Curtin University of Technology

Design and Analysis of Algorithms

Lecture 09 + 10

Dynamic Programming

Dynamic programming

Topics:

- Basics of DP
- Knapsack Problem
- Matrix Chain Multiplication
- Longest common sub-sequences (LCS)
- Memoization

Read Chapter 15 (16) of Textbook

Curtin
University of Technology

Dynamic programming

- Similar to divide & conquer, but sub-problems are not independent
- Sub-problem optimality leads to overall optimality
- Solution to each sub-problem is saved, rather than recomputed
- Begin with recursive solution then compute bottom-up
  - We will also learn the top-down approach (called memoization)

Curtin University of Technology

0/1 Knapsack problem

- Have n items; each item i has
  - weight w_i and
  - profit p_i
- Have a backpack which can hold c kilos
- Which items do I put in the backpack for maximum profit?

For formally:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is in the bag} \\ 0 & \text{if item } i \text{ is not in the bag} \end{cases}$$

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i \qquad \text{(Total profit)}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \le c \qquad \text{(Not too heavy)}$$

Curtin University of Technology

A greedy solution?

- Greedy on profit?
- Greedy on weight?
- Greedy on profit density $\frac{p_i}{w_i}$?

- A greedy solution is often not optimal

A recursive solution

- Either put the first item in the bag then

$$\text{maximise} \quad \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c - w_1$$

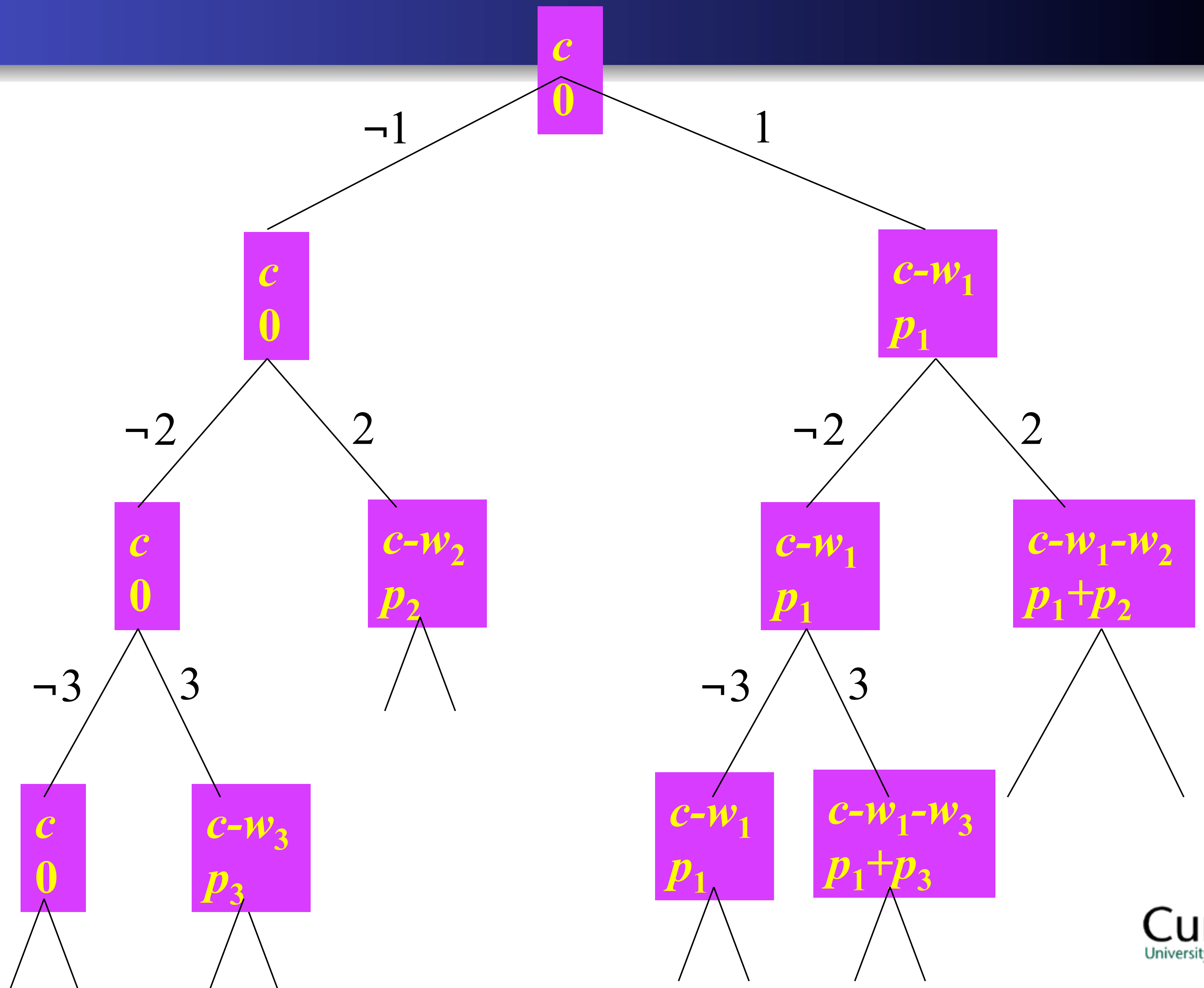
- Or don't put first item in and

$$\text{maximise} \quad \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c$$

Do both and take the maximum

Curtin University of Technology


c
0
$\neg 1$
1
c
0
$c-w_1$
p_1
$\neg 2$
2
$\neg 2$
2
c
0
$c-w_2$
p_2
$c-w_1$
p_1
$c-w_1-w_2$
p_1+p_2
$\neg 3$
3
$\neg 3$
3
c
0
$c-w_3$
p_3
$c-w_1$
p_1
$c-w_1-w_3$
p_1+p_3

Recursion (cont.)

- Define $P(i,k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ & \\ & \\ & \end{cases}$$

- Selecting n is not possible

Recursion (cont.)

- Define $P(i,k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ p_n & i = n \,\&\, w_n \le k \\ & \\ & \end{cases}$$

- Selecting n is possible

Recursion (cont.)

- Define $P(i,k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ p_n & i = n \,\&\, w_n \le k \\ P(i+1,k) & i < n \,\&\, w_i > k \\ & \end{cases}$$

- Selecting i is not possible

Curtin University of Technology

Recursion (cont.)

- Define $P(i,k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ p_n & i = n \,\&\, w_n \le k \\ P(i+1,k) & i < n \,\&\, w_i > k \\ \max(P(i+1,k), p_i + P(i+1,k-w_i)) & i < n \,\&\, w_i \le k \end{cases}$$

- Selecting i is possible, but do we want to?

Curtin University of Technology

Recursion (cont.)

- So, how do we convert *P* into an algorithm?

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ p_n & i = n \,\&\, w_n \le k \\ P(i+1,k) & i < n \,\&\, w_i > k \\ \max(P(i+1,k),\, p_i + P(i+1,k-w_i)) & i < n \,\&\, w_i \le k \end{cases}$$

- Have base cases and recursive call.

Curtin University of Technology

Recursive Solution

KNAPSACK-RECURSE(*i*, *k*)

if $(i = n)$ and $(w_n > k)$ **then return** 0

if $(i = n)$ and $(w_n <= k)$ **then return** p_n

if $(i < n)$ and $(w_i > k)$ **then**
  return KNAPSACK-RECURSE(*i* + 1, *k*)

if $(i < n)$ and $(w_i <= k)$ **then**
  ***x* := KNAPSACK-RECURSE (*i* + 1, *k*)**
  ***y* := KNAPSACK-RECURSE (*i* + 1, *k* - w_i)** + p_i
  return max(*x*, *y*)

Curtin University of Technology

Recursive Solution

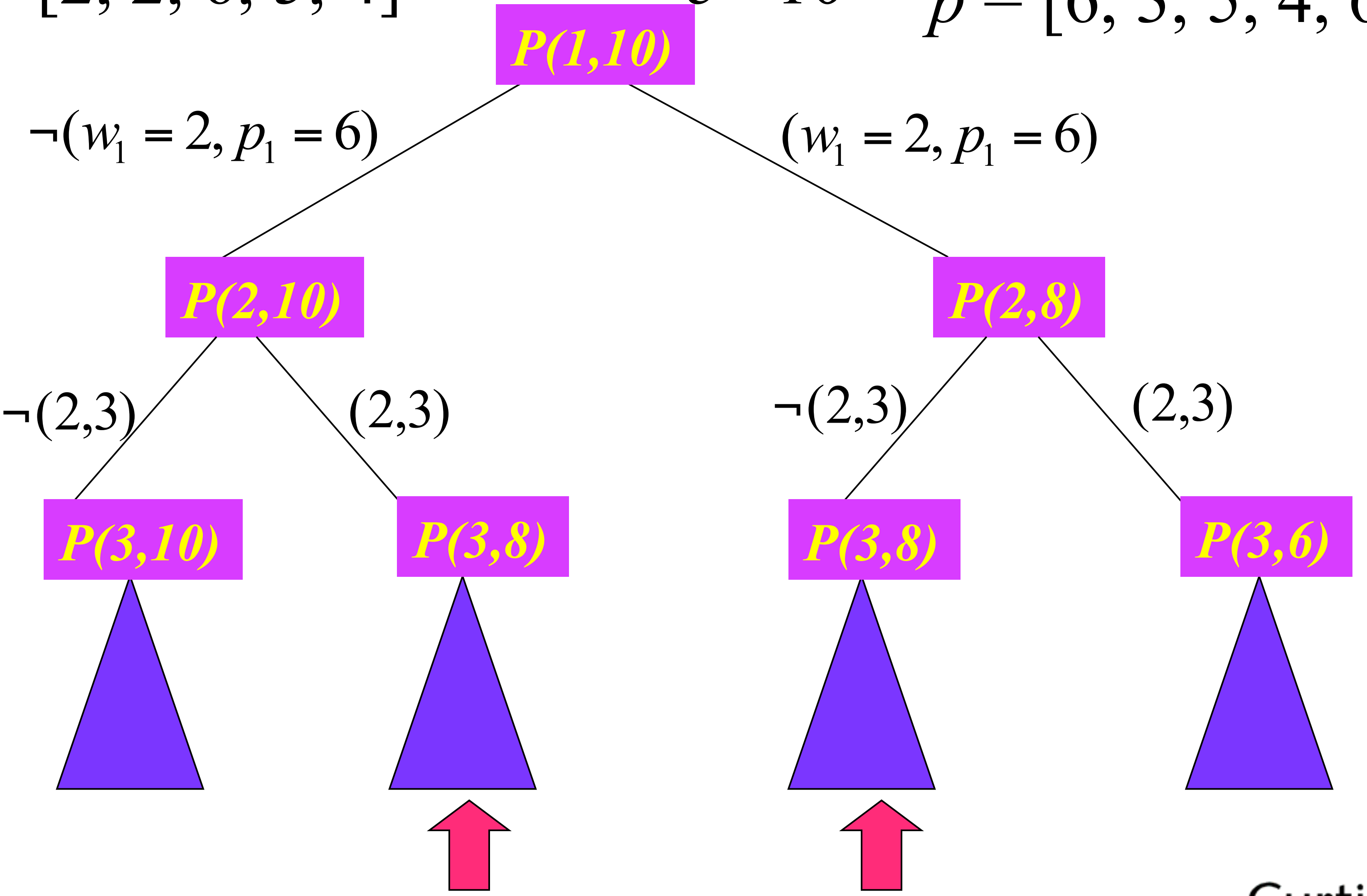
```
KNAPSACK-RECURSE(i, k)
  if (i = n) then
    if (w_n > k) then
      return 0
    else
      return p_n

  if (w_i > k) then
    return KNAPSACK-RECURSE(i +1, k)
  else
    x := KNAPSACK-RECURSE (i + 1, k)
    y := KNAPSACK-RECURSE (i + 1, k - w_i) + p_i
    return max(x, y)
```

Analysis of Recursive Solution

- $O(2^n)$ – prove it!
- Also you can see it from the recursive tree
- Problem is we recalculate some $P(i,k)$
- For example,
  - $n = 5$, $c = 10$
  - $w = [2, 2, 6, 5, 4]$
  - $p = [6, 3, 5, 4, 6]$

$w = [2, 2, 6, 5, 4]$ $c = 10$ $p = [6, 3, 5, 4, 6]$



Same sub-trees

Dynamic Programming

- Store each computed $P(i,k)$
- Use it if already calculated

We can compute $P(n,k)$ easily

k	1	2	…	j-1	j	$j+1$	…	c
$P(n,k)$	0	0	…	0	p_n	p_n	…	p_n

j is the first k where $w_n <= k$

Curtin University of Technology

Example

$n = 5 \quad c = 10 \quad w = [2, 2, 6, 5, 4] \quad p = [2, 3, 5, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4											
3											
2											
1											

Curtin University of Technology

Example

$n = 5 \quad c = 10 \quad w = [2, 2, 6, \mathbf{5}, 4] \quad p = [2, 3, 5, \mathbf{4}, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6						
3											
2											
1											

Curtin University of Technology

Example

$n = 5 \;\; c = 10 \;\; w = [2, 2, 6, \mathbf{5}, 4] \;\; p = [2, 3, 5, \mathbf{4}, 6]$

$i \backslash k$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	0 +4	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6					
3											
2											
1											

Curtin University of Technology

Example

$n = 5$ $c = 10$ $w = [2, 2, 6, \mathbf{5}, 4]$ $p = [2, 3, 5, \mathbf{4}, 6]$

$i \backslash k$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3											
2											
1											

+4

Example

$n = 5 \quad c = 10 \quad w = [2, 2, \mathbf{6}, 5, 4] \quad p = [2, 3, \mathbf{5}, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6					
2											
1											

Example

$n = 5 \;\; c = 10 \;\; w = [2, 2, 6, 5, 4] \;\; p = [2, 3, 5, 4, 6]$

*i**k*	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0 +5	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2											
1											

Example

$n = 5 \;\; c = 10 \;\; w = [2, \mathbf{2}, 6, 5, 4] \;\; p = [2, \mathbf{3}, 5, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0									
1											

Example

$n = 5 \quad c = 10 \quad w = [2, \mathbf{2}, 6, 5, 4] \quad p = [2, \mathbf{3}, 5, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0 +3	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1											

Example

$n = 5 \quad c = 10 \quad w = [2, 2, 6, 5, 4] \quad p = [2, 3, 5, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1									+2		11

Curtin University of Technology

Example

$n = 5 \;\; c = 10 \;\; w = [2, 2, 6, 5, 4] \;\; p = [2, 3, 5, 4, 6]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1											11

$x = [\,0, 0, 1, 0, 1\,]$ OR x= [1,1,0,0,1]

0-1 Knapsack DP Algorithm

Knapsack(S, C)

Input: Set S of n items with p_i profit and w_i weight, and maximum total weight C
Output: maximum profit $P[w]$ of a subset S with total weight at most w, for $w = 0, 1, \ldots, C$

```
for k = 0 to C do
    P[k] = 0
for i = n downto 1 do
    for k = C downto w_i do
        if P[k - w_i] + p_i > P[k] then
            P[k] = P[k - w_i] + p_i
```

Complexity: $O(nC)$ → pseudo polynomial

Memoization

```
KNAPSACK-RECURSE (i, k)
   if P[k] != UNKNOWN
        return P[k]
   if (i = n)
      if (w_n > k)
         return 0
      else
         return p_n
   if (w_i > k)
      return KNAPSACK-RECURSE (i+1, k)
   else
      x = KNAPSACK-RECURSE (i+1, k)
      y = KNAPSACK-RECURSE (i+1, k-w_i)
      P[k] = max (x, y)
      return P[k]
```

Matrix-chain Multiplication

Consider the matrix multiplication procedure

```
MATRIX_MULTIPLY(A,B)
    if columns[A] ≠ rows[B]
        then error "incompatible dimensions”
    else for i ← 1 to rows[A]
        do for j ← 1 to columns[B]
            do C[i, j] ← 0;
                for k ← 1 to columns [A]
                    do C[i, j] ← C[i, j] + A[i, k]*B[k, j];
    return C
```

Matrix-chain Multiplication (Cont.)

The time to compute a matrix product is dominated by the number of scalar multiplications in line 7.

If matrix A is of size $(p \times q)$ and B is of size $(q \times r)$, then the time to compute the product matrix is given by $p \times q \times r$

Example: Consider three matrices A1, A2, and A3 whose dimensions are respectively (10×100), (100×5), (5×50)

There are two ways to parenthesize these multiplications

I $((A_1 \times A_2) \times A_3)$

II $(A_1 \times (A_2 \times A_3))$

Curtin University of Technology

Matrix-chain Multiplication (Example)

First Parenthesization

Product $A_1 \times A_2$ requires $10 \times 100 \times 5 = 5000$ scalar multiplications

$A_1 \times A_2$ is a (10×5) matrix

$(A_1 \times A_2) \times A_3$ requires $10 \times 5 \times 50 = 2500$ scalar multiplications.

Total : 7,500 multiplications

Second Parenthesization

Product $A_2 \times A_3$ requires $100 \times 5 \times 50 = 25{,}000$ scalar multiplications

$A_2 \times A_3$ is a (100×50) matrix

$A_1 \times (A_2 \times A_3)$ requires $10 \times 100 \times 50 = 50{,}000$ scalar multiplications

Total : 75,000 multiplications

The first parenthesization is 10 times faster than the second one!!

How to pick the best parenthesization?

Curtin University of Technology

The matrix-chain multiplication

Problem:

Given a chain (A_1, A_2, . . . , A_n) of n matrices, where for i = 1,2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

The order in which these matrices are multiplied together can have a significant effect on the total number of operations required to evaluate the product.

An optimal solution to an instance of a matrix-chain multiplication problem contains within it optimal solutions to the sub-problem instances

Curtin
University of Technology

Let $P(n)$ be the number of alternative parenthesizations of a sequence of n matrices

We can split a sequence of n matrices between k^{th} and $(k+1)^{\text{th}}$ matrices for any $k = 1, 2, \ldots, n-1$ and we can then parenthesize the two resulting subsequences independently,

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

This is exponential in *n*

P(2) = 1
P(3) = 2

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$
$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$
$(A_1 \times (A_2 \times A_3)) \times A_4$

→ Total: P(4) = 5 alternatives

Consider $A_1 \times A_2 \times A_3 \times A_4 \times A_5$

if k = 1, then

$A_1 \times (A_2 \times A_3 \times A_4 \times A_5) \rightarrow 5$

if k = 2 then

$(A_1 \times A_2) \times (A_3 \times A_4 \times A_5) \rightarrow 2$

if k =3 then

$(A_1 \times A_2 \times A_3) \times (A_4 \times A_5) \rightarrow 2$

if k = 4, then

$(A_1 \times A_2 \times A_3 \times A_4) \times A_5 \rightarrow 5$

→ P(5) = 14 alternatives

$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \rightarrow$ P(6) = 14 + 5 + 2*2 + 5 + 14 = 42 alternatives
$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \rightarrow$ P(7) = 42 + 14 + 2*5 + 5*2 + 14 + 42 = 132
$A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \times A_8 \rightarrow$ P(8) = 132 + … + 132 = 429

Curtin University of Technology

Structure of the Optimal Parenthesization

Let $A_{i..j} = A_i \times A_{i+1} \times \ldots \times A_j$

An optimal parenthesization splits the product
$A_{i..j} = (A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$ for $1 \leq k < n$

The total cost of computing $A_{i..j}$
= cost of computing $(A_i \times A_{i+1} \times \ldots \times A_k)$
+ cost of computing $(A_{k+1} \times A_{k+2} \times \ldots \times A_j)$
+ cost of multiplying the matrices $A_{i..k}$ and $A_{k+1..j}$

$A_{i..k}$ must be optimal if we want $A_{i..j}$ to be optimal. If $A_{i..k}$ is not optimal then $A_{i..j}$ is not optimal.

Similarly $A_{k+1..j}$ must also be optimal.

Recursive Solution

We'll define the value of an optimal solution recursively in terms of the optimal solutions to sub-problems.

$m[i, j]$ = minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$

$m[1, n]$ = minimum number of scalar multiplications needed to compute the matrix $A_{1..n}$.

If $i = j$; the chain consists of just one matrix

$A_{i..i} = A_i$; no scalar multiplications

$m[i, i] = 0$ for $i = 1, 2, \ldots, n$.

$m[i, j]$ = minimum cost of computing the sub-products $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying these two matrices

Multiplying $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}p_kp_j$ scalar multiplications

Thus,

$$\boldsymbol{m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j \text{ for } i \leq k < j}$$

The optimal parenthesization must use one of these values for k, and we need to check them to find the best solution.

Therefore,

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

Let $\boldsymbol{s[i,j]}$ be the value of k at which we can split the product $A_i \times A_{i+1} \times \ldots \times A_j$

To obtain the optimal parenthesization, s[*i*, *j*] equals a value of *k* such that

$$\boldsymbol{m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \text{ for } i \le k < j}$$

Curtin University of Technology

Matrix_Chain_Order (p)

Input: sequence $(p_0, p_1, \ldots, p_n)$
Output: an auxiliary table $m[1..n, 1..n]$ with $m[i, j]$ costs and another auxiliary table $s[1..n, 1..n]$ with records of index k which achieves optimal cost in computing $m[i, j]$

```
n = length[p] - 1
for i = 1 to n
    do m[i, i] = 0
for l = 2 to n
    do for i = 1 to n - l + 1
        do j = i + l - 1
            m[i, j] = ∞
            for k = i to j - 1
                do q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] = q
                             s[i, j] = k
return m and s
```

Consider Four Matrices

A1: 10 × 20 A2: 20 × 50

A3: 50 × 1 A4: 1 ×100

p_0=10, p_1=20, p_2=50, p_3=1, p_4=100

$j \downarrow i \rightarrow$	1	2	3	4
1	0	--	--	--
2	10,000	0	--	--
3	1200	1000	0	--
4	2200	3000	5000	0

Consider $A_1 \times A_2 \times A_3 \times A_4$

if k =1, then

$A_1 \times (A_2 \times (A_3 \times A_4))$

$A_1 \times ((A_2 \times A_3) \times A_4)$

if k =2 then

$(A_1 \times A_2) \times (A_3 \times A_4)$

if k =3 then

$((A_1 \times A_2) \times A_3) \times A_4$

and $(A_1 \times (A_2 \times A_3)) \times A_4$

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \text{ for } i \le k < j$$

Example:

Consider, A1 (30×35), A2 (35×15), A3 (15×5), A4(5×10), A5(10×20), A6(20×25)

$j\downarrow / i\rightarrow$	1	2	3	4	5	6
1	0	--	--	--	--	--
2	15,750	0	--	--	--	--
3	7,875	2,625	0	--	--	--
4	9,375	4,375	750	0	--	--
5	11,875	7,125	2,500	1,000	0	--
6	15,125	10,500	5,375	3,500	5,000	0

$p_0=30$, $p_1=35$, $p_2=15$, $p_3=5$, $p_4=10$, $p_5=20$, $p_6=25$

Curtin University of Technology

Example:

Consider, **A1 (30×35), A2 (35×15), A3 (15×5), A4(5×10), A5(10×20), A6(20×25)**

$$m[1,6] = \min \begin{cases} m[1,1] + m[2,6] + p_0 p_1 p_6 = 0 + 10500 + 30x35x25 = 36750 \\ m[1,2] + m[3,6] + p_0 p_2 p_6 = 15750 + 5375 + 30x15x25 = 32375 \\ m[1,3] + m[4,6] + p_0 p_3 p_6 = 7875 + 3500 + 30x5x25 = 15125 \\ m[1,4] + m[5,6] + p_0 p_4 p_6 = 9375 + 5000 + 30x10x25 = 21875 \\ m[1,5] + m[6,6] + p_0 p_5 p_6 = 11875 + 0 + 30x20x25 = 26875 \end{cases}$$

$$= 15125$$

Curtin University of Technology

Example:

j↓/i→	1	2	3	4	5
2	1	-	-	-	-
3	1	2	-	-	-
4	3	3	3	-	-
5	3	3	3	4	-
6	3	3	3	5	5

(A1 .. A3) × (A4 .. A6)

(A1 × (A2 × A3)) × ((A4 × A5) × A6)

Curtin University of Technology

Print optimal parenthesis from *s*

Print-Optimal-Parens (*s*, *i*, *j*)

```
if i=j
  then print “A_i”
  else print “(“
        Print-Optimal-Parens (s, i, s[i, j])
        Print-Optimal-Parens (s, s[i, j]+1, j)
        print “)“
```

Summary of approach

- Design recursive solution
- Analyse recursive solution
- Look for repeated calculations
- Store recursive results bottom-up in table
- Traverse table to get solution

Curtin University of Technology

Longest common subsequence (LCS)

A subsequence is formed from a sequence by deleting zero or more elements (the remaining elements are in order)

Example:
Z = (**B A B A**) is a subsequence of X = (A **B** C **A** **B** B **A**)

Also: A, ACAA, etc. There are 2^m subsequences of a sequence of length m

- A **common** subsequence of two sequences is a subsequence of both.
- A **longest common** subsequence (LCS) of two sequences is the longest among the common subsequences of both.

Problem: Given two sequences X[1…m] and Y[1 …n], find **one** longest common subsequence of both.

Curtin University of Technology

Example:

Two sequences:

X = A B C A B B A

Y = C B A B C A

Also: CBBA, ABCA, etc

B A B A is one of the longest common subsequences of both

LCS Application

Application for Biology:

To compare the similarity between two or more DNA strands.

- A DNA strand consists of a string of molecules called *bases*.
- Possible bases:
  A = Adenine, G = Guanine, C = Cytosine, T = Thymine
  Set = {A, G, C, T}

LCS Application (Cont.)

Example:

S1= ACCGGTCGAGT

S2= GTCGTTCGGAAT

Similarity Criteria:

a) Similar if one string is a substring of another.

b) Similar if the number of changes needed to turn one into the other is small.

c) Find a string S3 where:
   - The elements in S3 appear in each of S1 and S2.
   - The elements must appear in the same order, but NOT necessarily consecutive
   - The longer the string S3 the more similar S1 and S2.

LCS Application (Cont.)

Unix Command *diff*

Compare two text files for their differences.

Consider two files:

X = $a_1, a_2, \ldots, a_m$ → current file; a_i = the ith line of file X

Y = $b_1, b_2, \ldots, b_n$ → modified file; b_i = the ith line of file Y

diff makes assumption that we can identify what the changes are by:

a) Finding an LCS → represent lines that have not been changed.
b) The lines changed can be obtained by removing unchanged parts

Curtin University of Technology

Brute force to find LCS

$\mathbf{X} = (x_1, x_2 , \ldots, x_m)$

$\mathbf{Y} = (y_1, y_2 , \ldots, y_n)$

- Brute force → check for each subsequence of X to see if it is also a subsequence of Y

Curtin University of Technology

Brute-Force Analysis

- There are 2^m subsequences of X
  - Think of a subsequence as a binary number
    - 1-bit character is *in*, 0-bit character is *out*
  - How many *m*-bit binary numbers are there?

- Checking = $O(n)$ time per subsequence

- Worst-case = $O(n\ 2^m)$ → exponential

Optimal-substructure of an LCS

Theorem 15.1 (Textbook)

Let $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$ be sequences, and let $Z = (z_1, z_2, \ldots, z_k)$ 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}

Curtin University of Technology

Optimal … Example

For Theorem 15.1 (Textbook)

1. **X = (ABCAD) and Y = (BCAFD)**
   If $x_m = y_n$, then $z_k = x_m = y_n = D$, and Z_{k-1} is an LCS of
   X_{m-1} = (ABCA) and Y_{n-1} = (BCAF)
   → Z = (…… D)

2. **X = (ABCBD) and Y = (ACAF)**
   If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z_k is an LCS of
   X_{m-1} = (ABCB) and Y = (ACAF)

3. **X = (ABCBD) and Y = (ACAF)**
   If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z_k is an LCS of
   X = (ABCBD) and Y_{n-1} = (ACA)

Curtin University of Technology

A recursive solution

There are either one or two subproblems to examine when finding an LCS of X and Y:

If $x_m = y_n$ then

- find LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$
- LCS of X and Y is obtained by appending x_m to the LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$

If $x_m \neq y_n$ then

- find LCS of $X_{m\text{-}1}$ and Y_n and
- find LCS of X_m and $Y_{n\text{-}1}$
- The longest LCS between the two LCS is the LCS of X and Y

Curtin University of Technology

A recursive solution

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

***c*[*i*, *j*] is the length of an LCS of the sequences X_i and Y_j**

Recursive algorithm for LCS

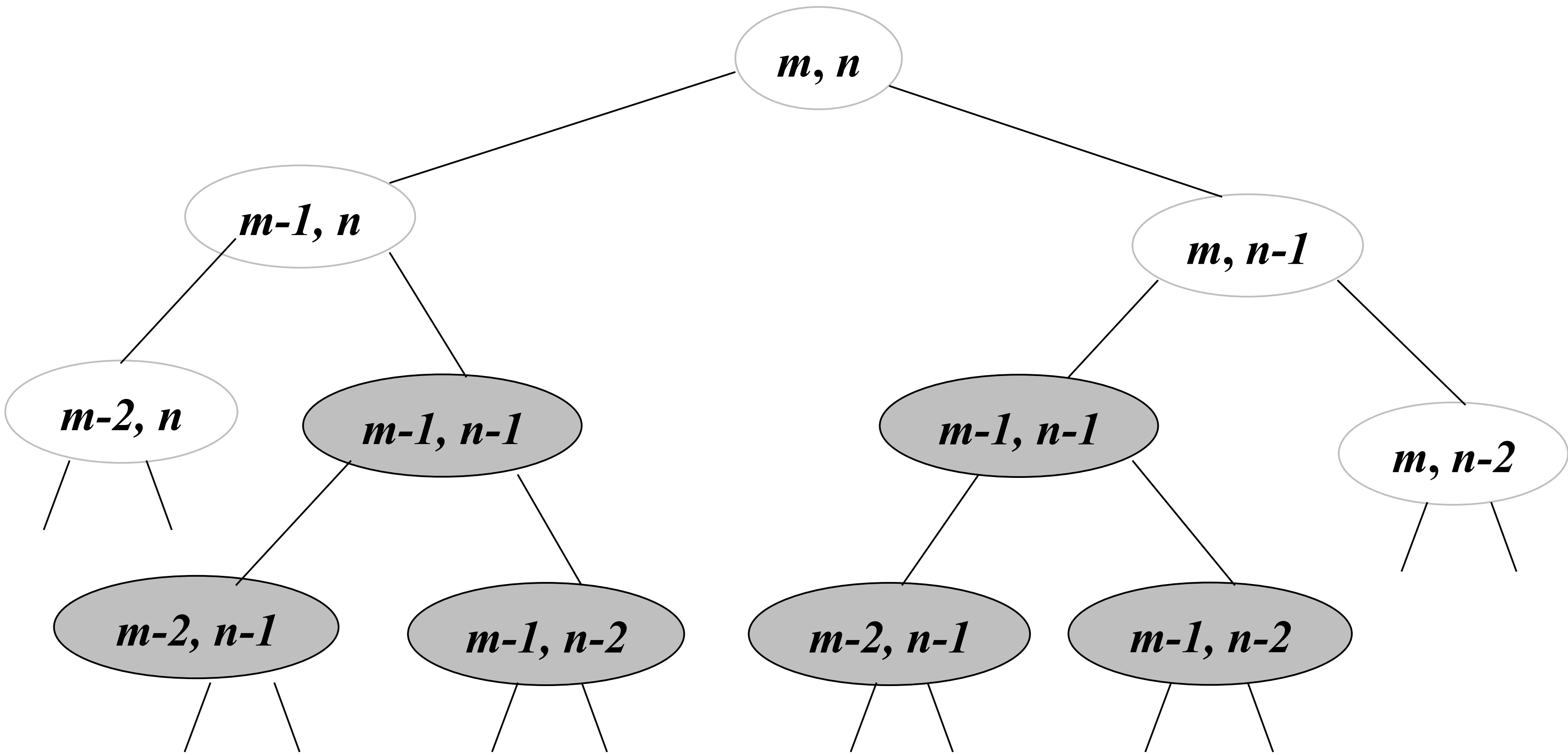
LCS (X, Y, i, j)
  if $(i \neq 0$ and $j \neq 0)$ **then**
    if $X[i] = Y[j]$
      then $c[i, j] \leftarrow \text{LCS}(X, Y, i\text{-}1, j\text{-}1) + 1$

    else $c[i, j] \leftarrow max\ \{\ \text{LCS}(X, Y, i\text{-}1, j), \text{LCS}(X, Y, i, j\text{-}1)\ \}$

Complexity:

Exponential because we have to solve sub-problems that were already solved.

Recursion Tree



Notice same subproblems are calculated → Not efficient!

Recursive solution – analysis

- The height of the tree is $O(m+n)$

- There are only $O(m * n)$ number of distinct LCS subproblems for two strings of length m and n

→ $m * n$ different $c[i, j]$

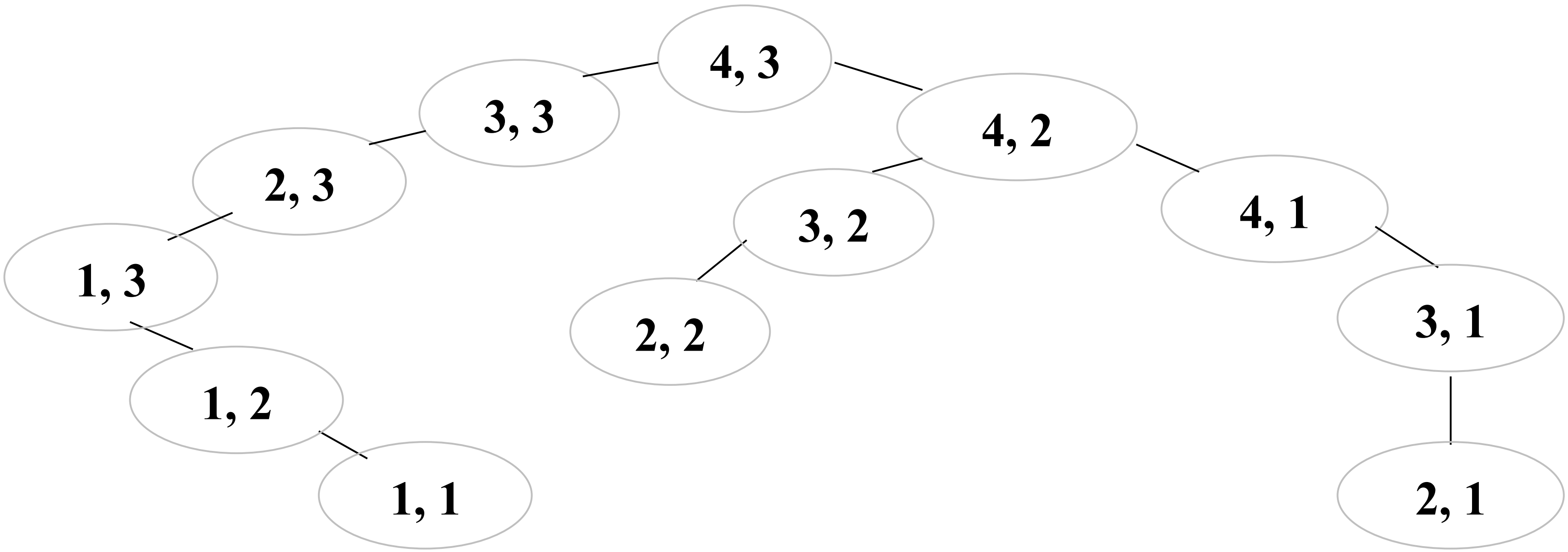
$i = m, m\text{-}1, m\text{-}2, \ldots, 2, 1$

$j = n, n\text{-}1, n\text{-}2, \ldots, 2, 1$

→ **we can use dynamic programming to compute the solutions bottom up**

Curtin University of Technology

Recursion Tree (Example)



For *m* = 4 and *n* = 3, there are 4 * 3 = 12 distinct LCS subproblems.

Computing the length of an LCS

LCS-Length (*X*,*Y*)

```
m = length[X]
n = length[Y]
for i = 1 to m
        do c[i, 0] = 0
for j = 1 to n
        do c[0, j] = 0
for i = 1 to m
     do for j = 1 to n
          if x_i = y_j
             then c[i, j] = c[i-1, j-1] +1
                  b[i,j] = "\"
          else if c[i-1, j] ≥ c[i, j-1]
             then c[i, j] = c[i-1, j]
                  b[i, j] = "↑"
          else c[i, j] = c[i, j-1]
                  b[i, j] = "←"
return c and b
```

Time = $O(mn)$ → constant work per table entry

Space = $O(mn)$

Example

X = (A, B, C, B, D, A, B), Y = (B, D, C, A, B, A)

	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

Table *c*

Example (cont.)

X = (A, B, C, B, D, A, B), Y = (B, D, C, A, B, A)

i		y_j	**B**	**D**	**C**	**A**	**B**	**A**
0	x_i	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	\	↑	↑	↑	\	↑

Table *b*

Curtin University of Technology

Constructing an LCS

From table *b*

- Use Print-LCS (b, X, m, n)
- Complexity: $O(m+n)$ → at least one of i and j is decremented in each stage of the recursion.

From table *c* (without table *b*)

- Each $c[i, j]$ entry depends on entries: $c[i\text{-}1, j\text{-}1]$, $c[i\text{-}1, j]$, and $c[i, j\text{-}1]$
- Given a $c[i, j]$, we can determine which of the three values was used to compute $c[i, j]$ → O(1)
- Complexity: $O(m+n)$

Curtin
University of Technology

Constructing an LCS

Print-LCS(*b*, *X*, *i*, *j*)

if $i = 0$ or $j = 0$

    then return

if $b[i,j]$ = "\"

    then Print-LCS(b, X, i-1, j-1)

        print x_i

else if $b[i, j]$ = "↑"

    then Print-LCS(b, X, i-1, j)

else Print-LCS(b, X, i, j-1)

Memoization

- **Memo**ization is a variation of dynamic programming approach while maintaining top-down strategy

- The idea is to *memoize* the natural, but inefficient, recursive algorithm

- Like in an ordinary dynamic programming, this approach maintains a table with sub-problem solutions
  - but the control structure for filling in the table is like the recursive algorithm

Curtin University of Technology

Memoization (cont.)

- Each table entry initially contains a **special value** to indicate that the entry has yet to be filled in
  - E.g., "-1" if each valid entry is a positive value
- When the sub-problem is first encountered during the execution of the recursive algorithm, its solution is computed and stored in the table
- Each subsequent time the sub-problem is encountered, the value stored in the table is looked up and returned

Curtin University of Technology

Memoization – LCS example

LCS (X, Y, i, j)

if ($i \neq 0$ and $j \neq 0$) **then**

if $c[i, j]$ = NIL

then if $X[i] = Y[j]$

then $c[i, j]$ = LCS(X, Y, i-1, j-1) + 1

else $c[i, j] = max$ { LCS(X, Y, i-1, j),

LCS(X, Y, i, j-1) }

Call the function recursively from LCS (X, Y, m, n)

Time = $O(mn)$ → constant work per table entry

Space = $O(mn)$

Curtin University of Technology

Matrix Chain Multiplication - Review

Let $A_{i..j} = A_i \times A_{i+1} \times \ldots \times A_j$

An optimal parenthesization splits the product
$A_{i..j} = (A_i \times A_{i+1} \times \ldots \times A_k) \times (A_{k+1} \times A_{k+2} \times \ldots \times A_j)$ for $1 \leq k < n$

The total cost of computing $A_{i..j}$
= cost of computing $(A_i \times A_{i+1} \times \ldots \times A_k)$
+ cost of computing $(A_{k+1} \times A_{k+2} \times \ldots \times A_j)$
+ cost of multiplying the matrices $A_{i..k}$ and $A_{k+1..j}$

$A_{i..k}$ must also be optimal if we want $A_{i..j}$ to be optimal. If $A_{i..k}$ is not optimal then $A_{i..j}$ is not optimal. Similarly $A_{k+1..j}$ must also be optimal.

Curtin University of Technology

Recursive Solution

We'll define the value of an optimal solution recursively in terms of the optimal solutions to sub-problems.

$m[i, j]$ = minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$

$m[1, n]$ = minimum number of scalar multiplications needed to compute the matrix $A_{1..n}$.

If $i = j$; the chain consists of just one matrix

$A_{i..i} = A_i$; no scalar multiplications

$m[i,i] = 0$ for $i = 1, 2, \ldots, n$.

$m[i, j]$ = minimum cost of computing the sub-products $A_{i..k}$ and $A_{k+1..j}$ + cost of multiplying these two matrices

Multiplying $A_{i..k}$ and $A_{k+1....j}$ takes $p_{i-1}p_kp_j$ scalar multiplications

Thus,

$$\boldsymbol{m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j \text{ for } i \leq k < j}$$

Curtin University of Technology

The optimal parenthesization must use one of these values for k, and we need to check them to find the best solution.

Therefore,

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

Let $\boldsymbol{s[i, j]}$ be the value of k at which we can split the product $A_i \times A_{i+1} \times \ldots \times A_j$

To obtain the optimal parenthesization, s[*i*, *j*] equals a value of *k* such that

$$\boldsymbol{m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_k p_j \text{ for } i \le k < j}$$

Curtin University of Technology

Memoization – Matrix Chain

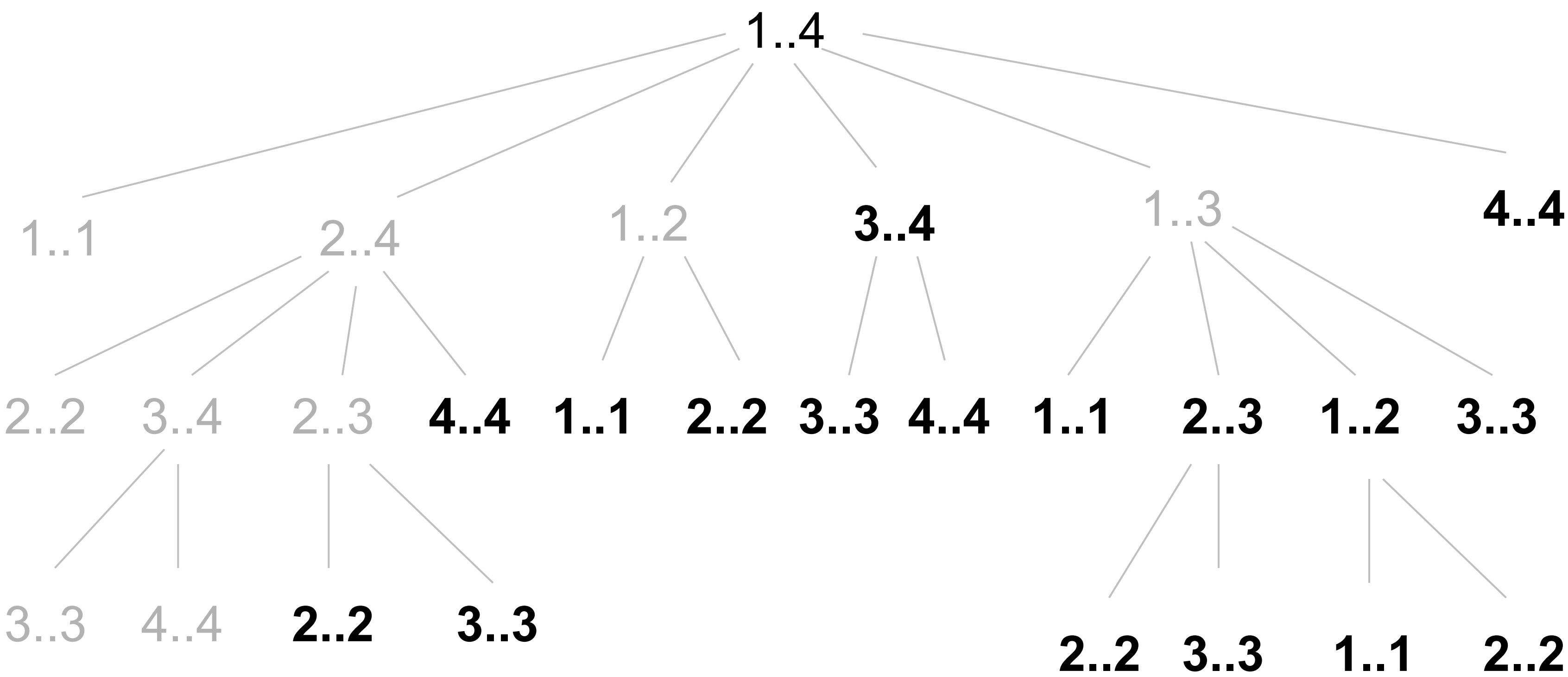
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{otherwise} \end{cases}$$

Curtin University of Technology

Inefficient Recursive matrix chain

```
Recursive-Matrix-Chain (p, i, j)
if i = j
    then return 0
m[i, j] = ∞
for k = i to j-1
    do q = Recursive-Matrix-Chain (p, i, k)
             + Recursive-Matrix-Chain (p, k+1, j)
             + p_{i-1} p_k p_j
        if q < m[i, j]
            then m[i, j] = q
return m[i, j]
```

Recursion Tree



Memoization version

```
Memoized-Matrix-Chain (p)
n ← length[p] - 1
 for i ← 1 to n
        do for j ← i to n
               m[i, j] ← ∞
 return Lookup-Chain (p, 1, n)
```

Memoization version (cont.)

Lookup-Chain (p, i, j)

if $m[i, j] < \infty$

  then return $m[i, j]$

if $i = j$

  then $m[i, j] = 0$

else for $k \leftarrow i$ to $j - 1$

  do $q \leftarrow$ Lookup-Chain (p, i, k)

    $+$ Lookup-Chain $(p, k+1, j)$

    $+ \, p_{i-1} p_k p_j$

    if $q < m[i, j]$

      then $m[i, j] = q$

return $m[i, j]$

0/1 Knapsack - Review

- Either put the first item in the bag then

$$\text{maximise} \quad \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c - w_1$$

- Or don't put first item in and

$$\text{maximise} \quad \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=2}^{n} w_i x_i \le c$$

Do both and take the maximum

Recursion

- Define $P(i, k)$ to be the maximum profit possible using items $i \dots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ & \\ & \end{cases}$$

- Selecting n is not possible

Curtin University of Technology

Recursion (cont.)

- Define $P(i,k)$ to be the maximum profit possible using items $i \dots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \,\&\, w_n > k \\ p_n & i = n \,\&\, w_n \le k \\ & \\ & \end{cases}$$

- Selecting n is possible

Curtin University of Technology

Recursion (cont.)

- Define $P(i, k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \;\&\; w_n > k \\ p_n & i = n \;\&\; w_n \le k \\ P(i+1,k) & i < n \;\&\; w_i > k \\ & \end{cases}$$

- Selecting i is not possible

Recursion (cont.)

- Define $P(i, k)$ to be the maximum profit possible using items $i \ldots n$ and capacity k

$$P(i,k) = \begin{cases} 0 & i = n \;\&\; w_n > k \\ p_n & i = n \;\&\; w_n \le k \\ P(i+1,k) & i < n \;\&\; w_i > k \\ \max(P(i+1,k),\, p_i + P(i+1,k-w_i)) & i < n \;\&\; w_i \le k \end{cases}$$

- Selecting i is possible, but do we want to?

Curtin University of Technology

Recursive solution

KNAPSACK-RECURSE (*i*, *k*)

if ($i = n$) and ($w_n > k$) **then return** 0

if ($i = n$) and ($w_n <= k$) **then return** p_n

if ($i < n$) and ($w_i > k$) **then**

  return KNAPSACK-RECURSE (*i*+1, *k*)

p_n $i = n$ & $w_n \le k$

if ($i < n$) and ($w_i <= k$) **then**

  ***x* := KNAPSACK-RECURSE (*i*+1, *k*)**

  ***y* := KNAPSACK-RECURSE (*i*+1, *k*-w_i)** + p_i

  return max (*x*, *y*)

0/1 Knapsack - Recursive

KNAPSACK-RECURSE (*i*, *k*)
  if ($i = n$) **then**
    if ($w_n > k$) **then**
      return 0
    else
      return p_n

  if ($w_i > k$) **then**
    return KNAPSACK-RECURSE (i+1, k)
  else
    x := KNAPSACK-RECURSE (i+1, k)
    y := KNAPSACK-RECURSE (i+1, k-w_i) + p_i
    return max (x, y)

Curtin University of Technology

0/1 Knapsack - Memoization

```
KNAPSACK-RECURSE (i, k)
  if P[k] != UNKNOWN
      return P[k]
  if (i = n)
      if (w_n > k)
          return 0
      else
          return p_n
  if (w_i > k)
      return KNAPSACK-RECURSE (i+1, k)
  else
      x ← KNAPSACK-RECURSE (i+1, k)
      y ← KNAPSACK-RECURSE (i+1, k-w_i)
      P[k] ← max (x, y)
      return max (x, y)
```

Curtin University of Technology

Bottom-up vs Top-down DP

- Top-down is better if some sub-problems in the sub-problem space need not be solved at all
  - memoization helps by solving only those sub-problems that are definitely required.

- Bottom-up dynamic programming is better by a constant factor if all sub-problems must be solved at least once
  - no overhead for recursion, and less overhead for maintaining the table.
  - can further reduce time or space in some problems with regular pattern accesses to the table

Curtin University of Technology

The End