# Programación dinámica

Vi que bastante gente se traba a la hora de formular un algoritmo que resuelta un problema de programación dinámica, ya teniendo una formulación recursiva del problema. Está bueno ver que hay dos formas de hacer programación dinámica: *top-down* o *bottom-up*. Cada una tiene sus ventajas y sus desventajas, pero se puede pasar de una a la otra.

top-down DP (uso DP para decir "programación dinámica", o "dynamic programming", en Inglés) es guardar los resultados de cálculos previos, para poder usarlos después. Bottom-up DP es ir construyendo soluciones cada vez más grandes, basándose en soluciones más chicas.

Como esto es todo medio nebuloso, vamos a poner un ejemplo que aparece en *Introduction to Algorithms*, de Cormen, Leisserson, Rivest y Stein. Lo vamos a pensar, y resolver de las dos formas. Es un problema bastante típico de programación dinámica.

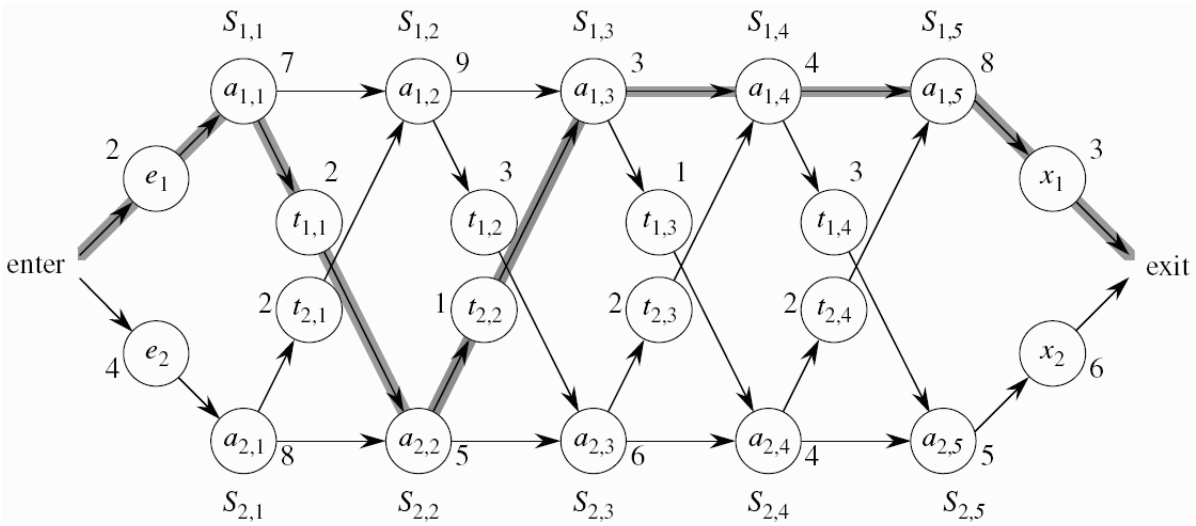
## Fábrica de autos

En este problema, somos ingenieros en una fábrica autos. Un auto es creado desde un montón de metal, pasando por varias estaciones, hasta salir por la puerta de la fábrica. Las estaciones están numeradas, $S_1, \cdots, S_n$. Un auto siempre pasa de la estación $S_i$ a la $S_{i+1}$ (no vamos a pintar un montón de metal, o ponerle motor a algo sin carrocería).

La planta es muy moderna, y cuenta con dos copias de cada de cada estación, para poder trabajar en paralelo. Estas están ordenadas en dos filas paralelas, y son $S_{i,1}$ y $S_{i,2}$, con $1 \leq i \leq n$. Además, cada estación $S_{i,j}$ puede mandar su resultado a $S_{i+1,j}$, o cruzarlo de fila a la estación $S_{i+1,\sigma(j)}$, con $\sigma(j) = -j + 3$ (vemos que $\sigma(1) = 2, \sigma(2) = 1$).

Un auto necesita un tiempo $a_{i,j}$ para terminar de etapa en la $i$-ésima estación de la fila $j$, y necesita un tiempo $t_{i,j}$ para cambiar de estación a la $S_{i+1,\sigma(j)}$ al salir de la estación $S_{i,j}$. Quedarse en su misma fila al cambiar de estación no cuesta nada. Por último, necesita $e_1$ tiempo para entrar por la fila 1, $e_2$ tiempo para entrar por la fila 2, $x_1$ tiempo para salir por la fila 1, y $x_2$ tiempo para salir por la fila 2.

En forma gráfica, con algunos valores de ejemplo puestos al lado de las variables:




De vez en cuando, nos piden que saquemos un auto con urgencia. Nuestro trabajo es decidir por donde tiene que pasar el auto, desde inicio hasta final, para que salga lo antes posible. ¿Cuán rápido podemos hacer un auto entero?

Vemos que probar todos los posibles caminos y quedarse con el mínimo, la primera idea que se nos ocurre, no es aceptablemente rápida: un camino es una selección de fila, para cada estación del 1 al $n$. Por lo tanto, como hay 2 posibles filas para cada estación, la cantidad de caminos es $2^n$. Claramente no podemos gastar tanto tiempo!

Ahora, recordemos que podemos intentar buscar las siguientes dos propiedades, cuando sospechemos que podemos usar programación dinámica:

(1) Subestructura óptima
(2) Subproblemas compartidos

Subestructura óptima significa que la solución óptima al problema grande está compuesta por soluciones óptimas a problemas idénticos, pero un poco más chicos. Subproblemas compartidos significa que para resolver un problema $X$, necesito resolver varios sub-problemas $X_1, \cdots, X_k$, y muchos de los sub-problemas que necesito para resolver un $X_i$, también los necesito para resolver un $X_j$. Ahora, ¿Qué nos podemos plantear sobre este problema, que exhiba ambas características? ¿Qué puedo plantearme como "problema", tal que para resolverlo, tenga varios sub-problemas idénticos pero un poco más chicos?

Pensalo. Pensalo un rato. Esta es la parte más difícil de hacer programación dinámica. La respuesta está en la página siguiente, pero no la mires hasta no haber estado un rato largo (ponele, 30 minutos, 1 hora) pensando y no te salió.

Te doy una página más para que te arrepientas de haber espiado la solución.

Bueno, la onda es que podemos ver que la mejor manera de llegar a la estación $i$-ésima, es o quedarse en la misma fila en la estación $i-1$-ésima, o cambiarse de fila en la $i-1$-ésima. Entonces, el mínimo tiempo que puedo tardar en completar la estación $i$-ésima en la fila $j$, es el tiempo que me toma la estación en sí, más el mínimo entre (quedarme en la fila más el mínimo que me toma la estación $i-1$-ésima en la fila $j$), y (el mínimo que me toma la estación $i-1$-ésima en la fila $\sigma(j)$, más el tiempo que me toma cambiarme de fila).

En otras palabras, si $f$ es la función que encuentra el mínimo tiempo:

$$f(i,j) = \min(f(i-1,j), f(i-1,\sigma(j)) + t_{i,\sigma(j)}) + a_{i,j}$$

Este problema, ahora, está escrito en forma recursiva. Los casos base son:

$$f(0,1) = e_1$$
$$f(0,2) = e_2$$

En cada estación, podía haber llegado desde 2 estaciones anteriores. La mejor forma de llegar a la estación $i$-ésima, tiene que ser o cambiándome de fila, o quedándome en la misma. En ambos casos, la mejor solución para la estación $i$-ésima, **debe** contener a la mejor solución para la estación $i-1$-ésima (de no ser así, podríamos reemplazar esta sub-solución en nuestra solución para la $i$-ésima, y tendríamos una mejor forma de resolver el problema, lo cual es absurdo porque nuestra manera original era óptima).

Ahora, esto sólo nos dice que es un problema recursivo, y que tiene subestructura óptima. Tendrá subproblemas compartidos?

Veamos qué pasa con un $f(i,j)$, qué otros $f(i',j')$ necesita para calcularse?
$f(i,j) \leftarrow f(i-1,j), f(i-1,\sigma(j))$
$f(i,j) \leftarrow f(i-2,j), f(i-2,\sigma(j)), f(i-2,\sigma(j)), f(i-2,j)$, pués $\sigma(\sigma(j)) = j$
$f(i,j) \leftarrow f(i-3,j), f(i-3,\sigma(j)), f(i-3,\sigma(j)), f(i-3,j), f(i-3,j), f(i-3,\sigma(j))$

Como vemos, los problemas que necesitamos resolver tienen muchas repeticiones. Si programáramos el algoritmo recursivo de forma normal, nuestra función, evaluada en $i,j$, llamaría 3 veces a nuestra función en $i-3,j$, y 3 veces a $i-3,\sigma(j)$. Esto obviamente crece cada vez más, a medida que vamos yendo más hondo en el árbol de llamadas.

Entonces, vemos que tiene subproblemas compartidos, y subestructura óptima. Qué bien, podemos usar DP :)

Ahora podemos usar o bottom-up dynamic programming, o top-down dynamic programming. Empecemos con la que es, a mi parecer, más fácil: top-down.

Esto consiste en guardarse los cómputos en algún lugar, y luego, cuando nos lo vuelven a pedir, no calcularlo otra vez, sino devolverlo de la vez que lo guardamos. El código es exactamente igual que la función recursiva que haríamos normalmente, con la sola modificación de que al principio de la función preguntamos "Che, ¿puede ser que yo ya haya calculado el valor de la función para los parámetros que fui pasado?", y en caso de ya haberlo calculado, lo devolvemos, de otra manera lo calculamos, guardamos y devolvemos.

En este caso, el código para eso sería:

```
MANERA-OPTIMA(i, j)
1  if cache[i][j] = −1
2     then cache[i][j] ← a[i][j] + min{
3             MANERA-OPTIMA(i − 1, j),
4             MANERA-OPTIMA(i − 1, σ(j)) + t[i − 1][σ(j)]
5             }
6  return cache[i][j]
```

Previamente, debemos inicializar $cache[i][j] = -1 \; \forall \; 1 \leq i \leq n, 1 \leq j \leq 2$, y $cache[0][1] = e_1$, $cache[0][2] = e_2$. Para calcular el resultado final, debemos preguntar el mínimo entre MANERA-OPTIMA$(n, 1) + x_1$ y MANERA-OPTIMA$(n, 2) + x_2$. Notar que como llenamos el cache inicialmente, no es necesario poner el caso base dentro del algoritmo en sí, ya va a estar contemplado por la información en el cache. Es puramente estilístico si poner el caso base en el algoritmo o escribirlo en el cache directamente.

Como vemos, la estructura de $f$ está presente en la función MANERA-OPTIMA, lo único que se hace es guardarse el cómputo una vez que lo calcula. Cuando pidamos MANERA-OPTIMA$(i, j)$, en algún momento de calcular MANERA-OPTIMA$(i - 1, j)$, va a calcular MANERA-OPTIMA$(i - 2, j)$ y MANERA-OPTIMA$(i - 2, \sigma(j))$ y guardarlos en *cache*. Luego, cuando intente calcular MANERA-OPTIMA$(i - 1, \sigma(j))$ y pida calcular MANERA-OPTIMA$(i - 2, \sigma(j))$ y MANERA-OPTIMA$(i - 2, j)$, no los va a volver a calcular - los devuelve del cache. Esta segunda llamada, es $\mathcal{O}(1)$.

Entonces, cada vez que vayamos por una "rama derecha" (la línea 4) de la recursión, esa llamada va a costar $\mathcal{O}(1)$, pues ya la habremos calculado y guardado[1] cuando computamos la rama izquierda (la línea 3). Vemos que MANERA-OPTIMA es, entonces, $\mathcal{O}(n)$. Hace una llamada a algo que es $\mathcal{O}(n - 1)$, y una llamada a algo que es $\mathcal{O}(1)$, más cosas de orden constante (tomar mínimo de 2 elementos y buscar elementos en una matriz).

[1] En realidad, calculamos y guardamos sus 2 subproblemas, con lo cual lo que es $\mathcal{O}(1)$ son los dos subproblemas de la rama derecha. Como la rama hace $\mathcal{O}(1)$ trabajo aparte de las llamadas recursivas, termina costando $\mathcal{O}(1)$ en total.

Esa es la clave que hace que top-down DP funcione. La estructura de la función es la misma, pero guardamos los cómputos previos para no tener que recalcularlos. Para los curiosos, esta técnica se llama *memoización* (sí, sin r).

Ahora veamos cómo haríamos la parte bottom-up. Esto es equivalente, en realidad, a ver *en qué orden se llena la matriz en la versión top-down.* También podemos pensar "¿Qué entradas de la matriz necesito para calcular la $(i, j)$-ésima?"

Una forma habitual de hacer esto es programar la top-down, e imprimir a pantalla cuando uno está guardando un valor en la cache por primera vez. Entonces uno ve la pantalla y dice "Ah, primero llené la primera fila, luego la primera columna, luego las diagonales con índice primo!", o lo que sea que uno pueda deducir del orden en que las cosas se imprimieron a pantalla. Si quieren, programen la versión top-down de arriba y vean en qué orden se imprimen las cosas a la pantalla.

Bueno, para resolver la entrada $(i, j)$-ésima, necesitamos las entradas $(i-1, j)$ y $(i-1, \sigma(j))$. Esto lo podemos o deducir de la función recursiva que describe el problema, o ver de la salida del algoritmo top-down. Luego, sabiendo estos dos valores, podemos completar la $(i, j)$ con el cálculo que hace la función $f$. Entonces, podríamos completar las casillas con un ciclo, en vez de con recursión:

```
COMPLETAR-MATRIZ
 1  matriz ∈ ℕ^{2×n}
 2  matriz[0][1] ← e_1
 3  matriz[0][2] ← e_2
 4  for i ← 1 to n
 5        do for j ← 1 to 2
 6                 do matriz[i][j] ← a[i][j] + min{
 7                     matriz[i − 1][j],
 8                     matriz[i − 1][σ(j)] + t[i − 1][σ(j)]
 9                     }
10  return min{matriz[n][1] + x_1, matriz[n][2] + x_2}
```

El análisis asintótico de este código es aún más fácil, es claramente $\mathcal{O}(n)$. Vemos que los dos programas corren con el mismo comportamiento asintótico, top-down teniendo el costo extra de la recursión[2].

Y eso es todo. El problema está resuelto. Vimos las dos formas de hacerlo, y, lo más importante, cómo intentar encarar estos ejercicios. La parte más difícil es encontrar una forma de describir el problema, que presente subestructura óptima y subproblemas

[2]Para los que hicieron Orga 2, armar y desarmar stack frames.

compartidos. ¿Cómo encontramos esta forma? Práctica. Práctica y práctica. Nadie nace imaginándose estas cosas al toque, te tenés que romper la cabeza una y otra vez, y empezás a encontrar patrones de cómo se comportan.

Ahora, si entendieron esto, les dejo algo para que piensen. ¿Cómo podríamos efectivamente *armar* el camino óptimo a traves de la fábrica, no sólo saber cuanto tiempo toma? ¿Hay alguna forma eficiente de devolver dicho camino, digamos en una lista o vector de estaciones? La solución está en la próxima página, ¡pero no espíes!

La idea acá, y siempre, es ir fijándote qué decisiones tomaste en cada punto, y guardarte esa decisión. Entonces, la pregunta que nos hacemos en cada estación, es si es mejor quedarse en esta fila, o cambiar de fila. Todo lo que tenemos que devolver para que alguien sepa qué camino tomar, es la lista de donde hago cada cosa. Es decir, podemos devolver un vector $v[1 \dots n]$, donde $v[i]$ es por cuál fila entramos a la $i$-ésima estación.

Lo único que va a cambiar en nuestra función, es que en vez de tomar el mínimo, que es equivalente a esto:

```
1  matriz[i][j] ← a[i][j]
2  izquierda ← matriz[i-1][j]
3  derecha ← matriz[i-1][σ(j)] + t[i − 1][σ(j)]
4  if izquierda < derecha
5      then matriz[i][j] ← matriz[i][j] + izquierda
6      else matriz[i][j] ← matriz[i][j] + derecha
```

Vamos a recordar *qué* mínimo tomamos:

```
1  matriz[i][j] ← a[i][j]
2  izquierda ← matriz[i-1][j]
3  derecha ← matriz[i-1][σ(j)] + t[i − 1][σ(j)]
4  if izquierda < derecha
5      then matriz[i][j] ← matriz[i][j] + izquierda
6           decision[i] ← j
7      else matriz[i][j] ← matriz[i][j] + derecha
8           decision[i] ← σ(j)
```

Entonces, al terminar, *decision* tendrá una lista de, para cada estación, qué fila me conviene usar. Esto, como dijimos antes, es un camino, y esta es la reconstrucción del camino pedido. Como vemos, no hay un gran cambio en nuestro código - es sólo agregar una línea, para recordar *qué* mínimo preferimos, y no sólo su valor. En general, esto es cierto para los problemas de programación dinámica. Si ya estamos calculando cuál es la mejor solución, pero sólo estamos tomando su *valor*, saber *cuál* es es solo un tema de guardarnos el resultado de nuestra decisión.