Problemas, Algoritmos, y Programación
Trabajo Práctico 1

Jonás Levy Alfie   Denys Bulavka   Martín Fixman
Facundo Gutiérrez

Segundo Cuatrimestre 2016

Índice

1. La Tienda de Apu

1.1. Soluciones teóricas al problema

1.1.1. Backtracking naïve

Una posible manera de solucionar el problema es usar backtracking normal. En particular, se puede definir una función $\mathcal{B} : \{Tamano\} \times precio \times precio \to precio$ que, dado un conjunto de tamaños de rosquillas $D_1, \ldots, D_N$, un precio P, y el precio ya pagado p, diga cuanta es la mayor cantidad de plata que puede gastar comprando las rosquillas en los argumentos para que en total gaste menos de P pesos.

$$\mathcal{B}(\varnothing, P, p) = \begin{cases} p & \text{si } p \leq P \\ 0 & \text{si } p > P \end{cases}$$

$$\mathcal{B}\left(\{D_1, D_2, \ldots, D_N\}, P, p\right) = \text{máx}\begin{pmatrix} \mathcal{B}\left(\{D_2, \ldots, D_N\}, P, p + D_1\right) \\ \mathcal{B}\left(\{D_2, \ldots, D_N\}, P, p\right) \end{pmatrix}$$

El resultado de $\mathcal{B}(D, P, 0)$ sería el mejor resultado del problema, ya que

- Homero empieza comprando 0 pesos en rosquillas.
- Si no hay rosquillas disponibles, Homero no puede gastar más plata en rosquillas.
- Si hay dos casos donde se gasta menos que P pesos, el mejor caso es el que se gastó el mayor valor.
- El resultado no va a ser menor a 0; lo peor que le puede pasar a Homero es no poder comprar ninguna rosquilla.
- Cada rosquilla se puede comprar solo una vez.
- Cada combinación de $F \subseteq D$ de rosquillas compradas por Homero es contada. En particular, cada $d \in D$ aparece y no aparece en el precio final, alternativamente.

Si se implementa una función B que devuelva el resultado de $\mathcal{B}$ recursivamente, la complejidad en tiempo sería.

$$\begin{aligned} \mathcal{O}(1) &\text{ en el caso base.} \\ \mathcal{O}(1) &\text{ en cada paso de la recursión.} \\ \mathcal{O}(2^N) &\text{ pasos de la recursión diferentes: dos paso por cada elemento de D.} \\ \mathcal{O}(2^N) \cdot \mathcal{O}(1) = \mathcal{O}(2^N) &\text{ complejidad de tiempo total.} \end{aligned}$$

Esto no es lo suficientemente rápido para las restricciones del problema, así que hay que usar otro método.

1.1.2. Meet in the Middle

Dado cierto número $\tau \in [1, N]$, separamos D en dos conjuntos.

$$A = D_1, \dots, D_\tau$$
$$B = D_{\tau+1}, \dots, D_N$$

Luego, definimos la función $\mathcal{V} : \{Tamano\} \times precio \times precio \to \{precio\}$ tal que $V(D, P, p)$ es el conjunto de rosquillas subconjunto de D que se pueden comprar con P pesos si ya se pagaron p pesos de una manera similar a $\mathcal{B}$.

$$\mathcal{V}(\varnothing, P, p) = \begin{cases} \{p\} & \text{si } p \leq P \\ \varnothing & \text{si } p > P \end{cases}$$
$$\mathcal{V}\left(\{D_1, D_2, \dots, D_N\}, P, p\right) = \begin{array}{c} \mathcal{V}\left(\{D_2, \dots, D_N\}, P, p + D_1\right) \\ \cup\, \mathcal{V}\left(\{D_2, \dots, D_N\}, P, p\right) \end{array}$$

Se puede ver que la complejidad en tiempo de calcular $\mathcal{V}(D, P, p)$ es $\mathcal{O}\left(2^{|D|}\right)$ usando el mismo cálculo que se usó para $\mathcal{B}$.

Si calculamos $Q = \mathcal{V}(A, P, 0); W = \mathcal{V}(B, P, 0)$, estos dos valores van a ser subconjuntos de cantidades de donas que se pueden comprar dentro de A y B. Si la suma de dos elementos de cada uno de los conjuntos es menor que P, entonces se pueden comprar cierta cantidad de rosquillas de A y cierta cantidad de B sin gastar más plata que la que se tiene. Calcular estos dos conjuntos tiene complejidad $\mathcal{O}(2^\tau + 2^{N-\tau})$.

Por cada elemento q de entre los $t \leq 2^{|A|}$ elementos de Q, se puede elegir cualquier elemento $w \in W$ de sus $y \leq 2^{|B|}$ elementos siempre y cuando la suma de cada uno sea menor o igual a P. Pero, ¿cuál elegir? Como queremos maximizar la cantida de donas compradas, se debería tomar, por cada $q \in Q$ el mayor elemento $w \in W$ posible tal que $q + w \leq P$, osea, $w \leq P - q$. Como $A \cup B = D$ esto da todas las soluciones posibles, y una de estas tiene que ser la solución óptima.

Usando una estructura de árbol balanceado para representar los conjuntos (como `set` de C++), la operación de buscar el mayor elemento menor o igual a otro elemento en W (similar a `lower_bound`) tiene complejidad $\mathcal{O}\left(\log |W|\right) = \mathcal{O}\left(\log 2^{|B|}\right) = \mathcal{O}\left(|B|\right) = \mathcal{O}\left(N - \tau\right)$. Como esto se tiene que hacer por cada elemento de Q, la complejidad final del "merge" es $\mathcal{O}(2^\tau \cdot (N - \tau)) = \mathcal{O}(N \cdot 2^\tau)$.

La complejidad final en tiempo de hacer el merge de los dos conjuntos es $\mathcal{O}(2^\tau + 2^{N-\tau} + N \cdot 2^\tau) = \mathcal{O}(2^{N-\tau} + N \cdot 2^\tau)$. Como la exponenciación crece mucho más rápido que la multiplicación, la complejidad menor se alcanza poniendo $\tau = {}^{1}\!/_{2}$. Esta complejidad entonces queda como $\mathcal{O}(2^{\frac{1}{2}} + N \cdot 2^{\frac{1}{2}}) = \mathcal{O}(N \cdot \frac{1}{2})$, que esta vez sí cumple con las restricciones del enunciado.

1.2. Código de la solución

```cpp
#include <cassert>
#include <iostream>
#include <set>
#include <vector>

// Backtracker(D).backtrack() construye una lista de costos posibles para
// comprar rosquillas de costo D_0, ..., D_m.
// Llama a backtrackFrom 2 veces por cada llamada desde D_0 hasta D_m, asi
// que la complejidad del algoritmo es O(2^m).
struct Backtracker
{
  const std::vector <int> &D;
  std::set <int> R;

  // Hacer backtracking en D_t, ..., D_m, mientras que se asume que ya se
  // compraron c rosquillas.
  // Cuando se llega al final, agregar el resultado al conjunto R.
  void backtrackFrom(int t, int c)
  {
    if (t == D.size())
      R.insert(c);
    else
    {
      backtrackFrom(t + 1, c);
      backtrackFrom(t + 1, c + D[t]);
    }
  }

  // Hacer backtracking para encontrar
  std::set <int> &backtrack()
  {
    if (R.empty())
      backtrackFrom(0, 0);

    return R;
  }

  Backtracker(const std::vector <int> &D)
  : D(D)
  {
  }
};

// Devuelve la mayor cantidad de rosquillas que se pueden comprar si vienen
// en paquetes de D_0, ..., D_n, y se tienen p pesos.
// La complejidad total es O(2^(n / 2) + 2^(n / 2) * n) = O(2^(n / 2) * n).
int comprarRosquillas(std::vector <int> &D, int p)
{
  // Particionar D en dos partes con igual tamano.
  std::vector <int> A(D.begin(), D.begin() + D.size() / 2);
  std::vector <int> B(D.begin() + D.size() / 2, D.end());

  // Buscar la lista de costos posibles para comprar rosquillas tanto en la
  // mitad izquierda como en la mitad derecha.
  // Cada uno de esos tiene complejidad temporal O(2^(n / 2)).
  std::set <int> valuesLeft = Backtracker(A).backtrack();
  std::set <int> valuesRight = Backtracker(B).backtrack();
```

```
  // Se deberian poder gastar 0 pesos en la mitad izquierda, ya que siempre
  // se puede no comprar ninguna rosquilla. Si esto no fuese cierto,
  // *--valuesLeft.upper_bound(x) podria fallar con x < min(valuesLeft).
  assert(valuesLeft.find(0) != valuesLeft.end());

  // Por cada elemento de la particiin de la derecha, buscar el elemento de
  // la particion de la izquierda tal que comprando esas partes de las dos
  // mitades cueste menos que p, pero que igual se compren la mayor cantidad
  // de donas. Entre todos los elementos, elegir el que maximice la suma.
  // Como std::set::upper_bound es un arbol balanceado, la complejidad
  // de esto es O(log 2^(n / 2)) = O(n) por cada elemento de valuesRight,
  // asi que la complejidad total es O(2^(n / 2) * n).
  int best = 0;
  for (int r : valuesRight)
  {
    if (r <= p)
      best = std::max(best, r + *--valuesLeft.upper_bound(p - r));
  }

  return best;
}

int main()
{
  int p, n;
  while (std::cin >> p >> n)
  {
    std::vector<int> D(n);
    for (int &k : D)
      std::cin >> k;

    std::cout << comprarRosquillas(D, p) << std::endl;
  }

  return 0;
}
```

2. El cumpleaños de Lisa

3. Experimentos nucleares

4. El error de Smithers