

## 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  $n_1, n_2, \dots, n_k \in \mathbb{N}$

Kwota  $K \in \mathbb{N}$

**Szukane:** Sposób na wydanie kwoty  $K$  używając najmniejszej możliwej liczby monet

Innymi słowy: ciąg liczb naturalnych  $a_1, a_2, \dots, a_k \in \mathbb{N}$  taki, że  $\sum_{i=1}^k a_i n_i = K$  minimalizujący sumę  $\sum_{i=1}^k a_i$

Przyjrzyjmy się optymalnemu rozwiązaniu  $a_1, a_2, \dots, a_k$  powyższego problemu. Przypuśćmy, że  $a_\ell > 0$  dla pewnego  $\ell \in \{1, 2, \dots, k\}$ . Zauważmy, że  $a_1, \dots, a_{\ell-1}, a_\ell - 1, a_{\ell+1}, a_k$  (rozwiązanie ze współczynnikiem  $a_\ell$  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  $n_1, n_2, \dots, n_k$ , optymalne rozwiązanie problemu wydawania reszty dla kwoty  $K$  zawsze możemy otrzymać z rozwiązania podproblemu dla kwoty  $K - n_\ell$  dla pewnego  $\ell$  poprzez zwiększenie współczynnika przy  $n_\ell$  o 1.

Stwierdzenie to możemy przeformułować jeszcze raz: przy ustalonych nominałach  $n_1, n_2, \dots, n_k$  optymalne rozwiązanie problemu wydawania reszty dla kwoty  $K$  możemy łatwo znaleźć, jeśli znamy optymalne rozwiązania podproblemów dla kwoty  $K - n_\ell$  dla **każdego**  $\ell \in \{1, 2, \dots, k\}$  – wystarczy przejrzeć rozwiązania podproblemów 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  $\infty$ , gdy jest to niemożliwe).

```
1: Procedura CHANGEMAKINGREC( $n_1, \dots, n_k \in \mathbb{N}, K \in \mathbb{N}$ )
2:   Jeżeli  $K=0$  wykonaj
3:     return 0
4:   Jeżeli  $K<0$  wykonaj
5:     return  $\infty$ 
6:    $\text{min} = \infty$ 
7:   Dla  $\ell = 1, 2, \dots, k$  powtarzaj
8:      $c = \text{CHANGEMAKINGREC}(n_1, \dots, n_k, K - n_\ell)$ 
9:     Jeżeli  $c < \text{min}$  wykonaj
10:       $\text{min} = c$ 
11:   return  $\text{min} + 1$ 
```

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. Ścisłej: przez  $T[kk]$  rozumiemy minimalną liczbę monet o nominałach ze zbioru  $n_1, n_2, \dots, n_k$  potrzebną do uzyskania kwoty  $kk$ .

```
1: Procedura CHANGEMAKING( $n_1, \dots, n_k \in \mathbb{N}, K \in \mathbb{N}$ )
2:    $T$  – tablica rozmiaru  $K + 1$  inicjowana wartością  $\infty$ 
3:    $T[0] = 0$ 
4:   Dla  $kk = 1, 2, \dots, K$  powtarzaj
5:      $T[kk] = \infty$ 
6:     Dla  $\ell = 1, 2, \dots, k$  powtarzaj
7:        $c = 1 + T[kk - n_\ell]$ 
8:       Jeżeli  $c < T[kk]$  wykonaj
9:          $T[kk] = c$ 
10:  return  $T[K]$ 
```

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

```
1: Procedura CHANGEMAKING2( $n_1, \dots, n_k \in \mathbb{N}, K \in \mathbb{N}$ )
2:    $T$  – tablica rozmiaru  $K + 1$  inicjowana wartością  $\infty$ 
3:    $P$  – tablica rozmiaru  $K + 1$ 
4:    $T[0] = 0$ 
5:   Dla  $kk = 1, 2, \dots, K$  powtarzaj
6:      $T[kk] = \infty$ 
7:     Dla  $\ell = 1, 2, \dots, k$  powtarzaj
8:        $c = 1 + T[kk - n_\ell]$ 
9:       Jeżeli  $c < T[kk]$  wykonaj
10:         $T[kk] = c$ 
11:         $P[kk] = \ell$ 
12:   Jeżeli  $T[kk] == \infty$  wykonaj
13:     return null
14:    $A$  – tablica rozmiaru  $k$  inicjowana wartością 0
15:    $kk = K$ 
16:   Dopóki  $kk > 0$  powtarzaj
17:      $A[P[kk]] += 1$ 
18:      $kk -= n_{P[kk]}$ 
19:  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)
```

rozwiązujące 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.

## Zadanie: skreślanie ciągów

Przypuśćmy, że mamy dany ciąg znaków, który chcemy wytrzeć. Nie jest to jednak tak proste, jak mogłoby się wydawać, bo (z powodu klątwy faraona) jesteśmy ograniczeni – możemy wycierać tylko niektóre znaki oraz niektóre wzorce składające się z dwóch znaków.

Przykłady: Dla ciągu  $xAAxB$  oraz wzorców  $x$ ,  $xA$ ,  $xB$ ,  $AB$ :

wycieramy  $xA$ , zostaje nam  $AxB$ ,

wycieramy  $x$ , zostaje  $AB$ ,

wycieramy  $AB$  i osiągamy sukces.

(Zauważmy, że nieprzemyślane wytarcie  $xB$  na początku spowoduje, że sukces nie będzie możliwy).

Dla ciągu  $()))()$  oraz wzorca  $()$  nie da się osiągnąć sukcesu.

Po trzykrotnym wytarciu wzorca zostaje nam ciąg  $))()$  i jest to najkrótszy ciąg, jaki możemy uzyskać.

Zadanie składa się z trzech części:

1. Dla zadanego ciągu znaków oraz listy wzorców określ, czy możliwe jest wytarcie wszystkich znaków przez wycieranie podanych wzorców (wartość zwracana przez metodę `Erasable`).
2. Wyznacz najmniejszą możliwą liczbę wytarć potrzebnych do wytarcia wszystkich znaków. Jeżeli nie jest to możliwe, zwróć `int.MaxValue` (parametr wyjściowy `crossoutsNumber` metody `Erasable`).
3. Wyznacz długość najkrótszego ciągu, jaki można uzyskać przez wycieranie podanych wzorców (wartość zwracana przez metodę `MinimumRemainder`).

### Punktacja:

Etap 1. 1 punkt,

Etap 2. 0.5 punktu,

Etap 3. 1 punkt.

### Wskazówki:

Zastosuj programowanie dynamiczne.

Dla każdego spójnego fragmentu podanego ciągu oblicz, czy fragment ten może być wytarty (skreślony).

Fragmenty identyfikuj poprzez indeksy początku i końca.

Fragment tekstu jest wycieralny, jeżeli:

- jest pusty
- jest jednym ze wzorców
- jest konkatenacją dwóch krótszych fragmentów wycieralnych
- zawiera (spójny) fragment wycieralny, a to co zostaje po jego usunięciu jest jednym ze wzorców

### Uwagi:

- Zastanowić się jak inaczej można opisać ostatnią wymienioną powyżej cechę tekstów wycieralnych – to znaczy, jakie powinno być wzajemne położenie wycieralnego fragmentu i pozostałego wzorca (lub jego części). Niektóre potencjalne położenia zawierają się już we wcześniejszych własnościach (i nie trzeba ich ponownie sprawdzać). Jeśli do implementacji sprawdzenia ostatniej cechy potrzebna Ci jest pętla, to znaczy że nie wybrałeś najprostszego sposobu.
- Wygodnie jest skorzystać z podanych funkcji pomocniczych `comparePattern`.