

Programowanie
Dynamiczne

LCS (longest common subsequence)

Dla dwóch ciągów chcemy znaleźć jego wspólny podciąg
 np. dla $a = "abc"$ oraz $b = "acd"$ wtedy podciąg to " ac "

Algorytm LCS znajduje taki podciąg wypełniając tabelę $m \times n$.

\diagdown	a	b	c
b	0	0	0
b	0	0	1
c	0	0	1

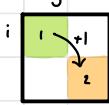
Pierwszy wiersz i kolumnę wypełniamy zerami, które oznaczają, że dla $a[i]$ i $b[i]$ długość w której któryś z nich jest 0, maksymalna długość naszego podciągu też jest równa zero.

Tabela ta przechowuje informację o długości najbliższego wspólnego podciągu w punkcie (i, j) dla $a = [0 \dots i]$ oraz $b = [0 \dots j]$.

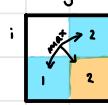
Algorytm:

1. Iterujemy po tabeli wierszami.

2. Jeżeli $a[i] == b[j]$ wpp.



$$t[i][j] = t[i-1][j-1]$$



$$t[i][j] = \max(t[i-1][j], t[i][j-1])$$

Imaginij stonę

- węź długość ciągu $a[i:j+1]$ mniejszą, a następnie dodaj 1
- węź max długość LCS dla $a[0..m-1]$: $b[0..n]$ lub $a[0..m]$: $b[0..n-1]$

3. Odczytujemy ciąg z prawego dolnego rogu do górnego lewego.

\diagdown	a	b	c
b	0	0	0
b	0	1	1
c	0	1	1

Wynik: **b** **c**

Tam gdzie stonki są ukośnie - tam jest masza litera.

Dowodzenie algorytmów dynamicznych

Pokaż teraz jak udowodnić algorytm LCS.

Aby udowodnić poprawność algorytmu dynamicznego, wystarczy wykonać:

1. Założenia Startowe

$$\begin{aligned} dp[i][0] &= 0 \\ dp[0][j] &= 0 \end{aligned}$$

2. Wyjaśnienie relacji między stanami:

$dp[i][j]$ - wartość LCS dla podciągu $a[0..i]$ $b[0..j]$

3. Wyjaśnienie procesu obliczania nowego stanu

$$dp[i][j] = \text{if } a[i] == b[j]$$

then $dp[i-1][j-1] + 1$

else $\max(dp[i-1][j], dp[i][j-1])$

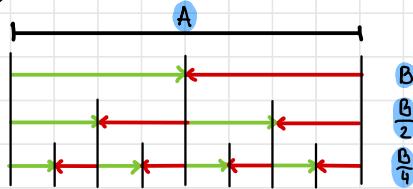
LCS pamięć $O(n)$ z odzyskiem

Dla samego znalezienia długości najdłuższego podciągu w $\mathcal{O}(n)$ wystarczy z ostatniego wiersza.

Metoda Hirschberga

Nasz oryginalny algorytm LCS idzie od lewej do prawej. Jednakże równie dobrze może być iterowany od prawej do lewej - nic to nie zmieni.

Wykorzystajmy metodę dziel i zwyciężaj oraz ten fakt do odzysku w taki sposób:



$LCS(A, B)$

$$m = \frac{i+j}{2}$$

$left = LCS_front(A[..mid], B)$

zwycięzcy LCS $O(n)$ pamięciowy

czyli pozycje w B

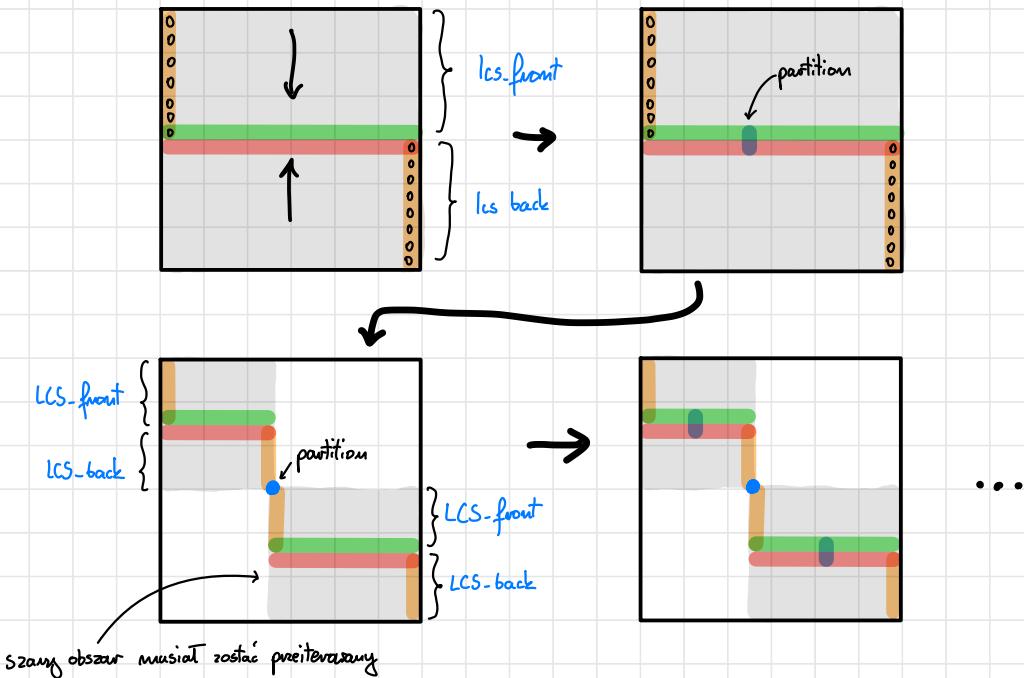
$right = LCS_back(A[mid..], B.reverse())$

Funkcja `max` zwraca x dla którego ta wartość jest maksymalna

$partition = \max(\forall_{0 \leq x \leq B}, \text{by: } left[x] + right[-1-x])$

$return LCS(A[..mid], B[..partition]) + LCS(A[mid..], B[partition..])$

Zauważamy, że algorytm utrzyma liniowość pamięci



Kod Algorytmu:

```

def lcs_hirschberg(A, B):
    # Przypadek gdy A jest puste
    if len(A) == 0:
        return []
    # Przypadek gdy A jest jedno-literowe
    elif len(A) == 1:
        if A[0] in B:
            return [A[0]]
        else:
            return []
    # Oblicz wynik dla lewego i prawego
    else:
        mid = len(A) // 2
        # Oblicz zwykłą LCS dla lewej połowy A i całego B
        score_left = lcs_score(A[:mid], B)
        # Oblicz zwykłą LCS dla prawej połowy A i całego B
        score_right = lcs_score(A[mid:][::-1], B[::-1])
        # Znajdź miejsce gdzie wiemy, że jest nowa litera
        partition = max(range(len(B) + 1), key=lambda x: score_left[x] + score_right[-1 - x])
        # Wywołaj algorytm rekurencyjnie na lewej i prawej połowie A dla B z odpowiednim podziałem
        return lcs_hirschberg(A[:mid], B[:partition]) + lcs_hirschberg(A[mid:], B[partition:])

# Zwykły LCS dla liniowej pamięci
def lcs_score(A, B):
    scores = [[0 for _ in range(len(B) + 1)] for _ in range(2)]
    for i in range(1, len(A) + 1):
        for j in range(1, len(B) + 1):
            if A[i - 1] == B[j - 1]:
                scores[i % 2][j] = scores[(i - 1) % 2][j - 1] + 1
            else:
                scores[i % 2][j] = max(scores[(i - 1) % 2][j], scores[i % 2][j - 1])
    return scores[len(A) % 2]

```

LPS (longest palindromic sequence)

Palindrom ma tę właściwość, że czytany od tyłu jest taki sam

abba
✓

abta
✗

W naszym algorytmie będziemy iterować zwiększając przedział i, j .
 $dp[i][j]$ - długość najdłuższego podcięgu palindromicznego dla przedziału od i do j .

1. Inicjalizujemy tablicę dp $m \times m$ wypełniając ją zerami, a następnie wypełniając przekątną jedynkami.

Robimy tak, ponieważ każda litera jest palindromem o długości 1.

2. Iterujemy wzajemnie kolejnych przekątnych wykonując:

$$dp[i][j] = \begin{cases} dp[i+1][j-1] + 2 & \text{dla } val(i) = val(j) \leftarrow \begin{matrix} i\text{-ta} \\ j\text{-ta litera} \\ \text{się zjadza} \end{matrix} \\ \max(dp[i][j+1], dp[i-1][j]) & \text{wpp.} \end{cases}$$

Innymi słowy - weź maks palindrom z $a \underline{\quad} a$ i dodaj 2 jeśli możemy go rozszerzyć, wpp. weź największy palindrom jaki udowodniliśmy w $a \underline{\quad} b$. Wiemy, że tam są poprawne wartości, ponieważ sprawdzamy najpiękniejsze przedziały równe 2, 3, ... itd.

Algorytm iteracyjny jest w ten sposób:

j\i	B	A	B	B	A	B
1	1	3	3	4	6	
1	1	2	4	4		
1	2	2	3			
	1	1	3			
	1	1				
						1



- I iteracja - sprawdzany ciągi 2-znakowe
- II iteracja - sprawdzany ciągi 3-znakowe
- III iteracja - sprawdzany ciągi 4-znakowe
- IV iteracja - sprawdzany ciągi 5-znakowe
- V iteracja - sprawdzany ciągi 6-znakowe

Aby odczytać wartość - możemy w trakcie wykonywania algorytmu zapisywać gdzie (wartości i, j) po raz pierwszy pojawiła się dotychczasowa maksymalna wartość.

Ten algorytm wymaga, aby ponizej przekątnej przechowywać zera, bowiem tam znajdują się wartości informujące nas o podążaniu zerowym między literami. Np. a b c d itd.

Pamięć limiowa

Mozemy zoptymalizować ten algorytm wykonując obserwacje, że jedynie 3 ostatnie przekątne są używane w każdej iteracji.

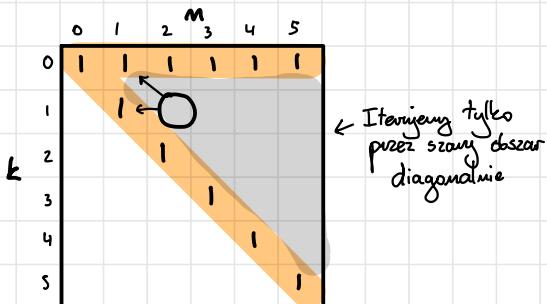
Dwumian Newtona

$$\binom{m}{k} = \frac{m!}{k!(m-k)!}$$

Rekurencja: $\binom{m}{k} = \begin{cases} 1 & \text{dla } k=0 \vee k=m \\ \binom{m-1}{k-1} + \binom{m-1}{k} & \text{dla } 1 < k < m \end{cases}$

Bardzo podobnie do poprzednich zadań.

Mozemy zapisywać $dp[m][k]$ i wykonać rekurencję z memoizacją.



Algorytm

$$dp[k][m] = 1 \quad \text{dla } k=0 \text{ lub } k=m$$

$$dp[k][m] - wartość \binom{m}{k}$$

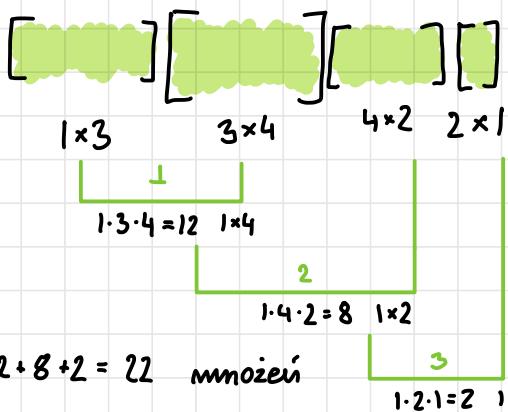
$$dp[k][m] = dp[k-1][m-1] + dp[k][m-1]$$

Optymalna kolejność mnożenia macierzy

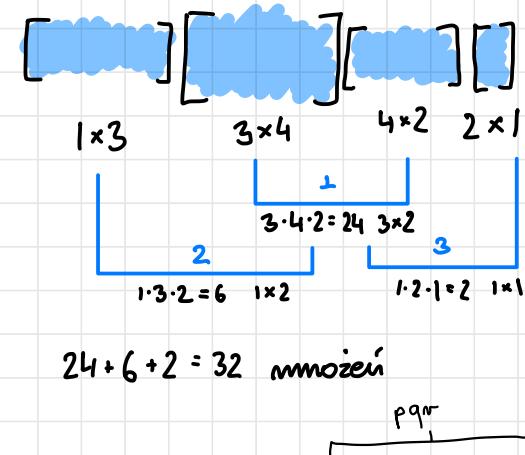
Koszt mnożenia każdej macierzy $p \times q$ przez $q \times r$ wynosi

$$p \cdot q \cdot r$$

1.

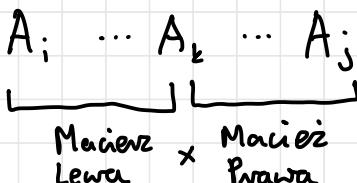


2.



Koszt mnożenia dwóch macierzy: $\text{koszt}(a, b) = a.h \cdot a.w \cdot b.w$

Interesuje nas taki podział k , że:



Koszt przemnożenia tych macierzy jest najmniejszy.

$dp[i][j]$ - Najtańsze przemnożenie macierzy od i do j

$$dp[i][j] = 0 \text{ dla } i=j$$

$$dp[i][j] = \min \left(\begin{array}{c} \text{koszt} \\ \text{przemnożenia} \\ \text{tych macierzy} \end{array} \right)$$

koszt wyliczenia tych macierzy

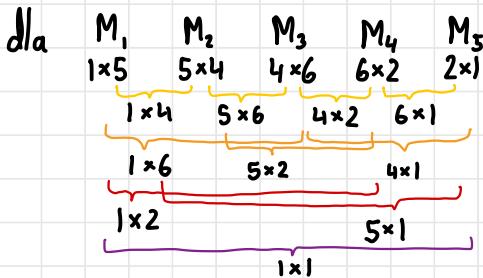
$$\text{koszt} (dp[i][k], dp[k+1][j]) + dp[i][k] + dp[k+1][j]$$

for k from $i+1$ to $j-1$

albo $d_{i-1} \cdot d_k \cdot d_j$ jeśli:

$$\begin{array}{ccccc} M_1 & M_2 & M_3 & M_4 & M_5 \\ 1 \times 5 & 5 \times 4 & 4 \times 6 & 6 \times 2 & 2 \times 1 \\ d_0 & d_1 & d_2 & d_3 & d_4 & d_5 \end{array}$$

$j \setminus i$	M_1	M_2	M_3	M_4	M_5
M_1	0	20	44	56	58
M_2	0	120	38	56	
M_3		0	48	36	
M_4			0	12	
M_5				0	



Odpis: Podczas wracania rekurencją do góry z wynikiem zebrać odpowiedzi.

Powyższe rozwiązywanie działa w czasie $O(n^3)$ oraz w życzliwości pamięciowej $O(n^2)$ z możliwą redukcją do $O(n)$.

Problem Plecakowy



Dyskretny

Wtedy musimy podejmować decyzję co spakować a co nie. Algorytm jest bardzo podobny jak do tego z wydawaniem meszy w wersji programowania dynamicznego.



$$O(n^{|V|})$$

↑ Rozmiar plecaka

Ciągły

Możemy spakować również ułamkowe części - tutaj algorytm zachowany już się aplikuje. Sortujemy przedmioty; po kolei wybieramy.

$$\text{Złożoność } O(n \log n)$$

Algorytm pseudowielomianowy – [algorytm](#), którego złożoność obliczeniowa jest pseudowielomianowa. Oznacza to, że zależy ona nie tylko od rozmiaru danych wejściowych, ale również od pewnego parametru charakterystycznego dla danego problemu.

Przykładowo dla problemu plecakowego istnieje algorytm pseudowielomianowy który go rozwiązuje w czasie $\Theta(n \cdot k)$, gdzie n to rozmiar danych wejściowych, a k to rozmiar plecaka.

Problem plecakowy dyskretny jest algorytmem pseudowielomianowym

Duszkowmy bez powtórzeń

Idea:

Jesli przedmiot mieści się w plecaku

then) Sprawdź co jest lepsze

a) Spakować ten przedmiot oraz
to co się mieści w plecaku mniejszym
o wagę tego przedmiotu

b) Spakować tyle co gdybyśmy tego
przedmiotu nie uwzględniali

else) Spakuj tyle co gdybyśmy tego
przedmiotu nie uwzględniali

waga	wartość	indeks	pojemność plecaka					
w	v	j	0	1	2	3	4	...
-	-	0	0	0	0	0	0	
4	5	1	0	0	0	0	5	
3	4	2	0	0	0	4	5	
2	3	3	0	0	3	4	5	
1	2	4	0	2	5	6	6	

$$\begin{aligned} dp[0][j] &= 0 \\ dp[i][0] &= 0 \end{aligned}$$

$dp[i][j]$ - maksymalne spakowanie plecaka o pojemności j
dla i przedmiotów

$$dp[i][j] = \text{if } w[i] \leq j$$

$$\text{then } dp[i][j] = \max (dp[i-1][j-w[i]] + v[i], dp[i-1][j])$$

$$\text{else } dp[i][j] = dp[i-1][j]$$

Dyskretny z powtórzeniami

Idea:

a) Spakuj tyle co zdobyśmy tego elementu nie uwzględniając

b) Sprawdzamy ile kosztowałby nas plecak zdobytych unieshi ten przedmiot + max wartości elementów w plecaku bez tego przedmiotu.

Tablica jest ma podstawię poprzedniej tablicy

size	0	1	2	3	4	5
(czyli: 5×2)	0	2	4	6	8	10

ponieważ sprawdzamy wszystkie elementy w pętli - potrzebujemy tablicy jednowymiarowej.

$$dp[i] = 0$$

$dp[i]$ - maksymalna wartość spakowania dla plecaka i -tego rozmiaru

$$dp[i] = \max_{\forall item < i} (dp[:], dp[i - w[item]] + v[item])$$

Kod:

for item in items :
 if $w[item] < i$

$$dp[i] = \max (dp[i], dp[i - w[item]] + v[item])$$

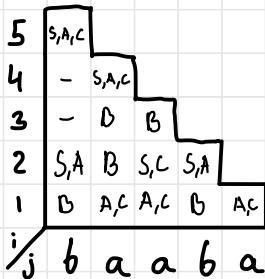
↑ maksymalna wartość otrzymana w poprzedniej iteracji

aktualny przedmiot + mniejszy plecak

CYK

Sprawdzanie czy wyrażanie należy do gramatyki bezkontekstowej:

$$\begin{aligned} S &\rightarrow AB \mid BC \\ A &\rightarrow BA \mid a \\ B &\rightarrow CC \mid b \\ C &\rightarrow AB \mid a \end{aligned}$$



$dp[i][j]$ - W jaki sposób możemy otrzymać podwyrażenie stara od j do $j+i$.

$dp[0][j]$ = znajdijemy wyrażenia końcowe w regułach i zapisujemy. Np. dla a mamy $A : C$ ponieważ

$$\begin{aligned} A &\rightarrow BA \mid a \\ C &\rightarrow AB \mid a \end{aligned}$$

$dp[i][j]$ = Weź wszystkie podstawa oraz sprawdź jakie mogą spełniać

Przykład

$$dp[4][1] = \left\{ \begin{matrix} a, aba \\ aa, ba \\ aab, a \end{matrix} \right.$$

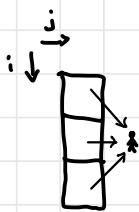
$$\begin{array}{c} \text{"ilorazy kartezańskie"} \\ \downarrow A, C \quad \downarrow \quad \downarrow B \quad \downarrow \\ dp[1][1] \times dp[3][2] = AB, CB = S, C \end{array}$$

Istnieje w gramatyce (S, C)

↑
Nie istnieje w gramatyce
 $A, C \times B = AB, CB$

Optymalna Trasa

Otrzymujemy tablicę dwuwymiarową z wagami i obliczamy trasę, która jest najtańsza idąc z lewego-górnego do prawnego-dolnego węzła.



$$dp[0][0] = \text{value}[0][0]$$

$dp[:, j]$ - najtańszy krok by tutaj dotrzeć

$$dp[i][j] = \min \left(\begin{array}{l} dp[i-1][j-1], \\ dp[i][j-1], \\ dp[i+1][j-1] \end{array} \right) + \text{value}[i][j]$$