Programación Dinámica

Miguel Ortiz

Programación Competitiva para ICPC

Octubre 2023

Programación Dinámica

- Paradigma de resolución de problemas
- Lo que vimos hasta ahora se resume a reducir un problema a una forma en la que podamos aplicar un algoritmo (búsqueda binaria, dfs, etc.)

Programación Dinámica

- Paradigma de resolución de problemas
- Lo que vimos hasta ahora se resume a reducir un problema a una forma en la que podamos aplicar un algoritmo (búsqueda binaria, dfs, etc.)
- Para resolver problemas con programación dinámica debemos diseñar nuestro propio algoritmo y aplicar optimizaciones

Programación Dinámica

- Paradigma de resolución de problemas
- Lo que vimos hasta ahora se resume a reducir un problema a una forma en la que podamos aplicar un algoritmo (búsqueda binaria, dfs, etc.)
- Para resolver problemas con programación dinámica debemos diseñar nuestro propio algoritmo y aplicar optimizaciones
- Podemos verificar que nuestra solución cumple con las propiedades para aplicar programación dinámica

Programación Dinámica

- Podemos resumir el paradigma como:
    1. Hallar una solución recursiva
    2. Aplicar *memoizacion*

Programación Dinámica

- Podemos resumir el paradigma como:
  1. Hallar una solución recursiva
  2. Aplicar *memoizacion*

- Método SRTBOT para diseño de soluciones recursivas

SRTBOT

- Subproblemas
- Relaciones
- Orden Topológico
- Caso Base
- Problema Original
- Complejidad de Tiempo

Programación Dinámica - Ejemplo

- La secuencia Fibonacci es una serie en la que cada número después de los dos primeros es la suma de los dos números anteriores, comenzando con 0 y 1
- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

Programación Dinámica - Ejemplo

- La secuencia Fibonacci es una serie en la que cada número después de los dos primeros es la suma de los dos números anteriores, comenzando con 0 y 1
- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$

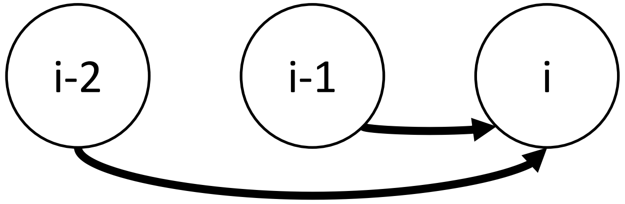
- Dado n, queremos encontrar el elemento F_n de la secuencia

Programación Dinámica - Ejemplo

- Subproblemas:
  - F_i para todo $i <= n$
- Relaciones:
  - $F_i = F_{i-1} + F_{i-2}$

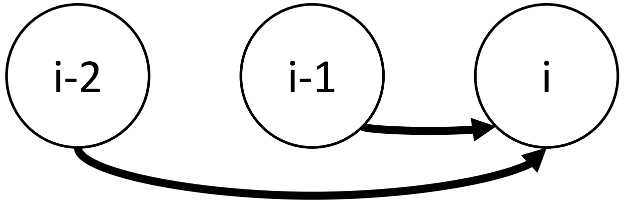
Programación Dinámica - Ejemplo

- Subproblemas:
  - F_i para todo $i \le n$
- Relaciones:
  - $F_i = F_{i-1} + F_{i-2}$
- Orden topologico:
  - Para calcular el valor del elemento i necesitamos el valor de los elementos i-1 y i-2



Programación Dinámica - Ejemplo

- Subproblemas:
  - F_i para todo i <= n
- Relaciones:
  - $F_i = F_{i-1} + F_{i-2}$
- Orden topologico:
  - Para calcular el valor del elemento i necesitamos el valor de los elementos i-1 y i-2



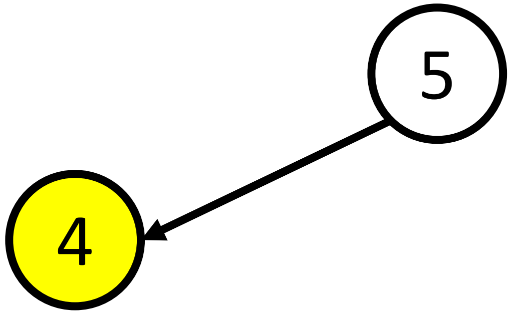
  - Es importante verificar que no hayan ciclos para que la recursion termine

Programación Dinámica - Ejemplo

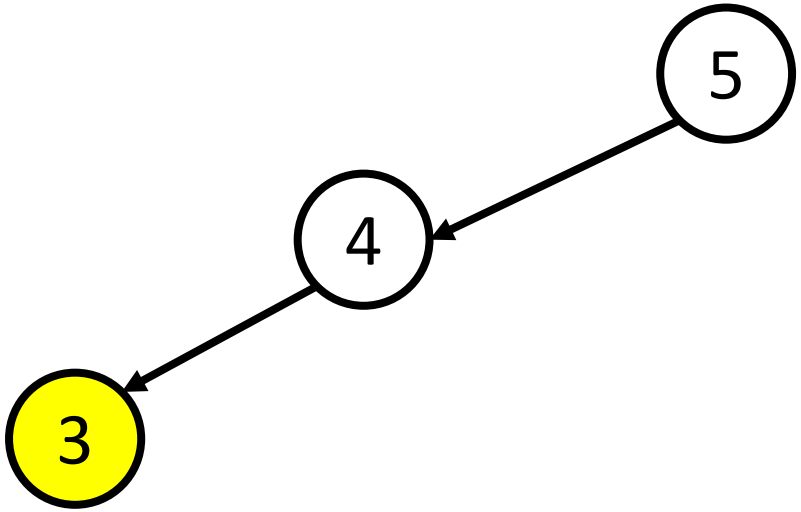
- Caso Base:
  - $F_0 = 0$
  - $F_1 = 1$
- Problema original:
  - F_n
- Tiempo:
  - $\sim O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones

Programación Dinámica - Ejemplo

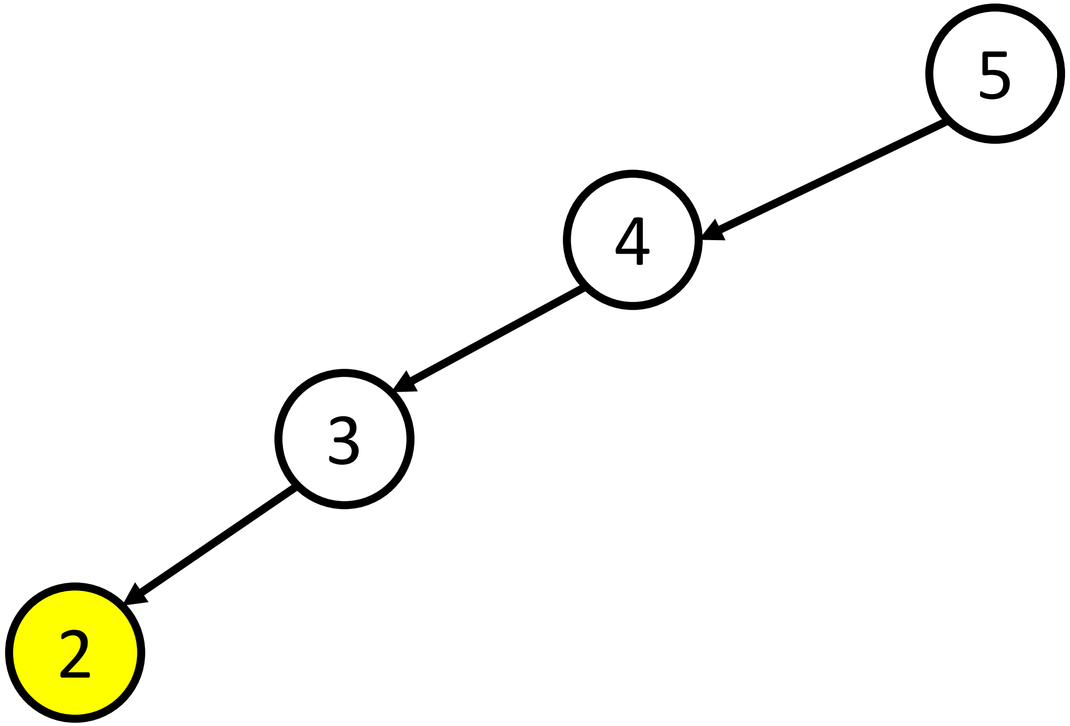
Programación Dinámica - Ejemplo



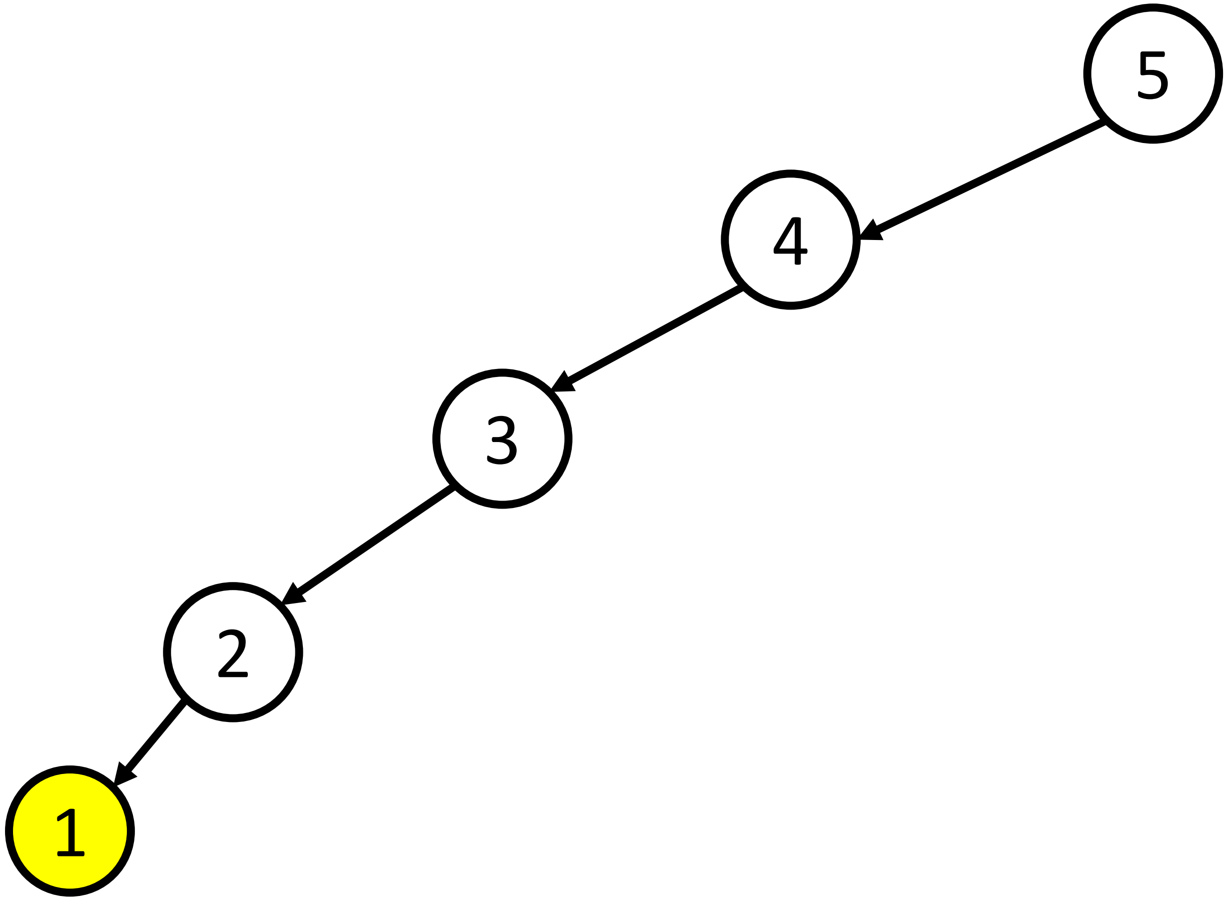
Programación Dinámica - Ejemplo



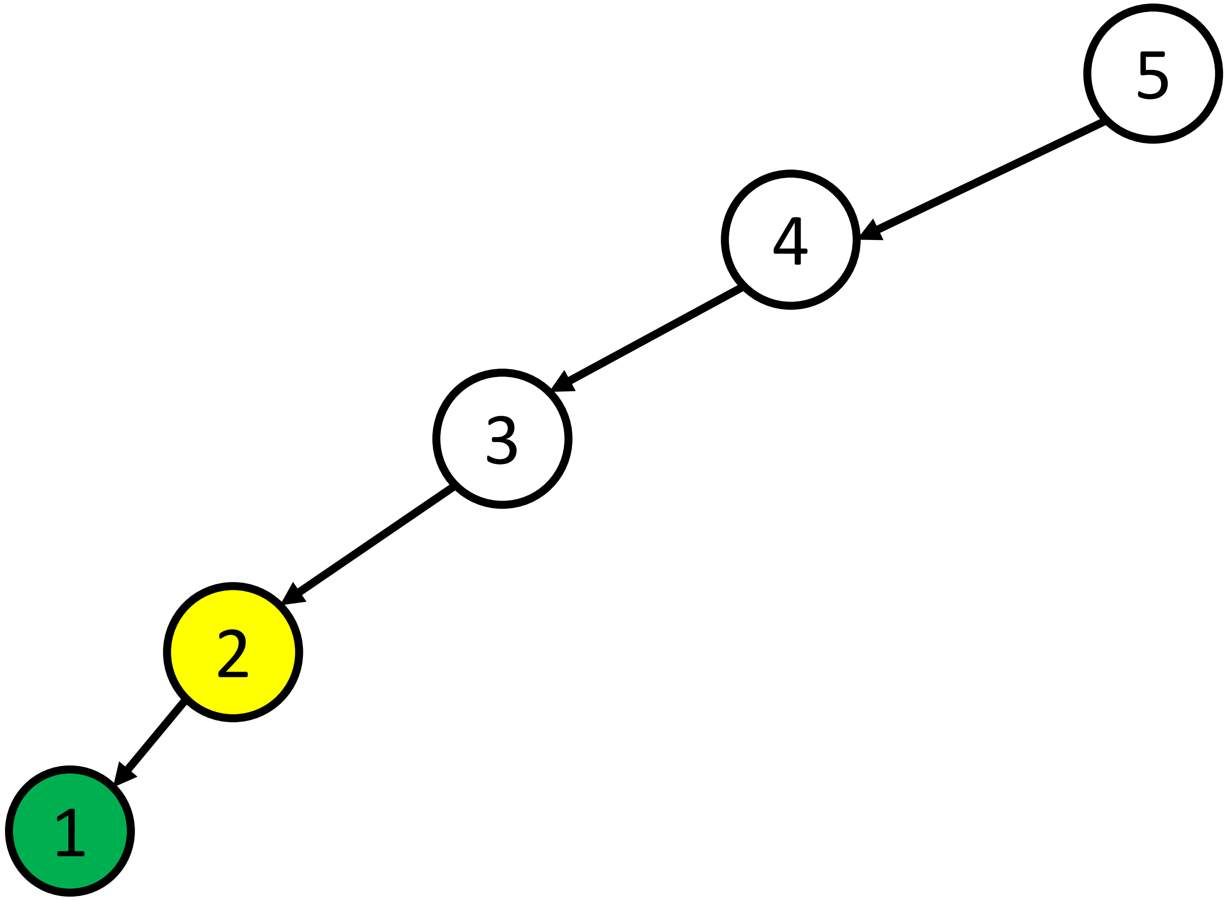
Programación Dinámica - Ejemplo



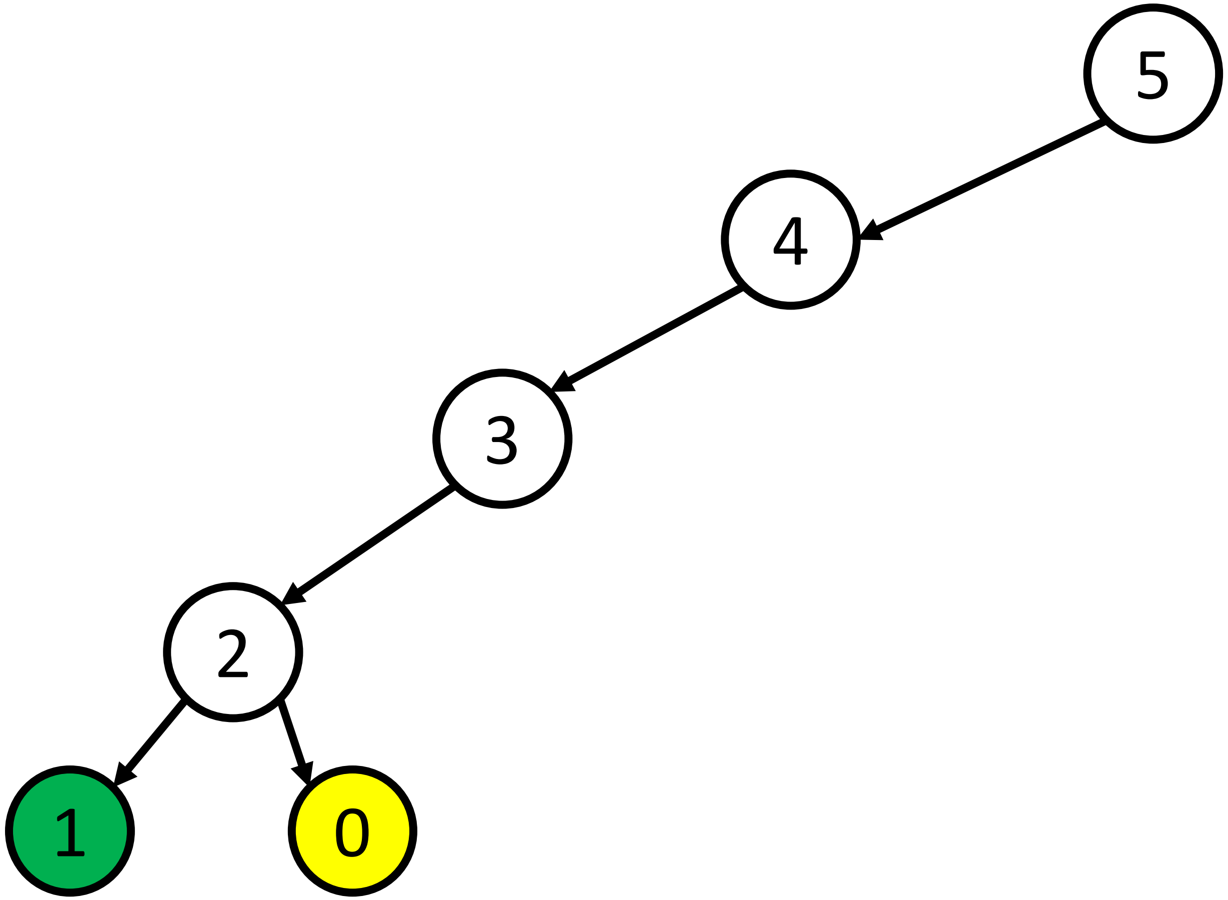
Programación Dinámica - Ejemplo



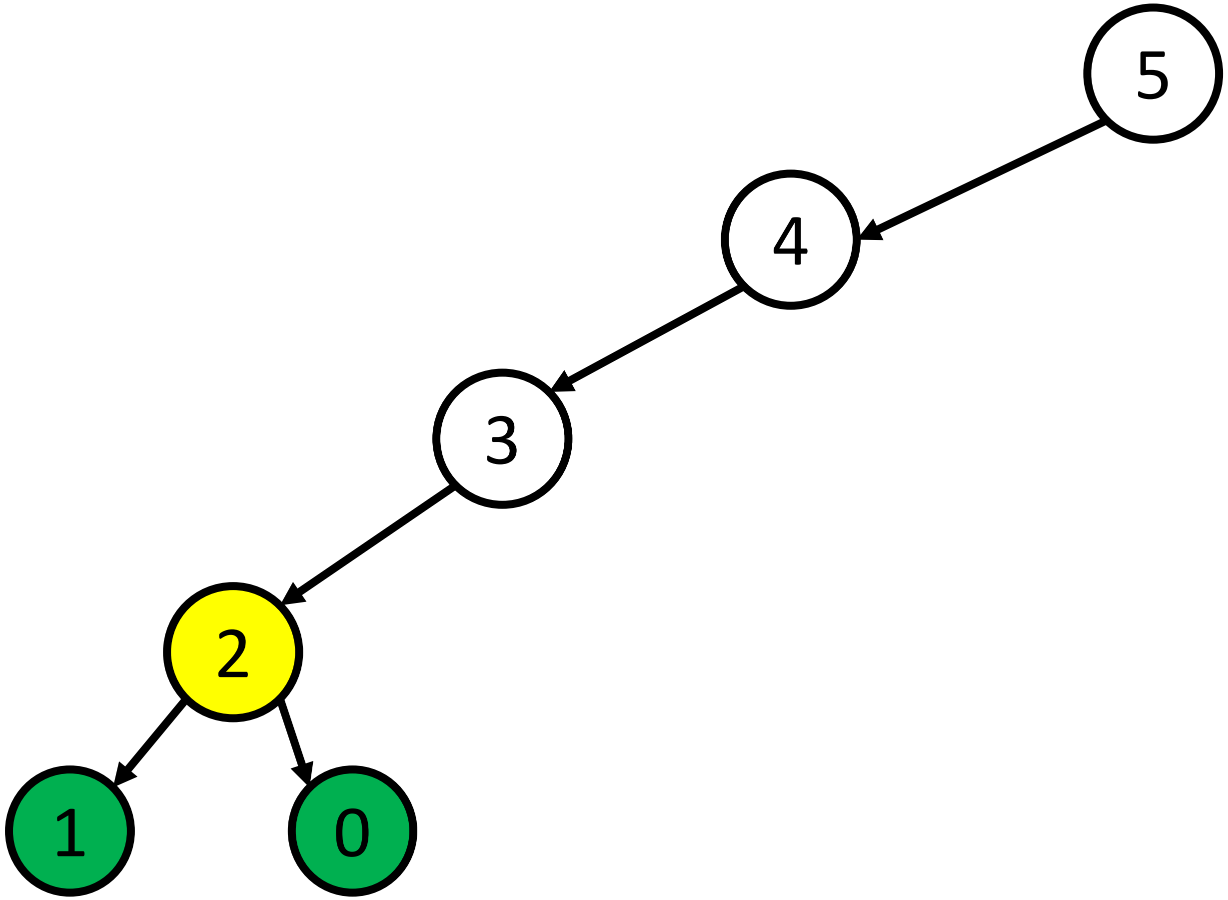
Programación Dinámica - Ejemplo



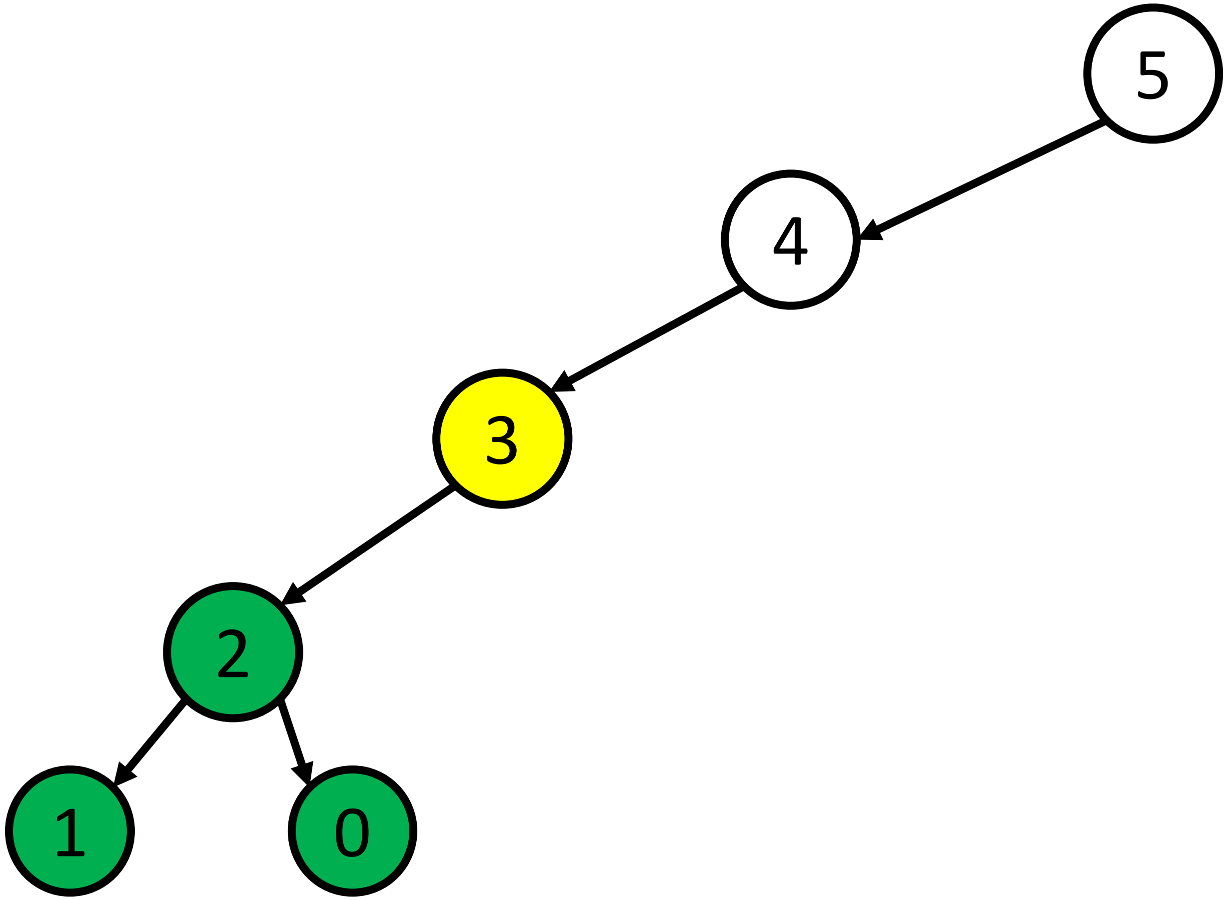
Programación Dinámica - Ejemplo



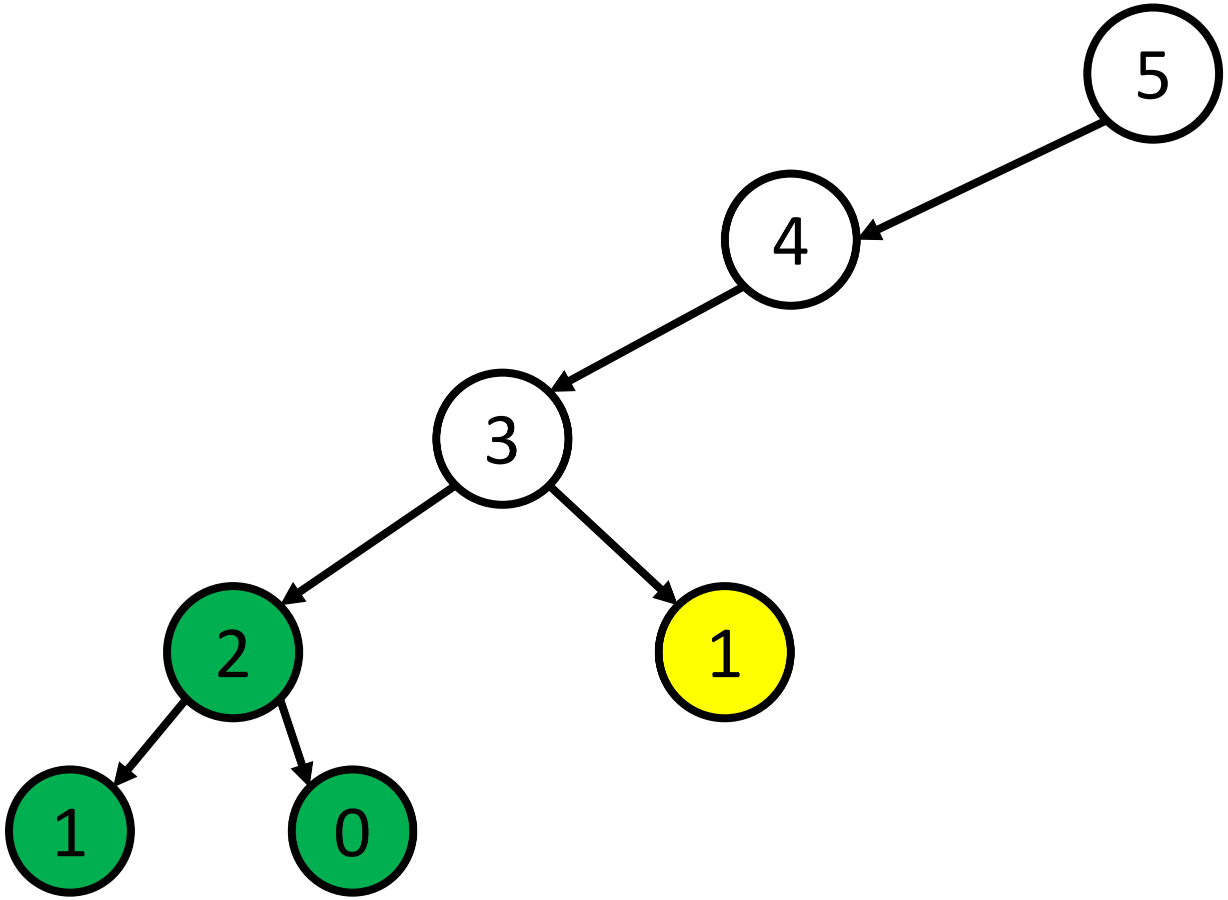
Programación Dinámica - Ejemplo



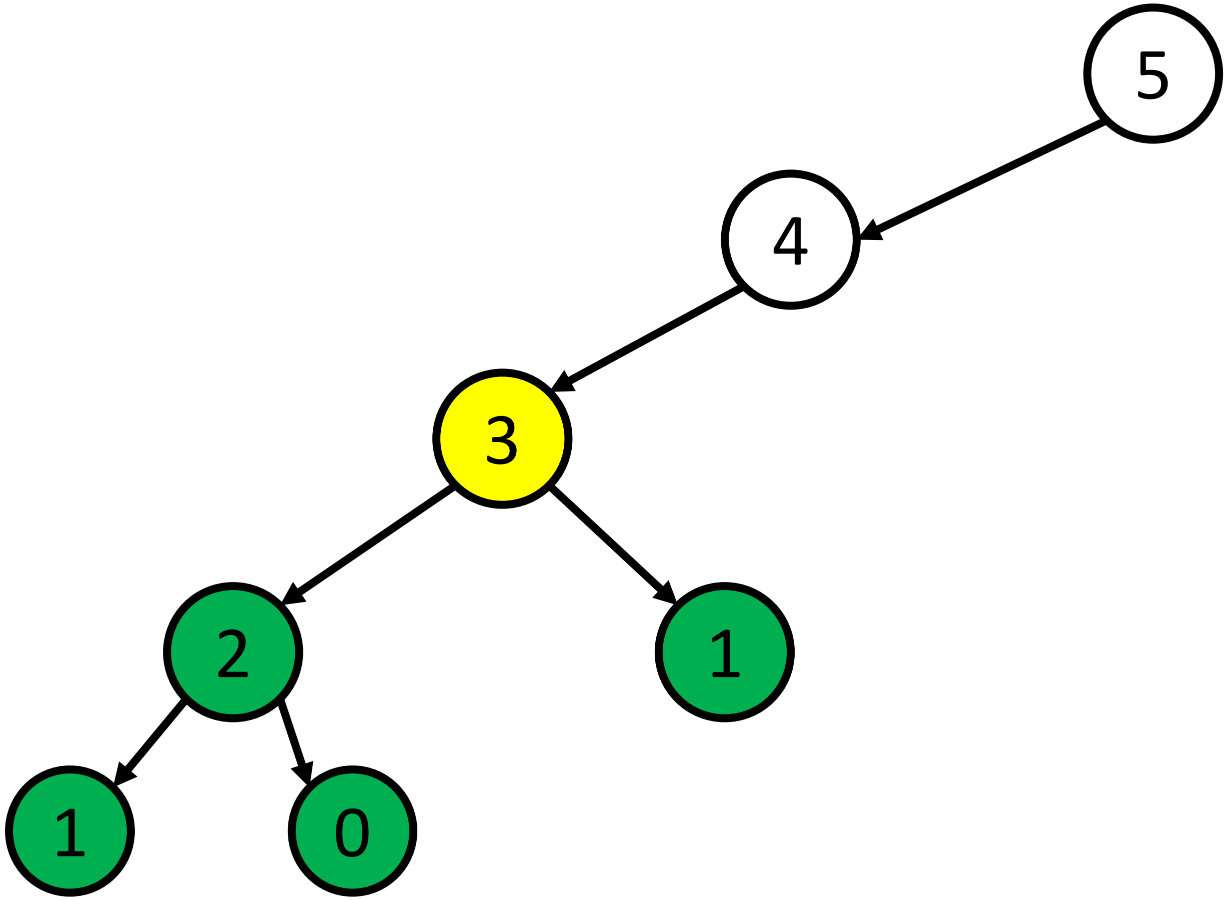
Programación Dinámica - Ejemplo



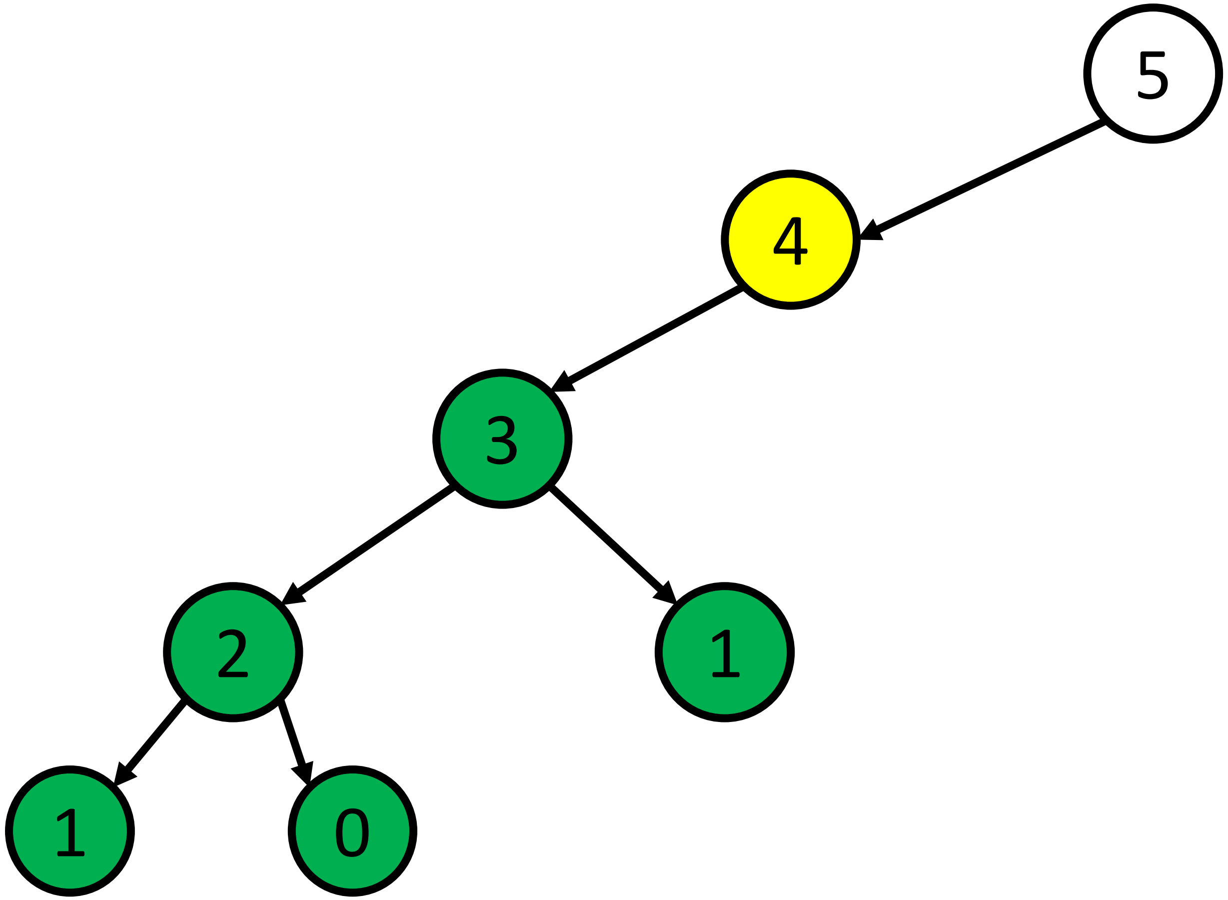
Programación Dinámica - Ejemplo



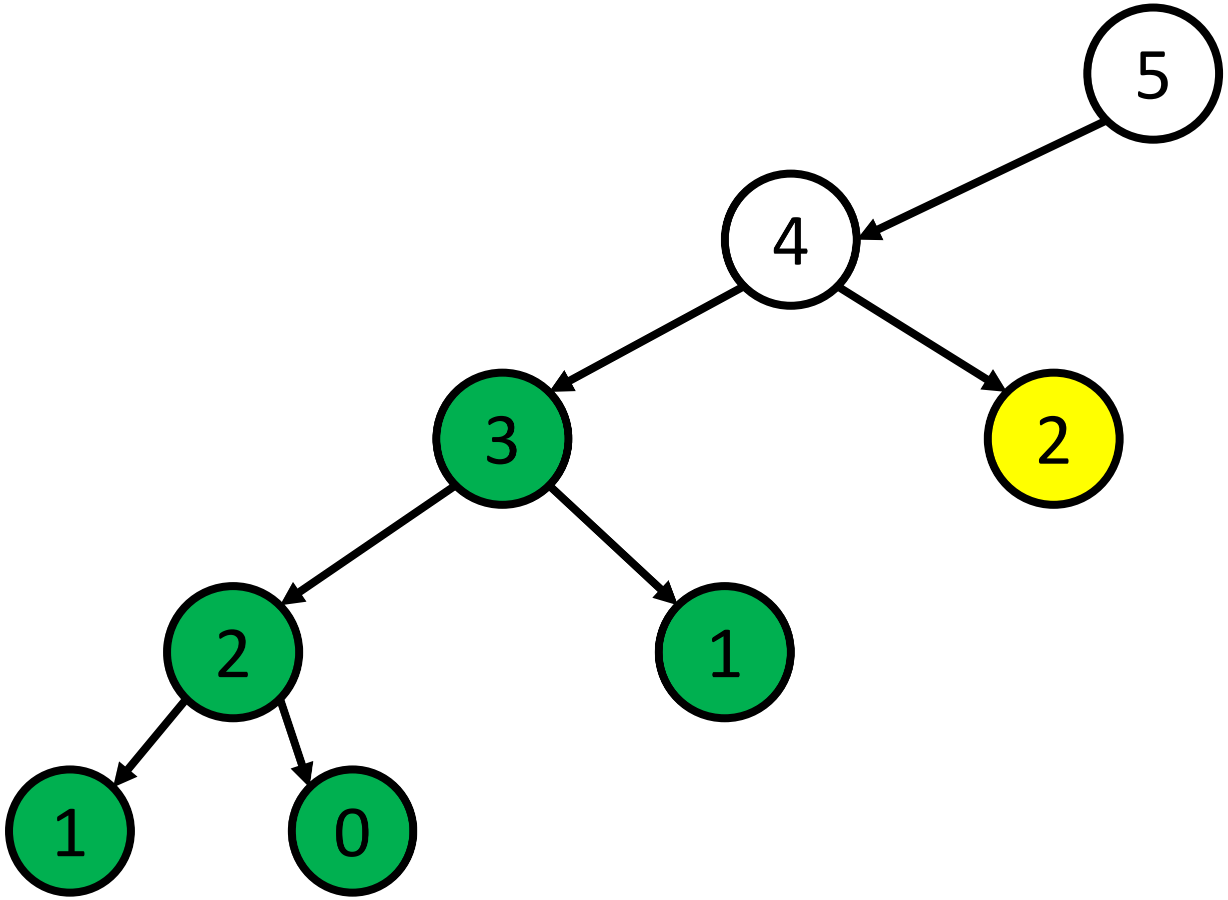
Programación Dinámica - Ejemplo



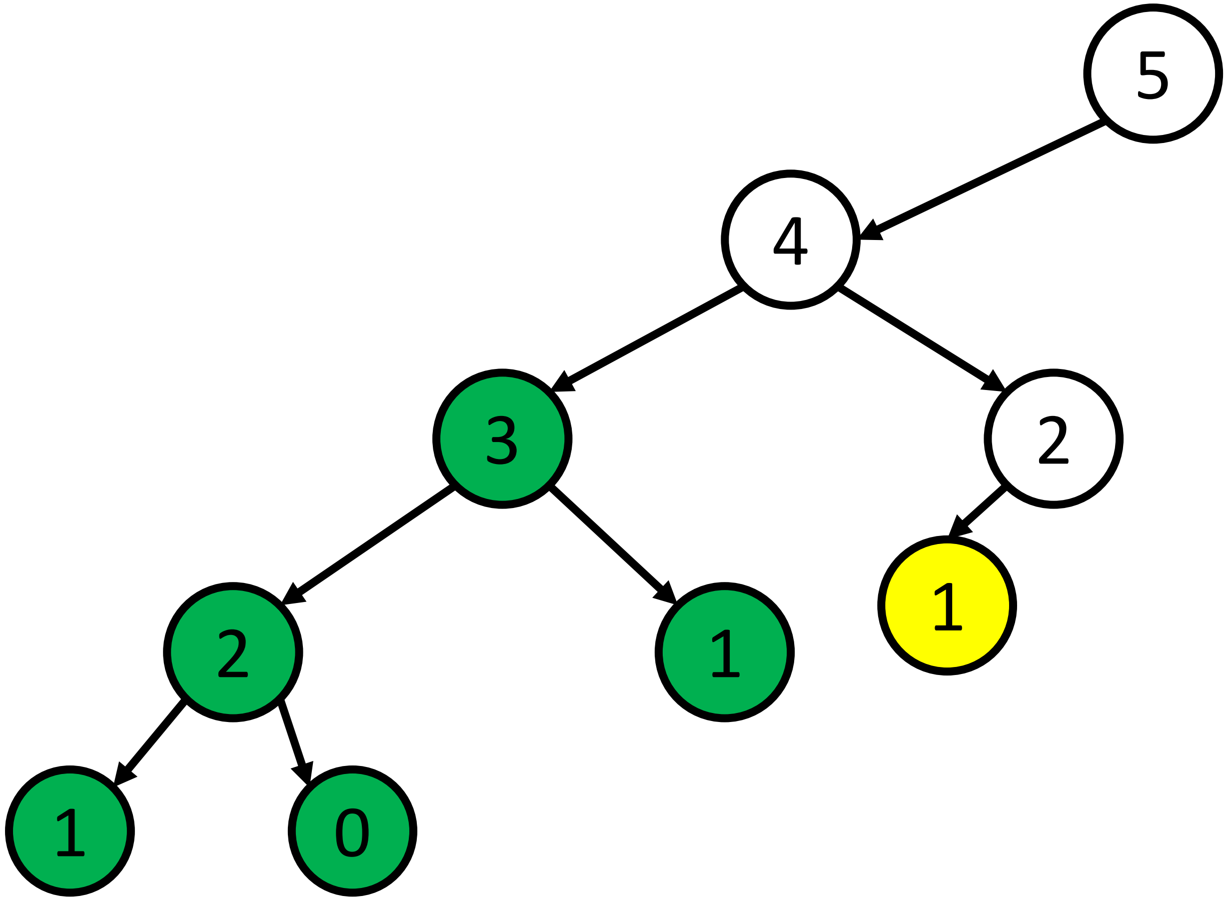
Programación Dinámica - Ejemplo



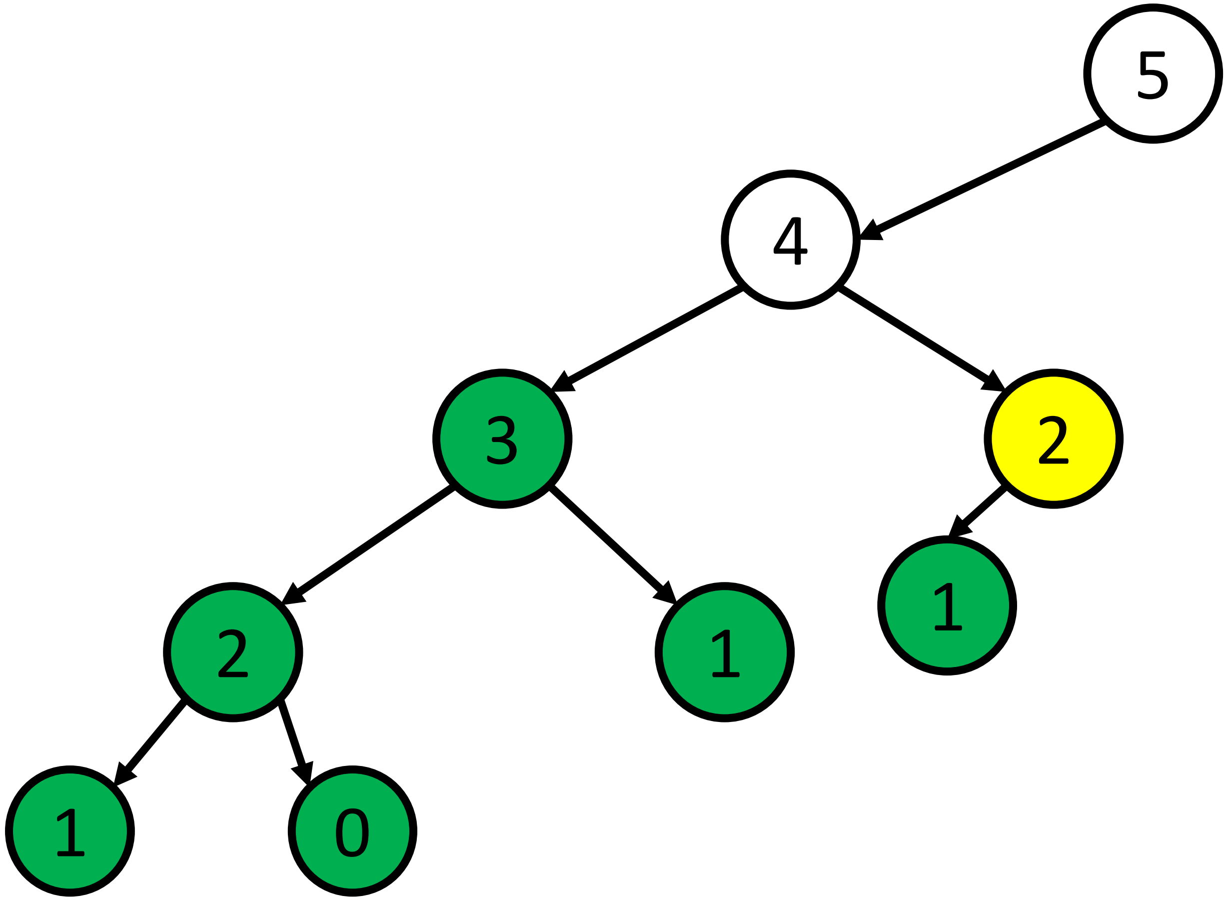
Programación Dinámica - Ejemplo



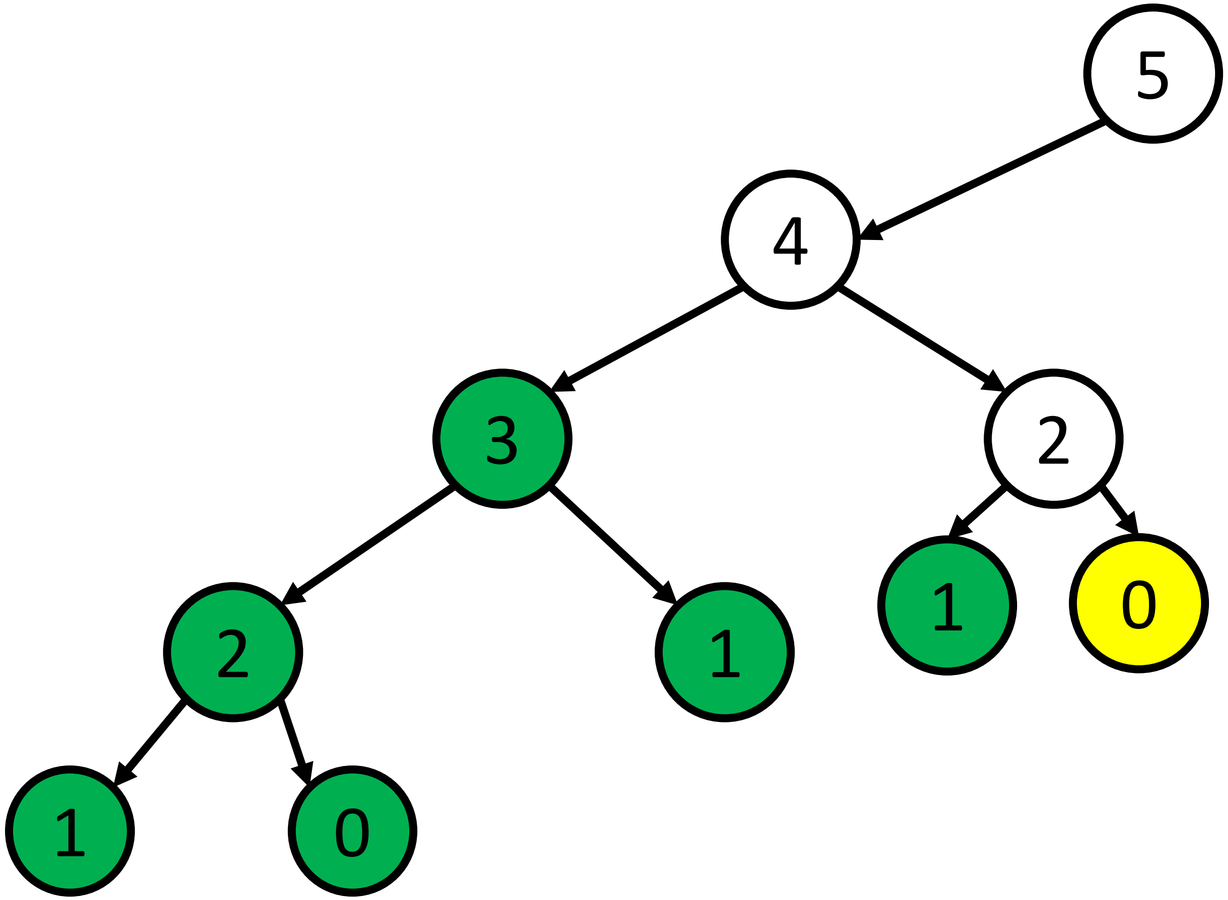
Programación Dinámica - Ejemplo



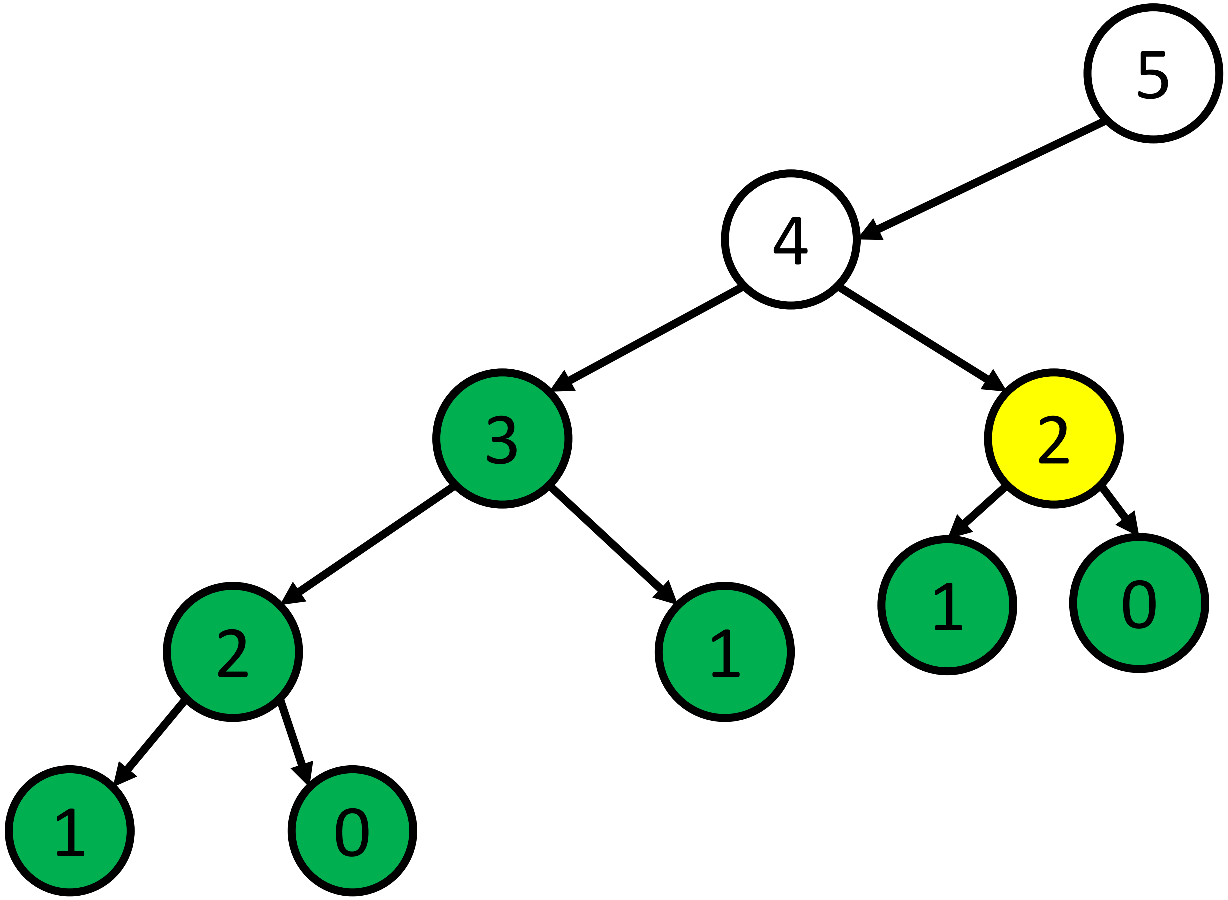
Programación Dinámica - Ejemplo



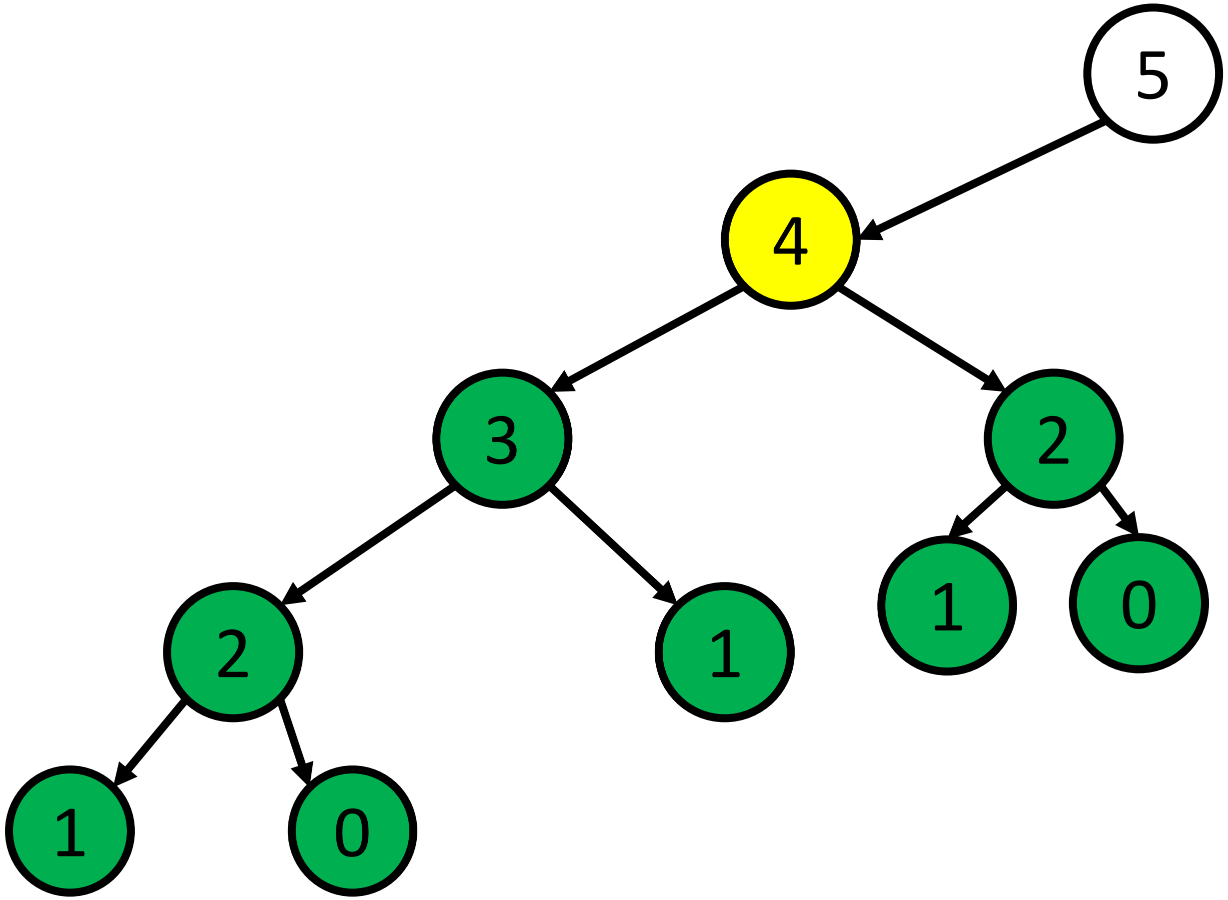
Programación Dinámica - Ejemplo



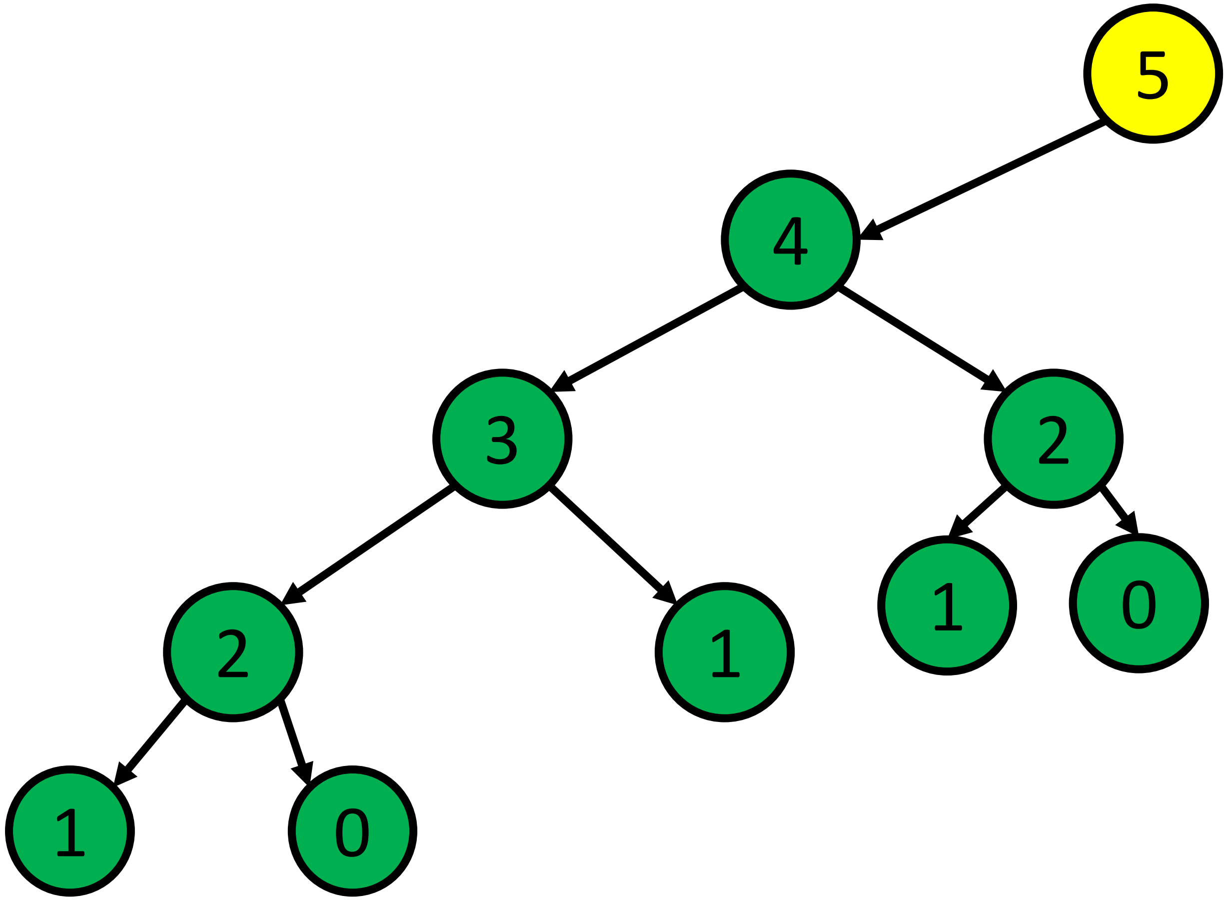
Programación Dinámica - Ejemplo



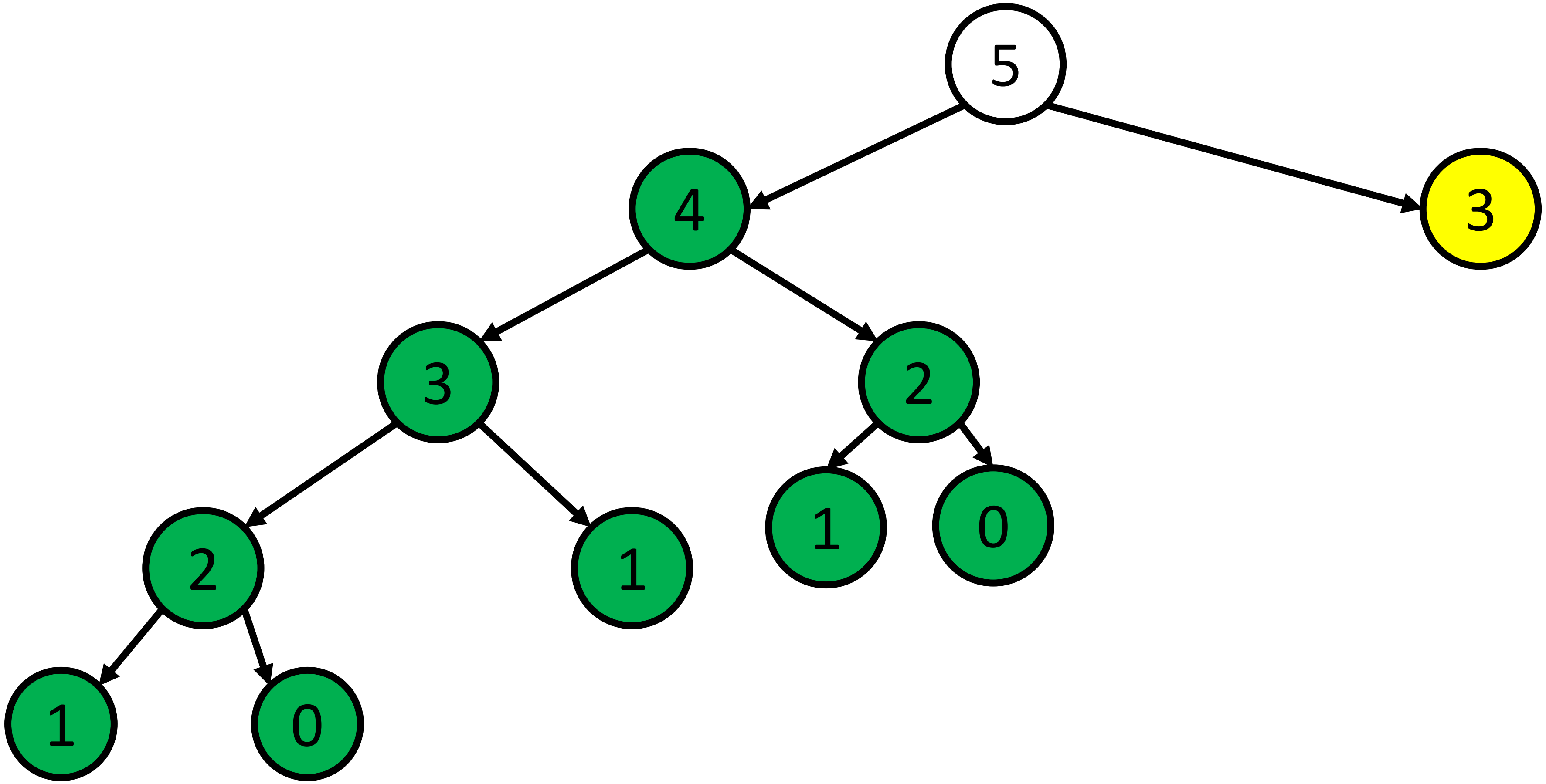
Programación Dinámica - Ejemplo



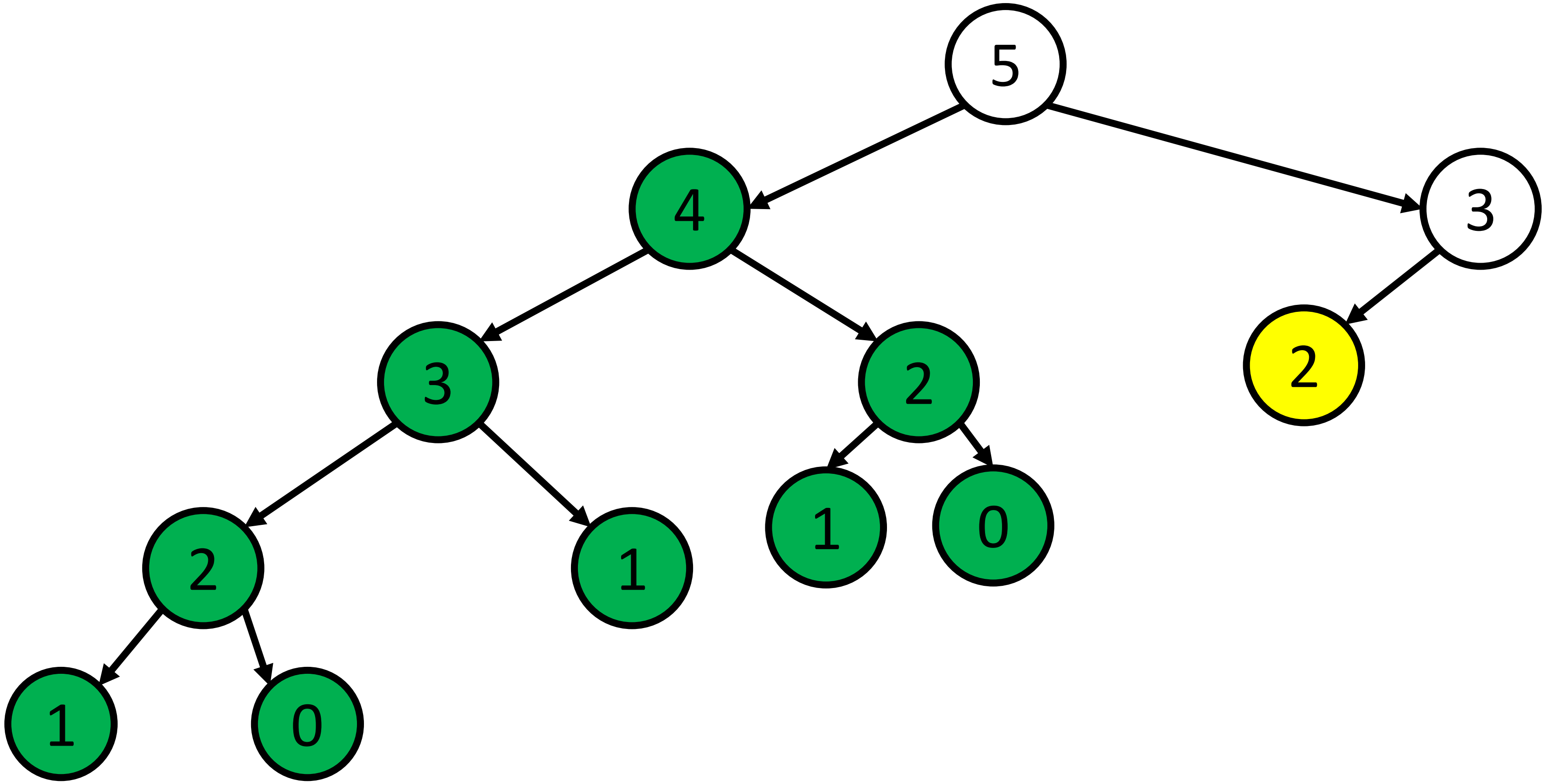
Programación Dinámica - Ejemplo



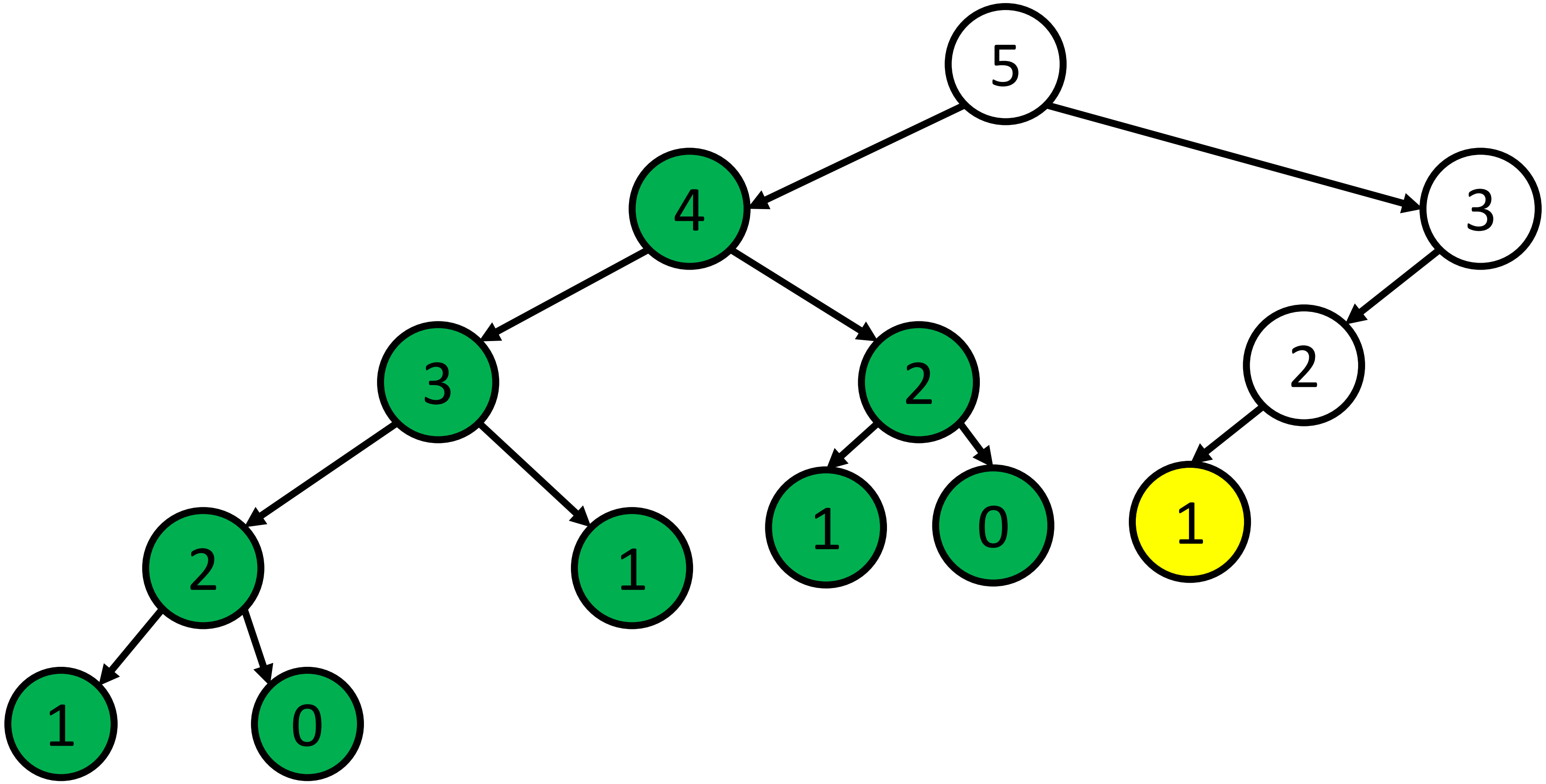
Programación Dinámica - Ejemplo



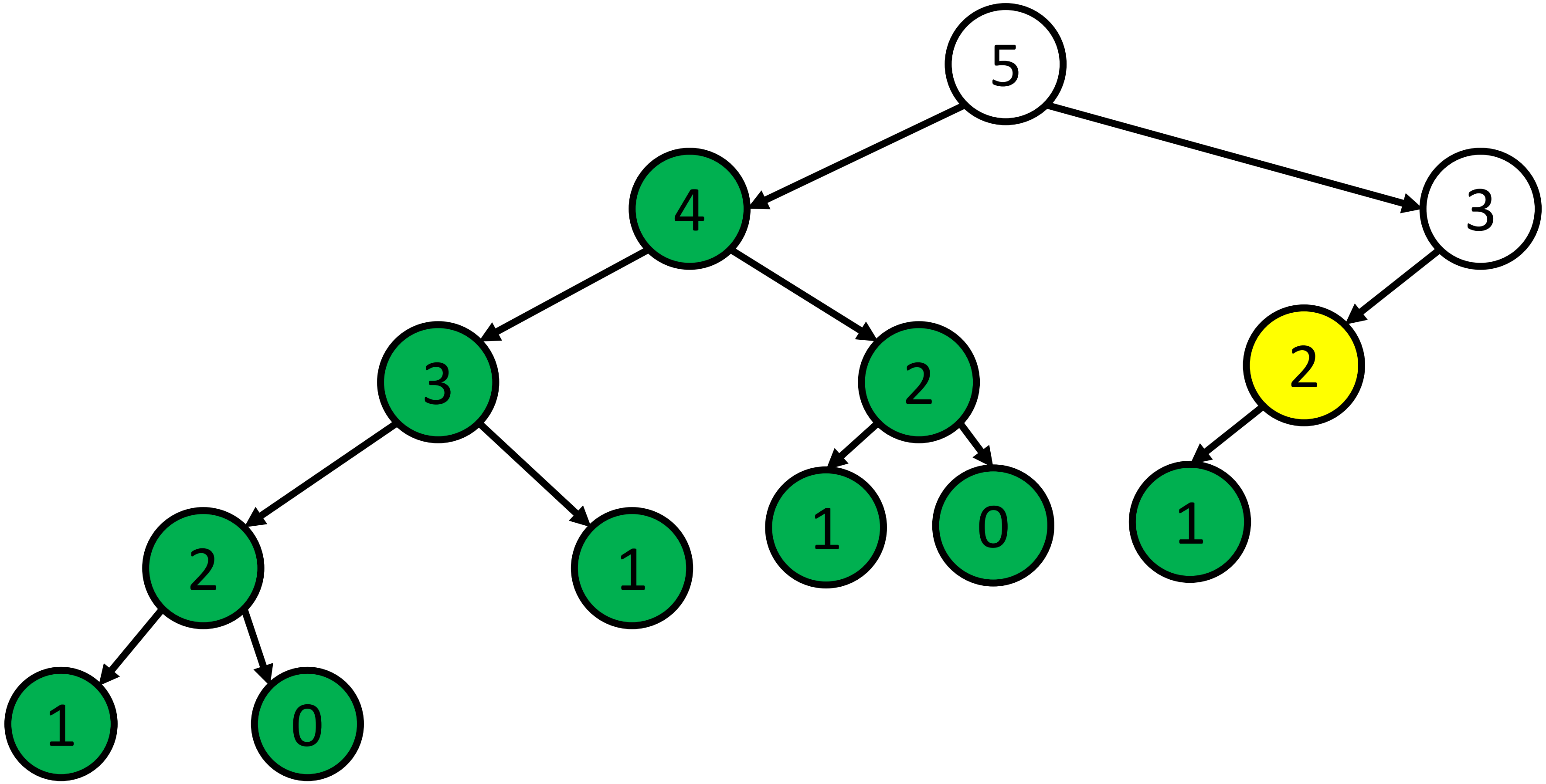
Programación Dinámica - Ejemplo



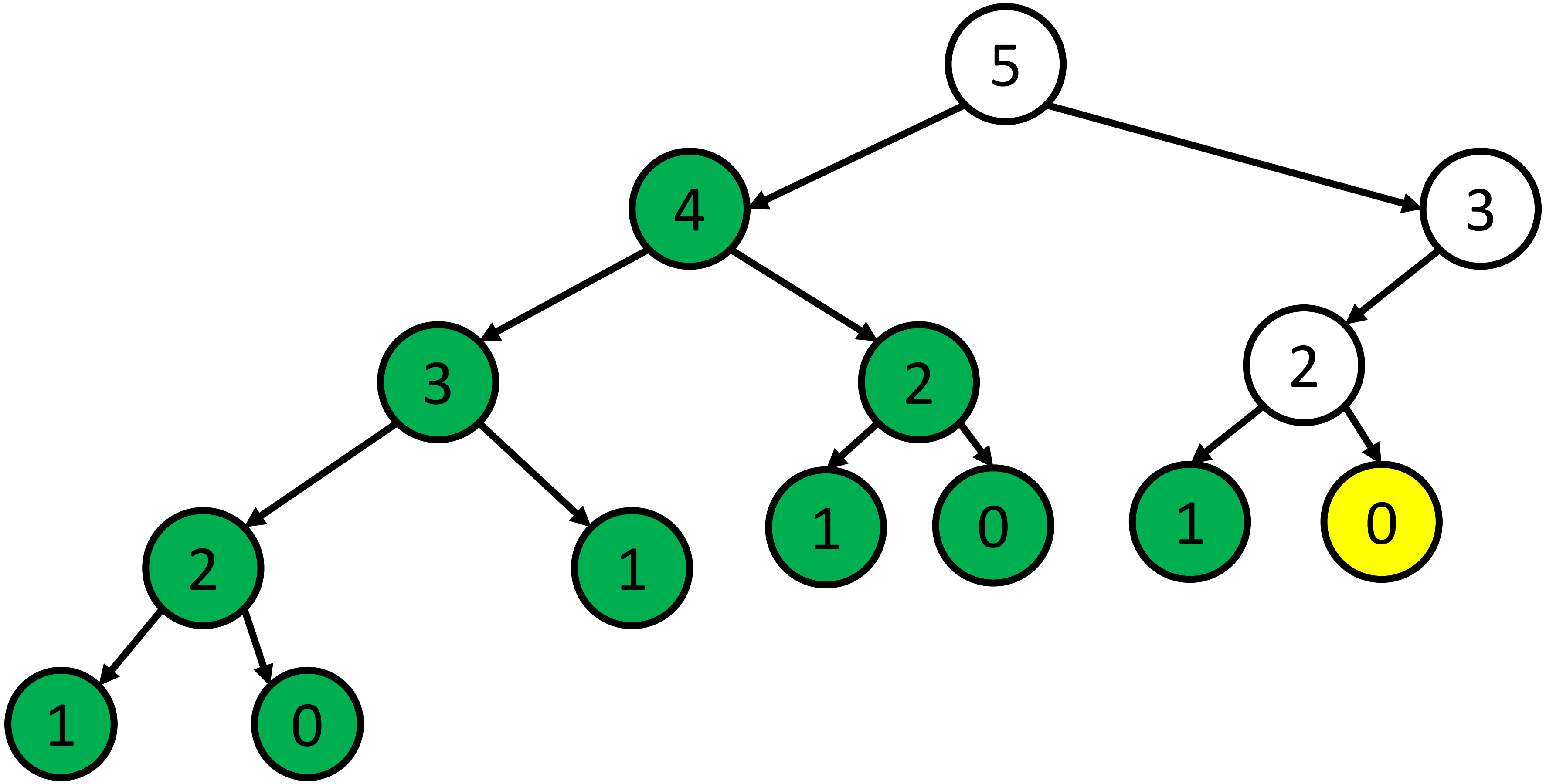
Programación Dinámica - Ejemplo



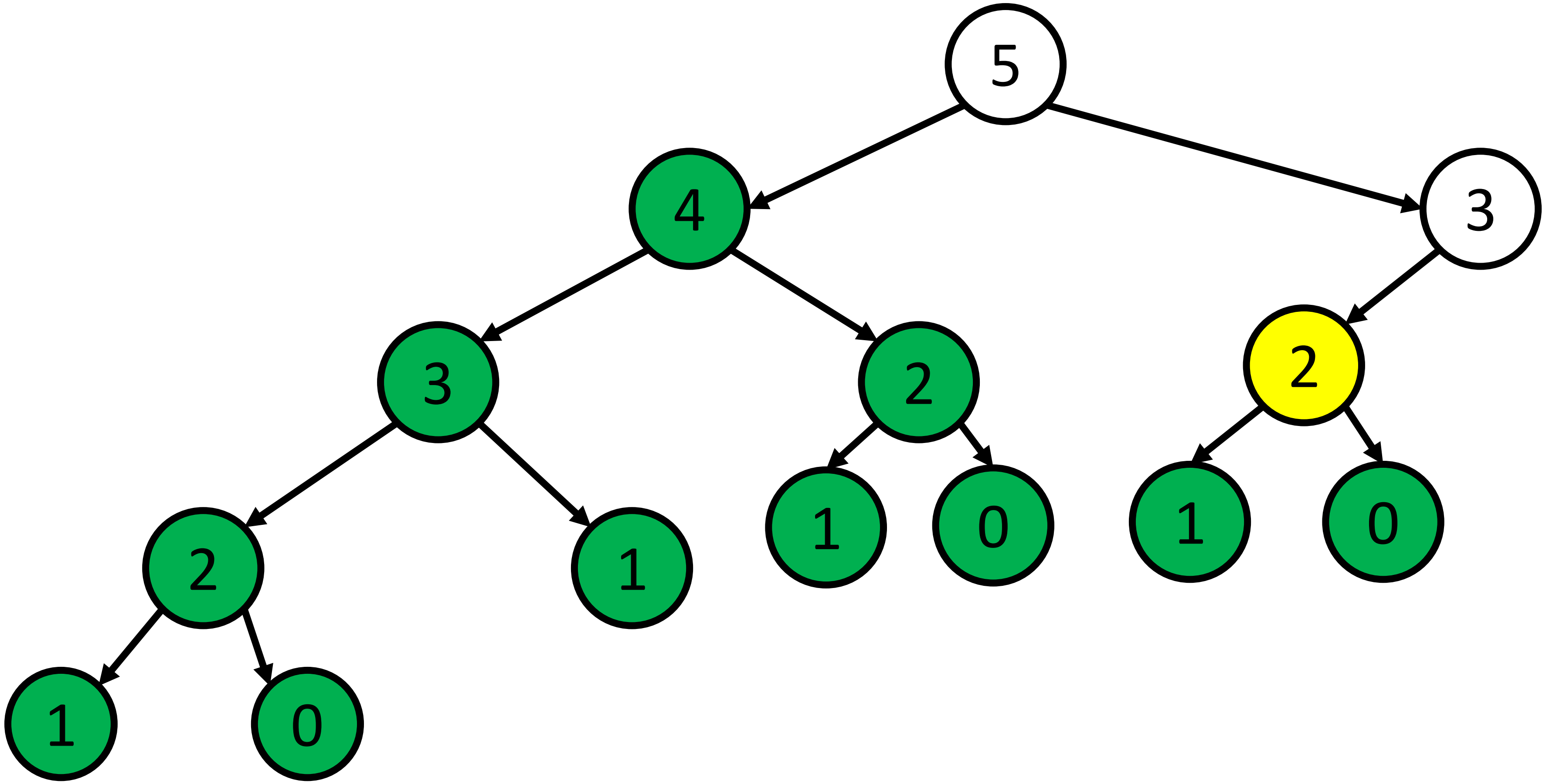
Programación Dinámica - Ejemplo



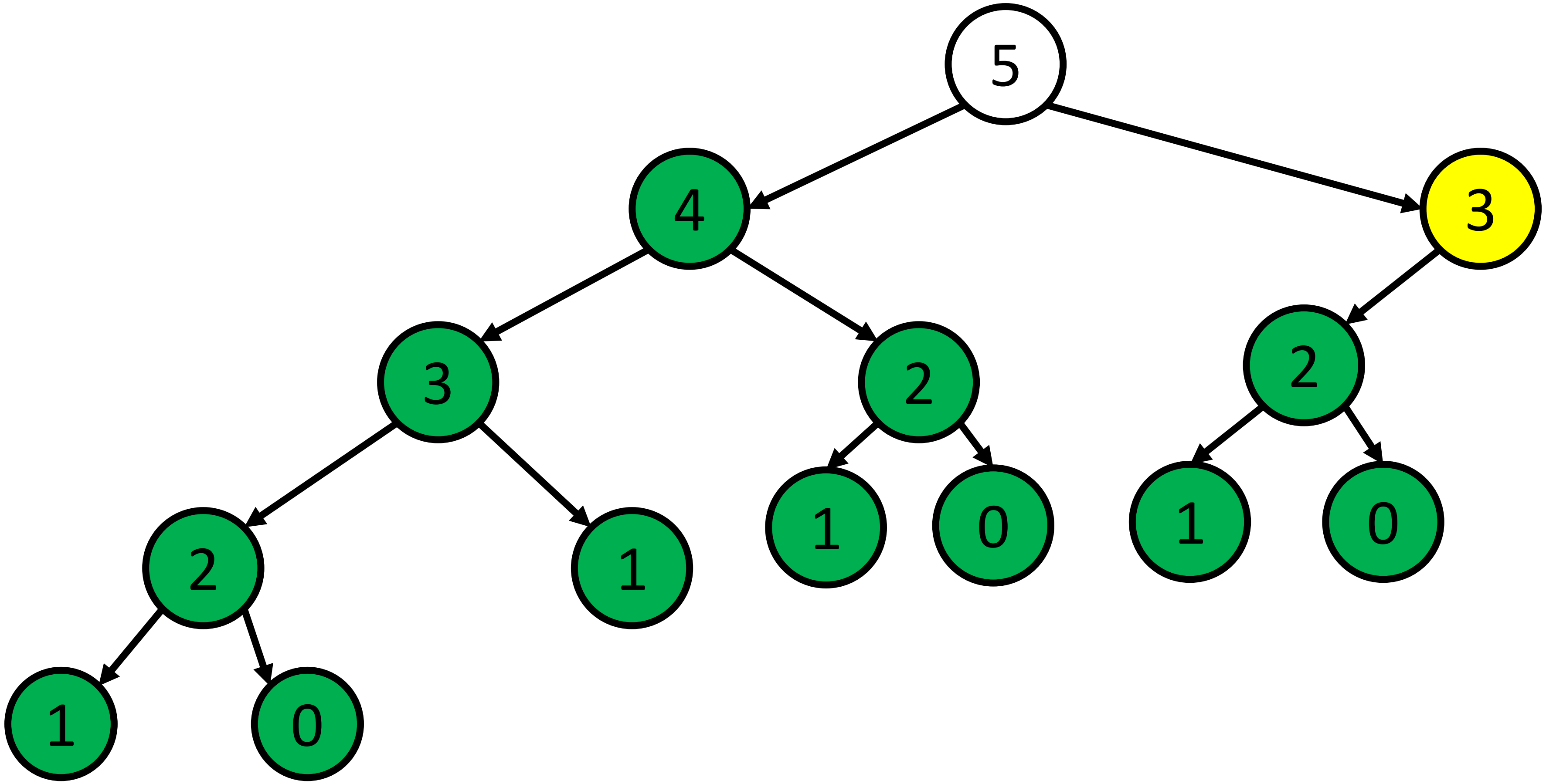
Programación Dinámica - Ejemplo



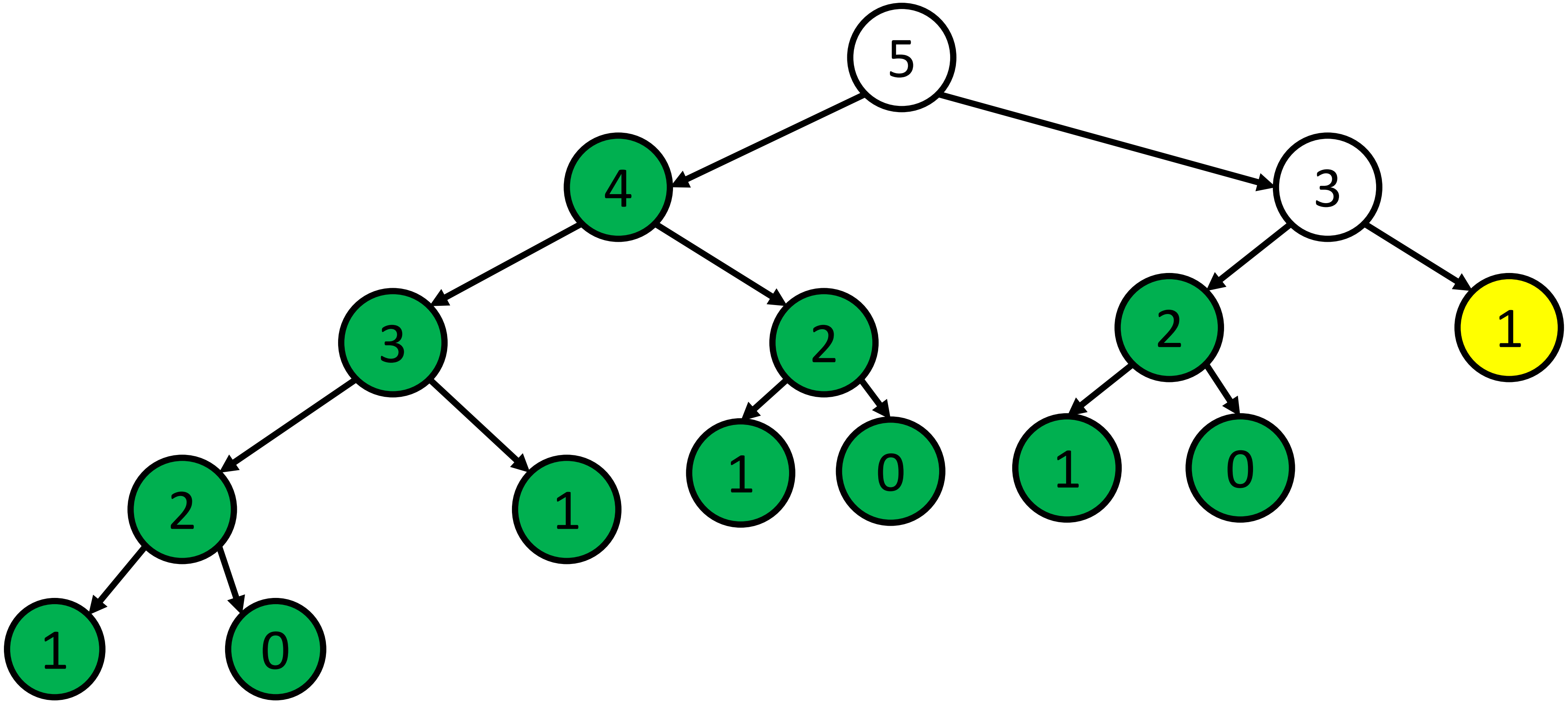
Programación Dinámica - Ejemplo



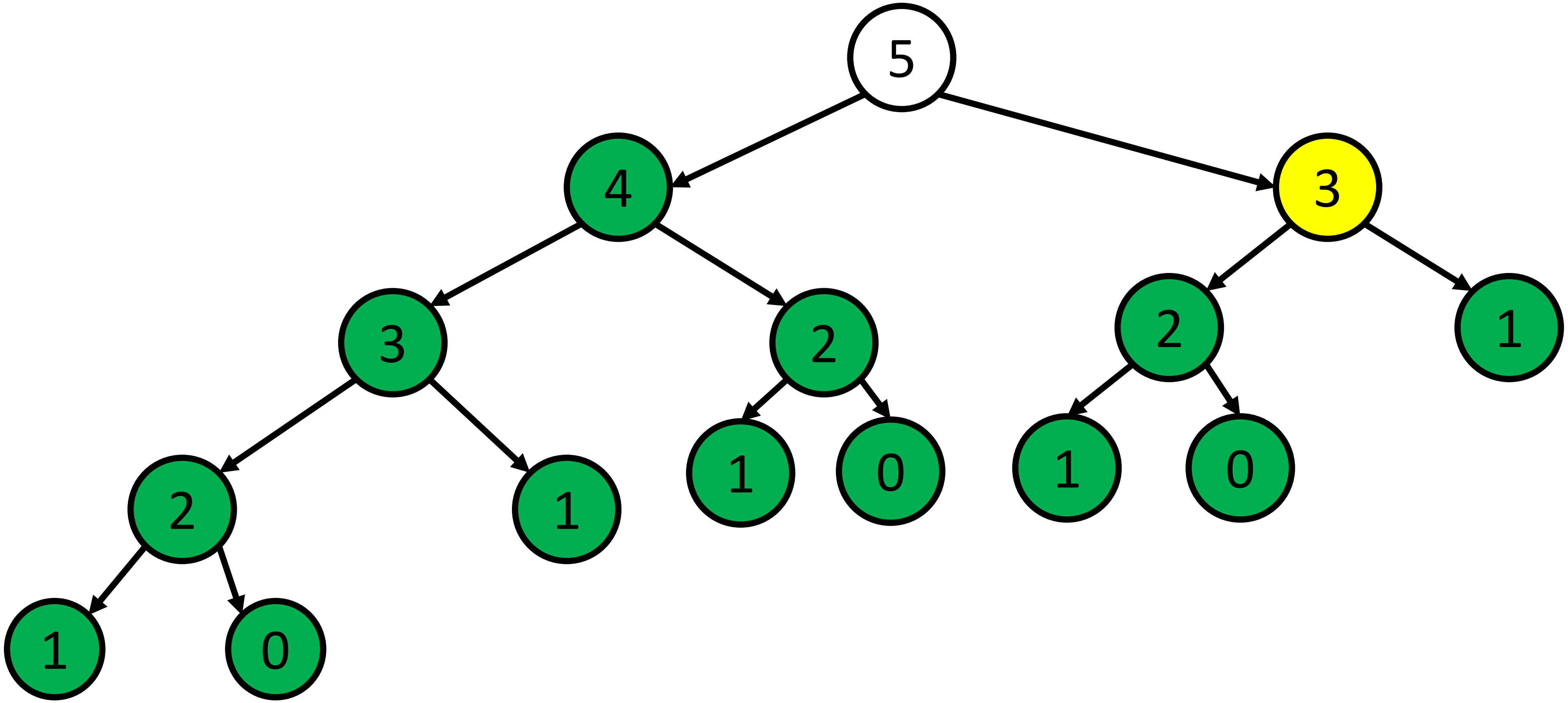
Programación Dinámica - Ejemplo



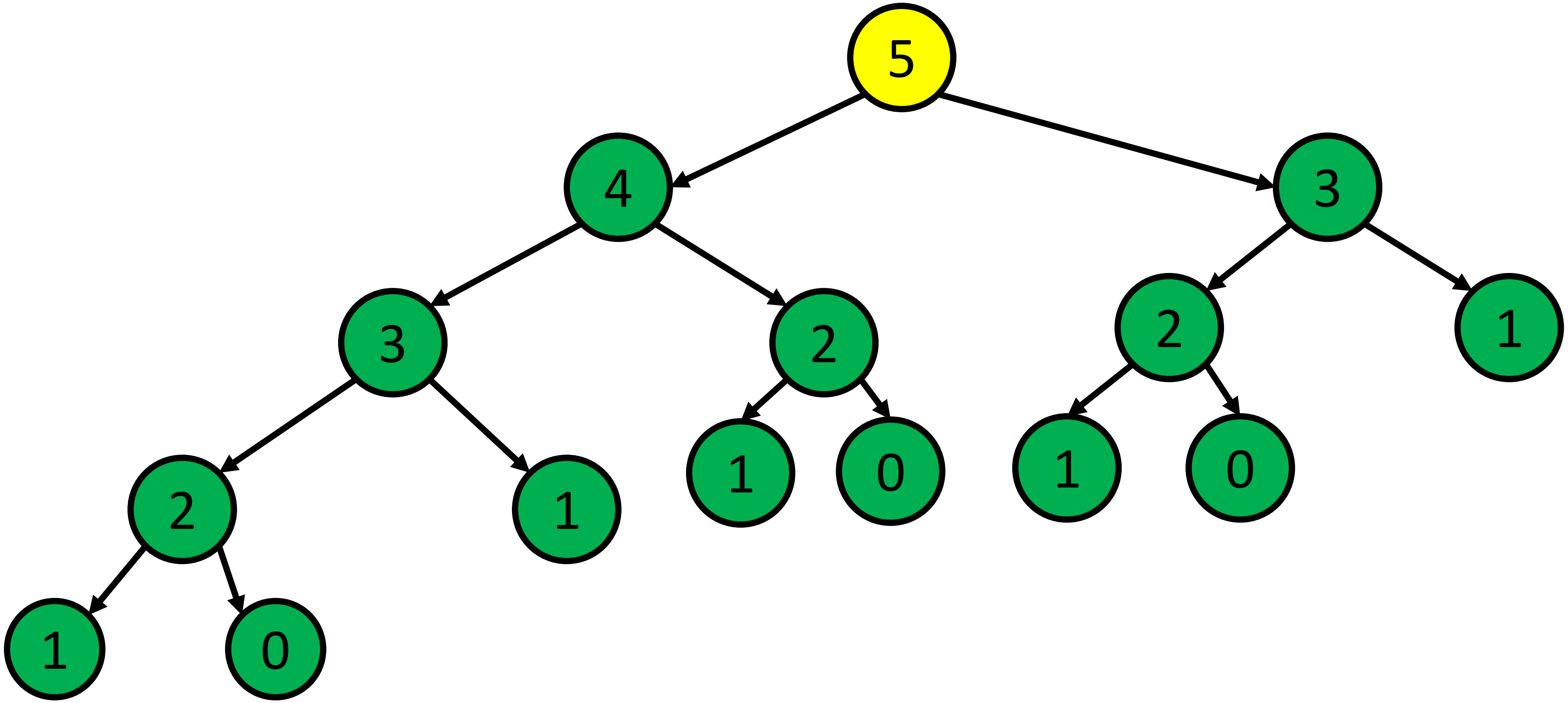
Programación Dinámica - Ejemplo



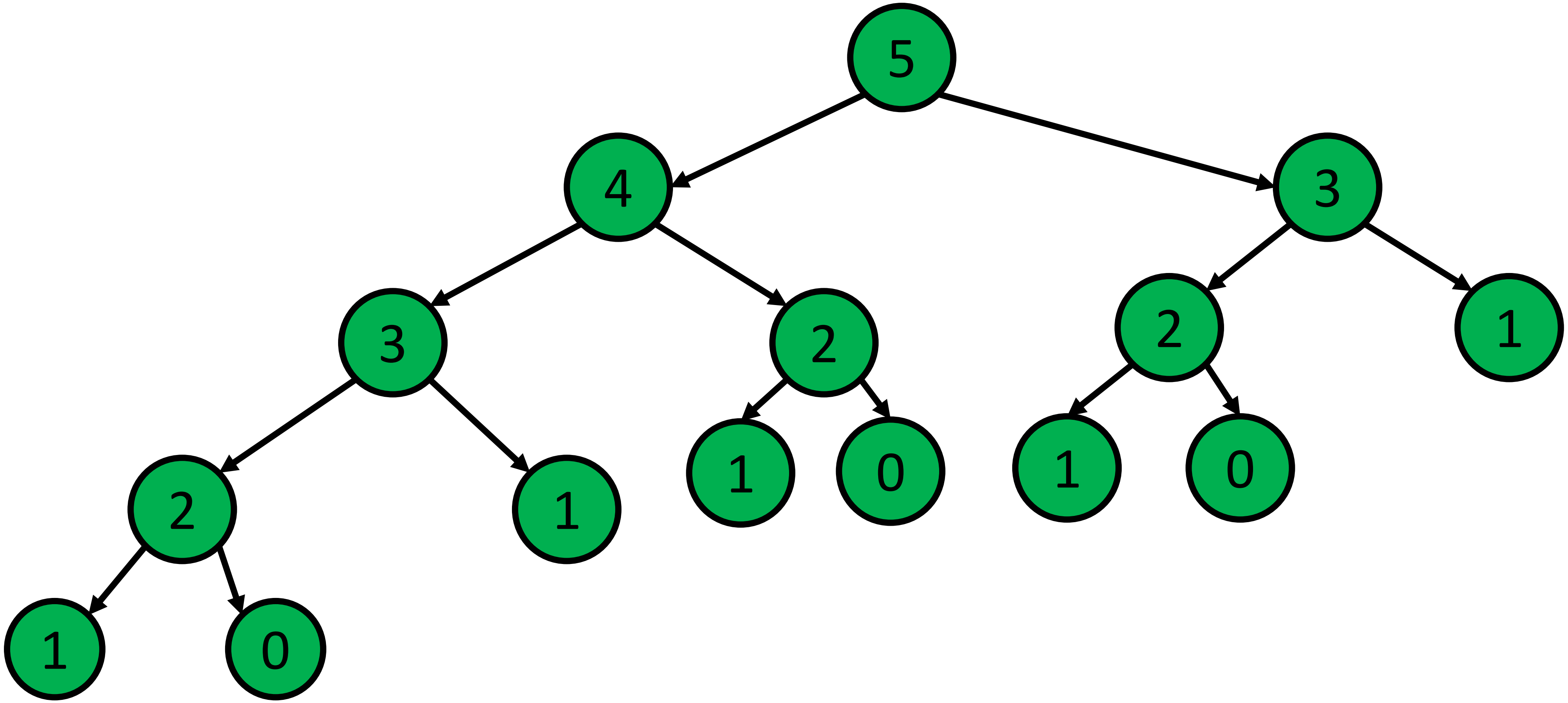
Programación Dinámica - Ejemplo



Programación Dinámica - Ejemplo



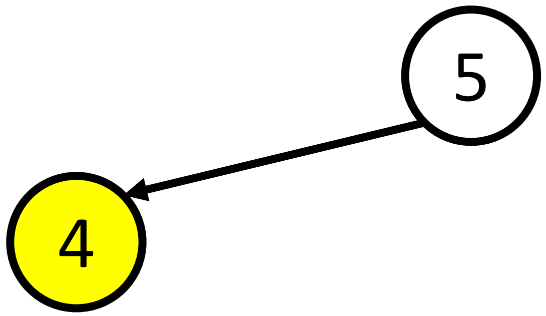
Programación Dinámica - Ejemplo



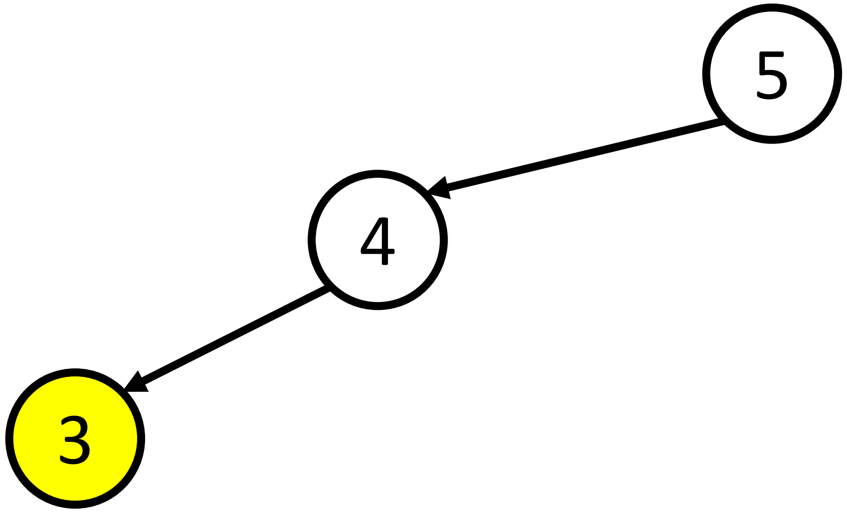
Ahora usemos memoizacion

Programación Dinámica - Ejemplo

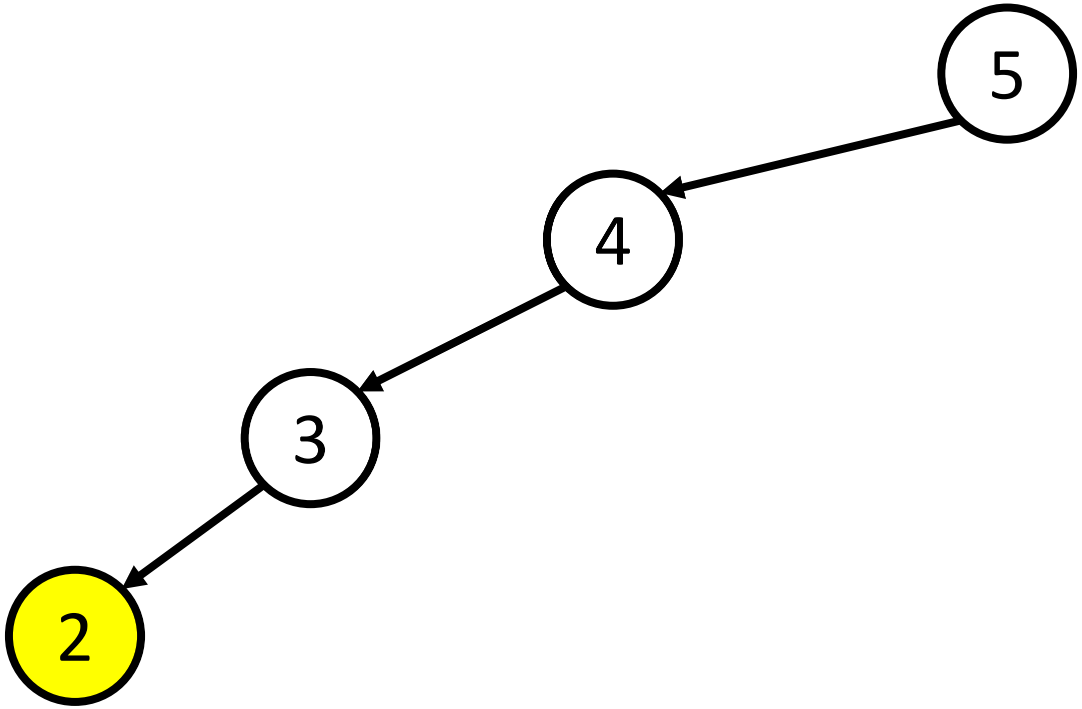
Programación Dinámica - Ejemplo



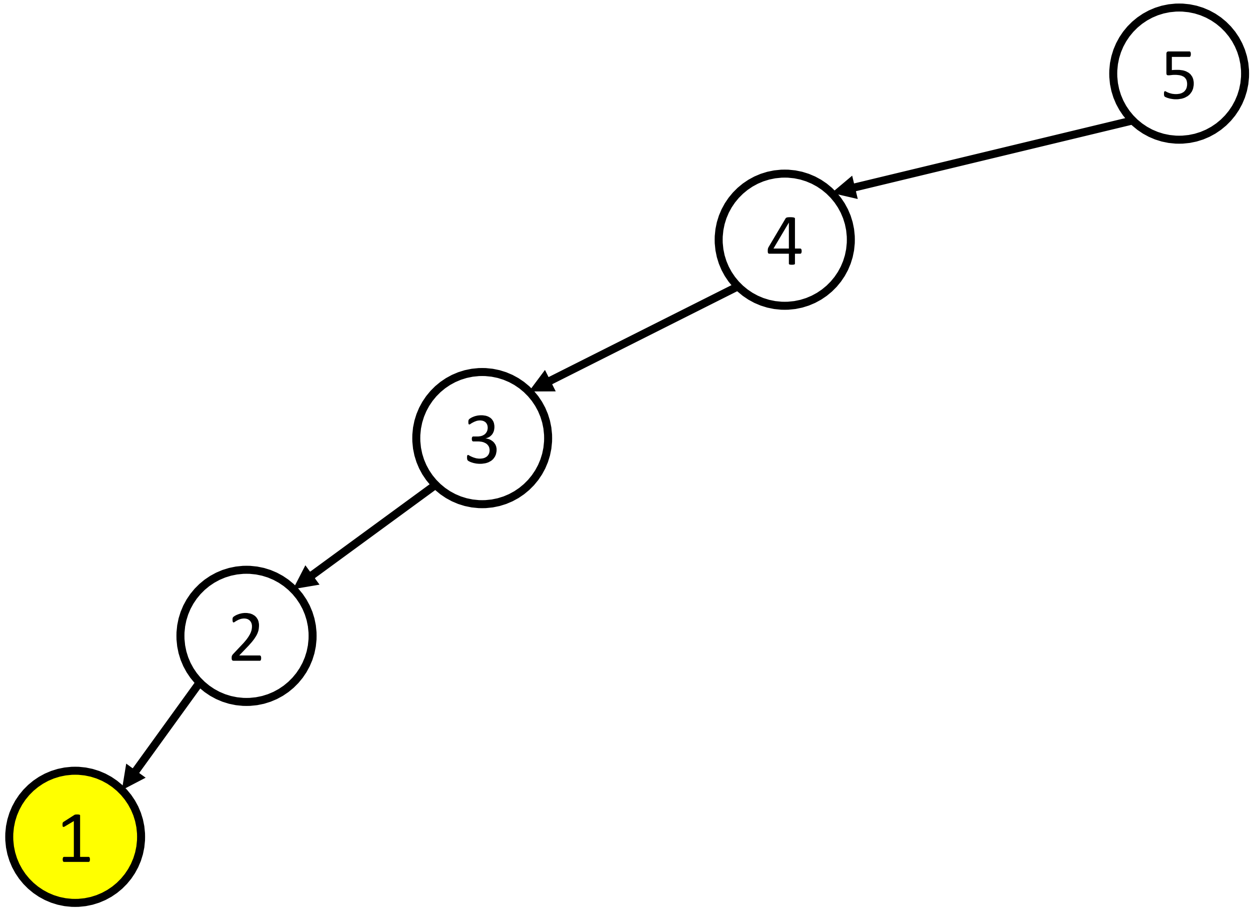
Programación Dinámica - Ejemplo



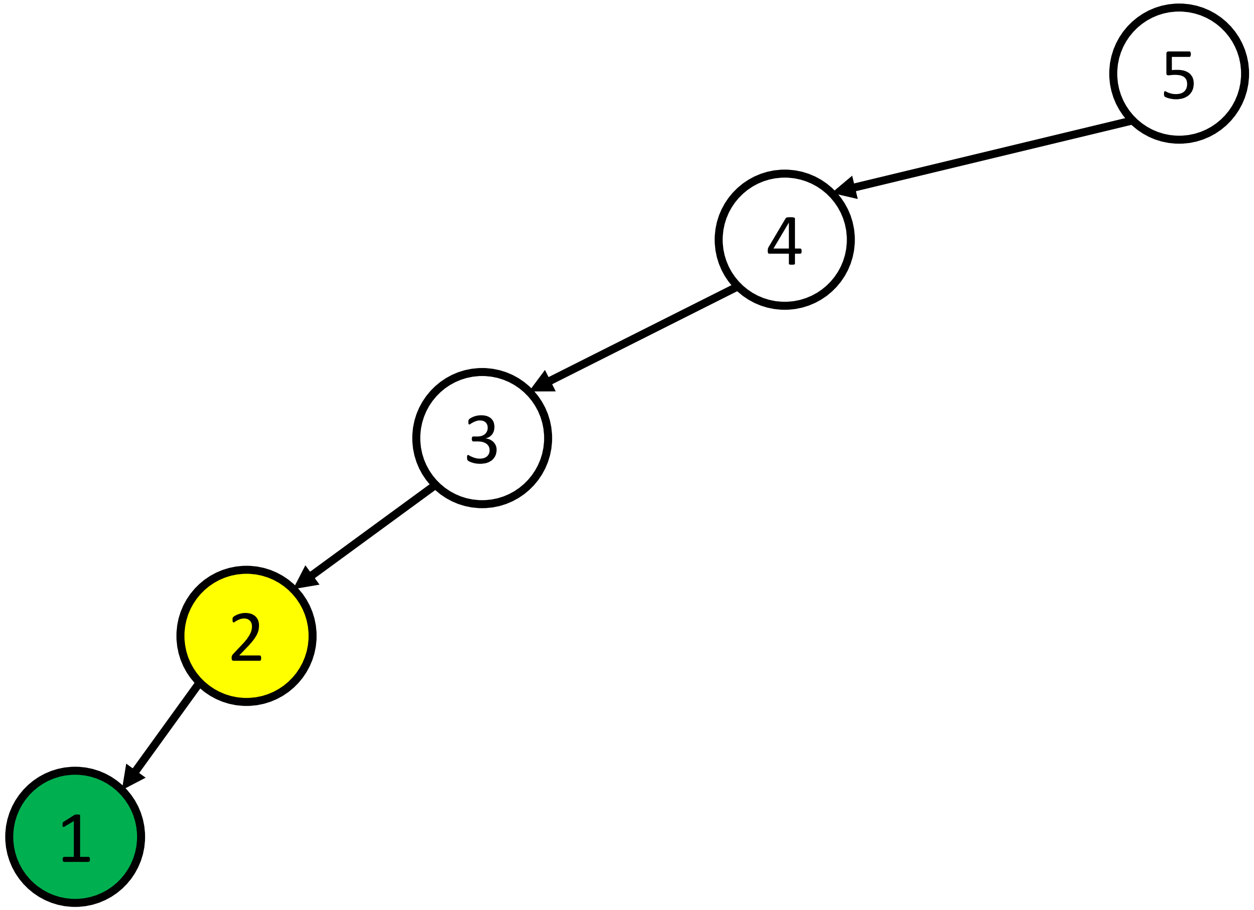
Programación Dinámica - Ejemplo



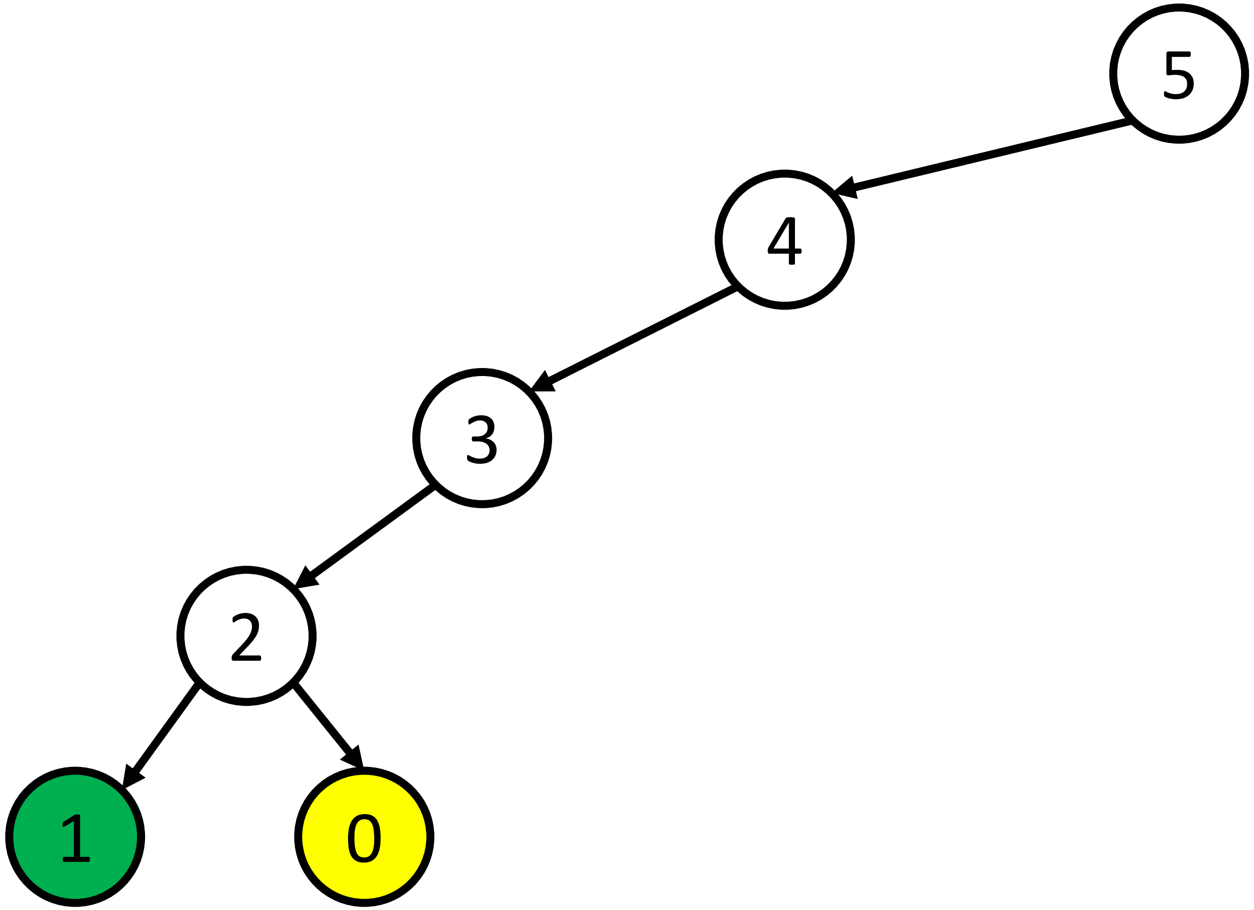
Programación Dinámica - Ejemplo



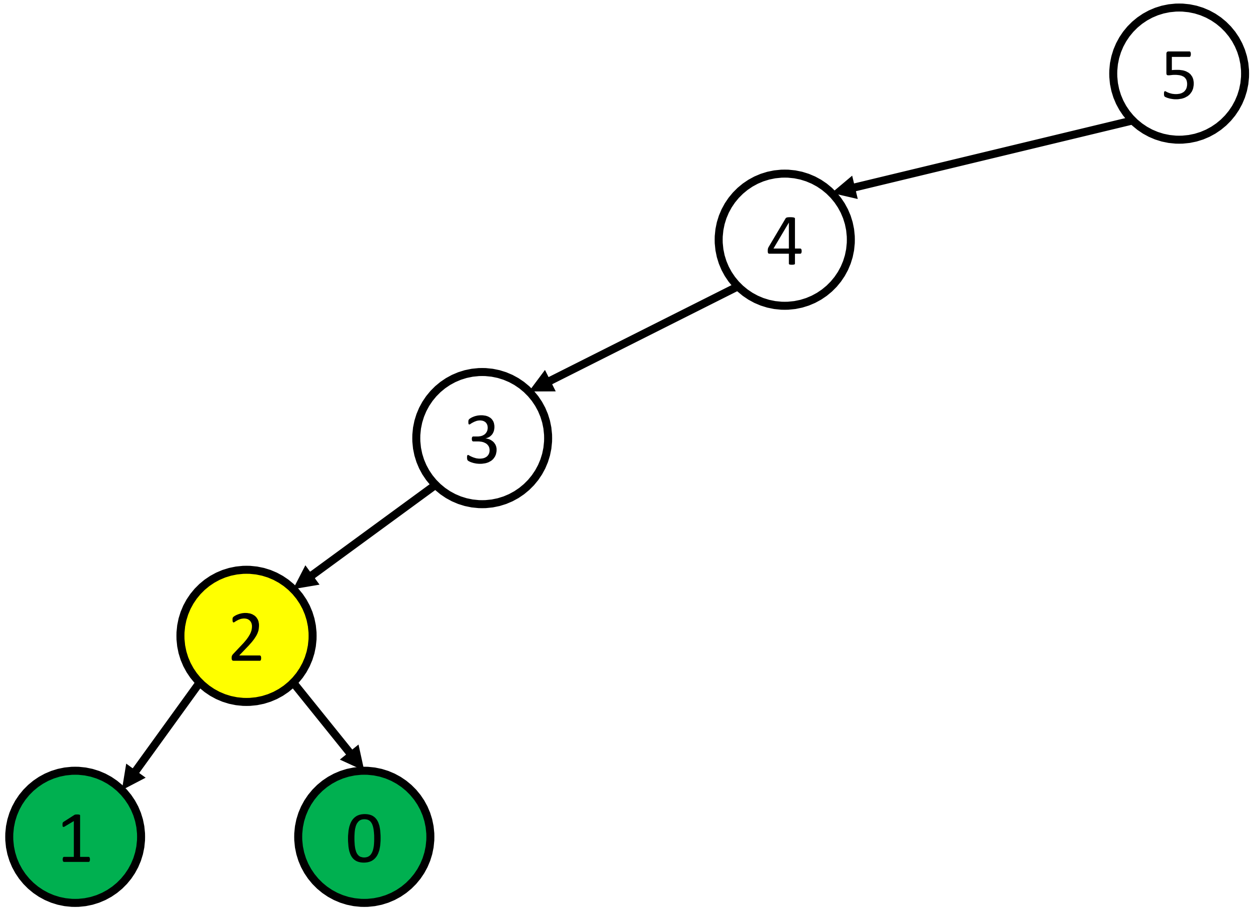
Programación Dinámica - Ejemplo



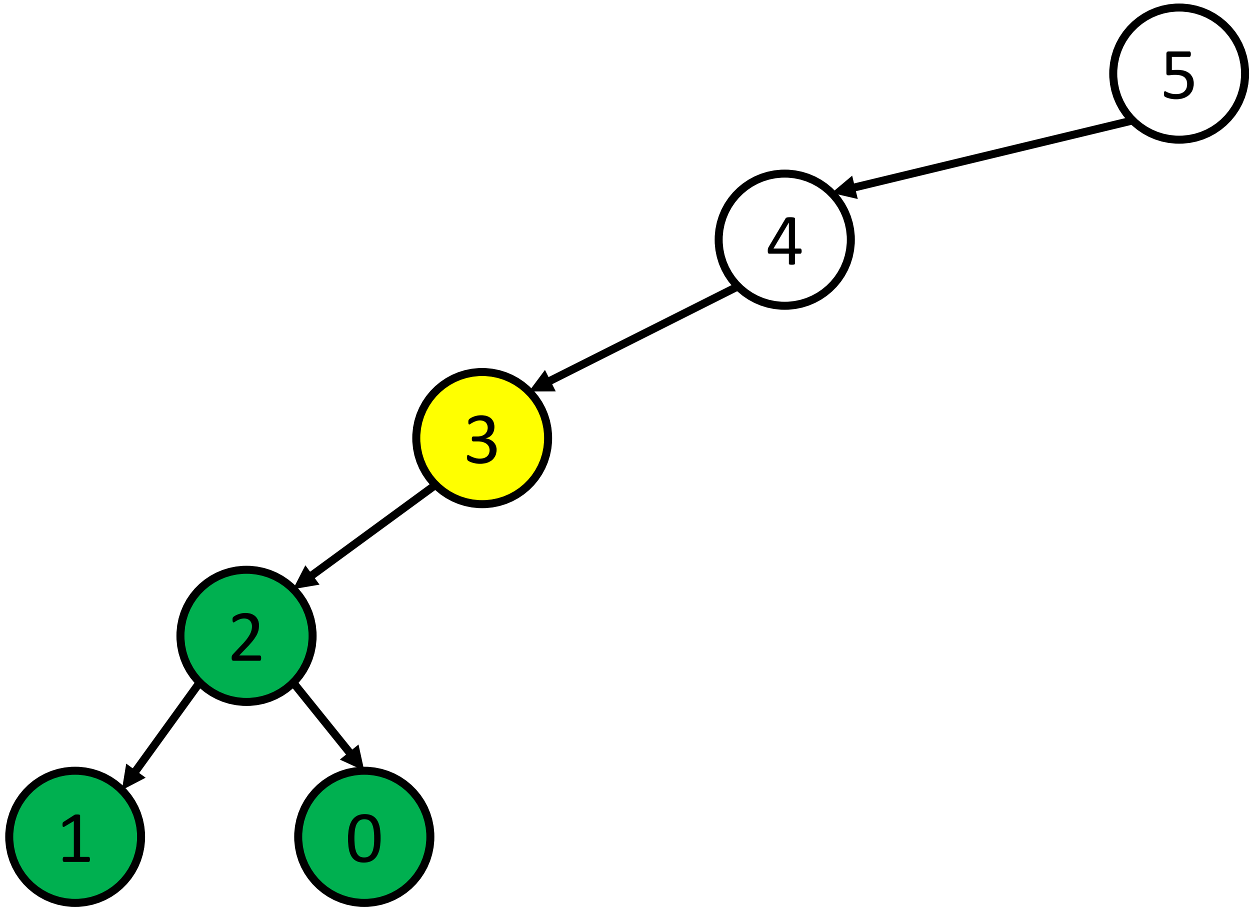
Programación Dinámica - Ejemplo



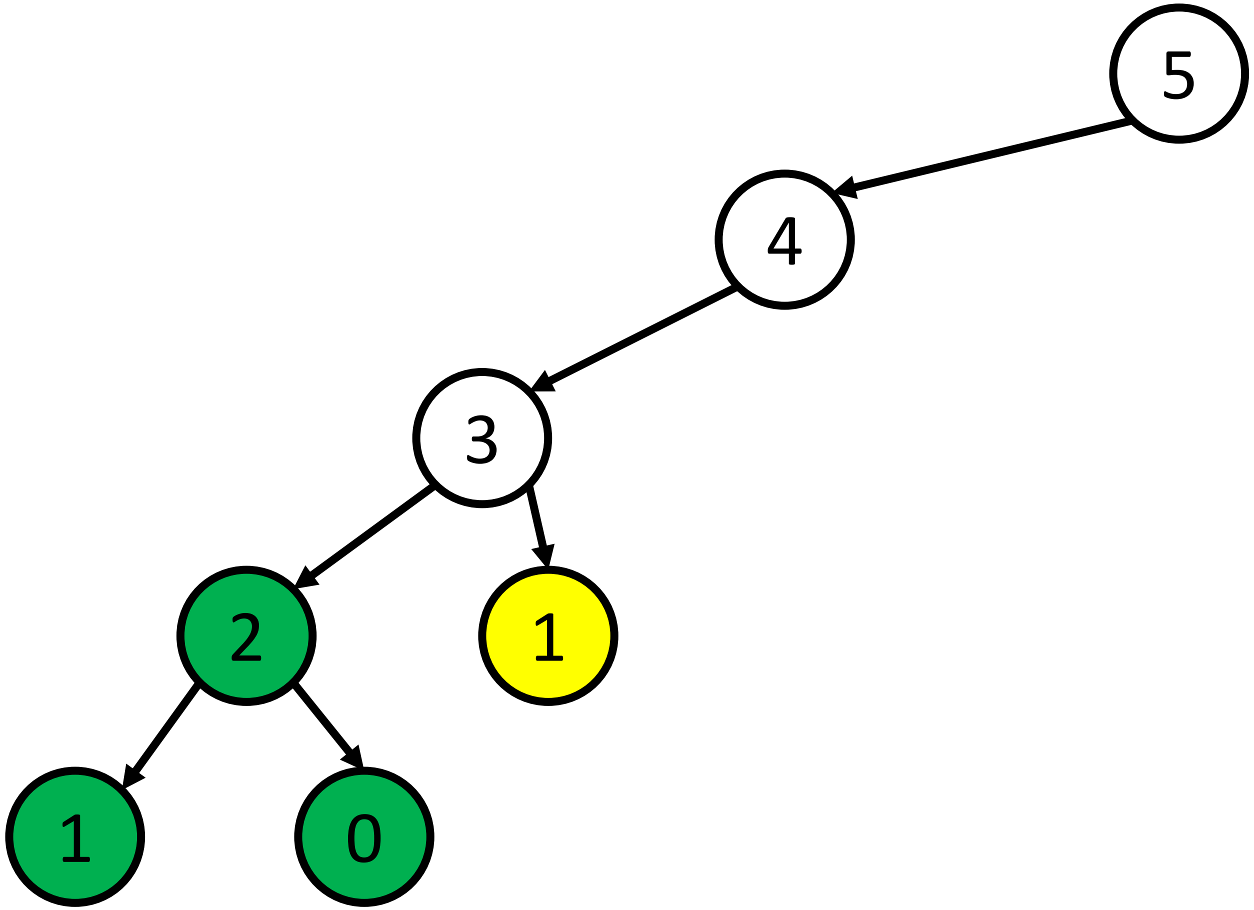
Programación Dinámica - Ejemplo



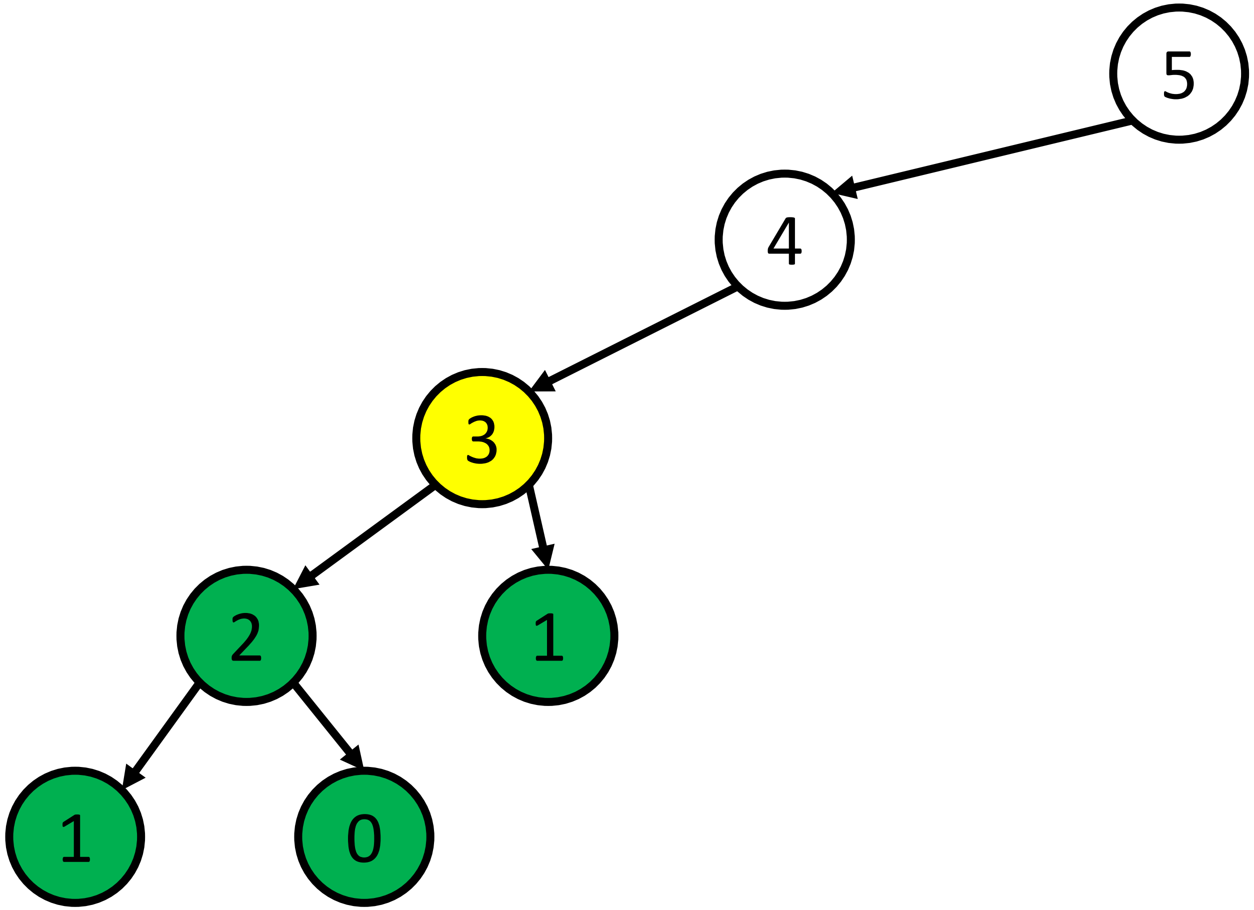
Programación Dinámica - Ejemplo



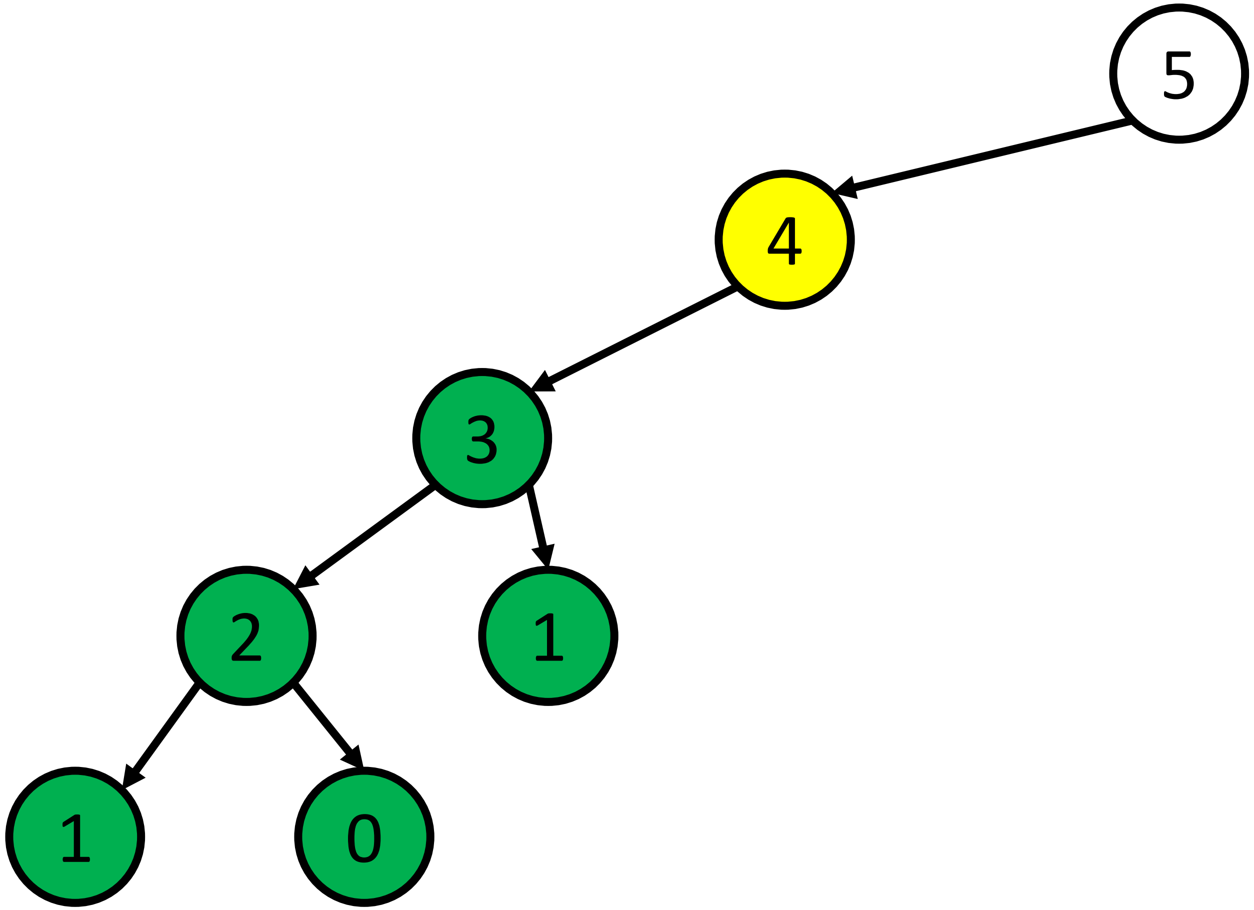
Programación Dinámica - Ejemplo



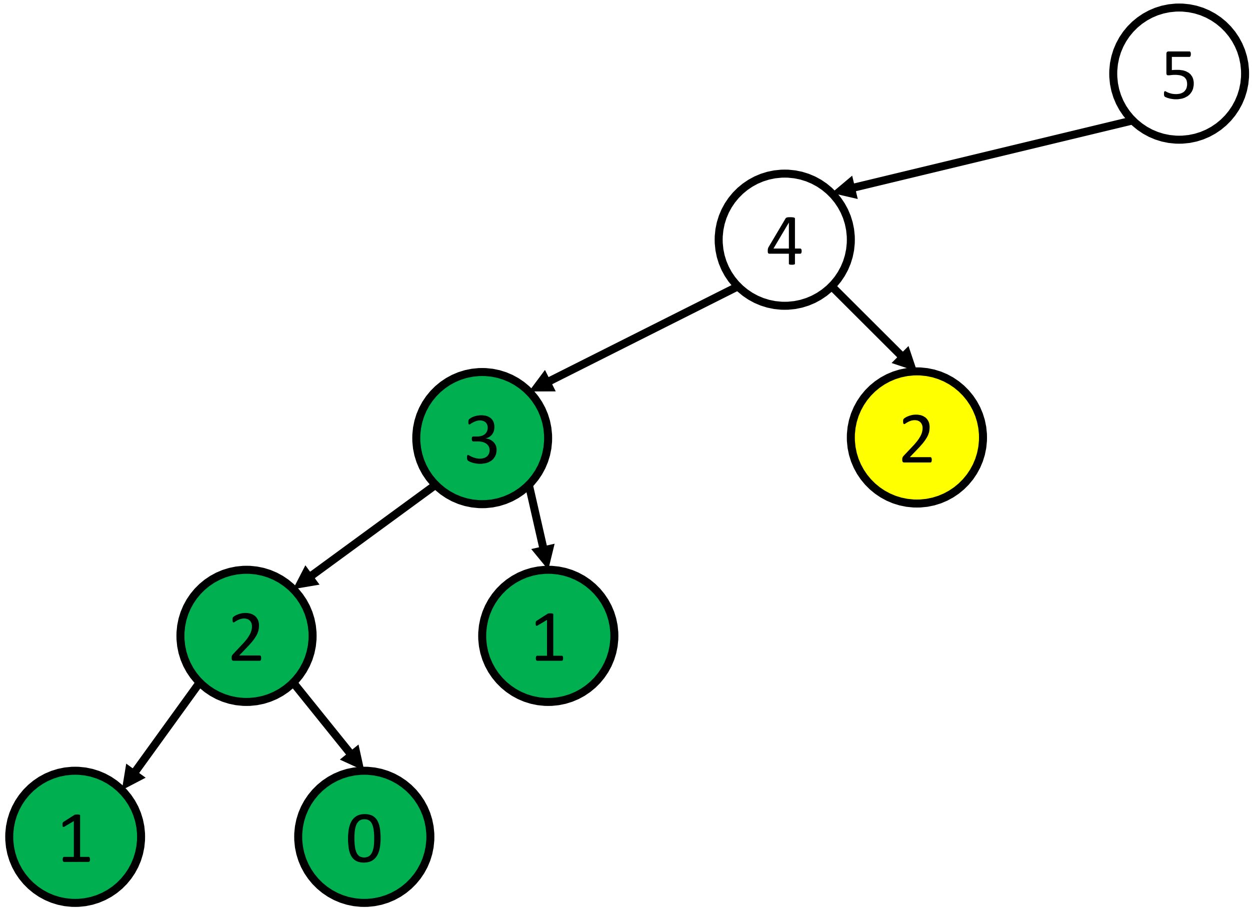
Programación Dinámica - Ejemplo



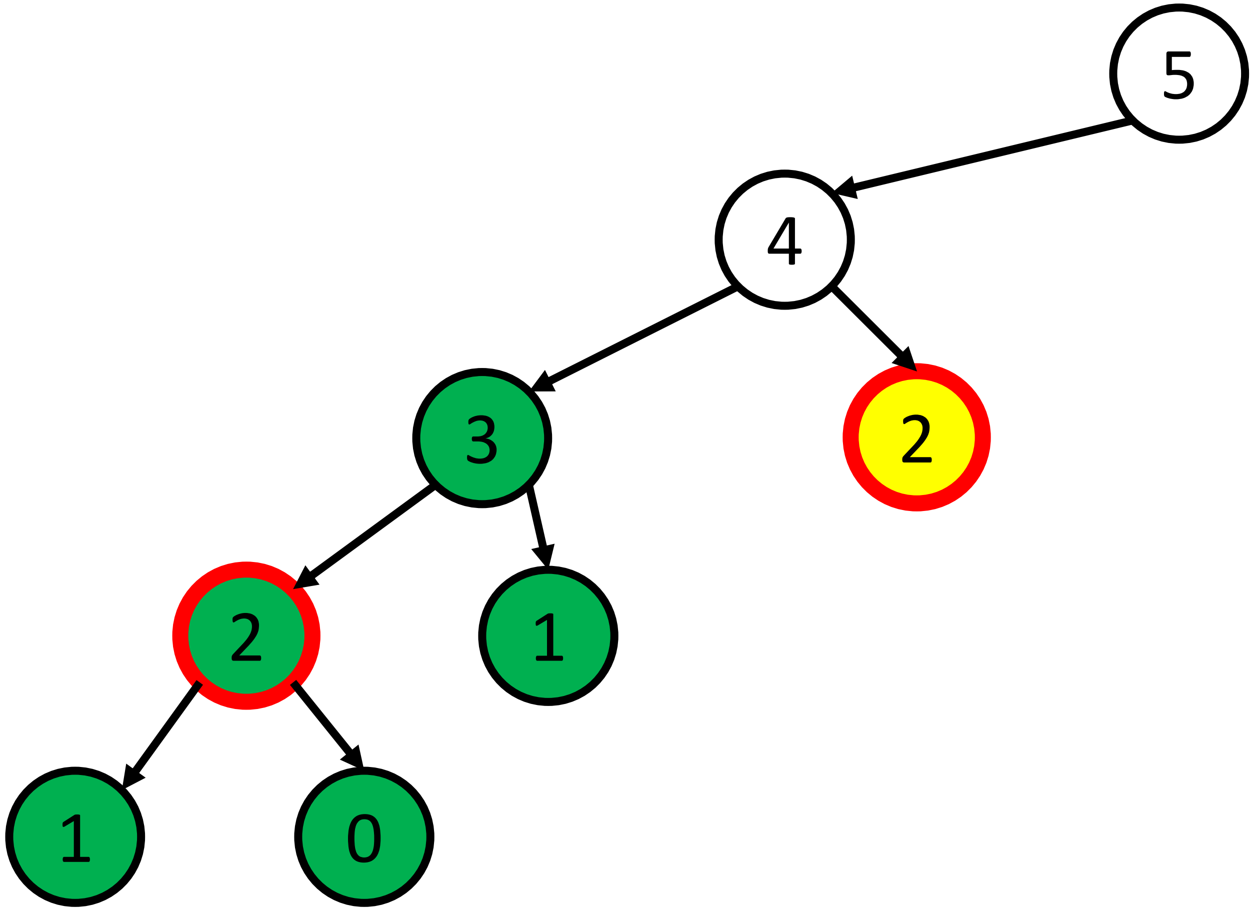
Programación Dinámica - Ejemplo



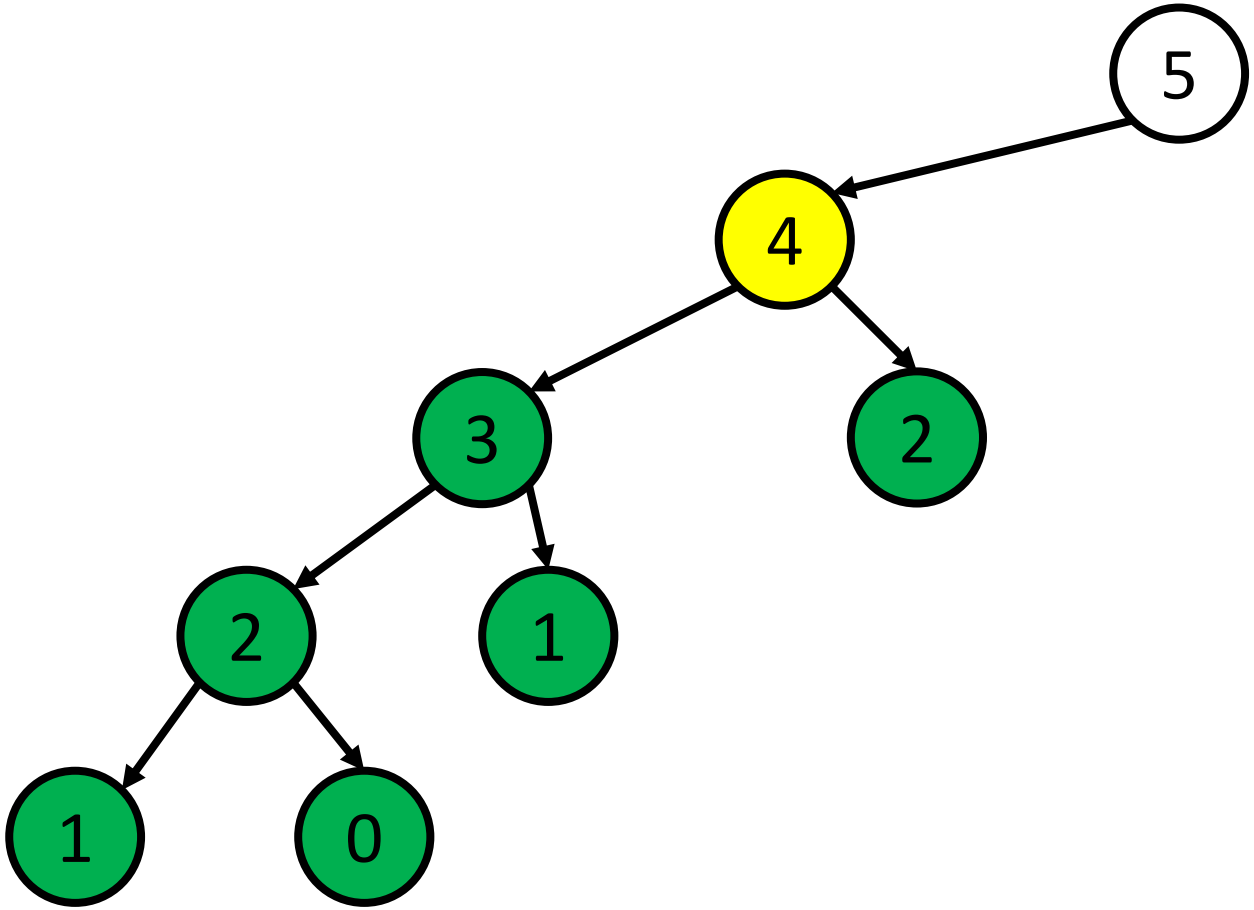
Programación Dinámica - Ejemplo



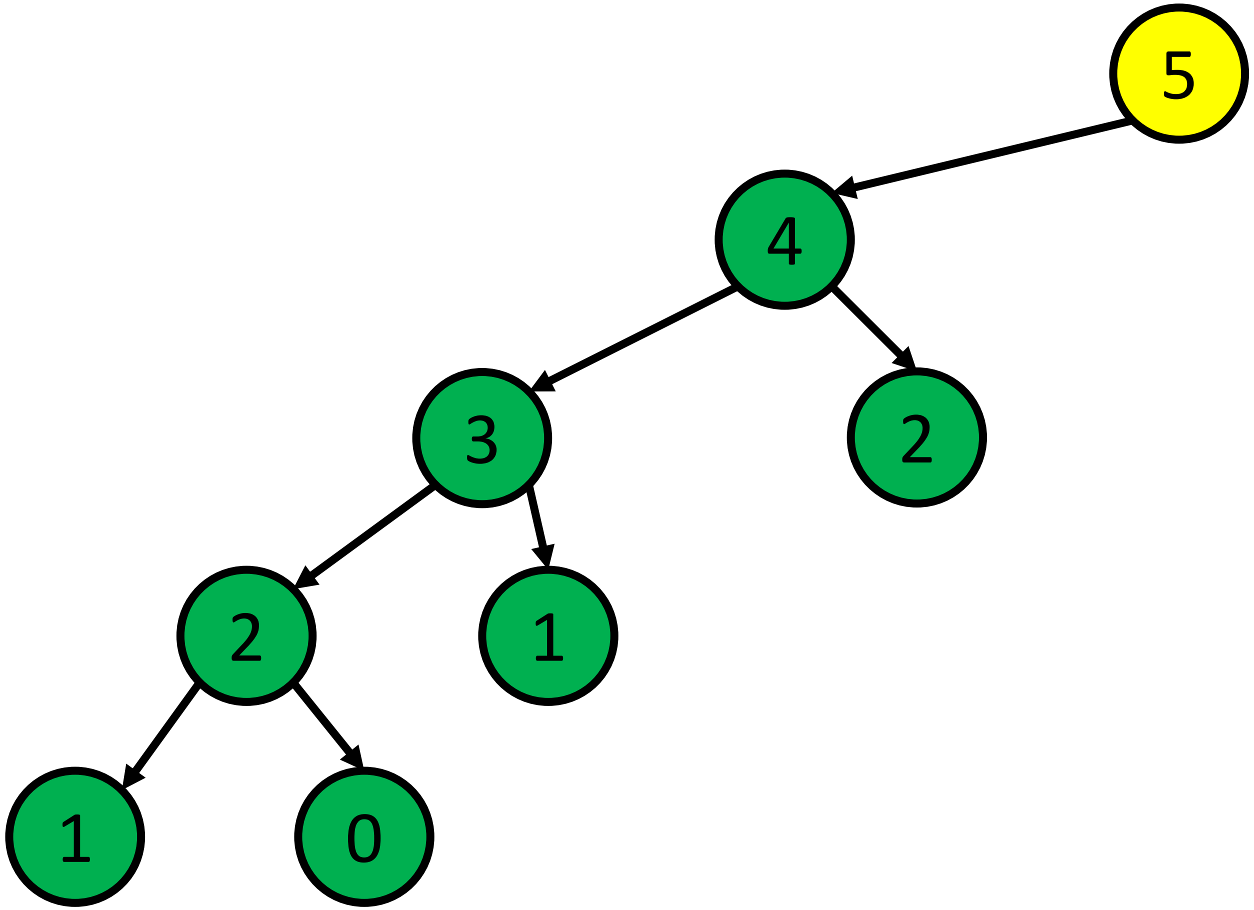
Programación Dinámica - Ejemplo



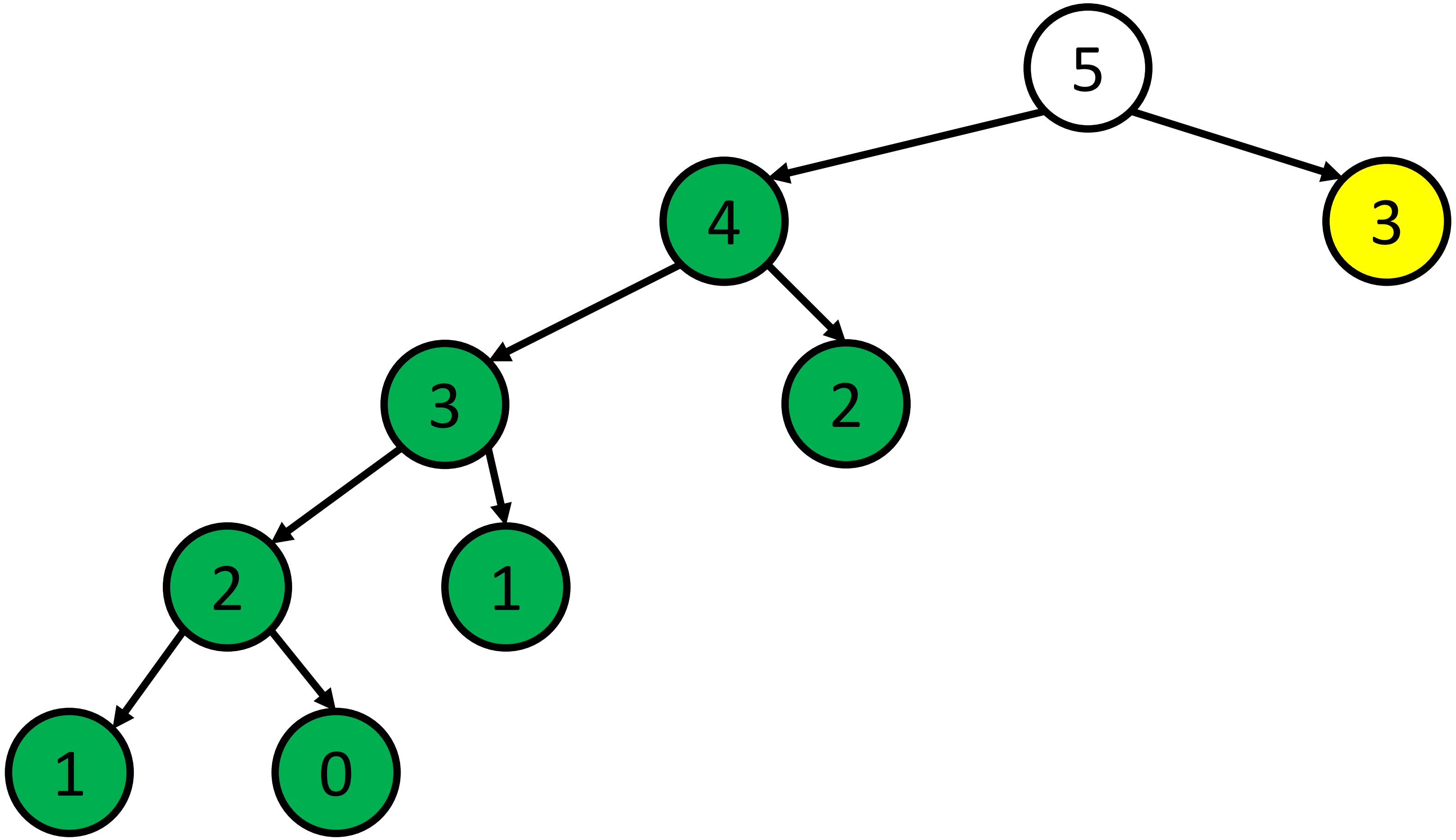
Programación Dinámica - Ejemplo



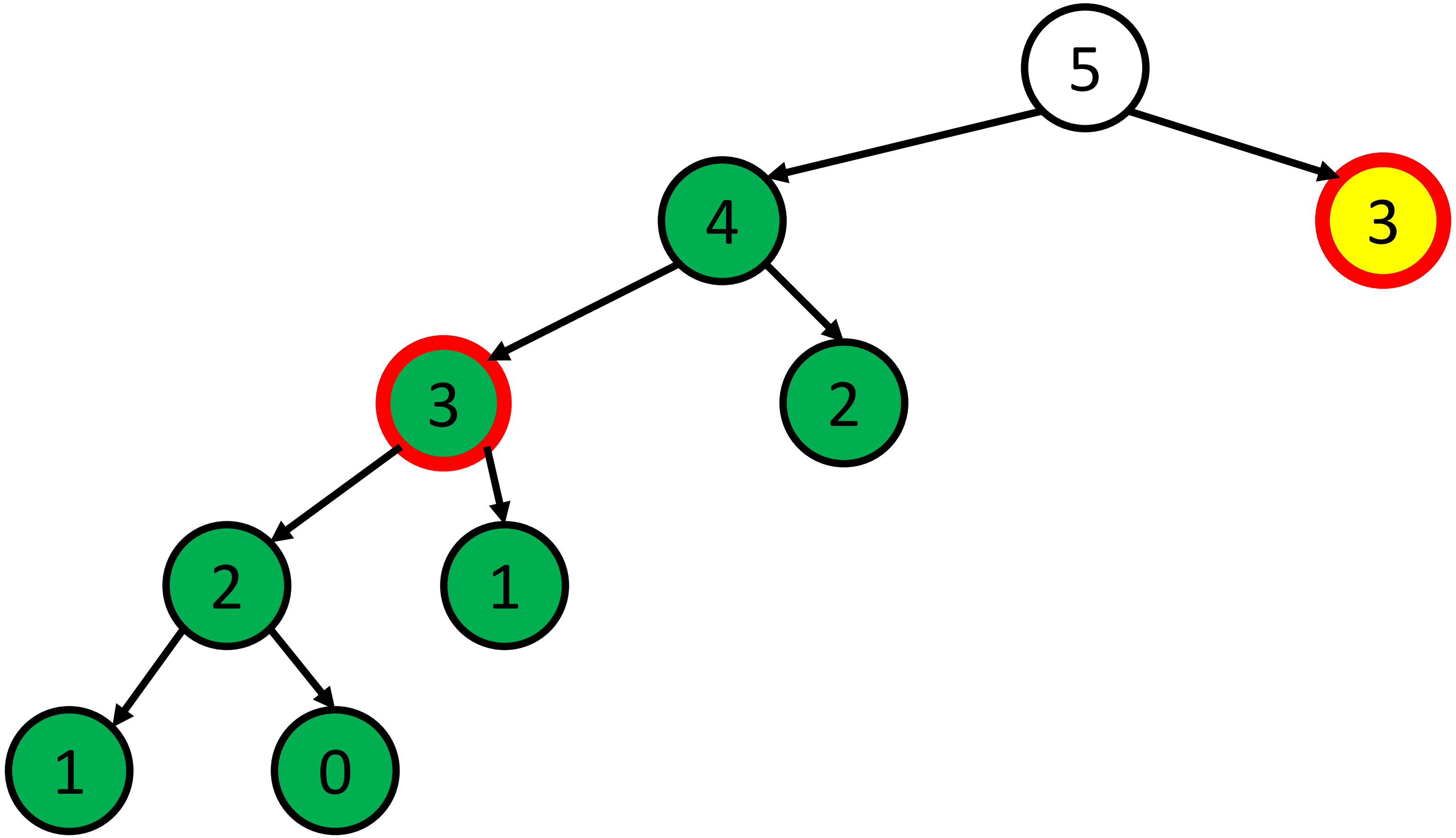
Programación Dinámica - Ejemplo



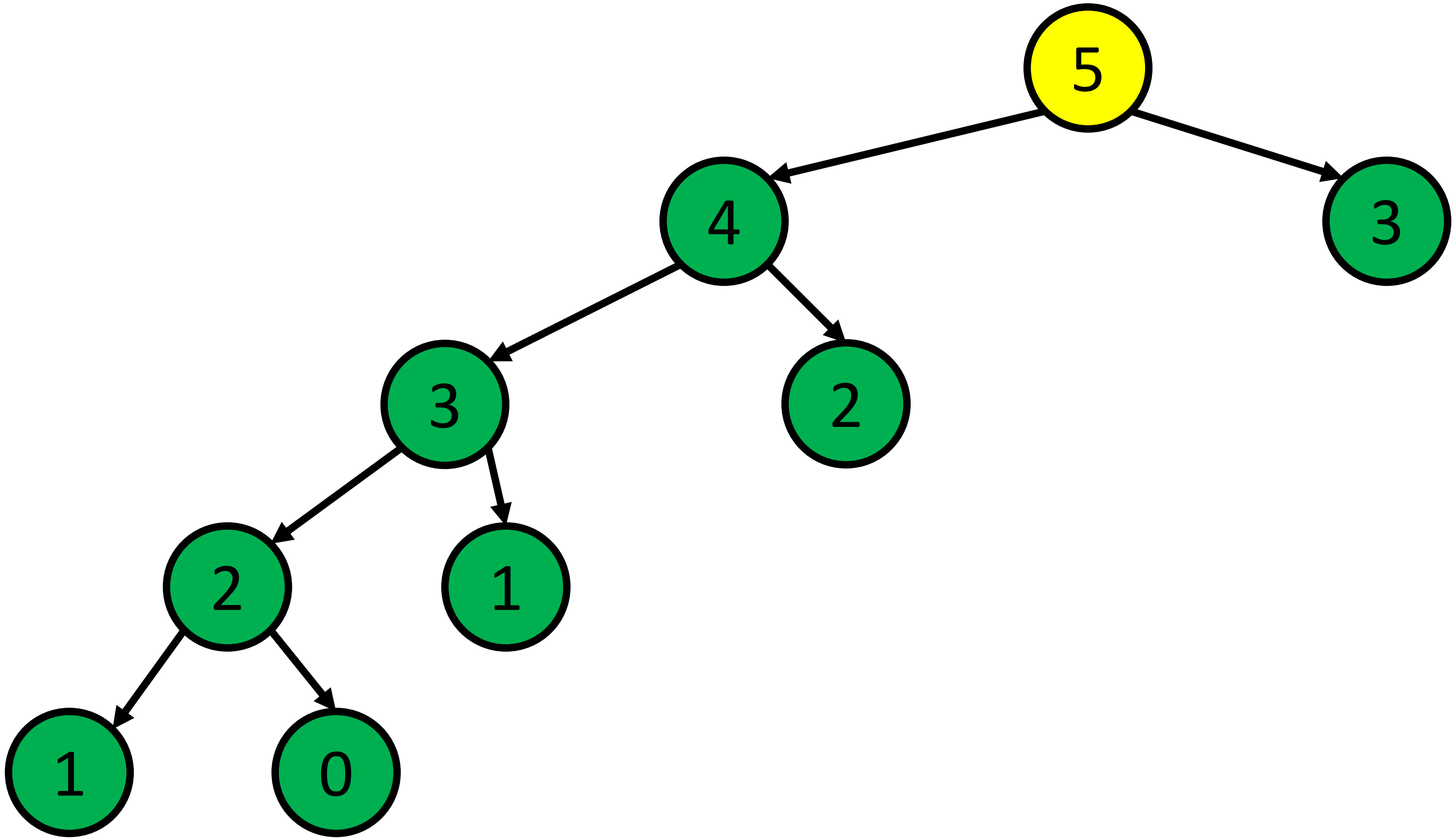
Programación Dinámica - Ejemplo



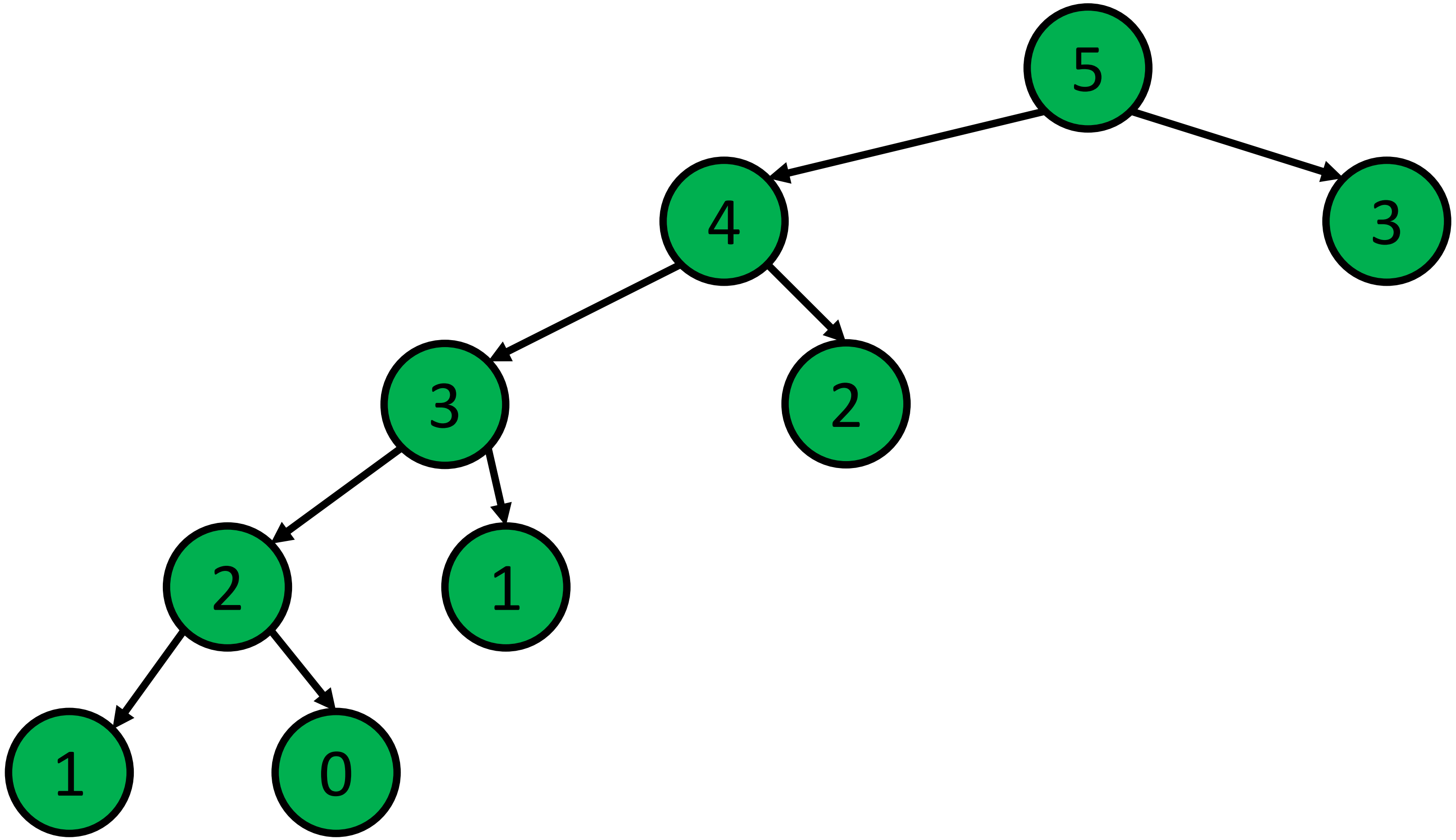
Programación Dinámica - Ejemplo



Programación Dinámica - Ejemplo



Programación Dinámica - Ejemplo



Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - $\sim O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones

Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - $\sim O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones
- Tiempo con memoizacion:
  - Cada estado solo se calcula viendo sus relaciones **una** sola vez

Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - ~$O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones
- Tiempo con memoizacion:
  - Cada estado solo se calcula viendo sus relaciones **una** sola vez
  - La complejidad se calcula: $O(cantidad\_estados \times O(calcular\_estado))$

Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - $\sim O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones
- Tiempo con memoizacion:
  - Cada estado solo se calcula viendo sus relaciones **una** sola vez
  - La complejidad se calcula: $\mathrm{O}(cantidad\_estados \times O(calcular\_estado))$
  - Asumimos que los estados de los que dependemos ya están calculados

Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - ~$O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones
- Tiempo con memoizacion:
  - Cada estado solo se calcula viendo sus relaciones **una** sola vez
  - La complejidad se calcula: $\mathrm{O}(cantidad\_estados \times O(calcular\_estado))$
  - Asumimos que los estados de los que dependemos ya están calculados
  - Para Fibonacci la relación es solo una suma $F_i = F_{i-1} + F_{i-2}$

Programación Dinámica - Ejemplo

- Tiempo sin memoizacion:
  - ~$O(\varphi^n) \approx O(1.6^n)$
  - Para n = 50 realiza ~16,069,380,442 operaciones
- Tiempo con memoizacion:
  - Cada estado solo se calcula viendo sus relaciones **una** sola vez
  - La complejidad se calcula: $\mathrm{O}(cantidad\_estados \times O(calcular\_estado))$
  - Asumimos que los estados de los que dependemos ya están calculados
  - Para Fibonacci la relación es solo una suma $F_i = F_{i-1} + F_{i-2}$
  - $O\big(n \times O(1)\big) = O(n)$

Programación Dinámica - Ejemplo

- Guardar y re-utilizar las soluciones de los subproblemas

Programación Dinámica - Ejemplo

- Guardar y re-utilizar las soluciones de los subproblemas
- Mantener las soluciones en alguna estructura de datos (usualmente un arreglo o matriz, a veces mapas)

Programación Dinámica - Ejemplo

- Guardar y re-utilizar las soluciones de los subproblemas
- Mantener las soluciones en alguna estructura de datos (usualmente un arreglo o matriz, a veces mapas)
- La funcion recursiva debe retornar la solucion del subproblema si ya fue resuelto, si no, debe calcular la respuesta y guardarla

Programación Dinámica - Ejemplo

```
int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;

  int res = fib(n - 1) + fib(n - 2);

  return res;
}
```

Programación Dinámica - Ejemplo

```
const int maxN = 1e5 + 5;
vector<int> memo(maxN, -1);

int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;

  int res = fib(n - 1) + fib(n - 2);

  return res;
}
```

Programación Dinámica - Ejemplo

```cpp
const int maxN = 1e5 + 5;
vector<int> memo(maxN, -1);

int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  if (memo[n] != -1) return memo[n];
  int res = fib(n - 1) + fib(n - 2);
  memo[n] = res;
  return res;
}
```

Forma general

```
vector<int> memo(maxN, -1);

int f(int estado) {
  if (estado == caso_base) return base;
  if (memo[estado] != -1) return memo[estado];
  int res = 0;
  for (int sub : relaciones[estado]) {
    res = combinar(res, f(sub));
  }
  memo[estado] = res;
  return res;
}
```

Problema – Rana

Hay N piedras numeradas $1, 2, \dots, N$. Cada piedra i tiene una altura h_i.

Una rana que inicialmente esta en la piedra 1 quiere llegar a la piedra N dando saltos de la siguiente forma:

- Si la rana esta en la piedra i, puede saltar a la piedra $i + 1$ o $i + 2$. El costo de dar un salto es de $|h_i - h_j|$, donde j es la piedra en la que cae al dar el salto.

Encontrar el costo mínimo para que la rana llegue a la piedra N.

Problema – Rana

- Subproblemas:
  - Observamos que, para que la rana llegue a la piedra N, debe llegar a las piedras $1 \leq i < N$ y luego llegar a N con otros saltos
  - $costo(i)$ para $i \leq N$

Problema – Rana

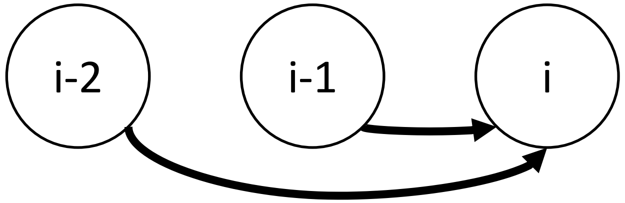
- Subproblemas:
  - Observamos que, para que la rana llegue a la piedra N, debe llegar a las piedras $1 \leq i < N$ y luego llegar a N con otros saltos
  - $costo(i)$ para $i \leq N$
- Relaciones:
  - Para llegar a la piedra i, debemos ver el menor costo entre saltar de la piedra $i - 1$ y la piedra $i - 2$

Problema – Rana

- Subproblemas:
  - Observamos que, para que la rana llegue a la piedra N, debe llegar a las piedras $1 \leq i < N$ y luego llegar a N con otros saltos
  - $costo(i)$ para $i \leq N$
- Relaciones:
  - Para llegar a la piedra i, debemos ver el menor costo entre saltar de la piedra $i - 1$ y la piedra $i - 2$
  - $costo(i) = \min(costo(i-1) + |h_{i-1} - h_i|, costo(i-2) + |h_{i-2} - h_i|)$

Problema – Rana

- Orden Topológico:
  - Para calcular $costo(i)$, necesitamos habar calculado $costo(i-1)$ y $costo(i-2)$



  - No se forman ciclos

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana
- Problema Original:
  - $costo(N)$

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana
- Problema Original:
  - $costo(N)$
- Tiempo:
  - Hay N subproblemas en total (uno por cada piedra)

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana
- Problema Original:
  - $costo(N)$
- Tiempo:
  - Hay N subproblemas en total (uno por cada piedra)
  - Revisar todas las relaciones de un subproblema es $O(1)$ porque solo hay que ver las dos piedras anteriores

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana
- Problema Original:
  - $costo(N)$
- Tiempo:
  - Hay N subproblemas en total (uno por cada piedra)
  - Revisar todas las relaciones de un subproblema es $O(1)$ porque solo hay que ver las dos piedras anteriores
  - Combinar las soluciones de los subproblemas es $O(1)$ para obtener el mínimo de las dos opciones

Problema – Rana

- Caso Base:
  - $costo(1) = 0$, la posición inicial de la rana
- Problema Original:
  - $costo(N)$
- Tiempo:
  - Hay N subproblemas en total (uno por cada piedra)
  - Revisar todas las relaciones de un subproblema es $O(1)$ porque solo hay que ver las dos piedras anteriores
  - Combinar las soluciones de los subproblemas es $O(1)$ para obtener el mínimo de las dos opciones
  - $O(N)$

Problema – Rana

```
vector<int> memo(maxN, -1);
vector<int> h;

int costo(int piedra) {
  if (piedra == 1) return 0;
  if (memo[piedra] != -1) return memo[piedra];
  int res = INF;
  for (int ant = max(1, piedra-2); ant <= piedra-1; ++ant) {
    res = min(res, costo(ant) + abs(h[piedra] - h[ant]));
  }
  return memo[piedra] = res;
}
```

Problema – Mochila

Hay N ítems numerados de $1, 2, \ldots, n$. Cada ítem tiene un peso w_i y un valor v_i.

Tenemos una mochila que aguanta un peso máximo de W.

¿Cuál es la máxima suma de valores que podemos cargar en la mochila sin que la suma de sus pesos sobrepase W?

- $1 \leq N \leq 100$
- $1 \leq W \leq 10^5$
- $1 \leq w_i \leq W$
- $1 \leq v_i \leq 10^9$

Problema – Mochila

- Subproblemas:
  - El problema tiene 3 valores que debemos mantener, qué ítem estamos revisando, el espacio/peso restante de la mochila, y el valor de los elementos hasta ahora

Problema – Mochila

- Subproblemas:
  - El problema tiene 3 valores que debemos mantener, qué ítem estamos revisando, el espacio/peso restante de la mochila, y el valor de los elementos hasta ahora
  - Queremos hallar la máxima suma de valores, puede ser buena idea que nuestra función devuelva el mejor valor posible y así no tenemos que guardar esa información en cada estado. Además, puede ser un valor muy grande y no entrar en memoria ($v_i \leq 10^9$)

Problema – Mochila

- Subproblemas:
  - El problema tiene 3 valores que debemos mantener, qué ítem estamos revisando, el espacio/peso restante de la mochila, y el valor de los elementos hasta ahora
  - Queremos hallar la máxima suma de valores, puede ser buena idea que nuestra función devuelva el mejor valor posible y así no tenemos que guardar esa información en cada estado. Además, puede ser un valor muy grande y no entrar en memoria ($v_i \leq 10^9$)
  - Guardar los otros dos valores en cada estado es factible ($N \leq 100$, $W \leq 10^5$)

Problema – Mochila

- Subproblemas:
  - El problema tiene 3 valores que debemos mantener, qué ítem estamos revisando, el espacio/peso restante de la mochila, y el valor de los elementos hasta ahora
  - Queremos hallar la máxima suma de valores, puede ser buena idea que nuestra función devuelva el mejor valor posible y así no tenemos que guardar esa información en cada estado. Además, puede ser un valor muy grande y no entrar en memoria ($v_i \leq 10^9$)
  - Guardar los otros dos valores en cada estado es factible ($N \leq 100$, $W \leq 10^5$)
  - $mochila(i, pesoRestante)$ para $i \leq N$ y $pesoRestante \leq W$

Problema – Mochila

- Relaciones:
  - Al ver un ítem tenemos dos opciones:
    1. Añadir el ítem a la mochila
    2. Ignorarlo y revisar el siguiente

Problema – Mochila

- Relaciones:
  - Al ver un ítem tenemos dos opciones:
    1. Añadir el ítem a la mochila
    2. Ignorarlo y revisar el siguiente
  - A esto le decimos el método “tomar o no tomar”

Problema – Mochila

- Relaciones:
  - Al ver un ítem tenemos dos opciones:
    1. Añadir el ítem a la mochila
    2. Ignorarlo y revisar el siguiente
  - A esto le decimos el método “tomar o no tomar”
  - Con esta idea se resuelven ***muchos*** problemas de programación dinámica

Problema – Mochila

- Relaciones:
  - Al ver un ítem tenemos dos opciones:
    1. Añadir el ítem a la mochila
    2. Ignorarlo y revisar el siguiente
  - A esto le decimos el método “tomar o no tomar”
  - Con esta idea se resuelven ***muchos*** problemas de programación dinámica

$$mochila(i, pesoRestante) = max\begin{cases} mochila(i+1, pesoRestante - w_i) + v_i \\ mochila(i+1, pesoRestante) \end{cases}$$

Problema – Mochila

- Orden Topologico:
  - Para calcular $mochila(i, pesoRestante)$ necesitamos el valor de $mochila(j, menosPeso)$ para $i < j$ y $pesoRestante \geq menosPeso$
  - Un posible orden topologico sigue el orden de los ciclos:

```
for (int item = N-1; item >= 0; item--) {
    for (int peso = 0; peso <= W; peso++) {
        // calcular mochila(item, peso)
    }
}
```

Problema – Mochila

- Caso Base:
  - $mochila(i, 0) = 0$ cuando ya no queda espacio en la mochila
  - $mochila(N, peso) = 0$ cuando hemos visto todos los ítems $(0..n-1)$
- Problema Original:
  - $mochila(0, W)$
- Tiempo:
  - Hay N ítems y W posibles valores de peso restante, por lo tanto hay $N \times W$ estados
  - Cada estado solo tiene dos relaciones, tomar o no tomar, $O(1)$
  - La complejidad es $O(NW)$

Problema – Mochila

```
const int maxN = 101;
const int maxW = 1e5+1;
vector<vector<int>> memo(maxN, vector<int>(maxW, -1));
// int memo[maxN][maxW];
// llenar memo de -1

int N, W;
vector<int> w(maxN), v(maxN);
```

Problema – Mochila

```
int mochila(int i, int pesoRestante) {
  if (i == N) return 0;
  if (pesoRestante == 0) return 0;
  if (memo[i][pesoRestante] != -1) return memo[i][pesoRestante];
  int res = 0;
  // tomar




  // no tomar


  return memo[i][pesoRestante] = res;
}
```

Problema – Mochila

```
int mochila(int i, int pesoRestante) {
  if (i == N) return 0;
  if (pesoRestante == 0) return 0;
  if (memo[i][pesoRestante] != -1) return memo[i][pesoRestante];
  int res = 0;
  // tomar
  if (pesoRestante >= w[i]) {
    res = max(res, mochila(i+1, pesoRestante - w[i]) + v[i]);
  }
  // no tomar


  return memo[i][pesoRestante] = res;
}
```

Problema – Mochila

```
int mochila(int i, int pesoRestante) {
  if (i == N) return 0;
  if (pesoRestante == 0) return 0;
  if (memo[i][pesoRestante] != -1) return memo[i][pesoRestante];
  int res = 0;
  // tomar
  if (pesoRestante >= w[i]) {
    res = max(res, mochila(i+1, pesoRestante - w[i]) + v[i]);
  }
  // no tomar
  res = max(res, mochila(i+1, pesoRestante));

  return memo[i][pesoRestante] = res;
}
```