se asegura que $i \neq j$, entonces $k = 0$ es un caso trivial para que los conjuntos sean disjuntos.

Esto nos lleva a la siguiente recursión:

$$\texttt{DP[i][j]} = \begin{cases} 0 & \text{si } i = j \vee mcd(\texttt{A[i]}, \texttt{A[j]}) \neq 1 \\ 1 & \text{si } mcd(\texttt{A[i]}, \texttt{A[j]}) = 1 \text{ y NO son disjuntos} \\ k = 1 + \texttt{DP[i+1][j+1]} & \text{si } mcd(\texttt{A[i]}, \texttt{A[j]}) = 1 \text{ y son disjuntos} \end{cases}$$

Cabe destacar que se debe manejar el caso borde donde $i+1$ o $j+1$ están fuera de los límites, en cuyo caso $\texttt{DP[i+1][j+1]} = 0$. Con esta recursión se puede observar que para calcular cualquier celda en la **fila** i, solo necesitamos conocer los valores de la **fila** $i+1$. Por lo tanto, no es necesario almacenar la tabla completa. En su lugar, se puede usar solo **dos vectores** de tamaño n:

- `prev`: Almacena los valores de la fila $i+1$ (la que se acaba de calcular).
- `curr`: Almacena los valores de la fila i (la que se está calculando actualmente).

Otra cosa a notar es que, debido a la dependencia de `DP[i+1][j+1]`, se debe llenar la tabla desde el final hacia adelante. En cada paso (i, j), se calcula `curr[j]` usando los valores de `prev` (específicamente `prev[j+1]`). Después de completar una fila i, `curr` se convierte en `prev` para la siguiente iteración (la fila $i-1$). La respuesta final será el valor máximo de `curr[j]` que se actualiza en cada paso (i, j).

Con este algoritmo, la complejidad de tiempo total es $O(n^2 \cdot log(M))$ Donde M es el elementos más grande del arreglo A. Como $log(M)$ es una constante, se tiene entonces como cota superior $O(n^2)$. Por otro lado, la complejidad de memoria adicional es $O(n)$ gracias al uso de `prev` y `curr`.

La implementación del programa se encuentra en el siguiente enlace. Para ejecutarlo, leer el `README.md`

3. Pregunta 3

La implementación del programa para la inicialización virtual de arreglos se encuentra en el siguiente enlace. Para ejecutarlo, leer el `README.md`

4. Pregunta 4

Para resolver este problema se hará uso una máscara de bits (un entero *mask*) para representar un subconjunto de maletas. Si el i-ésimo bit de *mask* está encendido (es 1), significa que la maleta i pertenece a ese subconjunto.

Por lo tanto, se define un arreglo `dp`, donde:

$$\texttt{dp[mask]} = \text{Costo mínimo para recoger el conjunto de maletas } mask.$$

El tamaño del arreglo `dp` es 2^n pues es la cantidad de subconjuntos totales para n maletas. Con esto, se procede a definir además dos arreglos con el fin de optimizar la recurrencia al pre-calcular los costos de los viajes simples y dobles:

- `simple_cost[i]`: Costo de Origen $\to i \to$ Origen.
- `double_cost[i][j]`: Costo mínimo de (Origen $\to i \to j \to$ Origen) o (Origen $\to j \to i \to$ Origen).

Así, se plantea el siguiente algoritmo:

- **Caso Base:** El costo de recoger un conjunto vacío de maletas es 0.

$$\texttt{dp}[0] = 0$$

- **Caso Recursivo:** se calcula el estado `dp[mask]` para todas las máscaras desde 1 hasta $(1 \ll n) - 1$ (o $2^n - 1$).

  Para esto, se considera el **último viaje** que completó la recolección de este conjunto. Por tanto, para cualquier *mask*, se fija un elemento i_{fija} que pertenezca a *mask*, en particular, el elemento con el índice más bajo. El último viaje que completó *mask* *tuvo* que incluir a i_{fija}, dejando así solo dos posibilidades:

  a) **El último viaje fue simple (solo por i_{fija}):** El estado anterior era *mask* sin i_{fija}.

  $$\text{Costo}_1 = \texttt{dp}[mask \texttt{ XOR } (1 \ll i_{\text{fija}})] + \texttt{simple\_cost}[i_{\text{fija}}]$$

  b) **El último viaje fue doble (por i_{fija} y otra maleta j):** El estado anterior era *mask* sin i_{fija} y sin j. Se debe iterar sobre todas las posibles j en la máscara $(j \neq i_{\text{fija}})$.

  $$\text{Costo}_2 = \min_{j \in mask, j > i_{\text{fija}}} (\texttt{dp}[mask \texttt{ XOR } (1 \ll i_{\text{fija}}) \texttt{ XOR } (1 \ll j)] + \texttt{double\_cost}[i_{\text{fija}}][j])$$

El valor final para `dp[mask]` es el mínimo de todas estas posibilidades.

$$\texttt{dp[mask]} = \min(\text{Costo}_1, \text{Costo}_2)$$

La respuesta al problema es el estado donde todas las maletas han sido recogidas, representado por: `dp[(1 ≪ n) - 1]` (es decir, la máscara con n bits encendidos).

Con este programa, la complejidad de tiempo del algoritmo está determinada por el cálculo de la tabla `dp` (que es la operación más costosa).

- El bucle externo itera sobre todas las 2^n máscaras. $O(2^n)$.
- Dentro de cada iteración de *mask*, el algoritmo se encarga de:
  - Encontrar i_{fija} (que es el primer bit encendido), en el peor de los casos, toma $O(n)$.
  - El cálculo de Costo_1 toma $O(1)$.
  - El cálculo de Costo_2 requiere un bucle sobre j que, en el peor de los casos, toma $O(n)$.

La complejidad total es: $O(2^n) \times O(n) = O(n \times 2^n)$ (el pre-cálculo, que toma $O(n^2)$, es dominado por este término). Por otro lado, la estructura de datos principal es el arreglo `dp` de tamaño 2^n y la memoria para el pre-cálculo de costos es $O(n^2)$. Por lo tanto, el algoritmo toma $O(2^n)$ en memoria.

La implementación del programa se encuentra en el siguiente enlace. Para ejecutarlo, leer el `README.md`