Explicación Práctica: Candy

Santiago Alessandri

October 2, 2011

Candy

In this contest a random number of boxes containing candies are disposed in M rows with N columns each (so, there are a total of *MxN* boxes). Each box has a number indicating how many candies it contains.

The contestant can pick a box (any one) and get all the candies it contains. But there is a catch (there is always a catch): when choosing a box, all the boxes from the rows immediately above and immediately below are emptied, as well as the box to the left and the box to the right of the chosen box. The contestant continues to pick a box until there are no candies left.

Can you maximize the number of candies he can pick?

Cuestiones Basicas

¿Cuál es el objetivo del problema?

El objetivo es maximizar la suma de los elementos elegidos cumpliendo con las restricciones planteadas en el problema.

¿Como puede resolverse?

Por las características del problema, esto se resuelve a través de *Dynamic Programming*. Y como todo problema de programación dinámica, la dificultad reside en como hacer el planteo de búsqueda de la solución.

Planteo General

No es necesario resolver todo el problema junto, es decir que se puede computar primero qué valores de cada fila es conveniente tomar y luego decidir qué filas elegir para maximizar el producto.

Es importante notar que no son necesarios dos métodos distintos, uno para resolver cada fila internamente y otro para saber qué filas elegir.

¿Por qué?

Si se presta atención, la condición entre elementos de la misma fila es la misma que entre columnas. La restricción es **no** elegir dos elementos contiguos al sumar.

Planteo General (cont.)

Supongamos que tenemos la funcion $f(array)$ nos devuelve la mayor suma de elementos tal que ninguno de los sumados es colindante.

Si partimos de esta matriz:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{matrix} f(F_1) \\ f(F_2) \\ f(F_3) \end{matrix} \rightarrow \begin{bmatrix} max_{F_1} \\ max_{F_2} \\ max_{F_3} \end{bmatrix}$$ Arreglo de máximos de filas.

$$f\left(\begin{bmatrix} max_{F_1} \\ max_{F_2} \\ max_{F_3} \end{bmatrix}\right) = max_{Matrix} \rightarrow$$ es el resultado al problema.

¿Cómo hacemos la función f?

Abstracción de la función

En este enfoque se parte de todo el problema y lo soluciona dividiendo el mismo en subproblemas de menor tamaño más fáciles de resolver.

Puntos a tener en cuenta:

- Si hay un solo elemento en el arreglo entonces la suma máxima es dicho elemento.
- Si el tamaño del arreglo es dos entonces la suma máxima es el mayor de ambos.
- En caso que haya mas de tres elementos entonces la suma máxima será la mayor entre la suma máxima a partir del siguiente elemento o el resultado de sumar el primer elemento mas la suma máxima a partir del tercero.

Definición de la función

Aplicando lo mencionado anteriormente podemos definir la función f de la siguiente forma:

$$f(x) = \begin{cases} x[0] & \text{if } size(x) = 1 \\ max(x[0], x[1]) & \text{if } size(x) = 2 \\ max(x[0] + f(x[2:]), f(x[1:])) & \text{if } size(x) > 2 \end{cases}$$

Código de la función en C++

```
uint64_t f(uint32_t *array, uint32_t size) {
  switch (size) {
    case 1:
      return array[0];
    case 2:
      return max(array[0], array[1]);
    default:
      uint64_t tmp = array[0] + f(array+2, size-2);
      return max(tmp, f(array+1, size-1));
  }
}
```

Problemas con esta solución

Utilizando esta función nos da el resultado correcto pero tenemos un serio problema:

- Es de $O(2^n)$. Por lo que se va de tiempo rotundamente.

¿Cómo podemos solucionar esto sin cambiar la estrategia de solución?

MEMOIZATION!

C++ code with Memoization

```
uint64_t f(uint32_t* array, uint32_t index,
           uint32_t size, map<uint32_t, uint64_t>& memo) {
    if (index == size - 1)
      return array[index];
    if (index == size - 2)
      return max(array[index], array[index + 1]);
    if (memo.find(index) != memo.end())
      return memo[index];
    uint64_t tmp = array[index] + f(array, index+2, size, memo);
    tmp = max(tmp, f(array, index + 1, size, memo));
    memo[index] = tmp;
    return tmp;
}
```

Este código ya es aceptado por el juez.

Abstracción de la función

A la inversa del enfoque *Top-Down* en este construiremos el resultado incrementalmente.
Justamente hacemos lo "opuesto" al método anterior:

- Para el primer elemento del arreglo, la suma máxima es el mismo.
- Para el segundo, es el mayor entre el primer elemento y sí mismo.
- Del tercer elemento en adelante, la suma máxima del arreglo hasta esa posición es: la suma máxima hasta el elemento anterior ó la suma entre él y la suma máxima del arreglo hasta el previo al anterior.

Abstracción de la función (cont.)

Se debe calcular el valor de la suma máxima para el arreglo a partir del primero e ir avanzando de a una posición.

De esta forma, para cada elemento que vayamos procesando, tendremos todos los valores necesarios para calcular la suma máxima hasta dicha posición ya calculados.

Ventajas:

- No es necesario un diccionario para mantener los valores intermedios, quedan en el mismo arreglo. $\rightarrow$ menos memoria.
- No existe un stack de recursión. El proceso es iterativo.
- Tiene una mejor escalabilidad.
- Es más eficiente. En este problema la solución fue 2.6 veces más rápida.

Código de la función en C++

```
uint32_t f(uint32_t* r, uint32_t size) {
    if (size > 1) {
        r[1] = max(r[0], r[1]);
    }
    for (uint32_t i(2); i < size; ++i) {
        r[i] = max(r[i-1], r[i] + r[i-2]);
    }
    return r[size-1];
}
```

Conclusiones

- Plantear el problema de forma sencilla simplifica la construcción de una solución al mismo. $\rightarrow$ *Divide & conquer*
- Hacer eficiente la resolución de cada una de las partes es la clave en este problema.
- A veces, la técnica de **memoization** funciona. La cual es mucho más fácil de aplicar que encontrar la dinámica iterativa.

El código fuente de las distintas opciones así como la presenteación, pueden descargarlo desde:

- `http://wiki.san-ss.com.ar/acm-icpc-4212`