Programowanie dynamiczne

18 stycznia 2019

1 FCANDY (Wcale nie gra w minima)

1.1 Wejście

N $(1 \leqslant i \leqslant N \leqslant 100)$ - liczba rodzajów cukierków.

k_i $(1 \leqslant k_i \leqslant 500)$ - liczba cukierków danego rodzaju

c_i $(1 \leqslant c_i \leqslant 200)$ - liczba kalorii danego cukierka

Dodatkowo zdefiniujmy:

$K := max_i k_i$

$C := max_i c_i$

1.2 O co chodzi?

Mamy dany multizbiór S, należy znaleźć podział na dwa rozłączne podzbiory $P, P - S$, taki aby zminimalizować wartość bezwzględną różnic sum elementów w podzbiorach.

$$f(P) := sum(P) - sum(P - S)$$
$$\hat{f}(S) := min_{P \subseteq S} |f(P)|$$

1.3 Bruteforce $O(2^{NK})$

Sprawdzamy wszystkie możliwe kombinacje P i wypisujemy tą z najmniejszą wartością $\hat{f}(P)$. Algorytm będzie działał w czasie $O(2^{NK})$

Obserwacja 1. *Różnica dla zerowej liczby cukierków wynosi 0.*

Obserwacja 2. $jc - (n - j)c = (2j - n)c$

```
vector<int> k;
vector<int> c;
int brute(int wynik, int i) {
        if(i > k.size()) return abs(wynik);
        int res = INT_MAX;
        for(j = 1; j <= k[i], j++) {
                int czesc = (2 * j - k[i]) * c[i];
                res = min(res, brute(wynik + czesc, i+1);
        }
        return res;
}
res = brute(1);
```

1.4 Programowanie dynamiczne $O(CK^2N^2)$

Obserwacja 3. *Liczba możliwych różnic jest ograniczona.*

$$M_1 := CKN \leqslant 10^7$$
$$\forall_{P \subseteq S} - M_1 \leqslant f(P) \leqslant M_1$$

Obserwacja 4. *Nie trzeba rozpatrywać wszystkich kombinacji.*
Dla każdego prefiksu interesuje nas jedynie zbiór uzyskanych różnic częsciowych.

Obserwacja 5. *Mając dany zbiór możliwych różnic częsciowych funkcji f dla pierwszych i rodzajów cukierków potrafimy policzyć zbiór możliwych wartości częsciowych dla $i + 1$ rodzajów cukierków.*

Obserwacja 6. *Wystarczy pamiętać jedynie dwa ostatnie zbiory.*

```
S[0] = {0}, S[1] = {};
for(i = 1,...,n) {
        S[i mod 2] = {};
        for(v : S[(i+1) mod 2]) {
                for(j = 0,...,k[i]) {
                        dc = (2*j - k[i])*c[i];
                        S[i mod 2].insert(v + dc);
                }
        }
}
res = INT_MAX;
for(v : S[n%2]) res = min(res,|v|);
```

Algorytm działa w pamięci $O(M_1)$ oraz czasie

$O(KNM_1 log M_1)$ z std::set

$O(KNM_1) = O(CK^2N^2)$ z std::unordered_set

Dla dużych danych algorytm jest nadal za wolny.

1.5 Optymalizacja $O(CK^2N)$

Lemat 1. *Wynik jest ograniczony z góry przez C.*

Dowód. Posortujmy cukierki pod względem liczby kalorii, a następnie ustawmy je w rządku. Połóżmy je następnie na osi X układu współrzędnych, a następnie nadajmy kolejne indeksy. Przydzielając cukierki parami (2k-1, 2k), otrzymamy pewne odcinki i punkty na osi X. Możemy tak uczynić zawsze, gdy liczba cukierków jest parzysta. W przypadku nieparzystym wystarczy dodać fikcyjny cukierek o kaloryczności 0.
Zauważmy, że suma ich długości jest ograniczona z góry przez C. □

Lemat 2. *Jeśli wartość funkcji na każdym prefiksie jest ograniczona i rośnie, to jeśli wartość na prefiksie przekroczy pewien próg, to wówczas tej kombinacji można nie rozpatrywać.*

Dowód.

$$D := \sum_i k_i c_i$$

$$min_{[a]} |\sum_i (2a_i - k_i) c_i|$$

$$min_{[a]} |2 \sum_i a_i c_i - D|$$

A więc w interesującym nas przypadku:

$$0 \leqslant 2 \sum_i a_i c_i \leqslant D + C \leqslant (K+1)C = M_2$$

$$M_2 = O(KC)$$

□

Wystarczy więc jedynie zmienić sposób wrzucania do zbioru, oraz nie wrzucać do niego elementów większych od M_2.
Otrzymaliśmy algorytm $O(KNM_2) = O(CK^2N)$. Niestety, jest on nadal zbyt wolny.

1.6 Optymalizacja $O(CKN)$

Obserwacja 7. *Dodając kolejne cukierki tak naprawdę dodajemy osiągalne sumy do zbioru. Można to interpretować jako odwiedzanie kolejnych wierzchołków, które te sumy reprezentują.*

Obserwacja 8. *Dwie najbardziej wewnętrzne pętle wykonują zdecydowanie za dużo operacji. Między innymi przepisujemy cały poprzedni zbiór, a następnie odwiedzamy wielokrotnie wcześniej już odwiedzone wierzchołki. Operacja dodania danego typu cukierków kosztowała nas O(MK). Zauważmy, że wystarczy wykonać jedynie O(M) operacji.*

```
// Odwiedzone
bool visited = (true, false, false, ..., false)
// Kolejki wierzcholkow do zaktualizowania
U[0] = {}, U[1] = {}
// Dla kazdego typu cukierkow
for (i = 1, ..., n) {
// Oblicz przyrost wynikajacy z dodania jednego nowego
    cukierka
        dc = 2 * c[i];
        for (v = dc,...,M) {
// Dla kazdego nieodwiedzonego wierzcholka, do ktorego da
     sie dojsc
                if(visited[v] && !visited[v - dc]) {
// Oznacz nowy wierzcholek jako odwiedzony
                        visited[v] = true;
// Dodaj do kolejki do przetworzenia
                        U[1].insert(v);
                }
        }
// Dla kazdego dodanego cukierka
        for(j = 2, ..., k[i]) {
// Stworz nowa pusta kolejke
                U[i%2] = {};
// Dla kazdego wierzcholka z  poprzedniej kolejki
                for(v : U[(i+1)%2]) {
// Znajdz wartosc sumy po dodaniu cukierka
                        nv = v + dc;
// Jesli wczesniej nie uzyskalismy tej sumy.
                        if(!visited[nv]) {
 // Oznacz uzyskanie sumy jako odwiedzony.
                                visited[nv] = true;
// Dodaj do nowej kolejki
                                U[i%2].insert(v);
                        }
                }
        }
}
```

Jako kolejkę wystarczy użyc standardowego std::vector. Powyższy algorytm działa w czasie $O(NM_2) = O(CKN)$.