Advanced Algorithms

Dynamic programming

Like divide-and-conquer, solve problem by combining the solutions to sub-problems.

Approach-

• solve sub problems,

• store results,

• use the results while solving bigger instances of the problem with out re-computing

DP vs DC

Divide and conquer is aimed at dividing the problem into smaller instances, solve instances and combine to get final solution.

Dynamic Programming, divides, solves, combines but there could be overlaps, memorizes earlier solutions and uses it

Application domain of DP

- Optimization problem: find a solution with optimal (maximum or minimum) value.
- *An* optimal solution, not *the* optimal solution, since may more than one optimal solution, any one is OK.

Typical steps of DP

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom-up fashion.
- Compute an optimal solution from computed/stored information.

Matrix Multiplication

- Note that any matrix multiplication between a matrix with dimensions **pxq** and another with dimensions **qxr** will perform **pxqxr** element multiplications creating an answer that is a matrix with dimensions **pxr**.

- Also note the condition that the second dimension in the first matrix and the first dimension in the second matrix must be equal in order to allow matrix multiplication (the number of columns of the first matrix should be equal to the number of rows of the second matrix) .

Matrix Multiplication

The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

for $1 \le i \le p$ and $1 \le j \le r$.

Example: If

$$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

then

$$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}$$

No. of multiplications =
3*6=18
OR: p*q*r=3*3*2=18

Matrix Multiplication

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p \times q}$ and $B_{q \times r}$ (with dimensions $p \times q$ and $q \times r$)

Result: Matrix $C_{p \times r}$ resulting from the product $A \cdot B$

MATRIX-MULTIPLY($A_{p \times q}$, $B_{q \times r}$)

1. **for** $i \leftarrow 1$ **to** p
2.   **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4.   **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

> Scalar multiplication in line 5 dominates time to compute C
> Number of scalar multiplications = pqr

Matrix Chain Multiplication

- Problem: given $<A_1, A_2, \ldots, A_n>$, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.
- Given some matrices to multiply, determine the best order to multiply them so you can minimize the number of single element multiplications.
  - i.e. Determine the way the matrices are parenthesized.

- First off, it should be noted that matrix multiplication is associative, but not commutative. Since it is associative, we always have:

- $((AB)(CD)) = (A(B(CD)))$, or any other grouping as long as the matrices are in the same consecutive order.
  i.e. Parenthesization does not change the result.

- BUT : $((AB)(CD)) \neq ((BA)(DC))$

Matrix-chain Multiplication

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication

- 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 $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 … A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Matrix-chain multiplication

- It may appear that the amount of work done won't change if you change the parenthesization of the expression, but we can prove that is not the case!

- Let us use the following example:
  - Let A be a 2x10 matrix
  - Let B be a 10x50 matrix
  - Let C be a 50x20 matrix

Matrix-chain multiplication

- – Let A be a 2x10 matrix
- – Let B be a 10x50 matrix
- – Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - – # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix

Matrix-chain multiplication

- Let A be a 2x10 matrix
- Let B be a 10x50 matrix
- Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) =

Matrix-chain multiplication

- Let A be a 2x10 matrix
- Let B be a 10x50 matrix
- Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - \# multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - \# multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.

Matrix-chain multiplication

- Let A be a 2x10 matrix
- Let B be a 10x50 matrix
- Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.
- Consider computing **(AB)C**:
  - # multiplications for (AB) =
  - # multiplications for (AB)C =
  - Total multiplications =

Matrix-chain multiplication

- Let A be a 2x10 matrix
- Let B be a 10x50 matrix
- Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.
- Consider computing **(AB)C**:
  - # multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix
  - # multiplications for (AB)C =
  - Total multiplications =

Matrix-chain multiplication

- Let A be a 2x10 matrix
- Let B be a 10x50 matrix
- Let C be a 50x20 matrix
- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.
- Consider computing **(AB)C**:
  - # multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix
  - # multiplications for (AB)C = 2x50x20 = 2000,
  - Total multiplications = 1000 + 2000 = 3000

Matrix-chain multiplication

- The second way is faster than the first!!!
- The multiplication sequence (parenthesization) is important.
- Different parenthesizations will have different number of multiplications for product of multiple matrices.
- Thus, our **goal** is:

  "Given a chain of matrices to multiply, determine the fewest number of multiplications necessary to compute the product."

Matrix-chain multiplication –MCM DP

- Denote $<A_1, A_2, \ldots, A_n>$ by $<p_0, p_1, p_2, \ldots, p_n>$
  - i.e, $A_1(p_0, p_1)$, $A_2(p_1, p_2)$, $\ldots$, $A_i(p_{i-1}, p_i)$, $\ldots$ $A_n(p_{n-1}, p_n)$
- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.
- Let P(n) denote the number of alternative parenthesizations of a sequence of n matrices:

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

- The solution to the recursion is $\Omega(2^n)$.
- So brute-force will not work.

Quiz

Four matrices M1, M2, M3 and M4 of dimensions pxq, qxr, rxs and sxt respectively can be multiplied is several ways with different number of total scalar multiplications. For example, when multiplied as
➢ ((M1 X M2) X (M3 X M4)), the total number of multiplications is pqr + rst + prt.
➢ (((M1 X M2) X M3) X M4), the total number of multiplications is pqr + prs + pst.
If p = 10, q = 100, r = 20, s = 5 and t = 80, then the number of scalar multiplications needed is
• 248000
• 44000
• 19000
• 25000

MCP DP Steps

- Step 1: structure of an optimal parenthesization
  - Let $A_{i..j}$ $(i \leq j)$ denote the matrix resulting from $A_i \times A_{i+1} \times \ldots \times A_j$
  - Any parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must split the product between A_k and A_{k+1} for some k, $(i \leq k < j)$.
  - The cost = # of computing $A_{i..k}$ + # of computing $A_{k+1..j}$ + # $A_{i..k} \times A_{k+1..j.}$
  - If k is the position for an optimal parenthesization, the parenthesization of "prefix" subchain $A_i \times A_{i+1} \times \ldots \times A_k$ within this optimal parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must be an optimal parenthesization of $A_i \times A_{i+1} \times \ldots \times A_k.$
  - $A_i \times A_{i+1} \times \ldots \times A_k \times A_{k+1} \times \ldots \times A_j$

MCP DP Steps

Step 1: Determine the structure of an optimal solution (in this case, a parenthesization).

High-Level Parenthesization for $A_{i..j}$

For any optimal multiplication sequence, at the last step you are multiplying two matrices $A_{i..k}$ and $A_{k+1..j}$ for some k. That is,

$$A_{i..j} = (A_i \cdots A_k)(A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$

Example

$$A_{3..6} = (A_3(A_4 A_5))(A_6) = A_{3..5}A_{6..6}.$$

Here $k = 5$.

MCP DP Steps

Step 1 – Continued: Thus the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

- How do we decide where to split the chain (what is k)?

  (Search all possible values of k)

- How do we parenthesize the subchains $A_{i..k}$ and $A_{k+1..j}$?

  (Problem has optimal substructure property that $A_{i..k}$ and $A_{k+1..j}$ must be optimal so we can apply the same procedure recursively)

MCP DP Steps

Step 1 – Continued:

Optimal Substructure Property: If final "optimal" solution of $A_{i..j}$ involves splitting into $A_{i..k}$ and $A_{k+1..j}$ at final step then parenthesization of $A_{i..k}$ and $A_{k+1..j}$ in final optimal solution must also be optimal for the subproblems "standing alone":

If parenthisization of $A_{i..k}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, leading to a contradiction.

Similarly, if parenthisization of $A_{k+1..j}$ was not optimal we could replace it by a better parenthesization and get a cheaper final solution, also leading to a contradiction.

MCP DP Steps

- Step 2: a recursive relation
  - Let m[i,j] be the minimum number of multiplications for $A_i \times A_{i+1} \times \ldots \times A_j$

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

MCP DP Steps

Step 2: Recursively define the value of an optimal solution.

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$

Proof: Any optimal sequence of multiplication for $A_{i..j}$ is equivalent to some choice of splitting

$$A_{i..j} = A_{i..k} A_{k+1..j}$$

for some k, where the sequences of multiplications for $A_{i..k}$ and $A_{k+1..j}$ also are optimal. Hence

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_k p_j.$$

MCP DP Steps

Step 2 – Continued: We know that, for some k

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j.$$

We don't know what k is, though

But, there are only $j - i$ possible values of k so we can check them all and find the one which returns a smallest cost.

Therefore

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

$(A_1 \times A_2) \times A_3$ $\quad C[1,3]$ $A_1 \times (A_2 \times A_3)$ ✓

$2\times3 \leftrightarrow 3\times4 \quad 4\times2$

$P_0\ P_1 \quad P_1\ P_2 \quad P_2\ P_3$

$C[1,1] \quad C[2,3]$

$C[1,1] \quad \underline{0} \quad \underline{24}$

$C[1,1]\ 2\times3\times4 = 24 ✓ \quad 0 \quad C[3,3]$ ✓

Dim $2\times4 \qquad 4\times2$

$\to \boxed{2\times4\times2 = 16}$ ✓

$\to 36$

$C[1,1] + C[2,3]$

$C[1,2] + C[3,3] + P_0 \times P_2 \times P_3$ ✓

$+ P_0 P_1 P_3$

$i\ K \qquad K+1\ j \quad P_{i-1}$

MCM DP Steps

- Step 3, Computing the optimal cost
  - Recursive algorithm will encounter the same subproblem many times.
  - In tabling the answers for subproblems, each subproblem is only solved once.
  - The second hallmark of DP: overlapping subproblems and solve every subproblem just once.

MCM DP Steps

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Our Table: $m[1..n, 1..n]$.
$m[i, j]$ only defined for $i \leq j$.

The important point is that when we use the equation

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

to calculate $m[i, j]$ we must have already evaluated $m[i, k]$ and $m[k + 1, j]$. For both cases, the corresponding length of the matrix-chain are both less than $j - i + 1$. Hence, the algorithm should fill the table in increasing order of the length of the matrix-chain.

MCM DP Steps

That is, we calculate in the order

$$m[1,2], m[2,3], m[3,4], \ldots, m[n-3, n-2], m[n-2, n-1], m[n-1, n]$$

$$m[1,3], m[2,4], m[3,5], \ldots, m[n-3, n-1], m[n-2, n]$$

$$m[1,4], m[2,5], m[3,6], \ldots, m[n-3, n]$$

$$\vdots$$

$$m[1, n-1], m[2, n]$$

$$m[1, n]$$

MCM DP Steps

- Step 3, Algorithm,
    - array $m[1..n,1..n]$, with $m[i,j]$ records the optimal cost for $A_i \times A_{i+1} \times \ldots \times A_j$.
    - array $s[1..n,1..n]$, $s[i,j]$ records index k which achieved the optimal cost when computing $m[i,j]$.
    - Suppose the input to the algorithm is $p = <p_0, p_1, \ldots, p_n>$.

MCM DP Steps

MATRIX-CHAIN-ORDER (p)

```
1   n ← length[p] − 1
2   for i ← 1 to n
3       do m[i, i] ← 0
4   for l ← 2 to n              ▷ l is the chain length.
5       do for i ← 1 to n − l + 1
6           do j ← i + l − 1
7               m[i, j] ← ∞
8               for k ← i to j − 1
9                   do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                      if q < m[i, j]
11                          then m[i, j] ← q
12                              s[i, j] ← k
13  return m and s
```

MCM DP

When designing a dynamic programming algorithm there are two parts:

1. Finding an appropriate optimal substructure property and corresponding recurrence relation on table items. Example:

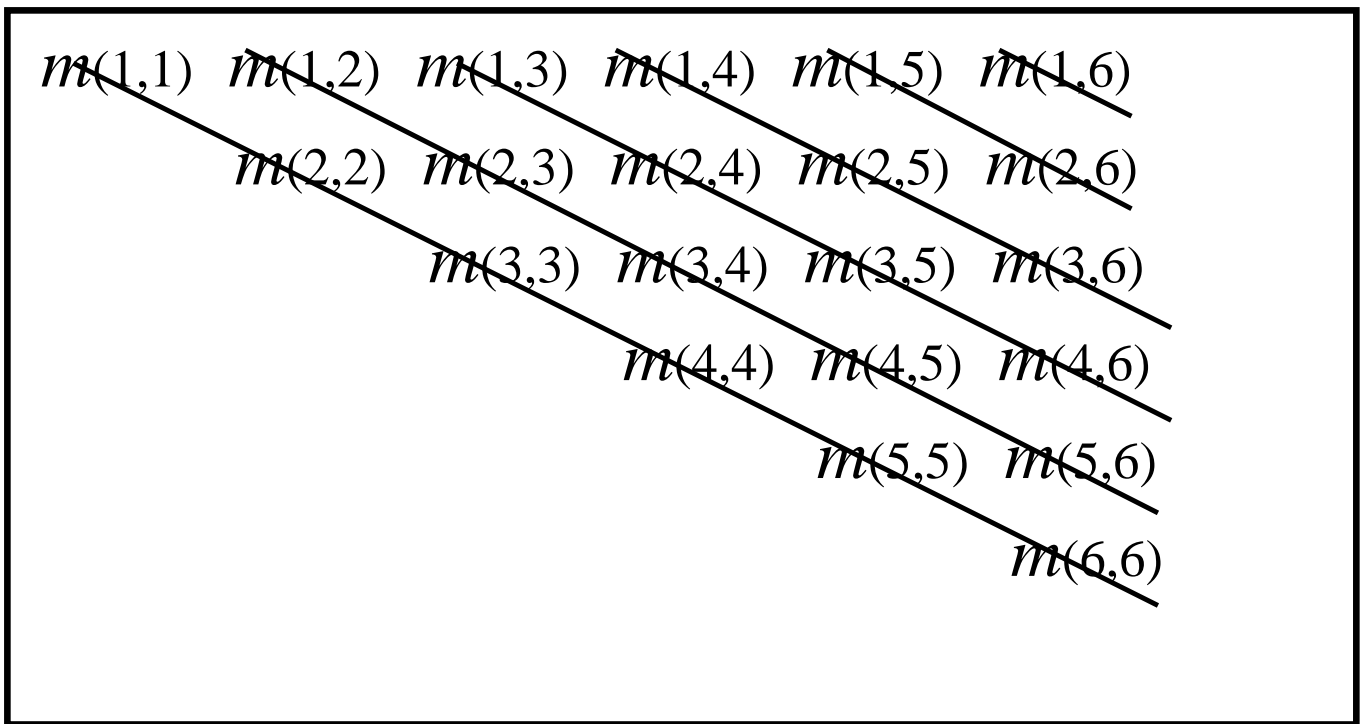$$m[i, j] = \min_{i \leq k < j} \left(m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j \right)$$

2. Filling in the table properly.
   This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated.

   In our example this means that by the time $m[i, j]$ is calculated all of the values $m[i, k]$ and $m[k + 1, j]$ were already calculated.

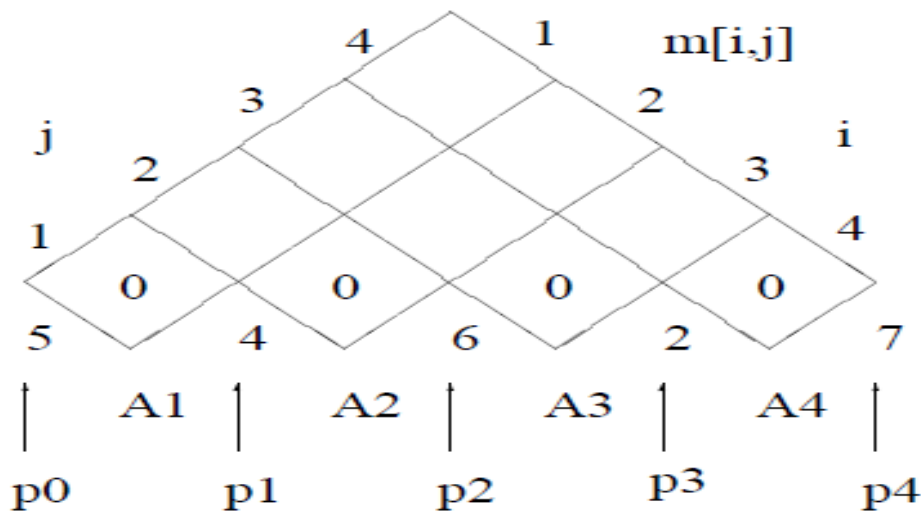# MCM DP—order of matrix computations

$m(1,1)$ $m(1,2)$ $m(1,3)$ $m(1,4)$ $m(1,5)$ $m(1,6)$

$m(2,2)$ $m(2,3)$ $m(2,4)$ $m(2,5)$ $m(2,6)$

$m(3,3)$ $m(3,4)$ $m(3,5)$ $m(3,6)$

$m(4,4)$ $m(4,5)$ $m(4,6)$

$m(5,5)$ $m(5,6)$

$m(6,6)$

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

Example: Given a chain of four matrices A_1, A_2, A_3 and A_4, with $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$ and $p_4 = 7$. Find $m[1,4]$.
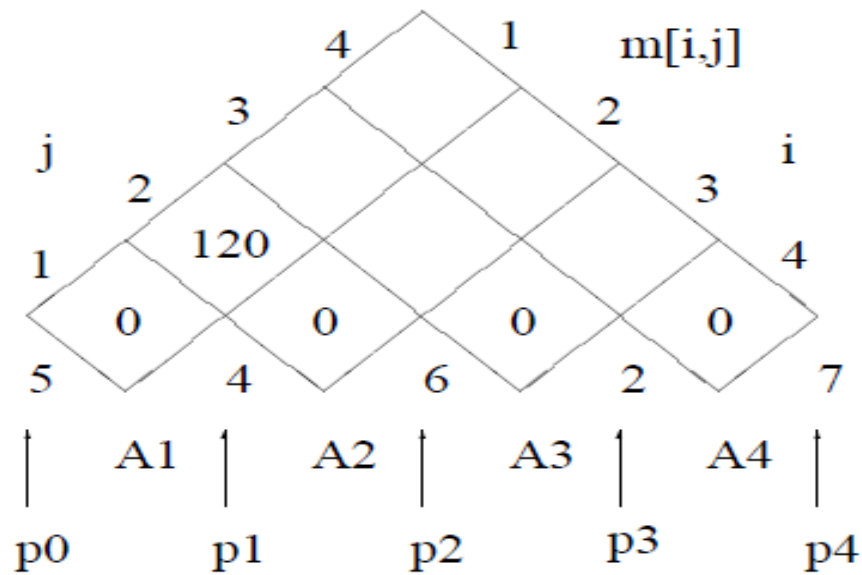
S0: Initialization

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Stp 1: Computing $m[1,2]$ **By definition**

$$m[1,2] = \min_{1 \leq k < 2} (m[1,k] + m[k+1,2] + p_0 p_k p_2)$$
$$= m[1,1] + m[2,2] + p_0 p_1 p_2 = 120.$$

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Stp 2: Computing $m[2,3]$ By definition

$$m[2,3] = \min_{2 \le k < 3} (m[2,k] + m[k+1,3] + p_1p_kp_3)$$
$$= m[2,2] + m[3,3] + p_1p_2p_3 = 48.$$
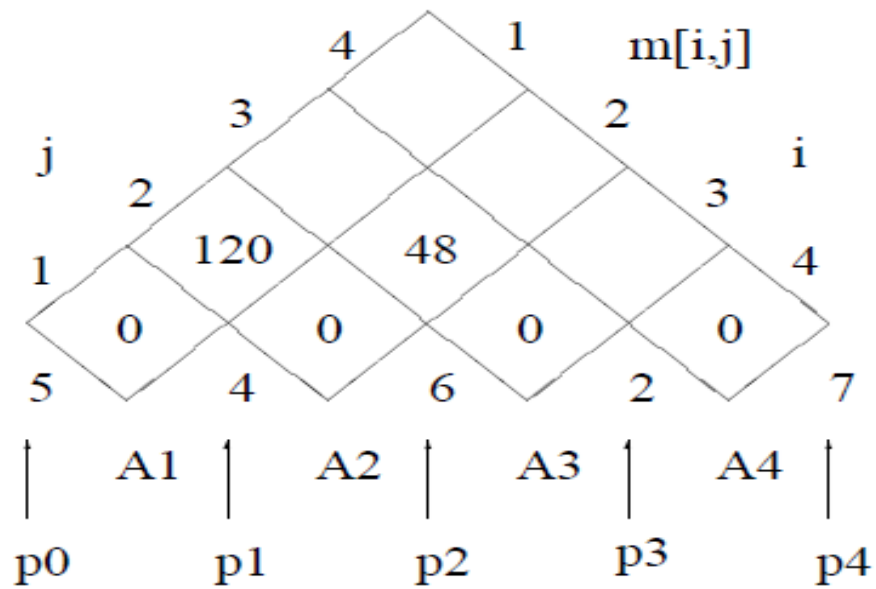
$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$

Stp3: Computing $m[3,4]$ **By definition**

$$\begin{aligned} m[3,4] &= \min_{3 \leq k < 4} (m[3,k] + m[k+1,4] + p_2 p_k p_4) \\ &= m[3,3] + m[4,4] + p_2 p_3 p_4 = 84. \end{aligned}$$

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$
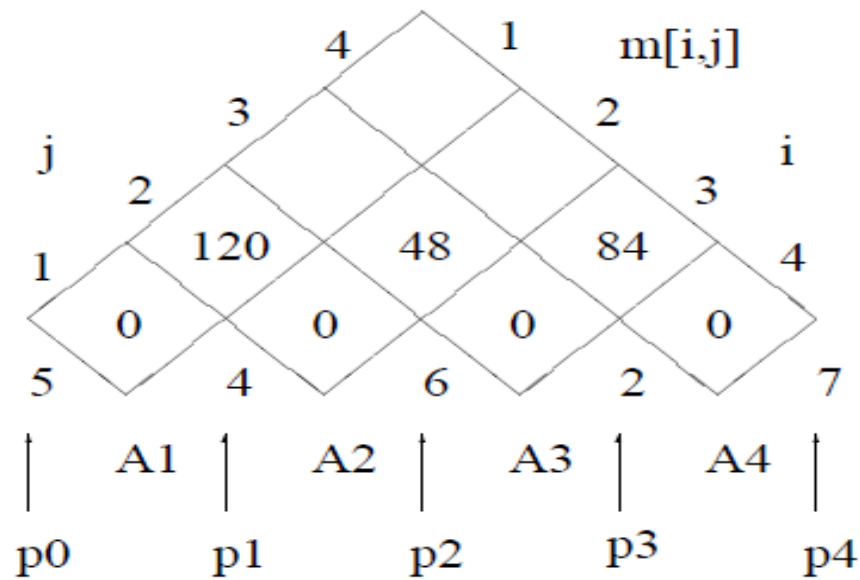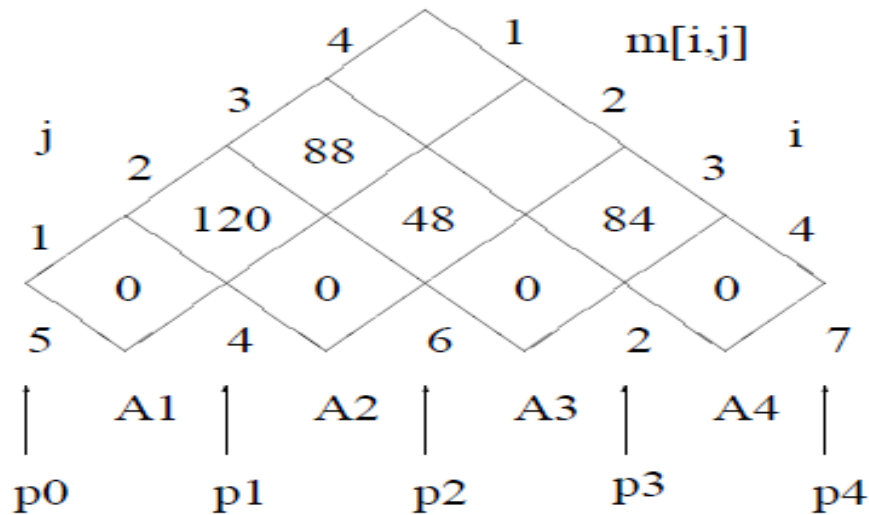
Stp4: Computing $m[1,3]$ By definition

$$m[1,3] = \min_{1 \leq k < 3} (m[1,k] + m[k+1,3] + p_0 p_k p_3)$$

$$= \min \left\{ \begin{array}{l} m[1,1] + m[2,3] + p_0 p_1 p_3 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 \end{array} \right\}$$

$$= 88.$$

m[i,j]

```
              4           1        m[i,j]
          3                   2
 j                88                    i
      2                         3
          120        48        84
 1                                  4
      0          0         0          0
 5        4          6         2          7
 |    A1  |    A2  |    A3  |    A4  |
 p0        p1        p2        p3        p4
```

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i < j \end{cases}$$

Stp5: Computing $m[2,4]$ By definition

$$
\begin{aligned}
m[2,4] &= \min_{2 \le k < 4} (m[2,k] + m[k+1,4] + p_1p_kp_4) \\
&= \min \left\{ \begin{array}{l} m[2,2] + m[3,4] + p_1p_2p_4 \\ m[2,3] + m[4,4] + p_1p_3p_4 \end{array} \right\} \\
&= 104.
\end{aligned}
$$

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i < j \end{cases}$$
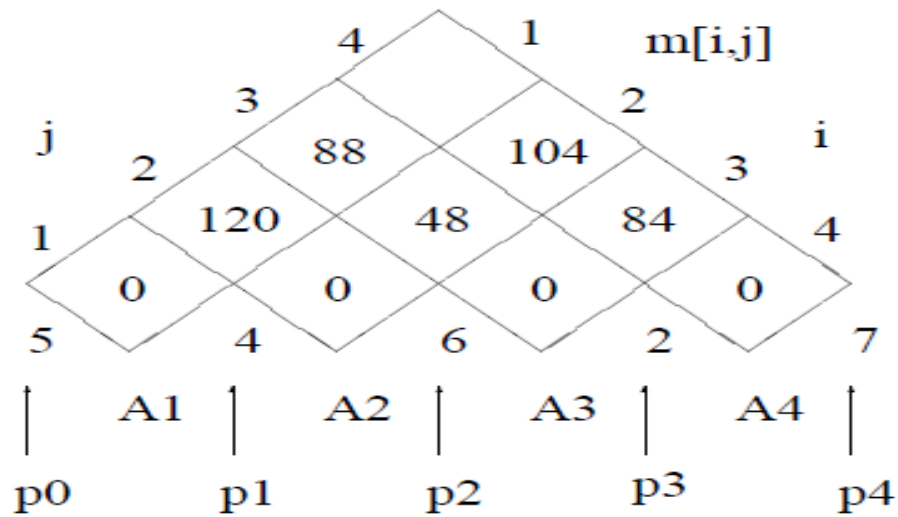
St6: Computing $m[1,4]$ By definition

$$m[1,4] = \min_{1 \le k < 4} (m[1,k] + m[k+1,4] + p_0 p_k p_4)$$

$$= \min \left\{ \begin{array}{l} m[1,1] + m[2,4] + p_0 p_1 p_4 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \end{array} \right\}$$

$$= 158.$$
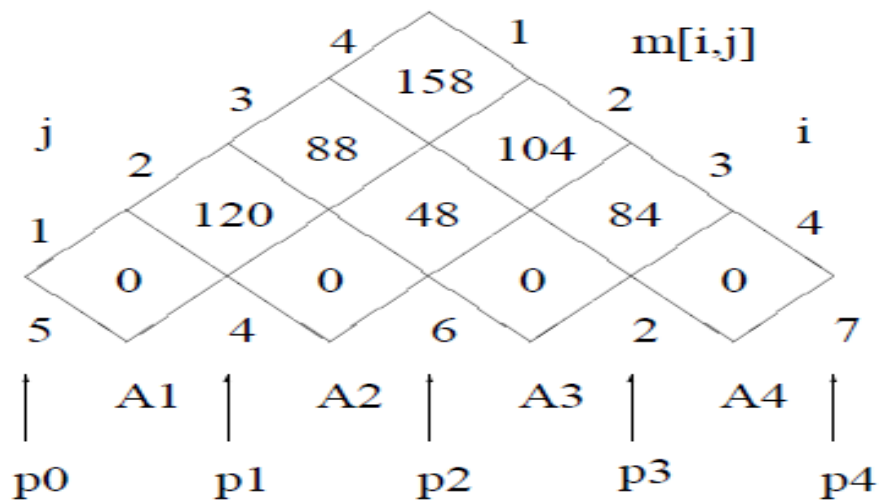
	4	3	2	1
1	3	1	1	0
2	3	2	0	
3	3	0		
4	0			

- (A1A2A3)(A4)
- ((A1)(A2A3))(A4)

MCM DP Steps

- Step 4, constructing a parenthesization order for the optimal solution.
  - Since $s[1..n,1..n]$ is computed, and $s[i,j]$ is the split position for $A_iA_{i+1}\ldots A_j$, i.e, $A_i\ldots A_{s[i,j]}$ and $A_{s[i,j]+1}\ldots A_j$, thus, the parenthesization order can be obtained from $s[1..n,1..n]$ recursively, beginning from $s[1,n]$.

MCM DP Steps

- Step 4, algorithm

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

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

MCM DP Steps

Step 4: Construct an optimal solution from computed information – extract the actual sequence.

Example of Finding the Multiplication Sequence:
Consider $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.

$$s[1, 6] = 3 \quad (A_1 A_2 A_3)(A_4 A_5 A_6)$$
$$s[1, 3] = 1 \quad (A_1 (A_2 A_3))$$
$$s[4, 6] = 5 \quad ((A_4 A_5) A_6)$$

Hence the final multiplication sequence is

$$(A_1 (A_2 A_3))((A_4 A_5) A_6).$$

```
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])
}
```

Complexity: The loops are nested three deep.

Each loop index takes on $\leq n$ values.
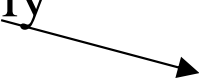
Hence the **time complexity** is $O(n^3)$. Space complexity $\Theta(n^2)$.

Running Time

- #overall subproblems × #choices.
  - In matrix-chain multiplication, $O(n^2) \times O(n) = O(n^3)$

Example

- Show how to multiply this matrix chain optimally

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

MCM DP Example

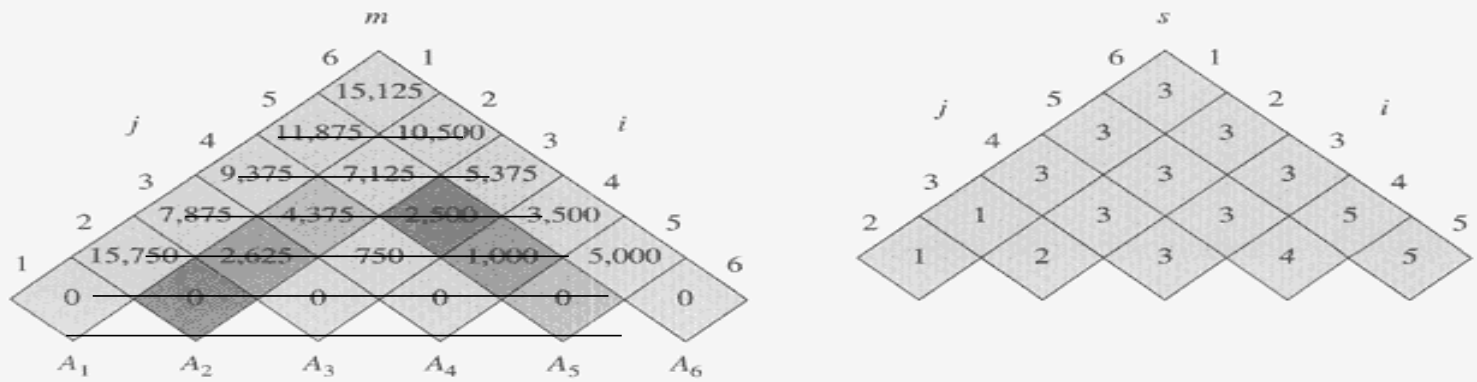**Figure 15.3** The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 , \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 , \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$

$$= 7125 .$$

Example

- Show how to multiply this matrix chain optimally

$$((A1)(A2.A3))((A4)(A5.A6))$$
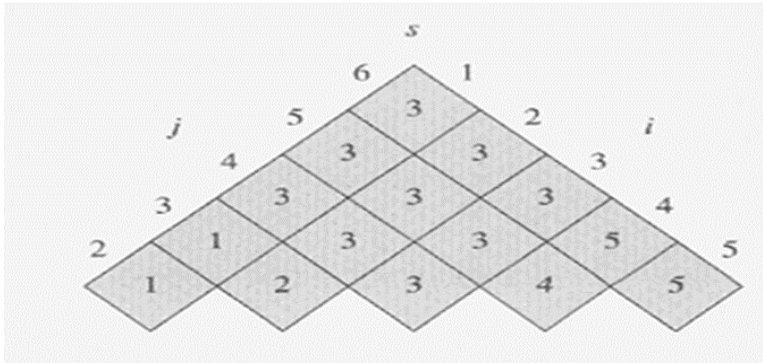
Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Acknowledgments

- Slides adapted from:
  http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/

- https://home.cse.ust.hk/~dekai/271/notes/L12/L12.pdf

- https://www.youtube.com/watch?v=_WncuhSJZyA

- Cormen H. T., Leiserson C. E., Rivest R. L. and Stein C., " Introduction to Algorithms", Chapter 29, Second Ed., PHI, India, 2006

Longest Common Subsequence (LCS)

- DNA analysis/DNA similarity testing, two DNA string comparison.
- DNA string: a sequence of symbols A,C,G,T.
    - S=ACCGGTCGAGCTTCGAAT

- Subsequence (of *X*): is *X* with some symbols left out.
    - *Z*=CGTC is a subsequence of *X*=ACGCTAC.

- Common subsequence *Z* (of *X* and *Y*): a subsequence of *X* and also a subsequence of *Y*.
    - *Z*=CGA is a common subsequence of both *X*=ACGCTAC and *Y*=CTGACA.

- Longest Common Subsequence (LCS): the longest one of common subsequences.
    - *Z'* =CGCA is the LCS of the above *X* and *Y*.

- LCS problem: given $X=<x_1, x_2,…, x_m>$ and $Y=<y_1, y_2,…, y_n>$, find their LCS.

LCS Intuitive Solution –brute force

- List all possible subsequences of X, check whether they are also subsequences of Y, keep the longer one each time.
- Each subsequence corresponds to a subset of the indices $\{1,2,\ldots,m\}$, there are 2^m. So exponential.

LCS DP –Step 1: Optimal Substructure

- Characterize optimal substructure of LCS.
- Theorem : Let $X=<x_1, x_2,\ldots, x_m> (= X_m)$ and

$$Y=<y_1, y_2,\ldots,y_n> (= Y_n)$$

and $Z=<z_1, z_2,\ldots, z_k> (= Z_k)$ 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 the LCS of X_{m-1} and Y_{n-1}.

2. if $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is the LCS of X_{m-1} and Y_n.

3. if $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is the LCS of X_m and Y_{n-1}.

LCS DP –Step 2:Recursive Solution

- What the theorem says:
  - If $x_m = y_n$, find LCS of X_{m-1} and Y_{n-1}, then append x_m.
  - If $x_m \neq y_n$, find LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}, take which one is longer.
- Overlapping substructure:
  - Both LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1} will need to solve LCS of X_{m-1} and Y_{n-1}.

- $c[i, j]$ is the length of LCS of X_i and Y_j .

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

56

Dynamic Programming
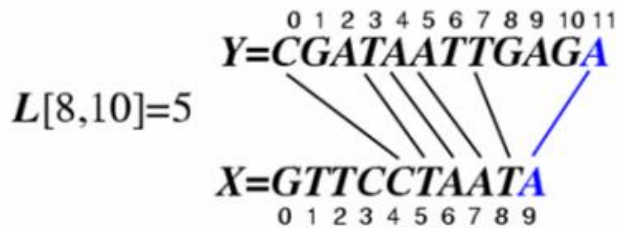
LCS DP –Step 2:Recursive Solution

- What the theorem says:
  - If $x_m = y_n$, find LCS of X_{m-1} and Y_{n-1}, then append x_m.
  - If $x_m \neq y_n$, find LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}, take which one is longer.

Case 1:

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11$$
$$Y=CGATAATTGAGA$$

$L[8,10]=5$

$$X=GTTCCTAATA$$
$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

Case 2:

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$$
$$Y=CGATAATTGAG$$

$L[9,9]=6$
$L[8,10]=5$

$$X=GTTCCTAATA$$
$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

- **$c[i,j]$ is the length of LCS of X_i and Y_j .**

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

LCS DP-- Step 3:Computing the Length of LCS

- $c[0..m, 0..n]$, where $c[i,j]$ is defined as above.

  – $c[m,n]$ is the answer (length of LCS).

- $b[1..m, 1..n]$, where $b[i,j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i,j]$.

  – From $b[m,n]$ backward to find the LCS.

Recursive Algorithm for LCS

$\text{LCS}(x, y, i, j)$
  if $x[i] = y[j]$
      then $c[i, j] \leftarrow \text{LCS}(x, y, i{-}1, j{-}1) + 1$
      else $c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i{-}1, j),$
                               $\text{LCS}(x, y, i, j{-}1) \}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursive Tree

Recursive solution contains a small number of distinct subproblems repeated many times.
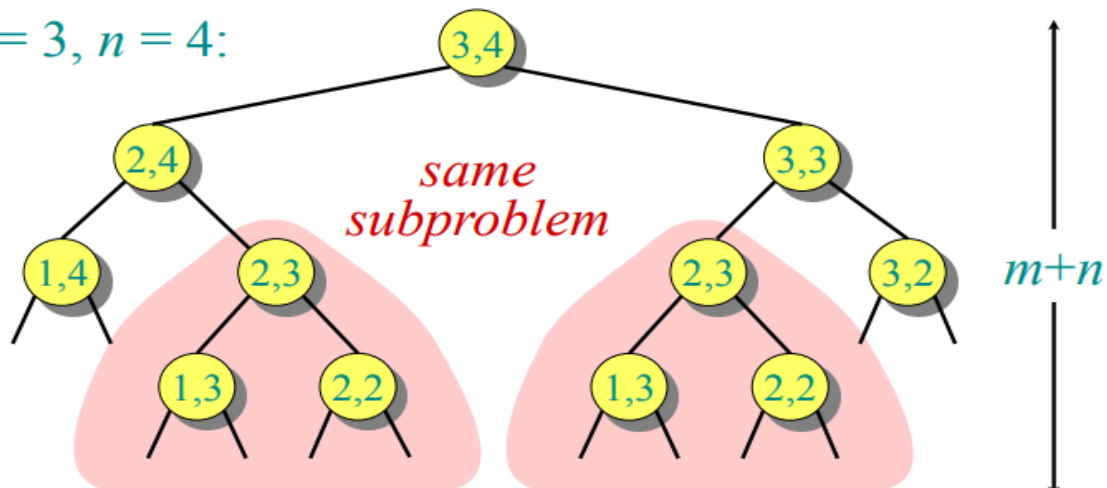
Solution – *memoization*: It is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Time taken is $\Theta(mn)$;

LCS computation example

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

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

61

LCS computation example

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

LCS

Let c_{ij} = length of LCS of $x_1 x_2 ... x_i$ and $y = y_1 y_2 ... y_j$.

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

$$b[i,j] = \begin{cases} \text{"}\nwarrow\text{"} & \text{if } x_i = y_j, \\ \text{"}\uparrow\text{"} & \text{if } x_i \neq y_j \text{ and } c[i-1,j] \geq c[i,j-1], \\ \text{"}\leftarrow\text{"} & \text{if } x_i \neq y_j \text{ and } c[i-1,j] < c[i,j-1]. \end{cases}$$

LCS DP Algorithm

$LCS - Length(X, Y)$

1    $m \leftarrow length[X]$
2    $n \leftarrow length[Y]$
3   **for** $i \leftarrow 1$ **to** m
4   **do** $c[i, 0] \leftarrow 0$
5   **for** $j \leftarrow 0$ **to** n
6   **do** $c[0, j] \leftarrow 0$
7   **for** $i \leftarrow 1$ **to** m
8   **do for** $j \leftarrow 1$ **to** n
9   **do if** $x_i = y_j$
10   **then** $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
11                $b[i, j] \leftarrow$ "↖"
12   **else if** $c[i - 1, j] \geq c[i, j - 1]$
13   **then** $c[i, j] \leftarrow c[i - 1, j]$
14                  $b[i, j] \leftarrow$ "↑"
15   **else** $c[i, j] \leftarrow c[i, j - 1]$
16                  $b[i, j] \leftarrow$ "←"
17   **return** c **and** b

LCS DP –step 4: Constructing LCS

We reconstruct the path by calling Print-LCS(b, X, n, m) and following the arrows, printing out characters of X that correspond to the diagonal arrows (a $\Theta(n + m)$ traversal from the lower right of the matrix to the origin):

```
PRINT-LCS(b, X, i, j)
1   if i == 0 or j == 0
2        return
3   if b[i, j] == "↖"
4        PRINT-LCS(b, X, i − 1, j − 1)
5        print xᵢ
6   elseif b[i, j] == "↑"
7        PRINT-LCS(b, X, i − 1, j)
8   else PRINT-LCS(b, X, i, j − 1)
```

Solve

- X=<BACDB>   Y=<BDCB>

Solution

LCS length

		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m = 5$	B	0	1	2	2	3

$X = \langle BACDB \rangle$

$Y = \langle BDCB \rangle$

$LCS = \langle BCB \rangle$

Solution

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

Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i - 1, j]$, $c[i, j - 1]$, and $c[i - 1, j - 1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "$\nwarrow$" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Exercise

Determine LCS in (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0)

Acknowledgments

- ## Slides adapted from:
  http://www.facweb.iitkgp.ac.in/~sourav/Lecture-12.pdf
- https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/LCS.pdf

Finding Shortest Path in Multistage Graph using Dynamic Programming

Multistage Graph

- To find the **shortest path between the source vertex s and the destination vertex t.**
- A multistage graph is a directed graph which is divided into stagesV1, V2, ….
- Vertices from one stage are connected to vertices of next stage (no edges between vertices of the same stage and from a vertex of current stage to previous stage).
- The first and the last stage have single vertex.

Multistage graph

G(V , E)
<u , v> u in v(i) and v in v(i+1)
|v1|=|vk|=1

c(i,j) is length of edge <i , j>
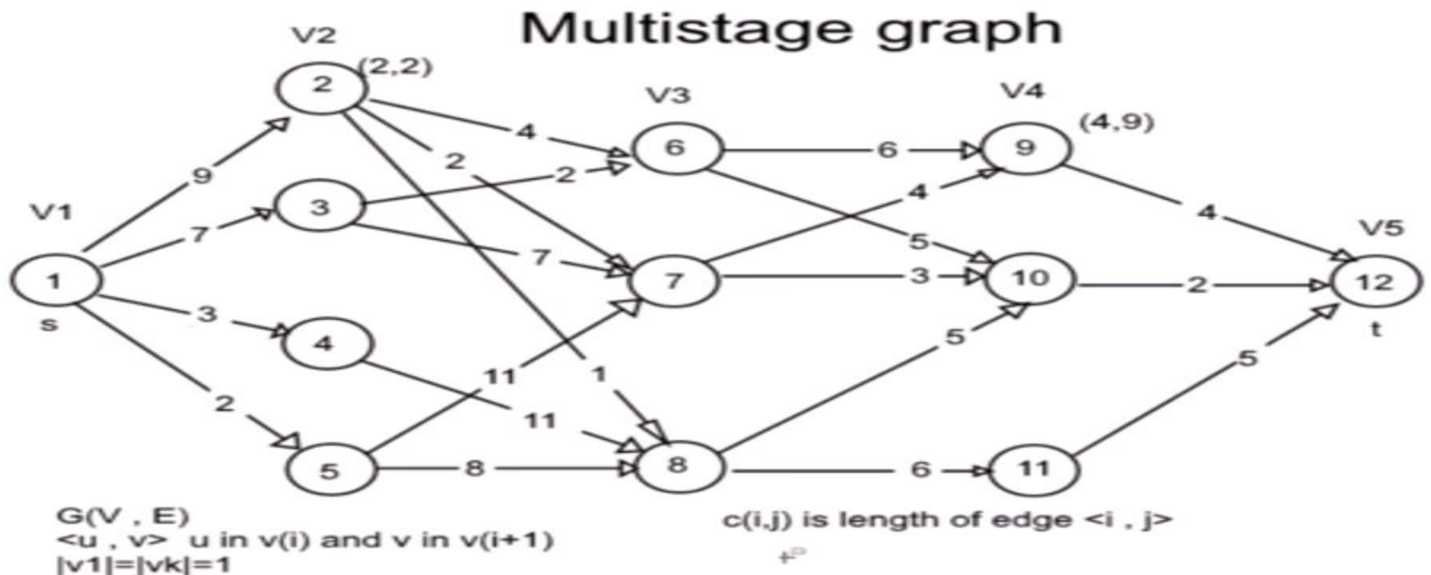
Applying Greedy approach to solve

Multistage graph
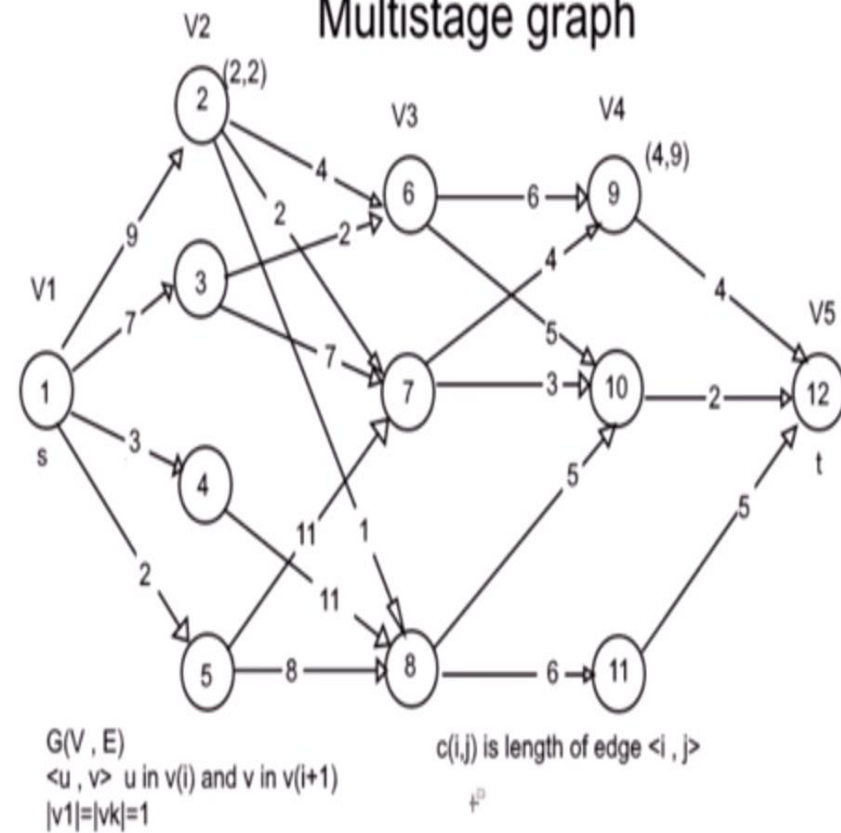
- Greedy Choice 1:
- Edge:(1,5) (5,8) (8,10) (10,12)
- Cost: 2 + 8 + 5 + 2 =17
- Choice 2:
- Edge:(1,2) (2,7) (7,10) (10,12)
- Cost: 9 + 2 + 3 + 2 =16
- Greedy choice fails

Applying Brute force to solve

Brute Force: Enumerate all possible paths
And find the minimum cost path.

Multistage graph

G(V , E)
<u , v> u in v(i) and v in v(i+1)
|v1|=|vk|=1

c(i,j) is length of edge <i , j>
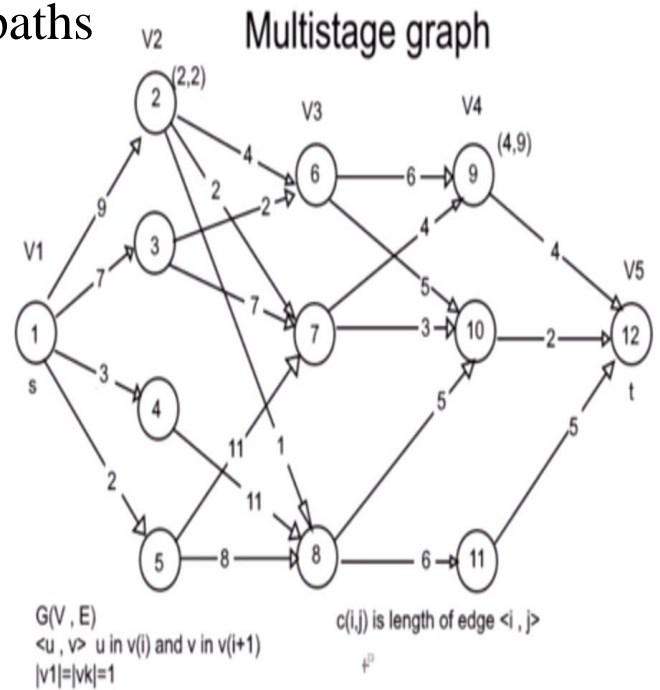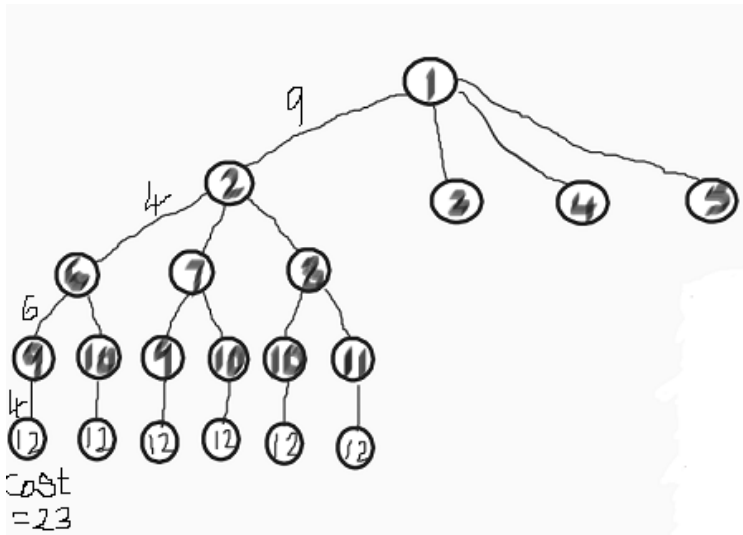
Solving using Dynamic Programming

- Forward approach
- Backward approach

Solving using Dynamic Programming

Multistage graph

G(V , E)
<u , v> u in v(i) and v(i+1)
|v1|=|vk|=1

c(i,j) is length of edge <i , j>

Solving using Dynamic Programming

Multistage graph

G(V , E)
<u , v> u in v(i) and v in v(i+1)
|v1|=|vk|=1
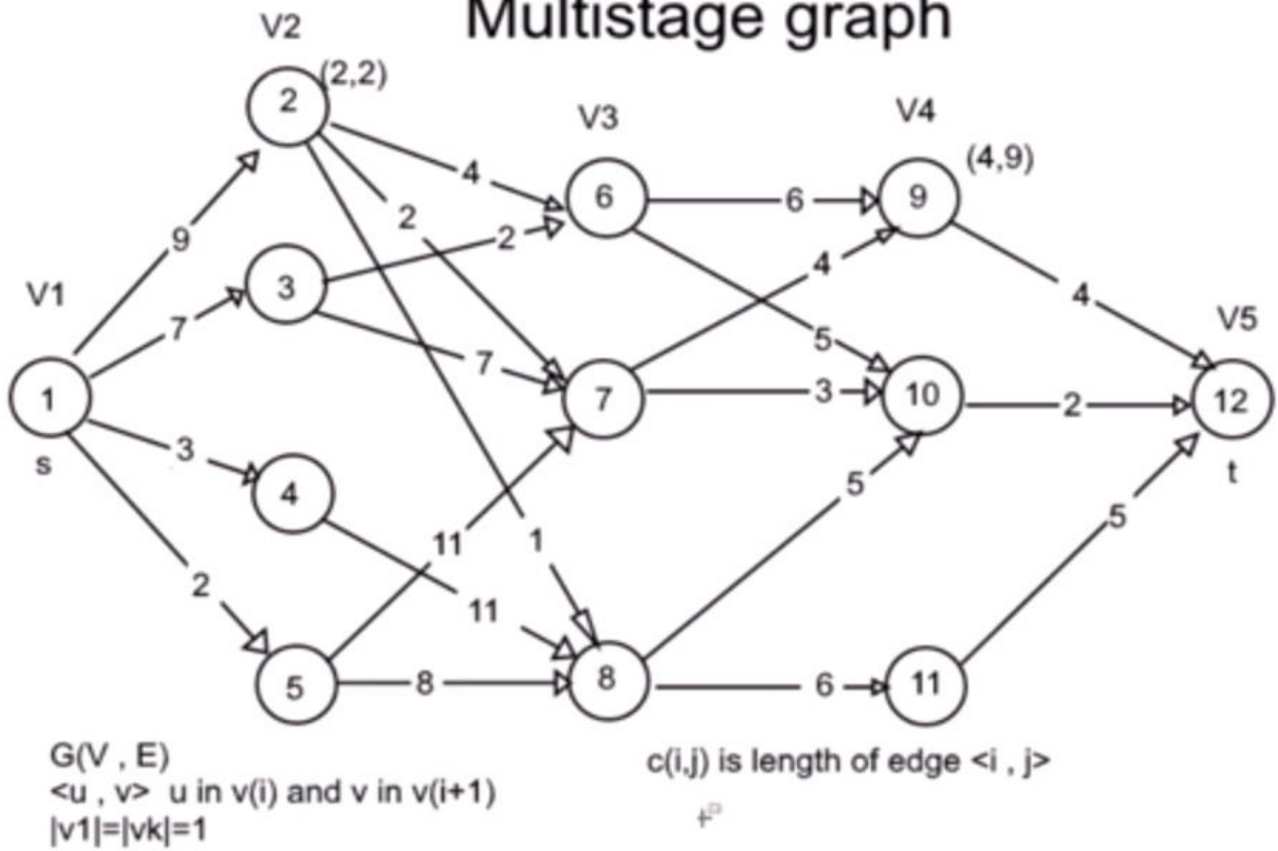
c(i,j) is length of edge <i , j>
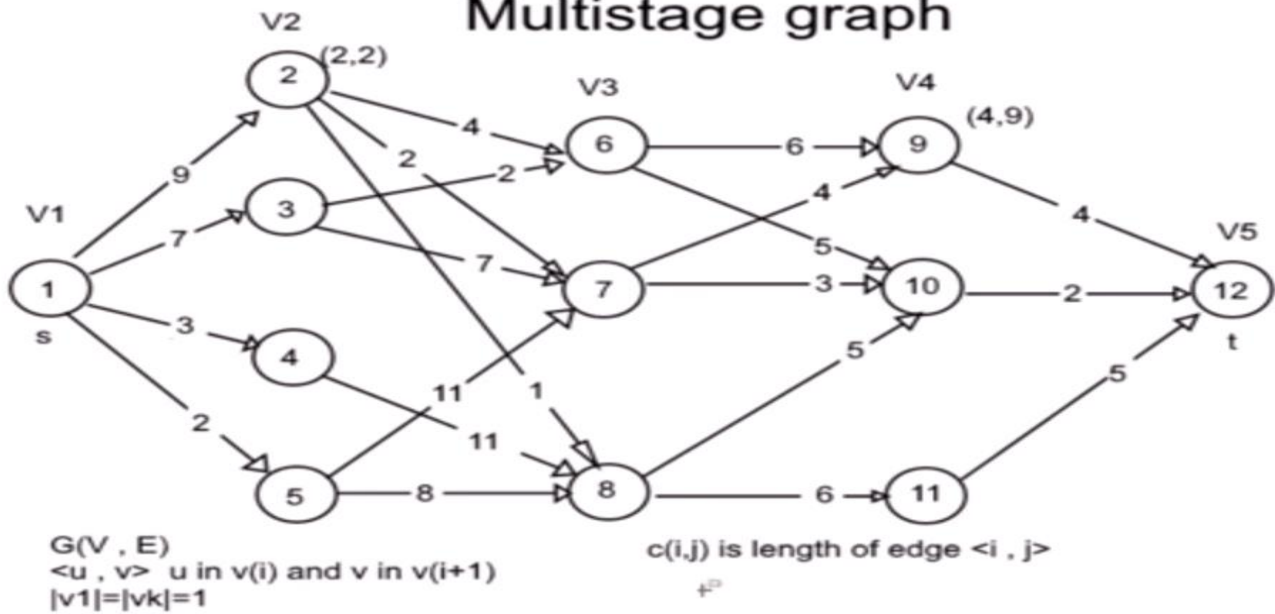
cost(i , j) = min{ c(j , l) + cost(i+1, l) }

stage

node

legth of edge <j , l>
if there is no edge infinity

Multistage graph

cost(4,9) = c(9,12) = 4
cost(4,10) = c(10,12) = 2
cost(4,11) = c(11,12) = 5

path : 1 -> 2 -> 7 -> 10 -> 12
1 -> 3 -> 6 -> 10 -> 12

cost(3,6) = min {c(6,9) + cost(4,9),c(6,10) + cost(4,10)} = min { 6+ 4, 5+2} = 7
cost(3,7) = min{c(7,9)+cost(4,9), c(7,10) + cost(4,10)} = min{4 + 4, 3 + 2} = 5
cost(3,8) = min{c(8,10)+ cost(4,10), c(8,11)+cost(4,11)} = min{5+2, 6+5}=7

Multistage graph

$$cost(2,2) = min(c(2,6)+cost(3,6), c(2,7)+cost(3,7), c(2,8)+cost(3,8)) = min(4+7,2+5,1+7) = 7$$

$$cost(2,3) = min(c(3,6)+cost(3,6), c(3,7)+cost(3,7)) = min(2+7, 7+5) = 9$$

$$cost(2,4) = min(c(4,8)+cost(3,8)) = 11+7 = 18$$

$$cost(2,5) = min(c(5,7)+cost(3,7), c(5,8)+cost(3,8)) = min(11+5,8+7) = 15$$

cost(4,9) = c(9,12) = 4
cost(4,10) = c(10,12) = 2
cost(4,11) = c(11,12) = 5

cost(3,6) = min {c(6,9) + cost(4,9),c(6,10) + cost(4,10)} = min { 6+ 4, 5+2} = 7
cost(3,7) = min{c(7,9)+cost(4,9), c(7,10) + cost(4,10)} = min{4 + 4, 3 + 2} = 5
cost(3,8) = min{c(8,10)+ cost(4,10), c(8,11)+cost(4,11)} = min{5+2, 6+5}=7

cost(2,2) =min{c(2,6)+cost(3,6) , c(2,7) +cost(3,7), c(2,8)+cost(3,8)} = min{4+7,2+5,1+7} = 7
cost(2,3) = min{c(3,6)+cost(3,6), c(3,7) +cost(3,7)} = min{2+7, 7+5} = 9
cost(2,4)= min{c(4,8) +cost(3,8)} = 11+7 = 18
cost(2,5) = min{c(5,7)+cost(3,7), c(5,8) +cost(3,8)} = min{11+5,8+7} =15

vertex	1	2	3	4	5	6	7	8	9	10	11	12
Cost		7	9	18	15	7	5	7	4	2	5	0
d-destination		7	6	8	8	10	10	10	12	12	12	12

Solving using Dynamic Programming: Forward approach

$$cost(1,1) = \min\{ c(1,2) + cost(2,2) , c(1,3) + cost(2,3)$$
$$c(1,4) + cost(2,4), c(1,5) + cost(2,5)\}$$

$$= \min\{9 + 7, 7 + 9, 3 + 18, 2 + 15\} = 16 \quad d(1,1) = 2,3$$

vertex	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d-destination	2/3	7	6	8	8	10	10	10	12	12	12	12

Multistage Graph pseudo code : forward approach

```
1    Algorithm FGraph(G, k, n, p)
2    // The input is a k-stage graph G = (V, E) with n vertices
3    // indexed in order of stages. E is a set of edges and c[i, j]
4    // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5    {
6        cost[n] := 0.0;
7        for j := n − 1 to 1  step −1 do
8        { // Compute cost[j].
9            Let r be a vertex such that ⟨j, r⟩ is an edge
10           of G and c[j, r] + cost[r] is minimum;
11           cost[j] := c[j, r] + cost[r];
12           d[j] := r;
13       }
14       // Find a minimum-cost path.
15       p[1] := 1; p[k] := n;
16       for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17   }
```

Solving using Dynamic Programming: Backward approach

- bcost(i,j) : Minimum cost path from vertex s to vertex j in Vi.
- $bcost(i,j) = \min\{bcost(i-1, k) + c(k,j)\}$
- $\quad k \in Vi\text{-}1$
- $bcost(2,2) = 9$
- $bcost(2,3) = 7$
- $bcost(2,4) = 3$
- $bcost(2,5) = 2$
- $bcost(3,6) = \min(bcost(2,2)+c(2,6),$
- $\quad\quad\quad bcost(2,3)+c(3,6)\}$
- $\quad = \min\{9+4, 7+2\}=9$
- $bcost(3,7) = \min(bcost(2,2)+c(2,7),$
- $\quad\quad bcost(2,3)+c(3,7)$
- $\quad\quad\quad bcost(2,5)+c(2,7)\}$
- $\quad = \min\{9+2, 7+7, 2+11\}=11$

Multistage graph

V2

(2,2)

V3 V4

V1

(4,9)

V5

G(V , E)
<u , v> u in v(i) and v in v(i+1)
|v1|=|vk|=1

c(i,j) is length of edge <i , j>
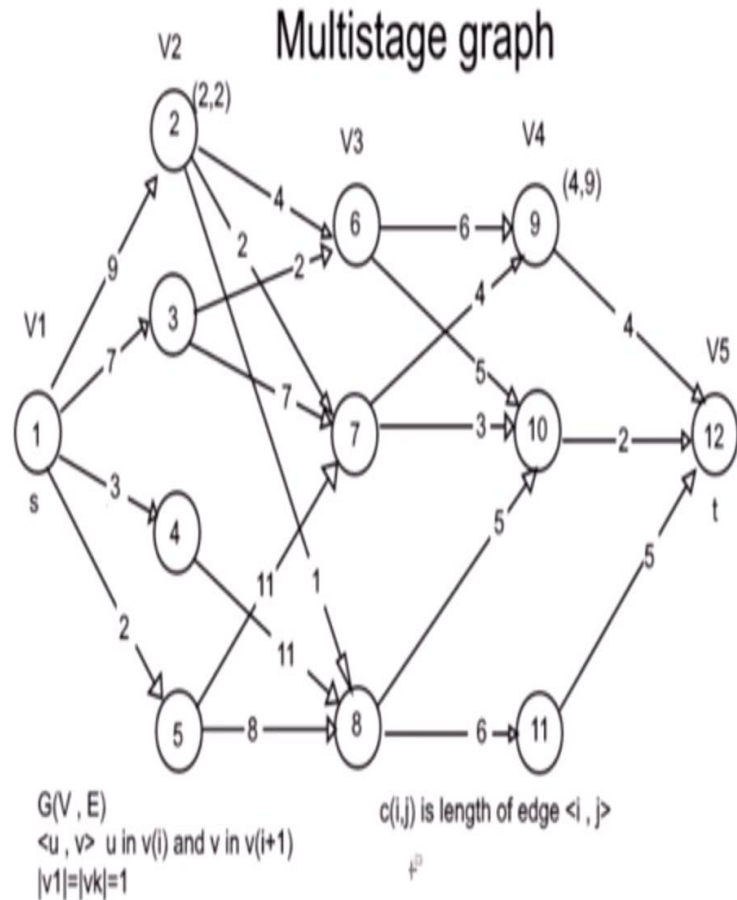
Solving using Dynamic Programming: Backward approach

$$bcost(3, 7) = 11$$
$$bcost(3, 8) = 10$$
$$bcost(4, 9) = 15$$
$$bcost(4, 10) = 14$$
$$bcost(4, 11) = 16$$
$$bcost(5, 12) = 16$$

Multistage Graph pseudo code: backward approach

```
1    Algorithm BGraph(G, k, n, p)
2    // Same function as FGraph
3    {
4        bcost[1] := 0.0;
5        for j := 2 to n do
6        { // Compute bcost[j].
7            Let r be such that ⟨r, j⟩ is an edge of
8            G and bcost[r] + c[r, j] is minimum;
9            bcost[j] := bcost[r] + c[r, j];
10           d[j] := r;
11       }
12       // Find a minimum-cost path.
13       p[1] := 1; p[k] := n;
14       for j := k − 1 to 2 do p[j] := d[p[j + 1]];
15   }
```

Solve

Find minimum cost path from s to t in the multistage graph given below using:

a.Forward approach

b.Backward approach
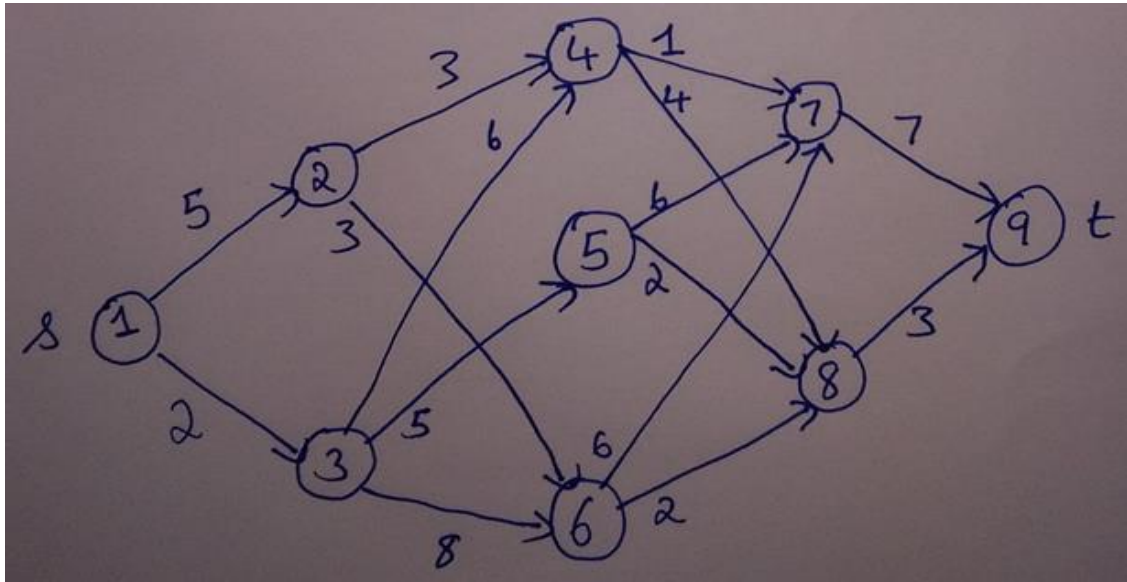
Longest Increasing Subsequence (LIS)

Longest Increasing Subsequence (LIS)

- Given a sequence **A** of size **N**, find the **length** of the **longest increasing subsequence** from a given sequence .
  The longest increasing subsequence means to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique.
- **Note:** Duplicate numbers are not counted as increasing subsequence.

Longest Increasing Subsequence (LIS)

- The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

- For example, the length of LIS for

- {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}

  or   {10, 22, 33, 41, 60, 80}

- .

Longest Increasing Subsequence (LIS)

```
Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3
The longest increasing subsequence is 3, 10, 20


Input: arr[] = {3, 2}
Output: Length of LIS = 1
The longest increasing subsequences are {3} and {2}


Input: arr[] = {50, 3, 10, 7, 40, 80}
Output: Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

Longest Increasing Subsequence (LIS)

- **Method 1:** Recursion.

- ***Optimal Substructure:*** Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

-

Longest Increasing Subsequence (LIS)

- Then, L(i) can be recursively written as:
- **L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i];**
  or L(i) = 1, if no such j exists.
- To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.
- Formally, the length of the longest increasing subsequence ending at index i, will be 1 greater than the maximum of lengths of all longest increasing subsequences ending at indices before i, where arr[j] < arr[i] (j < i). Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems. The recursive tree is given below

```
Input   : arr[] = {3, 10, 2, 11}
f(i): Denotes LIS of subarray ending at index 'i'

(LIS(1)=1)

        f(4)   {f(4) = 1 + max(f(1), f(2), f(3))}
     /    |    \
 f(1)   f(2)   f(3) {f(3) = 1, f(2) and f(1) are > f(3)}
         |     |  \
       f(1)   f(2)   f(1) {f(2) = 1 + max(f(1)}
               |
             f(1) {f(1) = 1}
```

*

Longest Increasing Subsequence (LIS)

Complexity Analysis:

- **Time Complexity:** The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems as explained in the recursive tree diagram.

-

Longest Increasing Subsequence (LIS)

- **Method 2:** Dynamic Programming.

- We can see that there are many subproblems in the above recursive solution which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

Longest Increasing Subsequence (LIS)

- Input : arr[] = {3, 10, 2, 11} LIS[] = {1, 1, 1, 1} (initially) **Iteration-wise simulation :**

- arr[2] > arr[1] {LIS[2] = max(LIS [2], LIS[1]+1)=2}

- arr[3] < arr[1] {No change}

- arr[3] < arr[2] {No change}

- arr[4] > arr[1] {LIS[4] = max(LIS [4], LIS[1]+1)

$$=max(1,1+1)=2\}$$

- arr[4] > arr[2] {LIS[4] = max(LIS [4], LIS[2]+1)

$$=max(1,2+1)=3\}$$

- arr[4] > arr[3] {LIS[4] = max(LIS [4], LIS[3]+1)

$$=max(1,1+1)=2\}$$

- We can avoid recomputation of subproblems by using tabulation as shown next:

Longest Increasing Subsequence (LIS)

- Input : arr[] = {3, 10, 2, 11}
- We can avoid recomputation of subproblems by using tabulation as shown next:

arr[ ]	3	10	2	11
LIS[ ]	1	2	1	3

Dynamic Programming implementation

- /* lis() returns the length of the longest
- increasing subsequence in arr[] of size n */
- int lis(int arr[], int n)
- {   int lis[n];
-         lis[0] = 1;
-         /* Compute optimized LIS values in
-         bottom up manner */
-         for (int i = 0; i < n; i++)
-                 lis[i] = 1;
-         for (int i = 1; i < n; i++)
-         {   for (int j = 0; j < i; j++)
-                         if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
-                                 lis[i] = lis[j] + 1;
-         }
-         // Return maximum value in lis[]
-         return *max_element(lis, lis+n);
- } *

Longest Increasing Subsequence (LIS)

- **#** Dynamic programming Python implementation of LIS problem
- # lis returns length of the longest increasing subsequence in arr of size n
- def lis(arr):
-         n = len(arr)
-         # Declare the list (array)for LIS and initialize LIS values for all indexes
-         lis = [1]*n
-         # Compute optimized LIS values in bottom up manner
-         for i in range (1 , n):
-                 for j in range(0 , i):
-                         if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
-                         lis[i] = lis[j]+1
-

Longest Increasing Subsequence (LIS)

- # Initialize maximum to 0 to get
- # the maximum of all LIS
- maximum = 0
- # Pick maximum of all LIS values
- for i in range(n):
- maximum = max(maximum , lis[i])
- return maximum
- # end of lis function
- # Driver program to test above function
- arr = [10, 22, 9, 33, 21, 50, 41, 60]
- print "Length of lis is", lis(arr)

Output: Length of lis is 5
Time Complexity: $O(n^2)$.

Longest Increasing Subsequence (LIS)

-

Arr[ ]	10	22	9	33	21	50	41	60
LIS	1	`						

Longest Increasing Subsequence (LIS)

-

Arr[ ]	10	22	9	33	21	50	41	60
LIS	1	2	1	3	2	4	4	5

Acknowledgments

- https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/

Dynamic programming

Rod cutting

Dynamic programming

Approach-
- solve sub problems,
- store results,
- use the results while solving bigger instances of the problem with out re-computing

DP vs DC

Divide and conquer is aimed at dividing the problem into smaller instances, solve instances and combine to get final solution.

Dynamic Programming, divides, solves, combines but there could be overlaps, memorizes earlier solutions and uses it

Rod Cutting

Problem: Given a rod of length 'n' and a table of prices P_i, sell it by cutting it into pieces so as to maximize revenue r_n

Example: rod of 4 inch length
- Cut into pieces of length
- $P_1 + P_1 + P_1 + P_1 = 1+1+1+1=4$
- $P_2 + P_2 = 5+5=10$
- $P_1 + P_1 + P_2 = 1+1+5=7$ **OR** $P_1 + P_2 + P_1 = 1+5+1=7$ **OR** $P_2 + P_1 + P_1 = 5+1+1=7$
- $P_1 + P_3 = 1+8=9$ **OR** $P_3 + P_1 = 8+1=9$
- $P_4 = 9$
- **Price of the Optimal cut : $P_2 + P_2 = 5+5=10$**

Length i	1	2	3	4	5	6	7	8	9	10
Price P_i	1	5	8	9	10	17	17	20	24	30

Rod cutting

In general, rod of length 'n' can be cut in 2^{n-1} different ways, since we can choose cutting, or not cutting, at all distances i ($1 \leq i \leq n-1$) from the left end

$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots\ldots, r_{n-1} + r_1)$

Recursive approach is

$r_n = \max(p_i + r_{n-i})$ $1 \leq i \leq n$

Rod cutting -4 inch rod example
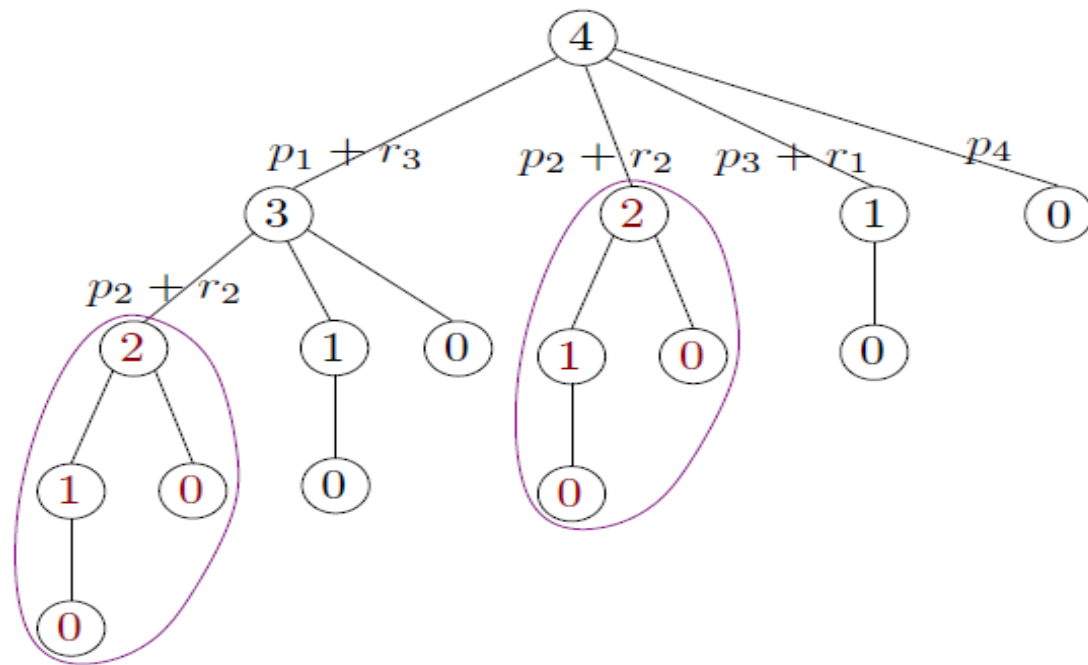
Recursive top down implementation

if $n = 0$ **then**
   | **return** 0;
end
$q = -\infty$;
for $i = 1$ **to** n **do**
   | $q = \max(q, p[i] + \text{Cut-Rod}(p, n - i))$;
end
return q;

DP for rod cutting

- After solving smaller instances of problem, store the values, it can be used to solve the bigger instances
- At the cost of memory speed up execution
- Two approaches- Top down or Bottom up
- Both have same time complexity $O(n^2)$

$$C(i) = \max\{V_k + C(i-k)\}$$
$$1 <= k <= i$$

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20
Len(i)	1	2	3	4	5	6	7	8
Optimal	1	5	8	10				

$$C(1) = 1$$

$$C(2) = \max \begin{cases} V_1 + C(1) = 1 + 1 = 2 \\ V_2 = 5 \end{cases}$$

$$C(3) = \max \begin{cases} V_1 + C(2) = 1 + 5 = 6 \\ V_2 + C(1) = 5 + 1 = 6 \\ V_3 = 8 \end{cases}$$

$$C(4) = \max \begin{cases} V_1 + C(3) = 1 + 8 = 9 \\ V_2 + C(2) = 5 + 5 = 10 \\ V_3 + C(1) = 8 + 1 = 9 \\ V_4 = 9 \end{cases}$$

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20
Len(i)	1	2	3	4	5	6	7	8
Optimal	1	5	8	10	13	17	18	22

$$C(5) = \max \begin{cases} V_1 + C(4) = 1 + 10 = 11 \\ V_2 + C(3) = 5 + 8 = 13 \\ V_3 + C(2) = 8 + 5 = 13 \\ V_4 + C(1) = 9 + 1 = 10 \\ V_5 = 10 \end{cases}$$

$$C(6) = \max \begin{cases} V_1 + C(5) = 1 + 13 = 14 \\ V_2 + C(4) = 5 + 10 = 15 \\ V_3 + C(3) = 8 + 8 = 16 \\ V_4 + C(2) = 9 + 5 = 14 \\ V_5 + C(1) = 10 + 1 = 11 \\ V_6 = 17 \end{cases}$$

$$C(7) = \max \begin{cases} V_1 + C(6) = 1 + 17 = 18 \\ V_2 + C(5) = 5 + 13 = 18 \\ V_3 + C(4) = 8 + 10 = 18 \\ V_4 + C(3) = 9 + 8 = 17 \\ V_5 + C(2) = 10 + 5 = 15 \\ V_6 + C(1) = 17 + 1 = 18 \\ V_7 = 17 \end{cases}$$

$$C(8) = \max \begin{cases} V_1 + C(7) = 1 + 18 = 19 \\ V_2 + C(6) = 5 + 17 = 22 \\ V_3 + C(5) = 8 + 13 = 21 \\ V_4 + C(4) = 9 + 10 = 19 \\ V_5 + C(3) = 10 + 8 = 18 \\ V_6 + C(2) = 17 + 5 = 22 \\ V_7 + C(1) = 17 + 1 = 18 \\ V_8 = 20 \end{cases}$$

Length	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20
Len(i)	1	2	3	4	5	6	7	8
Optimal	1	5	8	10	13	17	18	22

$$C(6) = \max \begin{cases} V_1 + C(5) = 1 + 13 = 14 \\ V_2 + C(4) = 5 + 10 = 15 \\ V_3 + C(3) = 8 + 8 = 16 \\ V_4 + C(2) = 9 + 5 = 14 \\ V_5 + C(1) = 10 + 1 = 11 \\ V_6 = 17 \end{cases}$$

$$C(8) = \max \begin{cases} V_1 + C(7) = 1 + 18 = 19 \\ V_2 + C(6) = 5 + 17 = 22 \\ V_3 + C(5) = 8 + 13 = 21 \\ V_4 + C(4) = 9 + 10 = 19 \\ V_5 + C(3) = 10 + 8 = 18 \\ V_6 + C(2) = 17 + 5 = 22 \\ V_7 + C(1) = 17 + 1 = 18 \\ V_8 = 20 \end{cases}$$

The optimal way to cut the rod of size 8 is to have a cut of size two and a cut of size six.

Rod Cutting

Without dynamic programming, the problem has a complexity of $O(2^n)$!

For a rod of length 8, there are 128 (or 2^{n-1}) ways to cut it!

With dynamic programming, and this top down approach, the problem is reduced to $O(n^2)$

Top down approach

MEMOIZED-CUT-ROD(p,n)

Let r[0…n] be a new array

For i=0 to n

   r[i]=-∞

Return MEMOIZED-CUT-ROD-AUX(p,n,r)

Top down approach(contd.)

MEMOIZED-CUT-ROD-AUX(p,n,r)

if r[n]>=0 // Checks to see if the desired value is already computed/known

    return r[n]

if n==0

  q=0

else

  q=$-\infty$

  for i=1 to n

    q=max(q, p[i]+MEMOIZED-CUT-ROD-AUX(p,n-i,r)

r[n]=q

return q

- The top-down approach has the advantages that it is easy to write given the recursive structure of the problem, and only those subproblems that are actually needed will be computed.
- It has the disadvantage of the overhead of recursion.

Bottom up approach

BOTTOM-UP-CUT-ROD(p,n)
Let r[0…n] be a new array
r[0]=0
for j=1 to n
 q=-∞
 for i=1 to j
  q=max(q, p[i]+r[j-i])
 r[j]=q
return r[n]

Extended-bottom up- approach

- EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1. let r[0..n] and s[0..n] be new arrays
2. r[0] = 0
3. for j = 1 to n
4.   q = -INF
5.   for i = 1 to j
6.   if q < p[i] + r[j-i]
7.   q = p[i] + r[j-i]
8.   s[j] = i
9.   r[j] = q
10. // Print optimal cuts :We then trace the choices made back through the table s with this procedure
11. i = n
12.   while i > 0
13.   print s[i]
14.   i = i - s[i]
15.   return r and s

- The bottom-up approach requires extra thought to ensure we arrange to solve the subproblems before they are needed.
- (Here, the array reference $r[j - i]$ ensures that we only reference subproblems smaller than j, the one we are currently working on.)

Printing result

// Print optimal cuts :We then trace the choices made
     back through the table *s* with this procedure

11. i = n

12. while i > 0

13.   print s[i]

14.   i = i - s[i]

15. return r and s

Exercise

I	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]											

Exercise

I	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

Acknowledgements

- https://www.youtube.com/watch?v=ElFrskby_7M

Dynamic programming

Egg Dropping Puzzle

Egg Dropping Puzzle

- Input

n eggs, building with k floors

- *output*

Find the number of attempts it takes to find
  out from which floor the egg will break.

OR

finding threshold/critical/pivot floor

Assumptions

- Suppose that we wish to know which stories in a 20-story building are safe to drop eggs from, and which will cause the eggs to break on landing.
- We make a few assumptions:

➢ An egg that survives a fall can be used again.

➢ A broken egg must be discarded.

➢ The effect of a fall is the same for all eggs.

➢ If an egg breaks when dropped, then it would break if dropped from a higher floor.

➢ If an egg survives a fall then it would survive a shorter fall.

Egg Dropping Puzzle

- Linear checking

- Binary search approach

- Binary search and linear combined approach

- Block approach

- DP approach

- This problem has many applications in the real world such as avoiding a call out to the slow HDD, or attempting to minimize cache misses, or running a large number of expensive queries on a database.

Egg Dropping Puzzle-recursion

- **N eggs, k floors**
- Recursion: try dropping an egg from each floor from 1 to k and calculate the minimum number of dropping needed in worst case.
- Base cases –
  - **Eggs – 1, floors – x** : play safe and drop from floor 1, if egg does not break then drop from floor 2 and so on. So in worst case x times an egg needs to be dropped to find the solution.
  - **Floors = 0**: No trials are required.
  - **Floors = 1**: 1 trails is required.
  - **Eggs = 0:** no trails
  - **Egg = 1:** x trails
- For rest of the case, if an egg is dropped from x^{th} floor then there are only 2 outcomes which are possible. Either egg will break OR egg will not break.
  - **If egg breaks** – check the floors lower than x. So problem is reduced is n-1 eggs and x-1 floors.
  - **If egg does not break** – check the floors higher than x floors with all the n eggs are remaining. So problem is reduced to n eggs and k-x floors.

$$eggDrop(n,k) = 1+min\{max\ (eggDrop(n-1,x-1)\ ,\ eggDrop(n,k-x)\ )\ ,\ x\ in\ 1{:}k\}$$

Egg Dropping Puzzle-recursion

eggDrop(n,k) = 1+min{max (eggDrop(n-1,x-1) , eggDrop(n,k-x)) , x in 1:k}

(2,2) = 1+ min{max(ed(1,0) , ed(2,1)) ,max(ed(1,1), ed(2,0))}

Eggs Floors->	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	1					
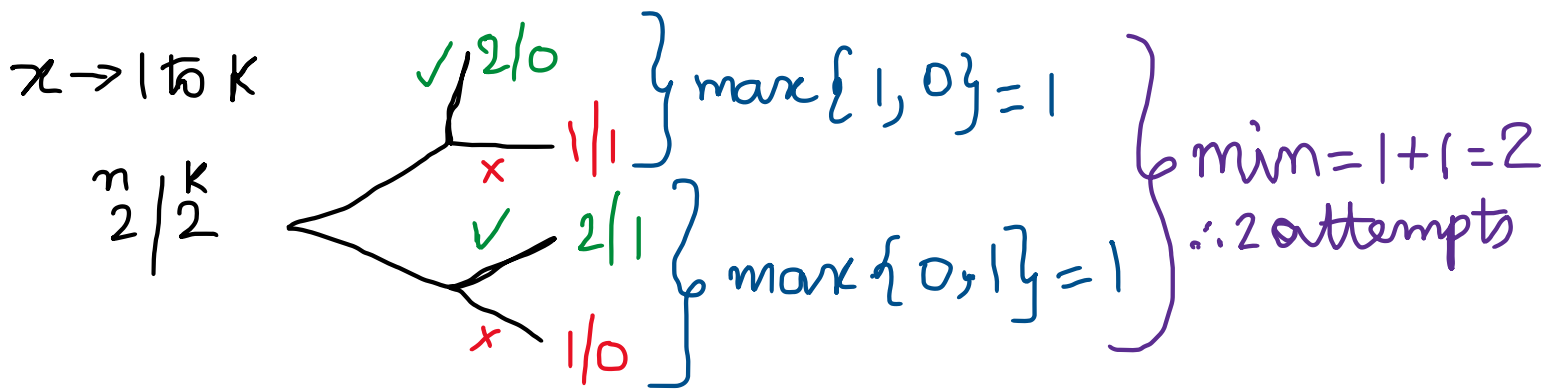
$x \rightarrow 1$ to K

$\overset{n}{2} | \overset{K}{2}$

✓ 2|0
✗ 1|1   } max{1,0}=1

✓ 2|1
✗ 1|0   } max{0,1}=1

} min=1+1=2
∴ 2 attempts

Egg Dropping Puzzle-recursion

eggDrop(n,k) = 1+min{max(eggDrop(n-1,x-1) ,
eggDrop(n,k-x)) , x in 1:k}

Eggs/Floors->	0	1	2	3	4	5	6
1							
2							
3							

Egg Dropping Puzzle-recursion

eggDrop(n,k) = 1+min{max(eggDrop(n-1,x-1) , eggDrop(n,k-x)) , x in 1:k}

Eggs/Floors->	0	1	2	3	4	5	6
1	0	1	2	3	4	5	6
2	0	1	2	2	3	3	3
3	0	1	2	2	3	3	3
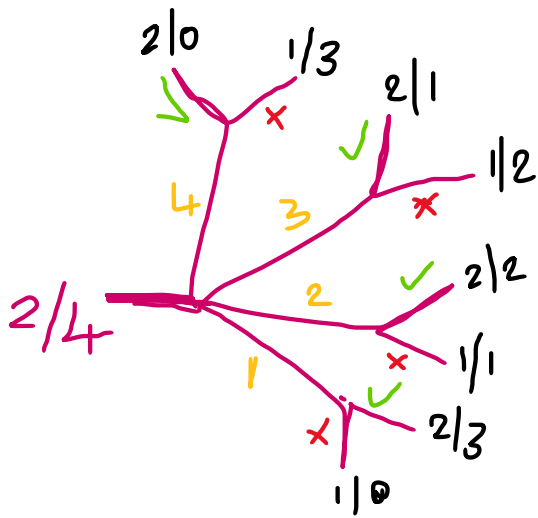
Egg Dropping Puzzle-Recursion

```c
int max(int a, int b)
{     return (a > b) ? a : b; }

int eggDrop(int n, int k)
{
    if (k == 1 || k == 0)
      return k;

    if (n == 1)
      return k;

    int min = INT_MAX, x, res;

     for (x = 1; x <= k; x++)
     {
       res = max( eggDrop(n - 1, x - 1), eggDrop(n, k - x));
       if (res < min)    min = res;
     }
    return min + 1;
}
```

Egg Dropping Puzzle-DP

```c
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int eggDrop(int n, int k)
{
     int eggFloor[n + 1][k + 1];
    int res,i,j,x;

    for (i = 1; i <= n; i++)
    {
       eggFloor[i][1] = 1;
       eggFloor[i][0] = 0;
    }

    for (j = 1; j <= k; j++)
       eggFloor[1][j] = j;

    for (i = 2; i <= n; i++)
    { for (j = 2; j <= k; j++)
       {eggFloor[i][j] = INT_MAX;
          for (x = 1; x <= j; x++)
          {
             res = 1 + max(
                     eggFloor[i - 1][x - 1],
                     eggFloor[i][j - x]);
             if (res < eggFloor[i][j])
                eggFloor[i][j] = res;
          }
       }
    }

// eggFloor[n][k] holds the result
    return eggFloor[n][k];
}
```

References

- https://medium.com/@parv51199/egg-drop-problem-using-dynamic-programming-e22f67a1a7c3
- https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/

Edit Distance

- The **Edit Distance** (or **Levenshtein distance**) is a metric for measuring the amount of difference between two strings.
  - The Edit Distance is defined as the minimum number of edits needed to transform one string into the other.

- It has many applications, such as spell checkers, natural language translation, and bioinformatics.

Edit Distance

- The problem of finding an edit distance between two strings is as follows (i.e. the minimum distance to convert one string to another string):
  - Given an **initial string s**, and a **target string t**, what is the minimum number of changes that have to be applied to *s* to turn it into *t* ?

- The list of valid changes are:
  1) Inserting a character
  2) Deleting a character
  3) Changing a character to another character (replace).

Example

Suppose we have two strings s,t

e.g. s = kitten

    t = sitting

and we want to transform s into t.

We use *edit operations*:

        1. insertions

        2. deletions

        3. substitutions (replace)

Example

- Input: str1 = "bmsse", str2 = "bmscse"

  Output: 1 We can convert str1 into str2 by inserting a 'c'.

- Input: str1 = "cat", str2 = "cut"

  Output: 1 We can convert str1 into str2 by replacing 'a' with 'u'.

- Input: str1 = "sunday", str2 = "saturday"

  Output: 3 Last three and first characters are same.

  We basically need to convert "un" to "atur".

  This can be done using below three operations.

  Replace 'n' with 'r', insert t, insert a

- What about:

  s = darladidirladada

  t = marmelladara

  Tough…

Solution

- The idea is process all characters one by one staring from either from left or right sides of both strings.

- Let us traverse from right corner, there are two possibilities for every pair of character being traversed.

   m: Length of str1 (first string)

   n: Length of str2 (second string)

Solution

- If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.

- Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.

  – Insert: Recur for m and n-1

  – Remove: Recur for m-1 and n

  – Replace: Recur for m-1 and n-1

Edit Distance

- Now, how do we use this to create a DP solution?

    – We simply need to store the answers to all the possible recursive calls.

    – In particular, all the possible recursive calls we are interested in are determining the edit distance between prefixes of s and t.

D(i,j) = score of **best alignment from *s1..si to t1..tj***

D(i-1,j-1), if si=tj *//copy*

= min {

D(i-1,j-1)+1, if si!=tj *//substitute*

D(i-1,j)+1 *//insert*

D(i,j-1)+1 *//delete*

}

Algorithm

int EditDistance(char s[1..n], **char** t[1..m])

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

    for i **from** 0 **to** n d[i, 0] := i
    for j **from** 1 **to** m d[0, j] := j

    for i **from** 1 **to** m
        for j **from** 1 **to** n
            if s[i] = t[j] **then**
             d[i,j]=d[i-1,j-1]
          else
          d[i, j] := minimum(

                        d[i-1, j] + 1, *//insertion*
                        d[i, j-1] + 1, *// deletion*
                        d[i-1, j-1] + 1 *// substitution*)

    return d[m,n]

Edit Distance

- Consider the following example with s="keep" and t="hello".
    - To deal with empty strings, an extra row and column have been added to the chart below:

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
e	3	3	2	2	3	4
p	4	4	3	3	3	4

- An entry in this table simply holds the edit distance between two prefixes of the two strings.
    - For example, the highlighted square indicates that the edit distance between the strings "he" and "keep" is 3.

Edit Distance
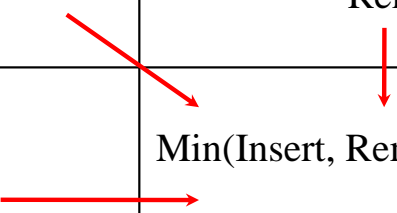
- In order to fill in all the values in this table we will do the following:

    1) Initialize values corresponding to the base case in the recursive solution.

	Null String	h	e	l	l	o
Null String	0	1	2	3	4	5
k	1					
e	2					
e	3					
p	4					
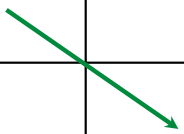
Table entries

If (s[i] != t[j])

Replace	Remove
Insert	Min(Insert, Remove,Replace)+1

If (s[i] = t[j]) copy the diagonal value

Edit Distance

- In order to fill in all the values in this table we will do the following:
  2) Loop through the table from the top left to the bottom right. In doing so, simply follow the recursive solution.
     - If the characters you are looking at match,

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2				
e	3					
P	4					

Just bring down the up&left value.

     - Else the characters don't match,

		h	e	l	l	o
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1			
e	3					
p	4					

min (1+ above,
1+ left, 1+diag up)

Exercise

- str1= "adceg"   str2= "abcfg"

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1					
d	2					
c	3					
e	4					
g	5					

Exercise

- str1= "adceg"   str2= "abcfg"

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2					
c	3					
e	4					
g	5					

Exercise

- str1= "adceg"   str2= "abcfg"

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2	1	1	2	3	4
c	3					
e	4					
g	5					

Exercise

- str1= "adceg"   str2= "abcfg"

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2	1	1	2	3	4
c	3	2	2	1	2	3
e	4					
g	5					

Exercise

- str1= "adceg"   str2= "abcfg"

	Null	a	b	c	f	g
Null	0	1	2	3	4	5
a	1	0	1	2	3	4
d	2	1	1	2	3	4
c	3	2	2	1	2	3
e	4	3	3	2	2	3
g	5	4	4	3	3	2

- Totally two operations are required.

Reference

- https://www.geeksforgeeks.org/edit-distance-dp-5/