Programación Dinámica

Constanza Vázquez

Requisitos

Se asume familiaridad básica con:

- **C++ básico**: variables, `if/else`, `for`, funciones y `vectores`.
- **Complejidad** $O(\cdot)$ a nivel intuitivo (de la clase anterior).
- **Recursión**: escribir y leer una función que se llama a sí misma. (*)

Idea

1. Idea
2. Técnicas
3. Ejercicios

Problema del profesor borracho

Hay n ($1 \leq n \leq 10^5$) cervezas en fila; la i-ésima da a_i ($1 \leq a_i \leq 10^9$) puntos de ebriedad.

El profesor quiere el **máximo puntaje**, pero está tan borracho que **al tomar una cerveza bota las dos adyacentes**.

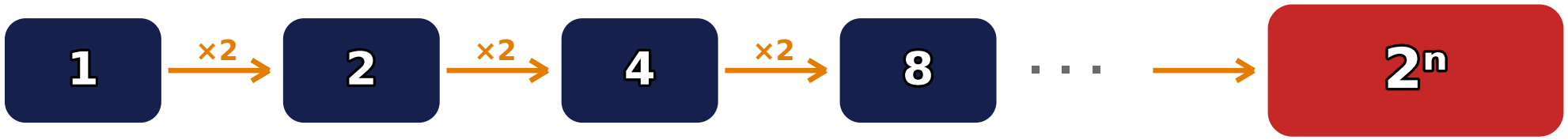
n	**cervezas**	**máx. puntaje**
4	`1 2 1 3`	5
3	`3 5 3`	6
6	`3 1 1 2 1 2`	7

Ejemplo 1: tomo la 2ª y la 4ª ($2 + 3 = 5$). Ejemplo 2: tomo la 1ª y la 3ª ($3 + 3 = 6$).

¿Cuántas opciones hay?

Para cada cerveza decido: **la tomo** o **no la tomo**.

Con n cervezas eso son hasta 2^n combinaciones posibles.



Para $n = 60$, 2^{60} es un Time Limit Exceeded asegurado. **Probar todas es imposible.**

Necesitamos una idea más inteligente que probar todo.

La idea recursiva

Miremos solo la **última** cerveza (la i). Solo hay dos escenarios:

- **No la tomo** → me quedo con lo mejor de las cervezas $0 \ldots i-1$.
- **La tomo** → gano a_i, pero pierdo la $i-1$, así que sumo lo mejor de $0 \ldots i-2$.

Me quedo con el mejor de los dos:

$$\text{resolver}(i) = \max\left(\ \text{resolver}(i-1),\ \ \text{resolver}(i-2) + a_i\ \right)$$

Casos base: $\text{resolver}(0) = a_0$, y si no quedan cervezas el puntaje es 0.

Este tipo de igualdad se llama **recurrencia**.

¿Qué es la programación dinámica?

> **Programación dinámica (DP)**: resolver un problema **combinando las soluciones de subproblemas más pequeños**, guardando cada solución para **reusarla** en lugar de recalcularla.

Funciona cuando el problema cumple **dos propiedades**:

Subestructura óptima

la mejor solución se arma con las mejores subsoluciones (nuestro `max`)

Subproblemas superpuestos

los mismos subproblemas se repiten… y frenan la recursión

El borracho cumple ambas → es un problema de DP.

La plantilla de un DP

Casi todos los DP recursivos tienen **la misma forma**:

```
tipo dp(estado) {
    if (/* caso base */)
        return /* valor base */;
    tipo res = /* combinar dp(estados más pequeños) */;
    return res;
}
```

- **Estado**: qué describe cada subproblema (en el borracho, el índice `i`).
- **Casos base**: los subproblemas que se responden sin recursión.
- **Transición**: cómo un estado se arma a partir de estados más pequeños.

DP propuesto para el profesor borracho

```cpp
int n;
int a[10];

int resolver(int i) {
    if (i < 0) return 0;                // no quedan cervezas
    if (i == 0) return a[0];            // solo queda la primera
    return max(resolver(i - 1),            // no tomo la i
               resolver(i - 2) + a[i]);  // sí tomo la i
}
// llamar desde main resolver(n - 1)
```

…pero se cuelga

Prueba con $n = 60$: el programa **nunca termina**.

Cada llamada genera **dos** llamadas más, que generan dos cada una… el número de llamadas crece como 2^n.

10^{15} operaciones ≈ semanas de cómputo · el programa nunca termina

El problema es sencillo, pero nuestra solución es **exponencial**. ¿Por qué tanto trabajo?

Para verlo con claridad, miremos el caso más simple posible que tiene la misma forma.

La sucesión de Fibonacci

Una sucesión muy conocida: **cada número es la suma de los dos anteriores**, empezando por 0 y 1.

0 y 1 son el punto de partida; de ahí, cada término suma los dos previos

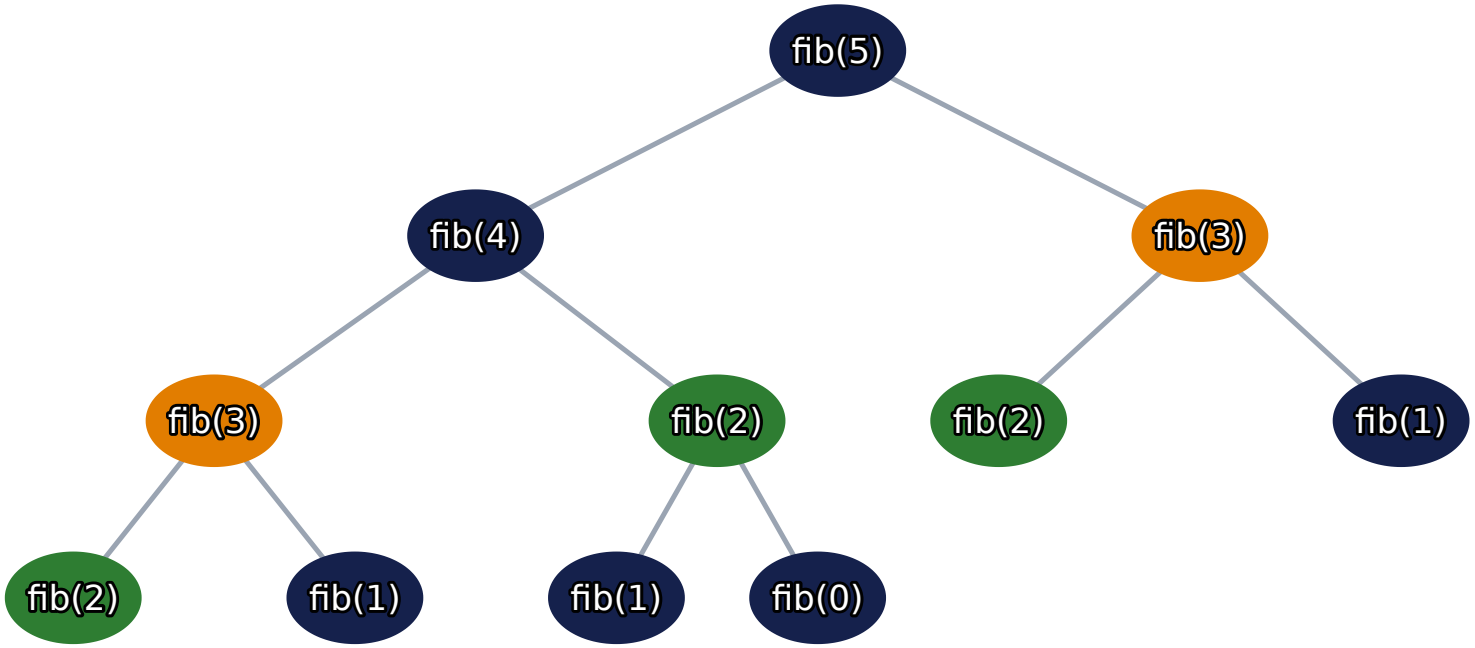
Fibonacci tiene exactamente la misma estructura "cada llamada hace dos":

$$\text{fib}(k) = \text{fib}(k-1) + \text{fib}(k-2)$$

```
int fib(int k) {
    if (k <= 1) return k;
    return fib(k - 1) + fib(k - 2);
}
```

Dibujemos el árbol de llamadas de `fib(5)` ...

El árbol de `fib(5)`



fib(3) se calcula 2 veces · **fib(2)**, 3 veces · y peor cuanto mayor es n

`fib(3)` se calcula **2 veces**. `fib(2)`, **3 veces**. Y cuanto más grande el n, peor.

Estos son los **subproblemas superpuestos** que definimos: el mismo subproblema resuelto una y otra vez. **Ahí se va todo el tiempo.**

Técnicas

1. Idea
2. Técnicas
3. Ejercicios

Memoización: guarda y reúsa

La idea es simple: **la primera vez que calculo un subproblema, guardo su resultado**. La próxima vez que lo necesito, lo **leo** en lugar de recalcularlo.

```
vector<int> memo(n, -1)

int fib(int k) {
    if (k <= 1) return k;
    if (memo[k] != -1) return memo[k];
    return memo[k] = fib(k - 1) + fib(k - 2);
}
```

Ahora cada `fib(k)` se calcula **una sola vez**. A esto se le llama **top-down**.

Con memo vs sin memo

Es el **mismo código** con **tres líneas nuevas**… y un mundo de diferencia:

Sin memo · $O(2^n)$

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

$n = 50$ → nunca termina

Con memo · $O(n)$

```
int fib(int k) {
    if (k <= 1) return k;
    if (memo[k] != -1) return memo[k];
    return memo[k] = fib(k - 1) + fib(k - 2);
}
```

$n = 50$ → instantáneo

Memo para el profesor borracho

Lo mismo en nuestro problema: agregamos `memo` y `visto`, y **guardamos antes de retornar**.

```
int a[10];
vector<int> memo(a.size(), -1);

int resolver(int i) {
    if (i < 0) return 0;
    if (i == 0) return a[0];
    if (memo[i] != -1) return memo[i];
    return memo[i] = max(resolver(i - 1),
                         resolver(i - 2) + a[i]);
}
```

n estados, cada uno se calcula una vez → $O(n)$.

Tabulación: de abajo hacia arriba

Otra forma de lo mismo: en vez de recursión, **llenamos una tabla en orden**, desde los casos base.

```
int dp[100];
dp[0] = 0;
dp[1] = 1;                              // casos base
for (int i = 2; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 2];  // transición
```

Sin recursión ni pila de llamadas: solo un `for`. A esto se le llama **bottom-up**.

Top-down vs bottom-up

Top-down · memoización

- Recursión + `memo`.
- Calcula **solo lo que necesita**.
- Sale casi directo de la recurrencia.

Bottom-up · tabulación

- Un `for` que llena la tabla.
- Sin pila de recursión.
- Suele ser un poco más rápido.

Vocabulario de un DP

- **Estado**: cada subproblema (en el borracho, el índice `i`; en Fibonacci, `n`).
- **Transición**: cómo se combina con estados más pequeños (`dp[i-1]`, `dp[i-2]`).
- **Recurrencia**: la fórmula que liga el estado con los más pequeños.
- **Casos base**: los estados que se responden sin recursión.

Ejercicios

1. Idea
2. Técnicas
3. **Ejercicios**

Monedas: mínimo

Tienes monedas de valores $c_1, \ldots, c_n$ (puedes repetir cada una) y quieres formar una suma x usando la **menor cantidad de monedas**. Si es imposible, responde -1.

monedas	x	respuesta
`1 5 7`	11	3

$11 = 5 + 5 + 1 \rightarrow$ 3 monedas. No hay forma con 2.

Monedas: mínimo — código

```cpp
int minMonedas(int x, vector<int>& monedas) {
    vector<int> dp(x + 1, INT_MAX);
    dp[0] = 0;                                  // 0 monedas para formar 0
    for (int v = 1; v <= x; v++)
        for (int c : monedas)
            if (v - c >= 0)
                dp[v] = min(dp[v], dp[v - c] + 1);
    return dp[x] == INT_MAX ? -1 : dp[x];       // -1 si es imposible
}
```

x estados, cada uno prueba n monedas $\rightarrow O(n \cdot x)$.

Monedas: de cuántas formas

Mismas monedas: ahora cuenta **de cuántas maneras** puedes formar x (como el orden importa, $2+3$ y $3+2$ son distintas).

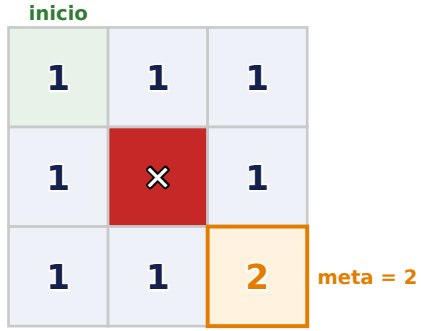
Es el **mismo esqueleto** con **dos cambios**: `min` → **suma**, y `dp[0] = 0` → `dp[0] = 1`.

```
int formas(int x, vector<int>& monedas) {
    vector<long long> dp(x + 1, 0);
    dp[0] = 1;                                  // una forma de formar 0: no usar nada
    for (int v = 1; v <= x; v++)
        for (int c : monedas)
            if (v - c >= 0)
                dp[v] = (dp[v] + dp[v - c]);
    return dp[x];
}
```

Caminos en grilla

Tienes una grilla de $n \times n$ ($1 \leq n \leq 1000$); cada casilla es **libre** (`.`) o una **trampa** (`*`). Cuenta **cuántos caminos** van desde la esquina **superior izquierda** hasta la **inferior derecha**, moviéndote solo a la **derecha** o hacia **abajo** y sin pisar trampas.

Como el número puede ser enorme, la respuesta se da **módulo** $10^9 + 7$.



Caminos en grilla — código

```cpp
const int MOD = 1e9 + 7;
vector<vector<long long>> dp(n + 1, vector<long long>(n + 1, 0));

for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) {
        if (g[i][j] == '*') continue;
        if (i == 1 && j == 1) { dp[i][j] = 1; continue; }
        dp[i][j] = (dp[i - 1][j] + dp[i][j - 1]) % MOD;
    }
```

Práctica: Mochila 0/1 (Knapsack)

Tienes n objetos; el objeto i pesa w_i y vale v_i. Con una mochila que aguanta peso W, elige objetos para **maximizar el valor total** sin pasarte del peso.

$1 \leq n \leq 1000, 1 \leq W \leq 10^5, 1 \leq v_i, w_i \leq 1000.$

$n \cdot W$	**pesos** w	**valores** v	**máx. valor**
4 · 10	`4 8 5 3`	`5 12 8 1`	13

Tomo los objetos de peso 4 y 5 → valen $5 + 8 = 13$, y pesan $9 \leq 10$.

Práctica: subsecuencia creciente (LIS)

Dado un arreglo de n enteros, encuentra el **largo** de la subsecuencia **estrictamente creciente** más larga (los elementos no tienen que ser contiguos).

$1 \leq n \leq 2 \cdot 10^5$, $1 \leq a_i \leq 10^9$.

n	**arreglo**	**respuesta**
8	7 3 5 3 6 2 9 8	4

Una de largo 4: $3, 5, 6, 9$.

¿Dudas?