[CSE3081(2반)] 알고리즘 설계와 분석

2020학년도 2학기

강의자료

(2020.10.27 화요일)

서강대학교 공과대학 컴퓨터공학과
임 인 성 교수

본 강의에서 제작하여 제공하는 PDF 파일, 동영상, 그리고 예제 코드 등의 강의 자료의 저작권은 특별히 명기되어 있지 않은 한 서강대학교에 있습니다.

본인의 학습 목적 외에 공개된 장소에 올리거나 타인에게 배포하는 등의 행위를 금합니다. 협조 부탁합니다.

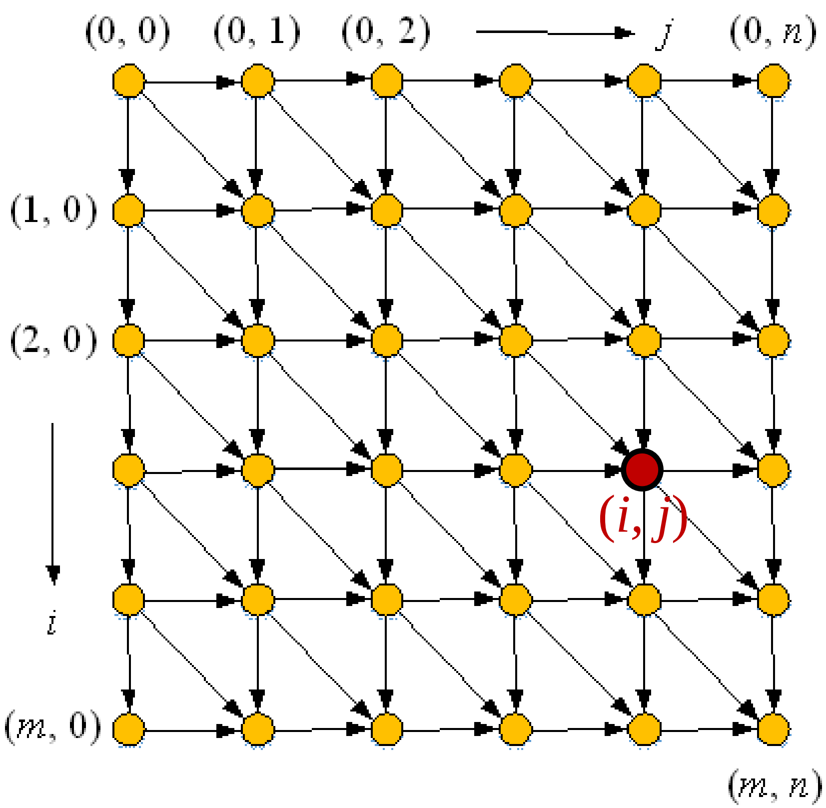
[주제 4]

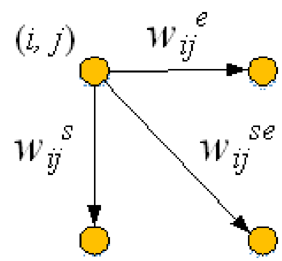
Dynamic Programming

Principles of Dynamic Programming

$$C_{ij} = \text{the cost of the shortest path from } (0,0) \text{ to } (i,j)$$

$$\longrightarrow \quad C_{ij} = \min\{C_{i-1,j} + w^{s}_{i-1,j},\ C_{i-1,j-1} + w^{se}_{i-1,j-1},\ C_{i,j-1} + w^{e}_{i,j-1}\}$$





- **Recursive formulation**
- **Optimal substructure**
- **Overlapping subproblems**
- **Bottom-up approach**

Optimal Substructure

From Wikipedia

- Dynamic programming algorithms are **often used for optimization.**

- A problem is said to **have optimal substructure**
  - if a solution to a given optimization problem can be constructed efficiently from optimal solutions of its subproblems.

- Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure.
  - Such optimal substructures are **usually described by means of recursion**.

$$C_{ij} = \min\{C_{i-1,j} + w^{s}_{i-1,j},\ C_{i-1,j-1} + w^{se}_{i-1,j-1},\ C_{i,j-1} + w^{e}_{i,j-1}\}$$

From Wikipedia

Overlapping Subproblems

- To solve a problem, we often need to **solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution**.

- A problem is said to **have overlapping subproblems** if
  - the problem can be 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.

$$C_{ij} = \min\{C_{i-1,j} + w^{s}_{i-1,j},\ C_{i-1,j-1} + w^{se}_{i-1,j-1},\ C_{i,j-1} + w^{e}_{i,j-1}\}$$

- The dynamic programming approach seeks to **solve each subproblem only once**, thus reducing the number of computations: (i) once the solution to a given subproblem has been computed, it is stored or **"memoized"**: (ii) the next time the same solution is needed, it is simply **looked up**.

- This approach is **especially useful when the number of repeating subproblems grows exponentially** as a function of the size of the input.

$$C_{ij} = \min\{C_{i-1,j} + w^{s}_{i-1,j},\ C_{i-1,j-1} + w^{se}_{i-1,j-1},\ C_{i,j-1} + w^{e}_{i,j-1}\}$$

From Wikipedia

- **If a problem can be solved by combining optimal solutions to non-overlapping sub-problems**, the strategy is called "divide-and-conquer" instead. This is why merge sort and quick sort are not classified as dynamic programming problems.

$$C_{ij} = \min\{C_{i-1,j} + w^{s}_{i-1,j},\ C_{i-1,j-1} + w^{se}_{i-1,j-1},\ C_{i,j-1} + w^{e}_{i,j-1}\}$$

The Checkerboard Problem

Courtesy of Wikipedia

- **Restrictions**
  - A checker can start at any square on the first row ($i = 1$).
  - It can move only diagonally left forward, diagonally right forward, or straight forward.
  - It must pay the cost $c[i][j]$ when visiting the (i, j)-position.

i \ j	1	2	3	4	5
1	7	3	5	6	1
2	2	6	7	0	2
3	3	5	7	8	2
4	7	6	1	1	4
5	6	7	4	7	8

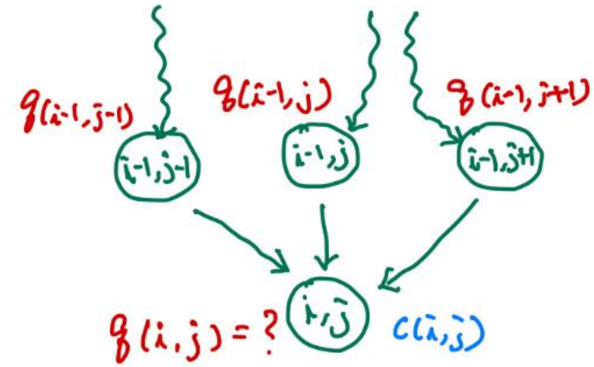
i \ j	1	2	3	4	5
1	7	3	5	6	1
2	2	6	7	0	2
3	3	5	7	8	2
4	7	6	1	1	4
5	6	7	4	7	8

Cost table c[i][j]

- **Problem**
  - Given a checkerboard with nxn squares, and a cost function $c[i][j]$, **find the minimum-cost path from the first row to the last row.**

i \ j	1	2	3	4	5
1	7	3	5	6	1
2	2	6	7	0	2
3	3	5	7	8	2
4	7	6	1	1	4
5	6	7	4	7	8

Cost table c[i][j]



- **Optimal substructure**

$q(i,j)$: the minimum cost to reach squrare (i,j)

$$q(i,j)=\begin{cases}\infty, & \text{if } j<1 \text{ or } j>n\\ c(i,j), & \text{if } i=1\\ \min\{\ q(i-1,j-1),q(i-1,j),q(i-1,j+1)\ \}+c(i,j), & \text{otherwise.}\end{cases}$$

i \ j	1	2	3	4	5
1	7	3	5	6	1
2	5	9	10	1	3
3	8	10	8	9	3
4	15	14	9	4	7
5	20	16	8	11	12

Q table q[i][j]

i \ j	1	2	3	4	5
1	0	0	0	0	0
2	1	0	-1	1	0
3	0	-1	1	0	-1
4	0	-1	0	1	0
5	1	1	1	0	-1

P table p[i][j]

```c
#include <stdio.h>
#define N 5
#define INFTY 100000
int c[N+1][N+2] = { -1, -1, -1, -1, -1, -1, -1,
    -1, 7, 3, 5, 6, 1, -1,  -1, 2, 6, 7, 0, 2, -
    1,   -1, 3, 5, 7, 8, 2, -1,  -1, 7, 6, 1, 1,
    4, -1,    -1, 6, 7, 4, 7, 8, -1};
int p[N+1][N+2], q[N+1][N+2];

int min3(int a, int b, int c) {
    …
}

void ComputeCBCosts(int n) {
  int i, j, min;

  for (i = 1; i <= n; i++) q[1][i] = c[1][i];
  for (i = 1; i <= n; i++) {
    q[i][0] = INFTY; q[i][n+1] = INFTY;
  }
  for (i = 2; i <= n; i++) {
    for (j = 1; j <= n; j++) {
      min = min3(q[i-1][j-1], q[i-1][j],
                               q[i-1][j+1]);
      q[i][j] = min + c[i][j];
      if (min == q[i-1][j-1]) p[i][j] = -1;
      else if (min == q[i-1][j]) p[i][j] = 0;
      else p[i][j] = 1;
    }
  }
}
```

```c
void PrintShortestPath(int n, int imin) {
  printf(" (%d, %d) <-", n, imin);
  if (n == 2)
    printf(" (%d, %d)\n", 1, imin + p[n][imin]);
  else
    PrintShortestPath(n-1, imin + p[n][imin]);
}

void ComputeCBShortestPath(int n) {
  int i, imin, min;

  ComputeCBCosts(n);

  imin = 1;  min = q[n][1];
  for (i = 2; i <= n; i++) {
    if (q[n][i] < min) {
      imin = i;  min = q[n][i];
    }
  }
  printf("*** The cost of the shortest path is
          %d.\n", q[n][imin]);

  PrintShortestPath(n, imin);
}

void main(void) {
    int n;

    n = N;
    ComputeCBShortestPath(n);
}
```

i \ j	1	2	3	4	5
1	0	0	0	0	0
2	1	0	-1	1	0
3	0	-1	1	0	-1
4	0	-1	0	1	0
5	1	1	1	0	-1

P table p[i][j]

i \ j	1	2	3	4	5
1	7	3	5	6	1
2	5	9	10	1	3
3	8	10	8	9	3
4	15	14	9	4	7
5	20	16	8	11	12

Q table q[i][j]

Longest Common Subsequence (LCS)

- Definitions
  - Given a sequence $X = <x_1, x_2, ..., x_m>$, another sequence $Z = <z_1, z_2, ..., z_k>$ is a ***subsequence*** of X if there exists a strictly increasing sequence $<i_1, i_2, ..., i_k>$ of indices of X such that for all $j = 1, 2, ..., k$, we have $x_{i_j} = z_j$.
    - A subsequence of a given sequence is just the given sequence with some elements (possibly none) left out.
    - Ex: $X = <A, B, C, B, D, A, B>$, $Z = <B, C, D, B>$ (<2, 3, 5, 7>)
  - 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.
    - Ex: $X = <A, B, C, B, D, A, B>$, $Y = <B, D, C, A, B, A>$, $Z_1 = <B, C, A>$, $Z_2 = <B, C, B, A>$, $Z_3 = <B, D, A, B>$
  - Given a sequence $X = <x_1, x_2, ..., x_m>$, $X_i = <x_1, x_2, ..., x_i>$ is the ith ***prefix*** of X, for $i = 0, 1, ..., m$.
    - Ex: $X = <A, B, C, B, D, A, B>$, $X_4 = <A, B, C, B>$, X_0 = null sequence

- **Problem**
  - Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, find a *longest common subsequence* of X and Y.

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

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

(m, n)

X_{m-1}, Y

(m-1, n)

(m, n-1)

(m-1, n-1)

X_{m-1}, Y_{n-1}

X …, Y_{n-1}