Advanced Analysis of Algorithm

Lecture-07: Dynamic Programming

Outline

- Dynamic Programming
  - Fibonacci Numbers
  - Edit Distance
  - Matrix Chain Multiplication
  - 0/1 Knapsack
  - Greedy Algorithms
    - Coin Change
    - Huffman Encoding
    - Activity Selection

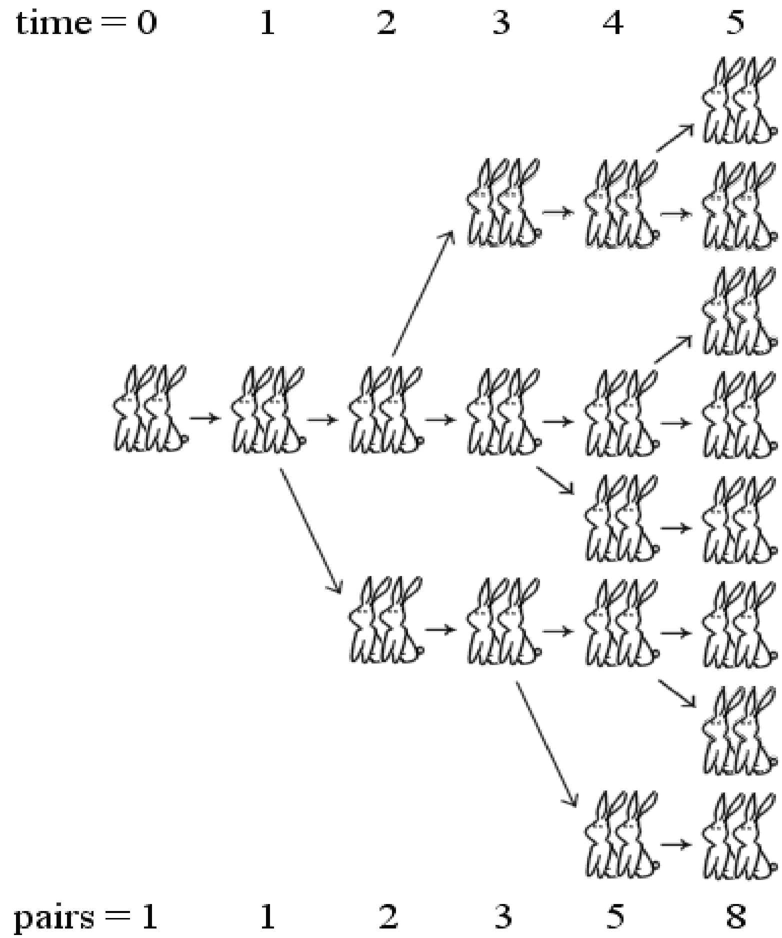
Dynamic Programming

- Suppose we put a pair of rabbits in a place surrounded by on all sides by a wall
- How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?
- Resulting sequence is
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Each number is the sum of previous two numbers

Rabbit Population

Fibonacci investigated (in the year 1202) how fast rabbits could breed in ideal circumstances.

The number of pairs of rabbits in the field at the start of each month is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



Fibonacci

Fibonacci is pronounced [fib-on-arch-ee].
Born in Pisa (Italy) about 1175 AD.

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ *for* $n \geq 2$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

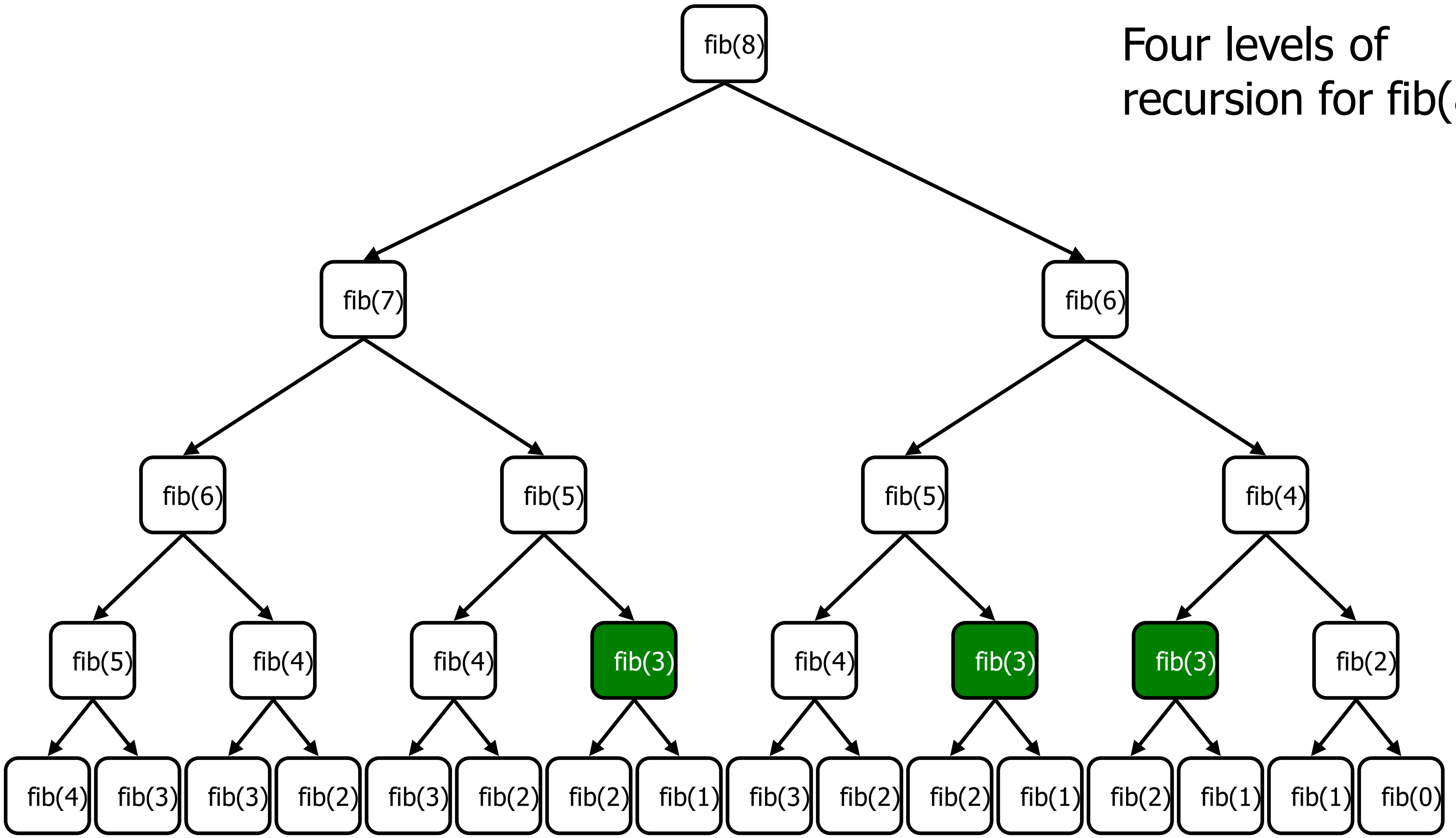
Fibonacci number

- The recursive definition of Fibonacci numb recursive algorithm for computing them:

```
FIB(n)
    if (n < 2) then return n
    else return FIB(n-1)+FIB(n-2)
```

Fibonacci number: Recursive Calls

Four levels of recursion for fib(8)



Fibonacci number: Recursive Calls

- A single recursive call to fib(n) results in
  - One recursive call to fib(n-1)
  - Two recursive call to fib(n-2)
  - Three recursive call to fib(n-3)
  - Four recursive call to fib(n-4) and
  - In general F_{k-1} recursive calls to fib(n-k)
- For each call, we are recomputing the same Fibonacci number from scratch

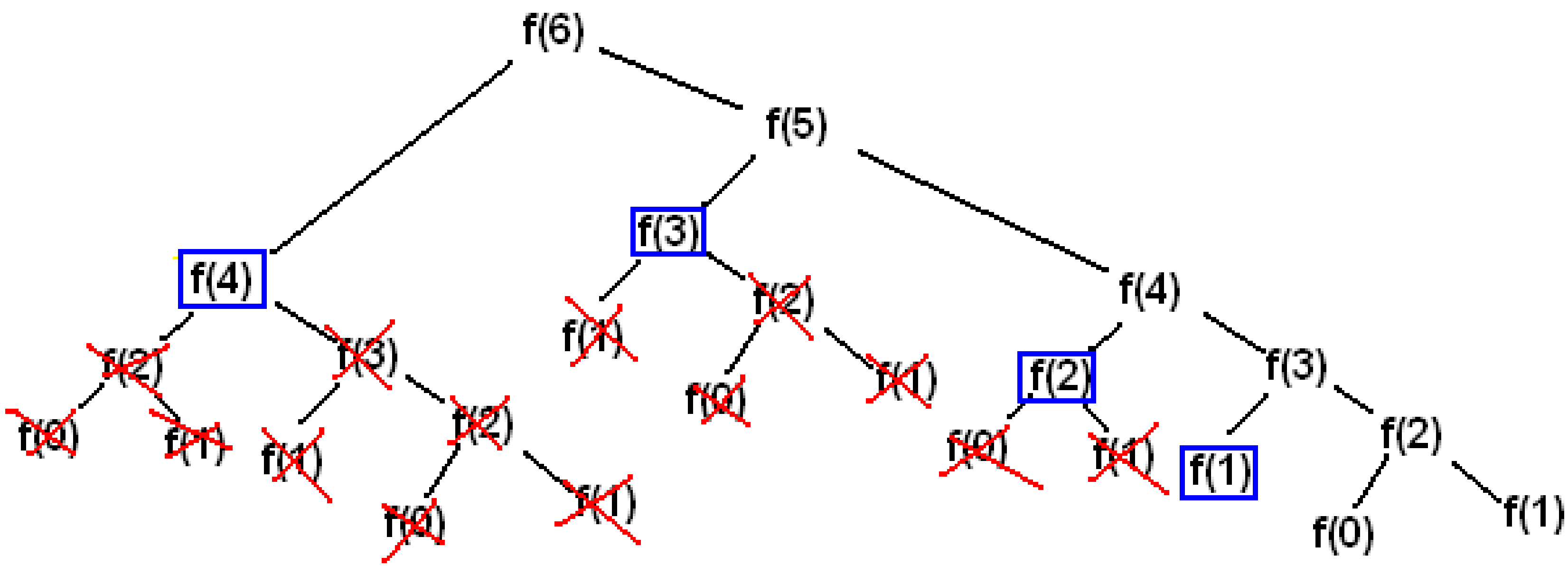
Fibonacci number Memoization

- We can avoid this unnecessary repetition by writing down the results of recursive calls and looking them up again if we need them later
- This process is called memoization
- **Memoization:** *Use a table to remember previously calculated values.* (Store a memo for oneself.)

Fibonacci number Memoization

```
MEMOFIB(n)
    if (n < 2) then return n
    if (F[n] is undefined) then
        F[n] ← MEMOFIB(n-1)+MEMOFIB(n-2)
    return F[n]
```

Recursion avoidance



Fibonacci number: Iterative Algorithm

```
ITERFIB(n)
    F[0] ← 0
    F[1] ← 1
    for i  ← 2 to n do
            F[i] ← F[i-1]+F[i-2]
    return F[n]
```

Fibonacci number: Iterative Algorithm

- This algorithm clearly takes O(n) time to compute F_n
- By contrast the original recursive algorithm takes

$$\Theta(\Phi^n), \Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

Dynamic Programming

- Dynamic programming is essentially recursion without repetition
- Developing a dynamic programming algorithm generally involves two separate steps

1. Formulate the problem recursively
   - Write down a formula for the whole problem as a simple combination of answers to smaller sub problems
2. Build up solution to recurrence from bottom up
   - Write an algorithm that starts with base cases and works its way up to the final solution

Dynamic Programming

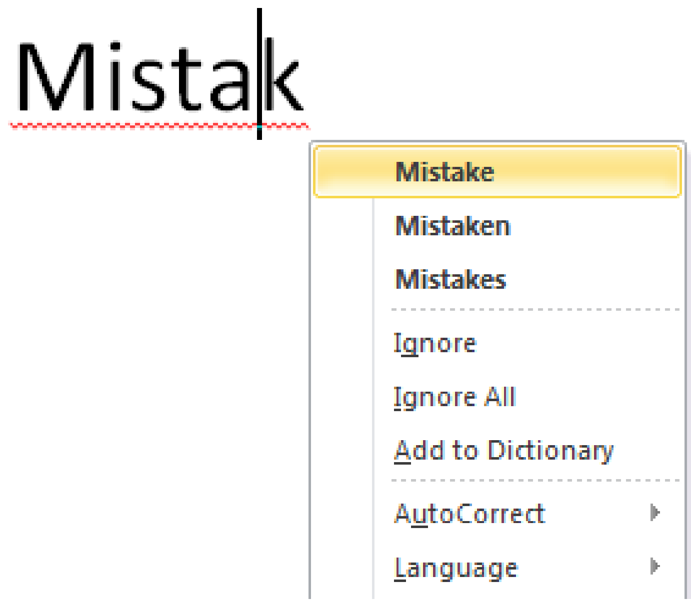
- Dynamic programming algorithms need to store the results of intermediate sub problems
- This is often but not always done with some kind of table

Edit Distance (Levenshtein distance)

- Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character.
- It is named after Vladimir Levenshtein, who considered this distance in 1965.
- It is useful in applications that need to determine how similar two strings are, such as:
  - Spell checking
  - Speech recognition
  - DNA analysis
  - Plagiarism detection

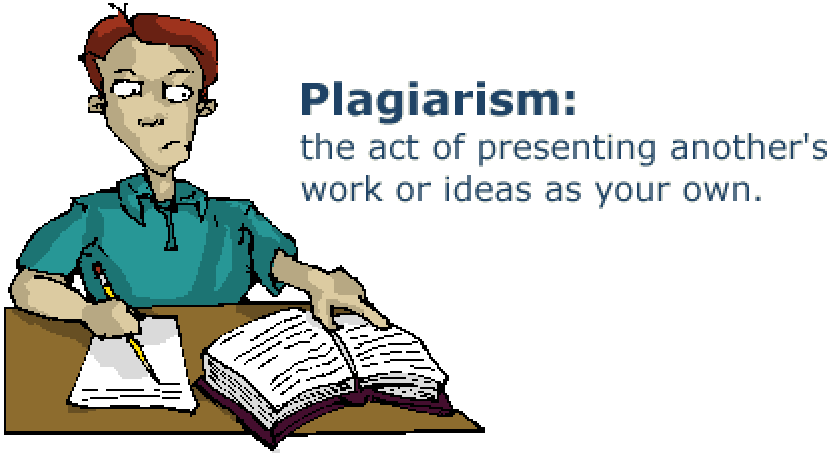
ED Applications Continue..

- Spelling Correction
  - if a text contains a word that is not in the dictionary, a `close' word, i.e. one with a small edit distance, may be suggested as a correction.



Edit Distance Application

- Plagiarism Detection
  - The edit distance provides an indication of similarity that might be too close in some situation



ED Applications Continue..

- Computational Molecular Biology
  - Similarities in DNA Sequences can provide
    - Clue to common evolutionary origin
    - Clue to common function

ED Applications Continue..

- Speech Recognition
  - Algorithm similar to those for the edit-distance problem are used in some speech recognition systems.

Editing Operations

FOOD

MOOD — F replaced with M

MON D — O replaced with N

MONED — E inserted

MONEY — D replaced with Y

Operations: Insertion, Deletion, Substitution, Matching

Edit Distance

- A better to display this editing process is to place the words above the other:

```
M A _ T H S
A _ R T _ S
```

Edit Distance (cont'd)

```
S D I M D M
-----------
M A _ T H S
A _ R T _ S
```

- The first word has a gap for every insertion (I) and the second word has a gap for every deletion (D)
- Columns with two different characters correspond to substitutions (S)
- Matches (M) do not count

Edit Distance (cont'd)

- Edit transcript
  - A string over the alpha bet M, S, I, D that describes a transformation of one string into another
  - Example

1+	1+	1+	0+	1+	0+	=	4
S	*D*	*I*	*M*	*D*	*M*		
M	A	_	T	H	S		
A	_	R	T	_	S		

Edit Distance (cont'd)

- In general it is not easy to determine the optimal edit distance
- For example, the distance between ALGORITHM and ALTRUISTIC is at most 6

```
A L G O R _ I _ T H M
A L _ T R U I S T I C
```

Edit Distance: DP Formulation

- Suppose we have an m-character string A and an n-character string B
- Define $E(i, j)$ to be the edit distance between the first i characters of A and the first j characters of B
- $E(i, j)$

$$\overbrace{\texttt{A L G O R}}^{i}\ \texttt{I T H M}$$

$$\underbrace{\texttt{A L \_ T R}}_{j}\ \texttt{U I S T I C}$$

- The edit distance between entire strings A and B is $E(m,n)$

Edit Distance: DP Formulation

- The gap representation for the edit sequences has a crucial "Optimal Substructure" property
- If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings

Edit Distance: DP Formulation

- Edit distance = 6

```
A L G O R _ I _ T H M
A L _ T R U I S T I C
```

- Remove last column
- Edit distance = 5

```
A L G O R _ I _ T H
A L _ T R U I S T I
```

Base Cases

- There are a couple of obvious base cases
  - The only way to convert an empty string into a string of j characters is by doing j insertions. Thus $E(0, j) = j$
  - The only way to convert a string of i characters into an empty string is with i deletions. Thus $E(i,0) = i$

Deletion

- Four possibilities for the last column in the shortest possible edit sequence
- Deletion: Last entry in the bottom row is empty

$$\overbrace{\texttt{A L G}}^{i=3}\ \texttt{O R I T H M}$$

$$\underbrace{\texttt{A L \_}}_{j=2}\ \texttt{T R U I S T I C}$$

- In this case:
  - E(i, j) = E(i-1, j) + 1
  - E(3,2) = E(2,2) + 1

Insertion

- Insertion: The last entry in the top row is empty

$$\overbrace{\texttt{A L G O R \_}}^{i=5}\ \texttt{I T H M}$$

$$\underbrace{\texttt{A L \_ T R U}}_{j=5}\ \texttt{I S T I C}$$

- In this case
  - $E(i, j) = E(i, j-1) + 1$
  - $E(5,5) = E(5,4) + 1$

Substitution

- Substitution: Both rows have characters in the last column

$$\overbrace{\texttt{A L G O}}^{i=4}\ \texttt{R I T H M}$$

$$\underbrace{\texttt{A L \_ T}}_{j=3}\ \texttt{R U I S T I C}$$

- If the characters are different then
  - E(i, j) = E(i-1, j-1) + 1
  - E(4,3) = E(3, 2) + 1

Match

- Match: Both rows have characters in the last column

$$\overbrace{\texttt{A L G O R}}^{i=5}\ \texttt{I T H M}$$

$$\underbrace{\texttt{A L \_ T R}}_{j=4}\ \texttt{U I S T I C}$$

- If the characters are same, no substitution is needed
  - $E(i, j) = E(i-1, j-1)$
  - $E(5, 4) = E(4, 3)$

Minimum Distance

- Thus the smallest edit distance E(i, j) is the smallest of the four possibilities

$$E(i,j) = \min \begin{pmatrix} \underbrace{E(i-1,j)+1}_{\text{deletion}} & \\ \underbrace{E(i,j-1)+1}_{\text{insertion}} & \\ \underbrace{E(i-1,j-1)+1}_{\text{substitution}} & \text{if } A[i] \neq B[j] \\ \underbrace{E(i-1,j-1)}_{\text{match}} & \text{if } A[i] = B[j] \end{pmatrix}$$

Example

- Consider the example

```
M A T H S
A R T S
```

- The edit distance would be E(5, 4)

Example (Cont'd)

- If we apply recursion to compute, we will have

deletion

insertion

$$E(4,5) = \min \begin{pmatrix} E(3,5)+1 & \\ E(4,4)+1 & \\ E(3,4)+1 & \textit{if } A[4] \neq B[5] \\ E(3,4) & \textit{if } A[4] = B[5] \end{pmatrix}$$

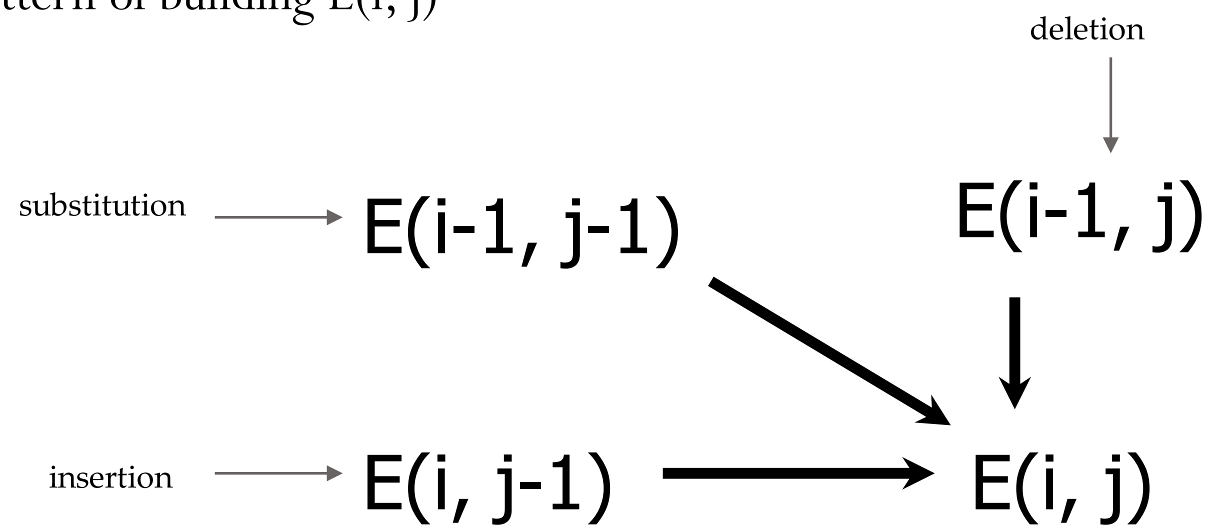
substitution

match

- Recursion clearly leads to the same repetitive call pattern that we have seen in Fibonacci sequence
- We will build the solution bottom up

Computing E(i, j)

- Pattern of building E(i, j)



- Use the base case E(0, j) to fill first row
- Use the base case E(i, 0) to fill first column
- Fill the remaining E matrix row by row

Fill Pattern

diagonal	*above*
left	*min (above + delete, diagonal + substitute, left + insert)*

Cost Matrix

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	1	1	1	1
A	↓ 2	0	1	1	1
T	↓ 3	1	1	0	1
H	↓ 4	1	1	1	1
S	↓ 5	1	1	1	0

Example

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ ↓ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘ ↓ 3	↘ ↓ 3	↘ 3
S	↓ 5	↓ 4	↘ ↓ 4	↘ ↓ 4	↘ ↓ 3

Solution Paths

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘↓ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘↓ 3

Solution Path 1

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ ↓ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘ ↓ 3	↘ ↓ 3	↘ 3
S	↓ 5	↓ 4	↘ ↓ 4	↘ ↓ 4	↘ ↓ 3

```
1+ 0+ 1+ 1+ 0 = 3
D  M  S  S  M
M  A  T  H  S
_  A  R  T  S
```

Solution Path 2

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘↓ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘↓ 3

1+	1+	0+	1+	0	= 3
S	S	M	D	M	
M	A	T	H	S	
A	R	T	_	S	

Solution Path 3

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ ↓ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘ ↓ 3	↘ ↓ 3	↘ 3
S	↓ 5	↓ 4	↘ ↓ 4	↘ ↓ 4	↘ ↓ 3

1+	0+	1+	0+	1+	0+	=	3
D	M	I	M	D	M		
M	A	_	T	H	S		
_	A	R	T	_	S		

Edit Distance DP Algorithm

```
int LevenshteinDistance(char s[1..m], char t[1..n])

   declare int d[0..m, 0..n]

   for i from 0 to m do d[i, 0] := i
   for j from 0 to n do d[0, j] := j

   for i from 1 to m
       for j from 1 to n
           if s[i-1] = t[j-1] then cost := 0 else cost := 1
           mc = min(d[i-1,j] + 1,          // deletion
                    d[i,j-1] + 1,          // insertion
                    d[i-1, j-1] + cost)    // substitution
           d[i, j] := mc

   return d[m, n]
```

Edit Distance Analysis

- There are $\Theta(n^2)$ entries in the matrix
- Each entry $E(i, j)$ takes $\Theta(1)$ time to compute
- The total running time is $\Theta(n^2)$

Matrix

- A rectangular Array ,denoted by some capital letter, say A, and is of the form given below is called a matrix of order m x n

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots. & a_{2j} & \dots. & a_{2n} \\ \dots. & \dots. & \dots. & \dots. & \dots. & \dots. \\ a_{i1} & a_{i2} & \dots. & \dots. & \dots. & a_{in} \\ \dots & \dots. & \dots. & \dots. & \dots. & \dots. \\ a_{m1} & a_{m2} & \dots. & a_{mj} & \dots. & a_{mn} \end{pmatrix}$$

Matrix

- Order of a Matrix : if A be a matrix having m rows and n columns, then its Order is $m \times n$ (read as m by n).
- General Element of a Matrix: In the Matrix A ,the element $\mathbf{a}_{ij}$ is called the general element . The Subscripts i stands for row and j stands for column.
- So $\mathbf{a}_{ij}$ lies at the intersection of the i^{th} row and the j^{th} column of the matrix A.
- Square matrix : if the number of rows in matrix A equals number of columns then matrix A is called a Square matrix

Matrix Multiplication

- Two matrices A and B are said to be conformable for multiplication if

Number of columns of A = Number of rows of B

- Let the matrix A = | a_{ij} | be of order m x **n** and B=|b $_{jk}$ | be of the order **n** x p.

  AB = C is defined where C= |c_{ik}| is of order n x n and

  For i = 1,2,…,m, k = 1,2,…,p.

$$c_{ij} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

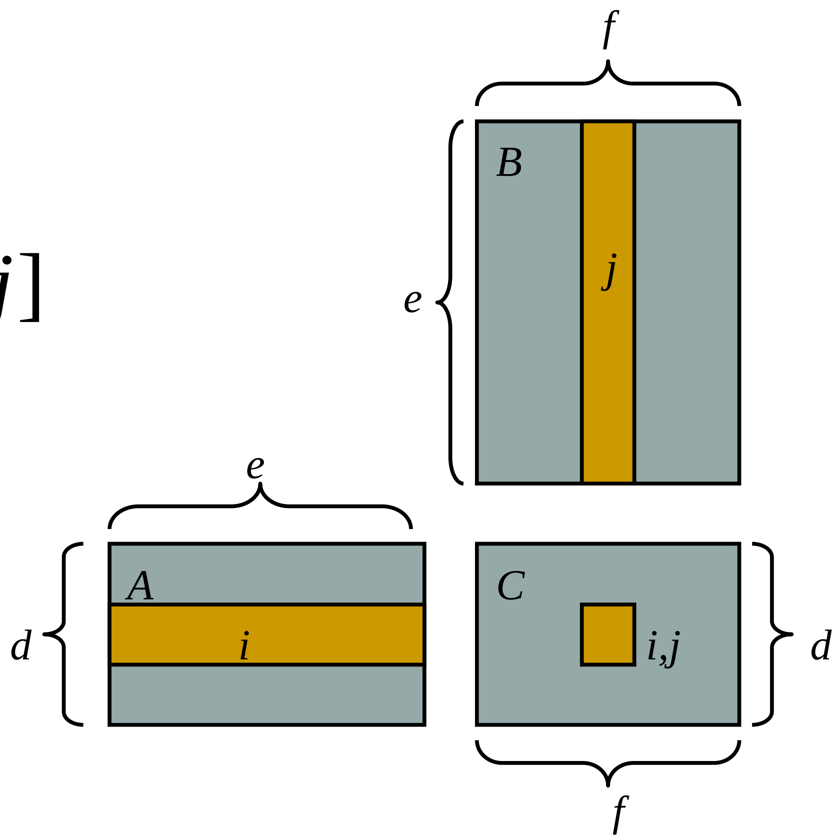
Matrix Multiply

- In particular, for $1 \leq i \leq p$ and $1 \leq j \leq r$

$$C[i, j] = \sum_{k=1}^{q} A[i, k] B[k, j]$$

- There are $(p \cdot r)$ total entries in C and each takes O(q) to compute
- Thus the total number of multiplications is

$(p \cdot q \cdot r)$



Example 1

$$\text{If } A = \begin{pmatrix} 2 & 5 & 4 \\ 1 & 7 & 6 \\ 2 & 3 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 4 & 5 \\ 8 & 7 & 6 \\ 9 & 2 & 1 \end{pmatrix}$$

Order of A = 3 x 3

Order of B= 3 x 3

Columns of A = Rows of B = 3

So multiplication is possible.

$$\begin{matrix} 2 & 5 & 4 \\ 1 & 7 & 6 \\ 2 & 3 & 1 \end{matrix} \qquad \begin{matrix} 3 & 4 & 5 \\ 8 & 7 & 6 \\ 9 & 2 & 1 \end{matrix}$$

$$C = \begin{pmatrix} \mathbf{2x3 + 5x8 + 4x9} & 2x4 + 5x7 + 4x2 & 2x5 + 5x6 + 4x1 \\ 1x3 + 7x8 + 6x9 & 1x4 + 7x7 + 6x2 & 1x5 + 7x6 + 6x1 \\ 2x3 + 3x8 + 1x9 & 2x4 + 3x7 + 1x2 & 2x5 + 3x6 + 1x1 \end{pmatrix}$$

1st element of the product matrix C is obtained by multiplying the elements of the 1st row of Matrix A with the elements of the 1st column in the Matrix B and then summing, and so on, all the elements are calculated

$$C = \begin{pmatrix} 82 & 49 & 25 \\ 73 & 65 & 52 \\ 39 & 29 & 29 \end{pmatrix}$$

Example 2

If $A = \begin{pmatrix} 2 & 5 & 4 \\ 1 & 7 & 6 \end{pmatrix}$ $B = \begin{pmatrix} 3 & 4 & 5 \\ 8 & 7 & 6 \\ 9 & 2 & 1 \end{pmatrix}$

Order of A = 2 x 3

Order of B= 3 x 3

Columns of A = Rows of B = 3

So multiplication is possible. Resultant Matrix will be of the Order 2 x 3.

Sequential Algorithm for Matrix Multiplication

```
procedure MATRIX _MULT (A, B, C)
    begin
        for i := 0 to n - 1 do
            for j := 0 to n - 1 do
            begin
                C[i, j] := 0
                for k := 0 to n - 1 do
                    C[i, j] := C[i, j] + A[i, k] * B [k, j]
            end
end MATRIX_MULT
```

General form for n = 3

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} =$$

$$\begin{bmatrix} a_{00}*b_{00}+a_{01}*b_{10}+a_{02}*b_{20} & a_{00}*b_{01}+a_{01}*b_{11}+a_{02}*b_{21} & a_{00}*b_{02}+a_{01}*b_{12}+a_{02}*b_{22} \\ a_{10}*b_{00}+a_{11}*b_{10}+a_{12}*b_{20} & a_{10}*b_{01}+a_{11}*b_{11}+a_{12}*b_{21} & a_{10}*b_{02}+a_{11}*b_{12}+a_{12}*b_{22} \\ a_{20}*b_{00}+a_{21}*b_{10}+a_{22}*b_{20} & a_{20}*b_{01}+a_{21}*b_{11}+a_{22}*b_{21} & a_{20}*b_{02}+a_{21}*b_{12}+a_{22}*b_{22} \end{bmatrix}$$

Processing for n = 3

Pass 1	i = 0 to n-1 i= 0	j= 0 to n-1 j= 0 to 2	k= 0 to n-1 K= 0 to2	C[i, j] := C[i, j] + A [i, k] * B [k, j]
		j=0	K=0	C[0,0] := C[0,0] + A[0, 0] * B [0,0]
			K=1	C[0,0] := C[0,0] + A[0, 0] * B [1,0]
			K=2	C[0,0] := C[0,0] + A[0, 0] * B [2,0]
		j= 1	K=0	C[0,1] := C[0,1] + A[1, 0] * B [0,1]
			K=1	C[0,1] := C[0,1] + A[1, 0] * B [1,1]
			K=2	C[0,1] := C[0,1] + A[1, 0] * B [2,1]
		j=2	K=0	C[0,2] := C[0,2] + A[1, 0] * B [0,2]
			K=1	C[0,2] := C[0,2] + A[1, 0] * B [1,2]
			K=2	C[0,2] := C[0,2] + A[1, 0] * B [2,2]

Cont'd

Pass 2	i = 0 to n-1 i= 1	j= 0 to n-1 j= 0 to 2	k= 0 to n-1 K= 0 to2	C [i, j] := C[i, j] + A [i, k] * B [k, j]
		j=0	K=0	C[1,0] := C[1,0] + A[0, 0] * B [0,0]
			K=1	C[1,0] := C[1,0] + A[0, 0] * B [1,0]
			K=2	C[1,0] := C[1,0] + A[0, 0] * B [2,0]
		j= 1	K=0	C[1,1] := C[1,1] + A[1, 0] * B [0,1]
			K=1	C[1,1] := C[1,1] + A[1, 0] * B [1,1]
			K=2	C[1,1] := C[1,1] + A[1, 0] * B [2,1]
		j=2	K=0	C[1,2] := C[1,2] + A[1, 0] * B [0,2]
			K=1	C[1,2] := C[1,2] + A[1, 0] * B [1,2]
			K=2	C[1,2] := C[1,2] + A[1, 0] * B [2,2]

Cont'd

Pass 3	i = 0 to n-1 i= 2	j= 0 to n-1 j= 0 to 2	k= 0 to n-1 K= 0 to2	C [i, j] := C[i, j] + A [i, k] * B [k, j]
		j=0	K=0	C[2,0] := C[1,0] + A[0, 0] * B [0,0]
			K=1	C[2,0] := C[1,0] + A[0, 0] * B [1,0]
			K=2	C[2,0] := C[1,0] + A[0, 0] * B [2,0]
		j= 1	K=0	C[2,1] := C[1,1] + A[1, 0] * B [0,1]
			K=1	C[2,1] := C[1,1] + A[1, 0] * B [1,1]
			K=2	C[2,1] := C[1,1] + A[1, 0] * B [2,1]
		j=2	K=0	C[2,2] := C[1,2] + A[1, 0] * B [0,2]
			K=1	C[2,2] := C[1,2] + A[1, 0] * B [1,2]
			K=2	C[2,2] := C[1,2] + A[1, 0] * B [2,2]

Complexity

- It is clear from the processing that sequential algorithm For the 3 by 3 matrix case requires 27 multiplications.
- The complexity of this algorithm is clearly (n^3).

Chain Matrix Multiply

- Suppose we wish to multiply a series of matrices
  - $A_1A_2A_3\ldots A_n$
- In what order should the multiplication be done?
- A $p \times q$ matrix A can be multiplied with a $q \times r$ matrix B
- The result will be a $p \times r$ matrix C

Chain Matrix Multiply

- Consider the case of 3 matrices:
  - A_1 is 5×4
  - A_2 is 4×6
  - A_3 is 6×2
- The multiplication can be carried out either as
  - $((A_1A_2)A_3)$ or
  - $(A_1(A_2A_3))$
- The cost of the two is
  - $((A_1A_2)A_3) = (5\cdot4\cdot6)+(5\cdot6\cdot2) = 180$
  - $(A_1(A_2A_3)) = (4\cdot6\cdot2)+(5\cdot4\cdot2) = 88$
- There is considerable savings achieved even for this simple example

Matrix Chain Multiplication Problem

- Given a sequence $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$, where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the number of operations

- If there are n items, there are n-1 ways in which the outer most pair of parenthesis can be placed
  - $(A_1)(A_2A_3A_4\ldots A_n)$
  - or $(A_1A_2)(A_3A_4\ldots A_n)$
  - or $(A_1A_2A_3)(A_4\ldots A_n)$
  - …
  - or $(A_1A_2A_3A_4\ldots A_{n-1})(A_n)$

- **Matrix Chain-Product Algorithm**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best

Chain Matrix Multiply

- In what order should we multiply a series of matrices $A_1A_2A_3\ldots A_{n?}$
- Matrix multiplication is an associative but not commutative operation
- We are free to add parenthesis the above multiplication but the order of matrices can not be changed

Matrix Chain Multiplication Problem

- Once we split just after the kth matrix, we create two sub-lists to be parenthesized, one with k and other with n-k matrices
  - $(A_1A_2A_3\ldots A_k)(A_{k+1}\ldots A_n)$

- Since these are independent choices, if there are L ways of parenthesizing the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$

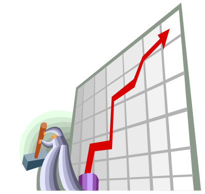
Matrix Chain Multiplication Problem

- This suggests the following recurrence for $P(n)$, the different ways of parenthesizing n items

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

- This is related to the famous function in combinatronics called Catalan numbers
- Catalan numbers are related to the number of different binary trees on n nodes
- Catalan number is given by the formula: $C(n) = \frac{1}{n+1}\binom{2n}{n}$

Matrix Chain Multiplication-DP



- The dynamic programming solution involves breaking up the problem into subproblems whose solutions can be combined to solve the global problem
- Let $A_{i..j}$ be the result of multiplying matrices i through j
- It is easy to see that $A_{i..j}$ is a p_{i-1} x p_j matrix

$$A_3 \quad A_4 \quad A_5 \quad A_6 \quad = \quad A_{3..6}$$

$$4\times 5 \quad 5\times 2 \quad 2\times 8 \quad 8\times 7 \quad = \quad 4\times 7$$

Matrix Chain Multiplication-DP

- At the highest level of parenthesization we multiply two matrices
  - $A_{1..n} = A_{1..k} A_{k+1..n}$ $1 \le k \le n-1$
- The question now is what is optimum value of k for the split and how do we parenthesize the sub-chains $A_{1..k} A_{k+1..n}$
- We cannot use divide and conquer because we do not know what is the optimum k
- We will have to consider all possible values of k and take the best of them
- We will apply this strategy to solve the sub-problems optimally

Dynamic Programming Formulation

- We will store the solutions to the subproblem in a table and build the table bottom up

- For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$

- The optimum can be described by the following recursive formulation

Matrix Chain Multiplication-DP

- If $i = j$, there is only one matrix and thus $m[i, i] = 0$ (the diagonal entries)

- If $i < j$, then we are asking for the product $A_{i..j}$

- This can be split by considering each k, $i \le k \le j$, as $A_{i..k}$ times $A_{k+1..j}$

- The optimum time to compute $A_{i..k}$ is m[i, k] and optimum time for $A_{k+1..j}$ is in m[k+1, j]

Matrix Chain Multiplication-DP

- Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is $p_k \times p_j$ matrix, the time to multiply them $p_{i-1} \times p_k \times p_j$
- This suggest the following recursive rule:
  - $m[i, i] = 0$
  - $m[i, j] = \min_{i \le k \le j}(m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$

Matrix Chain Multiplication-DP

- $m[i, j] = \min_{i \le k \le j}(m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$

- For a specific k, $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$
  $= A_i \ldots A_k$ (m[i, k] multiplications)
  $= A_{k+1} \ldots A_j$ (m[k+1, j] multiplications)
  $= A_{i \ldots j}$ ($p_{i-1} p_k p_j$ multiplications)

- We do not want to calculate m entries recursively
- How should we proceed?
- We will fill m along diagonals

Matrix Chain Multiplication-DP

- Set all $m[i, i] = 0$ using the base condition
- Compute cost of multiplication of a sequence of 2 matrices
- These are
  - $m[1, 2]$, $m[2, 3]$, $m[3, 4]$, …, $m[n\text{-}1, n]$
- m[1,2], for example is
  - $m[1, 2] = m[1, 1] + m[2, 2] + p_0 \cdot p_1 \cdot p_2$

Matrix Chain Multiplication-DP

- For example, for m for product of 5 matrices at this stage would be

m[1,1]	**←m[1,2]** ↓			
	m[2,2]	**←m[2,3]** ↓		
		m[3,3]	**←m[3,4]** ↓	
			m[4,4]	**←m[4,5]** ↓
				m[5,5]

Matrix Chain Multiplication-DP

- Next, we compute cost of multiplication for sequence of three matrices.
- These are
  - $m[1,3]$, $m[2,4]$, $m[3,5]$, …, $m[n\text{-}2, n]$
- $m[1,3]$, for example is

$$m[1,3] = \min \begin{Bmatrix} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 \end{Bmatrix}$$

Matrix Chain Multiplication-DP

- We repeat the process for sequence of four, five and higher number of matrices
- The final result will end up in m[1,n]
- Let us go through an example. We want to find the optimal multiplication order for

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5$$

(5×4) (4×6) (6×2) (2×7) (7×3)

Matrix Chain Multiplication-DP

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

Matrix Chain Multiplication-DP

$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2 = 0+0+5\cdot4\cdot6 = 120$

$m[2,3] = m[2,2] + m[3,3] + p_1 \cdot p_2 \cdot p_3 = 0+0+4\cdot6\cdot2 = 48$

$m[3,4] = m[3,3] + m[4,4] + p_2 \cdot p_3 \cdot p_4 = 0+0+6\cdot2\cdot7 = 84$

$m[4,5] = m[4,4] + m[5,5] + p_3 \cdot p_4 \cdot p_5 = 0+0+2\cdot7\cdot3 = 42$

Matrix Chain Multiplication-DP

	1	2	3	4	5
1	0	120			
2		0	48		
3			0	84	
4				0	42
5					0

Matrix Chain Multiplication-DP

$m[1,3] = m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_2 = 0+48+5\cdot4\cdot2 = 88$
$m[1,3] = m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 120+0+5\cdot6\cdot2 = 180$
minimum $m[1,3] = 88$ at $k = 1$

$m[2,4] = m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 = 0+84+4\cdot6\cdot7 = 252$
$m[2,4] = m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 = 48+0+4\cdot2\cdot7 = 104$
minimum $m[2,4] = 104$ at $k = 3$

$m[3,5] = m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 = 0+42+6\cdot2\cdot3 = 78$
$m[3,5] = m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 = 84+0+6\cdot7\cdot3 = 210$
minimum m[3,5] = 78 at $k = 3$

Matrix Chain Multiplication-DP

	1	2	3	4	5
1	0	120	88		
2		0	48	104	
3			0	84	78
4				0	42
5					0

Matrix Chain Multiplication-DP

$m[1,4] = m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0+104+5\cdot4\cdot7 = 244$

$m[1,4] = m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 120+84+5\cdot6\cdot7 = 414$

$m[1,4] = m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 88+0+5\cdot2\cdot7 = 158$

minimum $m[1,4] = 158$ at $k = 3$

$m[2,5] = m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 = 0+78+4\cdot6\cdot3 = 150$

$m[2,5] = m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 = 48+42+4\cdot2\cdot3 = 114$

$m[2,5] = m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 = 104+0+4\cdot7\cdot3 = 188$

minimum $m[2,5] = 114$ at $k = 3$

Matrix Chain Multiplication-DP

	1	2	3	4	5
1	0	120	88	158	
2		0	48	104	114
3			0	84	78
4				0	42
5					0

Matrix Chain Multiplication-DP

$m[1,5] = m[1,1] + m[2,5] + p_0 \cdot p_1 \cdot p_5 = 0+114+5\cdot4\cdot3 = 174$

$m[1,5] = m[1,2] + m[3,5] + p_0 \cdot p_2 \cdot p_5 = 120+78+5\cdot6\cdot3 = 288$

$m[1,5] = m[1,3] + m[4,5] + p_0 \cdot p_3 \cdot p_5 = 88+42+5\cdot2\cdot3 = 160$

$m[1,5] = m[1,4] + m[5,5] + p_0 \cdot p_4 \cdot p_5 = 158+0+5\cdot7\cdot3 = 263$

minimum $m[1,5] = 160$ at $k = 3$

Matrix Chain Multiplication-DP

	1	2	3	4	5
1	0	120	88	158	160
2	0	0	48	104	114
3	0	0	0	84	78
4	0	0	0	0	42
5	0	0	0	0	0

Order of Calculation of m entries

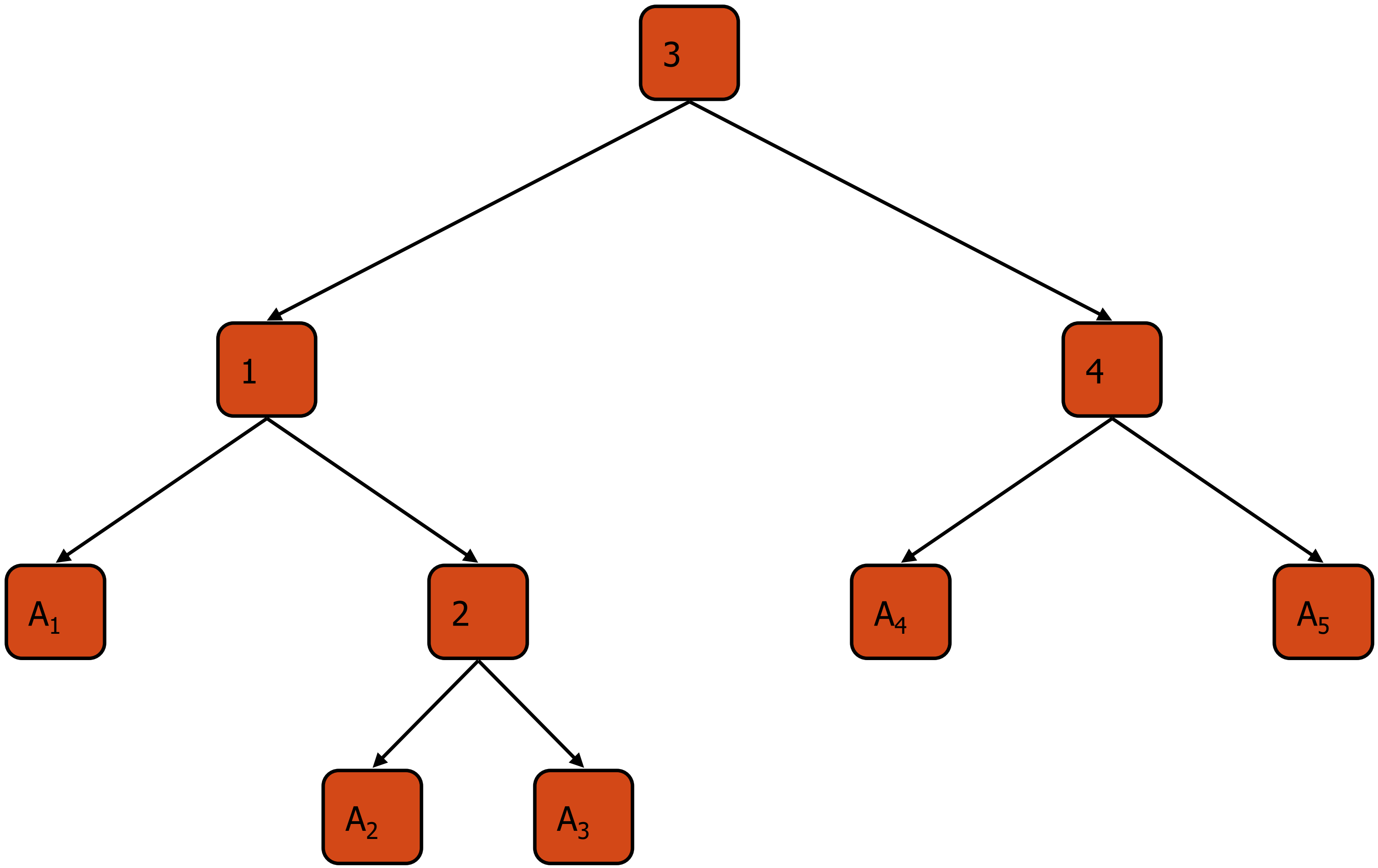
	1	2	3	4	5
1	0	1	5	8	10
2		0	2	6	9
3			0	3	7
4				0	4
5					0

Split k values

	1	2	3	4	5
1	0	1	1	3	3
2		0	2	3	3
3			0	3	3
4				0	4
5					0

Matrix Chain Multiplication-DP

Optimal order for multiplication: $((A_1(A_2A_3))(A_4A_5))$



Chain Matrix Multiplication Algorithm

```
MATRIXCHAIN(p, N)
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=1 to j-1 do
         t ← m[i,k]+m[k+1,j]+p_{i-1}+p_k+p_j
            if (t<m[i,j]) then
               m[i,j] ←t
               s[i,j] ←k
```

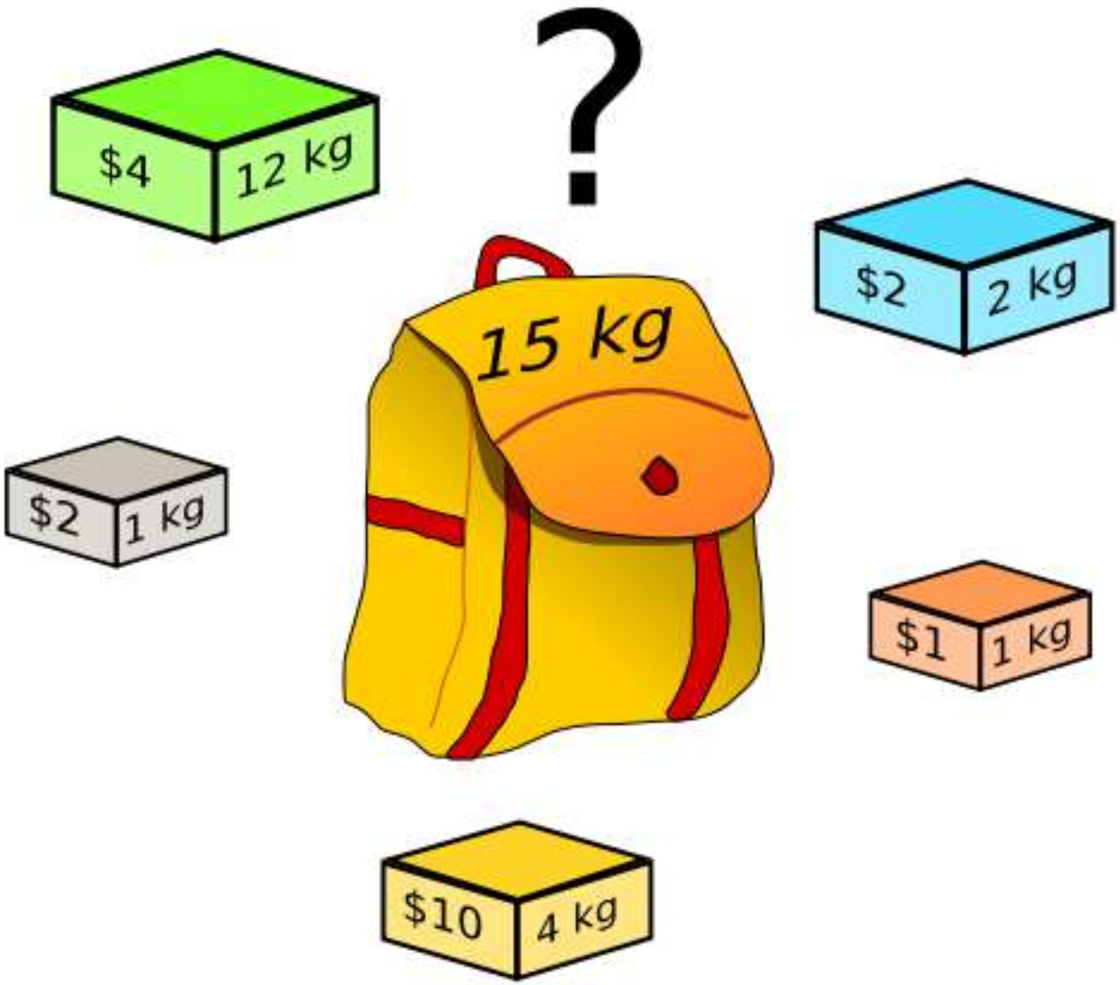
Analysis of Chain Matrix Multiplication

- There are three nested loops
- Each loop executes a maximum n times
- Total time is thus $\Theta(n^3)$
- Extracting the final sequence we use MULTIPLY algorithm

- MULTIPLY(i,j) Algorithm

  if (i=j) then return A[i]
  else k ←s[i,j]
  X ← MULTIPLY(i,k)
  Y ← MULTIPLY(k+1,j)
  return X · Y

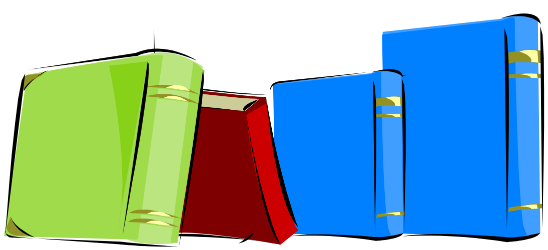
0/1 Knapsack



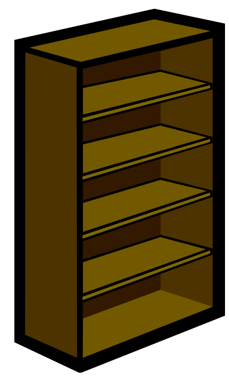
Example

- Given: A set S of n items, with each item i having
  - v_i - a positive value (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 in	2 in	2 in	6 in	2 in
Benefit:	20	3	6	25	80



9 in

"knapsack"

Solution:

- 5 (2 in)
- 3 (2 in)
- 1 (4 in)

0/1 Knapsack

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item I has some weight w_i and value v_i (all w_i, v_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Example 0/1 Knapsack Problem

Item$_i$	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Knapsack can hold W=20

0/1 Knapsack

- Mathematically, the problem is

$$\text{maximize} \sum_{i \in T} v_i$$

$$\text{subjected to} \sum_{i \in T} w_i \le w$$

- The problem is called a "0-1" problem, because each item must be entirely accepted or rejected

0/1 Knapsack Problem

- Try the brute-force solution
  - Since there are n items, there are 2^n possible combinations of the items (an item either chosen or not)
  - We go through all combinations and find the one with the most total value and with total weight less or equal to W
  - Running time will be $O(2^n)$
- Can we do better?
  - Yes, with an algorithm based on dynamic programming
  - We need to carefully identify the subproblems

0/1 Knapsack Problem

- Let us try this
  - If items are labeled 1,2, …, n, then a subproblem would be to find an optimal solution for S_k = items labeled 1,2,…, k
  - This is a valid subproblem definition
  - The question is:
    - Can we describe the final solution S_n in terms of subproblems S_k?
    - Unfortunately we cannot do that

Example

- Solution S_4
- Items chosen are 1,2,3,4
- Total weight: 2+3+4+5=14
- Total value: 3+4+5+8=20

Item_i	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Example

- Solution S_5
- Items chosen are 1,3,4,5
- Total weight: 2+4+5+9=20
- Total value: 3+5+8+10=26
- S_4 is not part of S_5
- The solution S4 is not part of the solution S5
- So our definition of a subproblem is flawed and we need another one

Item$_i$	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

0/1 Knapsack Problem-DP

- The Dynamic Programming Approach
  - For each $i \leq$ n and each $w \leq$ W, solve the knapsack problem for the first i objects when the capacity of knapsack is W.
  - Why will this work?
  - Because solution to larger subproblems can be built up easily from solutions to smaller ones

0/1 Knapsack Problem-DP

- We construct a matrix V[0..n, 0..W]
- For $1 \le i \le n$ and $0 \le j \le W$, $V[i,j]$ will store the maximum value of any set of objects $\{1,2,\ldots,i\}$ that can fit into a knapsack of weight j
- V[n, W] will contain the maximum value of all n objects that can fit into the entire knapsack of weight W

0/1 Knapsack Problem-DP

- To compute entries of V we will imply an inductive approach
- As a basis, $V[0,j]=0$ for $0 \leq j \leq W$ since if we have no items then we have no value
- We consider two cases

The 0/1 Knapsack Problem

- Leave object *i*
  - If we choose to not take object *i*, then the optimal value will come about by considering how to fill a knapsack of size j with the remaining objects $\{1,2,\ldots,i\text{-}1\}$
  - This is just V[*i*-1,j]

The 0/1 Knapsack Problem

- Take object i
  - If we take object i, then we gain a value of v_i
  - But we use up w_i of our capacity
  - With the remaining $j-w_i$ capacity in the knapsack, we can fill it in the best possible way with objects $\{1,2,\ldots,i-1\}$
  - This is $v_i+V[i-1,j-w_i]$
  - This is only possible if $w_i \leq j$

0/1 Knapsack

- Recursive formulation

$$V[i, j] = -\infty \quad \text{if } \; j < 0$$

$$V[0, j] = 0 \quad \text{if } \; j \geq 0$$

$$V[i, j] = \begin{cases} V[i-1, j] & \text{if } \; w_i > j \\ \max\{V[i-1, j], v_i + V[i-1, j-w_i]\} & \text{if } \; w_i \leq j \end{cases}$$

- A simple evaluation of this recursive definition is exponential

0/1 Knapsack

- So, as usual, we avoid re-computation by making a table
- Consider an example: max weight W is 11.
- There are five items to choose from

0/1 Knapsack:DP

Weight limit (j)	0	1	2	3	4	5	6	7	8	9	10	11
W_1=1, v_1=1	0	1	1	1	1	1	1	1	1	1	1	1
W_2=2, v_2=6	0	1	6	7	7	7	7	7	7	7	7	7
W_3=5, v_3=18	0	1	6	7	7	18	19	24	25	25	25	25
W_4=6, v_4=22	0	1	6	7	7	18	22	24	29	20	20	40
W_5=7, v_5=28	0	1	6	7	7	18	22	24	29	34	35	40

The [i,j] entry here will be V[i,j], the best value obtainable using the first i rows of terms if the maximum capacity were j

0/1 Knapsack: DP Algorithm

```
KNAPSACK(n, W)
  for w ← 0 to W do V[0,W] ← 0
  for i ← 0 to n do V[i,0] ← 0
  for w ← 0 to W do
     if (w_i ≤ w AND v_i + V[i-1,w-w_i] > V[i-1,w]) then
          V[i,w] ← vi+V[i-1,w-wi]
     else
          V[i,w] ← V[i-1,w]
```

Time Complexity: clearly O(n·W)