Chapter 14: Dynamic Programming II

Matrix Multiplication

- Let A be a matrix of dimension $p \times q$ and B be a matrix of dimension $q \times r$
- Then, if we multiply matrices A and B, we obtain a resulting matrix $C = AB$ whose dimension is $p \times r$
- We can obtain each entry in C using q operations → in total, pqr operations

Matrix Multiplication

- Example :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}$$

- How to obtain $c_{1,2}$?

Matrix Multiplication

- In fact, $((A_1A_2)A_3) = (A_1(A_2A_3))$ so that matrix multiplication is associative

➔ Any way to write down the parentheses gives the same result

- e.g., $(A_1((A_2A_3)A_4)) = (A_1(A_2(A_3A_4)))$
  $= ((A1A2)(A3A4)) = (((A_1A_2)A_3)A_4)$
  $= ((A_1(A_2A_3))A_4)$

Counting the number of parenthesizations

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

Recursive Function

$$= \frac{1}{n}\binom{2n-2}{n-1} = \Omega(\frac{4^n}{n^{3/2}})$$

as n is large.

- Remark: On the other hand, #operations for each possible way of writing parentheses are computed at most once ➔ Running time = O(C(2n-2, n-1)/n) ➔ Catalan Number

Matrix Multiplication

- **Question:** Why do we bother this?
- Because different computation sequence may use different number of operations!
- e.g., Let the dimensions of A_1, A_2, A_3 be:

  1x100, 100x1, 1x100 , respectively

#operations to get $((A_1A_2)A_3)$ = ??

#operations to get $(A_1(A_2A_3))$ = ??

Optimal Substructure (allows recursion)

- Lemma: Suppose that to multiply $B_1, B_2, \ldots, B_n$, the way with minimum #operations is to:

  (i) first, obtain $B_1 B_2 \ldots B_x$

  (ii) then, obtain $B_{x+1} \ldots B_{n-1} B_n$ //x =1, 2, …, n-1

  (iii) finally, multiply the matrices of part (i) and part (ii)

- Then, the matrices in part (i) and part (ii) **must** be obtained with min #operations

Optimal Substructure

- Let $f_{i,j}$ denote the min #operations to obtain the product $A_i A_{i+1} \dots A_j$

  ➔ $f_{i,i} = 0$
- Let r_k and c_k denote #rows and #cols of A_k
- Then, we have:
- Lemma: For any $j > i$,

$$f_{i,j} = \min_{i \le k < j} \{ f_{i,k} + f_{k+1,j} + r_i c_k c_j \}$$

Recursive-Matrix-Chain

- Define a function Compute_F(i,j) as follows:
- Compute_F(i, j)   /* Finding $f_{i,j}$ */

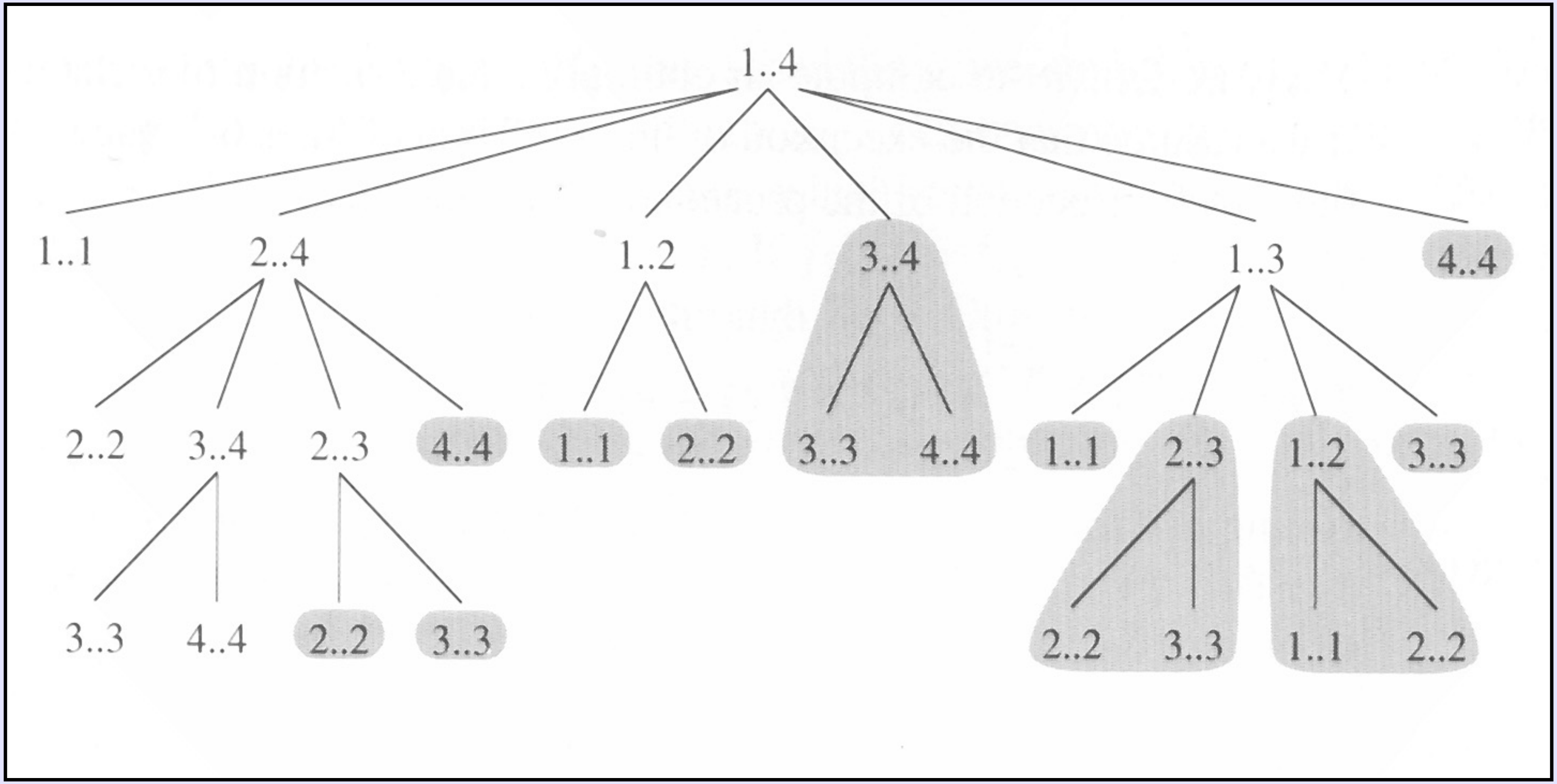
1. if (i == j) return 0;
2. m[i,j] = ∞;
3. for (k = i, i+1, …, j-1) {

   g = Compute_F(i,k) + Compute_F(k+1,j) + $r_i c_k c_j$;

   if (g < m[i,j]) m[i,j] = g;

   }
4. return m[i,j] ;

The recursion tree for the computation of RECURSUVE-MATRIX-CHAIN(P, 1, 4)



Time Complexity

- Question: Time to get Compute_F(1,n)?

$$\begin{cases} T(1) \geq 1 \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \ \ for\ n > 1 \end{cases}$$

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n$$

- By substituion method, we can show that Running time = $\Omega(2^n)$ How?

Overlapping Subproblems

- Here, we can see that :

  To Compute_F(i,j) and Compute_F(i,j+1),

  both have many COMMON subproblems:

  Compute_F(i,i), Compute_F(i,i+1),…, Compute_F(i,j-1)

- So, in our recursive algorithm, there are many redundant computations !

- Question: Can we avoid it ?

  Note: for (k = i, i+1, …, j-1) {

  g = Compute_F(i,k) + Compute_F(k+1,j) + $r_i \, c_k \, c_j$;

Bottom-Up Approach

- We notice that

  $f_{i,j}$ depends only on $f_{x,y}$ with $1 \leq y-x < j-i$, and $x \geq i$.

- Let us create a 2D table F to store all $f_{i,j}$ values once they are computed

- Then, compute $f_{i,j}$ for $j-i = 1,2,\ldots,n-1$

Bottom-Up Approach

BottomUp_F(p) /* Finding min #operations */

1. n = p.length -1 /* $A_i = p_{i-1} \times p_i$ */
2. for i = 1, 2, …, n, set F[i, i] = 0 ;
3. for (length = 1,…, n-1) {

   for i = 1, 2,…n-length

   Compute F[i, i+length]; // $\Theta(n)$

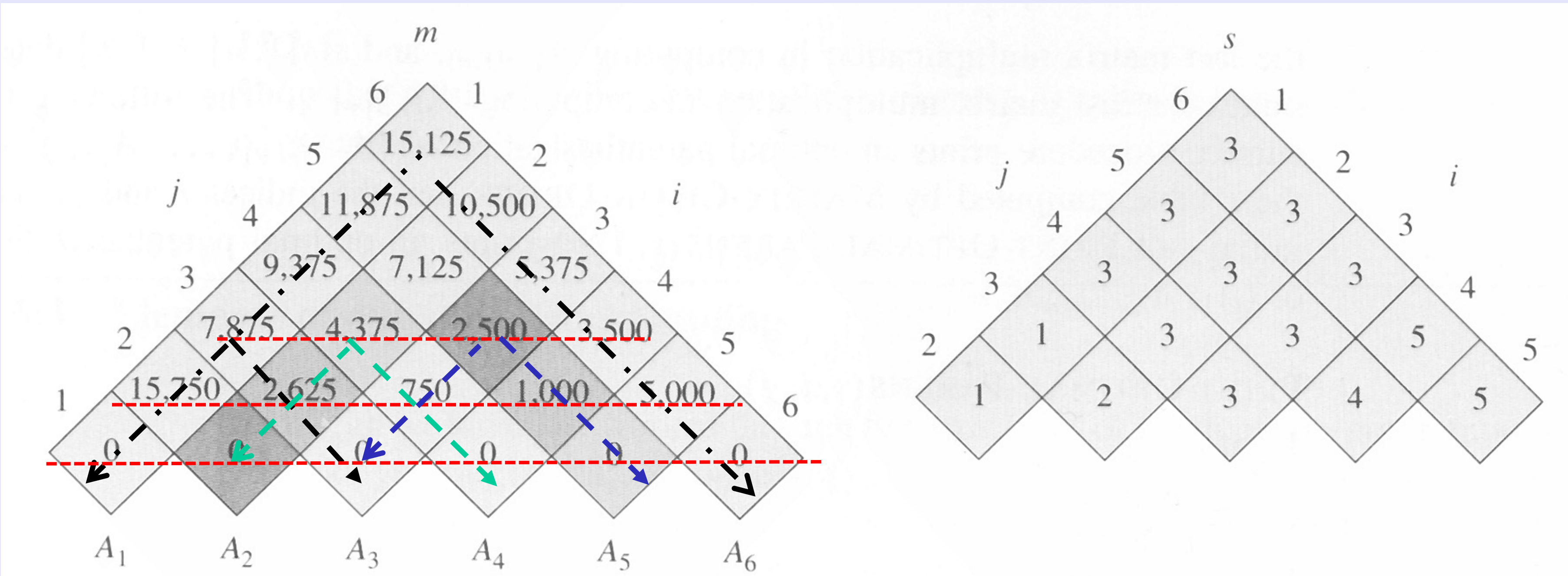
   /* Based on F[x,y] with |x-y| < length */ }
4. return F[1, n] ;

Running Time = $\Theta(n^3)$ why?

Example:

$$
\begin{array}{lll}
A_1 & 30 \times 35 & = p_0 \times p_1 \\
A_2 & 35 \times 15 & = p_1 \times p_2 \\
A_3 & 15 \times 5 & = p_2 \times p_3 \\
A_4 & 5 \times 10 & = p_3 \times p_4 \\
A_5 & 10 \times 20 & = p_4 \times p_5 \\
A_6 & 20 \times 25 & = p_5 \times p_6
\end{array}
$$

The m and s table computed by MATRIX-CHAIN-ORDER for n = 6



$$f_{i,j} = min_{i \le k < j} \{f_{i,k} + f_{k+1,j} + r_i c_k c_j\}$$

Optimal Solution: $((A_1(A_2 A_3))((A_4 A_5) A_6))$

Example

$m[2,5]$= **Min** {$m[2,2] + m[3,5] + p_1p_2p_5 =$
$0+2500+35\times15\times20 = 13000$,

$m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35\times5\times20$
$=7125$,

$m[2,4] + m[5,5] + p_1p_4p_5 = 4375+0+35\times10\times20$
$=11374$} = 11375 } =**7125**

Bottom-Up Approach

```
MATRIX_CHAIN_ORDER(p)
n ← length[p] –1
for i ← 1 to n
      do m[i, i] ← 0
for l ← 2 to n      // l is the chain length
      do for i ← 1 to n – l + 1
                do j ← i + l – 1
                    m[i, j] ← ∞
                    for k ← i to j – 1
                          do q ← m[i, k] + m[k+1, j]+ p_{i-1} p_k p_j
                              if q < m[i, j]
                                    then m[i, j] ← q
                                          s[i, j] ← k
return m and s
```

Remarks

- Again, a slight change in the algorithm allows us to get the exact sequence of steps (or the parentheses) that achieves the minimum number of operations

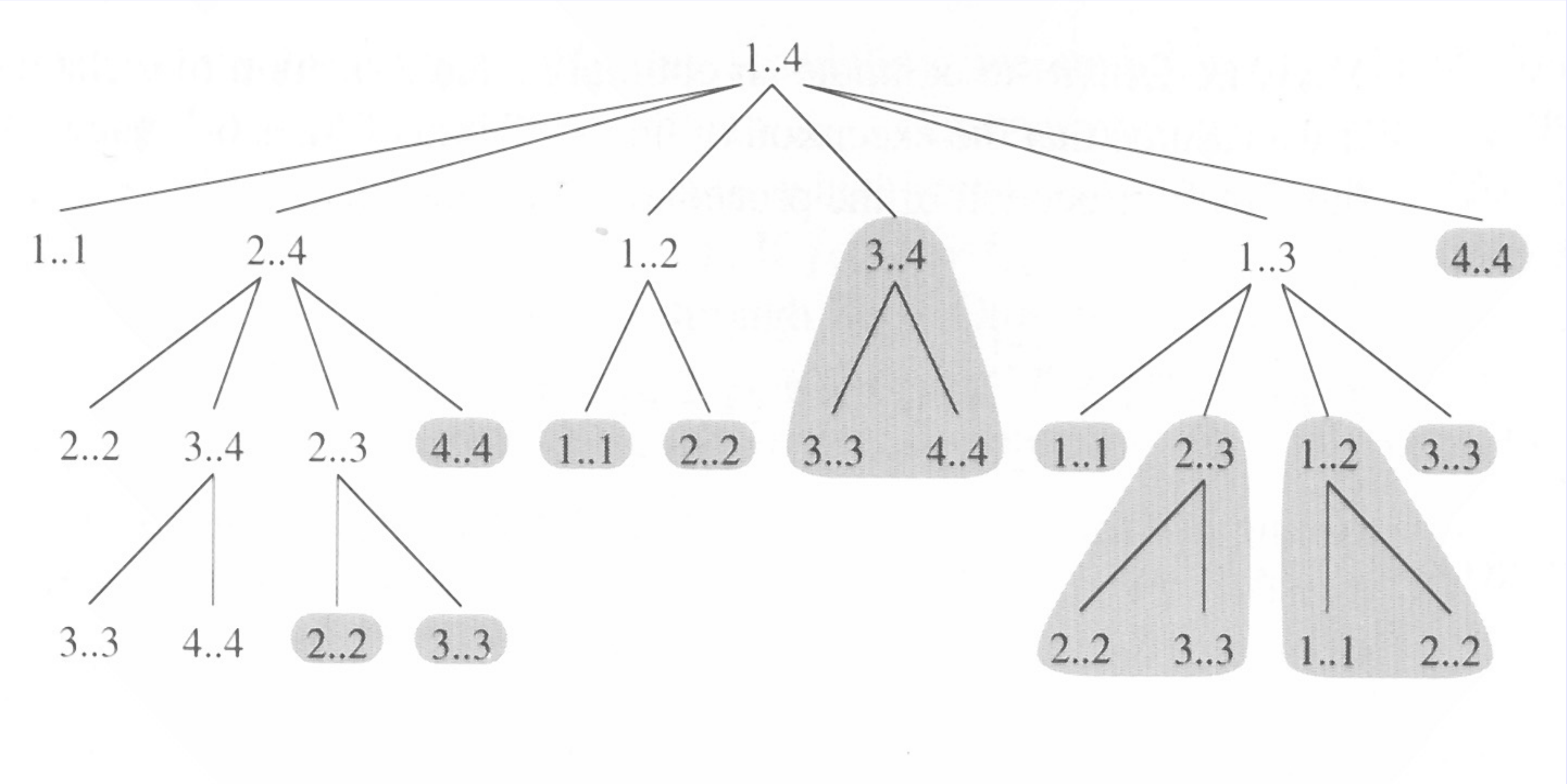
- Also, we can make minor changes to the recursive algorithm and obtain a memoized version (whose running time is $O(n^3)$)

Top down approach

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

Running time = $\Omega(2^n)$

The recursion tree for the computation of RECURSUVE-MATRIX-CHAIN(P, 1, 4)



Memoization

- Alternative approach to dynamic programming:
  - ✓ "Store, don't recompute."
  - ✓ Make a table indexed by subproblem.
  - ✓ When solving a subproblem: Lookup in table.
  - ✓ If answer is there, use it.
  - ✓ Else, compute answer, then store it.
- In bottom-up dynamic programming, we go one step further. We determine in what order we'd want to access the table, and fill it in that way.

MEMORIZED_MATRIX_CHAIN

```
MEMORIZED_MATRIX_CHAIN(p)
    n ← length[p] -1
    for i ← 1 to n
        do for j ← i to n
                do m[i, j] ← ∞
    return LC (m,p,1,n)
```

LOOKUP_CHAIN (LC)

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

Time Complexity: $O(n^3)$

When should we apply DP?

- *Optimal structure:* an optimal solution to the problem contains optimal solutions to subproblems.
  - Example: Matrix-multiplication problem
- *Overlapping subproblems:* a recursive algorithm revisits the same subproblem over and over again.

Optimal substructure

- Optimal substructure varies across problem domains in two ways:
  1. how many subproblems are used in an optimal solution to the original problem (e.g. matrix-chain multiplication has n^2 subproblems, see the 2D table), and
  2. how many choices we have in determining which subproblem(s) to use in an optimal solution. (e.g. each subproblem m[i, j] has j-i choices)
- Informally, running time depends on (# of subproblems overall) x (# of choices).

Overlapping subproblems

- These occur when a recursive algorithm revisits the same problem over and over.
- Solutions

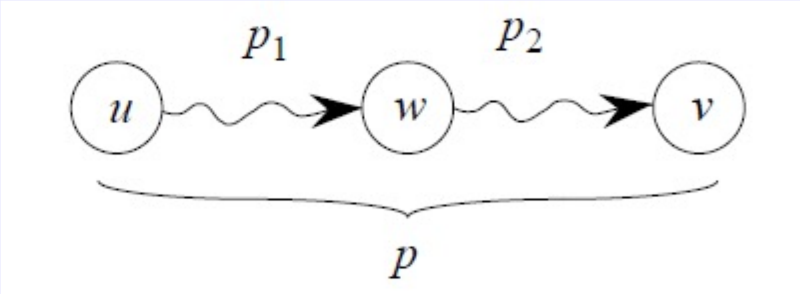
1. Bottom up
2. Memorization (memorize the natural, but inefficient)

Refinement

- One should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.
  - Unweighted shortest path:
    - Find a path from u to v consisting of the fewest edges. Good for Dynamic programming.
  - Unweighted longest simple path:
    - Find a simple path from u to v consisting of the most edges. Not good for Dynamic programming.

Shortest path

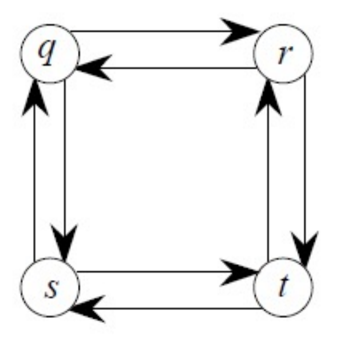
- Shortest path has optimal substructure.



- Suppose p is shortest path u -> v.
- Let w be any vertex on p.
- Let p_1 be the portion of p going u -> w.
- Then p_1 is a shortest path u -> w.

Longest simple path

- Does longest path have optimal substructure?



- Consider q -> r -> t = longest path q -> t. Are its subpaths longest paths? No!
- Longest simple path q -> r is q -> s -> t -> r.
- Longest simple path r -> t is r -> q ->s -> t .
- Not only isn't there optimal substructure, but we can't even assemble a legal solution from solutions to subproblems.

Practice at home

- Exercise: 14.2-1, 14.2-3, 14.2-5, 14.3-2, 14.3-3, 14.3-4

Homework

- Please use DP to find a maximum independent set in a tree. Let G = (V,E) be an undirected finite graph where V denotes the set of vertices and E denotes the set of edges. If G is connected and acyclic, then it is called a tree. A subset I of V is called an independent set of G if no two vertices of I are adjacent in G. Assume that a positive weight w(i) is associated with each vertex i. We define the weight w(I) of an independent set I to be the sum of the weights of all the vertices in I. That is, w(I) = $\sum_{i \in I} w(i)$. Further, an independent set is called a maximum weight independent set if it has maximum weight.

Homework

- Consider a chessboard of size $k \times n$. How many ways are there to cover the chessboard completely with n rectangular bars, each of size $k \times 1$ or $1 \times k$?
- For instance, when $k = 2$, $n = 3$, there are three different ways:
- (a) cover the leftmost column by a vertical bar, and the remaining region by two horizontal bars;
- (b) cover the rightmost column by a vertical bar, and the remaining region by two horizontal bars; or
- (c) cover each column with a vertical bar.
- Design an $O(n)$-time algorithm to compute the desired answer for any input k and n.