9 Inne problemy

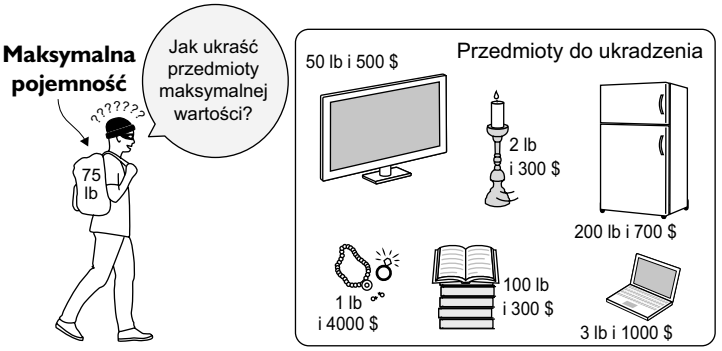
W tej książce omawiam wiele technik rozwiązywania problemów stosowanych w nowoczesnym oprogramowaniu. Analizując te techniki, posłużyłem się klasycznymi problemami informatycznymi. Jednak nie wszystkie klasyczne problemy pasowały do poprzednich rozdziałów. W tym rozdziale zebrałem te klasyczne problemy, który trudno było dołączyć do któregokolwiek z wcześniejszych rozdziałów. Są to problemy bonusowe, czyli ciekawsze problemy wymagające mniej pomocniczego rusztowania.

9.1. Problem plecakowy

Problem plecakowy (ang. *knapsack problem*) jest problemem optymalizacji, który zaczyna się od zwykłego zadania obliczeniowego (znalezienia najlepszego sposobu wykorzystania ograniczonych zasobów przy skończonym zbiorze możliwości) i przekształca go w ciekawą historię. Złodziej wchodzi do domu z zamiarem dokonania kradzieży. Ma ze sobą plecak i może ukraść tylko tyle, ile może w nim unieść. Jak ma zadecydować, co włożyć do plecaka? Problem został zilustrowany na rysunku 9.1.

Jeśli złodziej mógłby wziąć dowolną ilość dowolnego przedmiotu, wystarczyłoby podzielić wartość każdego elementu przez jego wagę, aby wybrać najwartościowsze przedmioty dla określonej ładowności plecaka. Ale aby scenariusz był bardziej realistyczny, załóżmy, że złodziej nie może wziąć połowy przedmiotu (np. 2,5 telewizora). Zamiast tego wymyślmy sposób rozwiązania wariantu „0/1” problemu, którego nazwa wynika z dodatkowej reguły: złodziej może wziąć tylko jedną sztukę przedmiotu albo nie brać go wcale.

Na początku zdefiniujmy klasę `NamedTuple` do przechowywania naszych przedmiotów.



Rysunek 9.1. Złodziej musi zadecydować, jakie przedmioty ukraść, ponieważ plecak ma ograniczoną ładowność

Listing 9.1. knapsack.py

```
from typing import NamedTuple, List

class Item(NamedTuple):
    name: str
    weight: int
    value: float
```

Gdybyśmy próbowali rozwiązać ten problem metodą siłową, sprawdzilibyśmy wszystkie dostępne kombinacje do umieszczenia w plecaku. Matematycy nazywają taki zbiór *zbiorem potęgowym*. Zbiór potęgowy zbioru (w naszym przykładzie zbioru przedmiotów) ma 2^*N* różnych potencjalnych podzbiorów, gdzie *N* to liczba przedmiotów. W związku z tym musielibyśmy przeanalizować 2^*N* kombinacji (O(2^*N*)). Jest to wykonalne tylko w przypadku niewielkiej liczby przedmiotów. Lepiej jest unikać metod, które rozwiązują problemy w wykładniczo rosnącej liczbie kroków.

Zamiast tego użyjemy techniki nazywanej *programowaniem dynamicznym*, które jest koncepcją podobną do memoizacji (rozdz. 1). Zamiast bezpośrednio rozwiązywać problem metodami siłowymi, w programowaniu dynamicznym rozwiązujemy problemy podrzędne będące składowymi większego problemu, przechowujemy wyniki, a następnie wykorzystujemy je do rozwiązywania większego problemu. Jeśli ładowność plecaka potraktujemy jak wartość dyskretną, możemy rozwiązać problem przy użyciu programowania dynamicznego.

Do rozwiązania na przykład problemu dla plecaka o ładowności 3 lb i 3 przedmiotów do ukradzenia zaczynamy od rozwiązania problemu dla ładowności 1 lb i 1 przedmiotu, ładowności 2 lb i 1 przedmiotu oraz ładowności 3 lb i 1 przedmiotu. Następnie możemy użyć tych wyników do rozwiązania problemu dla ładowności 1 lb i 2 przedmiotów, ładowności 2 lb i 2 przedmiotów, ładowności 3 lb i 2 przedmiotów. Na zakończenie szukamy rozwiązania dla wszystkich 3 przedmiotów.

Po drodze będziemy wypełniać tabelę, która wskazuje najlepsze rozwiązanie dla każdej kombinacji przedmiotów i ładowności. Nasza funkcja na początku wypełni tabelę, a następnie będzie znajdować rozwiązanie na podstawie tej tabeli[1].

Listing 9.2. knapsack.py kontynuacja

```
def knapsack(items: List[Item], max_capacity: int) -> List[Item]:
    # budowanie tabeli programowania dynamicznego
    table: List[List[float]] = [[0.0 for _ in range(max_capacity + 1)] for _
     in range(len(items) + 1)]
    for i, item in enumerate(items):
        for capacity in range(1, max_capacity + 1):
            previous_items_value: float = table[i][capacity]
            if capacity >= item.weight: # przedmiot mieści się w plecaku
                value_freeing_weight_for_item: float = table[i][capacity -
     item.weight]
                # weź tylko, jeśli wartościowszy niż poprzedni przedmiot
                table[i + 1][capacity] = max(value_freeing_weight_for_item +
     item.value, previous_items_value)
            else: # nie ma miejsca na ten przedmiot
                table[i + 1][capacity] = previous_items_value
    # opracowanie rozwiązania na podstawie tabeli
    solution: List[Item] = []
    capacity = max_capacity
    for i in range(len(items), 0, -1): # sprawdzanie od końca
        # czy ten przedmiot został użyty?
        if table[i - 1][capacity] != table[i][capacity]:
            solution.append(items[i - 1])
            # jeśli przedmiot został użyty, odejmij jego wagę
            capacity -= items[i - 1].weight
    return solution
```

Wewnętrzna pętla pierwszej części funkcji będzie wykonywana $N * C$ razy, gdzie N to liczba przedmiotów, a C to maksymalna ładowność plecaka. W związku z tym czas wykonywania algorytmu wynosi $O(N * C)$, co dla dużej liczby przedmiotów (na liście `items`) reprezentuje znaczną poprawę w porównaniu z metodami siłowymi. Na przykład dla 11 przedmiotów, które za chwilę dodamy, algorytm bazujący na metodach siłowych musiałby sprawdzić 2^11, czyli 2048 kombinacji. Przedstawiona funkcja programowania dynamicznego zostanie wykonania 825 razy, ponieważ maksymalna ładowność tego plecaka wynosi 75 niesprecyzowanych jednostek (11 * 75). Ta różnica rosłaby wykładniczo wraz z liczbą przedmiotów.

Przeanalizujmy działanie tego rozwiązania.

[1] Przed napisaniem tego rozwiązania przestudiowałem wiele dokumentów. Najważniejszym z nich była książka Roberta Sedgewicka *Algorithms*. Wyd. 2. Addison-Wesley, 1988, s. 596. Przejrzałem wiele przykładowych rozwiązań problemu 0/1 knapsack na stronie Rosetta Code. Najważniejszym z nich był program Pythona bazujący na programowaniu dynamicznym (http://mng.bz/kx8C), na którym oparłem tę funkcję, przenosząc ją z wersji Swift książki (kod został najpierw przeniesiony z Pythona na Swifta, a później z powrotem na Pythona).