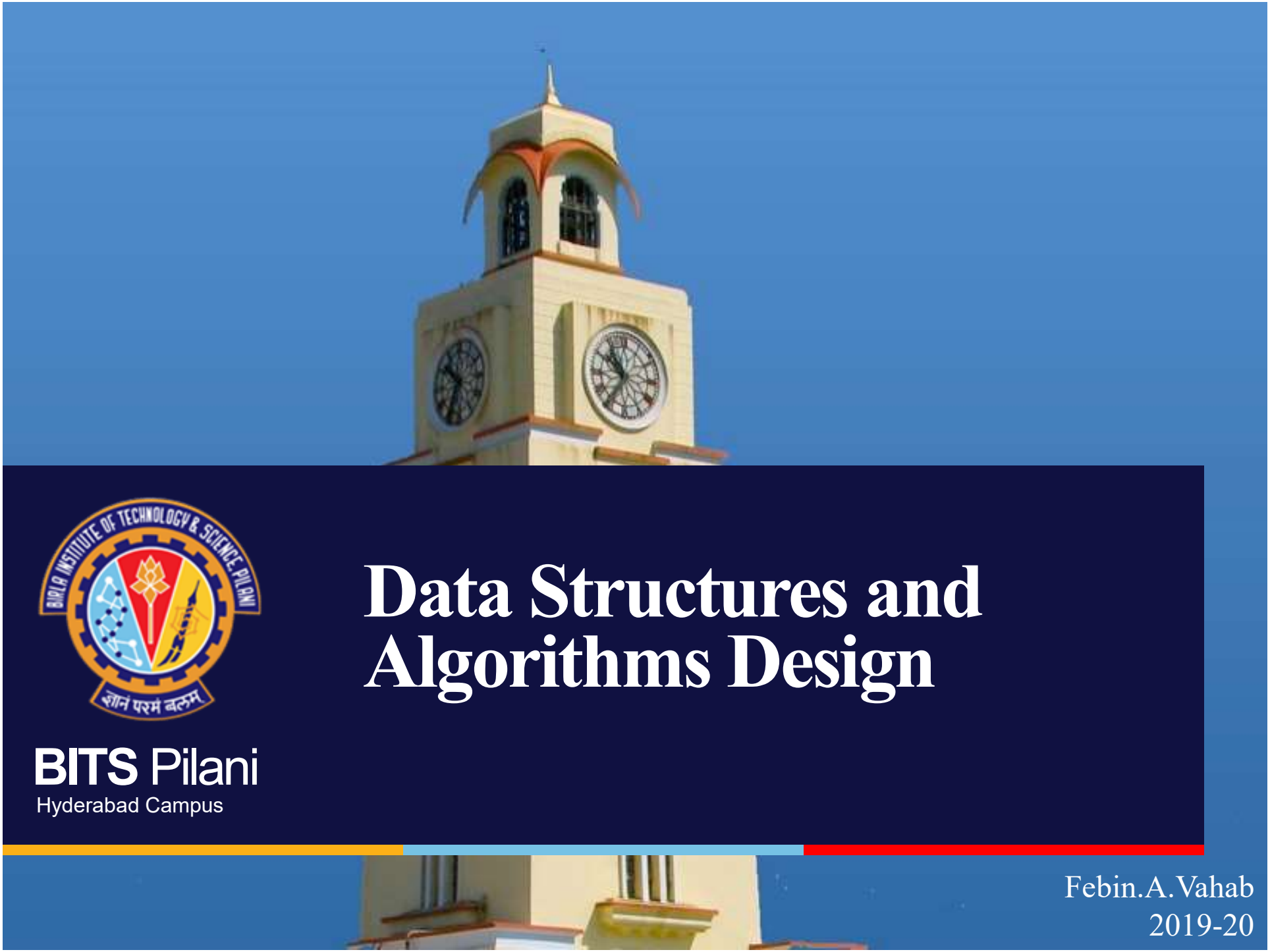


Data Structures and Algorithms Design

BITS Pilani
Hyderabad Campus

Febin.A.Vahab
2019-20

ONLINE SESSION 13 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
13	**Dynamic Programming - Design Principles and Strategy, Matrix Chain Product Problem, 0/1 Knapsack Problem, All-pairs Shortest Path Problem**	**T1: 5.3, 7.2**

Dynamic Programming

- Invented by a prominent U.S. mathematician, Richard Bellman
- The word “programming” in the name of this technique stands for “planning” and does not refer to computer programming.

Dynamic Programming

- Dynamic programming is a technique for solving problems with overlapping subproblems.
- Typically, given problem's solution can be related to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming

innovate achieve lead

- A straightforward application of dynamic programming can be interpreted as a special variety of space-for-time trade-off.
- Optimisation of plain recursion.

Dynamic Programming-Example

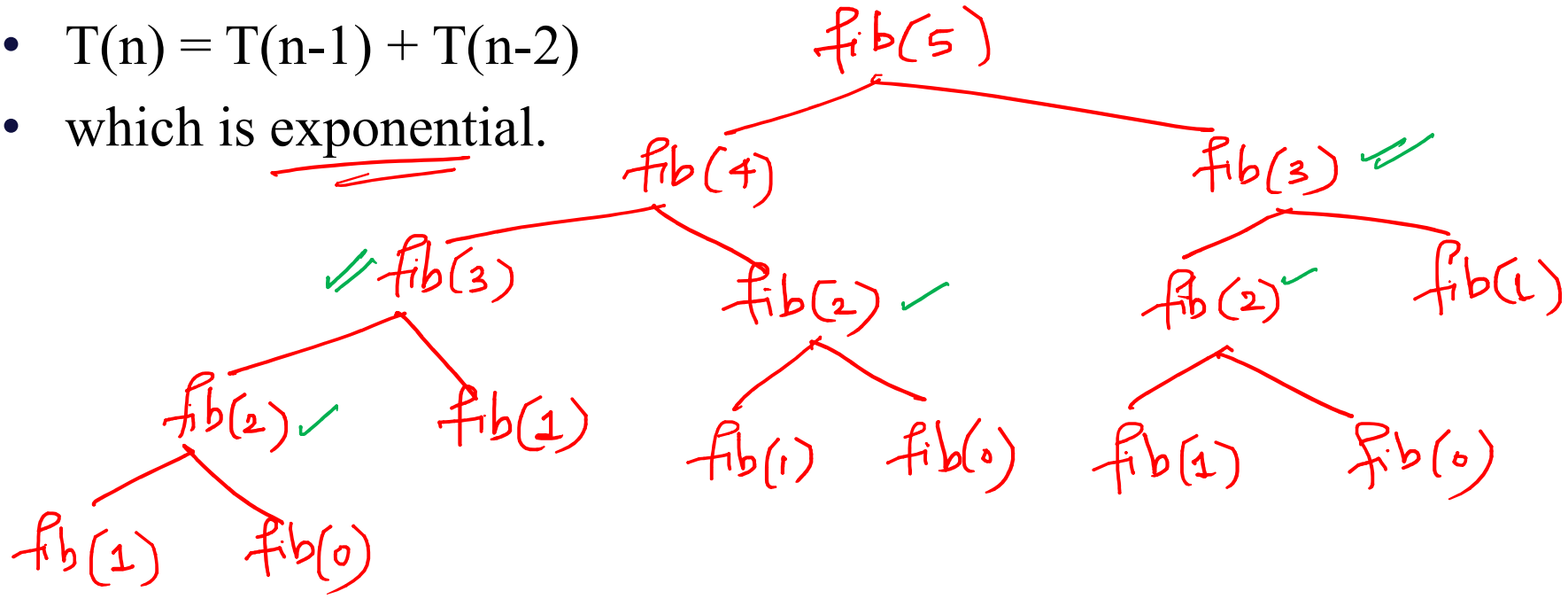
- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…
- F(n) = F(n-1) + F(n-2), F0 = 0 and F1 = 1

```
int fib(int n)
{
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

$T(n) = T(n-1) + T(n-2)$

Dynamic Programming-Example

- Time complexity:
- T(n) = T(n-1) + T(n-2)
- which is exponential.



Dynamic Programming-Properties

Overlapping Sub-problems:

- Sub-problems needs to be solved again and again.
- In recursion we solve those problems every time and in dynamic programming we solve these sub problems only once and store it for future use

Optimal Substructure:

- A problem can be solved by using the solutions of the sub problems

Dynamic Programming-Example


```
Algorithm DynamicFibonacci(n)
{
  f[0]=0,f[1]=1
 for(i =2;i<=n;i++)
 f[i]=f[i-1]+f[i-2]
return f[n];
}
```

Time complexity?? O(n)

Dynamic Programming-Example

- A man put a pair of rabbits in a place surrounded by a wall. How many pairs of rabbits will be there in a year if the initial pair of rabbits (male and female) are newborn and all rabbit pairs are not fertile during their first month of life but thereafter give birth to one new male/female pair at the end of every month?

Dynamic Programming

- The circumstances and restrictions are not realistic.
- Still, this isn't THAT unrealistic a situation in the short term.

Dynamic Programming

- Similar to Fibonacci problem.
- To solve Fibonacci's problem, we'll let f(n) be the number of pairs during month n.
- By convention, f(0) = 0. f(1) = 1 for our new first pair.
- f(2) = 1 as well, as conception just occurred.
- The new pair is born at the end of month 2, so during month 3, f(3) = 2.
- Only the initial pair produces offspring in month 3, so f(4) = 3.

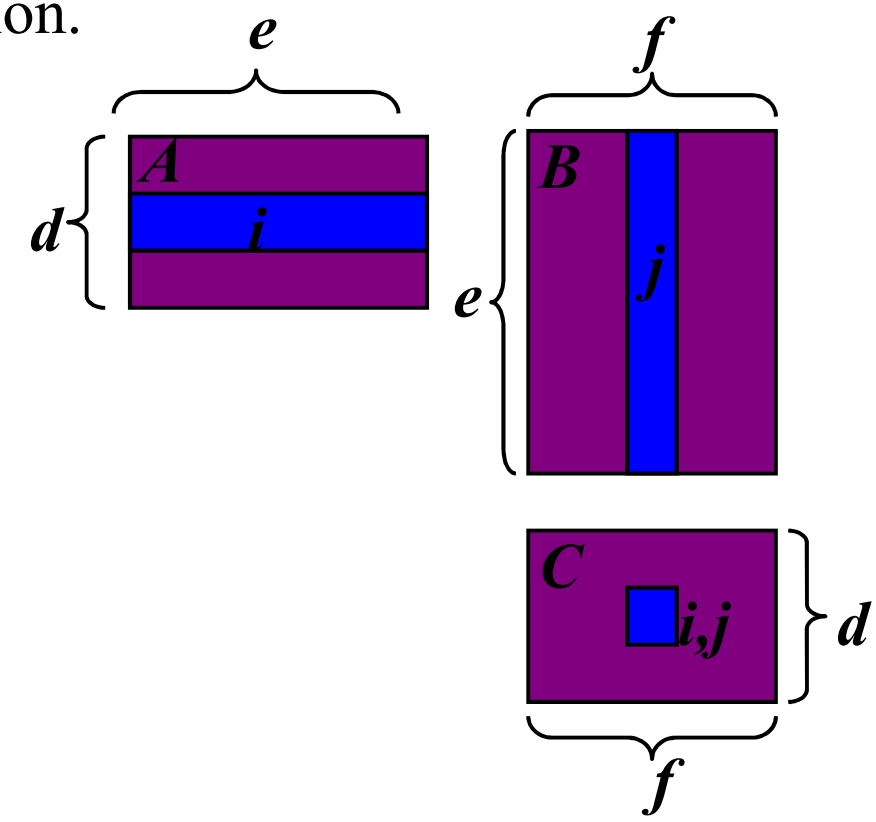
Dynamic Programming

- In month 4, the initial pair and the month 2 pair breed, so f(5) = 5. We can proceed this way, presenting the results in a table. At the end of a year, Fibonacci has 144 pairs of rabbits.

Month	0	1	2	3	4	5	6	7	8	9	10	11	12
Pairs	0	1	1	2	3	5	8	13	21	34	55	89	144

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - $O(d \cdot e \cdot f)$ time



$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

Matrix Multiplication

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

$\longrightarrow$ A = 2 x 3 matrix

B = 3 x 2 matrix

$$C = \begin{bmatrix} 7+18+33 & 8+20+36 \\ 28+45+66 & 32+50+72 \end{bmatrix}$$

$\longrightarrow$ No: of multiplications

$= \underline{12}$

$= \underline{2 \times 3 \times 2}$

$$C = \begin{bmatrix} 58 & 64 \\ 139 & 156 \end{bmatrix}$$

$\longrightarrow$ C = 2 x 2 matrix

i.e if A = m x n, B = n x q Then A x B = m x q

No: of ops = m x n x q

Matrix Chain-Products

- Matrix multiplication is associative
- $B \cdot (C \cdot D) = (B \cdot C) \cdot D$.
- Thus, we can parenthesize the expression for multiplication any way we wish and we will end up with the same answer.
- **Number of primitive (that is, scalar) multiplications in each parenthesization, however, might not be the same.**

Matrix Chain-Products

- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

BxC = 3x100x5 = 1500 ops 3x5 matrix

(BC x D) = (3x5)(5x5)
= 75 ops
3x5

Order of the resultant matrix ?

CxD = (100x5)(5x5) = 100x5x5 = 2500 ops 100x5

Bx CD = (3x100)(100x5) = 3x100x5 = 1500 ops
3x5

Matrix Chain-Products

innovate achieve lead

- **Matrix Chain-Product:**
- Suppose we are given a collection of n two-dimensional matrices for which we wish to compute the product
  - Compute $\mathbf{A=A_1*\ldots*A_n}$
  - A_i is $d_{i-1} \times d_i$ matrix, for $i = 1, 2, \ldots, n$.
  - ie. Input A[]={10,20,30,40,50}
  - A1 is10x20 matrix,A2=20X30 matrix,A3 is 30X40 matrix and A4 is 40X50 matrix.
  - Problem: How to parenthesize?

Matrix Chain Product

- Input A[]={10,20,30,40,50}
- A1=10 X 20
- A2= 20 X 30
- A3=30 X 40
- A4= 40 X 50
- A_i is $d_{i-1} \times d_i$ Matrix

- $A_{i..j} = A_{i..k} \times A_{k+1..j}$

- $A_{1..4} = (A_{1..2}) \times (A_{3..4})$

$d_0\ d_1\ d_2\ d_3\ d_4$

$d_0 \times d_1$

$d_1 \times d_2$

$d_2 \times d_3$

$d_3 \times d_4$

$k = 2$

$A1 \times A2 \times A3 \times A4$

$A_{1..2} = 10 \times 30$

$A_{3..4} = 30 \times 50$

$(A_{1..2})(A_{3..4}) = 10 \times 50$

$(A_{i..k})(A_{k+1..j}) = 10 \times 30 \times 50$

$= d_{i-1} \times d_k \times d_j$

Matrix Chain-Products

- The Matrix Chain-Products problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed.
- The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Matrix Chain-Products-Enumeration Approach

- **Matrix Chain-Product Algorithm:**
  - Try all possible ways to parenthesize $A=A_1*\ldots*A_n$
  - Calculate number of ops for each one
  - Pick the one that is best

$$
\begin{aligned}
A_1A_2A_3A_4 &= (A_1A_2)(A_3A_4) \\
&= A_1(A_2(A_3A_4)) = A_1((A_2A_3)A_4) \\
&= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))(A_4)
\end{aligned}
$$

Matrix Chain-Products-Enumeration Approach

- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes.
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Number of Binary trees with n nodes

- Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = (2n)!/(n+1)!*n!
- For n = 0,1, 2, 3, … values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …. So are numbers of Binary Search Trees.
- Total number of possible Binary Trees with n different keys (countBT(n)) = countBST(n) * n!

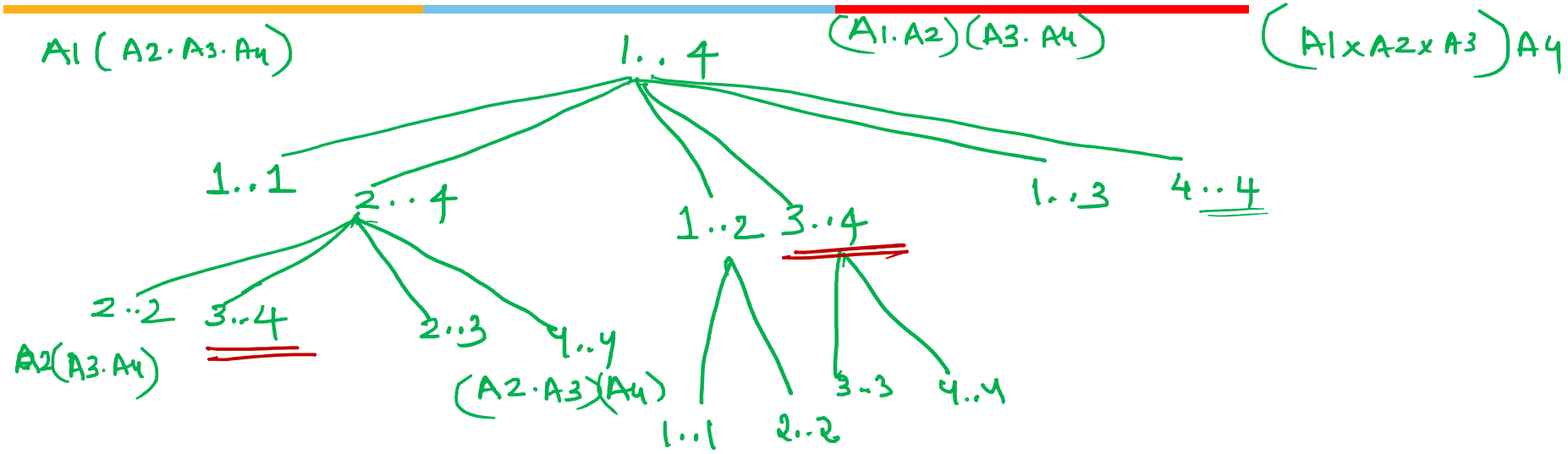
Matrix Chain-Products-Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Matrix Chain-Products-Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

Matrix Chain-Products-"Recursive" Approach



Example-Dynamic Programming

INPUT, A[] = $\{\underset{d_0}{5}, \underset{d_1}{4}, \underset{d_2}{6}, \underset{d_3}{2}, \underset{d_4}{7}\}$

ie There are 4 Matrices.

A1 = 5 x 4
A2 = 4 x 6
A3 = 6 x 2
A4 = 2 x 7

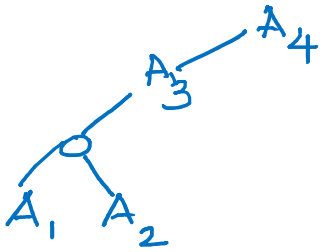
$= d_{i-1} \times d_i$

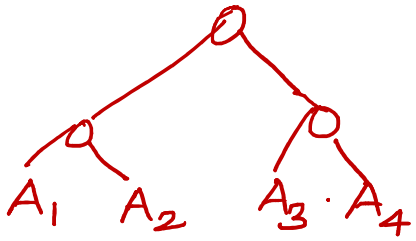
Find the minimum number of multiplications needed to compute $A_{1..4}$ ie [A1 x A2 x A3 x A4]

Possibilities

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

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





Example-Dynamic Programming

Step 1

Consider the smallest subproblem.

No: of matrix = 1

m[1,1] = number of multiplications needed to compute $A_{1..1}$ i.e A_1

= 0

By

m[2,2] = 0

m[3,3] = 0

m[4,4] = 0

Matrix-M

	1	2	3	4
1	0			
2		0		
3			0	
4				0

S

Example-Dynamic Programming

innovate achieve lead

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$
$A_4 = 2 \times 7$

Step - 2

Multiply 2 matrices

$m[1,2]$ = number of multiplications needed to compute $A_{1..2}$ i.e $A_1 \times A_2$

$= (5 \times 4)(4 \times 6)$ (A_1 A_2) } order of the resultant matrix $= 5 \times 6$

$= 5 \times 4 \times 6 = 120$

$m[2,3] = A_2 \times A_3$

$= (4 \times 6)(6 \times 2)$ } order of resultant matrix $= 4 \times 2$

$= 4 \times 6 \times 2 = 48$

$m[3,4] = A_3 \times A_4$

$= (6 \times 2)(2 \times 7)$ } order of resultant matrix $= 6 \times 7$

$= 6 \times 2 \times 7 = 84$

	1	2	3	4
1	0	120 (5x6)		
2		0	48 (4x2)	
3			0	84 (6x7)
4				0

k

	1	2	3	4
1		1		
2			2	
3				3
4				

Example-Dynamic Programming

A1 = 5x4
A2 = 4x6
A3 = 6x2
A4 = 2x7

Step 3: Multiply 3 Matrices

m[1,3] = No: of multiplications needed to compute $A_{1..3}$ i.e A1 x A2 x A3

2 possibilities

(i) $A_1 (A_2 \cdot A_3)$

(ii) $(A_1 \cdot A_2) A_3$

(i) $A_1 (A_2 \cdot A_3)$ A1 (x) $(A_2 \cdot A_3)$

- No: of multiplications to perform $A_{1..1}$ +
- No: of multiplications to perform $A_{2..3}$ +
- No: of multiplications to perform A1 x $A_{2..3}$

= m[1,1] + m[2,3] + (5x4x2)

= 0 + 48 + 40 = 88

	1	2	3	4
1	0	120 (5x6)		
2		0	48 (4x2)	
3			0	84 (6x7)
4				0

Matrix, m

	1	2	3	4
1		1		
2			2	
3				3
4				

Example-Dynamic Programming

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$
$A_4 = 2 \times 7$

(ii) $(A_1 \cdot A_2) A_3$ ✓ $(A1 \cdot A2) * A3$

$m[1,2] + m[3,3] + (5 \times 6 \times 2)$

$= 120 + 0 + 60 = 180$

Order of the resultant matrix $= 5 \times 2$

$\min(88, 180) = 88$ ✓

$\therefore m[1,3] = 88$ ✓

	1	2	3	4
1	0	120 $_{5\times6}$	88 $_{5\times2}$	
2		0	48 $_{4\times2}$	
3			0	84 $_{6\times7}$
4				0

Step 3 : continues

$m[2,4]$

2 possibilities

(i) $A_2 (A_3 \cdot A_4)$

(ii) $(A_2 \cdot A_3) A_4$ //

	1	2	3	4
1		1	1	
2			2	
3				3
4				

Example-Dynamic Programming

$A_1 = 5 \times 4$
$A_2 = 4 \times 6$
$A_3 = 6 \times 2$
$A_4 = 2 \times 7$

(i) $A_2(A_3 \cdot A_4)$

$m[2,2] + m[3,4] + [4 \times 6 \times 7]$

$= 0 + 84 + 168 = \underline{252}$ }

(ii) $(A_2 \cdot A_3)\, A_4$

$m[2,3] + m[4,4] + (4 \times 2 \times 7)$

$= 48 + 0 + 56 = \underline{104}$ }

$\min(252, 104) = \underline{104}$

$\therefore m[2,4] = \underline{104}$

} order of resultant matrix $= \underline{4 \times 7}$

	1	2	3	4
1	$0_{5\times4}$	$120_{5\times6}$	$88_{5\times2}$	
2		$0_{4\times6}$	$48_{4\times2}$	$104_{4\times7}$
3			$0_{6\times2}$	$84_{6\times7}$
4				$0_{2\times7}$

	1	2	3	4
1		1	1	
2			2	3
3				3
4				

Example-Dynamic Programming

innovate achieve lead

Step 4: Multiply 4 matrices

Use the formula

$(A1)(A2\ A3)\ A4$

$m[1,4]$

$= \min \begin{bmatrix} m[1,1] + m[2,4] + [5 \times 4 \times 7]), \\ m[1,2] + m[3,4] + [5 \times 6 \times 7]), \\ m[1,3] + m[4,4] + [5 \times 2 \times 7]) \end{bmatrix}$

$= \min [(0 + 104 + 140), (120 + 84 + 210), (88 + 0 + 70)]$

$= \min (244, 414, 158)$

$= 158$

	1	2	3	4
1	$0_{5\times4}$	$120_{5\times6}$	$88_{5\times2}$	$158_{5\times7}$
2		$0_{4\times6}$	$48_{4\times2}$	$104_{4\times7}$
3			$0_{6\times2}$	$84_{6\times7}$
4				$0_{2\times7}$

m

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

General formula $m[i,j] = \{m[i,k] + m[k,j] + d_{i-1}\ d_k \times d_j\}$

Example-Dynamic Programming

k	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

((A1)((A2)(A3))A4

Matrix Chain-Products-Dynamic Programming

- Define **subproblems**:
  - Find the best parenthesization of $A_i * A_{i+1} * \ldots * A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{1,n}$.

Matrix Chain-Products-Dynamic Programming

- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_1*\ldots*A_i)*(A_{i+1}*\ldots*A_n)$.
  - Then the optimal solution $N_{1,n}$ is the sum of two optimal subproblems, $N_{1,i}$ and $N_{i+1,n}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

Matrix Chain-Products- Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_{i-1} \times d_i$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_{i-1} d_k d_j\}$$

$$N_{i,i} = 0$$

Matrix Chain Products-Dynamic Programming Algorithm

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

for $i \leftarrow 1$ **to** $n - 1$ **do**

$N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ **to** $n - 1$ **do**

{ $b = j - i$ is the length of the problem }

for $i \leftarrow 0$ **to** $n - b - 1$ **do**

$j \leftarrow i + b$

$N_{i,j} \leftarrow +\infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$

return $N_{0,n-1}$

Matrix Chain Products-Dynamic Programming Algorithm


```
Matrix-Chain(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 m and s; (Optimum in m[1, n])
}
```

Matrix Chain-Products-Dynamic Programming Algorithm

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry.

Matrix Chain Products-Dynamic Programming Algorithm

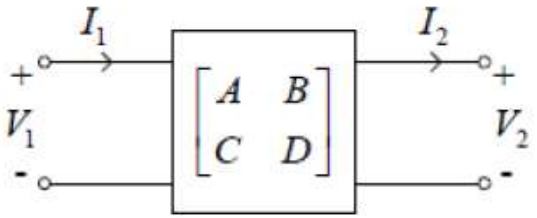
- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do problems of "length" 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

Matrix Chain Products-Dynamic Programming –Example explained

MCP-Example

Matrix Chain Products-Applications

- **Perspective projections**, which is the foundation for 3D animation-Orthographic projection (sometimes orthogonal projection) is a means of representing three-dimensional objects in two dimensions.
- Minimum and Maximum values of an expression with * and +
- **Network analysis (electrical circuits)- Network analysis** is the process of finding the voltages across, and the currents through, every component in the network. For example, when two or more N-port networks are connected in cascade, the combined network is the product on the individual ABCD matrices
- Used extensively in NLP and Machine learning: Principal Component Analysis and Singular Value Decomposition



Read about "Word to Vectors—Natural Language Processing"

Matrix Chain Products-Applications

The ABCD Matrix is defined as:

$$\begin{bmatrix} V_1 \\ I_1 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} V_2 \\ I_2 \end{bmatrix}$$

For which:

$$A = \left.\frac{V_1}{V_2}\right|_{I_2=0}, \quad B = \left.\frac{V_1}{I_2}\right|_{V_2=0}$$

$$C = \left.\frac{I_1}{V_2}\right|_{I_2=0}, \quad D = \left.\frac{I_1}{I_2}\right|_{V_2=0}$$

"The usefulness of the ABCD matrix is that cascaded two port networks can be characterized by simply multiplying their ABCD matrices"

Matrix Chain Products-Applications

- Used extensively in NLP and Machine learning

Read about "Word to Vectors—Natural Language Processing"

- **sentence**=" Word Embeddings are Word converted into numbers "
- A *word* in this **sentence** may be "Embeddings" or "numbers " etc.
- A *dictionary* may be the list of all unique words in the **sentence.**
- So,a dictionary may look like ['Word', 'Embeddings', 'are', 'Converted', 'into', 'numbers']
- A ve*ctor* representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else.
- The vector representation of "numbers" in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

Matrix Chain Products-Applications

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =['He','She','lazy','boy','Neeraj','person']

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

	He	She	lazy	boy	Neeraj	person
D1	1	1	2	1	0	0
D2	0	0	1	0	1	1

Matrix Chain Products-Applications

Co-occurrence matrix.

Corpus = He is not lazy. He is intelligent. He is smart.

	He	is	not	lazy	intelligent	smart
He	0	4	2	1	2	1
is	4	0	1	2	2	1
not	2	1	0	1	0	0
lazy	1	2	1	0	0	0
intelligent	2	2	0	0	0	0
smart	1	1	0	0	0	0

Principal Component Analysis and Singular Value Decomposition

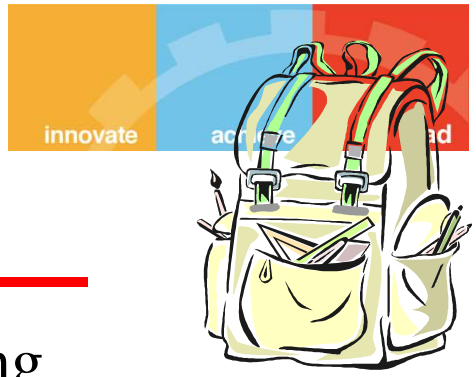
The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

The 0/1 Knapsack Problem

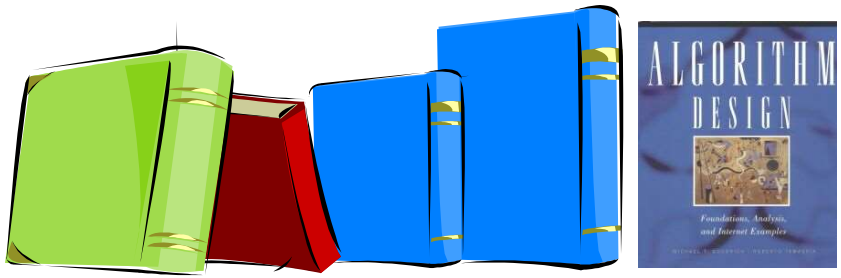
- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - p_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- This problem is called **a "0- 1 " problem**, because each item must be entirely accepted or rejected.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum p_i x_i$
  - Constraint: $\sum_{i \varepsilon T} w_i x_i \leq W$

Example



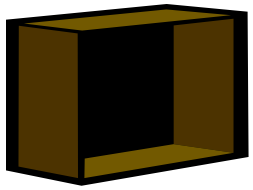
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 kg	2 kg	2 kg	6 kg	2 kg
Benefit:	$20	$3	$6	$25	$80

"knapsack"



box of width 9 in

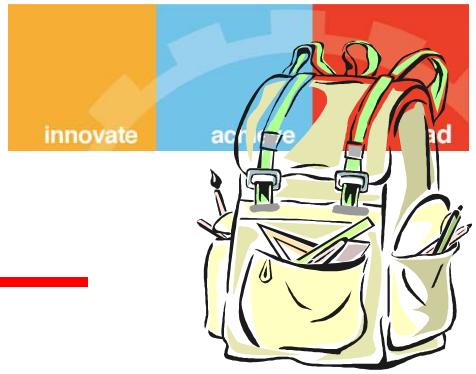
Solution:
- item 5 ($80, 2 kg)
- item 3 ($6, 2 kg)
- item 1 ($20, 4 kg)

Example

Example

Example

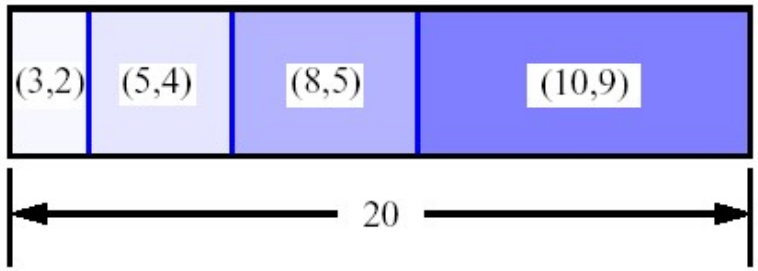
A 0/1 Knapsack Algorithm, First Attempt



- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:

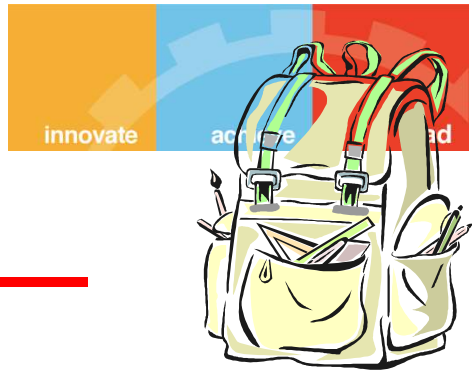
Best for S_5:



A 0/1 Knapsack Algorithm, First Attempt

- Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots I_k\}$.
- The solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

A 0/1 Knapsack Algorithm, Second Attempt

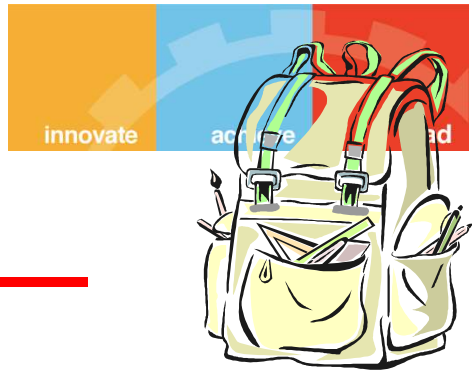


- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item k
  -

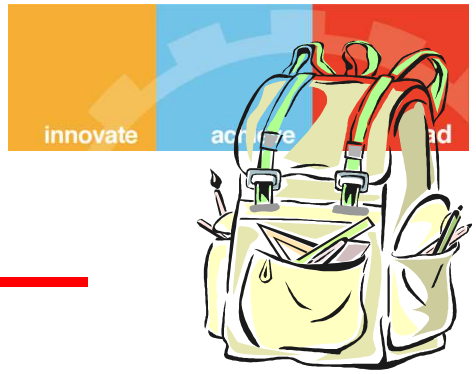
A 0/1 Knapsack Algorithm, Second Attempt



- **Case 1**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight w, if item k weighs greater than w.

- Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

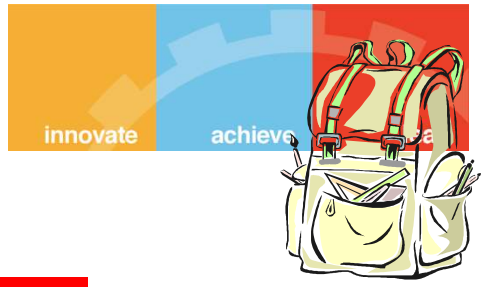
A 0/1 Knapsack Algorithm, Second Attempt



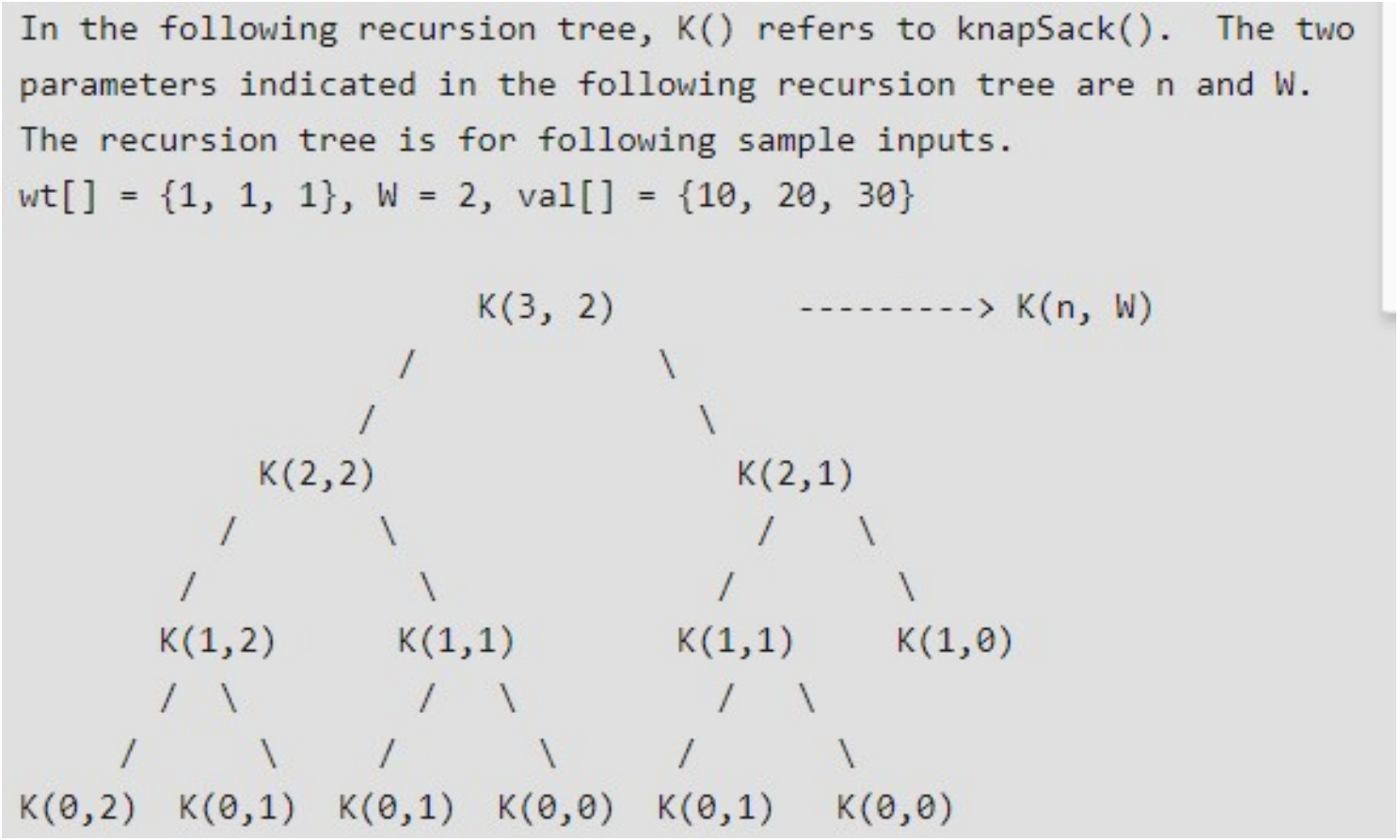
- **Case 2**
- The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight $w-w_k$, plus item k.

- You need to compare the values of knapsacks in both case 1 and 2 and take the maximal one.

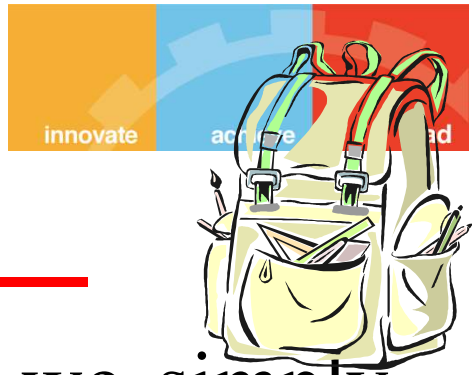
A 0/1 Knapsack Algorithm, Second Attempt



- Recursively, we will STILL have an $O(2^n)$ algorithm.

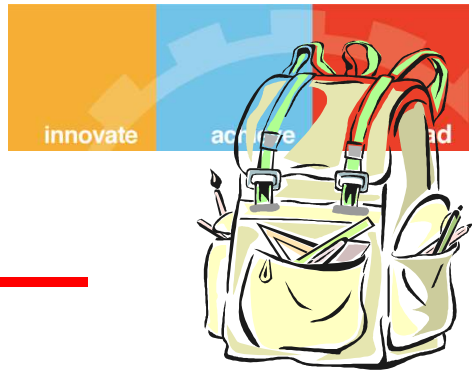


A 0/1 Knapsack Algorithm, Second Attempt



- But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

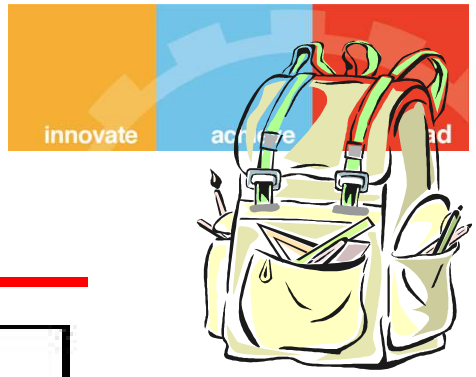
0/1 Knapsack Algorithm



0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

0/1 Knapsack Algorithm



Running time

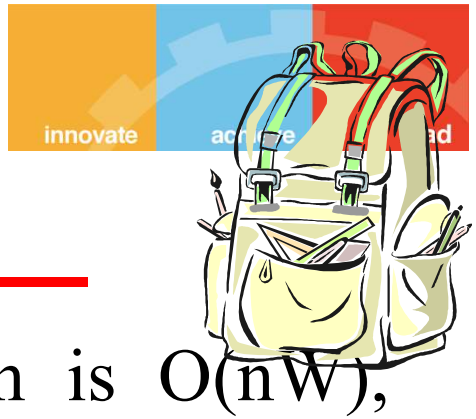
```
for w = 0 to W                    O(W)
    B[0,w] = 0
for i = 1 to n
    B[i,0] = 0
for i = 1 to n                    Repeat n times
    for w = 0 to W                O(W)
        < the rest of the code >
```

What is the running time of this algorithm?

O(n*W)

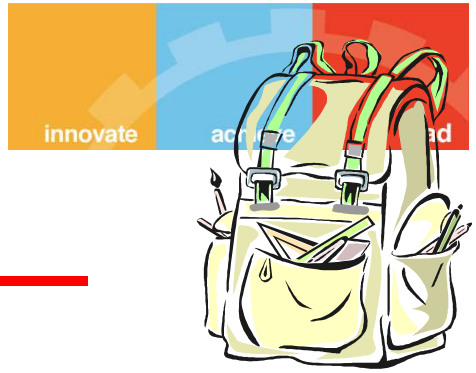
Remember that the brute-force algorithm takes $O(2^n)$

0/1 Knapsack Algorithm



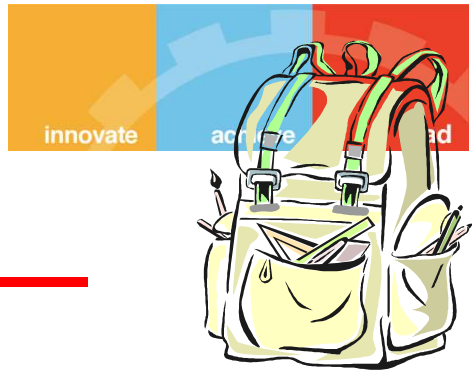
- Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops.
- Depending on W, either the dynamic programming algorithm is more efficient or the brute force **$O(2^n)$,** algorithm could be more efficient.
- (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

0/1 Knapsack Algorithm



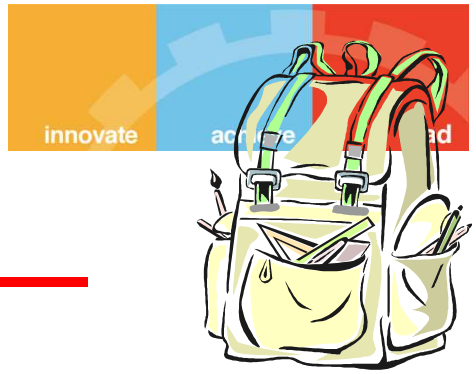
- **Pseudo-polynomial time algorithm**
- Running time of the algorithm depends on a parameter W that, strictly speaking, is not proportional to the size of the input (the n items, together with their weights and benefits, plus the number W).
- If W is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute force method.
- Thus, technically speaking, this algorithm is not a polynomial-time algorithm, for its running time is not actually a function of the size of the input

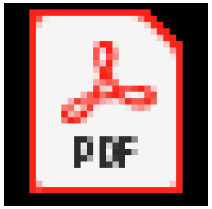
0/1 Knapsack Algorithm



- **Pseudo-polynomial time algorithm**
- Running time depends on the magnitude of a number given in the input, not its encoding size

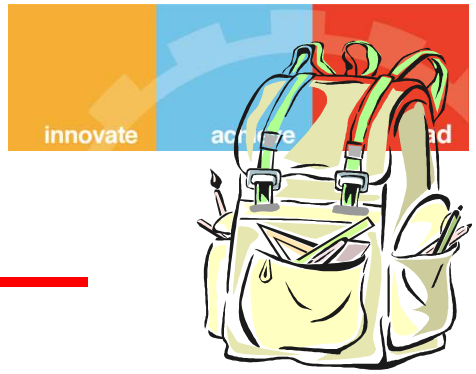
0/1 Knapsack Algorithm





Knapsack
Example Traced

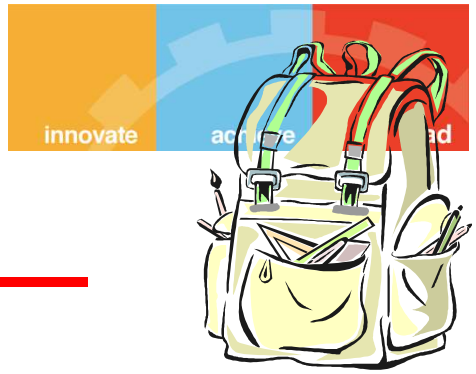
0/1 Knapsack Algorithm-Example2



- **W = 10**

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

0/1 Knapsack Algorithm-Example:Ans



Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

THANK YOU!

BITS Pilani
Hyderabad Campus