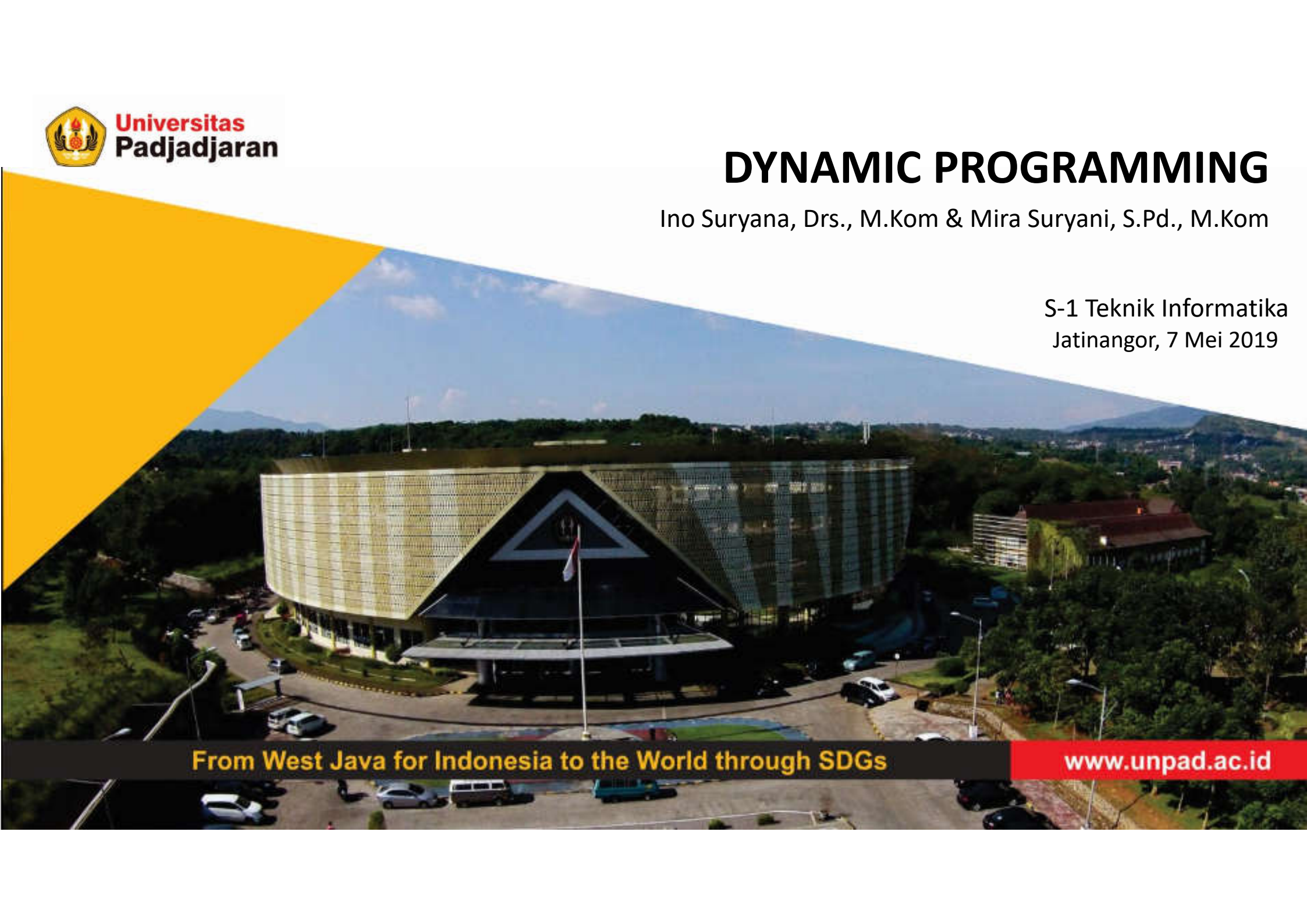


DYNAMIC PROGRAMMING

Ino Suryana, Drs., M.Kom & Mira Suryani, S.Pd., M.Kom

S-1 Teknik Informatika
Jatinangor, 7 Mei 2019



From West Java for Indonesia to the World through SDGs

www.unpad.ac.id



Tujuan Pembelajaran

Setelah mempelajari dynamic programming diharapkan mahasiswa dapat memahami dan mengimplementasikan dynamic programming untuk menyelesaikan problem yang berkaitan dengan topik



Pembahasan Materi

- Pengenalan dynamic programming
- Kasus 1: Matrix-Chain Multiplication
- Memoization
- Kasus 2: Longest Common Subsequence



MATRIX-CHAIN MULTIPLICATION



PENGENALAN DYNAMIC PROGRAMMING

- Seperti metode divide and conquer, dynamic programming memecahkan problem dengan menggabungkan solusi untuk subproblem.
- "Pemrograman" di sini mengacu pada **perencanaan menggunakan metode tabular**, bukan untuk menulis program komputer.
- Seperti yang kita lihat dalam kuliah sebelumnya, algoritma yang didasarkan pada metode divide and conquer membagi problem menjadi subproblem independen, memecahkan subproblem secara rekursif, dan kemudian menggabungkan solusi untuk memecahkan masalah asli.

Sebaliknya, pemrograman dinamis berlaku ketika subproblem tidak independen, yaitu ketika subproblem berbagi subsubproblem.

- Ketika subproblem tidak independen, algoritma divide & conquer bekerja lebih dari yang diperlukan, yaitu berulang kali memecahkan subsubproblem umum.
- Algoritma dynamic programming memecahkan setiap sub-problem **hanya sekali** dan kemudian menyimpan jawabannya dalam sebuah **tabel**, untuk penggunaan selanjutnya.



PENGENALAN DYNAMIC PROGRAMMING

- Dynamic programming biasanya diterapkan untuk masalah **optimisasi**. Dalam problem seperti itu, ada banyak kemungkinan solusi. Setiap solusi memiliki nilai, dan kita ingin mencari solusi dengan optimal nilai (minimum atau maksimum).
- Kita menyebut solusi semacam itu sebagai **sebuah solusi optimal** untuk problem tersebut, seperti berlawanan dengan solusi optimal umumnya, karena mungkin ada beberapa solusi yang mencapai nilai optimal.



TAHAPAN DYNAMIC PROGRAMMING

1. Cirikan struktur solusi optimal.
2. Secara rekursif menentukan nilai solusi optimal.
3. Hitung nilai solusi optimal dengan cara bottom-up.
4. Bangun solusi optimal dari informasi yang dikomputasi.

Contoh pertama pemrograman dinamis kita adalah algoritma untuk memecahkan masalah **matrix-chain multiplication**.



Matrix-chain Multiplication Problem

Problem:

Diberikan sekuen (sebuah rangkaian) (A_1, A_2, \dots, A_n) dari n buah matriks yang akan dikalikan. Dimana untuk $i = 1, 2, \dots, n$ matriks A_i memiliki dimensi $P_{i-1} \times P_i$, Proses fully parenthesize produk A_1, A_2, \dots, A_n merupakan cara untuk meminimalisir jumlah perkalian skalar yang mungkin terjadi

- Sebuah produk matriks disebut fully parenthesized jika ia terdiri dari single matriks atau hasil produk dari dua matriks yang dikelilingi kurung
- Matrix multiplication bersifat associative, dan semua kurung menghasilkan produk yang sama. Tapi perbedaan peletakan kurung menghasilkan cost yang berbeda dalam hal perkalian skalarnya. **Lihat Contoh.**



Matrix-chain Multiplication Problem

- Algoritma standar untuk mengalikan dua matriks diberikan oleh pseudocode berikut.

MATRIX-MULTIPLY(A, B)

1. if $\text{ncolumns}[A] \neq \text{nrows}[B]$
2. then error "incompatible dimensions"
3. else for $i \leftarrow 1$ to $\text{nrows}[A]$
4. do for $j \leftarrow 1$ to $\text{ncolumns}[B]$
5. do $C[i, j] \leftarrow 0$
6. for $k \leftarrow 1$ to $\text{ncolumns}[A]$
7. do $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
8. return C

//ncolumns = number of columns

//nrows = number of rows



Matrix-chain Multiplication Problem

- Kita dapat mengkalikan dua matriks A dan B hanya jika mereka kompatibel satu sama lain: jumlah kolom di A equal dengan jumlah baris di B
- Jika A adalah matriks ($p \times q$) dan B adalah matriks ($q \times r$), maka hasil matriks C-nya adalah ($p \times r$)
- Waktu untuk mengkomputasi C didominasi oleh jumlah perkalian scalar di baris ke-7, dimana totalnya adalah $p \cdot q \cdot r$



Matrix-chain Multiplication Problem

Contoh:

Perhatikan permasalahan sebuah rangkaian berikut (A_1, A_2, A_3) terdiri dari 3 matriks. Seandainya dimensi dari matriks tersebut adalah 10×100 , 100×5 , dan 5×50 berturut-turut.

Produk A_1, A_2, A_3 dapat dilakukan fully parenthesized dengan 2 cara berbeda: $((A_1 A_2) A_3)$ dan $(A_1 (A_2 A_3))$

Jika kita mengkalikan sesuai dengan aturan kurung $((A_1 A_2) A_3)$, kita melakukan $5000 + 2500 = 7500$ perkalian skalar

Jika kita mengkalikan sesuai dengan aturan kurung $(A_1 (A_2 A_3))$, kita melakukan $25000 + 50000 = 75000$ perkalian skalar

Dengan demikian, menghitung produk sesuai dengan tanda kurung pertama adalah 10 kali lebih cepat.



Matrix-chain Multiplication Problem

Contoh:

Dalam berapa banyak cara yang berbeda, kita dapat mengurung produk A_1, A_2, A_3, A_4 ?

Dalam 5 cara berbeda:

$$(A_1 (A_2 (A_3 A_4)))$$

$$(A_1 ((A_2 A_3) A_4))$$

$$((A_1 A_2)(A_3 A_4))$$

$$((A_1 (A_2 A_3)) A_4)$$

$$(((A_1 A_2) A_3) A_4)$$

Bagaimana dengan A_1, A_2, A_3, A_4, A_5 ?



Menghitung Jumlah Kurung

- Misalkan $P(n)$ menunjukkan jumlah tanda kurung alternatif dari sebuah urutan n matriks.
- Ketika $n = 1$, hanya ada satu matriks dan oleh karena itu hanya ada salah satu cara untuk sepenuhnya mengurung produk matriks.
- Ketika $n \geq 2$, produk matriks yang dikurungkan sepenuhnya adalah produk dari dua subproduk matriks sepenuhnya dikurungkan, dan perpecahan antara dua subproduk dapat terjadi antara (k) ke dan $(k + 1)$ matriks untuk setiap $k = 1, 2, \dots, n-1$.
- Jadi, untuk jumlah total tanda kurung yang berbeda, kita memiliki rekurensi:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$



Menghitung Jumlah Kurung

Dengan metode menghasilkan fungsi, dapat ditunjukkan bahwa solusinya adalah $P(n) = C(n-1)$, dimana

$$C(n) = \frac{1}{n+1} \binom{2n}{n} \\ = \Omega(4^n / n^{3/2})$$

$C(n)$ is the n th Catalan number.

Jadi jumlah total kurung adalah eksponensial dalam n , dan metode brute-force memeriksa semua kemungkinan tanda kurung adalah strategi yang buruk untuk menentukan tanda kurung optimal dari rantai matriks.

n	$C(n)$
1	1
2	2
3	5
4	14
5	42
6	132



Catalan Number menghitung banyak hal selain jumlah tanda kurung; sebagai contoh:

- $C(n)$ adalah jumlah cara $2n$ orang duduk melingkar meja bisa berjabat tangan dalam pasangan tanpa menyilangkan tangan.
- $C(n)$ adalah jumlah pohon biner yang berbeda dengan n node.



Solving the matrix-chain multiplication problem by dynamic programming

Step 1:

Mencirikan struktur tanda kurung yang optimal.

- Langkah pertama dalam paradigma pemrograman dinamis adalah menemukan **substruktur optimal** dan kemudian menggunakannya untuk membangun solusi optimal untuk problem dari solusi optimal untuk subproblem.
- Untuk masalah matrix-chain multiplication, kita dapat melakukan langkah ini sebagai berikut.
- Notation: $A_{i..j}$ menunjukkan matriks yang dihasilkan dari evaluasi produk $A_i A_{i+1} \dots A_j$, dimana $i \leq j$.
- Jika problemnya adalah nontrivial (yaitu $i < j$), maka setiap tanda kurung produk $A_i A_{i+1} \dots A_j$ harus membagi produk antara A_k dan A_{k+1} untuk beberapa k dalam kisaran $i \leq k < j$. Biaya kurung ini adalah (biaya komputasi matriks $A_{i..k}$) + (biaya komputasi $A_{k+1..j}$) + (biaya mengalikannya bersama-sama).



- Substruktur optimal:

Misalkan tanda kurung optimal $A_i A_{i+1} \dots A_j$ membagi produk antara A_k dan A_{k+1} . Kemudian tanda kurung dari subchain $A_i A_{i+1} \dots A_k$ dalam kurung optimal $A_i A_{i+1} \dots A_j$ harus menjadi tanda kurung yang optimal dari $A_i A_{i+1} \dots A_k$.

Kenapa? Jika ada cara yang lebih murah untuk mengurung $A_i A_{i+1} \dots A_k$, maka kita bisa mengganti tanda kurung itu dalam tanda kurung yang optimal dari $A_i A_{i+1} \dots A_j$ untuk menghasilkan cara lain untuk mengurungkan $A_i A_{i+1} \dots A_j$ yang biayanya lebih rendah dari yang optimal: sebuah kontradiksi.

Demikian pula kurung dari subchain $A_{k+1} A_{k+2} \dots A_j$ dalam kurung optimal $A_i A_{i+1} \dots A_j$ harus menjadi tanda kurung optimal $A_{k+1} A_{k+2} \dots A_j$.



- Sekarang kita menggunakan substruktur optimal untuk menunjukkan bahwa kita dapat membangun solusi optimal untuk problem dari solusi optimal untuk subproblem.
- Setiap solusi untuk turunan nontrivial dari masalah matrix-chain multiplication perlu memecah produk, dan setiap solusi optimal mengandung solusi optimal untuk instance subproblemnya.
- Dengan demikian, solusi optimal untuk masalah tersebut dapat dibangun dengan:
 1. Memecah masalah menjadi dua submasalah (mis., secara optimal memberi tanda kurung $A_i A_{i+1} \dots A_k$ dan $A_i A_{i+1} \dots A_k$);
 2. Menemukan solusi optimal untuk instance subproblem;
 3. Menggabungkan solusi subproblem optimal ini.

Ketika kita mencari tempat yang tepat untuk membagi produk, kita harus mempertimbangkan semua tempat yang memungkinkan sehingga kita yakin telah menemukan yang optimal.



Step 2:

Secara rekursif menentukan nilai solusi optimal.

- Kami mendefinisikan biaya solusi optimal secara rekursif dalam hal solusi optimal untuk subproblem.
- Kita memilih sebagai submasalah kami: masalah penentuan biaya minimum tanda kurung $A_i A_{i+1} \dots A_j$ untuk $1 \leq i \leq j \leq n$
- Misalkan $m[i, j]$ menjadi jumlah minimum perkalian skalar diperlukan untuk menghitung matriks $A_{i..j}$.
- Untuk problem penuh, biaya cara termurah untuk menghitung $A_{1..n}$ akan menjadi $m[1, n]$.
- Definisi rekursif untuk biaya minimum untuk mengurung produk $A_i A_{i+1} \dots A_j$ adalah:

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



- Untuk melacak cara membangun solusi optimal, mari kita definisikan $s[i, j]$ menjadi nilai k di mana kita dapat membagi produk $A_i A_{i+1} \dots A_j$ untuk mendapatkan kurung yang optimal.
- Dengan kata lain, $s[i, j]$ sama dengan nilai k sedemikian sehingga $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Step 3:

Hitung nilai solusi optimal dengan cara bottom-up.

- Ini adalah problem sederhana untuk menulis algoritma rekursif berdasarkan pengulangan pada slide sebelumnya. Namun, algoritma rekursif ini akan membutuhkan waktu yang eksponensial, yang tidak lebih baik daripada metode brute-force untuk memeriksa setiap cara untuk mengurung produk.
- Satu pengamatan penting adalah hanya ada total $\binom{n}{2} + n = \Theta(n^2)$ subproblem berbeda. Satu masalah untuk setiap pilihan i dan j memuaskan $1 \leq i \leq j \leq n$



- Algoritma rekursif dapat menemukan banyak subproblem beberapa kali di cabang yang berbeda dari pohon rekursi. Properti **overlapping subproblem** ini adalah ciri kedua dari penerapan dynamic programming (ciri pertama adalah **substruktur optimal**).
- Alih-alih menghitung solusi untuk pengulangan di Step 3 secara rekursif, paradigma pemrograman dinamis menghitung biaya optimal dengan menggunakan tabel dalam pendekatan **bottom-up**.
- Pseudocode berikut mengasumsikan bahwa matriks A_i memiliki dimensi $p_{i-1}p_i$ untuk $i = 1, 2, \dots, n$.
- Input adalah sebuah urutan $p = (p_0, p_1, \dots, p_n)$
- Prosedur ini menggunakan tabel tambahan **m** **[1..n, 1..n]** untuk menyimpan biaya $m[i, j]$ dan tabel tambahan **s** **[1..n, 1..n]** yang mencatat indeks **k** mana yang mencapai biaya optimal dalam komputasi $m[i, j]$. Tabel **s** dapat digunakan untuk membangun solusi optimal.



MATRIX-CHAIN-ORDER(p)

1. $n \leftarrow \text{length}[p] - 1$
2. for $i \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. for $L \leftarrow 2$ to n // L is the matrix-chain length
5. do for $i \leftarrow 1$ to $n - L + 1$
6. do $j \leftarrow i + L - 1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$
9. do $q \leftarrow m[i, k] + m[k + 1, j] + p[i - 1] \cdot p[k] \cdot p[j]$
10. if $q < m[i, j]$
11. then $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. return m and s



Contoh Soal:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25



Tabel m[1..6, 1..6]

		j					
		1	2	3	4	5	6
i	1	0	15750	7875	9375	11875	15125
	2		0	2625	4375	7125	10500
	3			0	750	2500	5375
	4				0	1000	3500
	5					0	5000
	6						0



Tabel $s[1..6, 1..6]$

		j					
		1	2	3	4	5	6
i	1		1	1	3	3	3
	2			2	3	3	3
	3				3	3	3
	4					4	5
	5						5
	6						



$$m[3,4] = \min\{m[3,3] + m[4,4] + p[2] \cdot p[3] \cdot p[4] = 0 + 0 + 15 \cdot 5 \cdot 10 = 750 \\ = 750.$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p[1] \cdot p[2] \cdot p[5] = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p[1] \cdot p[3] \cdot p[5] = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p[1] \cdot p[4] \cdot p[5] = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} \\ = 7125.$$

- Jumlah minimum perkalian skalar untuk mengalikan 6 matriks adalah $m[1,6] = 15125$, dan kurung optimal adalah $(A_1(A_2A_3))((A_4A_5)A_6)$.



Step 4:

Bangun solusi optimal dari informasi yang dikomputasi.

- Algoritma MATRIX-CHAIN-ORDER menentukan jumlah perkalian skalar optimal yang diperlukan untuk menghitung produk matrix-chain. Itu tidak secara langsung menunjukkan cara mengalikan matriks.
- Solusi optimal dapat dibangun dari yang dihitung informasi yang disimpan dalam tabel $s[1..n, 1..n]$.
- Setiap entri $s[i, j]$ mencatat nilai k sedemikian sehingga optimal tanda kurung dari $A_i A_{i+1} \dots A_j$ membagi produk antara A_k dan A_{k+1} . Jadi, perkalian matriks akhir dalam komputasi $A_{1..n}$ secara optimal adalah $A_{1..s[1,n]}; A_{s[1,n]+1..n}$

Multiplikasi matriks sebelumnya dapat dihitung secara rekursif: $s[1, s[1, n]]$ menentukan perkalian matriks terakhir yang mengimbangi $A_1 \dots s[1, n]$; $s[s[1, n] + 1, n]$ menentukan perkalian matriks terakhir dalam komputasi $A_{s[1, n] + 1} \dots n$.



- Prosedur rekursif berikut mencetak tanda kurung optimal $(A_i, A_{i+1}, \dots A_j)$, diberikan tabel yang dihitung oleh MATRIX-CHAIN-ORDER dan indeks i dan j .
Panggilan PRINT-OPTIMAL-PARENS $(s, 1, n)$ mencetak optimal tanda kurung $(A_1, A_2, \dots A_n)$.

PRINT-OPTIMAL-PARENS(s, i, j)

1. if $i == j$
2. then print " A_i ;
3. else print "("
4. PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
5. PRINT-OPTIMAL-PARENS($s, s[i, j]+1, j$)
6. print ")"



- Running time dari algoritma dynamic programming tergantung pada produk dari dua faktor: jumlah subproblem dan jumlah pilihan untuk setiap subproblem.
- Untuk perkalian rantai-matriks, ada $\Theta(n^2)$ subproblem berbeda, dan di setiap subproblem paling banyak $n-1$ pilihan. Oleh karena itu, worst case running time-nya adalah $O(n^3)$.
- Dari perspektif teknik, kapan kita harus melihat untuk solusi dynamic programming untuk suatu masalah?
- Dynamic Programming berlaku untuk problem optimasi ketika masalah memiliki dua bahan utama: substruktur optimal dan overlapping subproblem.



Memoization

- Memoisasi adalah variasi dynamic programming yang sering menawarkan efisiensi dari pendekatan dynamic programming yang biasa sambil mempertahankan strategi top-down.
- Seperti dalam dynamic programming biasa, kita maintain tabel dengan solusi subproblem, tetapi proses untuk mengisi tabel lebih seperti algoritma rekursif.
- Algoritma rekursif memoized mempertahankan entri dalam tabel untuk solusi setiap subproblem. Setiap entri tabel awalnya berisi **nilai khusus** untuk menunjukkan bahwa entri belum diisi. Ketika subproblem pertama kali ditemui selama eksekusi algoritma rekursif, solusinya dihitung dan kemudian disimpan dalam tabel. Setiap kali berikutnya masalah yang dihadapi, nilai yang disimpan dalam tabel hanya dilihat dan dikembalikan.



MEMOIZED-MATRIX-CHAIN(p)

1. $n \leftarrow \text{length}[p] - 1$
2. for $i \leftarrow 1$ to n
3. do for $j \leftarrow i$ to n
4. do $m[i, j] \leftarrow \infty$
5. return LOOKUP-CHAIN($p, 1, n$)

LOOKUP-CHAIN(p, i, j)

1. if $m[i, j] < \infty$ then return $m[i, j]$
2. if $i == j$ then $m[i, j] \leftarrow 0$
3. else for $k \leftarrow i$ to $j-1$
4. do $q \leftarrow \text{LOOKUP-CHAIN}(p, i, k)$
 $+ \text{LOOKUP-CHAIN}(p, k+1, j) + p[i-1] p[k] p[j]$
5. if $q < m[i, j]$ then $m[i, j] \leftarrow q$
6. return $m[i, j]$



LONGEST COMMON SUBSEQUENCE (LCS)



Longest Common Subsequence (LCS)

- Aplikasi dalam bioinformatika: membandingkan DNA (asam deoksiribonukleat) dari dua atau lebih organisme yang berbeda.
- Untaian DNA terdiri dari serangkaian molekul yang disebut basa, di mana basis yang mungkin adalah adenin, guanin, sitosin, dan timin. Merepresentasikan masing-masing basis ini dengan huruf awal mereka, seuntai DNA dapat dinyatakan sebagai string di atas himpunan terbatas {A, C, G, T}.
- Misalnya, DNA dari satu organisme mungkin
S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA,
sedangkan DNA dari organisme lain mungkin
S2 = GTCGTTTCGGAATGCCGTTGCTCTGTAAA.
- Salah satu tujuan membandingkan dua untai DNA adalah untuk menentukan seberapa "mirip" kedua untai itu, karena beberapa ukuran seberapa dekat hubungan kedua organisme itu dilihat dari DNAny.



Longest Common Subsequence (LCS)

- Salah satu cara untuk mengukur kemiripan untai S1 dan S2 adalah dengan menemukan untai S3 terpanjang. Basis pada S3 harus muncul dalam S1 dan S2 dalam urutan yang sama, tetapi tidak harus secara berurutan.

Untai S3 terpanjang adalah GTCGTCGGAAGCCGGCCGAA

- Gagasan kesamaan ini dapat diformalkan sebagai longest common subsequence problem.
- Suburutan dari urutan yang diberikan hanyalah urutan dengan nol atau lebih elemen yang tersisa.

Sebagai contoh, $Z = \langle B, C, D, B \rangle$ adalah kelanjutan dari $X = \langle A, B, C, B, D, A, B \rangle$.



Longest Common Subsequence (LCS)

- Diberikan dua urutan X dan Y, kita mengatakan bahwa urutan Z adalah common subsequence dari X dan Y jika Z adalah urutan kedua X dan Y.
- Misalnya, jika $X = \langle A, B, C, B, D, A, B \rangle$, dan $Y = \langle B, D, C, A, B, A \rangle$, urutan $\langle B, C, A \rangle$ adalah urutan umum dari X dan Y, dan begitu juga urutan $\langle B, C, B, A \rangle$.

Urutan $\langle B, C, B, A \rangle$ adalah *longest common subsequence* (LCS) X dan Y, karena tidak ada urutan X dan Y yang umum yang memiliki panjang lebih dari 4.

Urutan $\langle B, D, A, B \rangle$ juga merupakan LCS dari X dan Y.

Longest-Common-Subsequence (LCS) problem:

Diberikan dua urutan X dan Y, cari maximum-length common subsequence dari X dan Y.



- Pendekatan brute-force untuk memecahkan masalah LCS adalah dengan sebutkan semua urutan $X = \langle x_1, x_2, \dots, x_m \rangle$ dan periksa setiap urutan untuk melihat apakah itu juga merupakan urutan dari Y , menjaga jejak yang paling lama ditemukan.

Ada 2^m urutan X , jadi pendekatan ini membutuhkan waktu eksponensial.

- Masalah LCS dapat dipecahkan secara efisien menggunakan dinamis pemrograman, sebagai berikut.

Step 1: Mencirikan longest common subsequence

- Notasikan: Diberikan sebuah sekuen $X = \langle x_1, x_2, \dots, x_m \rangle$, kita menentukan k prefix dari X , untuk $k = 0, 1, 2, \dots, m$, sebagai $X_k = \langle x_1, x_2, \dots, x_k \rangle$. Sebagai contoh, jika $X = \langle A, B, C, B, D, A, B \rangle$, maka $X_4 = \langle A, B, C, B \rangle$ dan X_0 adalah sekuen kosong $\langle \rangle$.
- Masalah LCS memiliki properti substruktur yang optimal.



Substruktur Optimal LCS

Misalkan $X = \langle x_1, x_2, \dots, x_m \rangle$ dan $Y = \langle y_1, y_2, \dots, y_n \rangle$ adalah sekuen, and misalkan $Z = \langle z_1, z_2, \dots, z_k \rangle$ adalah sebuah 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} .
- Karakterisasi ini menunjukkan bahwa LCS dari dua urutan berisi di dalamnya LCS awalan dari dua urutan.
 - Dengan demikian, masalah LCS memiliki properti substruktur yang optimal.
 - Perhatikan juga bahwa masalah LCS memiliki properti overlappingsubblem. Untuk menemukan LCS X dan Y , kita mungkin perl untuk menemukan LCS untuk X dan Y_{n-1} dan X_{m-1} dan Y . Tapi masing-masing submasalah ini memiliki submasalah untuk menemukan LCS X_{m-1} dan Y_{n-1} .



Step 2: Sebuah Solusi Rekursif

Misalkan $c[i, j]$ menunjukkan panjang LCS dari urutan X_i dan Y_j .

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

Perhatikan bahwa subproblem dikesampingkan berdasarkan kondisi di problem.



- Langkah 3: Menghitung panjang LCS
- Berdasarkan persamaan pada slide sebelumnya, kita dapat dengan mudah menulis algoritme rekursif eksponensial-waktu untuk menghitung panjangnya dari LCS. Namun, karena hanya ada $\Theta(mn)$ yang berbeda submasalah, kita dapat menggunakan pemrograman dinamis untuk menghitung solusi dari bawah ke atas.
- Prosedur LCS-LENGTH berikut membutuhkan dua urutan $X = \langle x_1, x_2, \dots, x_m \rangle$ dan $Y = \langle y_1, y_2, \dots, y_n \rangle$ sebagai input. Ini menyimpan $c[i, j]$ nilai dalam tabel $c[0 \dots m, 0 \dots n]$ yang entrinya dihitung dalam urutan baris-utama. Ini juga mempertahankan tabel $b[1 \dots m, 1 \dots n]$ menjadi menyederhanakan konstruksi solusi optimal.
 - $b[i, j]$ menunjuk ke entri tabel yang sesuai dengan optimal solusi subproblem dipilih saat menghitung $c[i, j]$.
 - $c[m, n]$ berisi panjang LCS X dan Y .



LCS-LENGTH(X, Y)

1. $m \leftarrow \text{length}[X]$
2. $n \leftarrow \text{length}[Y]$
3. for $i \leftarrow 1$ to m do $c[i, 0] \leftarrow 0$
4. for $j \leftarrow 0$ to n do $c[0, j] \leftarrow 0$
5. for $i \leftarrow 1$ to m
6. do for $j \leftarrow 1$ to n
7. do if $x_i == y_j$
8. then $c[i, j] \leftarrow c[i-1, j-1] + 1$
9. $b[i, j] \leftarrow "\nwarrow"$
10. else if $c[i-1, j] \geq c[i, j-1]$
11. then $c[i, j] \leftarrow c[i-1, j]$
12. $b[i, j] \leftarrow "\uparrow"$
13. else $c[i, j] \leftarrow c[i, j-1]$
14. $b[i, j] \leftarrow "\leftarrow"$
15. return c and b



Contoh:

- The tables c and b produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$:

		j	0	1	2	3	4	5	6
			y_j	(B)	D	(C)	A	(B)	(A)
i	x_i	0	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	



- Running time LCS-LENGTH adalah $O(mn)$, karena setiap tabel entri membutuhkan waktu $O(1)$ untuk menghitung.

Langkah 4: Membangun LCS

- Tabel b dapat digunakan untuk membuat LCS X dan Y dengan cepat. Kita cukup mulai dari b [m, n] dan menelusuri tabel berikut ini panah.
- An “ \nwarrow ” pada entri b [i, j] menyiratkan bahwa $x_i = y_j$ adalah elemen dari LCS.
- Prosedur rekursif PRINT-LCS mencetak LCS X dan Y dalam urutan yang benar.
- Panggilan awal adalah PRINT-LCS (b, X, panjang [X], panjang [Y]).
- Waktu berjalan PRINT-LCS adalah $O(m + n)$, karena setidaknya satu i dan j dikurangi pada setiap tahap rekursi.



PRINT-LCS(b, X, i, j)

1. if $i == 0$ or $j == 0$
2. then return
3. if $b[i, j] == "\nwarrow"$
4. then PRINT-LCS(b, X, i-1, j-1)
5. print x_i
6. elseif $b[i, j] == "\uparrow"$
7. then PRINT-LCS(b, X, i-1, j)
8. else PRINT-LCS(b, X, i, j-1)



**ANY
QUESTIONS?**



Sesi Materi Berakhir
TERIMA KASIH