Programación Dinámica

[1] Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Algoritmos y Estructuras de Datos III

¿Qué es la Programación dinámica?

Definición

La programación dinámica es una técnica para resolver problemas, que consiste en dividir una instancia de un problema en subproblemas, resolviendo una sóla vez cada instancia del problema para poder obtener la respuesta de una instancia del mismo eficientemente.

- Los subproblemas se pueden solapar. Es decir, dos subproblemas pueden tener subsubproblemas en común.
- La idea clave es calcular cada subproblema una sóla vez y usar la respuesta obtenida en todos los subproblemas que requieren de ella.
- Esto es lo que hace que las soluciones sean muy eficientes.

Cosas a tener en cuenta

¿Qué es un subproblema?

- Depende del problema. En general es una instancia más chica del problema original que puede tener algunas restricciones.

¿Cómo me conviene elegir mis subproblemas?

- Depende del problema. No hay una técnica que funcione siempre. Lo mejor es practicar mucho y tener varios ejemplos a mano.

¿Qué subsubproblemas necesito para resolver un subproblema?
¿Como combino los resultados de ellos?

- Depende del problema.

Programación dinámica es una técnica, no un algoritmo. Responder estas preguntas para un problema en particular suele ser lo más importante de la solución.

Cuidado

No siempre la respuesta al problema original coincide con un subproblema.

A los bifes

Problema 1

En la guarida del tesoro hay una hilera de n cofres con monedas. El i-ésimo cofre tiene m_i monedas. Los cofres están protegidos con un sistema de alarmas. Sin embargo, como se produjeron muchas falsas alarmas últimamente, se decidió que no se activen las alarmas cuando se abre un sólo cofre sino cuando se abren al menos dos cofres consecutivos de la hilera. Los dueños sospechan que un ladrón abrirá en su apuro varios cofres seguidos y será fácil detectarlos. Sin embargo, se dieron cuenta que el sistema no es infalible y un ladrón con la información de como funciona el sistema se podría robar muchas monedas. Nos piden que calculemos cual es la máxima cantidad de monedas que se pueden llevar sin activar las alarmas.

Idea informal

Vamos a resolverlo con programación dinámica:

- Definimos un subproblema como la máxima cantidad de monedas que se pueden llevar si roban sólo de los primeros k cofres. Llamamos S_k a esta cantidad.
- Para resolver S_k, notamos que tenemos dos opciones, o abrimos el cofre k o no. En el primer caso el máximo es $S_{k-2} + m_k$. En el segundo es S_{k-1}.
- Nos queda la fórmula $S_k = max(S_{k-2} + m_k, S_{k-1})$.
- La respuesta al problema es S_n.

Faltan los casos base

$S_0 = 0$ y $S_1 = m_1$.

Implementando una dinámica (Versión iterativa)

Una forma de implementar soluciones con programación dinámica es de manera iterativa o bottom-up:

- Empiezo resolviendo los subproblemas más simples y voy avanzando hacia los más complejos.
- Cada vez que resuelvo un subproblema, guardo su respuesta en alguna estructura.
- Para resolver un subproblema busco en mi estructura las respuestas a los subsubproblemas que necesito.

A esta técnica de ir "llenando una tablita" se la llama *tabulación*.

Código del problema (Versión iterativa)

```
calcularRoboMaximo(n, m[]) {
    S[0] = 0;
    S[1] = m[1];
    for k = 2 to n {
        S[k] = max(S[k-2] + m[k], S[k-1]);
    }
    return S[n];
}
```

El algoritmo es $O(n)$.

Ahorrando memoria

Una observación que suele ser útil en estos problemas es notar que no necesitamos tener siempre guardados todos los subproblemas más chicos para calcular los que faltan:

- Para calcular S_k sólo necesito S_{k-1} y S_{k-2}.
- De hecho para calcular todos los subproblemas desde S_k a S_n sólo necesito saber los resultados de S_{k-2} en adelante.
- Me puedo olvidar de los resultados que ya no necesite.
- Me basta recordar sólo los dos resultados anteriores.

Al calcular S_k llamo A_0, A_1, A_2 a los resultados de S_k, S_{k-1}, S_{k-2} respectivamente.

Me olvidé del título

```
calcularRoboMaximo(n, m[]) {
    A[1] = 0;
    A[0] = m[1];
    for k = 2 to n {
        A[2] = A[1];
        A[1] = A[0];
        // Avanzo una posicion
        A[0] = max(A[2] + m[k], A[1]);
    }
    return A[0];
}
```

Ahora el algoritmo consume $O(1)$ de memoria.

Implementando una dinámica (Versión recursiva

Otra forma de implementar una dinámica es por medio de una recursión (top-down):

- Llamamos a una función que resuelva nuestro problema.
- La función para resolver un subproblema llama recusivamente a los subsubproblemas que necesita.
- Cuando llegamos a un caso base simplemente devolvemos su respuesta.

¡Como en un backtracking!

Código del problema (Versión recursiva)

```
calcularRoboMaximo(k, m[]) {
    if k == 0 {
        res = 0;
    }
    else if k == 1 {
        res = m[1];
    }
    else {
        res = max(cRM(k-2, m) + m[k], cRM(k-1, m));
    }
    return res;
}
```

Basta llamar a *calcularRoboMaximo*(n, m).

¿Cuál es la complejidad?

Así como esta este código es exponencial en n.

Problema del código

El problema es que estamos llamando muchas veces a la misma función con los mismos parámetros, va a devolver siempre el mismo valor.

Solución

Guardamos la respuesta a los subproblemas que ya calculamos y antes de resolver un llamado recursivo nos fijamos si no tenemos ya la respuesta guardada.

A esta técnica se la llama *memoización* (sin la 'r').
A la forma de implementar una dinámica de esta manera se le dice recursiva memoizada.

Código del problema (Versión final)

```
calcularRoboMaximo(k, m[]) {
+    if calculeRM[k] {
+        return RM[k];
+    }

     if k == 0 {
         res = 0;
     }
     else if k == 1 {
          res = m[1];
     }
     else {
          res = max(cRM(k-2, m) + m[k], cRM(k-1, m));
     }

+    calculeRM[k] = true;
+    RM[k] = res;
     return res;
}
```

Segundo round

Problema 2

Se tiene una fila de n enteros positivos. Sea a_i el i-ésimo número de la fila. Se quiere pintar algunos de ellos de rojo y el resto de azul, de manera que la diferencia entre la suma de los números rojos y la suma de los números azules sea mínima.
Dar una forma de pintarlos que minimice esta diferencia si se sabe que la suma de los números está acotada por m.

Idea

Ya no nos alcanza con un subproblema tan simple:

- Dado un $0 \leq i \leq n$ queremos ver si podemos hacer que los números rojos sumen $0 \leq r \leq m$ pintando los primeros i. Llamamos $S(i, r)$ al resultado de este subproblema.
- Podemos elegir pintar el i-ésimo o no.
- Si lo hacemos, nos queda sumar $r - a_i$ con los primeros $i - 1$ ($S(i - 1, r - a_i)$).
- Si no lo hacemos, nos queda sumar r con los primeros $i - 1$ ($S(i - 1, r)$).
- Luego, $S(i, r) = S(i - 1, r - a_i)$ or $S(i - 1, r)$
- Finalmente, recorremos para todo j los $S(n, j)$ y nos quedamos con el que de True más cercano a la mitad de la suma de todos los números.

Pintarte la cara

Ya podemos obtener la mínima diferencia pero... ¿Cómo pintamos los números para obtener esta diferencia?

- Recorremos la tabla de atras para adelante.
- Empezamos por $S(n, j)$ para el j óptimo.
- Dado que estamos en $S(k, l)$, si hay una solución pintando a_k de rojo, $S(k-1, l-a_k)$ debería ser True. Si lo es, pintamos a_k de rojo y nos movemos a $S(k-1, l-a_k)$
- si es False, entonces sólo hay solución pintando a_k de azul. Lo pintamos y pasamos a mirar $S(k-1, l)$.
- Repetimos esto hasta llegar a $S(0, 0)$.

Código del problema

```
pintarConMinDif(n, a[]) {
    S[0][0] = true;
    S[0][j] = false para j > 0;
    for i = 1 to n {
        for r = 0 to m {
            S[i][r] = S[i-1][r];
            if r >= a_i {
                S[i][r] |= S[i-1][r-a_i];
            }
        }
    }
    bestR = 0;
    sumTot = sum(a);
    for r = 1 to m {
        if S[n][r] and |sumTot - 2*bestR| > |sumTot - 2*r| {
            bestR = r;
        }
    }
    ...
}
```

Código que pinta

```
pintarConMinDif(n, a[]) {
    ...
    int faltaSumar = bestR;
    for i = n to 1 {
        if S[i-1][s] {
            color[i] = 'azul';
        }
        else {
            color[i] = 'rojo';
                faltaSumar -= a[i];
            }
    }
}
```

Tiene complejidad $O(nm)$.

No hay dos sin tres

Problema 3

Dada una palabra (string) de n caracteres, se quiere calcular la mínima cantidad de letras que se deber borrar de la palabra para que lo que quede sea un palíndromo.
Una versión un poco más difícil:
https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2399

Idea

Si p es la palabra, notamos $p(i, j)$ a la subpalabra de p que empieza en el caractér i y termina en el j.

- Podemos pensar que un subproblema es calcular la mínima cantidad de letras a borrar para que $p(i, j)$ sea un palíndromo. Llamamos $S(i, j)$ al resultado de este subproblema.
- Se que el palíndromo que me quede va a empezar con una letra en la posición k y terminar en la posición l con $i \leq k \leq l \leq j$ (con p[k] = p[l]).
- La respuesta a $S(i, j)$ va a ser $S(k + 1, l - 1) + (k - i) + (j - l)$.
- Pruebo con todos los k y l posibles y me quedó con el mínimo de todos.

Optimizando una dinámica

Si bien la solución que queda es polinomial, se puede mejorar bastante todavía.
La idea es reducir a un subproblema lo más rápido posible:

- En lugar de preguntar donde está el primer caractér de la solución, pregunto si es o no el primero.
- Si no lo es, la respuesta es simplemente $S(i+1, j)+1$.
- Análogamente, si el último no es parte de la solución, la respuesta es $S(i, j-1)+1$.
- Si los dos extremos están en la solución es $S(i+1, j-1)$.
- $S(i, j)$ es el mínimo de estos tres casos.

El código

```
minimizarPalindromo(n, p) {
    S[i][j] = 0 si i > j
    for i = 0 to n-1 {
        for j = i to n-1 {
            S[i][j] = min(S[i+1][j] + 1, S[i][j-1] + 1);
            if p[i] == p[j] {
                S[i][j] = min(S[i][j], S[i+1][j-1]);
            }
        }
    }
    return S[0][n-1];
}
```

Profe, ¡no me anda!

Ese código tiene un problema:

- Para calcular $S(0, n-1)$ necesitamos $S(1, n-1)$.
- Como estamos recorriendo los subproblemas $S(1, n-1)$ lo calculamos despues que $S(0, n-1)$.
- Hay que recorrerlos en orden de longitud del substring $(j - i)$.

Notemos para calcular la respuesta para cierta longitud del substring sólo necesitamos los de longitud a lo sumo 2 caracteres menor. Con esto podríamos reducir la complejidad espacial a $O(n)$ (no lo vamos a hacer).

El código (arreglado)

```
minimizarPalindromo(n, p) {
    S[i][j] = 0 si i > j
    for d = 0 to n-1 {
        for i = 0 to n-1-d {
            j = i + d;
            S[i][j] = min(S[i+1][j] + 1, S[i][j-1] + 1);
            if p[i] == p[j] {
                S[i][j] = min(S[i][j], S[i+1][j-1]);
            }
        }
    }
    return S[0][n-1];
}
```

Last but not least

Travelling Salesman Problem (TSP)

Un comerciante tiene que entregar objetos que vendió durante el día en las casas de n personas distintas.
Empieza en la casa de la persona número 1 y sabe que para ir de la casa de la persona i a la persona j tiene que caminar $d(i, j)$ metros.
Debe pasar por cada casa una vez a entregar los pedidos de esa casa. Puede finalizar el recorrido en cualquiera de las n casas.
¿Cuál es la mínima distancia que debe recorrer para entregar todos los pedidos?

Idea

Nuestro subproblema ahora va a ser más complejo:

- Subproblema: Dado que me falta recorrer un conjunto C de casas ($C \subset \{1, 2, \ldots, n\}$) y actualmente estoy en la casa i ¿Cuánto es lo mínimo que tengo que caminar para pasar por todas las casas que me faltan?
- Llamamos $S(C, i)$ a este subproblema.
- Si la próxima casa a visitar es la j, lo mínimo será $d(i, j) + S(C - \{j\}, j)$.
- Luego la solución de $S(C, i)$ es el mínimo de esto para todo $j \in C$.

Detalles de implementación

Algunos problemas que surgen al tratar de implementar esto:

- Los parámetros del subproblema son muy complejos. No es tan claro como hacer una tabla para las respuestas de los subproblemas.
- No es tan claro como hacer para recorrer los subproblemas en orden.
- En estos casos lo mejor es hacerlo de manera recusiva.
- Para mantener los resultados de los subproblemas ya calculados podemos usar un diccionario.
- Dependiendo de la implementación del diccionario puede afectar a la complejidad del algoritmo.

¡A programar!

Intenten implementar un algoritmo que use programación dinámica que resuelva TSP.
La complejidad es $O(n^2 2^n)$, sigue siendo exponencial, pero es bastante mejor que la fuerza bruta en $O(n!)$.

Resumen (versión iterativa)

Puntos a tener en cuenta en la versión iterativa:

- Hay que poder saber cuales son todos los subproblemas posibles de antemano.
- Hay que establecer algún orden para recorrerlos de manera que siempre tengamos ya calculados los que necesitemos.
- Suele ser muy fácil de implementar y el código queda de pocas líneas.
- El análisis de complejidad suele ser casi inmediato.
- Es más eficiente que la versión recursiva en términos prácticos.
- Me puedo olvidar de los subproblemas que ya no necesito para ahorrar memoria.

Resumen (versión recursiva)

Puntos a tener en cuenta en la versión recursiva:

- Hay que asegurarse que el programa siempre termina.
- Sólo calcula los subproblemas estrictamente necesarios para calcular el problema original, por los que nos puede ahorrar mucho tiempo en el caso donde no haga falta gran parte de ellos.
- No le interesa la estructura de los subproblemas, ni saber cuales van a ser todos, ni que se recorran en un orden particular.
- El análisis de complejidad suele ser más difícil.
- Hay que tener cuidado con los lenguajes que limitan el stack para recursión, ya que resolver una instancia que puede generar llamadas recursivas muy profundas.

Fin

¿Preguntas?