Programación dinámica

Matías Hunicken

Training Camp 2021

Contacto: matihunicken@gmail.com

Contenidos

Sucesión de Fibonacci

Definición

La sucesión de Fibonacci se define como $fib_0 = 0, fib_1 = 1$, $fib_n = fib_{n-1} + fib_{n-2}$ (para $n \geq 2$). Los primeros términos son $0, 1, 1, 2, 3, 5, 8, 13, 21...$

- La definición nos sugiere una forma de computarlo (función recursiva).

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

Sucesión de Fibonacci

Definición

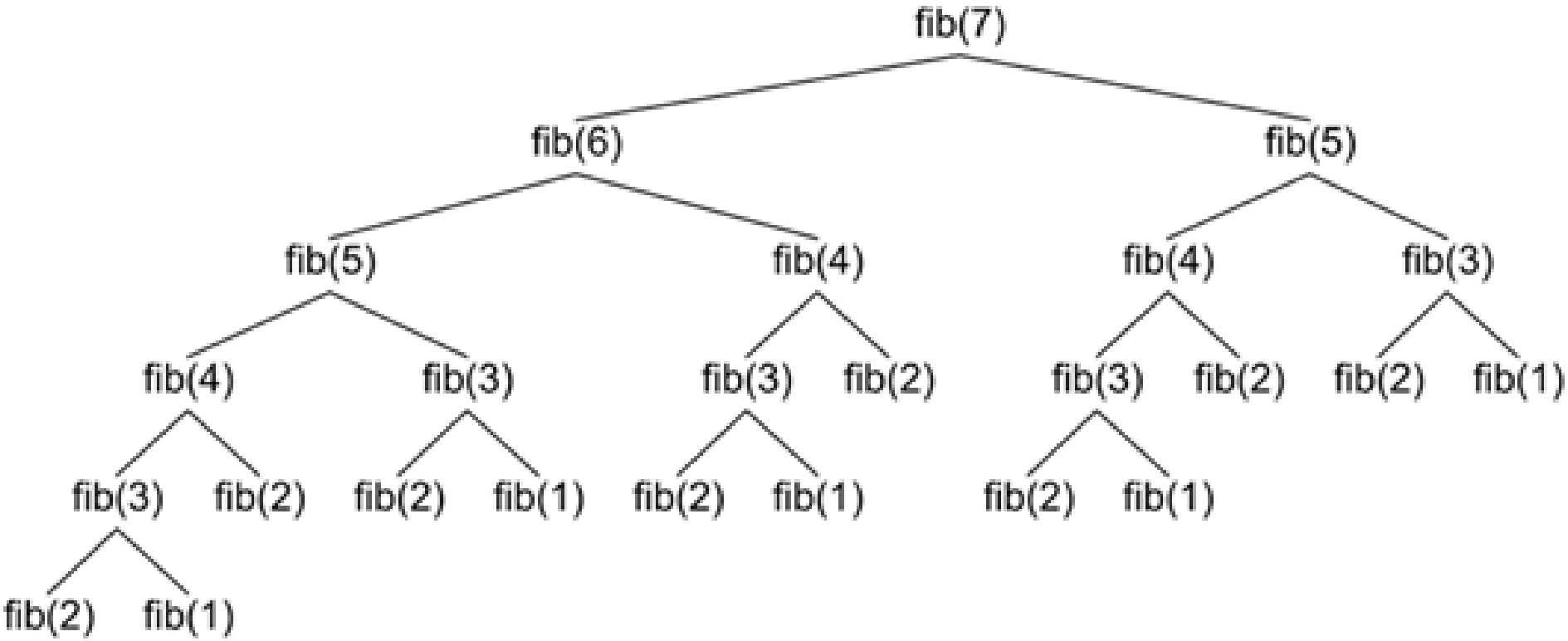
La sucesión de Fibonacci se define como $fib_0 = 0, fib_1 = 1$, $fib_n = fib_{n-1} + fib_{n-2}$ (para $n \geq 2$). Los primeros términos son $0, 1, 1, 2, 3, 5, 8, 13, 21...$

- La definición nos sugiere una forma de computarlo (función recursiva).

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

Notemos que este programa es ineficiente porque calcula muchas veces lo mismo (por ejemplo, fib(n-2) se calcula en fib(n) y también en la llamada recursiva a fib(n-1).

Fibonacci (cont.)



- La complejidad es exponencial en n.

Fibonacci - dinámica

- Podemos hacerlo mejor: en lugar de recalcular la respuesta cada vez, la calculamos una sola vez y guardamos el resultado.

```
int mem[MAX_N+1];
int fib(int n){
  if(mem[n] >= 0)
    return mem[n]
  int res;
  if(n <= 1)
    res = n;
  else
    res = fib(n-1) + fib(n-2);
  mem[n] = res;
  return res;
}
// (al principio de main):
  fill(mem, mem+MAX_N+1, -1);
```

- En el arreglo mem guardamos los resultados para los n's que ya calculamos (usamos un valor negativo para indicar que no lo calculamos).
- Ahora la complejidad de calcular todos los fibonaccis hasta n es $O(n)$.

Programación dinámica - definición

- La programación dinámica se puede ver como una recursión con una caché para guardar resultados ya calculados.
- Podemos verla como un método para resolver un problema, que lo parte en subproblemas más pequeños, los cuáles resuelve recursivamente, guardando los resultados para no calcularlos más de una vez.
- En el contexto de programación dinámica, llamamos "estado" a cada combinación de parámetros de la misma. En el caso de Fibonacci, el estado es el valor de n.
- Para calcular el costo de la programación dinámica, la cuenta suele ser "cantidad de subproblemas" * "costo de calcular el valor de un subproblema dado que los subproblemas de los cuales depende ya están calculados"
- Para realizar programación dinámica, necesitamos que no haya dependencia cíclica entre los subproblemas.

Fibonacci - versión iterativa

```
int fib[MAXN+1];
void calcFibs(int n){
//  calcula todos los fibonaccis hasta n
//  y los guarda en el arreglo fib
  fib[0] = 0;
  fib[1] = 1;
  for(int i = 2; i <= n; ++i)
    fib[i] = fib[i-1] + fib[i-2];
}
```

Progamación dinámica iterativa vs. recursiva

- Tanto la forma iterativa como la recursiva tienen ventajas y desventajas.
- Forma iterativa:
  - Suele ser más eficiente en tiempo, porque se evitan las llamadas a función.
  - A veces permite ahorrar memoria (lo veremos más adelante).
- Forma recursiva:
  - Para muchos es el modo más "natural" de pensar la programación dinámica.
  - No requiere pensar en el orden en el que debemos calcular los sub-problemas, que no siempre es obvio.
  - Es más eficiente cuando hay muchos valores de la tabla que no es necesario calcular (no es muy común, pero puede ocurrir).

Problemas de optimización

- La programación dinámica se puede usar también para resolver algunos problemas de optimización (encontrar la mejor solución, de acuerdo a cierto criterio).

Recorrido óptimo en grilla

Definición

Dada una grilla de enteros no negativos, determinar el camino desde la esquina superior izquierda a la esquina inferior derecha que maximice la suma de los números en el camino, dado que sólo nos podemos mover hacia abajo o hacia la derecha.

5	2	1	0
2	1	5	3
0	4	4	3
1	5	2	4

(Respuesta: 24)

Recorrido óptimo en grilla - Recurrencia

Definimos $F_{i,j}$ = "Valor del mayor camino hasta la posición (n, m) si empezamos desde la posición (i, j)"

(Por lo tanto, Respuesta = $F_{1,1}$)

Recorrido óptimo en grilla - Recurrencia

Definimos $F_{i,j}$ = "Valor del mayor camino hasta la posición (n, m) si empezamos desde la posición (i, j)"

(Por lo tanto, Respuesta = $F_{1,1}$)

Caso base:

$$F_{n,m} = Grid_{n,m}$$

Recorrido óptimo en grilla - Recurrencia

Definimos $F_{i,j}$ = "Valor del mayor camino hasta la posición (n, m) si empezamos desde la posición (i, j)"

(Por lo tanto, Respuesta = $F_{1,1}$)

Caso base:

$$F_{n,m} = Grid_{n,m}$$

Casos recursivos:

$$F_{i,m} = Grid_{i,m} + F_{i+1,m} \qquad (i < n)$$
$$F_{n,j} = Grid_{n,j} + F_{n,j+1} \qquad (j < m)$$
$$F_{i,j} = Grid_{i,j} + \max(F_{i+1,j}, F_{i,j+1}) \qquad (i < n \;\&\&\; j < m)$$

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3
1	5	2	4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3
1	5	2	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3
1	5	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3
1	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0
2 19	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1	0 10
2 19	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2	1 17	0 10
2 19	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5	2 19	1 17	0 10
2 19	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Ejemplo

5 24	2 19	1 17	0 10
2 19	1 17	5 16	3 10
0 15	4 15	4 11	3 7
1 12	5 11	2 6	4 4

Recorrido óptimo en grilla - Solución iterativa

```
// input:
int G[MAX_N+1][MAX_N+1];
int n,m;

int dp[MAX_N+1][MAX_N+1];

void calc_dp_iter() {
  for(int i = n; i >= 1; --i) {
    for(int j = m; j >= 1; --j) {
      if(i == n && j == m) dp[i][j] = G[i][j];
      else {
        dp[i][j] = 0;
        if(i < n)
          dp[i][j] = max(dp[i][j], G[i][j] + dp[i+1][j]);
        if(j < m)
          dp[i][j] = max(dp[i][j], G[i][j] + dp[i][j+1]);
      }
    }
  }
}
```

Recorrido óptimo en grilla - Solución recursiva

```
// input:
int G[MAX_N+1][MAX_N+1];
int n,m;

// (dp debe ser inicializado en -1 antes de llamar a f)
int dp[MAX_N+1][MAX_N+1];

int f(int i,  int j) {
// (atencion a la referencia (&) para actualizar el valor de
//  la tabla cuando manipulamos r)
  int &r = dp[i][j];
  if(r >= 0) return r;
  if(i == n && j == m) r = G[i][j];
  else {
    r = 0;
    if(i < n) r = max(r, G[i][j] + f(i+1, j));
    if(j < m) r = max(r, G[i][j] + f(i, j+1));
  }
  return r;
}
```

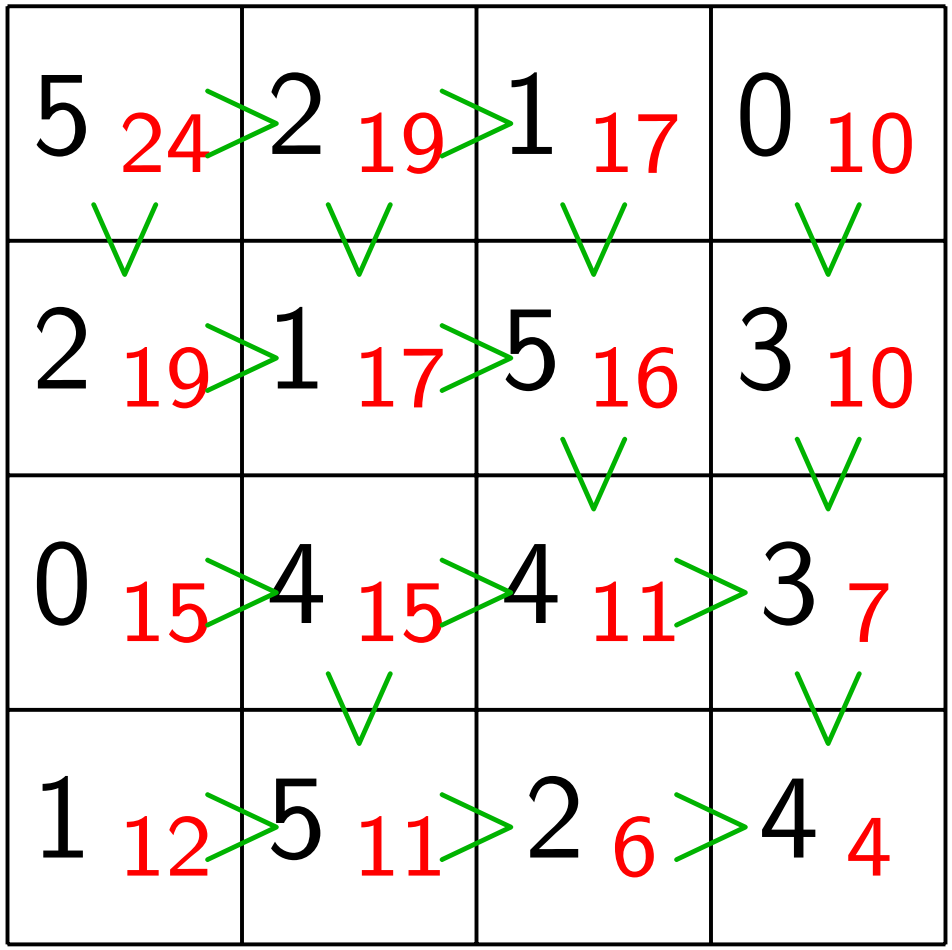
Recorrido óptimo en grilla - Generar solución

Dado que tenemos guardada la respuesta para cada subproblema, es sencillo saber si un movimiento es óptimo o no:

- Es óptimo moverme hacia abajo desde (i, j) cuando:
  $$F_{i,j} = Grid_{i,j} + F_{i+1,j}.$$
- Es óptimo moverme hacia la derecha desde (i, j) cuando:
  $$F_{i,j} = Grid_{i,j} + F_{i,j+1}.$$

Usando esto, es sencillo generar una solución: Empiezo desde $(1, 1)$ y realizo movimientos óptimos hasta llegar a (n, m).

Recorrido óptimo en grilla - Generar solución (ejemplo)



Recorrido óptimo en grilla - Generar solución (código)

```
string sol = "";

// (gen_sol_from debe ser llamada despues de calcular dp)
void gen_sol_from(int i, int j) {
  if (i == n && j == m) return;
  if (i < n && dp[i][j] == G[i][j] + dp[i+1][j]) {
    sol.push_back('A'); // (Abajo)
    gen_sol_from(i+1, j);
  } else {
    sol.push_back('D'); // (Derecha)
    gen_sol_from(i, j+1);
  }
}
```

Recorrido óptimo en grilla - Contar soluciones

- En los problemas de optimización, a veces nos piden decir cuántas soluciones óptimas hay.
- Esto es sencillo de realizar una vez que sabemos para cada paso si es óptimo o no, usando otra DP:

Recorrido óptimo en grilla - Contar soluciones

- En los problemas de optimización, a veces nos piden decir cuántas soluciones óptimas hay.
- Esto es sencillo de realizar una vez que sabemos para cada paso si es óptimo o no, usando otra DP:

Definimos $C_{i,j}$ = "Cantidad de caminos óptimos desde la posición (i, j) hasta (n, m)"

Recorrido óptimo en grilla - Contar soluciones

- En los problemas de optimización, a veces nos piden decir cuántas soluciones óptimas hay.
- Esto es sencillo de realizar una vez que sabemos para cada paso si es óptimo o no, usando otra DP:

Definimos $C_{i,j}$ = "Cantidad de caminos óptimos desde la posición (i, j) hasta (n, m)"

Caso base:

$$C_{n,j} = C_{i,m} = 1$$

Recorrido óptimo en grilla - Contar soluciones

- En los problemas de optimización, a veces nos piden decir cuántas soluciones óptimas hay.
- Esto es sencillo de realizar una vez que sabemos para cada paso si es óptimo o no, usando otra DP:

Definimos $C_{i,j}$ = "Cantidad de caminos óptimos desde la posición (i, j) hasta (n, m)"

Caso base:

$$C_{n,j} = C_{i,m} = 1$$

Caso recursivo:

$$C_{i,j} = (F_{i,j} == Grid_{i,j} + F_{i+1,j}) * C_{i+1,j} \; + \\ (F_{i,j} == Grid_{i,j} + F_{i,j+1}) * C_{i,j+1} \qquad (i < n \;\&\&\; j < m)$$

Recorrido óptimo en grilla - Contar soluciones (código)

```
int cnt[MAX_N + 1][MAX_N + 1];

int calc_num_sols() {
  for(int i = n; i >= 1; --i) {
    for(int j = m; j >= 1; --j) {
      if(i == n && j == m) cnt[i][j] = 1;
      else {
        cnt[i][j] = 0;
        if(i < n && dp[i][j] == G[i][j] + dp[i+1][j])
          cnt[i][j] += cnt[i+1][j];
        if(j < m && dp[i][j] == G[i][j] + dp[i][j+1])
          cnt[i][j] += cnt[i][j+1];
      }
    }
  }
  return cnt[1][1];
}
```

Recorrido óptimo en grilla - Generar k-ésima solución

- A veces, no nos piden cualquier solución, sino la k-ésima lexicográfica.

5	2	1	0
2	1	5	3
0	4	4	3
1	5	2	4

En este caso, las soluciones (ordenadas lexicográficamente) son:

1. ADDADA
2. DADADA
3. DDAADA

La solución es similar a encontrar cualquier solución, excepto que decidimos qué paso hacer dependiendo de la cantidad de soluciones de los subproblemas.

Recorrido óptimo en grilla - k-ésima solución (código)

```
string sol = "";

void gen_kth(int i, int j, int k) {  // (index from 1)
  if(i == n && j == m) return;
  if(i < n && dp[i][j] == G[i][j] + dp[i+1][j]) {
    // (si moverme para abajo es optimo)
    if(cnt[i+1][j] >= k) {
      // (si hay al menos k soluciones hacia abajo)
      sol.push_back('A');
      gen_kth(i+1,j,k);
      return;
    }
    // (si moverme para abajo es optimo, pero la k-esima
    solucion no "cae" ahi, descuento la cantidad de
    soluciones hacia abajo antes de seguir)
    k -= cnt[i+1][j];
  }
  sol.push_back('D');
  gen_kth(i, j+1, k);
}
```

Recorrido óptimo en grilla - Optimización de memoria

- Podemos hacer una optimización de memoria, usando que en cada paso sólo accedemos al valor del subproblema en la fila actual o en la siguiente, pero nunca más allá.
- Por lo tanto, si lo hacemos iterativamente, sólo tenemos que guardar la fila actual y la que procesamos anteriormente.

- Esta optimización suele no ser necesaria, pero es importante tenerla en cuenta si la programación dinámica consume mucha memoria, o el problema está ajustado en tiempo.

Optimización de memoria - Código

```
int dp[2][MAX_N+1];

int calc_dp_iter() {
  for(int i = n; i >= 1; --i) {
    for(int j = m; j >= 1; --j) {
      if(i == n && j == m) dp[i%2][j] = G[i][j];
      else {
        dp[i%2][j] = 0;
        if(i < n)
          dp[i%2][j] = max(dp[i%2][j],G[i][j]+dp[1-i%2][j]);
        if(j < m)
          dp[i%2][j] = max(dp[i%2][j],G[i][j]+dp[i%2][j+1]);
      }
    }
  }
  return dp[1][1];
}
```

Traveling salesman problem

Problema

Hay n ciudades, y para cada par de ellas (i, j) existe un vuelo de i a la j, con costo $C_{i,j}$. Se pide determinar el menor costo de empezar en la primera ciudad, visitar todas las ciudades exactamente una vez y volver a la primera.

La solución obvia es $O(n!)$ (probar todas las permutaciones), pero podemos hacerlo un poco mejor con programación dinámica.

Traveling salesman problem - Solución

Problema

Hay n ciudades, y para cada par de ellas (i, j) existe un vuelo de i a la j, con costo $C_{i,j}$. Se pide determinar el menor costo de empezar en la primera ciudad, visitar todas las ciudades exactamente una vez y volver a la primera.

Definimos $F_{x,S}$ = "Valor del menor camino, dado que estamos en el nodo x y visitamos los nodos del conjunto S" (Por lo tanto, Respuesta = $F_{0,\{0\}}$)

Traveling salesman problem - Solución

Problema

Hay n ciudades, y para cada par de ellas (i, j) existe un vuelo de i a la j, con costo $C_{i,j}$. Se pide determinar el menor costo de empezar en la primera ciudad, visitar todas las ciudades exactamente una vez y volver a la primera.

Definimos $F_{x,S}$ = "Valor del menor camino, dado que estamos en el nodo x y visitamos los nodos del conjunto S" (Por lo tanto, Respuesta = $F_{0,\{0\}}$)

Caso base:

$$F_{x,\{0,\ldots,n-1\}} = C_{x,0}$$

Traveling salesman problem - Solución

Problema

Hay n ciudades, y para cada par de ellas (i, j) existe un vuelo de i a la j, con costo $C_{i,j}$. Se pide determinar el menor costo de empezar en la primera ciudad, visitar todas las ciudades exactamente una vez y volver a la primera.

Definimos $F_{x,S}$ = "Valor del menor camino, dado que estamos en el nodo x y visitamos los nodos del conjunto S" (Por lo tanto, Respuesta = $F_{0,\{0\}}$)

Caso base:

$$F_{x,\{0,\ldots,n-1\}} = C_{x,0}$$

Caso recursivo:

$$F_{x,S} = Min\left\{C_{x,y} + F_{x,S\cup\{y\}} \mid y \in \{0,\ldots,n-1\} - S\right\}$$

¿Cómo representar subconjuntos?

- En el problema anterior, tenemos que parte del estado de la DP es un conjunto. ¿Cómo representamos eso en código?
- Una opción es usar un vector con los elementos del conjunto, pero es muy caro en tiempo y memoria.

Una solución más eficiente cuando los conjuntos contienen pocos valores posibles, es usar la representación en binario de un número:

- Si los valores posibles en el subconjunto están en $\{0, ..., n-1\}$, representamos el subconjunto como un entero en $[0, 2^n)$.
- Los unos en la representación binaria del número representan los elementos contenidos en el subconjunto.
- Es decir, representamos S como $\sum_{k \in S} 2^k$.

Manipulación de bits

Representando el conjunto como entero, podemos hacer muchas operaciones sobre subconjuntos eficientemente:

$S \cap T$	:	`S & T`
$S \cup T$	:	`S | T`
$S - T$	:	`S & ~T`
$\{x\}$	:	`1 << x`
$\{0, ..., n-1\}$	:	`(1 << n) - 1`
$\|S\|$	:	`__builtin_popcount(S)`
$min(S)$	:	`__builtin_ctz(S)`
$max(S)$	:	`31 - __builtin_clz(S)`

Manipulación de bits

Representando el conjunto como entero, podemos hacer muchas operaciones sobre subconjuntos eficientemente:

$S \cap T$	:	`S & T`
$S \cup T$	:	`S | T`
$S - T$	:	`S & ~T`
$\{x\}$	:	`1 << x`
$\{0, ..., n-1\}$	:	`(1 << n) - 1`
$\|S\|$	:	`__builtin_popcount(S)`
$min(S)$	:	`__builtin_ctz(S)`
$max(S)$	:	`31 - __builtin_clz(S)`

Nota:

- Las funciones `__builtin_cosa` sólo están disponibles en GCC.
- Éstas son específicas para enteros de 32 bit. Hay funciones equivalentes para enteros de 64 bit: `__builtin_popcountll`, etc.

Traveling salesman problem - Código

```
int C[MAX_N][MAX_N];
int n;

int dp[MAX_N][1 << MAX_N];

int f(int x, int mask) { // (mask contiene los ya visitados)
  int &r = dp[x][mask];
  if(r >= 0) return r;
  if(mask == (1 << n) - 1) r = C[x][0];
  else {
    r = INF;
    for(int y = 0; y < n; ++y) {
      if((mask & (1<<y)) == 0) {
        r = min(r, C[x][y] + f(y, mask | (1<<y)));
      }
    }
  }
  return r;
} // respuesta: f(0,1)
```

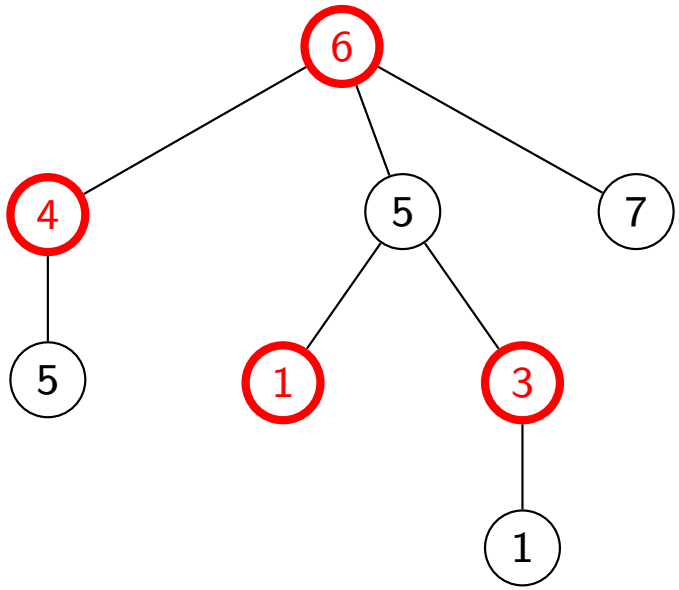
Traveling salesman problem - Complejidad

- La complejidad es $O(n^2 \times 2^n)$ ($n \times 2^n$ estados de la DP, $O(n)$ complejidad para cada estado).
- Esto es mucho mejor que la solución trivial $O(n \times n!)$, y nos permite resolver para n alrededor de 20.

DP en árboles - cubrimiento de vértices de costo mínimo

Problema

Se tiene una ciudad que consta de n esquinas conectadas por $n-1$ calles bidireccionales, de modo que todas las equinas están conectadas directa o indirectamente. Cada esquina x tiene un costo asociado para alumbrarla C_x. Se quiere saber el menor costo de alumbrar esquinas de modo que para toda calle, al menos uno de los extremos esté alumbrado.



Cubrimiento de vértices - solución (estados)

- Definimos un nodo cualquiera como raíz del árbol.
- Luego, podemos pensar que el subproblema es "resolver para un sub-árbol".
- Además, necesitamos saber si el padre del nodo raíz del sub-árbol fue pintado o no.

Definimos

$F_{x,false}$ como "Valor del menor cubrimiento del subárbol de x dado que no pintamos al padre de x".

$F_{x,true}$ como "Valor del menor cubrimiento del subárbol de x dado que pintamos al padre de x".

Entonces, la respuesta sería $F_{raiz,true}$ (para la raíz, nos "inventamos" un nodo padre que ya está pintado, para que no nos obligue a pintar la raíz).

Cubrimiento de vértices - solución (transiciones)

$F_{x,false}$ = "Valor del menor cubrimiento del subárbol de x dado que no pintamos al padre de x".

$F_{x,true}$ = como "Valor del menor cubrimiento del subárbol de x dado que pintamos al padre de x".

El caso más sencillo es el primero, ya que en ese caso estamos obligados a pintar x para cubrir la arista de x a su padre.

$$F_{x,false} = C_x + \sum_{y \in hijos(x)} F_{y,true}$$

En el segundo caso, podemos elegir si pintar x o no:

$$F_{x,true} = min(C_x + \sum_{y \in hijos(x)} F_{y,true}, \sum_{y \in hijos(x)} F_{y,false})$$

Cubrimiento de vértices - código

```
vector<int> g[MAX_N]; // Grafo no dirigido
int C[MAX_N];         // Costos

int F[MAX_N][2];      // DP

void dfs(int x, int father) {
  F[x][0] = C[x];
  F[x][1] = 0;
  for(int y: g[x]) {
    if(y == father) continue;
    dfs(y, x);
    F[x][0] += F[y][1];
    F[x][1] += F[y][0];
  }
  F[x][1] = min(F[x][1], F[x][0]);
}
// Para obtener la respuesta:
  dfs(0,-1);
  respuesta = F[0][1];
```

DP sobre dígitos

Problema

Dados dos enteros positivos U (hasta 10^{18}) y D (hasta 100), decir cuántos enteros positivos x hay tales que $x \leq U$, x es múltiplo de D, y la suma de dígitos de x es múltiplo de D.

Podemos construir el número x de izquierda a derecha, guardando:

- En qué posición estamos.
- Valor de x *mod* D.
- Valor de "suma de dígitos de x" *mod* D.
- Un flag que nos dice si ya somos menores que U o no.

En cada paso, iteramos para todos los dígitos posibles (si el flag es falso sólo podemos iterar hasta el valor del dígito correspondiente de U) y transicionamos a la siguiente posición actualizando los valores.

DP sobre dígitos - Código

```
string U; // Digitos del numero U
int D;

long long dp[MAX_DIGITS+1][MAX_D][MAX_D][2];

long long f(int k , int xModD, int sModD, int menor) {
  long long& r = dp[k][xModD][sModD][menor];
  if(r >= 0) return r;
  if(k == U.size()) r = !xModD && !sModD;
  else {
    r = 0;
    for(int t = 0; t <= 9; ++t) {
      if (!menor && t > U[k] - '0') break;
      r += f(k+1, (xModD * 10 + t) % D, (sModD + t) % D,
    menor || t < U[k] - '0');
    }
  }
  return r;
} // respuesta: f(0,0,0,0) - 1
  // (se debe restar 1 porque esta contando al 0)
```

Exponenciación de matrices

En “pizarrón” (si llegamos).

Conclusión

- Programación dinámica es uno de los tópicos más comunes en el contexto de programación competitiva, por lo que es muy importante dedicarle tiempo a practicarlo.
- La clave para resolver problemas de programación dinámica es definir los estados correctamente. Esto sólo se mejora practicando :)

Algunos problemas

- `https://atcoder.jp/contests/dp/tasks` : 26 problemas de DP variados y relativamente sencillos, que usan varias técnicas clásicas.
- `https://a2oj.com/Category33.html`: Problemas de DP ordenados por dificultad.
- DP con máscara:
  - `https://www.spoj.com/problems/ASSIGN/`
  - `https://codeforces.com/problemset/problem/11/D`
  - `https://codeforces.com/problemset/problem/16/E`
- DP sobre dígitos:
  - `https://www.spoj.com/problems/CPCRC1C/`
  - `https://www.urionlinejudge.com.br/judge/problems/view/2013`
  - `https://codeforces.com/contest/628/problem/D`
- DP en árboles:
  - `https://acm.timus.ru/problem.aspx?space=1&num=1039`
  - `https://codeforces.com/contest/461/problem/B`
- Exponenciación de matrices:
  - `https://codeforces.com/contest/222/problem/E`
  - `https://www.spoj.com/problems/TAP2015E/`

¡Gracias!