Własność optymalnej podstruktury

Programowanie dynamiczne ma zastosowanie w problemach wykazujących **własność optymalnej podstruktury** – to znaczy, kiedy optymalne rozwiązanie problemu łatwo jest uzyskać znając optymalne rozwiązania podproblemów. Rozważmy następujący przykład:

Problem wydawania reszty

Dane: Nominały monet $n_1, n_2 \ldots, n_k \in \mathbb{N}$
Kwota $K \in \mathbb{N}$

Szukane: Sposób na wydanie kwoty K używając najmniejszej możliwej liczby monet
Innymi słowy: ciąg liczb naturalnych $a_1, a_2, \ldots, a_k \in \mathbb{N}$ taki, że $\sum_{i=1}^{k} a_i n_i = K$ minimalizujący sumę $\sum_{i=1}^{k} a_i$

Przyjrzyjmy się optymalnemu rozwiązaniu $a_1, a_2, \ldots, a_k$ powyższego problemu. Przypuśćmy, że $a_\ell > 0$ dla pewnego $\ell \in \{1, 2, \ldots, k\}$. Zauważmy, że $a_1, \ldots, a_{\ell-1}, a_\ell - 1, a_{\ell+1}, a_k$ (rozwiązanie ze współczynnikiem a_ℓ zmniejszonym o 1) musi być optymalnym rozwiązaniem podproblemu wydawania reszty dla tych samych nominałów i kwoty $K - n_\ell$ (gdyby tak nie było, umielibyśmy skonstruować lepsze niż optymalne rozwiązanie dla wyjściowego problemu, co jest oczywiście niemożliwe).

Przeformułujmy powyższą obserwację: przy ustalonych nominałach $n_1, n_2 \ldots, n_k$, optymalne rozwiązanie problemu wydawania reszty dla kwoty K zawsze możemy otrzymać z rozwiązania podproblemu dla kwoty $K - n_\ell$ dla pewnego ℓ poprzez zwiększenie współczynnika przy n_ℓ o 1.

Stwierdzenie to możemy przeformułować jeszcze raz: przy ustalonych nominałach $n_1, n_2 \ldots, n_k$ optymalne rozwiązanie problemu wydawania reszty dla kwoty K możemy łatwo znaleźć, jesli znamy optymalne rozwiązania podbroblemów dla kwoty $K - n_\ell$ dla **każdego** $\ell \in \{1, 2, \ldots, k\}$ – wystarczy przejrzeć rozwiązania podbproblemów i wybrać to, które używa najmniejszej liczby monet.

Powyższe rozważania pokazują, że problem wydawania reszty ma własność optymalnej podstruktury.

Algorytm rozwiązujący problem

Powyższe rozumowanie możemy wprost przełożyć na kod rozwiązujący problem. Poniższa rekurencyjna funkcja ChangeMakingRec jako argumenty przyjmuje listę nominałów oraz kwotę K i zwraca minimalną liczbę monet o podanych nominałach potrzebną do uzyskania kwoty K (lub ∞, gdy jest to niemożliwe).

Procedura ChangeMakingRec($n_1, \ldots, n_k \in \mathbb{N}$, $K \in \mathbb{N}$)
  Jeżeli K==0 **wykonaj**
    return 0
  Jeżeli K<0 **wykonaj**
    return ∞
  min = ∞
  Dla $\ell = 1, 2, \ldots, k$ **powtarzaj**
    c = ChangeMakingRec($n_1, \ldots, n_k, K - n_\ell$)
    Jeżeli c < min **wykonaj**
      min = c
  return min + 1

Takie rozwiązanie jest poprawne, ale bardzo wolne. Zasadniczym problemem jest tutaj to, że dla niektórych problemów wyznaczamy to rozwiązanie wielokrotnie – przykładowo, dla $k = 2$ oraz $n_1 = n_2 = 1$ „najgłębsze" rekurencyjne wywołanie ChangeMakingRec$(1, 1, 0)$ zostanie wykonane 2^K razy.

Sposobem na poradzenie sobie z tym problemem jest spamiętywanie rozwiązań napotkanych podproblemów. W tym przypadku podproblemy odpowiadają liczbom naturalnym z zakresu od 0 do K, zatem ich rozwiązania możemy trzymać w jednowymiarowej tablicy. Ściślej: przez $T[kk]$ rozumiemy minimalną liczbę monet o nominałach ze zbioru $n_1, n_2 \ldots, n_k$ potrzebną do uzyskania kwoty kk.

```
Procedura ChangeMaking(n_1, ..., n_k ∈ ℕ, K ∈ ℕ)
    T − tablica rozmiaru K + 1 inicjowana wartością ∞
    T[0] = 0
    Dla kk = 1, 2, ..., K powtarzaj
        T[kk] = ∞
        Dla ℓ = 1, 2, ..., k powtarzaj
            c = 1 + T[kk − n_ℓ]
            Jeżeli c < T[kk] wykonaj
                T[kk] = c
    return T[K]
```

Powyższa procedura wyznacza tylko liczbę monet używanych przez optymalne rozwiązanie, łatwo ją jednak poprawić tak, aby zwracała optymalne rozwiązanie.

```
Procedura ChangeMaking2(n_1, ..., n_k ∈ ℕ, K ∈ ℕ)
    T − tablica rozmiaru K + 1 inicjowana wartością ∞
    P − tablica rozmiaru K + 1
    T[0] = 0
    Dla kk = 1, 2, ..., K powtarzaj
        T[kk] = ∞
        Dla ℓ = 1, 2, ..., k powtarzaj
            c = 1 + T[kk − n_ℓ]
            Jeżeli c < T[kk] wykonaj
                T[kk] = c
                P[kk] = ℓ
    Jeżeli T[kk] == ∞ wykonaj
        return null
    A − tablica rozmiaru k inicjowana wartością 0
    kk=K
    Dopóki kk > 0 powtarzaj
        A[P[kk]] += 1
        kk -= n_{P[kk]}
    return A
```

Zadanie: wydawanie reszty

Uzupełnić metody
static int? NoLimitsDynamic(int amount, int[] coins, out int[] change)
static int? Dynamic(int amount, int[] coins, int[] limits, out int[] change)
rozwiazujące problem wydawania reszty minimalną liczbą monet odpowiednio bez ograniczeń na liczbę monet danego nominału i z takimi ograniczeniami.

Opis parametrów i wyniku znajduje się w pliku ChangeMaking.cs

Wskazówki (do wersji z limitami):

- Użyć tablicy prostokątnej do pamiętania optymalnej liczby monet dla danego podzadania
- W komórce [i,j] pamiętać rozwiązanie dla i pierwszych nominałów oraz kwoty j
- Tablicę wypełniać wierszami
  - Zwiększenie numeru wiersza to uwzględnienie kolejnego rodzaju monety w rozwiązaniu
  - Zwiększenie numeru kolumny to zwiększenie kwoty reszty
- Może być potrzebna jeszcze trzecia pętla
- Może też przydać się druga tablica o analogicznej strukturze

Punktacja:
Część 1 - bez limitów

- 0.5 – tylko liczba monet (wartość zwracana funkcji)
- 1.0 – pełne rozwiązanie (wartość zwracana i parametr change)

Część 2 - z limitami

- 1.0 – tylko liczba monet (wartość zwracana funkcji)
- 1.5 – pełne rozwiązanie (wartość zwracana i parametr change)

Kara za złamanie ograniczenia na złożoność pamięciową (podanego w pliku ChangeMaking.cs): po -0.5 pkt za każdą metodę.