CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

Chapter 14. Dynamic Programming

Introduction

- Dynamic programming solves problems by combining the solutions to subproblems.
  - similar to the divide-and-conquer method
  - "programming" refers to a tabular method, not to writing computer code
- Difference with divide-and-conquer
  - dynamic programming applies when the subproblems overlap
  - subproblems share subsubproblems
- When the subproblems overlap, a divide-and-conquer algorithm does more work than necessary
  - repeatedly solves common subsubproblems
- A dynamic programming algorithm solves each subsubproblem just once and then saves its answer in a table, avoiding recomputing.

Applications

- Dynamic programming typically applies to optimization problems.
  - Such problems can have many possible solutions.
  - Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value.
  - There may be multiple "optimal" solutions that achieve the optimal value.

Steps

- 1. Characterize the structure of an optimal solution.
- 2. Recursively define the value of an optimal solution.
- 3. Compute the value of an optimal solution, typically in a bottom-up fashion.
- 4. Construct an optimal solution from computed information
  - this step can be omitted if we are only interested in the value of the optimal solution, and not the solution itself.

14.1 Rod Cutting

Introduction

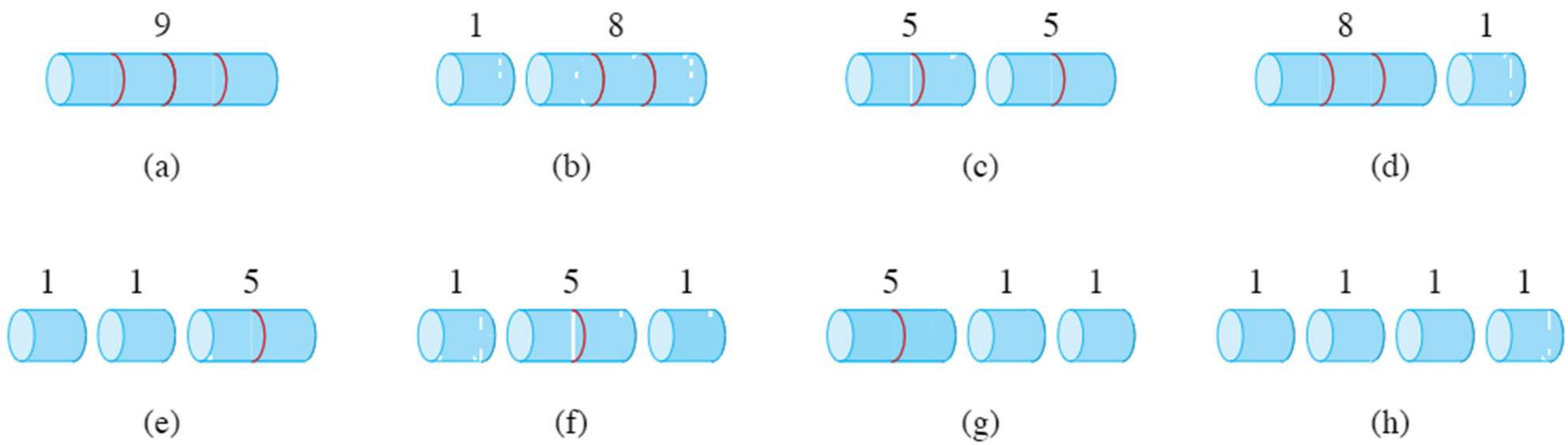
- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells.
- Each cut is free.
- The company has a table giving, for $i = 1,2, \ldots$, the price p_i in dollars that they charge for a rod of length i inches.
- The length of each rod in inches is always an integer.
- The management of Serling Enterprises wants to know the best way to cut up the rods.

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

price table

Example

- 8 possible ways of cutting up a rod of length 4.



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

The Rod-Cutting Problem

- Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \dots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

- We can cut up a rod of length n in 2^{n-1} different ways.
  - We have an independent option of cutting or not cutting at distance i inches from the left end, for $i = 1, 2, \dots, n-1$.

- We will denote a decomposition into pieces using ordinary additive notation, such as $7 = 2 + 2 + 3$.

- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then optimal decomposition $n = i_1 + i_2 + \cdots + i_k$ of the rod into pieces of lengths $i_1, i_2, \dots, i_k$ provides the maximum corresponding revenue $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

The Rod-Cutting Problem: Example

- Suppose this is the price table.

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

- Then,
  - $r_1 = 1$ from solution $1 = 1$ (no cuts),
  - $r_2 = 5$ from solution $2 = 2$ (no cuts),
  - $r_3 = 8$ from solution $3 = 3$ (no cuts),
  - $r_4 = 10$ from solution $4 = 2 + 2$,
  - $r_5 = 13$ from solution $5 = 2 + 3$,
  - $r_6 = 17$ from solution $6 = 6$ (no cuts),
  - $r_7 = 18$ from solution $7 = 1 + 6$ or $7 = 2 + 2 + 3$,
  - $r_8 = 22$ from solution $8 = 2 + 6$,
  - $r_9 = 25$ from solution $9 = 3 + 6$,
  - $r_{10} = 30$ from solution $10 = 10$ (no cuts)

The Rod-Cutting Problem: Insights

- Generally, we can express the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods.

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

- The first argument p_n corresponds to making no cuts at all.

- In other cases, we make an initial cut of the rod into two pieces of size i and $n - i$ for each $i = 1, 2, 3, \ldots, n - 1$.

- Then, we can obtain optimal revenues r_i and r_{n-i} from those two pieces.

The Rod-Cutting Problem: Insights

- For each i, you make the first cut at place i.
- The overall optimal solution for this case incorporates optimal solutions to the two resulting subproblems.
  - maximizing revenue from each of those two pieces.

- Optimal substructure
  - optimal solutions to a problem incorporate optimal solutions to related subproblems, which you may solve independently.

The Rod-Cutting Problem: Insights

- We can view the decomposition as follows:
  - The left piece of length i
  - The right piece of length $n-1$

- The left piece can no longer be cut.
- We can only cut the right piece.
- This view does not limit the ways of cutting the rods.

- Then, we can make a simpler version of the equation for calculating r_n.

- $r_n = \max\{p_i + r_{n-i}: 1 \leq i \leq n\}$
  - When $i = n$, it means that we are not cutting the rod at all. We assume $r_0 = 0$.

- Now, an optimal solution embodies the solution to only one related subproblem rather than two.

The Rod-Cutting Problem: Divide-and-Conquer?

- CUT-ROD: top-down and recursive

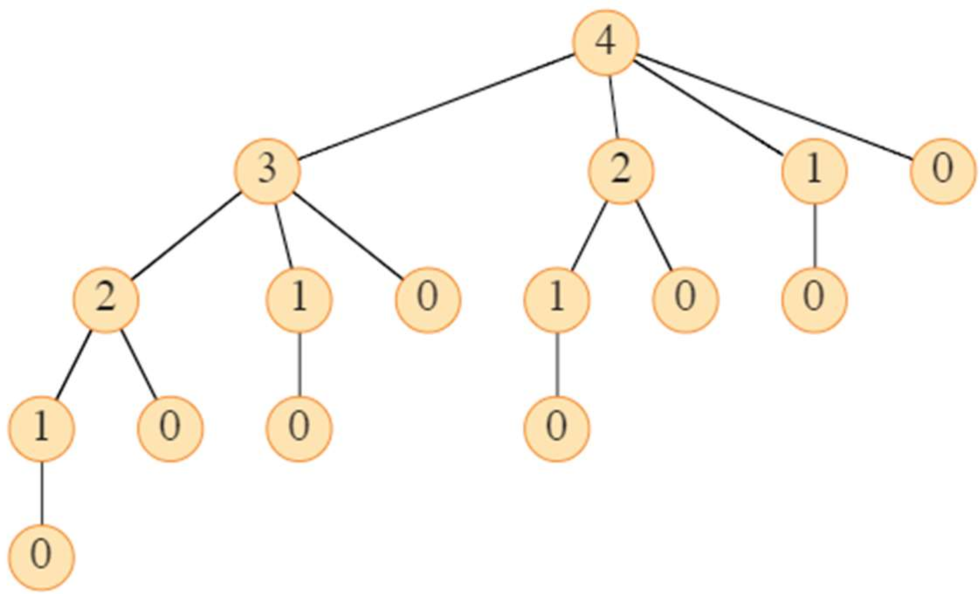
```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max {q, p[i] + CUT-ROD(p, n − i)}
return q
```

- Is this efficient?

The Rod-Cutting Problem: Divide-and-Conquer?

- CUT-ROD is inefficient because it is recursively called over and over again with the same parameter values. It solves the same problem repeatedly.

- Recursion tree



The Rod-Cutting Problem: Running Time of CUT-ROD

- $T(n)$ denotes the total number of calls made to CUT-ROD(p, n) for a particular value of n.

- It is equal to the number of nodes in a subtree whose root is labeled n in the recursion tree.

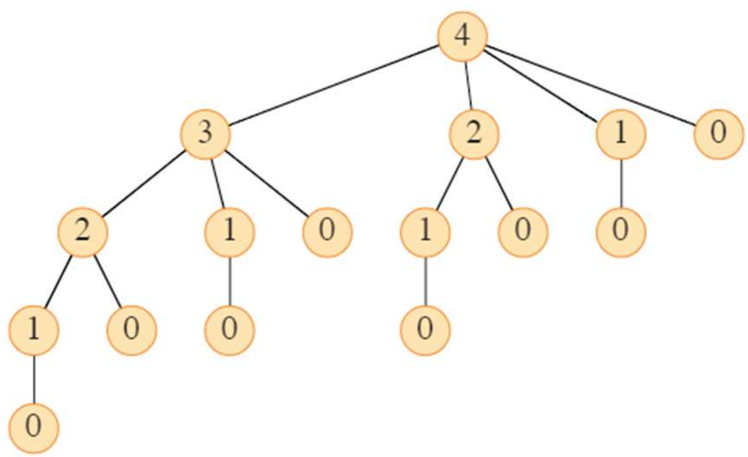
- $T(0) = 1, T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

- The initial 1 is for the call at the root.
- The term $T(j)$ counts the number of calls (including recursive calls) due to call CUT-ROD($p, n - i$), where $j = n - i$.

- Solving this recurrence relation, $T(n) = 2^n = \Theta(2^n)$.

The Rod-Cutting Problem: Running Time of CUT-ROD

- Conceptual explanation of why $T(n) = \Theta(2^n)$
  - A rod of length n has $n-1$ potential locations to cut.
  - Each possible way to cut up the rod makes a cut at some subset of these $n-1$ locations (including the empty set which corresponds to no cut.)
  - There are 2^{n-1} subsets.
  - Each leaf in the recursion tree corresponds to one possible way to cut up the rods. There are 2^{n-1} leaves.
  - Each node in the tree corresponds to a recursion call to CUT-ROD.
  - Thus, at least 2^{n-1} recursion calls should be made.



The Rod-Cutting Problem: Dynamic Programming

- Instead of solving the same problems repeatedly, arrange for each subproblem to be solved only once.

- The first time we solve a subproblem, we save its solution in a table.

- If we need to refer to this subproblem's solution again later, just look it up. Don't recompute it.

- We need memory to store solutions, so dynamic programming can be viewed as a time-memory trade-off.

The Rod-Cutting Problem: Dynamic Programming

- Top-down approach
  - We write the procedure recursively in a natural manner, but modify to save the result of each subproblem.
  - When solving a subproblem, the procedure first checks whether the solution has been previously saved.
  - If the solution is saved, the procedure simply returns the value.
  - If the solution is not saved, the procedure computes the value and saves it.

The Rod-Cutting Problem: Dynamic Programming

- Top-down approach

```
MEMOIZED-CUT-ROD(p, n)
let r[0 : n] be a new array        // will remember solution values in r
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                        // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n                 // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q                           // remember the solution value for length n
return q
```

The Rod-Cutting Problem: Dynamic Programming

- Bottom-up approach
  - We solve the subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved.
  - In this way, when solving a particular problem, there are already saved solutions for all of the smaller subproblems its solution depends upon.
  - You need to solve each subproblem only once, and when you first see it, you have already solved all of its prerequisite subproblems.

- In most cases, top-down and bottom-up approach of dynamic programming have the same asymptotic running time.

- However, the bottom-up approach often has much better constant factors, since it has lower overhead for procedure calls.

The Rod-Cutting Problem: Dynamic Programming

- Bottom-up approach

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] be a new array        // will remember solution values in r
r[0] = 0
for j = 1 to n                     // for increasing rod length j
    q = −∞
    for i = 1 to j                 // i is the position of the first cut
        q = max {q, p[i] + r[j − i]}
    r[j] = q                       // remember the solution value for length j
return r[n]
```

서강대학교 SOGANG UNIVERSITY

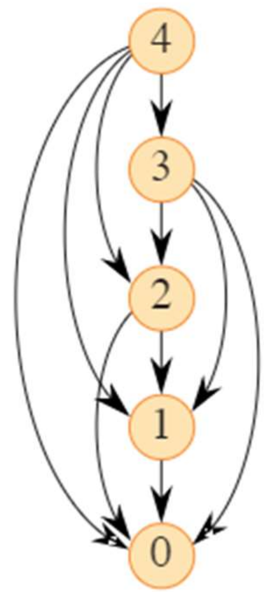
Running Time of Dynamic Programming Approaches

- Running time of BOTTOM-UP-CUT-ROD
  - We have a nested for loop.
  - Line 3-7 forms the outer for loop, where j is incremented from 1 to n.
  - Line 5-7 forms the inner for loop, which iterates j times.
  - Thus, the running time of BOTTOM-UP-CUT-ROD is $\Theta(n^2)$.

- Running time of MEMOIZED-CUT-ROD
  - Note that a recursive call to solve a previously solved problem returns immediately.
  - MEMOIZED-CUT-ROD-AUX solves each problem for sizes $0, 1, \ldots, n$ just once.
  - To solve a problem of size n, the for loop of lines 6-7 iterates n times.
  - The total number of iterations for this for loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series $(1 + 2 + \cdots + n)$, giving a total of $\Theta(n^2)$ iterations.

Subproblem Graph

- The subproblem graph shows how subproblems depend on one another.
- The subproblem graph for the rod-cutting problem.



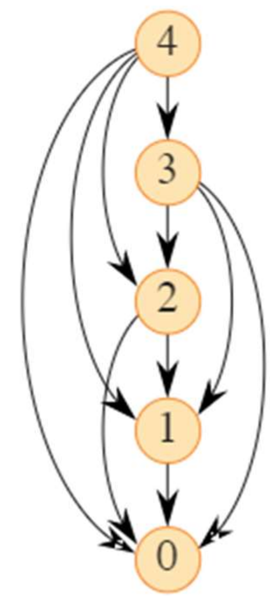
- For example, to solve a rod-cutting problem for length 4, we need the solutions to rod-cutting problem for lengths 0, 1, 2, 3.

Subproblem Graph

- Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph.

- The number of subproblems is equal to the number of vertices in the subproblem graph.

- In this common case, the running time of dynamic programming is linear in the number of vertices and edges.



Reconstructing a Solution

- The procedures MEMOIZED-CUT-ROD and BOTTOM-UP-CUT-ROD return the value of an optimal solution to the rod-cutting problem, but they do not return the solution itself: a list of piece sizes.

- In order to obtain the solution, the dynamic programming approach should record not only the optimal value but also the choice for each subproblem.

Reconstructing a Solution

- A procedure that records the optimal solution as well as the optimal value.
  - When $p[i] + r[j - i]$ is the maximum, we record $s[j] = i$.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] and s[1 : n] be new arrays
r[0] = 0
for j = 1 to n                        // for increasing rod length j
    q = -∞
    for i = 1 to j                    // i is the position of the first cut
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i                  // best cut location so far for length j
    r[j] = q                          // remember the solution value for length j
return r and s
```

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
$s[i]$		1	2	3	2	2	6	1	2	3	10

Reconstructing a Solution

- Now that we have the solution, we can print the cut locations.

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]          // cut location for length n
    n = n − s[n]        // length of the remainder of the rod
```

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
$s[i]$		1	2	3	2	2	6	1	2	3	10

- When $n = 10$, the procedure just prints 10.
- When $n = 7$, the procedure prints $1\ 6$.

14.2 Matrix-Chain Multiplication

Matrix-Chain Multiplication Problem: Introduction

- Given a sequence (chain) <$A_1, A_2, \ldots, A_n$> of n matrices to be multiplied, where the matrices aren't necessarily square, the goal is to compute the product

- $A_1 A_2 A_3 \ldots A_n$,

- while minimizing the number of scalar multiplications.

Matrix-Chain Multiplication: Example

- If the chain of matrices is <A_1, A_2, A_3, A_4>, then you can parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

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

- How we parenthesize a chain of matrices can have a significant impact on the cost of evaluating the product.

Cost of Multiplying Two Rectangular Matrices

- The procedure RECTANGULAR-MATRIX-MULTIPLY computes $C = C + A \cdot B$ for three matrices $A = (a_{ij})$, $B = (b_{ij})$, and $C = (C_{ij})$, where A is $p \times q$, B is $q \times r$, and C is $p \times r$.

```
RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
1  for i = 1 to p
2      for j = 1 to r
3          for k = 1 to q
4              c_ij = c_ij + a_ik · b_kj
```

- The running time of RECTANGULAR-MATRIX-MULTIPLY is dominated by the number of scalar multiplications in line 4, which is pqr.

Example: Order of Multiplication Affects Cost

- We have a chain of three matrices <A_1, A_2, A_3>.
- Suppose their dimensions are 10×100, 100×5, and 5×50, respectively.

- Case 1: $\left((A_1 A_2) A_3\right)$
  - $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute $A_1 A_2$, plus
  - $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3.
  - Total: 7500 scalar multiplications

- Case 2: $\left(A_1 (A_2 A_3)\right)$
  - $100 \cdot 5 \cdot 50 = 25000$ scalar multiplications to compute $A_2 A_3$, plus
  - $10 \cdot 100 \cdot 50 = 50000$ scalar multiplications to multiply A_1 by this matrix
  - Total: 75000 scalar multiplications

Problem Definition

- Matrix-chain multiplication problem

- Given a chain <$A_1, A_2, \dots, A_n$> of n matrices, where for $i = 1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \dots A_n$ in a way that minimizes the number of scalar multiplications. The input is the sequence of dimensions <$p_0, p_1, \dots, p_n$>.

- This problem does not entail actually multiplying matrices. The goal is only to determine an order for multiplying matrices that has the lowest cost.

- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications.
  - 7,500 vs. 75,000 scalar multiplications

Counting the Number of Parenthesizations

- Can't we just exhaustively check all possible parenthesizations?

- To find out, let us denote the number of alternative parenthesizations of sequence of n matrices by $P(n)$.

- When $n = 1$, there is just one matrix, so there is only one way to fully parenthesize the matrix product.

- When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$.

Counting the Number of Parenthesizations

- We obtain the recurrence

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

- When we solve the recurrence, $P(n) = \Omega(2^n)$.

- Thus, exhaustively searching all possible parenthesizations is not efficient.

Applying Dynamic Programming

- The four steps
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution
  - Construct an optimal solution from computed information

Step 1: Structure of an Optimal Parenthesization

- Let $A_{i:j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \dots A_j$.

- If $i < j$, to parenthesize the product $A_i A_{i+1} \dots A_j$, the product must split between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.

- That is, we first compute the matrices $A_{i:k}$ and $A_{k+1:j}$, and then multiply them together to produce the final product $A_{i:j}$.

- The cost
  - The cost of computing the matrix $A_{i:k}$, plus
  - the cost of computing the matrix $A_{k+1:j}$, plus
  - the cost of multiplying them together.

Step 1: Structure of an Optimal Parenthesization

- When computing $A_{i:j}$, suppose the optimal split is to split the product between A_k and A_{k+1}. In this solution, the parenthesization of each subchain must be optimal parenthesization.

- The subchain $A_{i:k}$ should have the optimal parenthesization.
- The subchain $A_{k+1:j}$ should have the optimal parenthesization.

- To build an optimal solution, we split the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \dots A_k$ and $A_{k+1} A_{k+2} \dots A_j$), find optimal solutions to the two subproblem instances, and then combine these optimal subproblem solutions.

- To ensure that we've examined the optimal split, we must consider all possible splits.

Step 2: A Recursive Solution

- Given the input dimensions <$p_0, p_1, p_2, \ldots, p_n$>, an index pair i, j specifies a subproblem.

- Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute $A_{1:n}$ is thus $m[1,n]$.

- We can define $m[i,j]$ recursively as follows:
- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$

- Since we do not know the optimal k, we need to try all possible values of k, $k = i, i+1, \ldots, j-1$.

Step 2: A Recursive Solution

- Recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \ldots A_j$ becomes

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

- To construct the optimal solution, we define $s[i,j]$ to be a value k at which you split the product $A_i A_{i+1} \ldots A_j$ in an optimal parenthesization.
- That is, $s[i,j]$ equals a value k such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

Step 3: Computing the Optimal Costs

- There is one subproblem for each choice of i and j satisfying $1 \leq i \leq j \leq n$.
- Thus, the number of subproblems is $\Theta(n^2)$.

- A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree.

- This property of overlapping subproblems is the second hallmark of when dynamic programming applies.
  - The first hallmark is the optimal substructure.

- We will use a tabular, bottom-up approach to solve the problem.

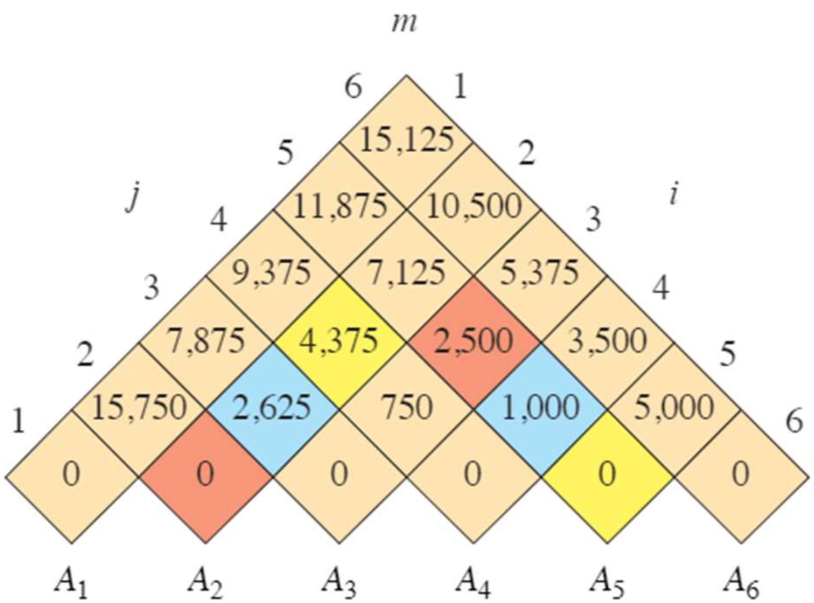
Step 3: Computing the Optimal Costs

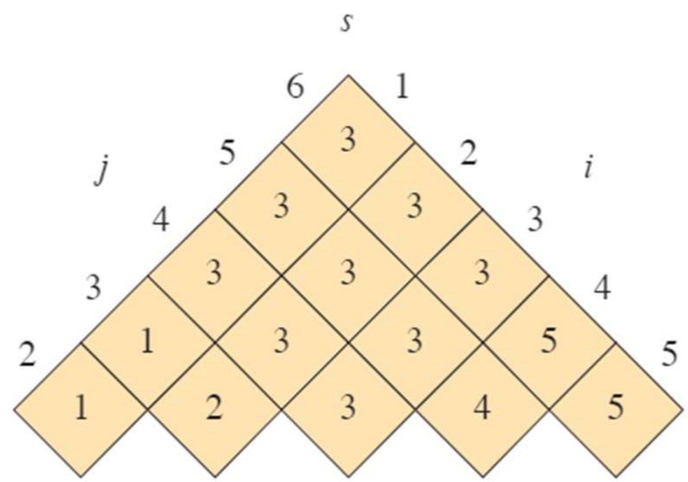
- Procedure MATRIX-CHAIN-ORDER

```
MATRIX-CHAIN-ORDER(p, n)
let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
for i = 1 to n                          // chain length 1
    m[i,i] = 0
for l = 2 to n                          // l is the chain length
    for i = 1 to n - l + 1              // chain begins at A_i
        j = i + l - 1                   // chain ends at A_j
        m[i,j] = ∞
        for k = i to j - 1              // try A_{i:k} A_{k+1:j}
            q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
            if q < m[i,j]
                m[i,j] = q              // remember this cost
                s[i,j] = k              // remember this index
return m and s
```

Step 3: Computing the Optimal Costs

- Tables for m and s
  - The optimal number of scalar multiplications: 15,125





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

Step 3: Computing the Optimal Costs

- Procedure MATRIX-CHAIN-ORDER has three nested for loops.
- Each loop index (l, i, and k) takes on at most $n - 1$ values.
- Running time of the procedure: $\Theta(n^3)$

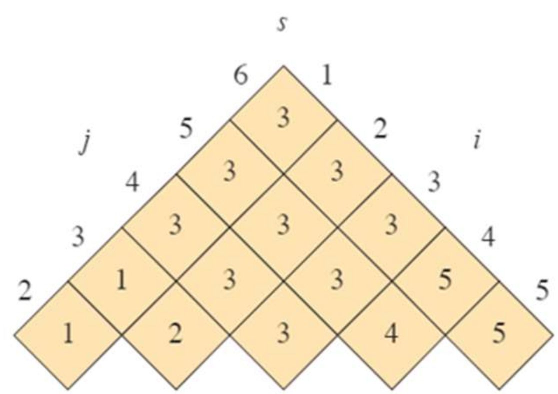
- The algorithm requires $\Theta(n^2)$ of memory space to store m and s tables.

Step 4: Constructing an Optimal Solution

- Procedure for printing the optimal solution.

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
      print "A"_i
else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

- Example
  - If we have the s table as in the right:
  - the optimal solution is $\left(\left(A_1(A_2A_3)\right)\left((A_4A_5)A_6\right)\right)$.



14.4 Longest Common Subsequence

Introduction

- Biological applications often need to compare the DNA of two (or more) different organisms.

- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, cytosine, guanine, and thymine.

- Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the 4-element set {A, C, G, T}.

- For example, the DNA of one organism may be
  - $S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

- and the DNA of another organism may be
  - $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

Introduction: DNA Strands

X = TCCCCGCTCTGCTCTGTCCGGTCACAGGACTTTTTGCCCTCTGTTCCCGGGTCCCTCAGGCGGCCACCCA
GTGGGCACACTCCCAGGCGGCGCTCCGGCCCCGCGCTCCCTCCCTCTGCCTTTCATTCCCAGCTGTCAAC
ATCCTGGAAGCTTTGAAGCTCAGGAAAGAAGAGAAATCCACTGAGAACAGTCTGTAAAGGTCCGTAGTGC
TATCTACATCCAGACGGTGGAAGGGAGAGAAAGAGAAAGAAGGTATCCTAGGAATACCTGCCTGCTTAGA
CCCTCTATAAAAGCTCTGTGCATCCTGCCACTGAGGACTCCGAAGAGGTAGCAGTCTTCTGAAAGACTTC
AACTGTGAGGACATGTCGTTCAGATTTGGCCAACATCTCATCAAGCCCTCTGTAGTGTTTCTCAAAACAG
AACTGTCCTTCGCTCTTGTGAATAGGAAACCTGTGGTACCAGGACATGTCCTTGTGTGCCCGCTGCGGCC
AGTGGAGCGCTTCCATGACCTGCGTCCTGATGAAGTGGCCGATTTGTTTCAGACGACCCAGAGAGTCGGG
ACAGTGGTGGAAAAACATTTCCATGGGACCTCTCTCACCTTTTCCATGCAGGATGGCCCCGAAGCCGGAC
AGACTGTGAAGCACGTTCACGTCCATGTTCTTCCCAGGAAGGCTGGAGACTTTCACAGGAATGACAGCAT
CTATGAGGAGCTCCAGAAACATGACAAGGAGGACTTTCCTGCCTCTTGGAGATCAGAGGAGGAAATGGCA
GCAGAAGCCGCAGCTCTGCGGGTCTACTTTCAGTGACACAGATGTTTTTCAGATCCTGAATTCCAGCAAA
AGAGCTATTGCCAACCAGTTTGAAGACCGCCCCCCCGCCTCTCCCCAAGAGGAACTGAATCAGCATGAAA
ATGCAGTTTCTTCATCTCACCATCCTGTATTCTTCAACCAGTGATCCCCCACCTCGGTCACTCCAACTCC
CTTAAAATACCTAGACCTAAACGGCTCAGACAGGCAGATTTGAGGTTTCCCCCTGTCTCCTTATTCGGCA
GCCTTATGATTAAACTTCCTTCTCTGCTGCAAAAAAAAAAAAAAA

Y = ATGTTAACCAAGGAATGGATCTGTGTCGTTCCACGTTCGAAGGCCTTTTCTGATGAAATGAAGATAGGTT
TCAACTCCACAGGTTATTGTGGTATGATCTTAACCAAAAATGATGAAGTTTTCTCCAAGATTACTGAAAA
ACCTGAATTGATTAACGATATCTTATTGGAATGTGGTTTCCCAAACACTTCTGGTCAAAAACCAAACGAA
TACAACTATTGAGTCCTTCACAAGTACAAGTATACACGTATACATGTATGTATATATATATGTATTTAAA
TGATAGAAGTAATTTCTATATGTATATGTCTATTCAATTTTTATTTCTAATGACTTTGAAATTTTATATT
TATTATTCTACACTTTATTATTAAAACTACATGCAATCAATGCCGCTAAGGTTACGATTCACCTTTTTAT
TACTTATTATTAATATTGTTGTATTACTTCCTTGAAAAAATATGTCTAAAGAGTCCTAATTTGGATTTTC
TTTTCCTCCTAACTTCACTGTCCTGCGCCTGCTTTCCTAACGCACCATCGCTAATACACCAGCTTTCATT
GCTTGTTGCTGCGCTATTGCTCGCATGGAACGTTTTCAGTGCGTCATCATCCTGGGATAAACTAAAGACT
AAGTCACCAGTTTCATTTGAGGCTTTTCTCTGCGTGTTGACAAAAGAGGACAGATCAACCATATCACCTG
GTTCACCATGAGAATGTCCTTACTTAGTTGCGAATTTGGTTCTGATCTGCCTTCAGACTTGGAATCATTC
ATTACTGTCATTACTTTTAGCCTATTCATTTTCTTCCTTGCCATCAGGTACAGGGATTTGACCACAGAGT
GTTGAAGGGGTGCATCGTCCGCTCTCGTAATAACCCTCCGATACTATTTTCATTGTTGGCACGTTGCACT
GAAAAGGGCACTTGGCACTGTGCACTTTTAATGTTTTCATTTTTCATCATATCATCATATGGCATTTCAT
AATGTTGACTCTTGTAGTTGAAGATTGAGTTTATCGTGTTACTGTTTCGTCTACCTTTCATATTATCAAT
CAGGTTGCGGTGTTGCATGTGGGAGATAGGACTGCCCGATCTTCTTCTTCTCGATTCTCCACTAAAATCC
TGTCTTTTATCGCTATCCAGCACACGATTGAGCTGTGAATTGCCGCACTTTTTAGGATAACCATCCTTGG

Introduction

- We compare two strands of DNA to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

- We can define similarity in many different ways.

- For example, we can say that the two DNA strands are similar if one is a substring of the other.

- Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Edit Distance Problem)

Introduction

- Yet another way to measure similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2.

- These bases must appear in the same order, but not necessarily consecutively.

- The longer the strand S_3 we can find, the more similar S_1 and S_2 are.

- In our example, the longest strand S_3 is:
- $S_3 = \mathrm{GTCGTCGGAAGCCGGCCGAA}$

- We call this notion of similarity the longest common subsequence (LCS).

Subsequence

- A subsequence of given sequence is just the given sequence with 0 or more elements left out.

- Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

- For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2,3,5,7 \rangle$.

Common Subsequence

- Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

- For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y.

- There is a longer common subsequence: $\langle B, C, B, A \rangle$. This is the longest common subsequence of X and Y. Its length is 4.
  - X and Y have no common subsequence of length 5 or higher.

Longest Common Subsequence Problem

- The input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and the goal is to find a maximum-length common subsequence of X and Y.

- We solve this problem using dynamic programming.

LCS: brute-force approach?

- We can solve the LCS problem with a brute-force approach: enumerate all subsequences of X and check each subsequence to see whether it is a also a subsequence of Y, keeping track of the longest subsequence we find.

- Since each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X.

- Because X has 2^m subsequences, this approach requires exponential time → inefficient.

Step 1: Characterizing a Longest Common Subsequence

- The LCS problem has an optimal-substructure property.
- We define the ith prefix of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

- For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$.
  - X_0 is the empty sequence.

Step 1: Characterizing a Longest Common Subsequence

- Theorem 14.1 (Optimal substructure of an LCS)

- Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.

3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Proof of Theorem 14.1

- Case 1
  - If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a longest common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$.
  - Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.
- Case 2
  - If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.
- Case 3: the proof is symmetric to case 2.

Step 2: A Recursive Solution

- If $x_m = y_n$, we first find an LCS of X_{m-1} and Y_{n-1}. Then, we append $x_m = y_n$ to this LCS to obtain an LCS of X and Y.

- If $x_m \neq y_n$, we need to solve two subproblems:
  - finding an LCS of X_{m-1} and Y
  - finding an LCS of X and Y_{n-1}
- whichever of these two LCSs is longer is an LCS of X and Y.

- Let us define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j.
- Then, the recursive formula is:

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

Step 3: Computing the Length of an LCS

- The LCS problem has $\Theta(mn)$ distinct subproblems.
- Dynamic programming can compute the solutions bottom-up.

```
LCS-LENGTH(X, Y, m, n)
let b[1 : m, 1 : n] and c[0 : m, 0 : n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m          // compute table entries in row-major order
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Step 3: Computing the Length of an LCS

- The c table and the b table

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 3: Computing the Length of an LCS

- The running time of procedure LCS-LENGTH is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Step 4: Constructing and LCS

- Using the b table, we can construct the longest common subsequence.

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return                    // the LCS has length 0
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i                 // same as y_j
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

Improving the Code

- In the LCS algorithm, we can eliminate the b table altogether.
- Each $c[i, j]$ entry depends on only three other table entries: $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$.
- Given the value of $c[i, j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b.

- If we need only the length of an LCS, we can reduce the c table as well.
- We need to use only two rows of table c at a time: the row being computed and the previous row.

Extra: Other Problems

Problem: World Series Odds

- Dodgers and Yankees are playing the World Series in which either team needs to win n games first.
- Suppose that each team has a 50% chance of winning any particular game.
- Let $p(i, j)$ be the probability that if Dodgers needs i games to win, and Yankees needs j games, Dodgers will eventually win the series.
- Ex: if $n = 4$, $p(2, 3) = 11/16$

- Compute $p(i, j)$ for an arbitrary n. $(0 \le i, j \le n)$

World Series Odds – Divide-and-Conquer

- Recursive formulation

$$P(i,j) = \begin{cases} 1, & \text{if } i = 0 \text{ and } j > 0 \\ 0, & \text{if } i > 0 \text{ and } j = 0 \\ \frac{P(i-1,j)+P(i,j-1)}{2}, & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

- Worst-case time complexity
  - T(n): maximum time taken by a call to p(i, j)
  - T(n) is exponential!

- What is the problem of this approach?

World Series Odds – Dynamic Programming

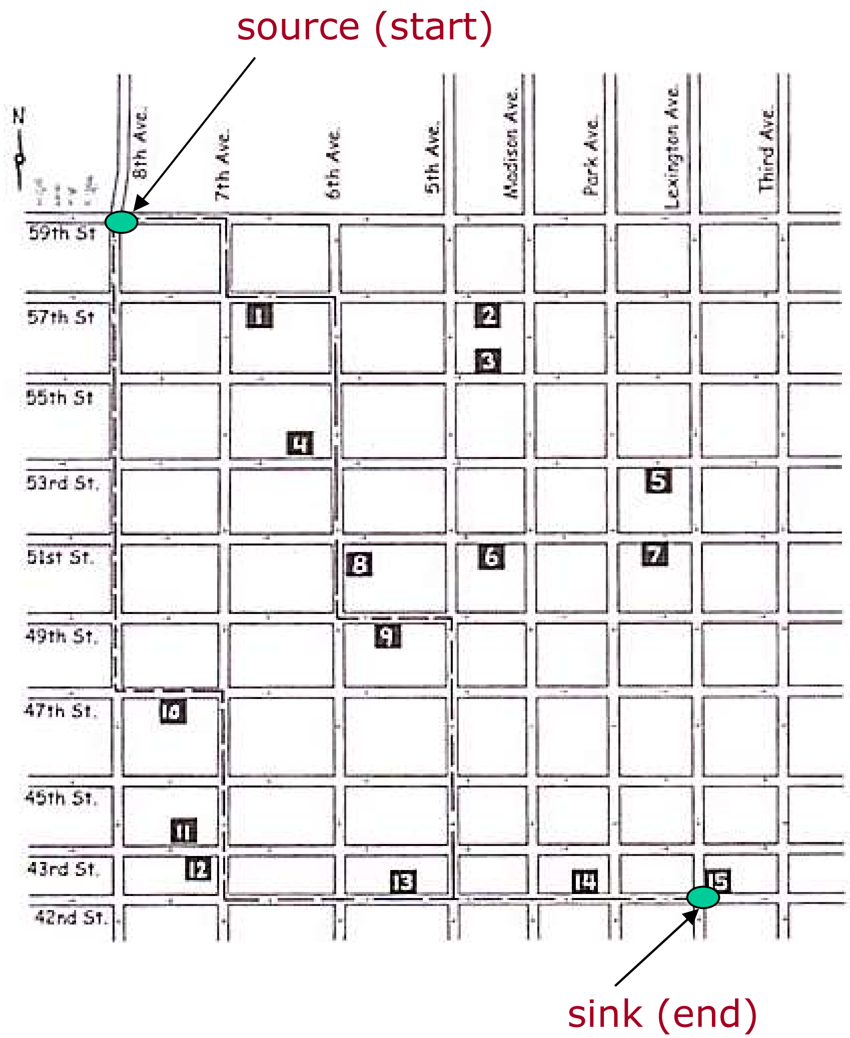
- Instead of computing the same problem repeatedly, fill in a table as suggested below.

4	1	15/16	13/16	21/32	1/2	
3	1	7/8	11/16	1/2	11/32	
2	1	3/4	1/2	5/16	3/16	
1	1	1/2	1/4	1/8	1/16	
0		0	0	0	0	
j \ i	0	1	2	3	4	

- Time Complexity: $O(n^2)$
  - Much better than divide-and-conquer!

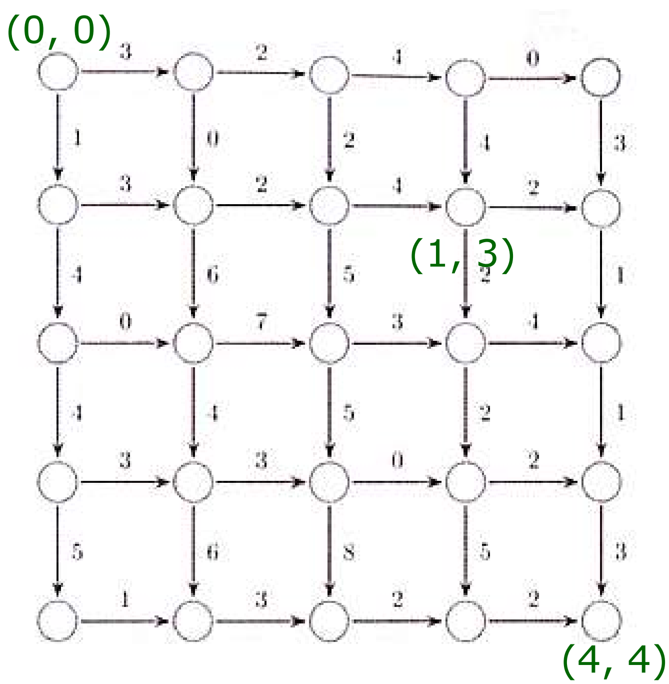
The Manhattan Tourist Problem

- Problem
  - Given two street corners in the borough of Manhattan in New York City, find the path between them with the maximum number of attractions, that is, a path of maximum overall weight.
  - Assume that a tourist may move either EAST or SOUTH only.

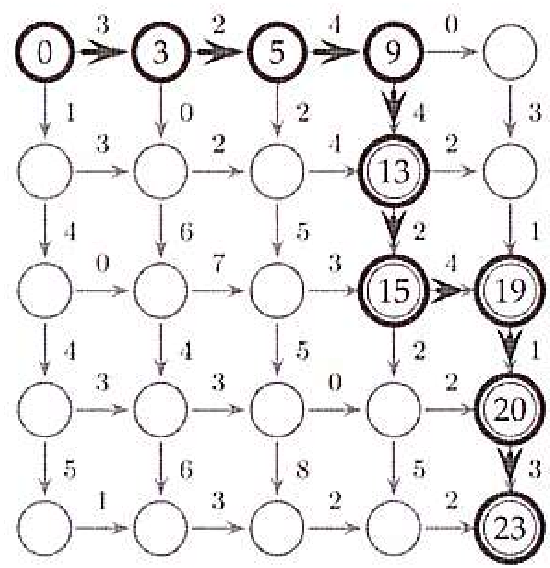


The Manhattan Tourist Problem

- Formal description
  - Given a weighted grid G of size (n, m) with two distinguished vertices, a source (0, 0) and a sink (n, m), find a longest path between them in its weighted graph

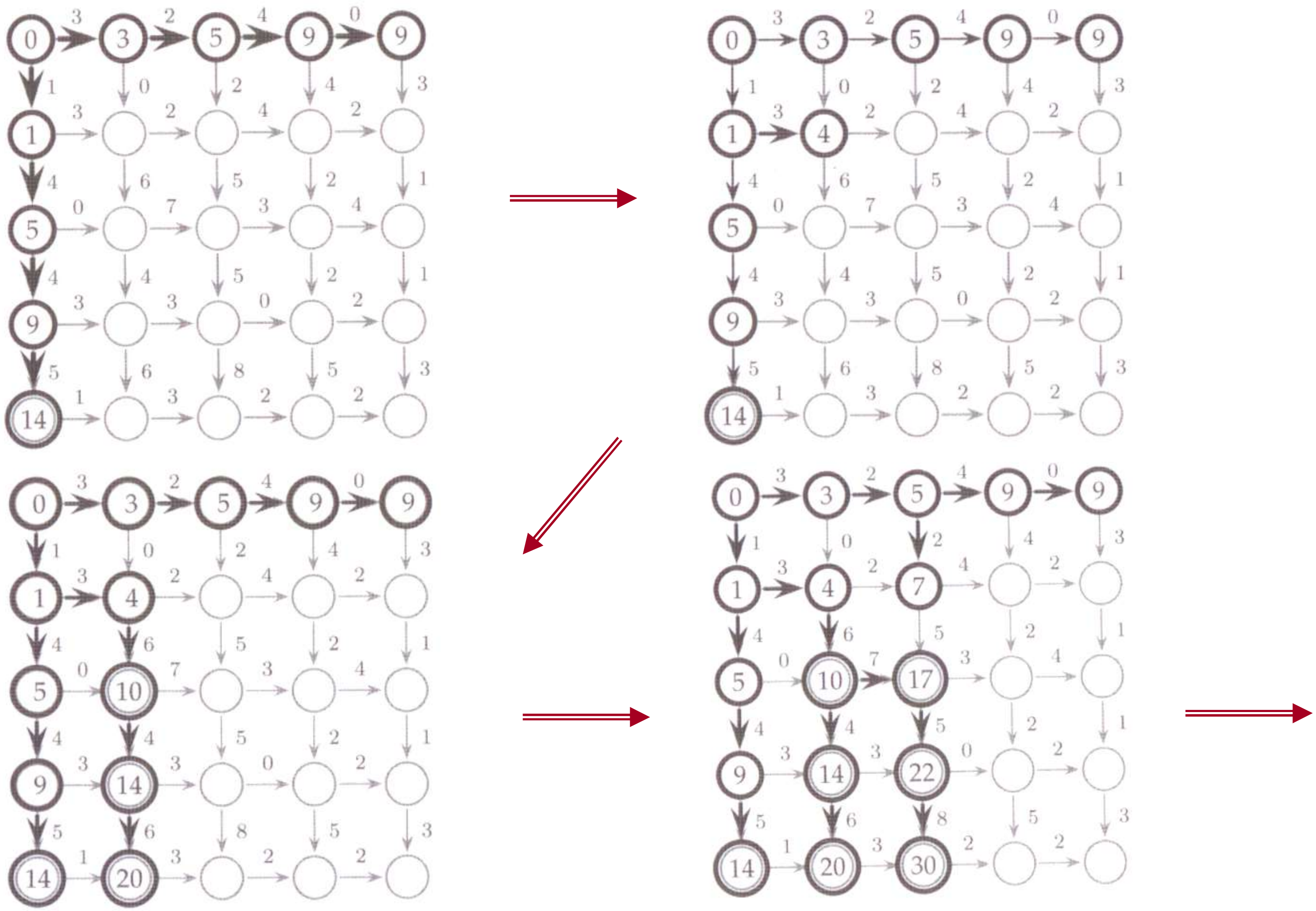


An example grid of size (4, 4)

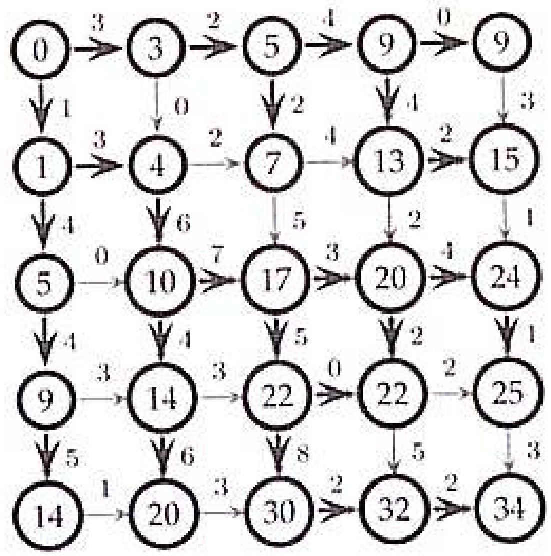


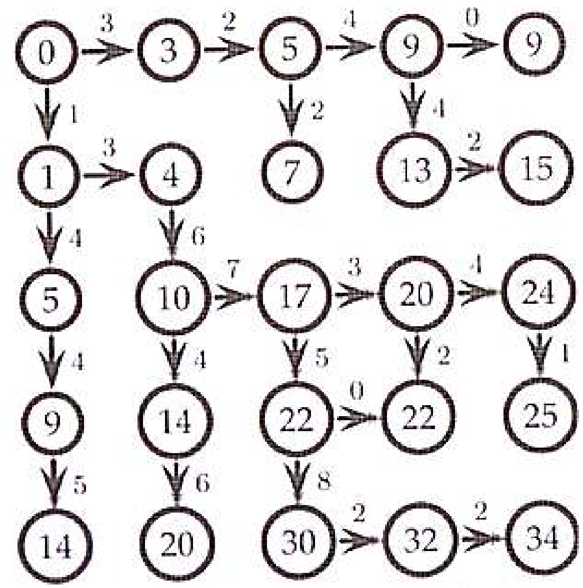
A possible selection determined by a greedy approach

Dynamic Programming: A pictorial description



Dynamic Programming: A pictorial description (cont.)





- Found longest path from source to every vertex in the grid

Dynamic Programming: Algorithm

- Recursive relation

$$s_{i,j} \equiv \text{ the length of the longest path from } (0,\,0) \text{ to } (i,j)$$

$$\boxed{s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{weight between } (i,j-1) \text{ and } (i,j) \end{cases}}$$

- Algorithm →

ManhattanTourist($\overset{\downarrow}{\mathbf{w}}, \vec{\mathbf{w}}, n, m$)

$s_{0,0} \leftarrow 0$
for $i \leftarrow 1$ to n
    $s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
for $j \leftarrow 1$ to m
    $s_{0,j} \leftarrow s_{0,j-1} + \vec{w}_{0,j}$
for $i \leftarrow 1$ to n
    for $j \leftarrow 1$ to m
        $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} \\ s_{i,j-1} + \vec{w}_{i,j} \end{cases}$
return $s_{n,m}$

- Time Complexity?

Edit Distance

- **Problem:** Given two strings $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$ $(n, m \geq 0)$, consider the problem of transforming A into B by either inserting a character into A, deleting a character from A, or replacing a character in A by another. Suppose that the costs per insertion, deletion, and replacement are c_{ins}, c_{del}, and c_{repl}, respectively. Then, find the minimum cost of transforming A into B.

```
A = S - N O W Y
B = S U N N - Y
```

```
A = - S N O W - Y
B = S U N - - N Y
```

```
A = E X P O N E N - T I A L
B = - - P O L Y N O M I A L
```

Let $C[i, j]$ be the minimum cost to transform $A_i = a_1 a_2 \cdots a_i$ into $B_j = b_1 b_2 \cdots b_j$ for $i \in [0, n]$ and $j \in [0, m]$.

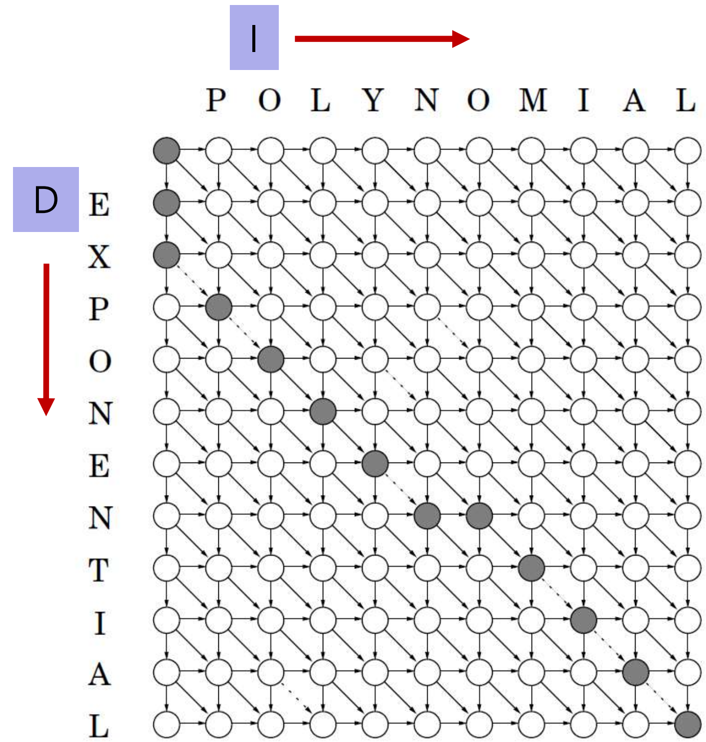
$$
\begin{aligned}
C[0,0] &= 0 \\
C[i,0] &= i * c_{del}, \quad i \in [1,n] \\
C[0,j] &= j * c_{ins}, \quad j \in [1,m] \\
C[i,j] &= \left\{ \begin{array}{ll} C[i-1,j-1] & \text{if } a_i = b_j \\ \min \left\{ \begin{array}{l} C[i,j-1] + c_{ins} \\ C[i-1,j] + c_{del} \\ C[i-1,j-1] + c_{repl} \end{array} \right\} & \text{if } a_i \neq b_j \end{array} \right\}, \quad i \in [1,n], j \in [1,m]
\end{aligned}
$$

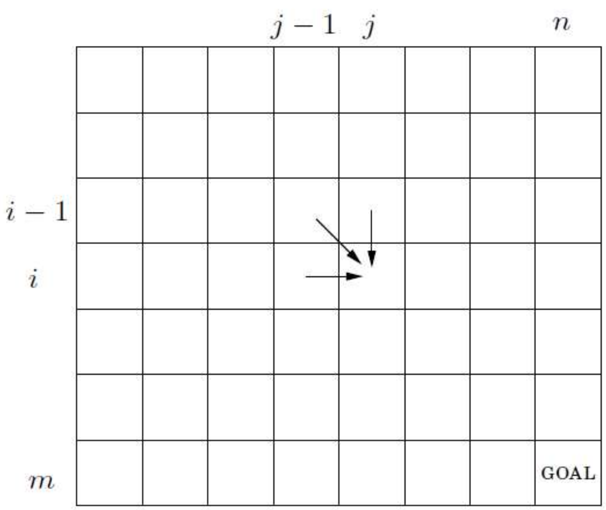
O(mn)

Edit Distance

- Example

A = E X P O N E N – T I A L
B = – – P O L Y N O M I A L





		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr