Introduction and Background
Advanced Algorithms & Data Structures

COMP4500/7500

Sep 8, 2020

Overview of this week

- Dynamic programming examples
  - Calculating Fibonacci numbers
  - Longest common subsequence (LCS) in strings
  - Matrix-chain multiplication example

Dynamic programming

Method for efficiently solving problems of certain kind.

- May apply to problems with **optimal sub-structure**:
  - An *optimal solution* to a problem can be expressed (recursively) in terms of *optimal solutions* to *sub-problems*.
- A naive recursive solution may be ineffficient (exponential) due to **repeated computations of subproblems.**
- Dynamic programming **avoids re-computation of sub-problems** by storing them.

Requires a deep understanding of the problem and its solution; however some standard formats apply.

Dynamic programming

- Dynamic programming constructs a solution “bottom up”:
  - Compute solutions to base-case sub-problems first;
  - then methodically calculate all intervening sub-problems;
  - until the required problem can be computed.
- Massive speed improvements are possible: from exponential-time to polynomial-time.

Memoisation

- **Memoisation** is a type of dynamic programming (i.e. solutions to sub-problems are stored so that they are never recomputed).
- Memoisation is “top-down”, in the same sense as recursion.
- It is often more “elegant” but has slightly worse constant factors.

Fibonacci numbers

Many problems are naturally expressed recursively.
However recursion can be computationally expensive.

Definition (Fibonacci numbers)

$$\begin{aligned} F(i) &= 1 && \text{if } i = 1 \text{ or } i = 2 \\ F(i) &= F(i-1) + F(i-2) && \text{otherwise} \end{aligned}$$

Thus the sequence of Fibonacci numbers starting from 1 is:

1	1	2	3	5	8	13	21	34..
$F(1)$	$F(2)$	$F(3)$	$F(4)$	$F(5)$	$F(6)$	$F(7)$	$F(8)$	$F(9)$..

$$F(35) = 9,227,465$$

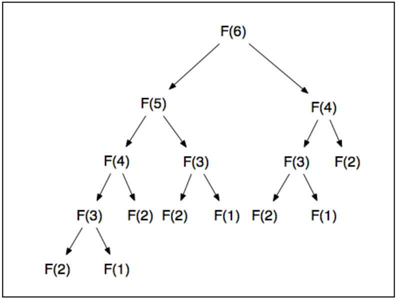
Recursive algorithm for calculating Fibonacci numbers

```
FIB(n)
if n == 1 or n == 2
        return 1
else
        return FIB(n − 1) + FIB(n − 2)
```

Corresponding recurrence:

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \quad \in \Theta(1.6^n)(approx)$$

Recursive calculations



- Lots of overlap/redundancy:
  $F(3)$ is calculated from scratch three times for $F(6)$
- Key idea: Instead of recalculating a number already seen, store the original calculation in an array, and just look it up when encountered later.

Dynamic implementation of Fibonacci

FIB-DYN(n)

```
T = new int[n]
T[1] = 1
T[2] = 1
for i = 3..n
        T[i] = T[i − 1] + T[i − 2]
return T[n]
```

The array elements correspond to the mathematical definition:

$T[1]$	$T[2]$	$T[3]$	$T[4]$	$T[5]$	$T[6]$	$T[7]$	$T[8]$	$T[9]$..
1	1	2	3	5	8	13	21	34..

Analysis of FIB-DYN(n): $T(n) = \sum_{i=3}^{n} \Theta(1) \quad \in \Theta(n)$

Memoised implementation of Fibonacci

Assume array T is a global variable, and we will never require a Fibonacci number past N.

```
FIB-INIT()
T = new int[N]
for i = 1..N
        T[i] = null
T[1] = 1
T[2] = 1
```

```
FIB-MEMO(n)
if T[n] == null
        T[n] = FIB-MEMO(n − 1) + FIB-MEMO(n − 2)
return T[n]
```

General principles of dynamic programming

Solving problems using recursion is often intuitive and elegant. However it can be massively inefficient.

If:

1. the problem has the **optimal substructure** property, and
2. its recursive solution has **overlapping subproblems**

then dynamic programming techniques may apply.
Benefit: One gets an efficient (polynomial-time) implementation (for the loss of elegance).
Fibonacci:

- Optimal substructure: a Fibonacci number can be calculated from smaller Fibonacci numbers
- Overlapping subproblems: $F(6)$ (etc.) recalculates $F(3)$ three times

Longest common subsequence (LCS)

Problem: find the longest (non-contiguous) sequence of characters shared between two strings

$$
\begin{array}{llllll}
S_1: & A & B & C & B & C \\
S_2: & C & A & B & B & D
\end{array}
$$

1. $LCS(S_1, S_2) = ABB$
2. Used in gene sequencing , File `diff`, ...

Developing a recursive description for LCS

Assume you have already solved any *strictly smaller* subproblem (s).

How can you use that to solve your (top-level) problem?

Identify the base cases.

Developing a recursive description for LCS

Assume that we have calculated for

$$S_1 : ABCBC \quad and \quad S_2 : CABBD$$

$$LCS(S_1, S_2) = ABB$$

What is the LCS of

$$ABCBCE \quad and \quad CABBDE?$$

More clearly: what is the LCS of

$$S_1.E \quad and \quad S_2.E?$$

(Note: using '.' for string concatenation)
Answer:

$$ABBE \quad Or: \quad LCS(S_1, S_2).E$$

Developing a recursive description for LCS

What about the LCS of

$$ABCBCX \quad and \quad CABBDY$$

that is,

$$S_1.X \quad and \quad S_2.Y?$$

where $X \neq Y$.

Trap: its *not necessarily* $LCS(S_1, S_2)$.

Recursive description for LCS (cont.)

For *LCS* of

$$S_1.X \quad and \quad S_2.Y$$

where $X \neq Y$, it is possible X is in S_2, or Y is in S_1
For instance: $S_1.D$ and $S_2.E$, that is, for

$$ABCBCD \quad and \quad CABBDE$$

$$LCS(S_1.D, S_2.E) = ABBD$$

Solution: when $X \neq Y$, recursively look at both possibilities, and pick the maximum.

$$LCS(S_1.X, S_2.Y) = \text{MAX}(LCS(S_1.X, S_2), LCS(S_1, S_2.Y))$$

Recall assumption we have the answers to all smaller subproblems

Recursive description for LCS (cont.)

We have covered the cases for $S_1.X$ and $S_2.Y$ where $X = Y$ and $X \neq Y$.

The only other possibility is that one or both are empty – the **base case(s)**.

$$LCS(\langle\rangle, S_2) = LCS(S_1, \langle\rangle) = \langle\rangle$$

where $\langle\rangle$ is the empty string

Recursive description for LCS (cont.)

Put it all together:

Definition (Longest common subsequence (LCS))

$$\begin{array}{rcl} LCS(\langle\rangle, S_2) & = & LCS(S_1, \langle\rangle) = \langle\rangle \\ LCS(S_1.X, S_2.X) & = & LCS(S_1, S_2).X \\ LCS(S_1.X, S_2.Y) & = & \textsc{max}(LCS(S_1, S_2.Y), LCS(S_1.X, S_2)) \\ & & \quad \text{provided } X \neq Y \end{array}$$

Recursive description for LCS Length

Simplify to finding the *length* of an *LCS*.

Definition (Length of longest common subsequence)

$$\begin{aligned} LCS(\langle\rangle, S_2) &= LCS(S_1, \langle\rangle) = 0 \\ LCS(S_1.X, S_2.X) &= LCS(S_1, S_2) + 1 \\ LCS(S_1.X, S_2.Y) &= \textsc{max}(LCS(S_1, S_2.Y), LCS(S_1.X, S_2)) \\ &\quad \text{provided } X \neq Y \end{aligned}$$

Recursive calculations

Recursive implementation using arrays and indexes.

Initial call: $\text{LCS-LENGTH-REC}(S_1, S_2, n, m)$

where $length(S_1) = n$ and $length(S_2) = m$ (indexing from 1)

$\text{LCS-LENGTH-REC}(S_1, S_2, i, j)$

if $i == 0$ or $j == 0$ // Base case
    return 0
else
    if $S_1[i] == S_2[j]$
        return $\text{LCS-LENGTH-REC}(S_1, S_2, i-1, j-1) + 1$
    else
        return $\text{MAX}($
            $\text{LCS-LENGTH-REC}(S_1, S_2, i-1, j),$
            $\text{LCS-LENGTH-REC}(S_1, S_2, i, j-1))$

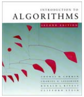
Recursive calculations

- There are $\Omega(2^{min(n,m)})$ possible subsequences to check
- We have solved an optimisation problem by finding optimal solutions to subproblems.
- A quick inspection confirms there will be overlapping subproblems, and hence extreme inefficiency for this recursive implementation.

Dynamic implementation

```
LCS-LENGTH-DYN(S1, S2, n, m)
T = new int[n][m]        soln = new int[n][m]
for i = 1 to n
    T[i, 0] = 0
for j = 1 to m
    T[0, j] = 0
for i = 1 to n
    for j = 1 to m
        if S1[i] == S2[j]
            T[i, j] = T[i − 1, j − 1] + 1        soln[i, j] = ↖
        else if T[i − 1, j] > T[i, j − 1]
                T[i, j] = T[i − 1, j]        soln[i, j] = ↑
            else
                T[i, j] = T[i, j − 1]        soln[i, j] = ←
```

$T(n) \in \Theta(nm)$



Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Matrix-chain multiplication – problem overview

Task: Find an order in which to multiply the chain of matrices

$$M_1.M_2.M_3 \ldots M_n$$

which has the *least cost* (i.e. will be fastest).

Assume that the matrices may have different dimensions, but that the multiplication is well-defined
(e.g. #columns of M_i = #rows of M_{i+1}).

Example: The matrix chain $M_1.M_2.M_3$ can be multiplied in two different ways:

$$M_1.(M_2.M_3) \quad \text{or} \quad (M_1.M_2).M_3$$

Which one has the least cost (i.e. will be faster)?

Matrix-chain multiplication – problem overview

Why do we care?

Large matrix-chain multiplications are:

- needed in theoretical and applied physics,
- needed in mining large data sets in bioinformatics,
- applicable to graphs, when using an adjacency matrix representation.

Matrix multiplication

Example, with M_1 and M_2 both 2×2 matrices.

$$M_1 = \begin{matrix} w & x \\ y & z \end{matrix} \qquad M_2 = \begin{matrix} \alpha & \beta \\ \gamma & \delta \end{matrix}$$

$$M_1.M_2 = \begin{matrix} w.\alpha + x.\gamma & w.\beta + x.\delta \\ y.\alpha + z.\gamma & y.\beta + z.\delta \end{matrix}$$

Note:

$$M_1.M_2 \neq \begin{matrix} w.\alpha & x.\beta \\ y.\gamma & z.\delta \end{matrix}$$

Matrix multiplication

Matrices M_1 and M_2 can only be multiplied if they are **compatible**: #columns of M_1 = #rows of M_2.

If matrix M_1 is $\mathbf{p} \times \mathbf{q}$ and M_2 is $\mathbf{q} \times \mathbf{r}$ then $M_1.M_2$ is $\mathbf{p} \times \mathbf{r}$.

Example if M_1 is $\mathbf{1} \times \mathbf{3}$ and M_2 is $\mathbf{3} \times \mathbf{2}$:

$$M_1 = \begin{matrix} a & b & c \end{matrix} \qquad M_2 = \begin{matrix} d & g \\ e & h \\ f & i \end{matrix}$$

They are compatible (#columns of M_1 = #rows of M_2 = 3) and:

$$M_1.M_2 = \begin{matrix} a.d + b.e + c.f & a.g + b.h + c.i \end{matrix}$$

Costs of matrix multiplication

A straightforward algorithm to multiply *two* matrices A of dimension $\mathbf{p} \times \mathbf{q}$ and B of dimension $\mathbf{q} \times \mathbf{r}$:

MAT-MULT(A, B, p, q, r)

let C be a new $p \times r$ matrix
for $i = 1$ **to** p
    for $j = 1$ **to** r
        $C_{ij} = 0$
        for $k = 1$ **to** q
            $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$

- Total number of multiplications in the inner-loop is $p.q.r$.
- Time complexity is $\Theta(p.q.r)$.

Matrix multiplication

In general, matrix multiplication is not *commutative*, that is,

$$M_1.M_2 \neq M_2.M_1$$

but it is *associative*, that is,

$$M_1.(M_2.M_3) = (M_1.M_2).M_3$$

The key point motivating the problem is that calculating the result of $M_1.M_2.M_3$ can result in huge differences in time factors depending on which of the above two *parenthesising* choices is made.

Matrix multiplication

Recall if M_1 is $\mathbf{p} \times \mathbf{q}$ and M_2 is $\mathbf{q} \times \mathbf{r}$ then the number of multiplications required is $\mathbf{p}.\mathbf{q}.\mathbf{r}$

$$\begin{aligned} M_1 \quad & is \quad 10 \times 100 \\ M_2 \quad & is \quad 100 \times 5 \\ \text{Cost of } M_1.M_2 \quad & = \quad 10.100.5 \\ & = \quad 5000 \text{ iterations of inner loop} \end{aligned}$$

Matrix-chain multiplication

Now consider $M_1.M_2.M_3$ where

$$\begin{array}{lll} M_1 & is & 10 \times 100 \\ M_2 & is & 100 \times 5 \\ M_3 & is & 5 \times 50 \end{array}$$

By associativity:

- Case 1: $(M_1.M_2).M_3$, or
- Case 2: $M_1.(M_2.M_3)$

Case 1:

$$\begin{array}{rcl} \text{Cost of } M_1.M_2 & = & 5000 \\ \text{Cost of } (10 \times 5).M_3 & = & 10.5.50 = 2500 \\ & = & 7500 \textbf{ total} \end{array}$$

Case 2:

$$\begin{array}{rcl} \text{Cost of } M_2.M_3 & = & 100.5.50 = 25,000 \\ \text{Cost of } M_1.(100 \times 50) & = & 10.100.50 = 50,000 \\ & = & 75,000 \textbf{ total} \end{array}$$

Matrix-chain multiplication: the task

Task: Find an order in which to multiply the chain of matrices

$$M_1.M_2.M_3 \dots M_n$$

which has the *least cost* (i.e. will be fastest), assuming that each matrix $\mathbf{M_i}$ has dimensions $\mathbf{p_{i-1}} \times \mathbf{p_i}$ so that:

- each adjacent pair of matrices is compatible (#columns of M_i = #rows of M_{i+1}).
- The cost of $M_i.M_{i+1}$ is $p_{i-1}.p_i.p_{i+1}$.
- Matrix $M_i.M_{i+1}$ is of dimensions $\mathbf{p_{i-1}} \times \mathbf{p_{i+1}}$.

Remember that the cost depends on the order.
E.g. $M_1.(M_2.M_3)$ might be cheaper than $(M_1.M_2).M_3$.

Matrix-chain multiplication: the solution?

Why can't we just enumerate each possible way of multiplying n matrices together and check to see which one is cheapest?

How may ways are there of multiplying n matrices together?

$$M_1.M_2.....M_{n-1}.M_n$$

$$\begin{aligned} N(1) &= ? \\ N(2) &= ? \\ N(3) &= ? \\ N(4) &= ? \\ N(n) &= ? \quad \text{– can you define this as a recurrence relation?} \end{aligned}$$

Quick question

How may ways are there of multiplying n matrices together?

$$
\begin{array}{rcll}
N(1) & = & 1 & (M_1) \\
N(2) & = & 1 & \\
 & = & N(1) \times N(1) & (M_1).(M_2) \\
N(3) & = & 2 & \\
 & = & N(1) \times N(2) + & (M_1).(M_2.M_3) \\
 & & N(2) \times N(1) & (M_1.M_2).(M_3) \\
N(4) & = & 5 & \\
 & = & N(1) \times N(3) + & M_1.(M_2.(M_3.M_4)), \quad M_1.((M_2.M_3).M_4) \\
 & & N(2) \times N(2) + & (M_1.M_2).(M_3.M_4) \\
 & & N(3) \times N(1) & (M_1.(M_2.M_3)).M_4, \quad ((M_1.M_2).M_3).M_4
\end{array}
$$

$$N(n) \quad = \quad \textstyle\sum_{i=1}^{n-1} N(i) \times N(n-i)$$

Quick question

Show that $N(n) \in \Omega(2^n)$ where

$$\begin{aligned} N(1) &= 1 \\ N(n) &= \sum_{i=1}^{n-1} N(i) \times N(n-i) \end{aligned}$$

We have that:

$$\begin{aligned} N(1) &= 1 \\ N(n) &\geq N(1) \times N(n-1) + N(n-1) \times N(1) = 2 \times N(n-1) \end{aligned}$$

and we can solve the simpler lower-bound recurrence for the solution we seek.

Matrix-chain multiplication

Task: find the least cost of muliplying a chain of n matrices:

$$M_1.M_2.....M_{n-1}.M_n$$

Problem: there are an exponential number of possible ways to muliply them ($\Omega(2^n)$).

Solution: try to find a dynamic programming solution ...

- first step: think about the problem recursively (ignoring efficiency).
- second step: apply dynamic programming

Matrix-chain multiplication: recursive solution

Identify the sub-problems:
Let $C_{i..j}$ give the minimum cost of multiplying $M_i.M_{i+1}.\ldots.M_j$.
The solution to our problem will be given by $C_{1..n}$.

Identify the base cases:
The solution to each sub-problem $C_{i..i}$ is 0.

Define the recursive case:
Look again at the chain of length 4.

$$C_{1..4} = min \begin{cases} C_{1..1} + C_{2..4} + p_0.p_1.p_4 & M_1.(\ldots) \\ C_{1..2} + C_{3..4} + p_0.p_2.p_4 & (M_1.M_2).(M_3.M_4) \\ C_{1..3} + C_{4..4} + p_0.p_3.p_4 & (\ldots).M_4 \end{cases}$$

where

$$C_{2..4} = min \begin{cases} C_{2..3} + C_{4..4} + p_1.p_3.p_4 & (M_2.M_3).M_4 \\ C_{2..2} + C_{3..4} + p_1.p_2.p_4 & M_2.(M_3.M_4) \end{cases}$$

Matrix-chain multiplication: recursive solution

More generally:

$$C_{i..j} = min_{i \leq k < j}\{C_{i..k} + C_{k+1..j} + p_{i-1}.p_k.p_j\}$$

Final calculation:

$$C_{1..n} = min_{1 \leq k < n}\{C_{1..k} + C_{k+1..n} + p_0.p_k.p_n\}$$

Expanded:

$$C_{1..n} = min \begin{cases} C_{1..1} + C_{2..n} + p_0.p_1.p_n & (k = 1) \\ C_{1..2} + C_{3..n} + p_0.p_2.p_n & (k = 2) \\ C_{1..3} + C_{4..n} + p_0.p_3.p_n & (k = 3) \\ \ldots \\ C_{1..n-1} + C_{n..n} + p_0.p_{n-1}.p_n & (k = n - 1) \end{cases}$$

Very inefficient to implement directly, but contains the intuition.

Dynamic solution for matrix-chain multiplication

Key insight 1: each (sub)problem $C_{i..j}$ depends on every ((sub)sub)problem

$$C_{i..k} \quad \text{and} \quad C_{k+1..j}$$

Key insight 2: Furthermore, each subproblem may be recalculated many times in the recursive definition.

Let's **convert it into a dynamic programming solution**:

- Calculate and store each sub-problem once only.
- Reduce an exponential-time solution to polynomial $\Theta(n^3)$!

Dynamic solution: storing solutions to subproblems

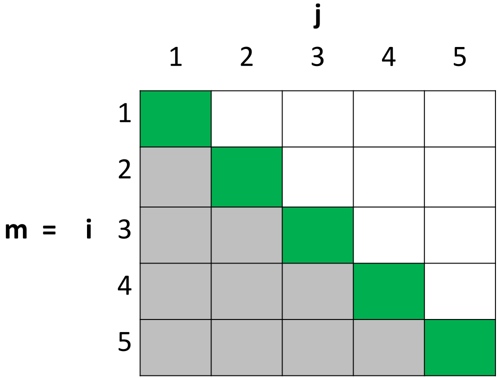
Store the solution $\mathbf{C_{i..j}}$ at $\mathbf{m}[\mathbf{i},\mathbf{j}]$ in a $\mathbf{n} \times \mathbf{n}$ array $\mathbf{m}$:

		j				
		1	2	3	4	5
	1					
	2					
m = i	3					
	4					
	5					

- $\mathbf{m}[\mathbf{i},\mathbf{j}]$ is the minimum cost parenthesisation of $\mathbf{M_i} \ldots \mathbf{M_j}$.
- We are only every interested in entries $\mathbf{m}[\mathbf{i},\mathbf{j}]$ where $\mathbf{i} \leq \mathbf{j}$, since the cost of $C_{5..2}$ is nonsensical.

Dynamic solution: calculating subproblems

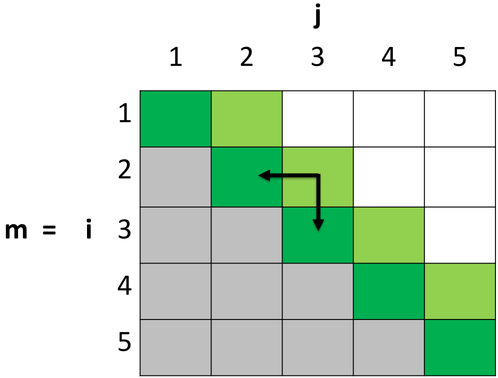
What order should we calculate sub-problems?



- **Base cases** (**m**[**i**, **i**] $=$ **0**) don't depend on any other problems, and can be calculated first.
- What about the rest?

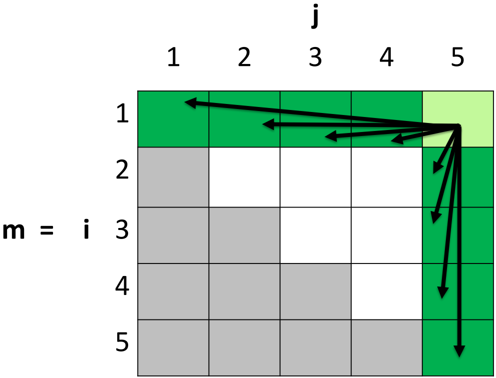
Dynamic solution: calculating subproblems

What order should we calculate sub-problems?



- For $1 \leq i < n-1$, cases $\mathbf{m}[\mathbf{i}, \mathbf{i}+\mathbf{1}]$ depend on $\mathbf{m}[\mathbf{i}, \mathbf{i}]$ and $\mathbf{m}[\mathbf{i}+\mathbf{1}, \mathbf{i}+\mathbf{1}]$, and so they could be calculated next.
- More generally?

Dynamic solution: calculating subproblems

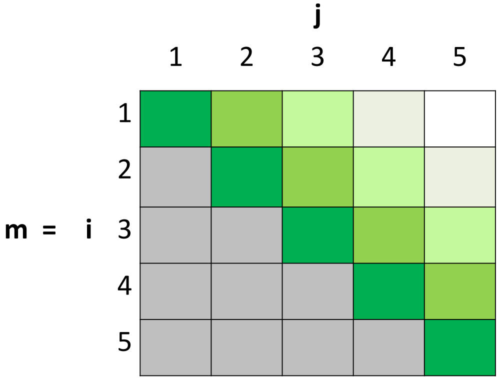


- Consider entry $m[1,5]$. It depends on:

$$\begin{array}{cccc} m[1,1] & m[1,2] & m[1,3] & m[1,4] \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ m[2,5] & m[3,5] & m[4,5] & m[5,5] \end{array}$$

Dynamic solution: calculating subproblems

What order should we calculate sub-problems?



- calculate all $\mathbf{m}[\mathbf{i},\mathbf{i}]$ for $1 \le i \le n$, then
- calculate all $\mathbf{m}[\mathbf{i},\mathbf{i}+\mathbf{1}]$ for $1 \le i \le n{-}1$, then
- calculate all $\mathbf{m}[\mathbf{i},\mathbf{i}+\mathbf{2}]$ for $1 \le i \le n{-}2$, etc.

Dynamic solution: code for matrix-chain multiplication

Finding a solution

MATRIX-CHAIN-ORDER(p, n)

$m = \text{new } int[n, n] \quad \mathbf{s} = \text{ new } \mathbf{int}[\mathbf{n}, \mathbf{n}]$
for $i = 1$ **to** n
    $m[i, i] = 0 \quad \mathbf{s}[\mathbf{i}, \mathbf{i}] = \mathbf{i}$
for $l = 2$ **to** n
    for $i = 1$ **to** $n - l + 1$
        $j = i + l - 1$
        $m[i, j] = min_{i \leq k < j}\{$
            $m[i, k] + m[k + 1, j] + (p_{i-1}.p_k.p_j)$
        $\}$
        $\mathbf{s}[\mathbf{i}, \mathbf{j}] = \mathbf{k}$

For $\Theta(n^2)$ sub-problems we must find the minimum of a set of possibilities, making it a $\Theta(n^3)$ operation.

Dynamic solution: code for matrix-chain multiplication

```
MATRIX-CHAIN-ORDER(p, n)
for i = 1 to n
        m[i, i] = 0
for l = 2 to n
        for i = 1 to n − l + 1
                j = i + l − 1
                m[i, j] = ∞
                for k = i to j − 1
                        q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                        if q < m[i, j]
                                m[i, j] = q
                                s[i, j] = k
return s
```

Overview of today

Matrix-chain multiplication example:

- Recursive structure of solution
- Dynamic programming solution
- Performing the matrix-chain multiplication

Notation for finding the minimum/maximum

Dynamic programming applies to *optimisation problems*, (find max or min)

- Example: Find the minimum element in array A of size n:

  $$\text{MIN}(A[1], A[2], .., A[n])$$

  More convenient/general notation:

  $$\text{MIN}_{1 \leq k \leq n} A[k]$$

  Note that this finds the minimum *value* in the array *not* the index of the minimum value
- Find the maximum weight edge in a graph:

  $$\text{MAX}(weight(u, v), weight(u, w), \ldots, weight(v, u) \ldots)$$
  $$(\text{MAX}(weight(e_1), weight(e_2), .., weight(e_{|E|})))$$

  More succinctly:

  $$\text{MAX}_{u,v \in V} weight(u, v) \qquad \text{MAX}_{e \in E} weight(e)$$

Notation for finding the minimum/maximum

In general,

$$\text{MIN}_{i \in \{a,b,..z\}} e \quad = \quad \text{MIN}(e[a/i], e[b/i], \ldots, e[z/i])$$

Where $e[a/i]$ replaces i with a in expression e:

$$(A[i])[3/i] = A[3]$$

We commonly use MIN for a range of *numbers*:

$$min_{1 \leq i \leq N} e = min(e[1/i], e[2/i], \ldots, e[N/i])$$

(Note: $1 \leq i \leq N$ is a readable shorthand for $i \in 1..N$)
Translation to code is straightforward:

Finding the index

$min = \infty \quad \mathbf{k} = \mathbf{-1}$
for $i = 1..N$
    if $e < min$
        $min = e \quad \mathbf{k} = \mathbf{i}$

$min = \infty$
for $i = 1..N$
    $min = \text{MIN}(min, e)$