Metody rozwiązywania problemów

Beata Laszkiewicz

Część III

PROGRAMOWANIE DYNAMICZNE

Problem plecakowy

Mamy danych n przedmiotów:

- każdy o ustalonej wartości $p_i > 0, i = 1, \ldots, n$
- oraz o ustalonej masie $w_i > 0, i = 1, \ldots, n$,

oraz plecak, który może pomieścić W kilogramów, $W > 0$.

Naszym zadaniem jest wybrać pewną liczbę przedmiotów tak, by w plecaku pozostało jak najmniej wolnego miejsca oraz by jego wartość była jak największa.

Zadanie optymalizacyjne

Przykład:

Niech $W = 23, n = 4$, wartości i masy przedmiotów przedstawiono w tabeli:

i	1	2	3	4
p_i	2	5	7	10
w_i	1	3	2	3
$\frac{p_i}{w_i}$	2	$1\frac{2}{3}$	$3\frac{1}{2}$	$3\frac{1}{3}$

Czy strategie dają optymalne rozwiązanie?

Patrząc na przykład można zauważyć, że zastosowanie strategii III nie daje optymalnego rozwiązania - można zapakować plecak tak, aby jego wartość była równa **80**.

Programowanie dynamiczne

- Metoda programowania dynamicznego pozwoli nam uzyskać optymalne rozwiązanie:
- Rozwiązujemy wszystkie mniejsze podproblemy.
  W przypadku problemu plecakowego oryginalny problem można zredukować na dwa sposoby:
  - zmniejszamy zestaw przedmiotów (patrzymy tylko na przedmioty $1, 2, \dots, i$ dla $i \leq n$),
  - zmniejszamy pojemność plecaka (do $j \leq W$).
- Rozwiązania wszystkich podproblemów możemy w prosty sposób zapamiętać w tablicy dwuwymiarowej.

Programowanie dynamiczne

- $P[i][j]$ oznacza najlepszą wartość plecaka, którego waga W nie przekracza j, zapakowanego pewną kombinacją przedmiotów o numerach od 1 do i.
- Rozwiązanie problemu przechowuje komórka $P[n][W]$.

Tabela – pierwszy przedmiot

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0																							
3	0																							
4	0																							

- Już dla W=1 można wypełnić plecak przedmiotem nr 1.
- Nie ma żadnych problemów, dla kolejnych j = 1, 2, ..., W dokładamy przedmiot do plecaka (waży tylko 1 kg).
- Warto zauważyć, że gdy plecak ma W = 0, to nie ma on żadnej wartości (wypełniona zerami zerowa kolumna).

Tabela – drugi przedmiot

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	**6**	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4																					
3	0																							
4	0																							

+5

- Drugi przedmiot można dołożyć dopiero wtedy, gdy waga plecaka W=3; dlatego wcześniej pakujemy tylko przedmiot nr 1.
- Dla j=3 zastanawiamy się:
  - Czy lepiej dołożyć przedmiot nr 2 do plecaka (trzeba coś wyciągnąć i zrobić miejsce),
  - czy brać poprzednie upakowanie...

Tabela – drugi przedmiot

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6																				
3	0																							
4	0																							

+5 +5 +5 +5

- Dla kolejnych możliwych wag plecaka (j=4, 5, ...) postępujemy analogicznie...
  - Czy lepiej dołożyć przedmiot nr 2 do plecaka (trzeba coś wyciągnąć i zrobić miejsce),
  - czy brać poprzednie upakowanie...

Po analizie dwóch przedmiotów

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
3	0																							
4	0																							

Trzeci przedmiot

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
3	0	2																						
4	0	+7																						

- Trzeci przedmiot można dołożyć dopiero wtedy, gdy waga plecaka W=2; dlatego wcześniej wybieramy wcześniejsze upakowanie plecaka.
- Analogicznie jak w poprzednim kroku zastanawiamy się (do j=2, 3, ...)
  - Czy lepiej dołożyć przedmiot nr 3 do plecaka (trzeba coś wyciągnąć i zrobić miejsce),
  - czy brać poprzednie upakowanie...

Tabela po analizie trzeciego przedmiotu

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0																							
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
3	0	2	7	9	14	16	21	23	28	30	35	37	42	44	49	51	56	58	63	65	70	72	77	79
4	0																							

- Analogicznie postępujemy z przedmiotem nr 4....

Efekt końcowy

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
3	0	2	7	9	14	16	21	23	28	30	35	37	42	44	49	51	56	58	63	65	70	72	77	79
4	0	2	7	10	14	17	21	24	28	31	35	38	42	45	49	52	56	59	63	66	70	73	77	**80**

Do czego jest potrzebny zerowy wiersz?

- Żeby napisać porządny algorytm, który pracowałby również dla przedmiotu o numerze 1 (w kolejnych krokach odwołujemy się do wartości P[i][j] z wcześniejszych wierszy...)
- Wiersz musi być wyzerowany....

Algorytm:

Dane:

n - liczba przedmiotów

p[1..n] - tablica wartości kolejnych przedmiotów

w[1..n] - tablica mas kolejnych przedmiotów

W - maksymalna pojemność plecaka (W jednostek)

Wynik:

P[0..n][0..W] - tablica najlepszych "upakowań" plecaka

```
for j = 0, 1, 2, ..., W
   P[0][j] = 0;
for i = 0, 1, 2, ..., n
   P[i][0] = 0;

for i = 1, 2, ..., n
   for j = 1, 2, 3, ..., W
      if (j≥w[i] and P[i-1][j]<P[i][j-w[i]]+p[i]) //(*)
         P[i][j] = P[i][j-w[i]]+p[i];
      else
         P[i][j] = P[i-1][j];
```

(*) – sprawdzamy, co się bardziej opłaca:
jeśli masa plecaka jest nie mniejsza niż masa i-tego przedmiotu oraz opłaca się dołożyć i-ty przedmiot – dokładamy go, zwiększając odpowiednio wartość plecaka.

Programowanie dynamiczne

- Metoda stosowana zwykle do problemów optymalizacyjnych, jej istotą jest obliczenie rozwiązania wszystkich podproblemów optymalizacyjnych, zaczynając od problemów małych, a kończąc na większych.
- Nie potrafimy przewidzieć, z których rozwiązań mniejszych podproblemów będziemy korzystać w kolejnych krokach i dlatego przygotowujemy się na każdą ewentualność – rozwiązujemy wszystkie mniejsze podproblemy.
- Obliczone rozwiązania są najczęściej zapisywane w tablicy, dzięki czemu dostęp do nich jest bardzo szybki.

Programowanie dynamiczne

- Programowanie dynamiczne poprawia inne metody (np. dziel i zwyciężaj) w sytuacji, kiedy wymagają one wielokrotnego liczenia rozwiązań tych samych podproblemów.
- Żaden z podproblemów nie jest rozwiązywany wielokrotnie.
- Metoda dzieli problem na podproblemy w taki sposób, by rozwiązanie optymalne było łatwo osiągalne z optymalnych rozwiązań podproblemów.

Programowanie dynamiczne a problem plecakowy

- Mniejsze podproblemy odpowiadają mniejszej wadze plecaka i mniejszej liczbie przedmiotów.

Zasady optymalności Bellmana

Na każdym kroku podejmować najlepszą decyzję
z uwzględnieniem stanu wynikającego
z poprzednich decyzji.

Złożoność

- Algorytm zachłanny: $O(n \log_2 n)$
- Programowanie dynamiczne: nW
  - zależy od liczby danych i wartości jednej z nich (co jest charakterystyczne dla programowania dynamicznego),
  - algorytm efektywny, jeśli W nie jest za duże.

Jak poznać numery przedmiotów?

- Można utworzyć dodatkową tablicę Q, skojarzoną z tablicą P.
- Można próbować odzyskać numery wybranych przedmiotów analizując tablicę P.

Symulacja

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
2	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46
3	0	2	7	9	14	16	21	23	28	30	35	37	42	44	49	51	56	58	63	65	70	72	77	79
4	0	2	[illegible]	10	[illegible]	17	[illegible]	24	[illegible]	31	[illegible]	38	[illegible]	45	[illegible]	52	56	59	[illegible]	66	70	73	77	**80**

Algorytm odzyskiwania numerów przedmiotów

```
i = n
j = W
while (j > 0)
      if (P[i][j] == P[i-1][j])
            i = i-1
      else
            wypisz nr i
            j = j – w[i]
```

Obliczanie $\binom{n}{k}$

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

```
wsp (n,k)
if (n=k or k=0) return 1
else return wsp(n-1,k)+wsp(n-1, k-1)
```

- Wielokrotne rozwiązywanie tych samych problemów
- Złożoność wykładnicza

Jak wypełniamy tablicę?

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Programowanie dynamiczne:

```
for i = 1, 2, ..., n
      t[i][0] = 1

for j = 1, 2, ..., k
      t[j][j] = 1
      for i = j+1, j+2, ..., n
            t[i][j]=t[i-1][j-1]+t[i-1][j]
```

- dużą zaletą tej metody jest to, że obliczamy wszystkie wartości symbolu Newtona w zadanym przedziale,
- po obliczeniu wszystkich wartości możemy tylko odczytywać wyniki z tablicy (jest to bardzo dobra optymalizacja, jeżeli ilości takich sprawdzeń symbolu Newtona byłaby bardzo duża).

Dodatkowe oszczędności

- możemy znacznie zredukować koszty pamięciowe:
  - obliczenie kolejnej przekątnej trójkąta Pascala wymaga znajomości jedynie wartości z poprzedniej przekątnej.
  - zamiast tablicy n×k wystarcza tablica n×2, a nawet tablica n×1.

Koncepcja: rozwiązuj problem "od końca"

- rozwiąż problem dla jednego elementu, przy różnych wartościach parametru sterującego, zapamiętaj wyniki,
- dodaj następny element do problemu,
- zbuduj rozwiązanie problemu powiększonego o nowy składnik, dokonując optymalnego wyboru w oparciu o poprzednio zapisane rozwiązania.

Jak wykorzystać tablicę jednowymiarową?

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

Problem do rozwiązania: Truskawkowe pole

Rolnicza spółdzielnia owocowo-warzywna TuttiFrutti organizuje truskawkowe żniwa. W celu oszacowania zbiorów firma wdraża satelitarny system prognozowania zbiorów. Truskawki rosną na polach, tworząc plantacje w kształcie prostokąta. Znając szacowaną wielkość zbioru na każdym polu, oblicz maksymalny zbiór jaki może zebrać truskawkowy kombajn, zakładając, że przemierza on plantacje z pola początkowego (lewy górny róg), poruszając się jedynie w dół lub w prawo, do pola końcowego (prawy dolny róg plantacji).

1	3	7	2	2
8	2	4	8	7
8	4	9	7	1
5	7	1	3	4

Przykładowy rozkład truskawek na polu

Zapamiętaj!

- Istotą metody jest obliczenie rozwiązania wszystkich podproblemów, zaczynając od problemów małych, a kończąc na większych.
- Nie potrafimy przewidzieć, z których rozwiązań mniejszych podproblemów będziemy korzystać w kolejnych krokach i dlatego przygotowujemy się na każdą ewentualność – rozwiązujemy wszystkie mniejsze podproblemy.
- Obliczone rozwiązania są najczęściej zapisywane w tablicy, dzięki czemu dostęp do nich jest bardzo szybki.
- Żaden z podproblemów nie jest rozwiązywany wielokrotnie.
- Na każdym kroku podejmować najlepszą decyzję z uwzględnieniem stanu wynikającego z poprzednich decyzji.

Część III

PROGRAMOWANIE DYNAMICZNE – MATERIAŁ DODATKOWY DLA CHĘTNYCH

Najdłuższy wspólny podciąg

Niech W_1 i W_2 będą dwoma słowami (ciągami znaków). Mówimy, że W_1 jest **podciągiem** W_2 wtedy i tylko wtedy, gdy istnieje taki rosnący ciąg liczb naturalnych x_i, że zachodzi:

$$W_1[i] = W_2[x_i]$$

dla wszystkich $i = 1, 2, \dots, n$, gdzie n to długość słowa W_1, a $W_1[i]$ oznacza i-ty znak z W_1.

W prostszych słowach, W_1 jest podciągiem W_2, jeśli potrafimy tak wybierać z W_2 kolejne znaki, aby utworzyć wyraz W_1, np.

$$W_1 = ZIMA; W_2 = PRZEZIMOWA\acute{C}$$

Przyjęte oznaczenia

Jeśli $X = (x_1, x_2, \dots, x_m)$, to przez X_i oznaczamy podciąg $(x_1, x_2, \dots, x_i)$ $(i = 1, 2, \dots m)$. Dla wygody przez X_0 oznaczamy ciąg pusty.

Najdłuższy wspólny podciąg oznaczamy:

$$Z = LCS(X, Y)$$

Z jest wspólnym podciągiem *X* i *Y* o maksymalnej długości (ang. *longest common subsequnc*e)

Przykład

X=RABARBAR;   Y=LABRADOR

Przykłady różnych podciągów:

X=**RA**BARBAR;   Y=LAB**RA**DOR

X=**RA**BARBA**R**;   Y=LAB**RA**DO**R**

X=RA**BAR**BAR;   Y=LA**B**R**A**DO**R**

Najdłuższy wspólny podciąg (dł. 5):

X=R**AB**A**R**B**AR**;   Y=L**ABRA**DO**R**

Lemat

Niech $X = (x_1, x_2, \ldots, x_m), Y = (y_1, y_2, .., y_n)$ i niech $Z = (z_1, z_2, \ldots, z_k)$ będzie ciągiem z LCS(X,Y).

Wówczas:

- Jeśli $x_m = y_n$, to $z_k = y_n$ i $Z_{k-1} \in LCS(X_{m-1}, Y_{n-1})$.
- Jeśli $x_m \neq y_n$, to $z_k \neq x_m$, więc $Z \in LCS(X_{m-1}, Y)$.
- Jeśli $x_m \neq y_n$, to $z_k \neq y_n$, więc $Z \in LCS(X, Y_{n-1})$.

Wyjaśnienie lematu

Załóżmy, że znamy $LCS(X_i, Y_j)$.

- Jeżeli $x_{i+1} = z$ oraz $y_{i+1} = z$, to LCS dla tych dwóch ciągów może zostać utworzony poprzez dodanie znaku z na koniec LCS dla wyrazów X_i oraz Y_j.

Przykład (przedłużamy LCS o wspólny znak R):

R**AB**A**R**B**A** **R**
L**ABRA**DO **R**

Wyjaśnienie lematu – c.d.

- Jeżeli $x_{i+1} = z$, a $y_{j+1} = w$ oraz $w \neq z$, to $LCS(X_{i+1}, Y_{j+1})$ jest też *LCS* dla jednej z par: X_i, Y_{j+1} lub X_{i+1}, Y_j.

Ostatnią literką najdłuższego wspólnego podciągu dla X_{i+1}, Y_{j+1} nie mogą być jednocześnie z i w (bo to dwa różne znaki). Zatem jeden z ciągów: X_{i+1}, Y_{j+1} możemy bez żadnej straty pozbawić jego ostatniego znaku. Nie wiemy jednak, którego z nich, zatem sprawdzimy obie możliwości i wybieramy tę, która daje lepszy wynik.

Przykład (pozbywamy się litery K):

B**ATONI** K
AN**TONI**

Wniosek

Niech $c_{i,j}$ oznacza długość elementów z $LCS(X_i, Y_j)$.
Wówczas:

$$c_{i,j} = \begin{cases} 0 & jeśli\ i = 0\ lub\ j = 0 \\ 1 + c_{i-1,j-1} & jeśli\ i, j > 0, x_i = y_j \\ \max(c_{i,j-1}, c_{i-1,j}) & jeśli\ i, j > 0, x_i \neq y_j \end{cases}$$

Sposób obliczeń

Wniosek podaje prosty sposób na obliczenie długości najdłuższego wspólnego podciągu: należy obliczyć wszystkie wartości $c_{i,j}$, które dla prostoty będą pamiętane w tablicy, a następnie odczytać wynik w $c_{m,n}$.

Ponadto warto zauważyć, że:

- $c_{i,0} = 0$ dla każdego naturalnego i
- $c_{0,j} = 0$ dla każdego naturalnego j

Przykład

	ε	**R**	**A**	**B**	**A**	**R**	**B**	**A**	**R**
ε	0	0	0	0	0	0	0	0	0
L	0								
A	0								
B	0								
R	0								
A	0								
D	0								
O	0								
R	0								

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0								
B	0								
R	0								
A	0								
D	0								
O	0								
R	0								

Znak L nie zgadza się z żadnym znakiem słowa RABARBAR...

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0								
R	0								
A	0								
D	0								
O	0								
R	0								

Przy zgodności liter zwiększamy długość LCS o 1 (idziemy po przekątnej), w przeciwnym przypadku wybieramy maksymalną wartość z komórek (i-1, j) oraz (i, j-1).

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0								
A	0								
D	0								
O	0								
R	0								

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3
A	0								
D	0								
O	0								
R	0								

Czasami nie ma znaczenia, czy przepisujemy długość ciągu z góry, czy z lewej strony (brak strzałek do elementu tablicy).
Zależy to od zapisania funkcji MAX z dwóch wartości.

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3
A	0	1	2	2	3	3	3	4	4
D	0								
O	0								
R	0								

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3
A	0	1	2	2	3	3	3	4	4
D	0	1	2	2	3	3	3	4	4
O	0								
R	0								

Litera D nie występuje w ciągu RABARBAR – nic się nie zmienia w stosunku do poprzedniego wypełnienia tablicy.

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3
A	0	1	2	2	3	3	3	4	4
D	0	1	2	2	3	3	3	4	4
O	0	1	2	2	3	3	3	4	4
R	0								

Litera O nie występuje w ciągu RABARBAR – nic się nie zmienia w stosunku do poprzedniego wypełnienia tablicy.

	ε	R	A	B	A	R	B	A	R
ε	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	2	2	2	2	2	2
R	0	1	1	2	2	3	3	3	3
A	0	1	2	2	3	3	3	4	4
D	0	1	2	2	3	3	3	4	4
O	0	1	2	2	3	3	3	4	4
R	0	1	2	2	3	4	4	4	5

Długość najdłuższego wspólnego podciągu jest równa 5.

Algorytm

Dane: X, Y – ciągi

Wynik: Z = LCS(X,Y)

```
m = dlugosc(X)
n = dlugosc(Y)
for i  = 0, 1, ..., m
      c[i][0] = 0
for j  = 0, 1, ..., n
      c[0][j] = 0
for i = 1, 2, ..., m
      for j = 1, 2, ..., n
      if (x[i] == y[j])
            c[i][j] = 1+c[i-1][j-1]
      else
            if (c[i-1][j] ≥ c[i][j-1])
                  c[i][j] = c[i-1][j]
            else
                  c[i][j] = c[i][j-1]
```

Jak znaleźć właściwy LCS?

- Można zapamiętać „trasę”, wykorzystując dodatkową tablicę na pamiętanie drogi, którą podążamy.
- Można odzyskać kolejne elementy LCS wykorzystując algorytm wypełniania tablicy c.

Algorytm pozwalający wyznaczyć LCS

```
i = m
j = n
slowo = ”” //slowo puste
while (i> 0 and j >0)
      if (x[i] == y[j])
            slowo = x[i]+slowo
            i = i-1
            j = j-1
      else
            if (c[i-1][j] ≥ c[i][j-1])
                  i = i-1
            else
                  j = j-1
```

Koszt algorytmu

- Obliczenie każdego elementu tablicy c odbywa się w czasie stałym. Całkowity koszt wypełnienia tablicy c jest równy $n \cdot m$.
- Koszt skonstruowania najdłuższego podciągu na podstawie dodatkowej tablicy jest linowy.

Możliwe usprawnienia

- Można zrezygnować z dodatkowej tablicy do pamiętania kolejnych elementów LCS, ponieważ każde pole c[i][j] zależy wyłącznie od c[i-1][j-1], c[i][j-1] lub c[i-1][j] (co już zrobiliśmy).

- Jeśli zależy na jedynie na długości LCS a nie na jego konstrukcji, to można zredukować koszt pamięciowy procedury – wystarczy użyć dwóch tablic n-elementowych, ponieważ podczas obliczeń w każdej iteracji korzystamy tylko z dwóch wierszy tablicy c: aktualnie obliczanego i poprzedniego (w rzeczywistości można ograniczyć się do jednej tablicy...)

Odległość edycyjna - wprowadzenie

Automatyczne sprawdzanie pisowni:

Podczas napotkania błędu narzędzie zagląda do swojego słownika i stara się znaleźć inne słowa o zbliżonej pisowni.

Co to znaczy BLISKOŚĆ słów? Naturalną miarą odległości między dwoma słowami jest stopień, w jakim mogą zostać przyrównane lub dopasowane.

TECHNICZNIE: przyrównanie to sposób zapisania tych słów jednego nad drugim.

Przykład

SNOWY

SUNNY

S – N O W Y	– S N O W – Y
S U N N – Y	S U N – – N Y
koszt: 3	koszt: 5

Uwaga: Kreska oznacza „dziurę”, można ich wstawić dowolnie dużo do obu słów.

Koszt przyrównania obliczamy jako liczbę kolumn, w których występują różne litery.

Odległość edycyjna

- Odległością edycyjną między dwoma słowami nazywamy koszt ich najlepszego możliwego przyrównania.
- O odległości edycyjnej można myśleć jak o minimalnej liczbie operacji edycyjnych – wstawiania, usuwania, zmiany znaków – potrzebnych do przekształcenia pierwszego słowa w drugie.

Problem:

Dla danych dwóch słów **A** i **B** wyznaczyć ich odległość edycyjną, tzn. ile co najmniej znaków należy usunąć lub wstawić w jednym słowie, aby uzyskać drugie.

Przykłady

- Dla podanego przykładu nie istnieje żadne lepsze dopasowanie słów SNOWY i SUNNY niż to o koszcie równym 3:

  S – N O W Y

  S U N N – Y

- Przykład odległości edycyjnej równej 0:

  P I E S

  P I E S

- Przykład odległości edycyjnej równej 1:

  G R A N A T

  G R A N I T

Przykłady

DESKOROLKA

STOKROTKA

Jaka jest odległość edycyjna?

Jest równa 5...

~~DE~~S(K→T)OKRO(L→T)KA

Rozwiązanie

- W ogólności istnieje tak wiele możliwych przyrównań dwóch słów, że przeglądanie wszystkich możliwości w poszukiwaniu najlepszego rozwiązania jest nieefektywne.
- Można wykorzystać programowanie dynamiczne!!!

Jakie są podproblemy?

- Celem jest znalezienie odległości między dwoma słowami $X[1, \dots, m]$ oraz $Y[1, \dots, n]$.
- Podproblemem będzie znalezienie odległości edycyjnej między dwoma słowami $X[1, \dots, i]$ oraz $Y[1, \dots, j]$ dla $i = 1, 2, \dots, m$ oraz $j = 1, 2, \dots, n$.

Najlepsze przyrównanie dwóch słów

Niech $E_{i,j}$ oznacza najlepsze przyrównanie dwóch słów $X[1, \dots, i]$ oraz $Y[1, \dots, j]$:

Skrajnie prawa kolumna może przyjąć jedną z postaci:

$$\begin{matrix} x[i] \\ \_\_ \end{matrix} \quad \text{lub} \quad \begin{matrix} \_\_ \\ y[j] \end{matrix} \quad \text{lub} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

Przypadki

- W pierwszym przypadku wnosimy 1 do odległości edycyjnej (jedna zmiana) + koszt porównania słów $X[1, \dots, i-1]$ oraz $Y[1, \dots, j]$ (podproblem $E_{i-1,j}$)
- W drugim przypadku wnosimy 1 do odległości edycyjnej (jedna zmiana) + koszt porównania słów $X[1, \dots, i]$ oraz $Y[1, \dots, j-1]$ (podproblem $E_{i,j-1}$)
- W trzecim przypadku koszt jest równy:
  - 1, jeśli $x[i] \neq y[j]$
  - 0, jeśli $x[i = y[j]$

+ koszt porównania słów $X[1, \dots, i-1]$ oraz $Y[1, \dots, j-1]$ (podproblem $E_{i-1,j-1}$)

Podsumowanie

Ponieważ nie wiemy, który z mniejszych podproblemów jest właściwym, musimy sprawdzić wszystkie możliwości i wybrać najlepszą z nich (czyli minimum!):

$$E_{i,j} = \min\{1 + E_{i-1,j}, 1 + E_{i,j-1}, dist + E_{i-1,j-1}\},$$

gdzie dla wygody oznaczamy

$$dist = \begin{cases} 0, & gdy\ x[i] = y[j] \\ 1, & gdy\ x[i] \neq y[j] \end{cases}.$$

Przykład

Chcemy znaleźć minimalną odległość edycyjną między dwoma słowami:

X = FOKA

Y = KOTKA

	ε	K	O	T	K	A
ε	0	1	2	3	4	5
F	1	1	2	3	4	5
O	2					
K	3					
A	4					

+1? +1? +1?

Żadna litera wyrazu KOTKA nie pokrywa się z literą F, zatem z każdą kolejną literą słowa KOTKA minimalna odległość edycyjna się zwiększa, wybieramy:

$$\min\{1 + E_{i-1,j}, 1 + E_{i,j-1}, 1 + E_{i-1,j-1}\}$$

	ε	K	O	T	K	A
ε	0	1	2	3	4	5
F	1	1	2	3	4	5
O	2	2	1	2	3	4
K	3					
A	4					

+1? +1? +1?

Szare pole pokazuje zgodność liter, wtedy min. odległość edycyjna nie zmienia się w stosunku do $E_{i-1,j-1}$. W pozostałych sytuacjach wybieramy:

$$\min\{1 + E_{i-1,j}, 1 + E_{i,j-1}, 1 + E_{i-1,j-1}\}$$

	ε	K	O	T	K	A
ε	0	1	2	3	4	5
F	1	1	2	3	4	5
O	2	2 +1?	1 +1?	2	3	4
K	3	2 +1?	2	2	2	3
A	4					

	ε	K	O	T	K	A
ε	0	1	2	3	4	5
F	1	1	2	3	4	5
O	2	2	1	2	3	4
K	3 +1?	2 +1?	2 +1?	2 +1?	2	3
A	4 +1?	3	3 +1?	3	3	**2**

Minimalna odległość edycyjna jest równa 2.

FOKA -> KOTKA:

1) zamieniając F na K,
2) dodając literę T,

KOTKA -> FOKA:

1) zamieniając K na F,
2) usuwając literę T.

Algorytm

Dane:

X, Y – ciągi znaków o długości odpowiednio m, n

Wynik:

minimalna odległość edycyjna między X i Y (pamiętana w tablicy E[m][n])

```
for i = 0, 1 , 2, ...., m
   E[i][0] = i
for j = 0, 1, 2, ..., n
   E[0][j] = j

for i = 1 , 2, ...., m
   for j = 1 , 2, ...., n
      if (X[i] == Y[j])
         E[i][j] = E[i-1][j-1]
      else
         E[i][j] = min (E[i-1][j], E[i][j-1], E[i-1][j-1])+1
```

UWAGA: Wynik przechowywany jest w E[m][n].

Dziękuję za uwagę!