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 ≈ 2^n
liczba różnych parametrów przy obliczaniu Fib(n) ≈ 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, …, x_m, Y = y_1, …, y_n
NWP(X,Y) = Z, Z=z_1, …, z_k; Z to jeden z NWP (może ich być kilka!!)

programowanie dynamiczne
NWP = „Największy Wspólny Podciąg”

Przykłady:
NWP ({A B C D}, {A C D E}) = A C D
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  wwww wwww wwww
     3  eee eee
     4  rrr rrr rrr

cat -n tekst03b.txt
     1  qqq qqq qqq
     2  www wwww wwww
     3  eee eee
     4  ddd ddd ddd
     5  rrr rrr rrr
```

```
diff tekst03a.txt tekst03b.txt
2c2
< wwww wwww wwww
---
> www wwww wwww
3a4
> ddd ddd ddd
```

linia1 znak linia2
znak = „c” change, „a” append, „d” delete, ...

programowanie dynamiczne
NWP = „Największy Wspólny Podciąg”

X = x_1, …, x_m, Y = y_1, …, y_n, NWP(X,Y) = Z, Z=z_1, …, z_k
Dodatkowe oznaczenia: X_i = x_1, …, x_i (X_i to prefiks X)

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 NWP(X,Y) sprowadzamy do „mniejszych” problemów
NWP(X_m-1, Y_n-1), NWP(X_m-1, Y), NWP(X, Y_n-1)

Można zdef tablicę c[i,j] := długość 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_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

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

programowanie dynamiczne
NWP = „Największy Wspólny Podciąg”

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

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

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 * … A_n, minimalizujemy liczbę „*”

Mnożenie 2 macierzy a liczba mnożeń:
A_1 - macierz n x m, A_2 - macierz m x k, A_1*A_2 - macierz n x k
liczba mnożeń = n*m*k

Okazuje się, że liczba „*” zależy od nawiasów:
A_1 – 10 x 15, A_2 – 15 x 20, A_3 – 20 x 1
(A_1 * A_2) * A_3 – 3200 mnożeń
A_1 * (A_2 * A_3) – 450 mnożeń

Oznaczenie: A_i..j := A_i * A_i+1 * … * A_j-1 * A_j

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]:

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

programowanie dynamiczne
„nawiasowanie” przy mnożeniu ciągu macierzy

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

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 skladnik)

dla lewego składnika: s[1,s[1,n]] wyznacza pozycje „*” najw poziomu
w lewym składniku...
podobnie s[s[1,n]+1,n] wyznacza pozycje „*” 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] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2       return m[i, j]
3  if i == j
4       m[i, j] = 0
5  else for k = i to j − 1
6            q = LOOKUP-CHAIN(m, p, i, k)
                 + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_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[,]