Programacion Dinamica

Empezamos a ver la programacion dinamica, que es una tecnica de **optimizacion** que se utiliza para resolver problemas complejos dividiendolos en subproblemas mas simples. La idea es almacenar los resultados de los subproblemas ya resueltos para evitar recalcularlos, lo que reduce el tiempo de computo.

La programacion dinamica es especialmente util en problemas de optimizacion, donde se busca maximizar o minimizar una funcion objetivo. Se utiliza en una variedad de campos, incluyendo la teoria de grafos, la teoria de juegos, la bioinformatica y la economia.

1. Intucion con el problema de la escalera

> Imagina una escalera de n escalones. Puedes subir la escalera de dos maneras: un paso a la vez o dos pasos a la vez. ¿Cuántas formas diferentes hay de subir la escalera? (llegar al escalon n desde el escalon 0)

1.1 Formulacion

1.1.1 Formulacion Intuitiva Definimos:

$W(n)$ = el número de formas distintas de subir desde el escalón 0 hasta el escalón n

donde cada movimiento es un paso de 1 escalón o un paso de 2 escalones.

- "**Formas distintas**" significa secuencias diferentes de movimientos. Por ejemplo,
  - Tomar $(1, 1, 1)$ para llegar al escalón 3 es diferente de tomar $(2, 1)$, aunque ambos te lleven al escalón 3.
- Comenzamos desde el **escalón 0** (nivel del suelo). La pregunta es: ¿cuántas secuencias diferentes de 1s y 2s suman n?

1.1.3 Formulacion con Ejemplos A veces, la mejor manera de comprender $W(n)$ es verlo para valores pequeños de n.

Caso $n = 0$

- Técnicamente, si ya estás en el escalón 0, no te mueves.
- Por convención, decimos que $W(0) = 1$. Esa "1 forma" es simplemente la **secuencia vacía de movimientos** (no hacer nada).
- Matemáticamente, $\mathcal{S}_0 = \{()\}$, es decir, el conjunto que contiene la tupla vacía.

Caso $n = 1$

- Tienes 1 escalón que subir; el único movimiento válido es un paso de 1 escalón.
- Secuencias que te llevan del 0 al 1: (1).
- Por lo tanto, $W(1) = 1$.

Caso $n = 2$

- Tienes 2 escalones que subir. Posibles secuencias:
  1. $(1, 1)$ – dos movimientos de 1 escalón.
  2. (2) – un salto de 2 escalones.
- Hay **2 formas distintas**, por lo que $W(2) = 2$.

Caso $n = 3$

- Secuencias para subir 3 escalones:
  1. $(1, 1, 1)$
  2. $(1, 2)$
  3. $(2, 1)$
- Por lo tanto, $W(3) = 3$.

Caso $n = 4$ Listemos todas las secuencias explícitamente:

1. $(1, 1, 1, 1)$
2. $(1, 1, 2)$
3. $(1, 2, 1)$
4. $(2, 1, 1)$
5. $(2, 2)$

Así que hay 5 formas. Por lo tanto $W(4) = 5$.

Resumiendo:

$W(0) = 1$, $W(1) = 1$, $W(2) = 2$, $W(3) = 3$, $W(4) = 5$.

Esta secuencia $(1, 1, 2, 3, 5, \dots)$ recuerda a la **secuencia de Fibonacci**, solo que ligeramente desplazada.

1.2 Estrucutura

¿Por Qué Aparece Este Patrón (Similar a Fibonacci)? Concéntrate en llegar al escalón n. **¿Cómo podrías haber llegado allí?**

- O bien tomaste un movimiento de **1 paso** desde el escalón $n - 1$,
- O tomaste un movimiento de **2 pasos** desde el escalón $n - 2$.

Cualquier camino completo hacia el escalón n debe, por tanto, terminar en: - "…$n - 1$, luego un solo paso", o - "…$n - 2$, luego un paso doble".

Por lo tanto, cada forma distinta de llegar al escalón n se forma tomando una forma distinta de llegar a $n - 1$ y añadiendo (1), o una forma distinta de llegar a $n - 2$ y añadiendo (2). Eso implica:

$W(n) = W(n - 1) + W(n - 2)$,

con las **condiciones iniciales** $W(0) = 1$ y $W(1) = 1$.

Preguntas: 1. Como seria $W(n)$ si no pudieras dar pasos de 2 escalones? 2. Como seria $W(n)$ si pudieras dar pasos de 1, 2 y 3 escalones ? 3. Como seria $W(n)$ si pudieras dar pasos de 2 y 3 escalones ? —

1.2.1 Ejemplo de la recursividad

Comprobemos la consistencia:

$W(2) = W(1) + W(0) = 1 + 1 = 2$, $W(3) = W(2) + W(1) = 2 + 1 = 3$, $W(4) = W(3) + W(2) = 3 + 2 = 5$,

y así sucesivamente. Esto coincide exactamente con lo que encontramos al enumerar secuencias. Así que la recurrencia

$W(n) = W(n - 1) + W(n - 2)$

no es solo una conjetura—se deriva directamente de la estructura de cómo se puede dar el último paso (1 o 2) y de cómo ese último paso extiende soluciones más pequeñas.

Preguntas: 1. Para responder el valor de $W(n)$ que informacion es sufiente? 2. Que pasaria si quisieras calcular $W(n)$ para $n = 10^9$? —

2. Soluciones

2.1 Solucion Naive (fuerza bruta)

Una forma directa (sin PD) es escribir una función recursiva ingenua:

- `W(n) = W(n-1) + W(n-2)`,
- Casos base: `W(0)=1` y `W(1)=1`.

Sin embargo, si llamas a `W(n-1)`, volverás a llamar a `W(n-2)` y `W(n-3)`, etc. Mientras tanto, *también* llamas independientemente a `W(n-2)`, que a su vez llama a `W(n-3)`, `W(n-4)`, y así sucesivamente. Esta estructura lleva a una repetición **exponencial** de llamadas, porque los mismos subproblemas (por ejemplo, `W(n-3)`) se resuelven una y otra vez.

```python
def W(n):
    # Casos base
    if n == 0 or n == 1:
        return 1

    # Caso recursivo
    return W(n-1) + W(n-2)
```

Esta implementación es correcta matemáticamente pero extremadamente ineficiente para valores grandes de n. Veamos por qué con un ejemplo detallado.

2.1.1 Ejemplo naive Si queremos calcular $W(5)$, nuestra función realizará la siguiente secuencia de llamadas:

```
W(5)
  W(4)
     W(3)
        W(2)
           W(1) = 1
           W(0) = 1
        W(1) = 1
     W(2)
        W(1) = 1
        W(0) = 1
  W(3)
     W(2)
        W(1) = 1
        W(0) = 1
     W(1) = 1
```

Observa cómo $W(3)$ se calcula dos veces, $W(2)$ se calcula tres veces, $W(1)$ se calcula cinco veces, y $W(0)$ se calcula tres veces. Ya con un valor tan pequeño como $n = 5$, podemos ver la redundancia en los cálculos.

Conteo de Llamadas Para ser más específicos, contemos exactamente cuántas veces se llama a cada función:

Función	Número de llamadas
$W(5)$	1
$W(4)$	1
$W(3)$	2
$W(2)$	3
$W(1)$	5
$W(0)$	3
Total	**15**

Para calcular $W(5)$, la función se llama un total de 15 veces. Si el número fuera $W(n)$, el número de llamadas sería aproximadamente $O(2^n)$, lo que significa que:

- Para $n = 10$: aproximadamente 177 llamadas
- Para $n = 20$: aproximadamente 21,891 llamadas
- Para $n = 30$: aproximadamente 2,692,537 llamadas
- Para $n = 40$: aproximadamente 331,160,281 llamadas

Nota: No me dio tiempo de verificar y estudiar concretamente la formula de dodne se deriva este costo exponencial, pero es un resultado de claude y chatgpt.

Este crecimiento exponencial hace que la función sea prácticamente inutilizable para valores moderadamente grandes de n. Por ejemplo, si cada llamada a la función toma solo 1 microsegundo (0.000001 segundos), calcular $W(40)$ tomaría aproximadamente 331 segundos (más de 5 minutos), y $W(50)$ tomaría más de 13 días.

Cocinando Cebollas (no escarchas) Imaginemos que estamos preparando una cena con 5 platos, y cada plato necesita cebollas picadas:

1. Para el primer plato (análogo a $W(5)$), decides picar todas las cebollas desde cero.
2. Para hacerlo, primero preparas los ingredientes para el plato 4, lo que también requiere picar cebollas.
3. Mientras preparas el plato 4, también necesitas ingredientes para el plato 3, que también requiere cebollas.
4. Y así sucesivamente hasta llegar a los platos básicos (0 y 1).

Pero lo peor es que **olvidas** que ya has picado cebollas para el plato 3 cuando preparabas el plato 4, y vuelves a picarlas desde cero cuando las necesitas directamente para el plato 5. Y esto ocurre con cada subpreparación: repites el trabajo una y otra vez.

Un chef eficiente picaría todas las cebollas una sola vez y las utilizaría según sea necesario para cada plato. Esa es exactamente la idea detrás de la programación dinámica: resolver cada subproblema una sola vez y almacenar su resultado para usarlo cuando sea necesario.

2.2 Solucion Optima (programacion dinamica)

La Idea Central La programación dinámica (PD) es una técnica de diseño de algoritmos que se basa en dos propiedades fundamentales:

1. **Identificar Subproblemas Superpuestos**
   Vemos que subproblemas como $W(k)$ se repiten a través de las llamadas.
2. **Almacenar Soluciones (Memoizar)**
   Calcula $W(k)$ una vez, y guárdalo en un "memo" (un diccionario, array o lista). La próxima vez que necesites $W(k)$, lo recuperas instantáneamente.
3. **Reutilizar y Construir**
   Al completar estos resultados almacenados para valores más pequeños de k, puedes construir rápidamente la respuesta para valores más grandes de n.

Este enfoque se llama **Programación Dinámica** porque resuelves problemas grandes combinando soluciones a problemas más pequeños, *y* evitas resolver el mismo subproblema múltiples veces.

2.2.1 De Arriba Hacia Abajo (Recursión Memoizada/ Top Down)

- Comienza con tu función `formas(n)`:
  - Si $n = 0$ o $n = 1$, devuelve 1.
  - De lo contrario, devuelve `formas(n-1) + formas(n-2)`.
- **Pero** mantén un diccionario/array `memo`:
  - Antes de calcular `formas(n)`, verifica si `n` ya está en `memo`.
    - Si es así, simplemente devuelve ese valor almacenado.
    - Si no, calcúlalo, guárdalo en `memo`, y luego devuélvelo.

Este método mantiene la misma lógica recursiva pero previene el trabajo repetido al recordar resultados intermedios.

```
def W(n, memo={}):
    # Casos base
    if n == 0 or n == 1:
        return 1

    # Verificar si ya hemos calculado W(n)
    if n in memo:
        return memo[n]

    # Caso recursivo
    memo[n] = W(n-1, memo) + W(n-2, memo)
    return memo[n]
```

2.2.1.1 Ejemplo de la recursividad memoizada Imagina llamar a `W(5)`: 1. Verificas si `W(5)` está en `memo`. No lo está, así que calculas: W(5) = W(4) + W(3). 2. Al calcular `W(4)` verificas si está en `memo`; si no, haces `W(3)+W(2)`, etc. 3. Eventualmente completarás `W(0) = 1`, `W(1) = 1`, `W(2) = 2`, etc. 4. Una vez que se encuentra `W(k)` para valores más pequeños de k, nunca los recalculas.

Todo desde 0 hasta 5 se resuelve una vez. Luego `W(5)` devuelve rápidamente.

2.2.2 De Abajo Hacia Arriba (Tabulacion/ Bottom Up) En lugar de una visión recursiva, puedes tomar un enfoque *iterativo*:

1. Crea un array `dp` de tamaño $n + 1$.
2. Inicializa:
   - `dp[0] = 1` (el caso de secuencia vacía),
   - `dp[1] = 1`.
3. Luego para cada `i` desde 2 hasta n: dp[i] = dp[$i - 1$] + dp[$i - 2$].
4. Al final, `dp[n]` es nuestro $W(n)$ deseado.

En esencia, llenas una tabla fila por fila hasta llegar a `dp[n]`. Por eso a veces se le llama **tabulación**.

```python
def W(n):
    # Crear un array para almacenar los resultados
    dp = [0] * (n + 1)

    # Casos base
    dp[0] = 1
    dp[1] = 1

    # Llenar el array dp
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

2.2.2.1 Ejemplo de la tabulacion Imagina que estás llenando una tabla de resultados:

i	$W(i)$
0	1
1	1
2	2
3	3
4	5
5	8

2.2.3 Comparacion entre los dos metodos

Método	Espacio	Tiempo	Descripción
Recursión Memoizada	$O(n)$	$O(n)$	Almacena resultados intermedios en un diccionario.
Tabulación	$O(n)$	$O(n)$	Llena una tabla iterativamente desde el caso base hasta el caso deseado.
Recursión Simple	$O(n)$	$O(2^n)$	Llama a la función recursivamente sin almacenar resultados intermedios.

- Ambos métodos tienen un tiempo de ejecución de $O(n)$, pero la recursión simple es exponencial.
- Ambos métodos requieren $O(n)$ espacio, pero la recursión simple no almacena nada.
- La recursión memoizada es más intuitiva, pero la tabulación es más eficiente en términos de espacio y a veces más fácil de entender.

3 Programacion Dinamica (Planteamiento)

Tiene dos propiedades clave que la hacen efectiva: 1. **Subproblemas Superpuestos**
2. **Subestructura Óptima**

3.1 Subproblemas Superpuestos

3.1.1 Intuicion

- Puedes pensar en "subproblemas superpuestos" como cortar repetidamente las mismas verduras para diferentes platos.
- Si un enfoque recursivo ingenuo sigue resolviendo los mismos problemas pequeños una y otra vez, tienes un fuerte indicio de que almacenar en caché (memorizar) esas subsoluciones podría ahorrar mucho tiempo.

Por ejemplo, en el **problema de la escalera** (cuántas formas hay de subir n escalones tomando 1 o 2 a la vez): - La recursión ingenua para $W(n)$ llama a $W(n-1)$ y $W(n-2)$.
- Luego $W(n-1)$ llama a $W(n-2)$ y $W(n-3)$.
- Observa que $W(n-2)$ se está calculando dos veces—esta repetición ocurre *exponencialmente*.

Ese patrón repetido o "superposición" es una señal clara: *Almacena* $W(n-2)$ *para calcularlo solo una vez.*

3.1.2 Formalmente Un problema tiene **subproblemas superpuestos** si, cuando formulas una solución recursiva, el mismo subproblema se resuelve múltiples veces. Más formalmente, en un árbol de recursión, si algún nodo corresponde a un subproblema Sub(k) que aparece en más de una rama, indica subproblemas superpuestos.

3.2 Subestructura Óptima

3.2.1 Intuicion

- "Subestructura óptima" significa que la mejor solución (o solución total) a tu problema *se puede obtener combinando* las soluciones a subproblemas.
- Otro enfoque: el camino hacia una solución óptima no contradice las soluciones óptimas a los subproblemas en el camino.

En un escenario de **camino más corto**, si la ruta de A a D pasa por B y C, entonces la ruta de B a D es en sí misma una ruta óptima (si ya estás en B). De lo contrario, podrías mejorar tu ruta total de A a D mejorando la ruta de B a D.

En el problema de **conteo de la escalera**, no se trata de "óptimo" en el sentido de costo mínimo o máximo, pero el mismo principio se mantiene: el número de formas de llegar a n se puede obtener del número de formas de llegar a $n-1$ más el número de formas de llegar a $n-2$. Construyes tu respuesta final combinando respuestas más pequeñas.

3.2.2 Formalmente Un problema tiene **subestructura óptima** si una solución globalmente óptima (o total) puede componerse a partir de soluciones óptimas (o correctamente combinadas) a sus subproblemas. Formalmente, si OPT(P) es la solución óptima al problema P, y $P_1, P_2, \ldots, P_k$ son subproblemas, entonces:

$\text{OPT}(P) = \text{combinar}_(\text{OPT}(P_1), \text{OPT}(P_2), \ldots)$.

Para un problema de *conteo* (como el de la escalera), puedes ver "combinar" como sumar los conteos de subproblemas relevantes. Para un problema de *minimización* o *maximización*, "combinar" podría significar tomar el mínimo/máximo de subproblemas más un cierto costo local o recompensa.

4 Programacion Dinamica (Verificacion)

4.1 Verificacion Intuitiva

1. **Ves un Desglose Recursivo**:
   - Pregúntate: "¿Puede este problema expresarse en términos de versiones más simples de sí mismo?"
   - A menudo, intentas una recursión por fuerza bruta. Si notas que estás repitiendo el mismo subcálculo múltiples veces, esa es una señal reveladora.
2. **Patrones o Estados Repetidos**:

- ¿Sigues encontrando el mismo "estado" (por ejemplo, la misma solución parcial o la misma posición) desde diferentes ramas del árbol de solución?

- Esto puede aparecer en búsqueda de caminos, programación de tareas, problemas tipo mochila o conteo de caminos.

3. **Las Elecciones Locales No Rompen Soluciones Futuras**:
   - En problemas que requieren una solución "óptima" o de "conteo", puedes combinar resultados de subproblemas más pequeños sin tener que reevaluarlos o corregirlos más tarde.

   - Si ves un escenario donde una solución local o parcial puede necesitar ser deshecha o cambiada drásticamente, el problema podría no tener una estructura de PD limpia (o podría necesitar técnicas más avanzadas).

4.2 Verificacion Semi-Formal

4.2.1 Definir Recurrencia Un problema es típicamente adecuado para PD si puedes definir una **recurrencia**:

$\text{DP}[s] \;=$ alguna función de $\{\text{DP}[s_1], \text{DP}[s_2], \dots\}$,

donde: - s es un *estado*, que representa un subproblema, - $s_1, s_2, \dots$ son sub-estados más pequeños o más simples, - la combinación "alguna función de" podría ser `+`, `min`, `max`, `count`, u otro operador matematico.

4.2.2 Superposicion

- Una vez que has definido tus estados, compruebas si los estados se repiten (se superponen) en una expansión recursiva ingenua.

4.2.3 Subestructura

- Confirmas que resolver estos estados de forma aislada puede combinarse sin conflicto para producir la solución general.

- Si la solución a un subproblema podría necesitar ser modificada después de ver otros subproblemas, es posible que no tengas una verdadera subestructura óptima.

4.2.4 Casos Base

- Finalmente, asegúrate de que tienes casos base claros y bien definidos.
- Estos son los estados más simples que no requieren más descomposición y pueden ser resueltos directamente.
- Sin casos base, tu recursión podría entrar en un bucle infinito o no converger a una solución. #### 4.2.5 Ejemplo

1. **El Problema de la Escalera** (Contar formas de subir):
   - **Superposición**: `formas(k)` reaparece múltiples veces en una recursión ingenua.

   - **Subestructura Óptima**: El conteo para `formas(n)` = `formas(n-1)` + `formas(n-2)`.
2. **Problema de la Mochila** (Maximizar el valor dentro de un límite de peso):
   - **Superposición**: La recursión ingenua prueba subconjuntos que a menudo repiten elecciones parciales.

   - **Subestructura Óptima**: Si la mejor manera de llenar la capacidad W incluye un artículo, entonces el subproblema de llenar la capacidad restante ($W -$ peso) también debe resolverse de manera óptima.
3. **Camino más Corto en un Grafo Acíclico Dirigido**:
   - **Superposición**: Los caminos a ciertos vértices pueden ser examinados múltiples veces en un enfoque ingenuo.

   - **Subestructura Óptima**: El camino más corto a un nodo depende de los caminos más cortos a sus predecesores más un costo de borde.

4.3 Verificacion Practica

1. **Prueba una Pequeña Recursión/Fuerza Bruta Ingenua**:
   Escribe o pseudo-codifica una recursión por fuerza bruta. Si ves llamadas repetidas para el mismo subproblema, esa es una fuerte señal de subproblemas superpuestos.

2. **Formula una Recurrencia**:
   Comprueba si puedes escribir algo como: $\mathrm{DP}(x) = \min / \max /$suma de $\{\mathrm{DP}(x_1), \mathrm{DP}(x_2), \dots\} +$(quizás costo local). Si eso es claramente correcto y consistente, probablemente tengas subestructura óptima.

3. **Pregunta**: **"¿Cada Subproblema se Conecta Limpiamente?"**
   Si las soluciones parciales no entran en conflicto o requieren reelaboración cuando se combinan, estás bien.

4. **Mira el Espacio de Búsqueda**:
   Si la solución ingenua es exponencial pero el número de sub-estados distintos es mucho menor, eso sugiere fuertemente que la PD puede ayudar.