Problem plecakowy

Dane wejściowe

Pojemność plecaka : 7

waga: 1
wartość: 2

waga: 1
wartość: 1

waga: 3
wartość: 7

waga: 3
wartość: 6

waga: 2
wartość: 3

Problem

Które przedmioty należy włożyć do plecaka, aby ich **wartość była jak największa**, i aby zmieściły się do plecaka?

Dane wejściowe

Pojemność plecaka : 7

waga: 1
wartość: 2

waga: 1
wartość: 1

waga: 2
wartość: 3

waga: 3
wartość: 7

waga: 3
wartość: 6

n – liczba przedmiotów

p_i – wartość i-tego przedmiotu

w_i – waga i-tego przedmiotu

c – pojemność plecaka

x_i – zmienna decyzyjna

Cel: $\max \sum_i p_i x_i$

Ograniczenia: $\sum_i w_i x_i \leq c$

Dane wejściowe w postaci tabelarycznej

	1	**2**	**3**	**4**	**5**
p_i	2	1	3	7	6
w_i	1	1	2	3	3

Rozwiązanie optymalne:

$x^* = \{x_1, x_4, x_5\}$

Sumaryczna waga: 7

Sumaryczna wartość: 15

Problem plecakowy

Problem optymalizacyjny

– Znajdź podzbiór elementów, które zmieszczą się do plecaka a suma ich wartości będzie maksymalna –

Definicja problemu

Dany jest zbiór *n* elementów (każdy ma wartość p_i oraz wagę w_i) i plecak o pojemności *c*; x_i – zmienna decyzyjna. Znajdź podzbiór elementów, dla którego:

$$\max \sum_i p_i x_i$$

$$\sum_i w_i x_i \leq c$$

Rozwiązanie problemu

Zbiór elementów, których wartość jest maksymalna, a ich waga nie przekracza pojemności plecaka.

Klasa złożoności problemu

Jest to problem trudny obliczeniowo. Problem optymalizacyjny należy do klasy problemów NP-trudnych (w zwykłym sensie).

Problem decyzyjny

– Czy istnieje taki podzbiór elementów w plecaku, aby suma ich wartości była równa co najmniej *b*? –

Definicja problemu

Dany jest zbiór *n* elementów (każdy ma wartość p_i oraz wagę w_i), plecak o pojemności *c*, wartość progowa *b*; x_i – zmienna decyzyjna. Znajdź podzbiór elementów, dla którego:

$$\sum_i p_i x_i \geq b$$

$$\sum_i w_i x_i \leq c$$

Rozwiązanie problemu

Odpowiedź: TAK (istnieje taki podzbiór elementów) lub NIE (nie istnieje taki podzbiór elementów).

Klasa złożoności problemu

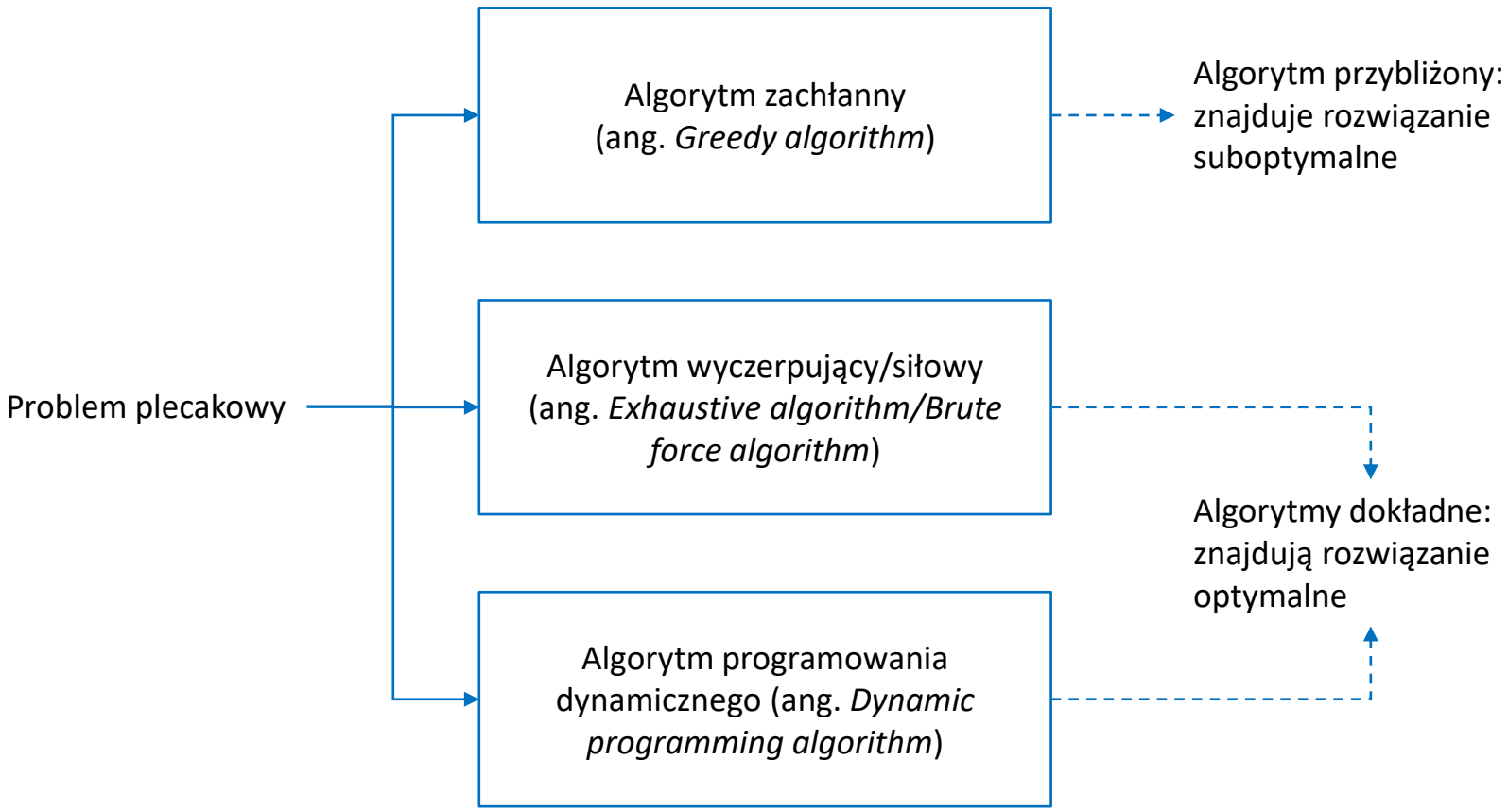
Jest to problem trudny obliczeniowo. Wersja decyzyjna jest w klasie problemów NP-zupełnych.

Problem plecakowy – różne warianty

Rodzaj problemu	**Dodatkowe ograniczenia**
Dyskretny problem plecakowy	
Binarny (0-1) ➢ każdy element występuje tylko raz	$x_i \in \{0,1\}$
Ograniczony ➢ zdefiniowana jest liczba elementów każdego typu - b_i	$x_i \in [0, b_i]$ $x_i \in \mathrm{C}$
Nieograniczony ➢ brak ograniczenia na liczbę elementów każdego typu	$x_i \geq 0$ $x_i \in \mathrm{C}$
Wielowymiarowy ➢ Plecak oraz elementy mają co najmniej 2 wymiary/parametry	$x_i \in \{0,1\}$ $c_i \in \mathrm{N}$ (c_i = *i*-ty wymiar)
Ciągły problem plecakowy	
➢ nie ma wymogu aby x_i było liczbą całkowitą	$x_i \geq 0$
Ponadto wartości parametrów p_i, w_i, n, c są liczbami naturalnymi.	$n, c \in \mathrm{N}, \forall_{i=1..n} p_i, w_i \in N$

W dalszej części będziemy rozważać tylko binarny problem plecakowy.

Jak można rozwiązać problem plecakowy?



Algorytm wyczerpujący (siłowy)

Binarny problem plecakowy

Dane wejściowe

- c – pojemność plecaka
- n – liczba przedmiotów w zbiorze
- x_i – zmienna decyzyjna (czy i-ty przedmiot jest w zbiorze)
- p_i – wartość i-tego przedmiotu
- w_i – waga i-tego przedmiotu

Cel

znajdź optymalne upakowanie plecaka (podzbiór przedmiotów, które mieszczą się w plecaku i mają maksymalną sumaryczną wartość)

Algorytm siłowy wykonuje pełen przegląd przestrzeni rozwiązań.

Algorytm wyczerpujący

```
Knapsack-brute-force()
  fmax = 0
  Dla każdej liczby X od 1 do 2^n-1
      zakoduj X w systemie binarnym na n bitach
      W(X) = suma wag przedmiotów w plecaku
      jeśli W(X) ≤ c     //rozwiązanie dopuszczalne
          f(X) = suma wartości przedmiotów w plecaku
          jeśli f(X) > fmax
              fmax = f(X)
              rozwiazanie = X
```

$$W(X) = \sum_{i=1}^{n} x_i \cdot w_i$$

$$f(X) = \sum_{i=1}^{n} x_i \cdot p_i$$

Algorytm wyczerpujący (siłowy): przykład

Dane wejściowe

c = 8
n = 4

i	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Przebieg działania algorytmu siłowego

	X binarnie				Rozwiązanie bieżące	Suma wag W(X)	Czy dopuszczalne?	Suma wartości f(X)
X	4	3	2	1				
1	0	0	0	1	{1}	2	+	4
2	0	0	1	0	{2}	1	+	2
3	0	0	1	1	{1,2}	3	+	7
4	0	1	0	0	{3}	4	+	6
5	0	1	0	1	{1,3}	6	+	10
6	0	1	1	0	{2,3}	5	+	9
7	0	1	1	1	{1,2,3}	7	+	13
8	1	0	0	0	{4}	4	+	8
9	1	0	0	1	{1,4}	6	+	12
10	1	0	1	0	{2,4}	5	+	11
11	1	0	1	1	{1,2,4}	7	+	15
12	1	1	0	0	{3,4}	8	+	14
13	1	1	0	1	{1,3,4}	10	-	18
14	1	1	1	0	{2,3,4}	9	-	17
15	1	1	1	1	{1,2,3,4}	11	-	21

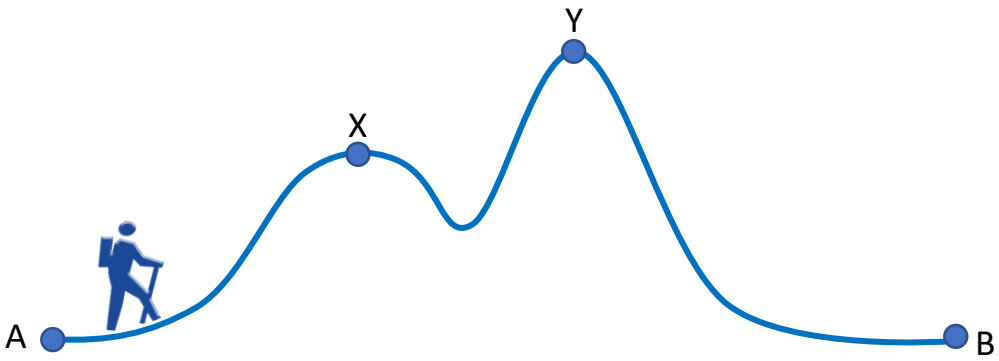
Rozwiązanie **optymalne** →

Zauważmy, że w rozwiązaniu optymalnym przedmioty nie wypełniają całego plecaka.

Algorytm zachłanny (ang. *greedy algorithm*) to ogólna technika algorytmiczna, która polega na rozwiązywaniu problemu optymalizacyjnego w następujący sposób:

1. Algorytm w każdym kroku dokonuje wyboru na podstawie oceny sąsiedztwa.
2. Wybiera przejście do lokalnego optimum (zawsze przechodzi do najlepszego rozwiązania sąsiedniego = decyzja zachłanna).
3. Jeśli w danym kroku nie można polepszyć rozwiązania, algorytm zatrzymuje się i zwraca rozwiązanie bieżące jako rozwiązanie problemu.

Przykład
Cel: wejdź na najwyższy punkt masywu.



Rozwiązanie:

a) Zachłanny turysta, który startuje z pkt. A dotrze to pkt. X (rozwiązanie suboptymalne).
b) Zachłanny turysta, który startuje z pkt. B dotrze do pkt. Y (rozwiązanie optymalne).

Projektując algorytm zachłanny (dla dowolnego problemu optymalizacyjnego)

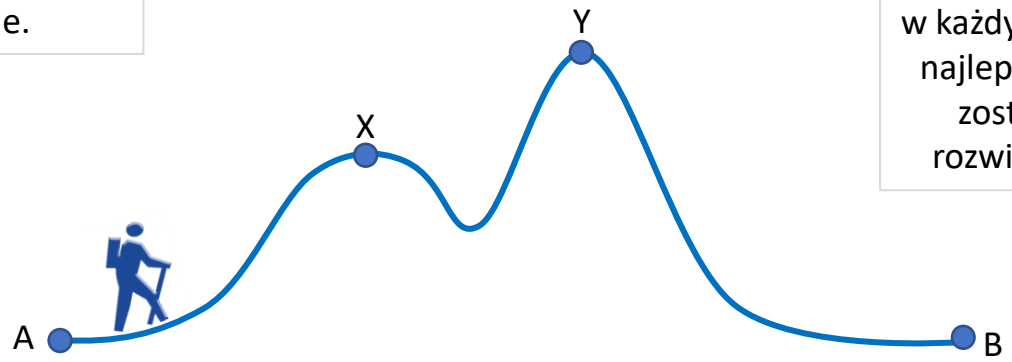
Określ stan początkowy (pierwsze rozwiązanie cząstkowe).

Zdefiniuj funkcję celu: co optymalizujesz i jak to obliczyć dla rozwiązania bieżącego.

Określ jakie warunki musi spełniać rozwiązanie dopuszczalne.

Ustal funkcję wyboru: w jaki sposób będzie w każdym kroku wybierany najlepszy element, który zostanie dodany do rozwiązania bieżącego.

Ustal warunki stopu.



Binarny problem plecakowy

Dane wejściowe

c – pojemność plecaka
n – liczba elementów w zbiorze
x_i – zmienna decyzyjna (czy i-ty element jest w zbiorze)
p_i – wartość i-tego elementu
w_i – waga i-tego elementu

Cel

znajdź optymalne upakowanie plecaka (podzbiór przedmiotów, które mieszczą się w plecaku i mają maksymalną sumaryczną wartość)

Algorytm zachłanny wybiera najbardziej opłacalne elementy i dodaje je do rozwiązania cząstkowego.

Algorytm zachłanny I

```
Knapsack-greedy-I()
  Posortuj przedmioty nierosnąco względem ich wartości (p_i)
  i = 1
     Wykonuj
        jeśli przedmiot i zmieści się w plecaku //rozwiązanie dopuszczalne
           dodaj przedmiot i do plecaka
           i = i+1
     dopóki jest miejsce w plecaku oraz i≤n
```

Algorytm zachłanny I: przykład

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Przebieg działania algorytmu zachłannego I

1. Posortowanie przedmiotów nierosnąco względem wartości (p_i)

i	*id*	w_i	p_i
1	4	4	**8**
2	3	4	**6**
3	1	2	**4**
4	2	1	**3**

$p_1 \geq p_2 \geq \ldots \geq p_n$

id to oryginalny identyfikator przedmiotu,
(taki jaki przedmiot miał na wejściu);
tutaj wpisane zostały właśnie te identyfikatory

2. Budowanie rozwiązania

Rozwiązanie bieżące	Suma wag W(X)	Suma wartości f(X)	Miejsce w plecaku	Elementy poza plecakiem
{}	0	0	8	4, 3, 1, 2
{4}	4	8	4	3, 1, 2
{4, 3}	8	14	0	1, 2

Rozwiązanie znalezione przez algorytm zachłanny I: {3,4} jest **suboptymalne**

Rozwiązaniem optymalnym jest zbiór przedmiotów: {1,2,4}.
Algorytm zachłanny I nie znajdzie tego rozwiązania.

Binarny problem plecakowy

Dane wejściowe

c – pojemność plecaka
n – liczba elementów w zbiorze
x_i – zmienna decyzyjna (czy i-ty element jest w zbiorze)
p_i – wartość i-tego elementu
w_i – waga i-tego elementu

Cel

znajdź optymalne upakowanie plecaka (podzbiór przedmiotów, które mieszczą się w plecaku i mają maksymalną sumaryczną wartość)

Algorytm zachłanny wybiera najbardziej opłacalne elementy i dodaje je do rozwiązania cząstkowego.

Algorytm zachłanny II

```
Knapsack-greedy-II()
  Posortuj przedmioty niemalejąco względem ich wag (w_i)
  i = 1
     Wykonuj
        jeśli przedmiot i zmieści się w plecaku //rozwiązanie dopuszczalne
           dodaj przedmiot i do plecaka
           i = i+1
     dopóki jest miejsce w plecaku oraz i≤n
```

Algorytm zachłanny II: przykład

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Przebieg działania algorytmu zachłannego II

1. Posortowanie przedmiotów niemalejąco względem wag (w_i)

i	*id*	w_i	p_i
1	2	**1**	3
2	1	**2**	4
3	3	**4**	3
4	4	**4**	4

$w_1 \leq w_2 \leq \ldots \leq w_n$

id to oryginalny identyfikator przedmiotu,
(taki jaki przedmiot miał na wejściu);
tutaj wpisane zostały właśnie te identyfikatory

2. Budowanie rozwiązania

Rozwiązanie bieżące	Suma wag W(X)	Suma wartości f(X)	Miejsce w plecaku	Elementy poza plecakiem
{}	0	0	8	2, 1, 3, 4
{2}	1	3	7	1, 3, 4
{2, 1}	3	7	5	3, 4
{2, 1, 3}	7	10	1	4

Rozwiązanie znalezione przez algorytm zachłanny II: {1,2,3} jest **suboptymalne**

Rozwiązaniem optymalnym jest zbiór przedmiotów: {1,2,4}. Algorytm zachłanny II znalazłby je gdyby wśród przedmiotów z tą samą wagą wybierał najpierw przedmiot o większej wartości.

Binarny problem plecakowy

Dane wejściowe

c – pojemność plecaka
n – liczba elementów w zbiorze
x_i – zmienna decyzyjna (czy i-ty element jest w zbiorze)
p_i – wartość i-tego elementu
w_i – waga i-tego elementu

Cel

znajdź optymalne upakowanie plecaka (podzbiór przedmiotów, które mieszczą się w plecaku i mają maksymalną sumaryczną wartość)

Algorytm zachłanny wybiera najbardziej opłacalne elementy i dodaje je do rozwiązania cząstkowego.

Algorytm zachłanny III

```
Knapsack-greedy-III()
  Posortuj przedmioty nierosnąco względem wartości na jednostkę masy (p_i/w_i)
  i = 1
     Wykonuj
        jeśli przedmiot i zmieści się w plecaku //rozwiązanie dopuszczalne
           dodaj przedmiot i do plecaka
           i = i+1
     dopóki jest miejsce w plecaku oraz i≤n
```

Algorytm zachłanny III: przykład

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Przebieg działania algorytmu zachłannego III

1. Posortowanie przedmiotów nierosnąco względem wartości na jednostkę masy (p_i/w_i)

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$$

i	*id*	w_i	p_i	P_i/w_i
1	2	1	3	**3**
2	1	2	4	**2**
3	4	4	8	**2**
4	3	4	6	**1,5**

id to oryginalny identyfikator przedmiotu, (taki jaki przedmiot miał na wejściu); tutaj wpisane zostały właśnie te identyfikatory

2. Budowanie rozwiązania

Rozwiązanie bieżące	Suma wag W(X)	Suma wartości f(X)	Miejsce w plecaku	Elementy poza plecakiem
{}	0	0	8	2, 1, 4, 3
{2}	1	3	7	1, 4, 3
{2, 1}	3	7	5	4, 3
{2, 1, 4}	7	15	1	3

Rozwiązanie znalezione przez algorytm zachłanny III: {1,2,4} jest **optymalne**

Dane wejściowe

c = 7
n = 5

id	w_i	p_i
1	2	5
2	1	4
3	4	12
4	1	2
5	3	10

Przebieg działania algorytmu zachłannego III

1. Posortowanie przedmiotów nierosnąco względem wartości na jednostkę masy (p_i/w_i)

i	*id*	w_i	p_i	P_i/w_i
1	2	1	4	**4**
2	5	3	10	**3,3**
3	3	4	12	**3**
4	1	2	5	**2,5**
5	4	1	2	**2**

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$$

id to oryginalny identyfikator przedmiotu, (taki jaki przedmiot miał na wejściu); tutaj wpisane zostały właśnie te identyfikatory

2. Budowanie rozwiązania

Rozwiązanie bieżące	Suma wag W(X)	Suma wartości f(X)	Miejsce w plecaku	Elementy poza plecakiem
{}	0	0	7	2, 5, 3, 1, 4
{2}	1	4	6	5, 3, 1, 4
{2, 5}	4	14	3	3, 1, 4
{2, 5, 1}	6	19	1	**3**, 4
{2, 5, 1, 4}	7	21	0	3

Przedmiot **3** jest pomijany na tym etapie, gdyż nie zmieści się do plecaka

Rozwiązanie znalezione przez algorytm zachłanny III: {2,5,1,4} jest **suboptymalne**

Rozwiązaniem optymalnym jest zbiór przedmiotów: {5,3}, fmax=22. Dla tego problemu algorytm zachłanny III nie znajdzie rozwiązania optymalnego

Kiedy powstała metoda Programowania Dynamicznego?

Twórcą podejścia jest Richard Bellman - 1952r. PD jest to strategia projektowania algorytmów stosowana przeważnie do rozwiązywania zagadnień optymalizacyjnych.

Na czym polega metoda PD?

1) rozwiąż problem dla jednego elementu przy różnych wartościach parametru sterującego i zapamiętaj wyniki
2) dodaj następny element do problemu
3) zbuduj rozwiązanie problemu powiększonego o nowy składnik dokonując optymalnego wyboru w oparciu o poprzednio zapisane rozwiązania

Kiedy możemy zastosować PD?

PD opiera się na podziale rozwiązywanego problemu na podproblemy względem kilku parametrów (np. liczba elementów w plecaku lub pojemność plecaka). W odróżnieniu od techniki ‚dziel i zwyciężaj', w PD ***podproblemy nie są rozłączne*** i cechuje je ***własność optymalnej struktury***. Zastosowanie dla tego typu problemów metody siłowej (*brute force*) prowadzi do ponad-wielomianowej liczby rozwiązań podproblemów, podczas gdy liczba różnych podproblemów jest wielomianowa.

Własność optymalnej struktury: dany problem ma własność optymalnej struktury jeśli jego optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów. Optymalne rozwiązanie problemu dla *k*-tego etapu jest jednocześnie rozwiązaniem optymalnych dla wszystkich etapów po nim następujących: *k* + 1, *k* + 2 ,…, *n*. Zatem optymalne rozwiązanie pierwszego etapu stanowi podstawę do uzyskania optymalnego rozwiązania na każdym kolejnym etapie dla całego problemu

Co oznacza, że **podproblemy nie są rozłączne**?

- Na przykład dla ciągu Fibbonacciego jeśli chcemy wyznaczyć element *n*-ty ($x_n = x_{n-1} + x_{n-2}$), to musimy wyznaczyć wszystkie elementy od x_0 do x_{n-1} (to są nasze podproblemy). Jeśli zapiszemy wszystkie elementy w tablicy (ich wartości się nie zmieniają) to wyznaczając element x_{n+1} możemy bazować na wyznaczonych poprzednio elementach.

- Symbol Newtona $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

  można wyznaczyć wzorem rekurencyjnym $\binom{n}{k} = \begin{cases} 1 & dla\ k = 0\ lub\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & dla\ 0 < k < n \end{cases}$

  Wówczas, zarówno wyznaczenie $\binom{n-1}{k-1}$, jak i $\binom{n-1}{k}$, będzie wymagało wyznaczenia $\binom{n-2}{k-1}$. Niektóre podproblemy byłyby wywołane wykładniczą liczbę razy podczas gdy liczba różnych podproblemów do wyznaczenia jest równa $O(n^2)$.

Co jest potrzebne aby zaprojektować algorytm techniką PD?

Należy sformułować równanie rekurencyjne Bellmana dla problemu (tj. równanie opisujące optymalną wartość funkcji celu dla problemu, które wykorzysta wyznaczone optymalne rozwiązania podproblemów o mniejszych rozmiarach). Poszczególne rozwiązania dla mniejszych podproblemów zapisywane są w tablicy.

Przykłady zastosowań dla problemów:

- Problem znajdowania najdłuższego wspólnego podciągu
- Problem plecakowy
- Problem dopasowania ciągów znakowych (algorytm Needelmana-Wunscha)
- Problem optymalnego nawiasowania macierzy

Złożoność obliczeniowa algorytmów wykorzystujących technikę PD zależy od klasy złożoności problemów, które te algorytmy rozwiązują. PD zazwyczaj stosuje się do rozwiązania problemów trudnych obliczeniowo, aby skrócić czas obliczeń. Można je także stosować do rozwiązania problemów z klasy P.

Binarny problem plecakowy

Dane wejściowe

c – pojemność plecaka
n – liczba elementów w zbiorze
x_i – zmienna decyzyjna (czy i-ty element jest w zbiorze)
p_i – wartość i-tego elementu
w_i – waga i-tego elementu

Algorytm programowania dynamicznego rozwiązuje optymalnie podproblemy (mniej elementów, mniejszy plecak) i na ich podstawie buduje rozwiązanie problemu wejściowego.

Cel

znajdź optymalne upakowanie plecaka (podzbiór przedmiotów, które mieszczą się w plecaku i mają maksymalną sumaryczną wartość)

Algorytm programowania dynamicznego

a) Struktury danych

Macierz kosztów/macierz PD – przechowuje rozwiązania podproblemów (kolejne wartości funkcji celu)
Macierz decyzyjna (opcjonalna) – przechowuje wartości zmiennych decyzyjnych

b) Funkcja rekurencyjna Bellmana (sposób wypełniania macierzy PD)

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Funkcja rekurencyjna Bellmana

$$V[0, j] = 0$$ ← Dodatkowy, zerowy wiersz w macierzy PD (wyzerowany).

$$V[i, 0] = 0$$ ← Dodatkowa, zerowa kolumna w macierzy PD (wyzerowana).

$$V[i, j] = \begin{cases} V[i-1, j] & if \quad w_i > j \\ max\{V[i-1, j], V[i-1, j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Jeśli *i*-ty element nie mieści się w plecaku, weź rozwiązanie optymalne obliczone, gdy tego elementu nie było. $\mathbf{x_i = 0}$

Jeśli *i*-ty element mieści się w plecaku, weź maksimum z wartości:
- Plecaka bez *i*-tego elementu ($\mathbf{x_i = 0}$)
- Plecaka z *i*-tym elementem ($\mathbf{x_i = 1}$). Wówczas w plecaku musi być miejsce na ten element (zrób miejsce: pomniejsz *j* - aktualny rozmiar plecaka o wagę *i*-tego elementu i do rozwiązania optymalnego dodaj wartość *i*-tego elementu).

Algorytm PD: przykład (wypełnianie macierzy kosztów)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

($x_i = 0$ for $V[i-1,j]$; $x_i = 0$ for $V[i-1,j]$, $x_i = 1$ for $V[i-1,j-w_i]$)

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0								
3	0								
4	0								

+4 +4 +4

Wiersz dla elementu nr 1

Wiersz dla elementu nr 2

Wiersz dla elementu nr 3

Wiersz dla elementu nr n

Kolumna j odpowiada plecakowi o pojemności j. Rozpatrujemy co się zmieści w plecaku, jeśli jest dostępnych od 1 do c jednostek jego pojemności.

Algorytm PD: przykład (wypełnianie macierzy kosztów)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 1$ dla $V[i-1,j-w_i]$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0								
4	0								

+3 +3

Komórka V[2,2]. Wybieramy maksimum dwóch wartości:
$\max\{V[1,2], V[1,2-w_2]+p_2\} = \max\{4, V[1,1]+3\} = \max\{4,3\} = 4$

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j]$; $x_i = 1$ dla $V[i-1,j-w_i]$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

A co jest w plecaku?

To jest rozwiązanie optymalne:
maksymalna wartość funkcji celu f^*
(max. suma wartości elementów w plecaku).

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j-w_i]$; $x_i = 1$ dla $V[i-1,j-w_i] + p_i$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

$x^* = \{\mathbf{x_4}\}$

Porównanie wartości V[*i,j*] z wartością V[*i-1,j*] (wiersz powyżej) pozwala stwierdzić czy element *i*-ty jest w rozwiązaniu:

- V[*i,j*] = V[*i-1,j*] → element *i*-ty nie został zabrany; $x_i = 0$
- V[*i,j*] > V[*i-1,j*] → element *i*-ty jest w plecaku; $x_i = 1$

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j],\, V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j]$; $x_i = 1$ dla $V[i-1,j-w_i]$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

$x^* = \{x_4\}$

- Idziemy do V[*i-1,j-w*$_i$], na podstawie której wyznaczyliśmy V[*i,j*].
- Czy x_3 jest w rozwiązaniu? V[3,4] = V[2,4], więc x_3 nie jest w rozwiązaniu. Przesuwamy się o wiersz w górę, sprawdzamy czy x_2 jest w rozwiązaniu.

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j]$; $x_i = 1$ dla $V[i-1,j-w_i]$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

$x^* = \{\mathbf{x_2}, x_4\}$

- V[2,4] > V[1,4], więc x_2 jest w rozwiązaniu.
- Przesuwamy się o wiersz w górę, sprawdzamy czy x_1 jest w rozwiązaniu.

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j],\, V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

($x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j-w_i]$; $x_i = 1$ dla $V[i-1,j-w_i]$)

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

$x^* = \{\mathbf{x_1}, x_2, x_4\}$

- V[1,3] > V[0,3], więc x_1 jest w rozwiązaniu.

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j],\, V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j-w_i]$; $x_i = 1$ dla $+ p_i$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie

$f^* = 15$

$x^* = \{x_1, x_2, x_4\}$

Rozwiązanie można też zapisać w postaci wektora X=[1,1,0,1]

- Doszliśmy do wiersza zerowego. Koniec.

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8

n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \le j \end{cases}$$

$x_i = 0$ dla $V[i-1,j]$; $x_i = 0$ dla $V[i-1,j-w_i]$; $x_i = 1$ dla $V[i-1,j-w_i]+p_i$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15
			x_1		x_2	x_4			

Rozwiązanie

$f^* = 15$

$x^* = \{x_1, x_2, x_4\}$

x_1 zajął 2 jednostki pojemności plecaka, x_2 – 1 jednostkę, x_4 – 4 jednostki.

Algorytm PD: przykład (odczytywanie rozwiązania)

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Rozwiązanie
$f^* = 15$
$x^* = \{x_1, x_2, x_4\}$

Macierz decyzyjna X

i \ j	1	2	3	4	5	6	7	8
1	0	1	1	1	1	1	1	1
2	1	0	1	1	1	1	1	1
3	0	0	0	0	1	1	1	1
4	0	0	0	1	1	1	1	1

Łatwo jest znaleźć rozwiązanie gdy mamy dodatkową **macierz decyzyjną**:

- $X[i,j]=1 \rightarrow x_i \in x^*$. Idź do $X[i-1, j-w_i]$.
- $X[i,j]=0 \rightarrow x_i \notin x^*$. Idź do $X[i-1,j]$.

Ale macierz ta pochłania dodatkową pamięć.

Algorytm PD: co jeszcze możemy odczytać z macierzy kosztów?

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Ta macierz PD zawiera **optymalne rozwiązania podproblemów** naszego problemu plecakowego:

- Rozwiązanie naszego problemu dla plecaka o dowolnej pojemności $c \in \{1, 2, 3, 4, 5, 6, 7\}$
- Rozwiązanie problemu dla podzbioru elementów {1, 2, 3} albo dla podzbioru {1, 2}.

Algorytm PD: co jeszcze możemy odczytać z macierzy kosztów?

Dane wejściowe

c = 8
n = 4

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15

Przykładowo odczytajmy rozwiązanie podproblemu:
c = 6
n = 3

id	w_i	p_i
1	2	4
2	1	3
3	4	6

Rozwiązanie
$f^* = 10$
$x^* = \{x_1, x_3\}$

Algorytm PD: co zrobić jeśli w zbiorze elementów pojawi się dodatkowy element?

Dane wejściowe

c = 8

n = 5

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8
5	3	7

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7	7
3	0	3	4	7	7	9	10	13	13
4	0	3	4	7	8	11	12	15	15
5									

Dodajemy kolejny wiersz i obliczamy wartości komórek w tym wierszu.

Algorytm PD: co zrobić jeśli chcemy powiększyć plecak?

Dane wejściowe

c = 10

n = 5

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8
5	3	7

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0		
1	0	0	4	4	4	4	4	4	4		
2	0	3	4	7	7	7	7	7	7		
3	0	3	4	7	7	9	10	13	13		
4	0	3	4	7	8	11	12	15	15		
5											

Dodajemy kolejne kolumny i obliczamy wartości komórek w tych kolumnach.

Algorytm PD: czasami problem ma więcej niż jedno rozwiązanie optymalne

Dane wejściowe

c = 7

n = 5

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8
5	3	7

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7
3	0	3	4	7	7	9	10	13
4	0	3	4	7	8	11	12	15
5	0	0	0	7	10	11	14	15

V[*i-1,j*] = V[*i-1,j-w*$_i$]+*p*$_i$

V[*i-1,j*] = V[*i-1,j-w*$_i$]+*p*$_i$, czyli mamy **więcej niż jedno rozwiązanie optymalne**:

- rozwiązanie z *i*-tym elementem w plecaku ($x_i \in x^*$);
- rozwiązanie bez *i*-tego elementu w plecaku ($x_i \notin x^*$).

Algorytm PD: czasami problem ma więcej niż jedno rozwiązanie optymalne

Dane wejściowe

c = 7

n = 5

id	w_i	p_i
1	2	4
2	1	3
3	4	6
4	4	8
5	3	7

$$V[0,j] = V[i,0] = 0$$

$$V[i,j] = \begin{cases} V[i-1,j] & if \quad w_i > j \\ \max\{V[i-1,j], V[i-1,j-w_i] + p_i\} & if \quad w_i \leq j \end{cases}$$

Macierz programowania dynamicznego V

i \ j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4
2	0	3	4	7	7	7	7	7
3	0	3	4	7	7	9	10	13
4	0	3	4	7	8	11	12	15
5	0	0	0	7	10	11	14	15

$f^* = 15$

Możemy odczytać wszystkie rozwiązania, jeśli zapamiętaliśmy gdzieś, że była taka sytuacja (np. w macierzy decyzyjnej, która wtedy nie może być boolowska).

W optymalnym plecaku mamy: $x_A^* = \{x_4, x_5\}$ lub $x_B^* = \{x_1, x_2, x_4\}$

Złożoność obliczeniowa i pamięciowa algorytmów dla problemu plecakowego

Złożoność obliczeniowa

Algorytm zachłanny
Złożoność algorytmu zachłannego jest taka jak złożoność algorytmu sortowania: $O(n \cdot \log_2 n)$

sortowanie $O(n \cdot \log_2 n)$
+ budowanie rozwiązania po posortowaniu $O(n)$
$= (n \cdot \log_2 n) + n = n \cdot (\log_2 n + 1)$

W notacji O (pomijamy stałe): $O(n \cdot \log_2 n)$

Algorytm wyczerpujący/siłowy
Złożoność wykładnicza: $O(2^n)$

Algorytm programowania dynamicznego
Złożoność pseudowielomianowa: $O(n \cdot c)$

Złożoność pamięciowa

Algorytm zachłanny
Złożoność liniowa: $O(n)$

Algorytm wyczerpujący/siłowy
Złożoność liniowa: $O(n)$

Algorytm programowania dynamicznego
Złożoność pseudowielomianowa: $O(n \cdot c)$