

# Uzupełnienia

## programowanie dynamiczne

### bardziej zaawansowane przykłady

„Dziel i zwyciężaj”

Dzielimy problem na mniejsze pod-problemy  
rekurencyjnie wywołujemy procedurę dla pod-problemów  
bardzo małe pod-problemy rozwiązujemy łatwo

Przykłady: MergeSort, QuickSort

„Programowanie dynamiczne”

Jak „dziel i zwyciężaj” ale unikamy  
wywołań rekurencyjnych z tymi samymi parametrami;  
zapamiętujemy wyniki tych wywołań (spamiętywanie, memoization ang.)  
gdzie zapamiętujemy? Odp: tablica, słownik

Przykład: element ciągu Fibonacciego

Mamy rekurencyjną procedurę  $Fib(n)$ ...  
liczba wywołań rekurencyjnych  $\approx 2^n$   
liczba różnych parametrów przy obliczaniu  $Fib(n) \approx n$

**Wniosek:** średnio  $(2^n)/n$  wywołań z tym samym parametrem  
nadaje się do programowania dynamicznego/ spamiętywania !!!

Przykład bardziej zaawansowany:

**LCS** = „Longest Common Subsequence”

**NWP** = „Największy Wspólny Podciąg”

dla 2 ciągów:  $X = x_1, \dots, x_m, Y = y_1, \dots, y_n$

$NWP(X, Y) = Z, Z = z_1, \dots, z_k; Z$  to jeden z NWP (może ich być kilka!!)

# programowanie dynamiczne

## NWP = „Największy Wspólny Podciąg”



Przykłady:

$\text{NWP}(\{A B C D\}, \{A C D E\}) = A C D$

$\text{NWP}(\{A B C D E F\}, \{B X D Y E H I J\}) = B D E$

NWP służy m.in. do określenia różnicy między 2 plikami tekstowymi:

„Ile operacji trzeba wykonać aby plik nr 1 stał się plikiem nr 2”

NWP jest używane w programie unixowym **diff** służącym do porównywania 2 plików tekstowych (elementy ciągów = linie tekstu)

```
cat -n tekst03a.txt
```

```
1 qqq qqq qqq
2 wwwww wwwww wwwww
3 eee eee
4 rrr rrr rrr
```

```
cat -n tekst03b.txt
```

```
1 qqq qqq qqq
2 www wwwww wwwww
3 eee eee
4 ddd ddd ddd
5 rrr rrr rrr
```

```
diff tekst03a.txt tekst03b.txt
```

```
2c2
< wwwww wwwww wwwww
```

```
---
```

```
> www wwwww wwwww
```

```
3a4
> ddd ddd ddd
```

*linia1 znak linia2*

znak = „c” change, „a” append, „d” delete, ...

# programowanie dynamiczne

## NWP = „Największy Wspólny Podciąg”



$X = x_1, \dots, x_m, Y = y_1, \dots, y_n, \text{NWP}(X, Y) = Z, Z = z_1, \dots, z_k$   
Dodatkowe oznaczenia:  $X_i = x_1, \dots, x_i$  ( $X_i$  to prefiks  $X$ )

### *Theorem 15.1 (Optimal substructure of an LCS)*

Let  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  be sequences, and let  $Z = \langle z_1, z_2, \dots, z_k \rangle$  be any LCS of  $X$  and  $Y$ .

1. If  $x_m = y_n$ , then  $z_k = x_m = y_n$  and  $Z_{k-1}$  is an LCS of  $X_{m-1}$  and  $Y_{n-1}$ .
2. If  $x_m \neq y_n$ , then  $z_k \neq x_m$  implies that  $Z$  is an LCS of  $X_{m-1}$  and  $Y$ .
3. If  $x_m \neq y_n$ , then  $z_k \neq y_n$  implies that  $Z$  is an LCS of  $X$  and  $Y_{n-1}$ .

Problem  $\text{NWP}(X, Y)$  sprowadzamy do „mniejszych” problemów  
 $\text{NWP}(X_{m-1}, Y_{n-1}), \text{NWP}(X_{m-1}, Y), \text{NWP}(X, Y_{n-1})$

Można zdef tablicę  $c[i, j] :=$  długość  $\text{NWP}(X_i, Y_j)$ ,  
na podstawie Tw ^ spełniającą zależność rekurencyjną:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

# programowanie dynamiczne

## NWP = „Największy Wspólny Podciąg”

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Można obliczyć  $c[i, j]$  rekurencyjnie, z użyciem „spamiętywania”, (góra → dół)  
 Można też bez rekurencji (dół → góra):

LCS-LENGTH( $X, Y$ )

```

1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i - 1, j - 1] + 1
12             b[i, j] = "↖"
13         elseif c[i - 1, j] >= c[i, j - 1]
14             c[i, j] = c[i - 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j - 1]
17             b[i, j] = "←"
18  return c and b
    
```

		j	0	1	2	3	4	5	6
		i	y <sub>j</sub>	B	D	C	A	B	A
0	x <sub>i</sub>		0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	←	↖
2	B		0	↖	←	←	↑	↖	←
3	C		0	↑	↑	↖	←	↑	↑
4	B		0	↖	↑	↑	↑	↖	←
5	D		0	↑	↖	↑	↑	↑	↑
6	A		0	↑	↑	↑	↖	↑	↖
7	B		0	↖	↑	↑	↑	↖	↑

Przy okazji obl  $c[i, j]$  obl także  $b[i, j]$ , które pozwala odtworzyć NWP !!!

# programowanie dynamiczne

## NWP = „Największy Wspólny Podciąg”

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

		<i>j</i>	0	1	2	3	4	5	6
			$y_j$	B	D	C	A	B	A
<i>i</i>	$x_i$	0	0	0	0	0	0	0	0
0		0	0	0	0	0	0	0	0
1	A	0	↑	↑	↑	↖	←	↖	↖
2	B	0	↖	←	←	↑	↖	←	←
3	C	0	↑	↑	↖	←	↑	↑	↑
4	B	0	↖	↑	↑	↑	↑	↖	←
5	D	0	↑	↖	↑	↑	↑	↑	↑
6	A	0	↑	↑	↑	↖	↑	↑	↖
7	B	0	↖	↑	↑	↑	↑	↖	↑

Jak odczytać NWP z  $c[,]$  oraz  $b[,]$  ??  
 zaczynamy od komórki  $c[m,n]$ ,  
 czyli prawy dolny róg;  
 idziemy w kierunku strzałki,  
 strzałki ukośne  
 to kolejne elem NWP(X,Y)

# programowanie dynamiczne

## „nawiasowanie” przy mnożeniu ciągu macierzy

$A_i$  – macierz, chcemy obliczyć  $A_1 * A_2 * \dots * A_n$ , min liczbę mnożeń liczb

Mnożenie 2 macierzy a liczba mnożeń:

$A_1$  - macierz  $n \times m$ ,  $A_2$  - macierz  $m \times k$ ,  $A_1 * A_2$  - macierz  $n \times k$   
liczba mnożeń =  $n * m * k$

Okazuje się, że liczba „\*” zależy od nawiasów:

$A_1$  –  $10 \times 15$ ,  $A_2$  –  $15 \times 20$ ,  $A_3$  –  $20 \times 1$

$(A_1 * A_2) * A_3$  – 3200 mnożeń

$A_1 * (A_2 * A_3)$  – 450 mnożeń

Oznaczenia:  $A_{i..j} := A_i * A_{i+1} * \dots * A_{j-1} * A_j$ ;  $A_i$  to macierz  $p_{i-1} \times p_i$

Jeśli mamy optymalne nawiasowanie  $A_{1..n}$  oraz

dla pewnego „k” mamy „\*” najwyższego poziomu między  $A_k$  i  $A_{k+1}$

to to nawiasowanie jest optymalne w  $A_{1..k}$  oraz w  $A_{k+1..n}$

Def koszt opt nawiasowania  $m[i,j]$  = min liczba mnożeń dla  $A_{i..j}$

Można podać następujący wzór rekurencyjny na  $m[i,j]$ :

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

# programowanie dynamiczne „nawiasowanie” przy mnożeniu ciągu macierzy

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases}$$

Jak to ^ obliczyć ?

Odp: rekurencja + spamiętywanie (metoda góra → dół)

Jak zamienić  $m[,]$  na konkretne nawiasowanie ?

**Def  $s[i, j] = k$** , dla którego mamy minimum we wzorze na  $m[i, j]$   
można obl  $s[i, j]$  razem z  $m[i, j]$  w proc rekurencyjnej...

$s[1, n]$  wyznacza pozycję „\*” najwyższego poziomu w  $A_{1..n}$ ,

$A_{1..n} = A_{1..s[1, n]} * A_{s[1, n]+1..n}$  (lewy składnik \* prawy składnik)

dla lewego składnika:  $s[1, s[1, n]]$  wyznacza pozycję „\*” najw poziomu  
w lewym składniku...

podobnie  $s[s[1, n]+1, n]$  wyznacza pozycję „\*” najw poziomu  
w prawym składniku...

w ten sposób wyznaczymy nawiasowanie całego  $A_{1..n}$

# programowanie dynamiczne „nawiasowanie” przy mnożeniu ciągu macierzy

MEMOIZED-MATRIX-CHAIN( $p$ )

```
1  $n = p.length - 1$   
2 let  $m[1..n, 1..n]$  be a new table  
3 for  $i = 1$  to  $n$   
4   for  $j = i$  to  $n$   
5      $m[i, j] = \infty$   
6 return LOOKUP-CHAIN( $m, p, 1, n$ )
```

LOOKUP-CHAIN( $m, p, i, j$ )

```
1 if  $m[i, j] < \infty$   
2   return  $m[i, j]$   
3 if  $i == j$   
4    $m[i, j] = 0$   
5 else for  $k = i$  to  $j - 1$   
6    $q = \text{LOOKUP-CHAIN}(m, p, i, k)$   
        $+ \text{LOOKUP-CHAIN}(m, p, k + 1, j) + p_{i-1}p_kp_j$   
7   if  $q < m[i, j]$   
8      $m[i, j] = q$   
9 return  $m[i, j]$ 
```

Rekurencyjna proc obl  $m[,]$   
stosująca spamiętywanie  
łatwo dodać obl także  $s[,]$





