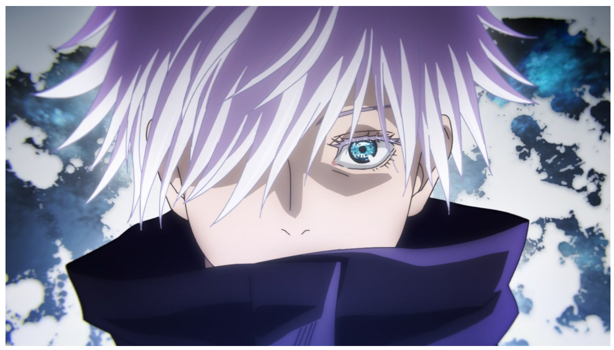
Problemas de programación dinámica



José Antonio Hernández López

Departamento de Informática y Sistemas
Universidad de Murcia

24 de marzo de 2025

Índice

1. Segmentación de palabras

1.1. Descripción del problema

Dado un texto sin espacios y un diccionario de palabras, devolver si es posible fragmentarlo de manera que todos los fragmentos pertenezcan al diccionario. Por ejemplo, dado el diccionario:

$$\text{diccionario} = \{\text{i, love, samsung}\}$$

Ejemplos de entradas y salidas son:

$$\text{entrada}_1 = \text{ilovesamsung} \Rightarrow \text{salida}_1 = \text{true}$$

$$\text{entrada}_2 = \text{ilovesamsungi} \Rightarrow \text{salida}_2 = \text{true}$$

$$\text{entrada}_3 = \text{ilovesamsunga} \Rightarrow \text{salida}_3 = \text{false}$$

$$\text{entrada}_4 = \text{xilovesamsung} \Rightarrow \text{salida}_4 = \text{false}$$

1.2. Ecuación de recurrencia

Sea $P(i)$, un función que devuelve `true` si la subcadena $s[0, i]$ (de los i primeros carácteres) puede segmentarse usando palabras del diccionario (y `false` en caso contrario). Entonces la relación de recurrencia es:

$$P(i) = \bigvee_{j=0}^{i-1} [P(j) \wedge s[j:i] \in \text{diccionario}]$$

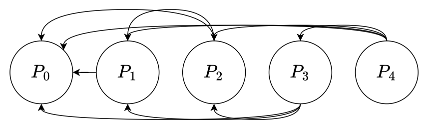


Figura 1: DAG de la segmentación de palabras

Es decir, $P(i)$ es `true` si existe algún j tal que $P(j)$ sea `true` (es decir, se pueda segmentar la cadena de 0 hasta j) y que el resto de la cadena (desde j hasta i) sea una palabra del diccionario. El caso base es simplemente que $P(i) =$ `true` si $i = 0$ (cadena vacía). De aquí en adelante $P(i)$ será representado como P_i.

1.3. Dimensiones de la tabla

La ecuación de recurrencia depende de un solo parámetro. Así pues, usaremos una tabla unidimensional. Por otro lado, tenemos $n + 1$ problemas diferentes $(P_0, P_1, \ldots, P_n)$ siendo n la longitud de la cadena. Por ello, la tabla tendrá tamaño $n + 1$.

1.4. DAG de dependencias y orden topológico

La Figura 1 muestra el DAG asociado a una palabra de 4 caracteres.

- P_4 depende de P_3, P_2, P_1, P_0.
- P_3 depende de P_2, P_1, P_0.
- P_2 depende de P_1, P_0.
- P_1 depende de P_0.
- P_0 es el caso base.

Así pues, el orden topológico con sentido es: $P_0, \ldots, P_n$. Siendo P_0 el caso base.

1.5. Implementación y orden

La implementación simplemente consiste en generar una tabla y recorrerla desde $i = 0$ hasta n e ir calculando $P_0, \ldots, P_n$ (orden topológico).

```python
def fragmentacion(palabra, diccionario):
    tabla = [False] * (len(palabra) + 1)
    for i in range(len(tabla)):
        if i == 0:
            tabla[0] = True
        else:
```

```
        r = False
        for j in range(i):
            r = r or (tabla[j] and (palabra[j:i] in diccionario))
        tabla[i] = r
    return tabla[len(palabra)], tabla

diccionario = {"i", "love", "samsung"}
palabra = "ilovesamsungi"
resultado, _ = fragmentacion(palabra, diccionario)
# True
```

El orden de este algoritmo es $\Theta(n^2)$ ya que son dos bucles anidados $i = 0, \ldots, n$ y $j = 0, \ldots, i-1$. Por otro lado la memoria es $\Theta(n)$ ya que se necesita una tabla unidimiensional de tamaño n.

1.6. Reconstrucción de la solución

La reconstrucción de la solución (sucessión de palabras del diccionario que forman la cadena) consiste en la siguiente idea:

1. Empezamos en el problema grande $P(n)$ y vemos cuál j hace `true` la expresión $P(j) \wedge s[j:i] \in$ diccionario.
2. Añadimos, para ese j, $s[j:i]$ a la reconstrucción y pasamos a $P(j)$.
3. Repetimos hasta que lleguemos a $j = 0$.

```
def reconstruccion(palabra, tabla, diccionario):
    i_actual = len(palabra)
    S = []
    if not tabla[i_actual]:
        return []
    while i_actual != 0:
        for j in range(i_actual):
            if tabla[j] and (palabra[j:i_actual] in diccionario):
                S.append(palabra[j:i_actual])
                i_actual = j
                break
    return S[::-1]

palabra = "ilovesamsungiiloveiisamsung"
_, tabla = fragmentacion(palabra, diccionario)
reconstruccion(palabra, tabla, diccionario)
# ['i', 'love', 'samsung', 'i', 'i', 'love', 'i', 'i', 'samsung']
```

1.7. Versión `memo`

Si usamos PD con memorización, solo tenemos que implementar la ecuación recurrente con un método recursivo y añadir una caché `memo` (usamos un diccionario).

```
def fragmentacion_memo(palabra, diccionario, memo, i):
    if i in memo:
        return memo[i]
    if i == 0:
        r = True
    else:
        r = False
        for j in range(i):
            r = r or (fragmentacion_memo(palabra, diccionario, memo, j)
                and (palabra[j:i] in diccionario))
    memo[i] = r
    return r

memo = {}
palabra = "ilovesamsungiiloveiisamsung"
fragmentacion_memo(palabra, diccionario, memo, len(palabra))
# True
```

Para caluclar el orden en este caso simplemente tenemos que ver el tiempo no recursivo que se tarda en resolver cada subproblema:

- Para calcular P_i de $i = 1$ hasta n, tenemos que hacer un bucle $j = 0, \dots, i-1$. Así pues, $t_{P_i} \sim \Theta(i)$.
- P_0 es el caso base y require tiempo constante $\Theta(1)$.

Así pues, tenemos:

$$\sum_{i=1}^{n} \Theta(i) + \Theta(1) = \Theta(n^2)$$

2. Números de Catalán

2.1. Descripción del problema

Los números de Catalán son una secuencia que aparecen en varios problemas de conteo. Dicho números satisfacen la siguiente encuación:

$$C_n = \begin{cases} 1 & n = 0 \\ \displaystyle\sum_{i=1}^{n} C_{i-1} C_{n-i} & n \geq 1 \end{cases}$$

Queremos obtener el término $n-$ésimo.

2.2. Dimensiones de la tabla

La ecuación de recurrencia depende de un único parámetro y tenemos un total de $n+1$ problemas $C_0, \ldots, C_n$. Así pues, lo que encaja aquí es una tabla unidimensional de tamaño $n+1$.

2.3. DAG de dependencias y orden topológico

En este caso,

- $C_5 = C_0C_4 + C_1C_3 + C_2C_2 + C_3C_2 + C_4C_0$. Así pues, C_5 depende de C_0, C_1, C_2, C_3, C_4.
- $C_4 = C_0C_3 + C_1C_2 + C_1C_1 + C_3C_0$. Así pues, C_4 depende de C_0, C_1, C_2, C_3.
- ...

De este modo, el grafo de dependencias es el mismo que Figura 1 (cambiando P_i por C_i). El caso base es C_0.

2.4. Implemetación y orden

Para cada $i = 0, \ldots, n$ calculamos el número C_i usando los anteriores.

```
def catalan(n):
    tabla = [0]*(n+1)
    for i in range(0, n + 1):
        if i == 0:
            tabla[i] = 1
        else:
            s = 0
            for j in range(1, i + 1):
                s += tabla[j-1] * tabla[i - j]
            tabla[i] = s
    return tabla, tabla[n]

catalan(10)
# ([1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796], 16796)
```

El orden de este algoritmo es $\Theta(n^2)$ ya que tenemos dos bucles anidados. Por otro lado la memoria usada es $\Theta(n)$.

2.5. Versión `memo`

Usando un diccionario `memo`, tenemos:

```
def catalan_memo(n, memo):
    if n in memo:
        return memo[n]
```

```
    if n == 0:
        numero = 1
    else:
        numero = 0
        for j in range(1, n + 1):
            numero += catalan_memo(j - 1, memo) * catalan_memo(n - j, memo)
    memo[n] = numero
    return numero

memo = {}
catalan_memo(10, memo)
# 16796
memo
# {0: 1, 1: 1, 2: 2, 3: 5, 4: 14, 5: 42, 6: 132, 7: 429,
# 8: 1430, 9: 4862, 10: 16796}
```

Para calcular el orden vemos el tiempo no recursivo en resolver cada subproblema:

- Para C_i con $i \geq 1$ tenemos que hacer un bucle $j = 1 \dots i$. Esto da un orden $\Theta(i)$.
- Para C_0 el trabajo es constante.

Así pues, tenemos:

$$\sum_{i=1}^{n} \Theta(i) + \Theta(1) = \Theta(n^2).$$

3. Distancia de edición

3.1. Descripción del problema

Dadas dos cadenas `str1` y `str2`, queremos encontrar el mínimo número de operaciones para convertir `str1` en `str2`. Las operaciones permitidas son:

1. Insertar un carácter
2. Borrar un carácter
3. Reemplazar un carácter

Este valor mínimo de operaciones se conoce como distancia Levenshtein.

3.2. Ecuación de recurrencia

Denotemos $D(i, j)$ como el mínimo número de operaciones para convertir los primeros i carácteres de `str1` y los primeros j de `str2`. La ecuación recursiva es la siguiente:

$$
D(i,j) = \begin{cases} \text{máx}(i,j), & \text{si } \text{mín}(i,j) = 0 \\ D(i-1,j-1), & \text{si } \texttt{str1}[i] = \texttt{str2}[j] \\ 1 + \text{mín} \begin{cases} D(i-1,j) & \text{(elimino un carácter de } \texttt{str1}) \\ D(i,j-1) & \text{(inserto un carácter en } \texttt{str1}) \\ D(i-1,j-1) & \text{(reemplazo un carácter de } \texttt{str1}) \end{cases} & \text{en otro caso} \end{cases}
$$

La idea es la siguiente:

1. Si una de las dos cadenas es vacía, el número de operaciones es el de borrar/insertar todos los caracteres de la no vacía.

2. Si los dos últimos carácteres coinciden, no hay que hacer nada y no se incrementa la distancia.

3. Si los dos últimos carácteres no coinciden, entonces tenemos tres posibilidades que incrementará en uno en la distancia:

   - Elimino el último carácter de `str1` $\rightarrow D(i-1,j)$
   - Inserto un carácter al final de `str1` que sea igual al último carácter de `str2` $\rightarrow D(i,j-1)$.
   - Reemplazo el último carácter en `str1` haciendo que sea igual al último de `str2` $\rightarrow D(i-1,-j)$.

3.3. Dimensiones de la tabla

La ecuación de recurrencia depende de dos parámetros (i,j). Esto es indicativo de una tabla bidimensional. Por otro lado estamos interesados en $D(n,m)$ donde n es la longitud de `str1` y m es la longitud de `str2`. Así pues, la tabla será de $(n+1) \times (m+1)$.

3.4. DAG de dependencias y orden topológico

La Figura 2 muestra el DAG para el caso de las palabras `horse` y `ros`. Los casos base están marcados en verde. Un posible orden topológico sería $i = 0, \ldots, n$ y de manera anidada $j = 0, \ldots, m$.

3.5. Implementación y orden

Generamos y recorremos la tabla en el orden escogido:

```
def edit_distance(str1, str2):
    n, m = len(str1), len(str2)
    dp = [[0 for _ in range(m + 1)] for _ in range(n + 1)]
    for i in range(n + 1):
        for j in range(m + 1):
```

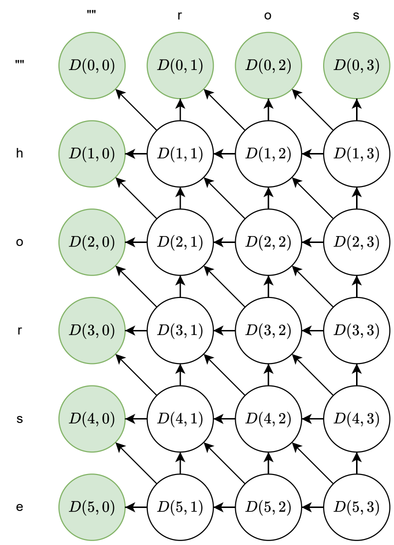


Figura 2: DAG de la distancia Levenshtein

```
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(
                    dp[i][j-1],      # inserta
                    dp[i-1][j],      # elimina
                    dp[i-1][j-1]     # reemplaza
                )
    return dp, dp[n][m]
```

El tiempo y memoria son $\Theta(nm)$ ya que recorremos una tabla $n \times m$.

3.6. Reconstrucción de la solución

Ahora estamos interesados en la secuencia de operaciones de edición. La idea es similar a los algoritmos de PD que hemos visto:

1. Empezamos con el problema grande $D(n,m)$ y vemos de qué subproblema viene el cálculo.

2. Añadimos la operación correspondiente y pasamos al subproblema más pequeño.
3. Repetimos hasta llegar a $D(0, 0)$.

```
def edit_distance_with_path(dp, str1, str2):
    edit_path = []
    i, j = len(str1), len(str2)
    while i > 0 or j > 0:
        if i > 0 and j > 0 and str1[i-1] == str2[j-1]:
            # caso base, caracteres iguales
            i -= 1
            j -= 1
        else:
            if i > 0 and j > 0 and dp[i][j] == dp[i-1][j-1] + 1:
                # reemplazar
                edit_path.append(f"Reemplaza '{str1[i-1]}' de la pos {i-1}"
                + "por '{str2[j-1]}' de la pos {j-1}")
                i -= 1
                j -= 1
            elif i > 0 and dp[i][j] == dp[i-1][j] + 1:
                # borrar
                edit_path.append(f"Borra '{str1[i-1]}' de la pos {i-1}")
                i -= 1
            else:
                # insertar
                edit_path.append(f"Inserta '{str2[j-1]}' en la pos {i-1}")
                j -= 1
    # dar la vuelta a la solución
    edit_path.reverse()
    return edit_path
```

3.7. Versión `memo`

La versión `memo` es la siguiente:

```
def edit_distance_memo(str1, str2, memo, i, j):
    if (i, j) in memo:
        return memo[(i, j)]
    if i == 0:
        r = j
    elif j == 0:
        r = i
    elif str1[i-1] == str2[j-1]:
        r = edit_distance_memo(str1, str2, memo, i - 1, j - 1)
    else:
        r = 1 + min(
```

```
            edit_distance_memo(str1, str2, memo, i, j - 1),      # inserta
            edit_distance_memo(str1, str2, memo, i - 1, j),      # elimina
            edit_distance_memo(str1, str2, memo, i - 1, j - 1)   # reemplaza
        )

    memo[(i, j)] = r
    return r

memo = {}
str1 = "horse"
str2 = "ros"
edit_distance_memo(str1, str2, memo, len(str1), len(str2))
# 3
```

Para ver el orden de este algoritmo, tenemos que sumar el tiempo no recursivo de todos los subproblemas. En este caso, todos los subproblemas se resuelven en $\Theta(1)$ (todos los casos, incluidos los casos base). En total hay $(n+1)(m+1)$ problemas $D(i,j)$ $i = 0, \ldots, n$ $j = 0, \ldots, m$. Así pues:

$$(n+1)(m+1)\Theta(1) = \Theta(nm).$$

4. Suma de un subconjunto

4.1. Descripción del problema

Dado un conjunto de números A, decidir si hay un subconjunto que sume M. Asumimos que todos los números de A y M son enteros. Por ejemplo, para

$$A = \{13, 11, 7\}, M = 20$$

La respuesta es sí: el subconjunto $\{13, 7\}$ suma 20.

$$A = \{13, 11, 7\}, M = 8$$

La respuesta es no.

4.2. Ecuación de recurrencia

Este problema es parecido al de la mochila y al del cambio de monedas. Supongamos que los número de A tienen un orden $a_1, \ldots, a_n$ y definimos $P(j,m)$ como la función booleana que dice si es cierto sumar m con un conjunto que tiene los elementos $a_1, \ldots, a_j$. En ese caso tenemos dos posiblidades: el objeto a_j es sumando o no:

$$P(j,m) = P(j-1,m) \vee P(j-1, m-a_j).$$

Los casos base son:

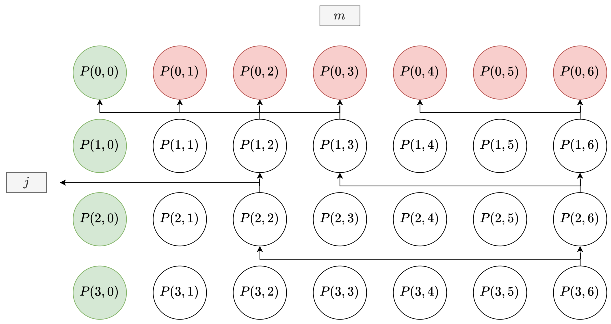


Figura 3: DAG de suma de un subconjunto

- Si $m < 0$ o $j < 0$, entonces $P(j, m) =$ `false` (me salgo de la tabla)
- Si $m = 0$ y $j \geq 0$, entonces $P(j, m) =$ `true`
- Si $j = 0$ y $m > 0$, entonces $P(j, m) =$ `false`

4.3. Dimensiones de la tabla

De manera similar a la mochila al del cambio, la tabla tiene dimensiones $(n + 1) \times (M + 1)$ donde n es el número de elementos de A.

4.4. DAG de dependencias y orden topológico

Supongamos $A = \{2, 3, 4\}$ y $M = 6$, entonces tenemos el DAG representeado en la Figura 3. Los casos base están pintados (en verde los `true` y en rojo los `false`). Se puede observar que los problemas de j dependen del nivel inferior. Así pues, un orden válido sería recorrer la tabla $j = 0, \ldots, n$ y, de manera anidada, $m = 0, \ldots, M$.

4.5. Implementación y orden

Construimos una tabla de dimensiones $(n + 1) \times (M + 1)$ y la rellenamos siguiendo el orden escogido:

```
def subsetsum(A, M):
    tabla = [[0 for _ in range(M + 1)] for _ in range(len(A) + 1)]

    for j in range(len(A) + 1):
        for m in range(M + 1):
            if m == 0 and j >= 0:
                tabla[j][m] = True
            elif j == 0 and m > 0:
                tabla[j][m] = False
```

```
        else:
            if m - A[j - 1] >= 0:
                r1 = tabla[j-1][m - A[j - 1]]
            else:
                r1 = False # me salgo de la tabla
            r2 = tabla[j-1][m]
            tabla[j][m] = r1 or r2
    return tabla, tabla[len(A)][M]
```

El orden y la memoria es $\Theta(nM)$ ya que tenemos simplemente que rellenar la tabla de dimensiones $(n+1) \times (M+1)$. Este algoritmo, como ocurre con el de la mochila, es pseudo-polinomial.

4.6. Reconstrucción de la solución

Empezamos con el problema más grande: $P(n, M)$. Si es `false`, entonces no hay nada que reconstruir. En caso contrario, vemos si añadiendo el elemento n en la suma obtenemos `true`. Si es así, añadimos el elemento n a la solución y restamos a M. En caso contrario, no añadimos nada. Pasamos al subproblema asociado a $n-1$ y así sucesivamente hasta llegar al problema $j=0$ (quedarnos sin objetos).

```
def reconstruccion(A, M, tabla):
    elemento_actual = len(A)
    cantidad_actual = M
    if not tabla[elemento_actual][cantidad_actual]:
        return "Sin solución"
    S = []
    while elemento_actual != 0 and cantidad_actual != 0:
        if tabla[elemento_actual - 1][cantidad_actual - A[elemento_actual - 1]]:
            S.append(A[elemento_actual - 1])
            cantidad_actual -= A[elemento_actual - 1]
        elemento_actual -= 1
    return S
```

4.7. Versión `memo`

```
def subset_memo(A, j, m, memo):
    if (j,m) in memo:
        return memo[(j,m)]
    if m < 0 or j < 0:
        r = False
    elif m == 0 and j>= 0:
        r = True
    elif j == 0 and m > 0:
        r = False
    else:
```

```
        r1 = subset_memo(A, j-1, m - A[j - 1], memo)
        r2 = subset_memo(A, j-1, m, memo)
        r = r1 or r2
    memo[(j,m)] = r
    return r
```

Se puede probar que el número de problemas está acotado superiormente por $O(nM)$ (esto no lo vamos a demostrar). Por otro lado, lo que se tarda en resolver cada problema es constante. Así pues, el orden es $O(nM)$.