[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.10.13 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

[주제 3]
Divide-and-Conquer Techniques
and
Sorting Techniques

Master Theorem 1

[Neapolitan 2.8]

- Let *a*, *b*, and *c* be nonnegative constants. The solution to the recurrence $\boldsymbol{T(1) = 1}$, and $\boldsymbol{T(n) = aT(n/c) + bn}$, for $n > 1$ for n a power of c is

  ① $T(n) = O(n)$, if $a < c$,

  ② $T(n) = O(n \log n)$, if $a = c$,

  ③ $T(n) = O(n^{\log_c a})$, if $a > c$.

Prove this by induction!

- Avoid divided-and-conquer if, for example,
  - An instance of size *n* is divided into two or more instances each almost of size *n*.
  - An instance of size *n* is divided into almost *n* instance of size *n*/*c*, where *c* is a constant.

The divide-and-conquer strategy often leads to efficient algorithms, although not always!

Master Theorem 2

Theorem If $T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$ for $a \geq 1, b > 1,$ and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d, \\ O(n^d) & \text{if } a < b^d, \\ O(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

※ a : the rate of subproblem proliferation — Bad!

(얼마나 빠른 속도로 subproblem의 개수가 증가하는가?)

b^d : the rate of work shrinkage — Good!

(각 subproblem당 요구되는 작업량이 얼마나 빠른 속도로 감소하는가?)

<L0> <L1> <L2>

$$\begin{cases} 1 \rightarrow a \rightarrow a^2 \rightarrow \cdots \\ n^d \rightarrow \left(\frac{n}{b}\right)^d = \frac{1}{b^d} \cdot n^d \rightarrow \left(\frac{1}{b^d}\right)^2 \cdot n^d \rightarrow \cdots \end{cases}$$

Finding the Closest Pair of 2D Points

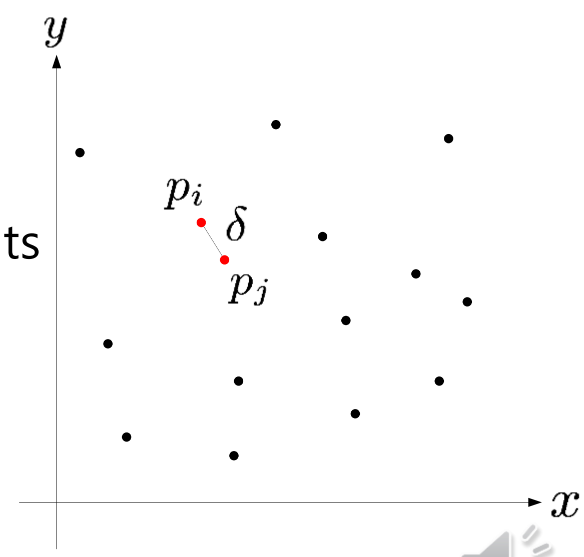
- **Problem**
  - Given n points in the plane, find the pair that is closest together.

- **Notation**

$$P = \{\, p_1, p_2, \cdots, p_n \,\}, \text{ where } p_i = (x_i, y_i)$$

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$\boxed{\text{Given a 2D point set } P \text{, find a pair of points } p_i,\ p_j \in P \text{ that minimizes } d(p_i, p_j).}$



- **Naïve algorithm**
  - Compute the distance between each pair of points and take the minimum → $O(n^2)$ - time

Applying the Divide-and-Conquer Strategy [Shamos and Hoey]

- Simple assumption for an easy explanation
  - No two points in P have the same x-coordinate or the same y-coordinate.

- General idea

[Preprocessing]

- Build a list P_x in which all the points in P have been sorted by increasing x-coordinate → $O(n \log n)$
- Build another list P_y in which all the points in P have been sorted by increasing y-coordinate → $O(n \log n)$

[Recursion for P with $|P| = n$]

- **[Divide]** Partition P into two subsets Q and R → $O(n)$
- **[Conquer]** Find the closest pairs in Q and R, respectively → $2T(n/2)$
- **[Combine]** Use this information to get the closest pair in P → $O(n)$

✓ Time-complexity

$$O(n \log n) + T(n) \text{ where } T(n) = c*n + 2T(n/2) \rightarrow \boldsymbol{O(n \log n)}$$

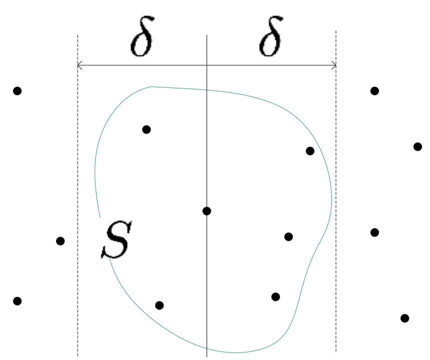
- The stage **[Divide]:** *Partition P into two subsets Q and R.*
  - Create Q and R, where
    - Q: the set of points in the first ceil(n/2) positions of the list P_x (the "left half"), and
    - R: the set of points in the final floor(n/2) positions of the list P_x (the "right half").
  - Furthermore, create Q_x, Q_y, R_x, and R_y, where
    - Q_x consisting of the points in Q sorted by increasing x-coordinate,
    - Q_y consisting of the points in Q sorted by increasing y-coordinate,
    - R_x consisting of the points in R sorted by increasing x-coordinate, and
    - R_y consisting of the points in R sorted by increasing y-coordinate.
  - ✓ Can be done in $O(n)$.

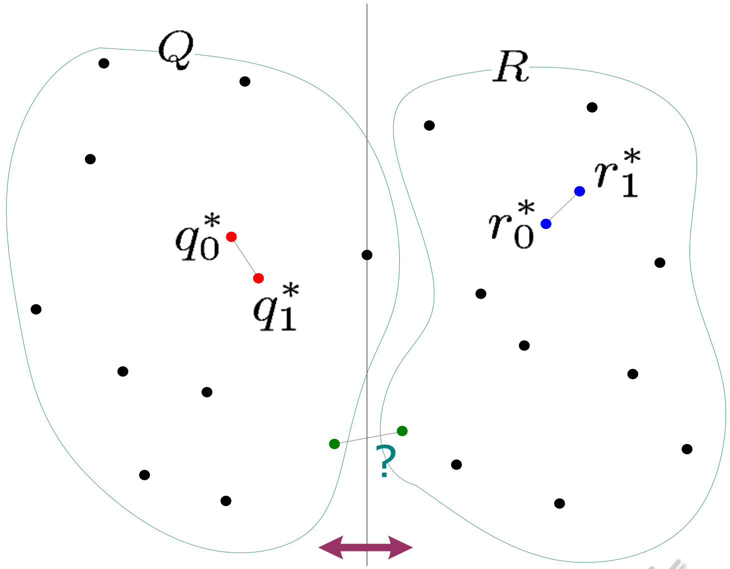
- The stage **[Conquer]:** *Find the closest pairs in Q and R, respectively.*
  - Recursively determine a closest pair (q_0^*, q_1^*) of points in Q.
  - Recursively determine a closest pair (r_0^*, r_1^*) of points in R.
  - ✓ Can be done in $2T(n/2)$.

- The stage **[Combine]:** *Use the obtained info. to get the closest pair in P.*
  - Question: Are there points $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$?
    - ✓ How can we answer this question in ***linear time***?
  - **[Fact 1]** (Why?)
    - If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L.
  - **[Fact 2]**
    - There exist $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$ if and only if there exist s, $s' \in S$ for which $d(s,s') < \delta$.

x^*: the x-coordinate of the rightmost point in Q

$\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$





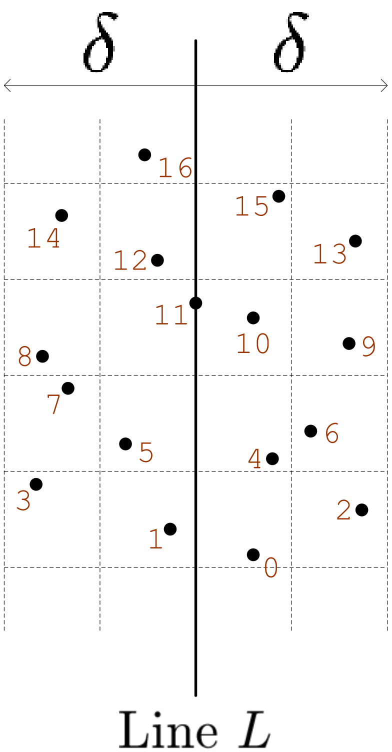
– **[Fact 3]**

- If $s,\ s' \in S$ have the property that $d(s,s') < \delta$, then s and s' are within 15 positions of each other in the sorted list S_y.

  ✓ S_y: the list consisting of the points in S sorted by increasing y-coordinate.

  – Each box contains at most one point of S. (Why?)

  – If two points in S are at least 16 positions apart in S_y, …



– **[Merge]**

$\boxed{O(n)}$

1. For each s in S_y, compute its distance to each of the next 15 points in S_y.
2. Let $s,\ s'$ be the pair achieving the minimum of these distances.
3. Compare $d(s,s')$ with δ.

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p*_0, p*_1) = Closest-Pair-Rec(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q*_0, q*_1) = Closest-Pair-Rec(Q_x, Q_y)
  (r*_0, r*_1) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q*_0, q*_1), d(r*_0, r*_1))
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
    to each of next 15 points in S_y
    Let s, s' be pair achieving minimum of these distances
    (O(n) time)

  If d(s,s') < δ then
    Return (s,s')
  Else if d(q*_0, q*_1) < d(r*_0, r*_1) then
    Return (q*_0, q*_1)
  Else
    Return (r*_0, r*_1)
  Endif
```

[Divide]

[Conquer]

[Combine]

[주제 4]
Dynamic Programming

Algorithm Design Techniques

- Divide-and-Conquer Method
- Dynamic Programming Method
- Greedy Method
- Backtracking Method
- Local Search Method
- Branch-and-Bound Method
- Etc.

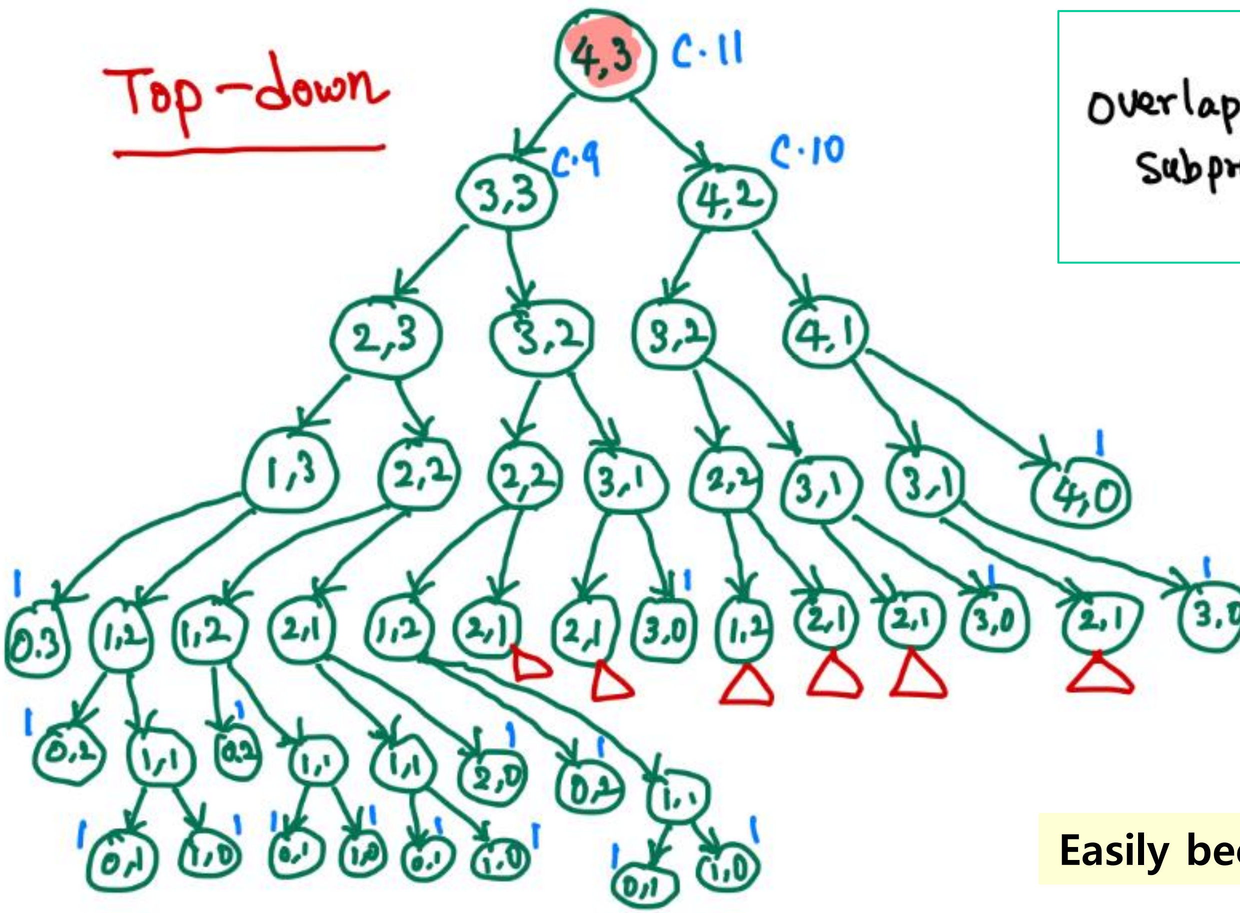
Dynamic Programming: Overview

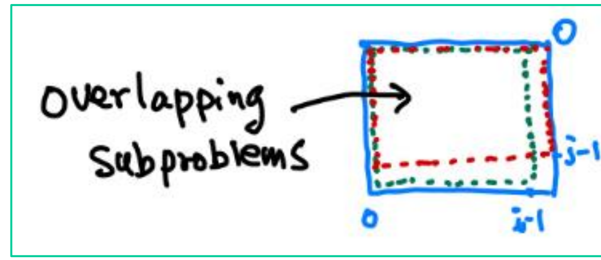
From Wikipedia

- **Dynamic programming** is both a mathematical optimization method and a computer programming method.
  - A complicated problem is **broken down into simpler sub-problems in a recursive manner**.
  - **Overlapping subproblems:** A problem is broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.
  - **Optimal substructure:** A solution to a given optimization problem can be constructed efficiently from optimal solutions of its subproblems.
  - When applicable, the method **takes far less time than other methods** that don't take advantage of the subproblem overlap **like the divide-and-conquer technique**.

Two Approaches for Recursive Formulation

$$\begin{cases} T(i,j) = T(i-1,j) + T(i,j-1) + c\cdot(2i+j), & i,j \geq 1 \\ T(i,0) = T(0,j) = 1, & i,j \geq 0 \end{cases}$$

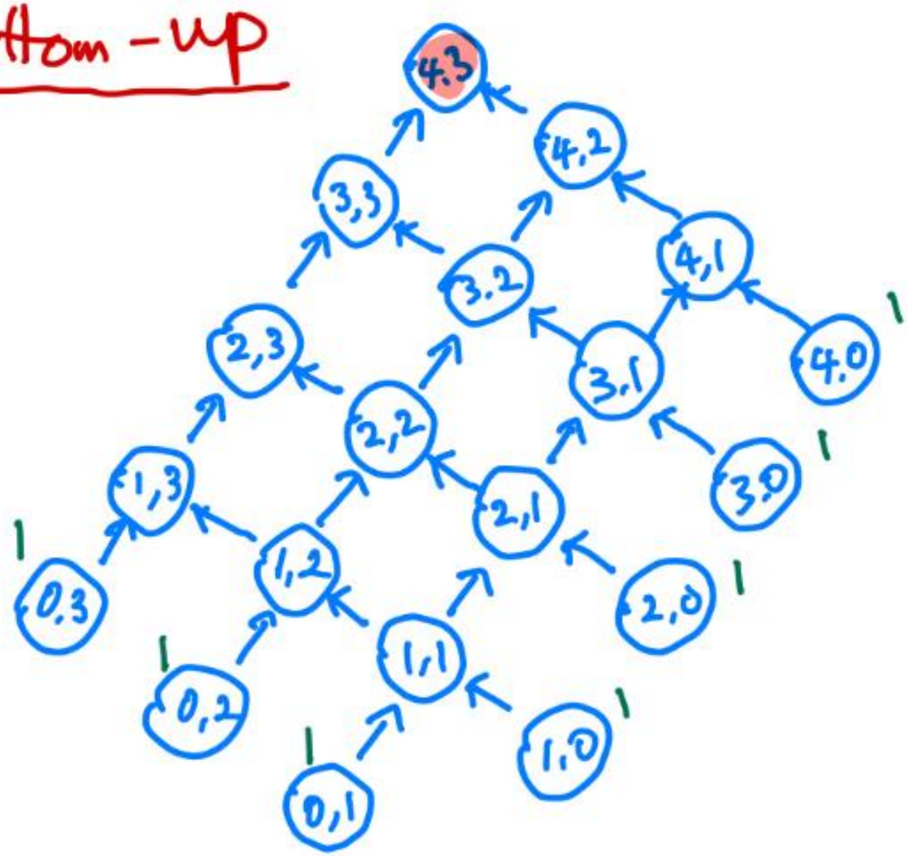




Easily becomes exponential!

$$\begin{cases} T(i,j) = T(i-1,j) + T(i,j-1) + c\cdot(2i+j), & i,j \geq 1 \\ T(i,0) = T(0,j) = 1, & i,j \geq 0 \end{cases}$$

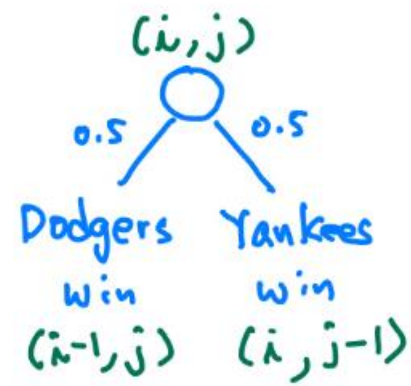
Bottom-up



Often much more efficient!

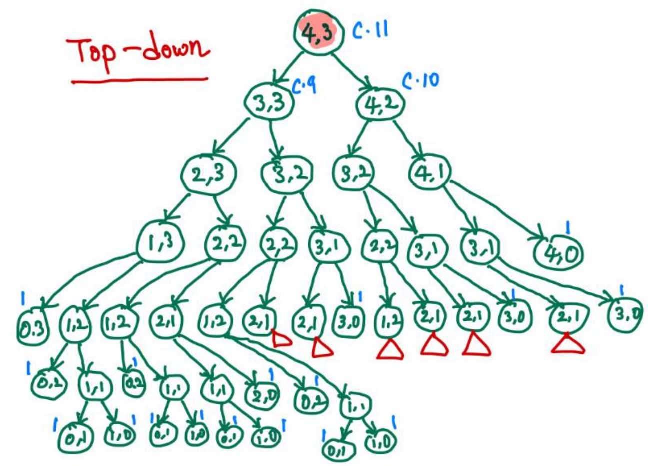
World Series Odds

- **Problem**
  - 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 game.
  - Let $\boldsymbol{P(i, j)}$ be the probability that if Dodgers needs i games to win, and Yankees needs j games, Dodgers will eventually win the Series.
  - Ex: $P(2, 3) = 11/16$
  - **Compute $\boldsymbol{P(i, j)}$ $(0 \leq i, j \leq n)$ for an arbitrary $\boldsymbol{n}$.**



☹ A Divide-and-Conquer Approach

- **Recursive formulation**

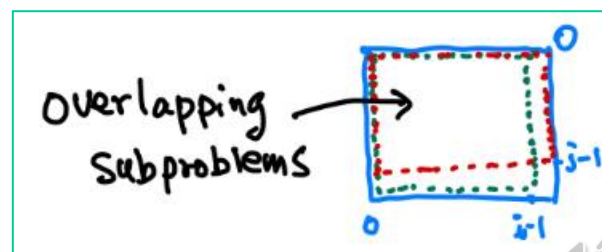


$$P(i,j) = \begin{cases} 1, & \text{if } i = 0 \text{ and } j > 0 \\ 0, & \text{if } i > 0 \text{ and } j = 0 \\ \frac{P(i-1,j)+P(i,j-1)}{2}, & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

- If we solve this recurrence relation in the divide-and-conquer way, …
  - Let $T(n)$ be the maximum time taken by a call to $P(i, j)$, where $i + j = n$. Then we can prove that $T(n)$ is exponential!

$$\left.\begin{array}{lcl} T(1) & = & 1 \\ T(n) & = & 2T(n-1) + c \end{array}\right\} \longrightarrow O(2^n)$$

- What is the problem of this approach?



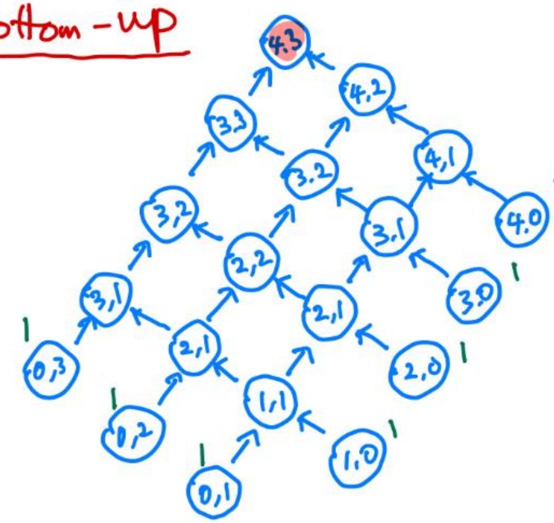
☺ A Dynamic Programming Approach

- Instead of computing the same 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	

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

Bottom-up



- **Time Complexity**
  – For input size (m, n), computing P(m, n) takes O(mn)-time.
  ➢ By far better than the Divide-and-Conquer approach.

Dynamic Programming

- When the **divide-and-conquer** approach produces **an exponential algorithm** where **the same sub-problems are solved iteratively**,
  1) Take the recursive relation from the divide-and-conquer algorithm, and
  2) **replace the recursive calls with table lookups by recording a value in a table entry instead of returning it.**

Top-down → Bottom-up

- **Three elements to consider in designing a dynamic programming algorithm**
  - **Recursive relation**
    - **Optimal substructure**
  - **Table setup**
  - **Table fill order**

$$B(i,j) = \begin{cases} B(i-1,j-1) + B(i-1,j), & \text{if } 0 < j < i \\ 1, & \text{if } j = 0 \text{ or } j = i \end{cases}$$

The Manhattan Tourist Problem

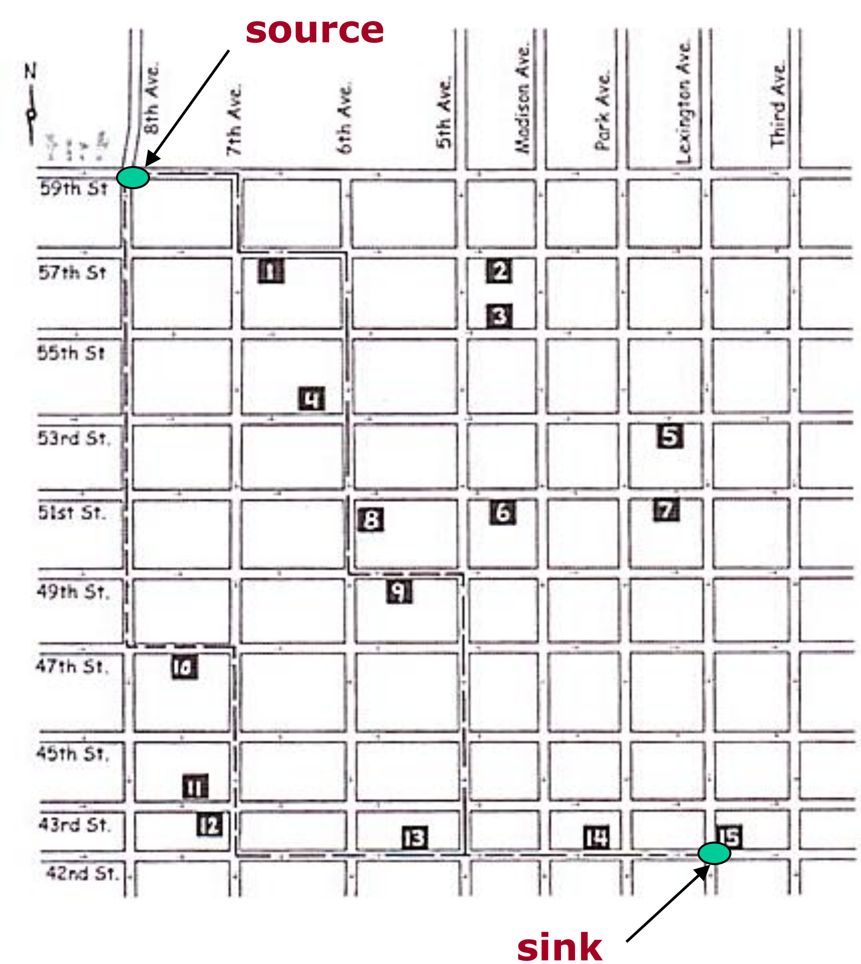
Courtesy of [Jones & Pevzner 6.3]

- **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.

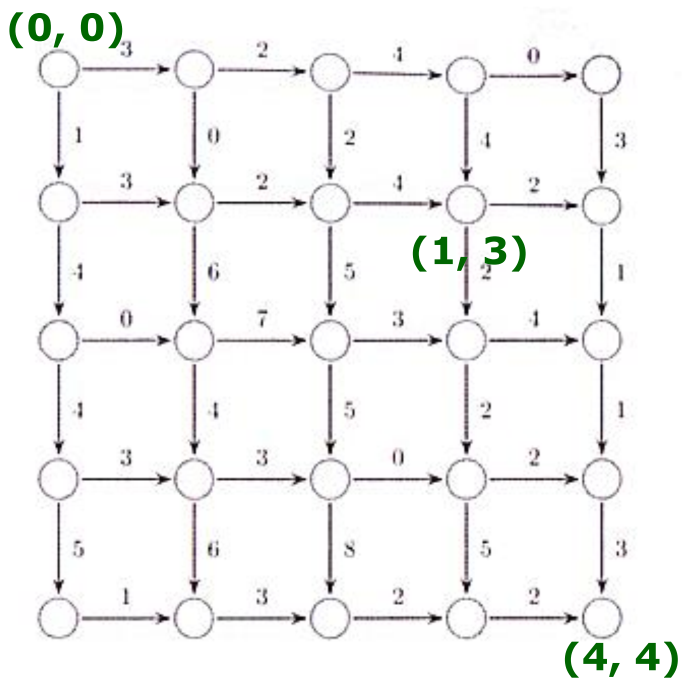
  ✓ Assume that a tourist may **move either to east or to south only**.

- **A brute force approach**
  - Search among all paths in the grid for the longest path!

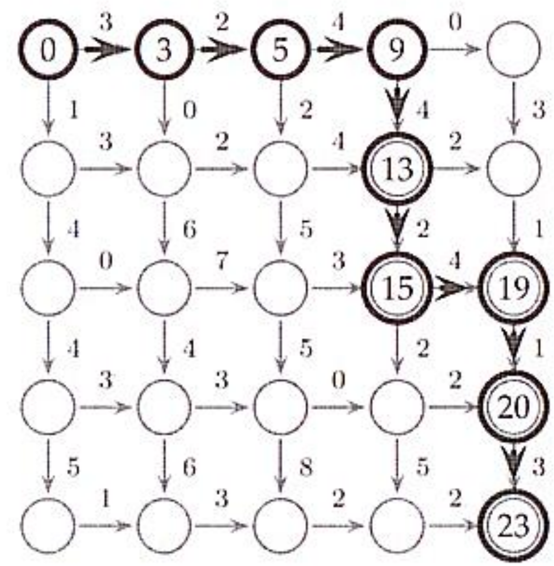
- **A greedy approach**
  - 다음 강의 주제



- **A formal description of this problem**
  - Given a weighted graph (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.

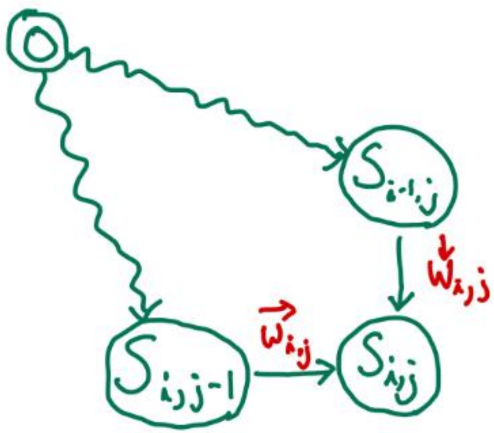


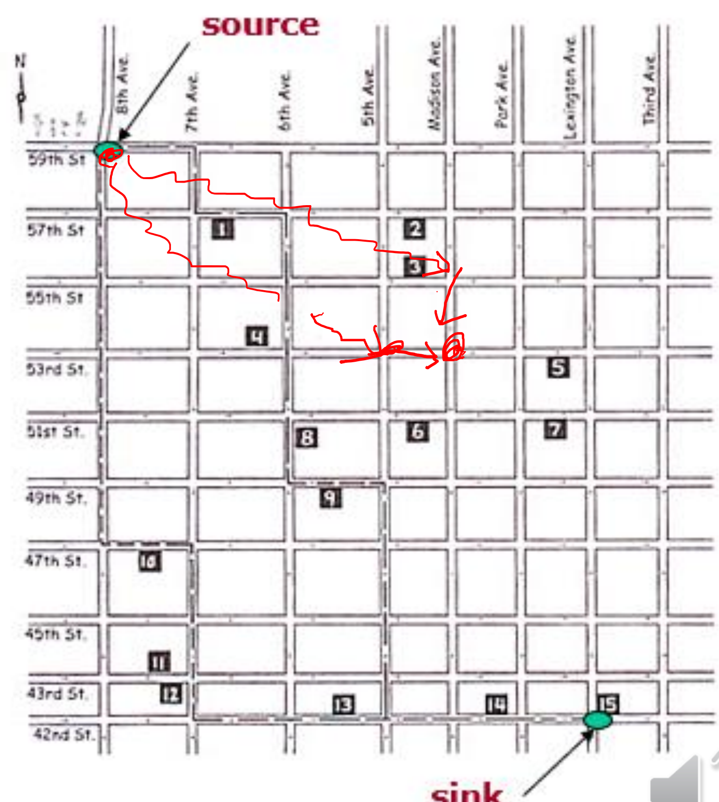
An example grid of size (4, 4)



A possible selection determined by a greedy approach

- **Basic idea**
  - How can you use the solutions of smaller problems to build a solution of a problem?





A given optimization problem can be constructed efficiently from optimal solutions of its subproblems.

→ **optimal substructure**