Capítulo 4

Programación dinámica

La programación dinámica es una técnica de optimización de algoritmos que se basa en la resolución de problemas a partir de subproblemas más simples y en almacenar los resultados de estos subproblemas para evitar cálculos redundantes. Esta técnica es especialmente útil en problemas que pueden ser descritos por relaciones de recurrencia y en los cuales es común que muchos subproblemas se repitan. Por ejemplo, esta última condición no se cumple, o es muy poco frecuente, en los algoritmos de ordenación vistos en el Capítulo 3. Sin embargo, como veremos en este capítulo, es muy común en diversos problemas de optimización.

4.1. Esquema general

La idea central de la programación dinámica es la descomposición del problema en subproblemas solapados, cuya solución se almacena en una estructura de datos, típicamente una tabla, para ser reutilizada posteriormente. Este enfoque, por tanto, reduce drásticamente el tiempo de ejecución de un algoritmo al evitar volver a computar subproblemas ya resueltos.

Este esquema se fundamenta en dos conceptos clave:

- **Subproblemas solapados**: Esta característica implica que un problema puede dividirse en subproblemas que se repiten con frecuencia. Un ejemplo muy claro es el cálculo del número enésimo de Fibonacci (se verá más abajo).
- **Almacenamiento de resultados**: La eficiencia de la programación dinámica se consigue almacenando los resultados de los subproblemas, de manera que sólo se resuelven una única vez y la solución está

disponible cuando vuelve a ser necesario. A su vez, esto se puede implementar de dos formas:

- **Memoización**:[1] Esta implementación sigue un enfoque de arriba hacia abajo (top-down) para resolver mediante **programación dinámica recursiva**, en el cual se almacenan los resultados de las llamadas recursivas ya realizadas.
- **Tabulación**: Esta técnica también se utiliza para evitar cálculos repetidos, pero los subproblemas se resuelven de abajo hacia arriba (bottom-up). Se construye una tabla que almacena los resultados de subproblemas más pequeños, los cuales se utilizan para resolver problemas más grandes de manera iterativa. Este enfoque lleva a la **programación dinámica iterativa**.

Para entender mejor la programación dinámica, veamos un ejemplo sencillo.

4.1.1. Ejemplo: el número de Fibonacci

Un ejemplo clásico y sencillo de programación dinámica es el cálculo de los números de Fibonacci. La secuencia de Fibonacci se define de la siguiente manera:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

Un ejemplo de pseudocódigo para esta definición se presenta a continuación:

```
función fibonacci(n):
    si n <= 1
        devuelve n
    si no
        devuelve fibonacci(n-1) + fibonacci(n-2)
```

La implementación recursiva de esta función es ineficiente porque recalcula los mismos valores múltiples veces, lo que lleva a una complejidad

[1] Este término no es un error ortográfico, sino que viene del latín *memorandum* (ser recordado). No debe confundirse con "memorización", que en computación puede tener otros significados.

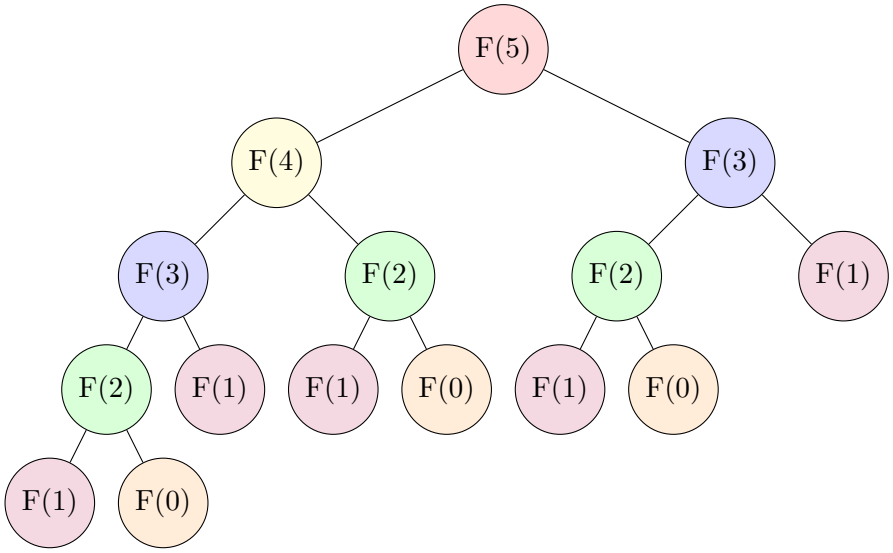

Figura 4.1: Desarrollo de llamadas recursivas para calcular F(5).

exponencial.[2] Esto se aprecia gráficamente en la Fig. 4.1. Podemos mejorar esto utilizando programación dinámica.

A continuación se presenta el pseudocódigo utilizando memoización (programación dinámica recursiva):

```
función fibonacci(n, memo):
    si n <= 1
        devuelve n
    si n en memo
        devuelve memo[n]
    si no
        memo[n] := fibonacci(n-1, memo) + fibonacci(n-2, memo)
        devuelve memo[n]
```

En esta versión, se utiliza un diccionario (o una estructura similar) llamado *memo* para almacenar los resultados de los subproblemas ya calculados, evitando así el recálculo.

Otra forma de resolver el problema es utilizando tabulación (programación dinámica iterativa). En lugar de resolver los subproblemas recursivamente, se construye una tabla de soluciones desde los casos triviales hasta los más complejos. El pseudocódigo para este enfoque se presenta a conti-

[2] Aproximadamente, del orden $\mathcal{O}(1{,}6^n)$.

nuación:

```
función fibonacci(n):
    si n <= 1
        devuelve n
    tabla[0] := 0
    tabla[1] := 1
    para i desde 2 hasta n:
        tabla[i] := tabla[i-1] + tabla[i-2]
    devuelve tabla[n]
```

En esta versión, se construye una tabla (por ejemplo, un vector) donde cada entrada i contiene el valor de $F(i)$. De esta manera, se evitan los cálculos repetidos y se mejora la eficiencia de la función.

Ambos enfoques de programación dinámica mejoran significativamente la eficiencia del cálculo de los números de Fibonacci. En el caso iterativo, es obvio que la complejidad pertenece a $\mathcal{O}(n)$. El caso recursivo es equivalente, aunque el cálculo de su complejidad sea menos intuitivo.

4.2. El problema de la mochila (discreta)

El "problema de la mochila" (en inglés, *knapsack problem*) es uno de los problemas clásicos en teoría de algoritmos y computación. En términos generales, consiste en seleccionar un subconjunto de elementos, cada uno con un peso y un valor, de manera que se maximice el valor total sin exceder un peso máximo permitido. Aunque aparentemente sencillo, muchos problemas reales pueden reducirse a una instancia del problema de la mochila, como la planificación de proyectos o la distribución de recursos.

El problema de la mochila puede formularse matemáticamente de la siguiente manera: dado un conjunto de n elementos, donde cada elemento i tiene un valor v_i y un peso w_i, y una capacidad máxima de la mochila W, se busca maximizar la suma de los valores de los elementos seleccionados, de modo que la suma de los pesos no exceda W.

$$\text{Maximizar } \sum_{i=1}^{n} v_i x_i$$

sujeto a:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

donde x_i es una variable binaria que indica si el elemento i es seleccionado ($x_i = 1$) o no ($x_i = 0$). Para resolver el problema de la mochila, podemos definir la relación de recurrencia que describe el valor máximo que se puede obtener:

$$K(i, w) = \begin{cases} 0 & \text{si } i = 0 \text{ o } w = 0 \\ K(i-1, w) & \text{si } w_i > w \\ \max(K(i-1, w), v_i + K(i-1, w-w_i)) & \text{si } w_i \leq w \end{cases}$$

donde $K(i, w)$ representa el valor máximo que se puede obtener utilizando los primeros i elementos con una capacidad de mochila de w. Esta relación de recurrencia tiene dos casos:

1. Caso base: Si no hay elementos ($i = 0$) o la capacidad de la mochila es cero ($w = 0$), entonces el valor máximo que se puede obtener es 0.

2. Exclusión del elemento: Si el peso del i-ésimo elemento es mayor que la capacidad actual de la mochila ($w_i > w$), entonces este elemento no puede incluirse en la solución óptima. En este caso, el valor óptimo es el mismo que el valor óptimo obtenido sin este elemento, es decir, $K(i-1, w)$.

3. Inclusión del elemento: Si el peso del i-ésimo elemento es menor o igual a la capacidad actual de la mochila ($w_i \leq w$), entonces se considera dos posibilidades: no incluir el i-ésimo elemento (valor $K(i-1, w)$) o incluirlo (valor $v_i + K(i-1, w-w_i)$). El valor máximo se obtiene eligiendo la opción que maximiza el valor total.

En este capítulo vamos a estudiar **la versión discreta** del problema de la mochila, en la cuál se añade la restricción de que los pesos de los objetos son discretos (valores enteros). Esto permite indexar en la tabla de programación dinámica los distintos estados en los que puede encontrarse la mochila a medida que se van seleccionando o no objetos.

Para resolver este problema utilizando programación dinámica, podemos definir una tabla `PD` donde `PD[i][w]` representa el valor máximo que se puede obtener utilizando los primeros `i` elementos con una capacidad de mochila de `w`. A continuación, se presenta el pseudocódigo utilizando memoización:

```
función mochila(n, W, pesos, valores, PD):
    si n = 0 o W = 0
        devuelve 0
```

```
    si (n, W) en PD
        devuelve PD[(n, W)]
    si pesos[n-1] > W
        PD[(n, W)] := mochila(n-1, W, pesos, valores, PD)
    si no
        PD[(n, W)] := max(
            mochila(n-1, W, pesos, valores, PD),
            valores[n-1] + mochila(n-1, W-pesos[n-1], pesos,
   valores, PD)
        )
    devuelve PD[(n, W)]
```

La versión iterativa utilizando tabulación rellena la tabla `PD` de manera iterativa, comenzando desde el primer elemento hasta el último y para todos los posibles estados de la mochila. Esta versión se presenta a continuación:

```
función mochila(n, W, pesos, valores):
    PD := matriz de ceros de tamaño (n+1, W+1)
    para i desde 1 hasta n:
        para w desde 1 hasta W:
            si pesos[i-1] > w:
                PD[i][w] := PD[i-1][w]
            si no:
                PD[i][w] := max(
                    PD[i-1][w],
                    valores[i-1] + PD[i-1][w-pesos[i-1]]
                )
    devuelve PD[n][W]
```

En esta versión, se construye una tabla `PD` donde cada entrada `PD[i][w]` contiene el valor máximo que se puede obtener utilizando los primeros `i` elementos con una capacidad de mochila de `w`.

Ambos enfoques, memoización y tabulación, tienen sus ventajas y desventajas. La memoización es más intuitiva y fácil de implementar, ya que sigue la estructura recursiva natural del problema. Sin embargo, puede tener una sobrecarga de memoria debido al almacenamiento de la pila de llamadas recursivas. La tabulación, por otro lado, es más eficiente en términos de espacio y evita la sobrecarga de llamadas recursivas, pero puede ser menos intuitiva ya que requiere la construcción explícita de la tabla de soluciones. Además, la tabulación siempre resuelve todos los subproblemas posibles, incluso los no necesarios, mientras que la memoización solo resuelve los subproblemas que realmente se necesitan.

Al igual que en el ejemplo de Fibonacci, la complejidad es fácil de inferir a partir de la versión iterativa. En este caso, el algoritmo pertenece a $\mathcal{O}(n \cdot W)$.[3]

4.3. La distancia de edición

La *distancia de edición*, también conocida como distancia de Levenshtein, es una métrica fundamental en el campo de la inteligencia artificial y el procesamiento del lenguaje natural. Esta métrica se utiliza para evaluar cuán similares son dos cadenas de texto, midiendo el número mínimo de operaciones necesarias para transformar una cadena en otra. Las operaciones permitidas son la inserción, eliminación y sustitución de caracteres. La importancia de la distancia de edición radica en su amplia aplicación en problemas como la corrección ortográfica, el reconocimiento de voz, la comparación de secuencias genéticas, la búsqueda aproximada de cadenas y la evaluación de sistemas de traducción automática, entre otros.

El problema de la distancia de edición se plantea de la siguiente manera: dadas dos cadenas s_1 y s_2, queremos encontrar el número mínimo de operaciones (inserciones, eliminaciones o sustituciones) necesarias para transformar s_1 en s_2.

Consideremos un ejemplo sencillo para ilustrar la distancia de edición entre dos cadenas. Supongamos que queremos transformar la cadena "casa" en "costa". Para ello:

- Sustituimos 'o' por la primera 'a': "casa" $\rightarrow$ "cosa".
- Insertamos 't' después de 's': "cosa" $\rightarrow$ "costa".

La distancia de edición en este caso es 2, ya que necesitamos una operación de inserción y una de sustitución.

Para resolver la distancia de edición computacionalmente lo podemos plantear como un problema recursivo en el cual se va realizando el cálculo para los prefijos de cada cadena. Para llegar a la relación de recurrencia $d(i, j)$, donde se comparan los prefijos $s_1[1..i]$ y $s_2[1..j]$:

1. Si el último carácter de ambas cadenas es el mismo ($s_1[i] = s_2[j]$), la distancia de edición es la misma que para las subcadenas anteriores, es decir, $d(i-1, j-1)$.

[3] Nótese que en este caso, hay dos variables que definen la talla del problema: el número de elementos n y el peso máximo de la mochila W.

2. Si el último carácter de ambas cadenas es diferente ($s_1[i] \neq s_2[j]$), debemos considerar la operación de menor coste entre:
   - Insertar el carácter $s_2[j]$ en s_1, lo que implica $d(i, j-1)+1$.
   - Eliminar el carácter $s_1[i]$, lo que implica $d(i-1, j)+1$.
   - Sustituir el carácter $s_1[i]$ por $s_2[j]$, lo que implica $d(i-1, j-1)+1$.

Por tanto, la relación de recurrencia para la distancia de edición es la siguiente:

$$d(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ d(i-1, j-1) & \text{si } s_1[i-1] = s_2[j-1] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si } s_1[i-1] \neq s_2[j-1] \end{cases}$$

A continuación se presenta el pseudocódigo para calcular la distancia de edición utilizando una versión iterativa con tabulación, que es la implementación más habitual:

```
función distancia_edición(s1, s2):
    n := |s1|
    m := |s2|
    DP := matriz de tamaño (n+1, m+1)

    para i desde 0 hasta n
        DP[i][0] := i

    para j desde 0 hasta m
        DP[0][j] := j

    para i desde 1 hasta n
        para j desde 1 hasta m
            si s1[i-1] = s2[j-1]
                DP[i][j] := DP[i-1][j-1]
            si no
                DP[i][j] := 1 + min(DP[i-1][j],
                                    DP[i][j-1],
                                    DP[i-1][j-1])

    devuelve DP[n][m]
```

Consideremos de nuevo el ejemplo de transformar “casa” en “costa”. Veamos cómo se llena la tabla `DP` paso a paso:

		c	o	s	t	a
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	3	3
s	3	2	2	1	2	3
a	4	3	3	2	2	2

Para entender la tabla y el cálculo de la distancia de edición, hay que tener en cuenta que:

- La primera fila y la primera columna representan las distancias cuando una de las cadenas es vacía.
- Cada celda `DP[i][j]` se llena utilizando la relación de recurrencia descrita anteriormente. Por ejemplo, `DP[1][1]` es 0 porque los caracteres ’c’ coinciden.
- Finalmente, `DP[4][5]` indica que la distancia de edición entre “casa” y “costa” es 2.

Con respecto a su complejidad, la distancia de edición mediante programación dinámica tiene un coste perteneciente a $\mathcal{O}(|s_1| \cdot |s_2|)$.

4.4. Recuperación de soluciones

En programación dinámica, cuando resolvemos un problema utilizando las técnicas que hemos visto (memoización y tabulación), normalmente nos centramos en calcular y almacenar los valores óptimos de los subproblemas, y no las soluciones específicas que los producen. Por ejemplo, en el problema de la mochila hemos calculado qué valor máximo se puede obtener pero el algoritmo no devuelve explícitamente los objetos que se seleccionarían.

Sin embargo, una vez que tenemos los valores óptimos, a menudo necesitamos recuperar la solución específica. Para hacer esto, podemos explorar la tabla de programación dinámica desde el final (donde se encuentra la solución al problema completo) hacia el principio (donde se encuentran los subproblemas más pequeños).

Este proceso se conoce como **recuperación de soluciones** y típicamente sigue estos pasos:

1. Identificar el valor óptimo: Comenzamos en el elemento de la tabla que contiene el valor óptimo de la solución completa.

2. Rastrear las decisiones: Desde este punto, rastreamos hacia atrás las decisiones que llevaron a este valor óptimo. Esto implica seguir el camino inverso de las decisiones tomadas durante la construcción de la tabla. En cada paso, determinamos si el valor actual fue obtenido incluyendo un cierto elemento o excluyéndolo (en el contexto de problemas de optimización como el de la mochila).

3. Registrar las decisiones: Cada vez que determinamos una decisión, la registramos como parte de la solución específica. Este proceso se repite hasta que llegamos a la base del problema (por ejemplo, cuando el tamaño del subproblema es cero).

Supongamos que estamos resolviendo el problema de la mochila y tenemos una tabla de programación dinámica `DP` donde `DP[i][w]` representa el valor óptimo obtenido considerando los primeros `i` elementos y una restricción de peso restante `w`. La solución final se encuentra en `DP[n][W]`, donde `n` es el número total de elementos y `W` es el peso máximo permitido.

Para recuperar la solución, comenzamos en `DP[n][W]`:

1. Si `DP[i][w]` es igual a `DP[i-1][w]`, entonces el n-ésimo elemento no se incluyó en la solución óptima.

2. Si `DP[i][w]` es diferente de `DP[i-1][w]`, entonces el n-ésimo elemento sí se incluyó en la solución óptima. En este caso, continuamos el rastreo con los $n - 1$ elementos restantes y la nueva restricción $w = w - w_n$.

3. Continuamos este proceso hasta llegar a `DP[0][0]`.

Este método de rastreo inverso asegura que identificamos exactamente qué elementos o decisiones forman parte de la solución óptima, proporcionando no solo el valor del resultado óptimo sino también la composición específica de la solución.

Para finalizar, conviene recordar que, según el problema, es posible que haya varias soluciones igualmente óptimas. El procedimiento anterior seguiría siendo válido siempre y cuándo tan solo necesitáramos recuperar una de estas posibles soluciones. En caso de que se quieran recuperar todas, habría que hacer ciertas modificaciones durante el rastreo.

4.5. Consideraciones adicionales

En este capítulo hemos visto la programación dinámica a través de dos enfoques: memoización (programación dinámica recursiva) y tabulación (programación dinámica iterativa). En teoría, ambos enfoques son equivalentes, pueden resolver los mismos problemas y tienen complejidades asintóticas iguales. Sin embargo, en la práctica sí presentan diferencias importantes. La memoización, siendo recursiva, puede resultar en un código más intuitivo y natural para problemas definidos recursivamente, pero conlleva una sobrecarga de llamadas recursivas y puede ser menos eficiente en términos de tiempo. En cambio, la tabulación evita la recursión, lo que puede hacerla más eficiente en tiempo al utilizar bucles iterativos, aunque puede requerir más memoria al almacenar todos los subproblemas intermedios (incluso los que no fueran necesarios). La elección entre ambos enfoques depende de la naturaleza del problema y de las restricciones de tiempo y espacio disponibles.