Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Agosto 2018

Soluciones recursivas a problemas

- Muchos algoritmos de utilidad son recursivos: para resolver un problema, se utilizan las soluciones a **subproblemas** fuertemente relacionados.
- En estos algoritmos:
  - Se **divide** el problema en “varios” subproblemas
    - 0 (casos base)
    - 1 o más (casos recursivos)
  - Estos se resuelven utilizando el mismo algoritmo
  - Se **utilizan** esas subsoluciones para resolver el problema original
- Ejemplo: Divide and Conquer (Algoritmos II)

¿En qué consiste programación dinámica?

- Técnica de solución de problemas **recursiva** (como D&C)
- Diferencia esencial:
  - Se aplica D&C cuando los subproblemas que se resuelven **son independientes entre sí**
  - Se aplica programación dinámica cuando los subproblemas **no son independientes entre sí**
- Cuando hay **superposición de subproblemas**, un algoritmo de Divide and Conquer realizaría **el mismo** trabajo múltiples veces
- Lo anterior ocurre ya que la solución a un mismo subproblema puede ser **recalculada** muchas veces si se la reutiliza como parte de varios subproblemas más grandes.

¿En qué consiste programación dinámica? (2)

- Solución: usar programación dinámica.
- PD propone **almacenar** las soluciones a subproblemas ya calculados
- Almacenar las soluciones permite calcularlas **una sola vez**, y luego leer el valor ya calculado cuando se lo vuelve a necesitar.
- PD = D&C + **caching**

Problemas de optimización

- Uno de los usos más importantes es en problemas de **optimización**: Se busca una solución que maximice un cierto puntaje u objetivo, en un espacio de soluciones posibles
- Para poder aplicar D&C o PD (técnicas recursivas) debe cumplirse el **principio del óptimo**:
  - **Las partes de una solución óptima** a un problema, deben ser **soluciones óptimas de los correspondientes subproblemas**
  - Permite obtener una solución óptima al problema original a partir de soluciones óptimas de los subproblemas
- Ejemplo donde **SI** se cumple:
  Ordenar con quicksort
- Ejemplo donde **NO** se cumple:
  Calcular un camino máximo simple entre u y v en un grafo

El esquema general (según el libro de CLRS)

Los algoritmos de programación dinámica se pueden organizar típicamente en 4 pasos que responden al siguiente esquema general:

1. Caracterizar la **estructura** de una solución (óptima)
2. Definir **recursivamente** el valor de una solución (óptima)
3. Computar el **valor** de una solución (óptima). Se calcula de manera *bottom-up* (CLRS) o *top-down* ("memoization")
4. (A veces) Construir una **solución** óptima a partir de la información obtenida en el paso 3

El paso 4 no siempre es necesario, ya que si solamente nos interesa el valor o puntaje de una solución óptima, pero no la solución en sí, muchas veces podemos evitar este paso.
Además, en problemas que no son de optimización, generalmente el paso 4 no aplica (Fibonacci, calcular **cantidad** de soluciones, etc)

Problema “de las monedas”

(Parte principal del ejercicio 3.10, práctica 3):

Dadas varias monedas, de valores posiblemente distintos, dar un cierto vuelto utilizando la mínima cantidad de ellas posible.

Más formalmente: Dadas n monedas, cuyos valores están dados por $v_1, v_2, \cdots, v_n \in \mathbb{Z}_{>0}$, y un $T \in \mathbb{Z}_{>0}$, se debe dar un $I \subseteq \{1, 2, 3, \cdots, n\}$ tal que $\sum_{i \in I} v_i = T$, y $|I|$ sea mínimo.

Estructura de una solución

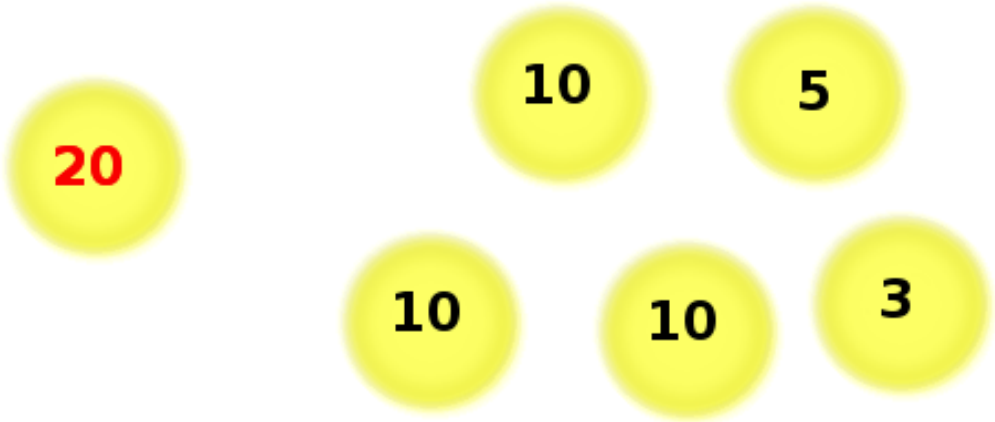
- Nuestro objetivo es llegar a un planteo recursivo del problema.
- Antes de escribir la fórmula de la solución, tenemos que relacionar la estructura que tiene la solución a un problema, con las soluciones a problemas más chicos.
- Algunas ideas generales para este paso:
  - Cuando hay muchos elementos “disponibles” en el problema (ej: monedas), a veces conviene enfocar en uno particular
  - Es muy común realizar una división en casos, que abarquen todas las opciones posibles (ej: o usamos esa moneda, o no la usamos)
  - En este paso, muchas veces ayuda hacer dibujos y ejemplos.

Estructura de una solución (cont)

- Ejemplo: $T = 58$, $v = \{10, 25, 10, 10, 5, 3, 5, 20\}$
- Hay únicamente dos posibilidades:
  - O bien utilizamos la última moneda (la de 20)
  - O bien no la utilizamos
- Analizamos cómo será S, **una solución óptima** en cada caso:
- Si $n \notin S$...
  - Tendrá que sumar $T = 58$ con **las demás** monedas.
  - Es decir, $\sum_{i \in S} v_i = 58$ con $S \subseteq \{1, 2, 3, \cdots, n-1\}$
  - De todas las maneras de formar $T = 58$ con $\{v_1, v_2, v_3, \cdots, v_{n-1}\}$, la cantidad mínima de monedas posibles tendrá que ser $|S|$
  - Si no fuera así, habría un $S' \subseteq \{1, 2, 3, \cdots, n-1\}$ con $\sum_{i \in S'} v_i = 58$ y $|S'| < |S|$. Pero tal S' no puede existir, pues sería una solución al problema original mejor que S.
  - Es decir, S **es solución óptima para el subproblema** de formar $T = 58$ con las monedas $\{1, 2, 3, \cdots, n-1\}$
- Si $n \in S$...

Estructura de una solución (cont...)

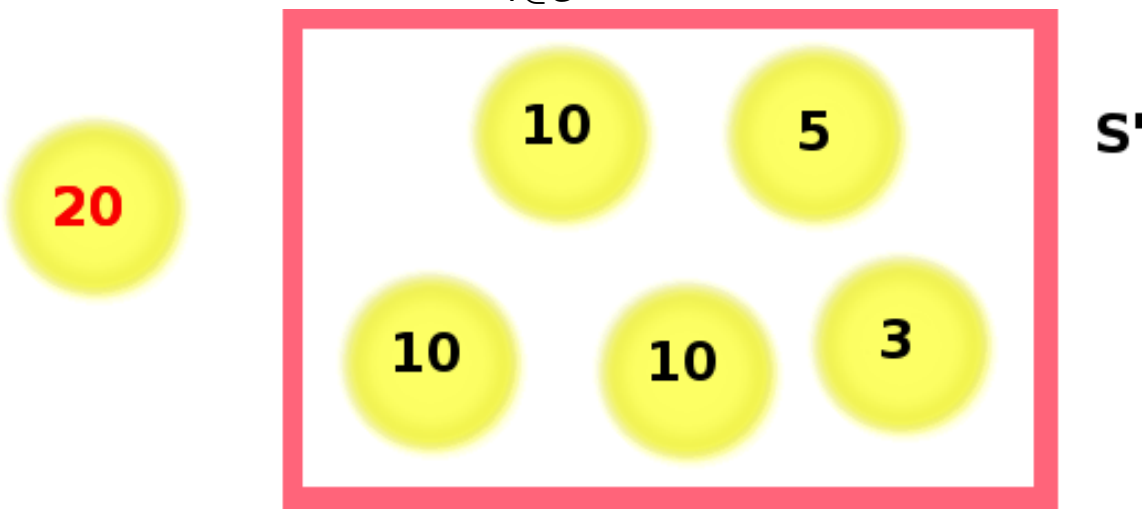
- Si $n \in S$...
  - Por ejemplo, S tendría la pinta del dibujo:



  - Es decir, S contiene algún subconjunto de monedas, que sabemos que contiene la moneda de valor $v_n = 20$
  - ¿Qué podemos decir sobre **las demás** monedas en S?

Estructura de una solución (cont.....)

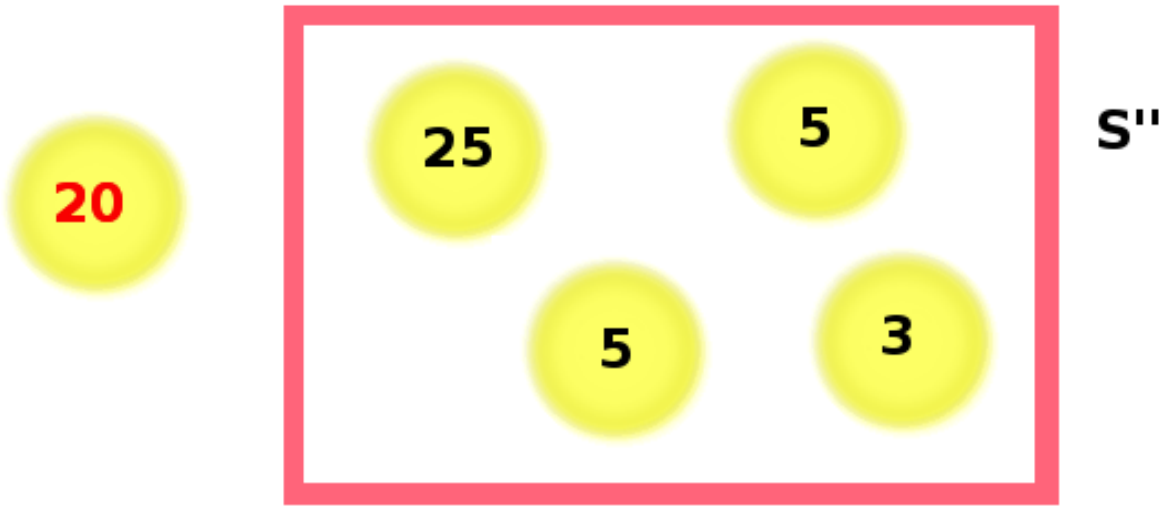
- Si $n \in S$...
  - $S = \{n\} \cup S'$, con $\sum_{i \in S'} v_i = 38$, $S' \subseteq \{1, 2, 3, \cdots, n-1\}$



  - De manera similar a lo que ocurría antes, de todas las maneras de formar 38 con $\{v_1, v_2, v_3, \cdots, v_{n-1}\}$, la cantidad mínima de monedas posibles tendrá que ser $|S'|$
  - Si no fuera así, habría un $S'' \subseteq \{1, 2, 3, \cdots, n-1\}$ con $\sum_{i \in S''} v_i = 38$ y $|S''| < |S'|$
  - Pero entonces, cambiando S' por S'' en nuestra solución...

Estructura de una solución (cont.....)

- Si $n \in S$...
  - ¡Obtendríamos que $\{n\} \cup S''$ es mejor solución que S!



  - Es decir, $S = \{n\} \cup S'$, donde S' **es solución óptima para el subproblema** de formar 38 con las monedas $\{1, 2, 3, \cdots, n-1\}$

Estructura de una solución (resumiendo)

- Como resultado del análisis anterior, hemos caracterizado una solución óptima S para el problema original:
  - Si $n \notin S$, entonces la mejor solución es $S = S_1$, siendo S_1 solución óptima para $\{v_1, v_2, v_3, \cdots, v_{n-1}\}$ dando el mismo vuelto T
  - Si $n \in S$, entonces la mejor solución es $S = \{n\} \cup S_2$, siendo S_2 solución óptima para $\{v_1, v_2, v_3, \cdots, v_{n-1}\}$ dando un vuelto $T - v_n$
- Por lo tanto, como esas son las únicas dos posibilidades, S deberá ser la que use menos monedas entre S_1 y $\{n\} \cup S_2$
- **Usando esta caracterización**, podremos elaborar una recursión que calcule la cantidad óptima de monedas necesarias

Fórmula recursiva

- Definimos:

$$f(i,t) \quad = \quad \begin{array}{c} \text{“mínima cantidad de monedas necesarias} \\ \text{para dar un vuelto } t \text{ con las monedas } \{1,2,\cdots,i\}, \\ \text{o } +\infty \text{ si es imposible”} \end{array}$$

- **Es muy importante** escribir esta definición de f “en palabras”: Si no sabemos **qué** queremos calcular con f, es imposible saber si la recursión que estamos dando es correcta o no.
- Como nuestro objetivo final es resolver el problema, también es importante decir **cómo se usa** f para obtener la respuesta al problema.
- En este caso, la cantidad óptima de monedas necesarias para dar el vuelto pedido es simplemente $f(n, T)$:
  - Por la definición de f, esa es la cantidad mínima necesaria para dar un vuelto T usando las monedas $\{1,2,\cdots,n\}$
  - Como esas son **todas** las monedas disponibles, $f(n, T)$ resulta ser simplemente la mínima cantidad de monedas necesarias, sin restringir cuáles

Fórmula recursiva (cont)

$$f(i,t) = \begin{cases} 0 & \text{si } i = 0, t = 0 \\ +\infty & \text{si } i = 0, t \neq 0 \\ \underbrace{f(i-1,t)}_{S_1} & \text{si } i > 0, v_i > t \\ min\left(\underbrace{f(i-1,t)}_{S_1}, \underbrace{1 + f(i-1,t-v_i)}_{\{i\} \cup S_2}\right) & \text{si } i > 0, v_i \leq t \end{cases}$$

- Al escribir la fórmula, hemos agregado al análisis anterior:
  - Los casos base (correspondientes a $i = 0$: no hay monedas)
  - El caso en el que $v_i > t$, y por lo tanto no es necesario considerar la opción de utilizar la moneda i, pues es imposible usarla para pagar el vuelto t

Algoritmo top-down

```
1  // d: Diccionario global de soluciones ya calculadas
2  monedas(i, t):
3      IF (i,t) not in d THEN
4          IF i=0 and t=0 THEN
5              d[i,t] = 0
6          IF i=0 and t!=0 THEN
7              d[i,t] = +INF
8          IF i>0 and v[i]>t THEN
9              d[i,t] = monedas(i-1, t)
10         IF i>0 and v[i]<=t THEN
11             d[i,t] = min(monedas(i-1,t), 1 + monedas(i-1, t-v[i]))
12     return d[i,t]
```

El diccionario `d` podría ser una simple matriz (con un valor especial como -1 para indicar un elemento no calculado), por lo que tiene acceso $O(1)$.

Notar que gracias a que guardamos las respuestas, **nunca se calcula un mismo subproblema más de una vez.**

Algoritmo bottom-up

```
1  monedas(v, n, T): // v indexado desde 1
2      f <- matrizDeEnteros[0..n , 0..T]
3      f[0,0] = 0
4      FOR t <- 1 TO T DO
5          f[0, t] = +INF
6      FOR i <- 1 TO n DO
7          FOR t <- 0 TO T DO
8              IF v[i] > t THEN
9                  f[i,t] = f[i-1, t]
10             ELSE
11                 f[i,t] = min(f[i-1,t], 1 + f[i-1, t - v[i])
12     RETURN f[n, T]
```

La complejidad del algoritmo resultante es $O(nT)$, tanto espacial como temporal. Se puede bajar la complejidad espacial a $O(T)$ (guardando sólo dos filas de la matriz f: cada una depende sólo de la anterior)
¿Cuál es la complejidad de la versión top-down?

Comparación general

Top-Down (memoization)	Bottom-Up
☺ Fácil de programar	☹ Más difícil: hay que pensar el orden de llenado de la tabla
☺ ☺ **Calcula exactamente los subproblemas necesarios**	☹ Calcula todos los subproblemas
☹ No permite ahorrar memoria olvidando soluciones	☺ ☺ **Suele permitir ahorrar memoria**
(usualmente) ☹ Mucho peor factor constante	(usualmente) ☺ Mucho mejor factor constante

Obtener la solución S

- Lo anterior es suficiente si solamente necesitamos calcular la cantidad mínima de monedas necesarias.
- ¿Pero cómo obtenemos la solución S eficientemente?
- Para cada subproblema $f(i, t)$, la solución óptima usa la moneda i o no la usa, según con cuál de las dos opciones se alcance el mínimo.
- Podemos almacenar junto a cada valor $f(i, t)$, un booleano que indique con cuál de las dos opciones se obtuvo el mínimo (o bien deducirlo a partir de los valores $f(i, t)$ almacenados)
- En este caso, ya no podemos usar la misma técnica para bajar la complejidad espacial del problema, pues necesitamos mantener la matriz en memoria para poder obtener la solución

Obtener la solución S: ejemplo

- $T = 58$, $n = 8$, $v = \{10, 25, 10, 10, 5, 3, 5, 20\}$
- $f(8, 58) = 1 + f(7, 38)$, así que usamos la moneda $v_8 = 20$
- $f(7, 38) = f(6, 38)$
- $f(6, 38) = 1 + f(5, 35)$, así que usamos la moneda $v_6 = 3$
- $f(5, 35) = f(4, 35) = f(3, 35)$
- $f(3, 35) = 1 + f(2, 25)$, así que usamos la moneda $v_3 = 10$
- $f(2, 25) = 1 + f(1, 0)$, así que usamos la moneda $v_2 = 25$
- $f(1, 0) = f(0, 0) = 0$
- De todo lo anterior, resultó $S = \{2, 3, 6, 8\}$

Referencias

- Introduction to Algorithms, 2nd Edition. MIT Press. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Página 323: 15 Dynamic Programming