Programación dinámica

En informática, la **programación dinámica** es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas, como se describe a continuación.

El matemático Richard Bellman inventó la **programación dinámica** en 1953 que se utiliza para optimizar problemas complejos que pueden ser discretizados y secuencializados.

Introducción

Una subestructura óptima significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Por ejemplo, el camino más corto entre dos vértices de un grafo se puede encontrar calculando primero el camino más corto al objetivo desde todos los vértices adyacentes al de partida, y después usando estas soluciones para elegir el mejor camino de todos ellos. En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven a su vez dividiéndolos en subproblemas más pequeños hasta que se alcance el caso fácil, donde la solución al problema es trivial.

Decir que un problema tiene *subproblemas superpuestos* es decir que se usa un mismo subproblema para resolver diferentes problemas mayores. Por ejemplo, en la sucesión de Fibonacci ($F_3 = F_1 + F_2$ y $F_4 = F_2 + F_3$) calcular cada término supone calcular F_2. Como para calcular F_5 hacen falta tanto F_3 como F_4, una mala implementación para calcular F_5 acabará calculando F_2 dos o más veces. Esto sucede siempre que haya subproblemas superpuestos: una mala implementación puede acabar desperdiciando tiempo recalculando las soluciones óptimas a problemas que ya han sido resueltos anteriormente.

Esto se puede evitar guardando las soluciones que ya hemos calculado. Entonces, si necesitamos resolver el mismo problema más tarde, podemos obtener la solución de la lista de soluciones calculadas y reutilizarla. Este acercamiento al problema se llama *memoización* (no confundir con memorización; en inglés es llamado memoization, véase en). Si estamos seguros de que no volveremos a necesitar una solución en concreto, la podemos descartar para ahorrar espacio. En algunos casos, podemos calcular las soluciones a problemas que de antemano sabemos que vamos a necesitar.

En resumen, la programación hace uso de:

- Subproblemas superpuestos
- Subestructuras óptimas
- Memorización

La programación toma normalmente uno de los dos siguientes enfoques:

- **Top-down**: El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de memorización y recursión.
- **Bottom-up**: Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

Originalmente, el término de *programación dinámica* se refería a la resolución de ciertos problemas y operaciones fuera del ámbito de la Ingeniería Informática, al igual que hacía la *programación lineal*. Aquel contexto no tiene relación con la programación en absoluto; el nombre es una coincidencia. El término también lo usó en los años 40

Richard Bellman, un matemático norteamericano, para describir el proceso de resolución de problemas donde hace falta calcular la mejor solución consecutivamente.

Algunos lenguajes de programación funcionales, sobre todo Haskell, pueden usar la memorización automáticamente sobre funciones con un conjunto concreto de argumentos, para acelerar su proceso de evaluación. Esto sólo es posible en funciones que no tengan efectos secundarios, algo que ocurre en Haskell pero no tanto en otros lenguajes.

Principio de optimalidad

Cuando hablamos de *optimizar* nos referimos a buscar alguna de las **mejores** soluciones de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz. En otros casos, aunque no sea posible aplicar la estrategia voraz, se cumple el **principio de optimalidad de Bellman** que dicta que «dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima». En este caso sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y Vencerás, técnica similar a la de programación dinámica. La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

- Una enorme cantidad de problemas.
- Problemas cuyas soluciones parciales se solapan.
- Grupos de problemas de muy distinta complejidad.

Empleos

Sucesión de Fibonacci

Esta sucesión puede expresarse mediante la siguiente recurrencia:

$$Fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 1 \end{cases}$$

Una implementación de una función que encuentre el **n**-ésimo término de la sucesión de Fibonacci basada directamente en la definición matemática de la sucesión realizando llamadas recursivas hace mucho trabajo redundante, obteniéndose una complejidad exponencial:

```
FUNC fib(↓n: NATURAL): NATURAL
INICIO
    SI n = 0 ENTONCES
        DEVOLVER 0
    SINOSI n = 1 ENTONCES
        DEVOLVER 1
    SINO
        devolver fib(n-1) + fib(n-2)
    FINSI
FIN
```

Si llamamos, por ejemplo, a `fib(5)`, produciremos un árbol de llamadas que contendrá funciones con los mismos parámetros varias veces:

1. `fib(5)`
2. `fib(4) + fib(3)`
3. `(fib(3) + fib(2)) + (fib(2) + fib(1))`
4. `((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`
5. `(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))`

En particular, `fib(2)` se ha calculado dos veces desde cero. En ejemplos mayores, se recalculan muchos otros valores de `fib`, o *subproblemas*.

Para evitar este inconveniente, podemos resolver el problema mediante programación dinámica, y en particular, utilizando el enfoque de memorización (guardar los valores que ya han sido calculados para utilizarlos posteriormente). Así, rellenaríamos una tabla con los resultados de los distintos subproblemas, para reutilizarlos cuando haga falta en lugar de volver a calcularlos. La tabla resultante sería una tabla unidimensional con los resultados desde 0 hasta n.

Un programa que calculase esto, usando Bottom-up, tendría la siguiente estructura:

```
FUNC Fibonacci (↓n: NATURAL): NATURAL
VARIABLES
    tabla: ARRAY [0..n] DE NATURALES
    i: NATURAL
INICIO
    SI n = 0 ENTONCES
        DEVOLVER 0
    SINOSI n = 1 ENTONCES
        DEVOLVER 1
    SINO
        tabla[0] := 0
        tabla[1] := 1
        PARA i = 2 HASTA n HACER
            tabla[i] := tabla[i-1] + tabla[i-2]
        FINPARA
        DEVOLVER tabla[n]
    FINSI
FIN
```

La función resultante tiene complejidad O(*n*), en lugar de exponencial.

Otro nivel de refinamiento que optimizaría la solución sería quedarnos tan sólo con los dos últimos valores calculados en lugar de toda la tabla, que son realmente los que nos resultan útiles para calcular la solución a los subproblemas.

El mismo problema usando Top-down tendría la siguiente estructura:

```
FUNC Fibonacci (↓n: NATURAL, □tabla: ARRAY [0..n] DE NATURALES): NATURAL
VARIABLES
    i: NATURAL
INICIO
    SI n <= 1 ENTONCES
        devolver n
```

```
    FINSI
    SI tabla[n-1] = -1 ENTONCES
        tabla[n-1] := Fibonacci(n-1, tabla)
    FINSI
    SI tabla[n-2] = -1 ENTONCES
        tabla[n-2] := Fibonacci(n-2, tabla)
    FINSI
    tabla[n] := tabla[n-1] + tabla[n-2]
    devolver tabla[n]
FIN
```

Suponemos que la tabla se introduce por primera vez correctamente inicializada, con todas las posiciones con un valor inválido, como por ejemplo -1, que se distingue por no ser uno de los valores que computa la función.

Coeficientes binomiales

El algoritmo recursivo que calcula los coeficientes binomiales resulta ser de complejidad exponencial por la repetición de los cálculos que realiza. No obstante, es posible diseñar un algoritmo con un tiempo de ejecución de orden O(nk) basado en la idea del Triángulo de Pascal, idea claramente aplicable mediante programación dinámica. Para ello es necesaria la creación de una tabla bidimensional en la que ir almacenando los valores intermedios que se utilizan posteriormente.

La idea recursiva de los coeficientes binomiales es la siguiente:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ si } 0 < k < n$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

La idea para construir la tabla de manera eficiente y sin valores inútiles es la siguiente:

	0	1	2	3	...	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...	...	...	...	...	...		
...	...	...	...	...	...	...	
n-1						C(n-1,k-1)	C(n-1,k)
n							C(n,k)

El siguiente algoritmo memorizado de estrategia Bottom-up tiene complejidad polinómica y va rellenando la tabla de izquierda a derecha y de arriba abajo:

```
FUNC CoeficientesPolinomiales ( ↓n, k: NATURAL): NATURAL
Variables
    tabla: TABLA DE NATURALES
    i, j: NATURAL
Inicio
    PARA i = 0 HASTA n HACER
```

```
            tabla[i][0] := 1
        FINPARA
        PARA i = 1 HASTA n HACER
            tabla[i][1] := i
        FINPARA
        PARA i = 2 HASTA k HACER
            tabla[i][i] := 1
        FINPARA
        PARA i = 3 HASTA n HACER
            PARA j = 2 HASTA i-1 HACER
                SI j <= k ENTONCES
                    tabla[i][j] := tabla[i-1][j-1] + tabla[i-1][j]
                FINSI
            FINPARA
        FINPARA
        devolver tabla[n][k]
    Fin
```

Por supuesto, el problema de los Coeficientes Binomiales también puede resolverse mediante un enfoque Top-down.

El viaje más barato por el río

En un río hay n embarcaderos, en cada uno de los cuales se puede alquilar un bote para ir a otro embarcadero que esté más abajo en el río. Suponemos que no se puede remontar el río. Una tabla de tarifas indica los costes de viajar entre los distintos embarcaderos. Se supone que puede ocurrir que un viaje entre i y j salga más barato haciendo escala en k embarcaderos que yendo directamente.

El problema consistirá en determinar el coste mínimo para un par de embarcaderos.

Vamos a llamar a la tabla de tarifas, T. Así, T[i,j] será el coste de ir del embarcadero i al j. La matriz será triangular superior de orden n, donde n es el número de embarcaderos.

La idea recursiva es que el coste se calcula de la siguiente manera:

C(i, j) = T[i, k] + C(k, j)

A partir de esta idea, podemos elaborar una expresión recurrente para la solución:

```
           0   si i = j
C(i, j)=
            Min(T(i,k) + C(k,j), T(i,j))   si i < k <= j
```

Un algoritmo que resuelve este problema es el siguiente, donde T es la matriz de tarifas, origen y destino los embarcaderos del que se parte y al que se llega respectivamente, y C la matriz en la que almacenaremos los resultados de los costes. La función MenorDeLosCandidatos devuelve el menor coste entre dos puntos, utilizando como base la recurrencia anteriormente expuesta.

```
    FUNC Embarcaderos ( ↓origen, destino, n: NATURAL, ↓T: MATRIZ DE NATURALES): NATURAL
    Variables
        C: MATRIZ DE NATURALES
        i, j: NATURAL
    Inicio
        PARA i = 1 HASTA n HACER
            C[i][i] := 0
```

```
        FINPARA
        PARA i = 1 HASTA n HACER
            PARA j = 1 HASTA n HACER
                C[i][j] := menorDeLosCandidatos(i, j, n, T, C)
            FINPARA
        FINPARA
        devolver C[n] [n]
    Fin
```

```
    FUNC menorDeLosCandidatos ( ↓origen, destino, n: NATURAL, ↓T, C: MATRIZ DE NATURALES): NATURAL
    Variables
        temp: NATURAL
    Inicio
        temp := MAX_NATURAL
        PARA i = origen+1 HASTA n HACER
            temp := min(temp, T[origen][i] + C[i][destino])
        FINPARA
        devolver temp
    Fin
```

Ejercicios resueltos con programación dinámica

- Ejecución de n tareas en tiempo mínimo en un sistema de dos procesadores A y B
- Programas en disco
- Problema de los sellos con programación dinámica
- Problema de la mochila con programación dinámica
- Problema del producto de una secuencia de matrices con programación dinámica
- Problema de las monedas con programación dinámica [1]
- Camino de coste mínimo entre dos nodos de un grafo dirigido
- Problema de la división de peso
- Problema de las vacas con programación dinámica
- Problema del Cambio de Palabra Programación Dinámica en JAVA [2]
- Problema de buscar la subsecuencia común más larga entre dos cadenas

Enlaces externos

- Investigación Operativa - El Sitio de Investigación Operativa en Español del Ing. Santiago Javez Valladares-Perú [3]
- Arquimedex - Investigación de Operaciones en la práctica [4]

Referencias

- Xumari, G.L. Introduction to dynamic programming. Wilwy & Sons Inc., New York. 1967.

Referencias

[1] http://es.wikibooks.org/wiki/Programaci%C3%B3n_din%C3%A1mica/Problema_de_las_monedas_con_programaci%C3%B3n_din%C3%A1mica

[2] http://es.wikibooks.org/wiki/Problema_del_cambio_de_palabra_%28programaci%C3%B3n_din%C3%A1mica_en_Java%29

[3] http://www.invope.com

[4] http://www.arquimedex.com

Fuentes y contribuyentes del artículo

Programación dinámica *Fuente*: http://es.wikipedia.org/w/index.php?oldid=63555745 *Contribuyentes*: Airen es, AlbertMA, Alfredobi, Alhen, Alwar, Andreasmperu, AngelHerraez, Angelherriv, AntonioLG, Cibi3d, Cuesteras, Ectofolio, Edub, El+level, Esf3ra, Especiales, Furti, GermanX, Gothmog, Graimito, Guynplaine, HanPritcher, Highsoftx980, Ignaciord, JRGL, Jkbw, Lex.mercurio, Manz, Migueli Jordan, Moonrak, Mr freeze360, Nacho rd, Nicolasdiaz, Oblongo, PabloCastellano, Poco a poco, Ricki.rg, Sgmonda, SuperBraulio13, Technopat, Tessier, 91 ediciones anónimas

Licencia