Programación Dinamica (DP)

Por Ariel Parra.

[ΓΩ5]

Definición

La programación dinámica es una técnica que combina la exactitud de la búsqueda completa y la eficiencia de los algoritmos voraces (**Greedy**). La programación dinámica puede aplicarse si el problema se puede dividir en subproblemas superpuestos que pueden resolverse de forma independiente (**Divide & Conquer**).

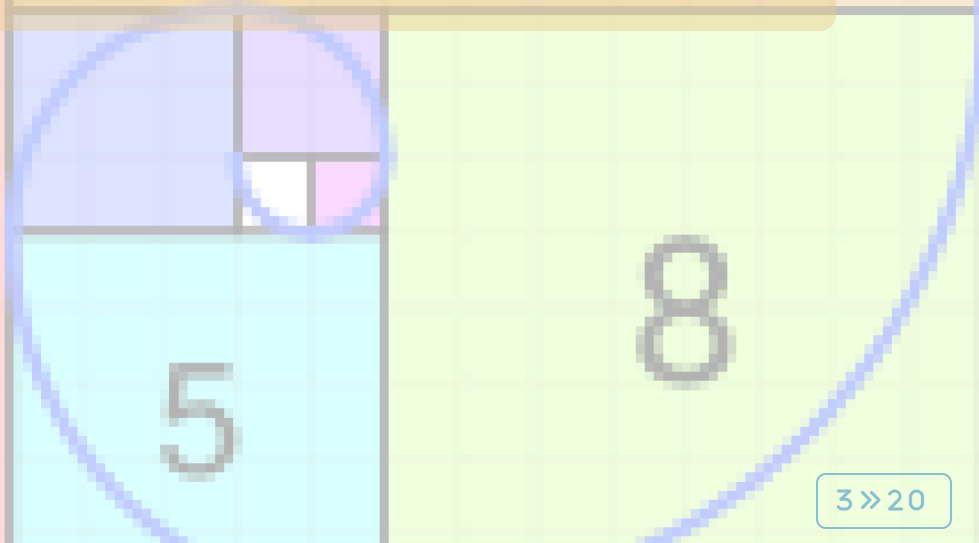
La programación dinámica tiene dos usos principales:

- **Sub-estructura óptima**. Un problema tiene sub-estructura óptima cuando la solución óptima a un problema se puede componer a partir de soluciones óptimas de sus sub-problemas.
- **Superposición de Problemas**. El cálculo de la solución óptima implica resolver muchas veces un mismo sub-problema. La cantidad de sub-problemas es “pequeña”.

> Al usar programación dinámica, la complejidad del algoritmo se reduce en el **tiempo** pero crece en el **espacio** debido al uso de estructuras como vectores o tablas. Estas estructuras almacenan las soluciones de los subproblemas, evitando el recalculo innecesario.

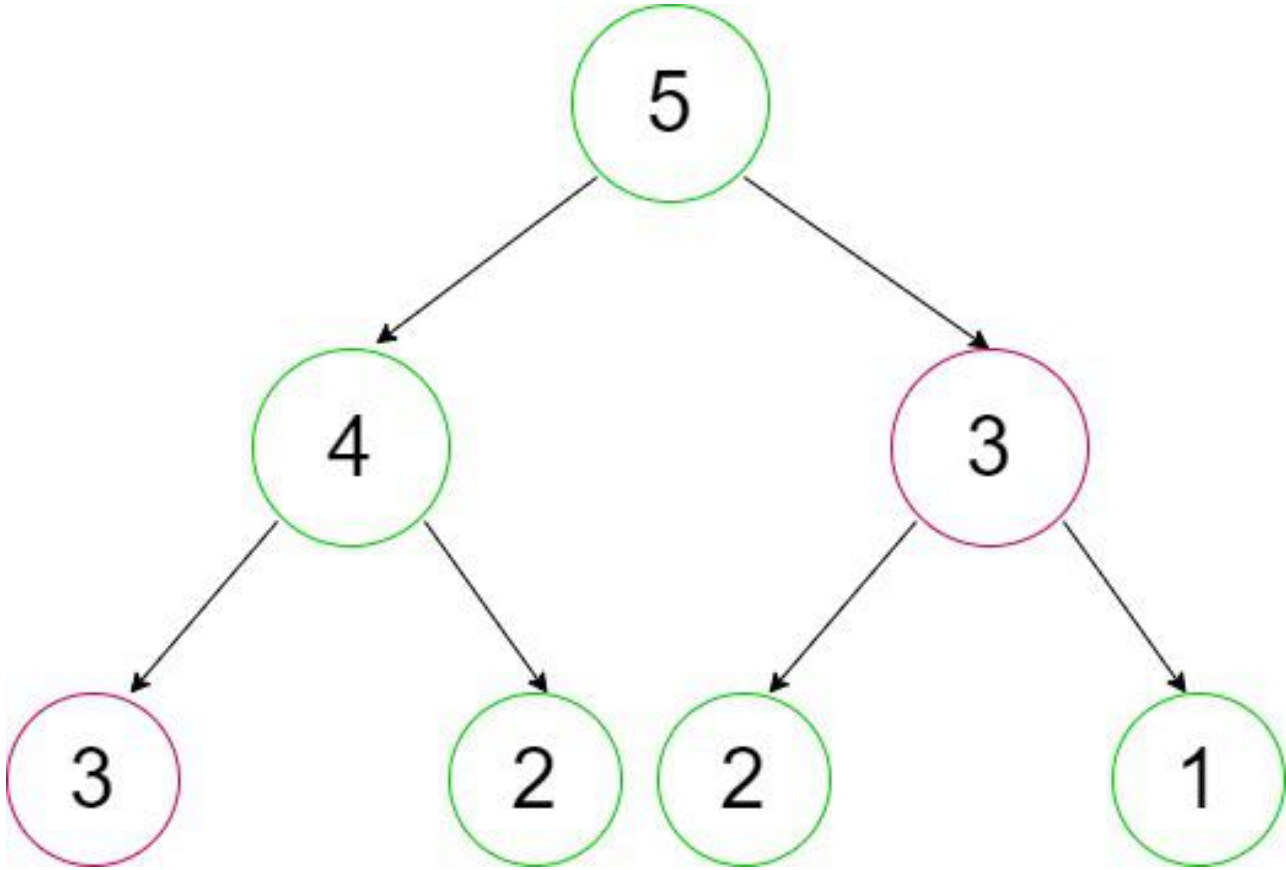
Sucesión de Fibonacci

> sucesión de Fibonacci es una sucesión infinita de números naturales. La sucesión comienza con dos números naturales (dependiendo de la referencia, con 0 y 1 en ciertos casos, otras inician con 1 y 1) y a partir de estos, **«cada término es la suma de los dos anteriores»**, es la relación de recurrencia que la define.



Fibonacci con recursión

```
ull fib(ull n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
    // O(2^ n) tiempo
}//  O(n) en memoria (profundidad de
//llamadas recursivas al stack)
```



Primera implementación con Top-Down

```cpp
ull fibo(ull n, vector<ull>& ans) {
    if (n <= 1)
        return n;
    if (ans[n] != -1)
        return ans[n];
    // tabulizando resultados
    ans[n] = fibo(n - 1, ans) + fibo(n - 2, ans);
    return ans[n];
}
int main() {
    ull n; cin >> n;
    vector<ull> ans(n + 1, -1);
    cout << fibo(n, ans) << endl;
    return 0;
}
```

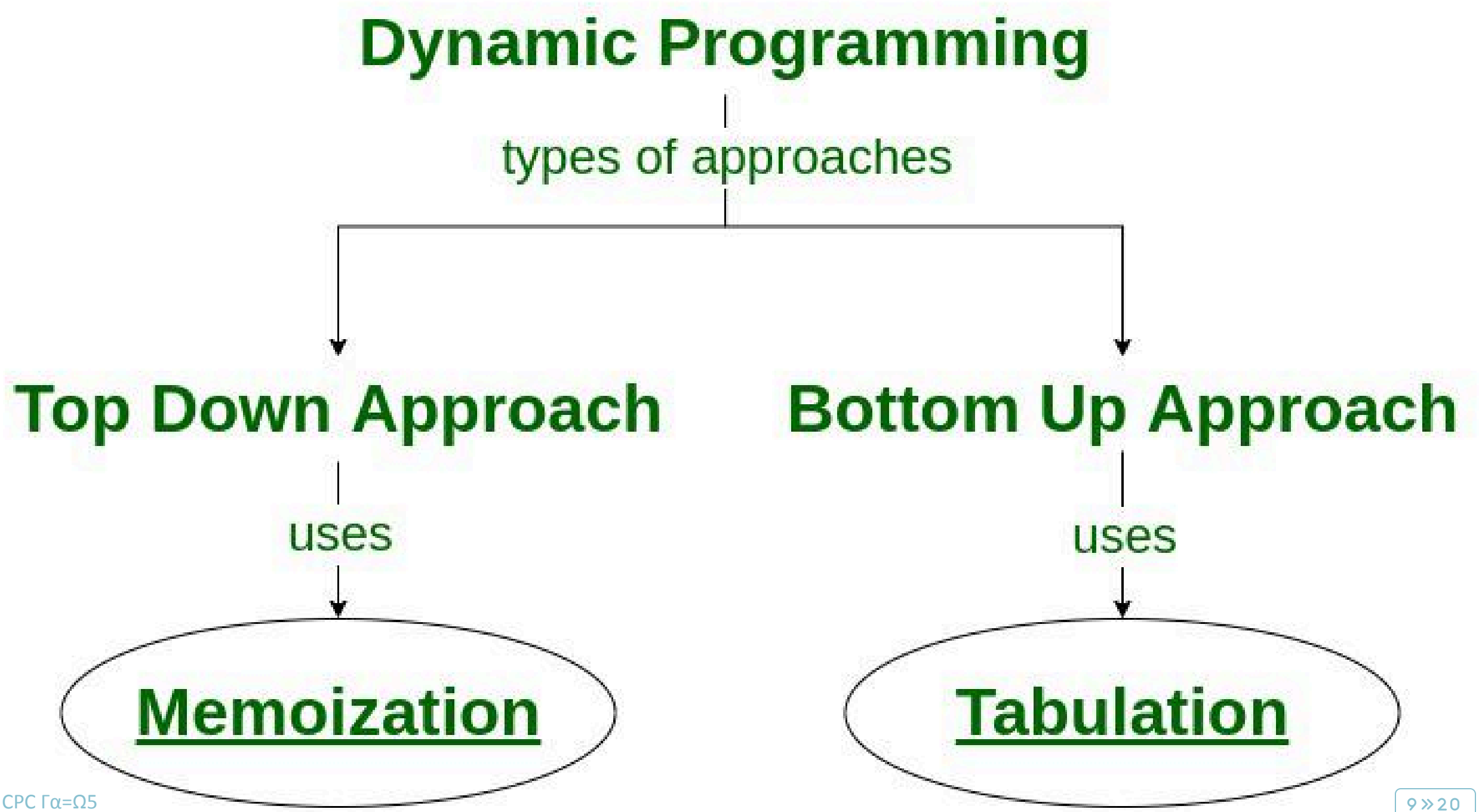
Implementación con bottom-up

```
ull fibo(ull n,vector<ull>& ans) {
    ans[0] = 0; ans[1] = 1;

    // fib iterativo
    for (ull i = 2; i <= n; ++i) {
        ans[i] = ans[i - 1] + ans[i - 2];
    }

    return ans[n];
}
int main() {
    ull n; cin >> n;
    vector <ull> ans(n+1);
    cout << fibo(n,ans);// O(n) en tiempo y espacio
    return 0;
}
```

Top-Down VS Bottom-Up

Top-Down	Bottom-Up
Pros:	**Pros:**
Transformación natural de la recursión de búsqueda completa.	**Más rápido** si se revisitan muchos subproblemas, ya que no hay sobrecarga por llamadas recursivas.
Calcula los subproblemas **solo cuando es necesario** (a veces es más rápido).	Computa subproblemas **solo cuando es necesario**, lo que puede ahorrar **memoria** con la técnica de DP "on-the-fly".

Top-Down	Bottom-Up
Contras:	**Contras:**
Más lento si se revisitan muchos subproblemas debido a la sobrecarga de llamadas recursivas (aunque esto no es penalizado en concursos de programación).	Para algunos programadores que prefieren la recursión, **puede no ser intuitivo**.
Si hay **M estados**, puede usar una tabla de tamaño **O(M)**, lo que puede llevar a un **MLE** en algunos problemas difíciles.	El DP de abajo hacia arriba **visita y llena** todos los M estados, lo que puede ser costoso en tiempo.


Dynamic Programming
types of approaches
Top Down Approach
Bottom Up Approach
uses
uses
Memoization
Tabulation

Criba de eratóstenes (Sieve of eratosthenes)

> La criba de Eratóstenes es un algoritmo que permite hallar muchos números primos menores que un número natural dado. Se forma una tabla con todos los números naturales comprendidos entre 2 y n, y se van tachando los números que no son primos de la siguiente manera

Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107
109	113		

```cpp
void sieve_of_eratosthenes(int n,vector<bool>& is_prime) {
    is_prime[0] = is_prime[1] = false;

    for (int p = 2; p * p <= n; ++p) {
        if (is_prime[p]) {
            for (int i = p * p; i <= n; i += p) {
                is_prime[i] = false;
            }
        }
    }
}

int main() {
    int n = 50; // Example: Find primes up to 50
    std::cout << "Primes up to " << n << " are: ";
    vector<bool> is_prime(n + 1, true);
    sieve_of_eratosthenes(n,is_prime);
    return 0;
}
```

Classical DP Examples

1. **Coin Change (CC)**: Encontrar el mínimo número de monedas o el número total de formas para hacer un cambio con un conjunto dado de denominaciones.
2. **Maximum Sum**: Hallar la subsecuencia o submatriz contigua con la suma máxima en una lista o matriz de números.
3. **Matrix Chain Multiplication**: Determinar el orden óptimo de multiplicación de una secuencia de matrices para minimizar el número de operaciones.
4. **Optimal Binary Search Tree**: Construir un árbol de búsqueda binario que minimice el costo promedio de búsqueda, dado un conjunto de claves y probabilidades.
5. **Paths in a Grid**: Contar o encontrar el camino óptimo desde una esquina de una cuadrícula a otra, con movimientos permitidos (como derecha y abajo).

6. **Knapsack Problems**: Seleccionar ítems con el mayor valor posible sin exceder un peso máximo.
7. **Edit Distance**: Calcular el número mínimo de operaciones (inserción, eliminación, sustitución) para transformar una cadena en otra.
8. **Counting Tilings**: Contar de cuántas formas se puede cubrir una región usando piezas específicas.
9. **Longest Increasing Subsequence (LIS)**: Encontrar la subsecuencia más larga que sea estrictamente creciente en una secuencia de números.
10. **Longest Common Subsequence (LCS)**: Hallar la subsecuencia más larga que sea común a dos secuencias, manteniendo el orden pero no la contigüidad.

Los 5 pasos para resolver problemas de DP según Reducible

1. **Visualizar ejemplos**: Es crucial para identificar patrones y restricciones en el problema. Utilizar modelos como grafos acíclicos dirigidos puede ser útil para representar soluciones de manera más clara.
2. **Definir subproblemas**: Encontrar una versión más simple del problema original que pueda ser resuelta. En el ejemplo de la subsecuencia creciente más larga, un subproblema podría ser encontrar la longitud de la subsecuencia terminando en un índice específico.
3. **Encontrar relaciones entre subproblemas**: Determinar cómo se conectan los subproblemas y cómo las soluciones de subproblemas pueden usarse para resolver el problema completo.
4. **Generalizar la relación**: Crear una fórmula o regla general para resolver cualquier instancia del subproblema basándose en la relación que se haya encontrado.
5. **Implementar la solución**: Resolver los subproblemas en el orden correcto, asegurándose de que los subproblemas necesarios para resolver un problema mayor ya hayan sido resueltos antes.

Problema en clase: "Dice Combinations" (Knapsack)

Tu tarea consiste en contar la cantidad de formas de construir la suma `n` lanzando un dado una o más veces. Cada lanzamiento produce un resultado entre 1 y 6.

Por ejemplo, si n=3, hay 4 formas:

```
1+1+1
1+2
2+1
3
```

> Recuerda que un problema Knapsack puede modelarse como el llenado de un contenedor de tamaño limitado con elementos (solución DP).

1. Definición del problema en términos de DP

- **Problema**: Queremos contar cuántas secuencias de tiradas de dados suman exactamente `n`, donde cada tirada produce un número entre `1` y `6`.
- **Estado DP (`dp[x]`)**: `dp[x]` representa la cantidad de formas de obtener la suma `x` usando secuencias de tiradas de dados. El valor final deseado es `dp[n]`.

2. Identificación de la relación de recurrencia

- La relación entre los estados se basa en considerar la última tirada de dado que pudo haberse lanzado para obtener una suma de `x`.
- La relación de recurrencia es:

$$\mathrm{dp}[x] = \sum_{i=1}^{6} \mathrm{dp}[x-i]$$

  Esto significa que para obtener `x`, podemos sumar todas las formas de obtener `x-1`, `x-2`, ..., `x-6`, asumiendo que se puede hacer una última tirada de 1 a 6.

3. Inicialización de la solución

- Inicializar el vector `dp` con `dp[0] = 1`, ya que hay exactamente 1 forma de sumar 0: no hacer ninguna tirada.
- El vector `dp` se utiliza como "memo" para almacenar los resultados de subproblemas ya resueltos.

4. Recorrer los subproblemas

- Iterar para calcular `dp[i]` para cada valor de `i` desde `1` hasta `n`, siguiendo la relación de recurrencia. Esto asegura que cada subproblema (suma menor que `n`) sea resuelto antes de resolver el subproblema final `dp[n]`.

5. Optimización modular

- Dado que los números pueden ser grandes, es necesario tomar el resultado módulo `10^9 + 7` después de cada operación de suma para evitar desbordamientos y cumplir con las restricciones del problema.

6. Resultado final

- El valor de `dp[n]` contiene la cantidad de formas de obtener la suma `n` y se imprime como la solución final.

```
vector<unsigned long long> dp(1e6+1); // también suelen llamar a este vector "memo"
int n;
cin >> n;
dp[0] = 1; // Base del DP: hay una manera de sumar 0 (no hacer nada)

for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= 6; ++j) {
        if (i - j >= 0)  dp[i] += dp[i - j];
    }
    dp[i] %= 1e9+7; // El problema pide imprimir con módulo 10^9+7
}
cout << dp[n];// O(n) tiempo & espacio
```

Problemas

- **Meta Hacker Cup 2024 R1 Problem B** Prime Subtractorization ⮭
- **687C** The Values You Can Make ⮭

Referencias

- *Dynamic Programming (DP) Tutorial with Problems*. Recuperado de https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/
- http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf
- Cao, M. et al. (s.f.). *Introduction to DP*. Recuperado de https://usaco.guide/gold/intro-dp
- Chen, N. et al. (s.f.). *Knapsack DP*. Recuperado de https://usaco.guide/gold/knapsack?lang=cpp
- kapoor, K. (2016). *Everything About Dynamic Programming*. Recuperado de https://codeforces.com/blog/entry/43256
- Reducible. (2020). *5 Simple Steps for Solving Dynamic Programming Problems*. Recuperado de https://www.youtube.com/watch?v=aPQY__2H3tE