Universidad Simón Bolívar
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Resuelto por:
Christopher Gómez

Tarea 4: Programación dinámica

1. Considere el algoritmo de *Programación Dinámica* propuesto en clase para el problema de distancia de edición entre dos cadenas de caracteres.

   Construya la tabla correspondiente al proceso para decidir la distancia de edición entre su **primer nombre** y su **primer apellido**, *sin ahorro de memoria*.

   Por ejemplo, si su nombre es *Fulano Mengano Apellidez Surnameson*, se desea que construya la tabla 6×9 para la distancia de edición entre `fulano` y `apellidez`.

   Para *Christopher Gómez* la tabla es:

		g	**o**	**m**	**e**	**z**
	0	1	2	3	4	5
c	1	1	2	3	4	5
h	2	2	2	3	4	5
r	3	3	3	3	4	5
i	4	4	4	4	4	5
s	5	5	5	5	5	5
t	6	6	6	6	6	6
o	7	7	6	7	7	7
p	8	8	7	7	8	8
h	9	9	8	8	8	9
e	10	10	9	9	8	9
r	11	11	10	10	9	9

   La tabla resultante se puede interpretar como que la distancia de edición entre `christopher` y `gomez` es 9, y se logra eliminando `chris`, intercambiando la `t` por una `g`, eliminando la `p` luego de la `o` e intercambiando la `h` y la `r` por una `m` y una `z` respectivamente. Esto es:

   - `christopher` $\rightarrow$ `topher` (5 eliminaciones).
   - `topher` $\rightarrow$ `gopher` (1 reemplazo).
   - `gopher` $\rightarrow$ `goher` (1 eliminación).
   - `goher` $\rightarrow$ `gomez` (2 reemplazos).

2. Sea $A[1..n]$ un arreglo de enteros.

   Decimos que B es un sub-arreglo de A si se pueden remover elementos del arreglo A, respetando el orden en el que aparecen, para obtener B.

   Decimos que $B[1..k]$ es bueno si el arreglo no está vacío y para todo i, tal que $1 \leq i \leq k$ se cumple que $B[i]$ es divisible entre i.

   Consideramos que dos sub-arreglos son diferentes si provienen de posiciones diferentes en A, incluso si los valores son iguales.

   Por ejemplo, si $A = [2, 2, 1, 22, 15]$

   Los sub-arreglos buenos serían 13, que son:

   - $[2]$
   - $[2, 2]$
   - $[2, 2, 15]$
   - $[2, 22]$
   - $[2, 22, 15]$
   - $[2]$
   - $[2, 22]$
   - $[2, 22, 15]$
   - $[1]$
   - $[1, 22]$
   - $[1, 22, 15]$
   - $[22]$
   - $[15]$

   Queremos saber la cantidad de sub-arreglos buenos de A.

   Se desea que diseñe un algoritmo usando *Programación Dinámica*, que resuelva este problema en tiempo $O(n^2)$ y con memoria adicional $O(n)$.

   Para resolver este problema vamos a definir una tabla $C[1..n][1..n]$, donde $C[i][j]$ es la cantidad de sub-arreglos buenos de tamaño i contienen al elemento $A[j]$. Notemos que:

$$C[i][j] = \begin{cases} 1 & \text{si } i = 1 \\ \sum_{k=1}^{j-1} C[i-1][k] & \text{si } i > 1 \text{ y } A[j] \text{ es divisible entre } i \\ 0 & \text{de otra forma} \end{cases}$$

   - Para el caso base, como cualquier entero es divisible entre 1, existe un sub-arreglo bueno de tamaño 1 que contiene a cada elemento de A.
   - Si $A[j]$ no es divisible entre i, entonces no existe un sub-arreglo bueno de tamaño i que contenga a $A[j]$.
   - Para el caso recursivo, la cantidad de sub-arreglos buenos de tamaño i que contienen a $A[j]$ es igual a la cantidad de sub-arreglos buenos de tamaño $i-1$ que se pueden formar con los elementos anteriores a $A[j]$, para luego agregarle $A[j]$ al final.
   - La solución al problema es la suma de todos los valores de la tabla C.

   Con esto, podemos diseñar un algoritmo de Programación Dinámica Bottom-Up que calcule la tabla C y devuelva la suma de todos sus valores. Veamos que uno de los casos de $C[i][j]$ depende de calcular la suma de los valores de la fila anterior de la tabla, que con el uso astuto de acumuladores se puede hacer mientras se construye la tabla, evitando que el algoritmo sea de tiempo $O(n^3)$.

Por otro lado, como para cada fila se necesita solamente la fila anterior, se pueden usar dos arreglos de tamaño n para almacenar la fila actual y anterior, y otro acumulador para el resultado, ya que no se tendrá toda la tabla para calcular la suma al final. Esto nos da un algoritmo de tiempo $O(n^2)$ y memoria adicional $O(n)$, veamos el pseudocódigo:

```
def cantidad_subarreglos_buenos(A: Arreglo[Entero]) -> Entero:
    n = A.tamaño
    C1 = [1] * n

    res = 0
    for j in [2..n]:
        C0 = C1 # Fila anterior de C
        C1 = [0] * n # Fila a calcular

        acum = 0 # Suma acumulada de C0
        for i in [1..n]:
            if A[j] % i == 0:
                C[i] += acum
                res += acum
            acum += C0[j]
    return res
```

Para el ejemplo dado, la tabla C sería:

	2	**2**	**1**	**22**	**15**
1	1	1	1	1	1
2	0	1	0	3	0
3	0	0	0	0	4
4	0	0	0	0	0
5	0	0	0	0	0

Y la suma de todos sus valores es: $1+1+1+1+1+1+3+4=13$.

Una implementación de este algoritmo en Bash se puede encontrar aquí.

3. Se desea que implemente, en el lenguaje de su elección, un cliente para probar el proceso de inicialización virtual de arreglos. Su programa debe cumplir con las siguientes características:

   - Al invocarse, debe recibir como argumento del sistema el tamaño del arreglo a utilizar.
   - El arreglo será indexado a cero (las posiciones válidas, para un arreglo de tamaño n, serán desde la 0 hasta la $n - 1$).

     Debe presentar al usuario un cliente con cuatro opciones:

     - `ASIGNAR POS VAL`, que tiene el efecto de asignar el valor `VAL` en la posición POS del arreglo.
       Su programa debe reportar un error si la posición `POS` no es una posición válida del arreglo.
     - `CONSULTAR POS`, que debe reportar si la posición `POS` está inicializada o no. En caso de estar inicializada, debe devolver el valor asociado a esa posición.
       Su programa debe reportar un error si la posición `POS` no es una posición válida del arreglo.
     - `LIMPIAR`, que tiene el efecto de limpiar la tabla y hace que **todas** las posiciones queden sin inicializar.
     - `SALIR`, que sale del programa.
   - Todas las operaciones involucradas en su programa deben tomar tiempo $\Theta(1)$.
   - Debe implementar estas operaciones siguiendo el proceso de inicialización virtual visto en clase, usando los dos arreglos auxiliares, **no** con métodos alternativos (como tablas de hash o tipos conjuntos de la librerías de su lenguaje de escogencia).

   Una implementación de este cliente en C se puede encontrar aquí.

4. ¡Esta aerolinea es un desastre! El transporte automatizado de equipaje se ha descompuesto y ha dejado las maletas de los pasajeros regadas por toda la pista. Es tu trabajo volver a recogerlas todas y colocarlas en el avión. Pero, ¡de prisa! Mientras más tiempo tomes, más se retrasará el vuelo en salir.

   Decides hacer una lista de todas las cosas que sabes, para organizarte mejor:

   - La cantidad, n, de equipajes que se han caído. También sabes que $1 \leq n \leq 24$.
   - La posición (x, y) de cada equipaje. Además, dadas las dimensiones del aeropuerto, sabes que $|x| \leq 100$ y $|y| \leq 100$ y que ambas dimensiones son números enteros.
   - El avión está en la posición $(0, 0)$ y es a donde quieres llevar todas las maletas que se han dispersado.
   - Transitar entre dos puntos a y b, toma tanto tiempo como **el cuadrado de la distancia cartesiana** entre ellos.
   - Solamente te puedes detener en la posición de una maleta o en la del avión, por temor a que te multen las autoridades del aeropuerto.
   - Puedes cargar a lo sumo dos maletas a la vez (una en cada mano) y una vez que tomas una maleta, solamente puedes soltarla en el avión.

   ¿Cuál es la mínima cantidad de tiempo necesaria para recoger todas las maletas?

   Se desea que diseñe un algoritmo usando *Programación Dinámica*, que resuelva este problema en tiempo y memoria $O(n \times 2^n)$.

   Sea S el conjunto total de las posiciones de las maletas, definamos una tabla $memo[R]$ tal que $memo[R]$ representa el tiempo mínimo necesario para recoger y llevar todas las maletas en $S \setminus R$ al avión, donde $R \subseteq S$. Así, R representa el subconjunto de maletas restantes por recoger y la solución está en $memo[\emptyset]$, es decir, cuando ya no hay maletas por recoger.

   Notemos que:

   - $memo[S] = 0$, ya que no se ha recogido ninguna maleta.
   - Para el caso recursivo, definamos la función $g : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}^*$, que devuelve el tiempo que toma transportarse de una posición a otra. Así, tenemos que, sea O la posición del avión:

     $$memo[R] = \min_{A \in S \setminus R} \begin{pmatrix} memo[R \cup \{A\}] + 2 \times g(O, A), \\ \min_{B \in S \setminus R}^{A \neq B} (memo[R \cup \{A, B\}] + g(O, A) + g(A, B) + g(B, 0)) \end{pmatrix}$$

     Así, para un subconjunto R se toma el mínimo entre:

     - El mínimo tiempo acumulado si se decide recoger una maleta en $S \setminus R$ y llevarla al avión.
     - El mínimo tiempo acumulado si se deciden recoger dos maletas en $S \setminus R$ y llevarlas al avión.

Así, un algoritmo de Programación Dinámica Top-Down que resuelve este problema es (sean n la cantidad de maletas, S el conjunto de posiciones de las maletas, O la posición del avión y g la función que devuelve el tiempo que toma transportarse de una posición a otra, implicitos):

```
memo = [None] * (2^n) # Para P(S) subconjuntos de S
ryanairDP({}, memo) # Llamada que resuelve el problema

def ryanairDP(R: Conjunto[Maleta], memo: Arreglo[Entero]):
    if R == S:
        return 0

    if memo[R] is not None:
        return memo[R]

    A = (S-R)[0] # Escoge alguna maleta que no se haya recogido

    # Se calcula el mínimo tiempo si se decide llevarla sola o con otra maleta
    mintiempo = ryanairDP(R U {A}, memo) + 2 * g(O, A)
    for B in S-R-{A}:
        costo = ryanairDP(R U {A, B}, memo) + g(O, A) + g(A, B) + g(B, O)
        if  costo < mintiempo:
            mintiempo = costo

    memo[R] = mintiempo
    return memo[R]
```

Como se puede ver, al no importar el orden en que se recogen las maletas (solamente cuántas y cuáles se recogen en cada paso), se puede tomar una maleta cualquiera que no se haya recogido y calcular el mínimo tiempo si se decide llevarla sola o con otra maleta. Como se llegan a considerar todos los subconjuntos de S, que son 2^n, y para cada uno se halla el mínimo entre las opciones en tiempo $O(n)$, el algoritmo toma tiempo $O(n \times 2^n)$.

Se puede hacer este algoritmo muy eficiente usando manipulación de bits para representar los subconjuntos de S y un arreglo de tamaño 2^n para almacenar la tabla $memo$. Se puede encontrar una implementación de este algoritmo en Julia aquí.