Chapter 3. (Part 1)

Dynamic Programming

경북대학교 배준현 교수

(joonion@knu.ac.kr)

Contents

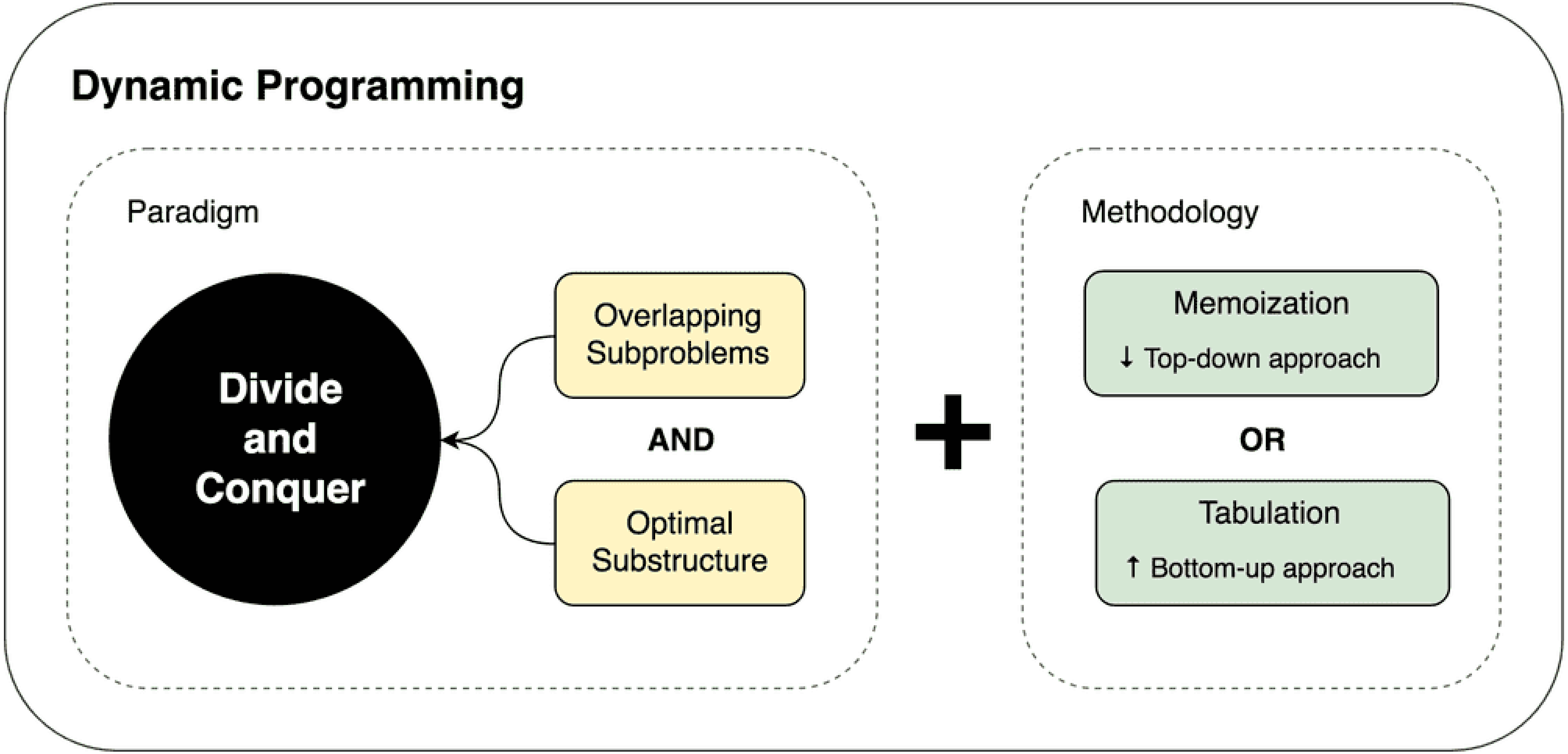
Dynamic Programming

- Dynamic Programming
  - is similar to divide-and-conquer in that
    - an instance of a problem is divided into smaller instances.
  - However, D.P. *solves small instances first*,
    - *store the results*, and later, whenever we need a result,
      - *look it up instead of recomputing* it.
  - The term "***dynamic programming***" comes from *control theory*,
    - "programming" means the use of an *array* (*table*)
      - in which a solution is constructed. (called *memoization*)
  - Dynamic Programming is a *bottom-up* approach,
    - whereas the divide-and-conquer is a *top-down* approach.

Dynamic Programming

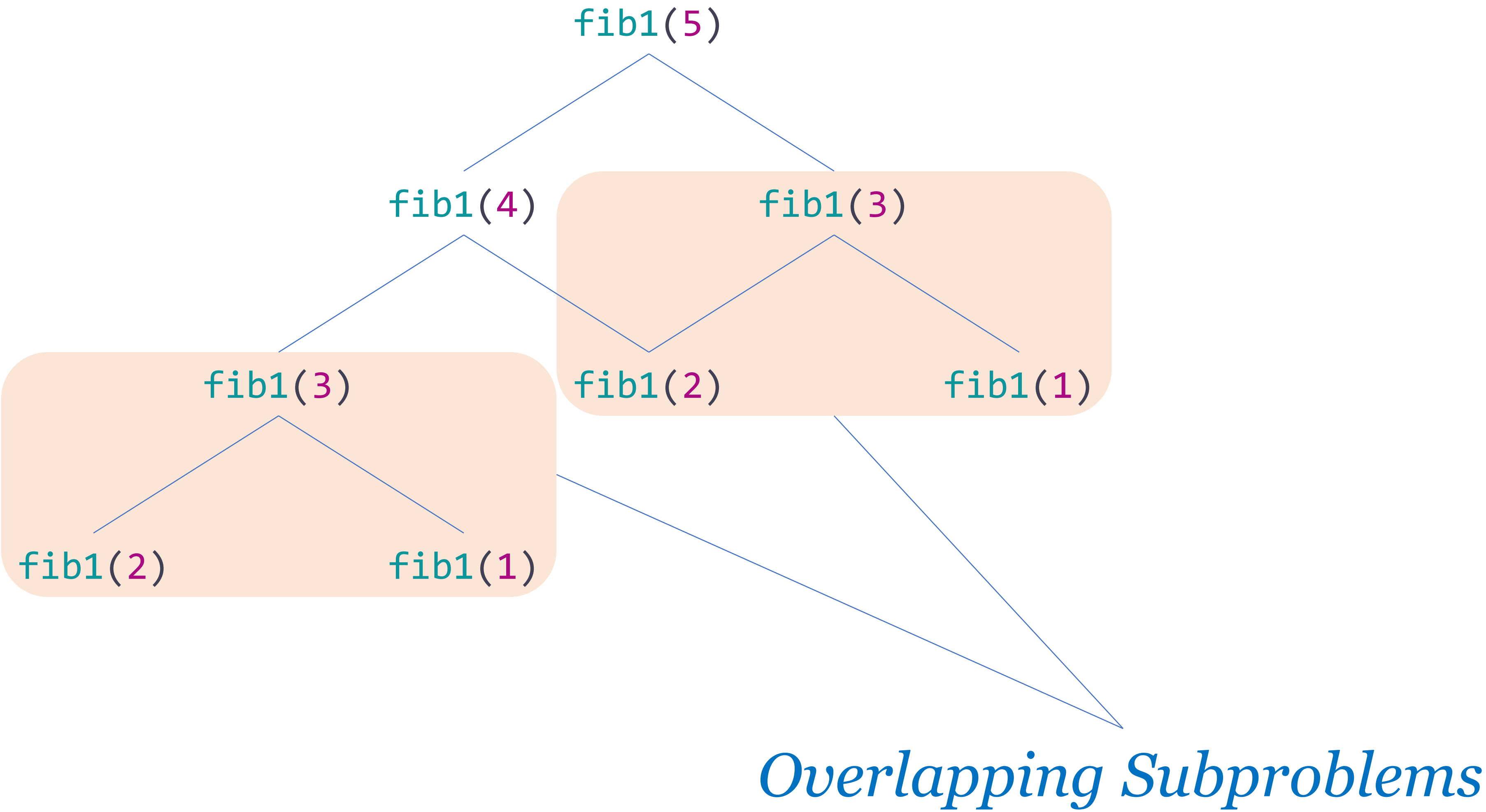
- The steps in the design of Dynamic Programming:
  1. *Establish* a ***recursive property*** that gives
     - the solution to an instance of the problem. (*top-down* approach)
  2. *Solve* an instance of the problem *in a* ***bottom-up fashion***
     - by solving smaller instances first (using *memoization*).

Dynamic Programming



https://trekhleb.dev/blog/2018/dynamic-programming-vs-divide-and-conquer/

- Fibonacci Sequence Revisited:



Dynamic Programming

- Using *Memoization*

```
typedef unsigned long long longint;
vector<longint> F;

longint fib2(int n) {
    if (n <= 1)
        F[n] = n;
    else if (F[n] == -1)
        F[n] = fib2(n - 1) + fib2(n - 2);
    return F[n];
}
```

```
F.resize(n + 1, -1);
cout << fib2(n) << endl;
```

- Using *Tabulation*

```
typedef unsigned long long longint;
vector<longint> F;

longint fib3(int n) {
    F.resize(n + 1);
    if (n <= 1)
        F[n] = n;
    else {
        F[0] = 0; F[1] = 1;
        for (int i = 2; i <= n; i++)
            F[n] = F[n - 1] + F[n - 2];
    }
    return F[n];
}
```

3.1 The Binomial Coefficient

- The Binomial Coefficient Problem:
  - The definition of *binomial coefficient* is given by:
    - $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, for $0 \leq k \leq n$.
    - It is hard to compute directly because $n!$ is very large even for small n.
  - Using the *recursive property* of the binomial coefficient:
    - $\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$
    - We can eliminate the need to compute $n!$ or $k!$.

3.1 The Binomial Coefficient

ALGORITHM 3.1: Binomial Coefficient Using Divide-and-Conquer

```
typedef unsigned long long LongInteger;

LongInteger bin(int n, int k)
{
    if (k == 0 || n == k)
        return 1;
    else
        return bin(n - 1, k) + bin(n - 1, k - 1);
}
```

3.1 The Binomial Coefficient

- The ***Inefficiency*** of Algorithm 3.1:
  - Algorithm 3.1 computes $2\binom{n}{k} - 1$ *terms* to determine $\binom{n}{k}$. (Exercise 3.1.2)
  - The problem is that
    - the *same instances* are *solved* in *each recursive call*.
    - (*overlapping subproblems*)
  - Recall that the *divide-and-conquer* approach is always *inefficient*
    - when an instance is divided into *two smaller instances*
      - that are *almost as large as* the original instance.

3.1 The Binomial Coefficient

Top-Down →

```
bin(4, 2)
        bin(3, 1)
                bin(2, 0)
                bin(2, 1)
                        bin(1, 0)
                        bin(1, 1)
        bin(3, 2)
                bin(2, 1)
                        bin(1, 0)
                        bin(1, 1)
                bin(2, 2)
```

← **Bottom-Up**

3.1 The Binomial Coefficient

- Design a more efficient algorithm using Dyn. Prog.
  - *Establish* a *recursive property*:
    - Let B be an array that $B[i][j]$ contains $\binom{n}{k}$.
    - $B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \ or \ j = i \end{cases}$
  - *Solve* an instance of the problem in a *bottom-up fashion.*
    - by *computing the rows* in B in sequence
      - *starting* with the *first* row.

3.1 The Binomial Coefficient

j, *k*

	B[i][j]	0	1	2	3	4
	0	1				
	1	1	1			
i, *n*	2	1	2	1		
	3	1	3	3	1	
	4	1	4	6	4	1

$$\begin{array}{ccc} B[i-1][j-1] & & B[i-1][j] \\ & \searrow & \downarrow \\ & & B[i][j] \end{array}$$

- You may recognize the array in this figure as *Pascal's triangle*.

3.1 The Binomial Coefficient

ALGORITHM 3.2: Binomial Coefficient Using Divide-and-Conquer

```
typedef unsigned long long LongInteger;

LongInteger bin2(int n, int k)
{
    vector<vector<LongInteger>> B(n + 1, vector<LongInteger>(n + 1));
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= min(i, k); j++)
            if (j == 0 || j == i)
                B[i][j] = 1;
            else
                B[i][j] = B[i - 1][j] + B[i - 1][j - 1];
    return B[n][k];
}
```

3.1 The Binomial Coefficient

- **Time Complexity of Algorithm 3.2:**
  - The parameters n and k are *not the size of input* to this algorithm.
    - Rather, they are the input, and the input size is
      - the *number of symbols* it takes to *encode* them.
  - For given n and k,
    - the number of passes for each value of i are:

i	0	1	2	3	…	k	$k+1$	…	n
Number of passes	1	2	3	4	…	$k+1$	$k+1$	…	$k+1$

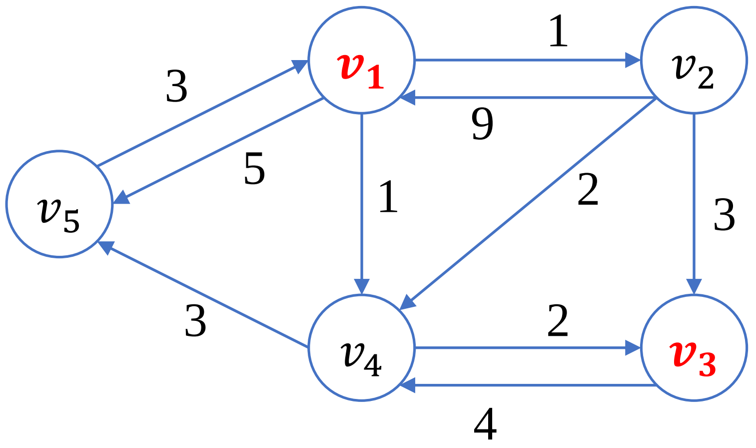
- Time Complexity of Algorithm 3.2:
  - The total number of passes is:
    - $1 + 2 + 3 + 4 + \cdots + k + (k+1) + (k+1) \ldots + (k+1)$
    - $= \frac{k(k+1)}{2} + (n-k+1)(k+1)$
    - $= \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)$.
  - We can argue that we have developed a *much more efficient* algorithm,
    - by using *dynamic programming* instead of *divide-and-conquer*.

3.1 The Binomial Coefficient

- *Improving* the Performance of Algorithm 3.2
  - Use only a *one-dimensional array* indexed from 0 to k,
    - instead of creating the *entire two-dimensional* array B.
    - Once a row is computed,
      - we no longer need the values in the row that proceeds it.
    - By the way, how can we do it with *only one* array?
  - Can we *reduce* the size of one-dimensional array *in half*?
    - Take advantage of the fact that
      - $\binom{n}{k} = \binom{n}{n-k}$.

3.2 Floyd's Algorithm for Shortest Paths

- Finding the Shortest Paths in a Graph:
  - from *each vertex* to *all other* vertices (*All Pairs* Shortest Paths)



- Shortest path from v_1 to v_3?
  - $length[v_1, v_2, v_3] = 1 + 3 = 4$
  - $length[v_1, v_4, v_3] = 1 + 2 = 3$
  - $length[v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$

3.2 Floyd's Algorithm for Shortest Paths

- The Shortest Paths as an ***Optimization Problem***:
  - There can be more than one *candidate solution*
    - to an instance of an optimization problem.
  - Each *candidate solution* has a *value* associated with it,
    - and a solution to the instance is any candidate solution
      - that has an *optimal value* (either *minimum* or *maximum*).
  - There can be more than one shortest paths from one vertex to another.
    - Our problem is to find *any one* of the shortest paths.

3.2 Floyd's Algorithm for Shortest Paths

- The **Brute-Force** Approach:
  - Determine, for each vertex,
    - the *lengths of all the paths* from that vertex to each other vertex,
      - and compute the *minimum* of these lengths.
  - The *total number of paths* from one vertex to another vertex is
    - $(n-2)(n-3)\cdots 1 = (n-2)!$ (*factorial time*)
    - This algorithm is *worse* than *exponential time*.
  - This is often the case with many optimization problems.
    - Our goal is to find a more efficient algorithm for this problem.

- ***Floyd's Algorithm***:
  - A *cubic-time* algorithm for the problem of (*All Pairs*) *Shortest Paths*.
    - not *exponential*, but *polynomial*.
  - Using *dynamic programming*,
    - First, we develop an algorithm
      - that determines *only the lengths* of the shortest paths.
    - After, we modify it to *produce shortest paths* as well.

3.2 Floyd's Algorithm for Shortest Paths

- Data Structure for the Floyd's Algorithm
  - We represent a *weighted* (*directed*) graph containing n vertices
    - by an array (*adjacency matrix*) W where
      - $$W[i][j] = \begin{cases} weight\ on\ edge & \text{if there is an edge from } v_i \text{ to } v_j \\ \infty & \text{if there is no edge from } v_i \text{ to } \mathrm{v_j} \\ 0 & if\ i = j \end{cases}$$
  - We also define an array D as a matrix that
    - contains the lengths of the shortest paths in the graph.

3.2 Floyd's Algorithm for Shortest Paths

W	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

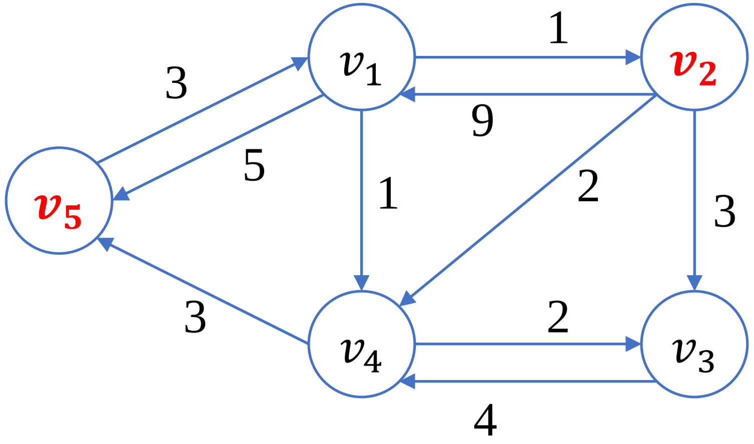
D	1	2	3	4	5
1	0	1	**3**	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

3.2 Floyd's Algorithm for Shortest Paths

- **Solving the Problem:**
  - If we can develop a way
    - to *calculate the values* in *D from those* in *W*,
      - we will have an algorithm for the Shortest Paths problem.
  - Let us create a *sequence* of $n + 1$ arrays $D^{(k)}$,
    - where $0 \le k \le n$ and where

$D^{(k)}[i][j] =$ length of a *shortest path* from v_i to v_j using *only vertices* in the set $\{v_1, v_2, \cdots, v_k\}$ as *intermediate vertices*.

3.2 Floyd's Algorithm for Shortest Paths



- $D^{(0)}[2][5] = length[v_2, v_5] = \infty$
- $D^{(1)}[2][5] = minimum(length[v_2, v_5], length[v_2, v_1, v_5])$
  $= minimum(\infty, 14) = 14.$
- $D^{(2)}[2][5] = D^{(1)}[2][5] = 14.$
- $D^{(3)}[2][5] = D^{(2)}[2][5] = 14.$
- $D^{(4)}[2][5] = minimum(length[v_2, v_1, v_5], length[v_2, v_4, v_5],$
  $length[v_2, v_1, v_4, v_5], length[v_2, v_3, v_4, v_5]$
  $= minimum(14,5, 13,10) = 5.$
- $D^{(5)}[2][5] = D^{(4)}[2][5] = 5.$

3.2 Floyd's Algorithm for Shortest Paths

- Solving the Problem:
  - Note that $D^{(n)}[i][j]$ is the length of a shortest path from v_i to v_j,
    - because it is the length of a shortest path from v_i to v_j
      - that is allowed to pass through *any of the other vertices*.
  - Therefore, we have established that
    - $D^{(0)} = W$ and $D^{(n)} = D$.
  - To determine *D* from *W*,
    - we need only find a way to obtain $D^{(n)}$ from $D^{(0)}$.

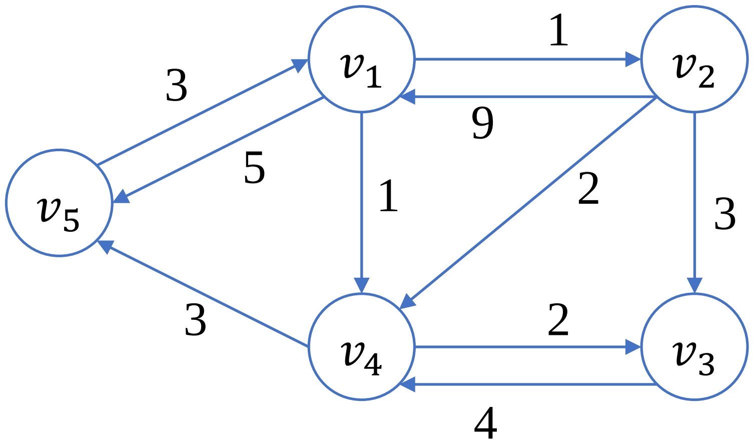
- The steps for Dynamic Programming:
  - Establish a recursive property
    - with which we compute $D^{(k)}$ from $D^{(k-1)}$.
  - Solve an instance of the problem in a *bottom-up fashion*
    - by repeating the process for $k = 1$ to n.
    - This process creates the sequence:

$$\underset{\substack{\uparrow \\ W}}{D^0},\ D^1,\ D^2,\ \cdots\ ,\ \underset{\substack{\uparrow \\ D}}{D^n}$$

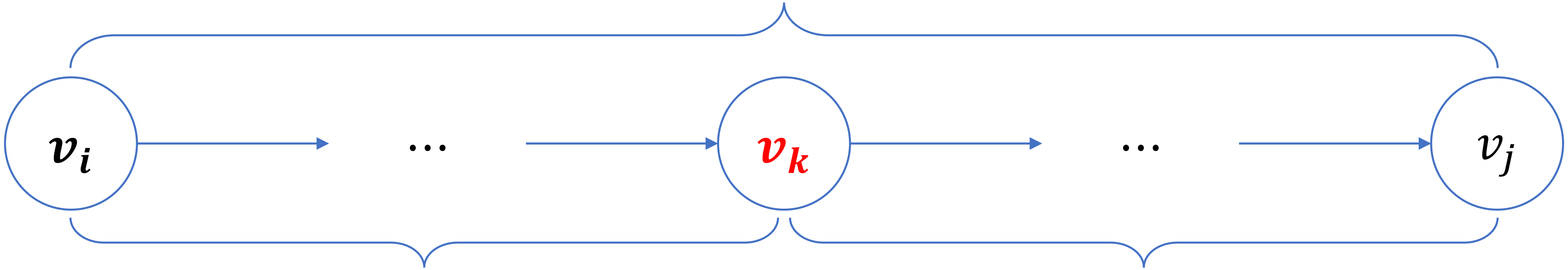
- Two cases of Establishing Recursive Property:
  1. At least one shortest path from v_i to v_j does not use v_k,
     - using only vertices in $\{v_1, v_2, \cdots, v_k\}$ as intermediate vertices.
     - Then, $D^{(k)}[i][j] = D^{(k-1)}[i][j]$.
  2. All shortest paths from v_i to v_j do use v_k,
     - using only vertices in $\{v_1, v_2, \cdots, v_k\}$ as intermediate vertices.
     - Then, $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$.

3.2 Floyd's Algorithm for Shortest Paths

- Examples of Two Cases:
  - **case 1**: $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$,
    - because when we include vertex v_5,
      - the shortest path from v_1 to v_3 is still $[v_1, v_4, v_3]$.
  - **case 2**: $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$,
    - because v_k cannot be an intermediate vertex on the subpath from v_i to v_k,
      - that subpath uses only vertices in $\{v_1, v_2, \cdots, v_{k-1}\}$ as intermediates.



A shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \cdots, v_k\}$



A shortest path from v_i to v_k using only vertices in $\{v_1, v_2, \cdots, v_k\}$

A shortest path from v_k to v_j using only vertices in $\{v_1, v_2, \cdots, v_k\}$

3.2 Floyd's Algorithm for Shortest Paths

- Considering Two Cases:
  - Let $D^{(k)}[i][j]$ be the *minimum* of the values from either Case 1 or Case 2.
    - $D^{(k)}[i][j] = minimum(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$.
  - Now we have established a recursive property (Step 1).
    - To accomplish Step 2,
      - create the sequence of arrays: $D^0, D^1, D^2, \cdots, D^n$.

3.2 Floyd's Algorithm for Shortest Paths

$D^0 = W$

$\boldsymbol{D^0}$	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

- $D^{(1)}[2][4] = minimum\left(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4]\right)$
  $= minimum(2,\ 9+1) = 2.$
- $D^{(1)}[5][2] = minimum\left(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2]\right)$
  $= minimum(\infty,\ 3+1) = 4.$
- $D^{(1)}[5][4] = minimum\left(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4]\right)$
  $= minimum(\infty,\ 3+1) = 4.$

- $D^{(2)}[5][4] = minimum\left(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4]\right)$
  $= minimum(4,\ 4+2) = 4.$

ALGORITHM 3.3: Floyd's Algorithm for Shortest Paths

```
#define INF 0xffff

typedef vector<vector<int>> matrix_t;

void floyd(int n, matrix_t& W, matrix_t& D)
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            D[i][j] = W[i][j];
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
}
```

3.2 Floyd's Algorithm for Shortest Paths

- Why can we use only one array D?
  - because the *values* in the kth row and the kth column
    - are *not changed* during the kth iteration of the loop.
  - In the kth iteration, the algorithm assigns
    - $D[i][k] = minimum(D[i][k], D[i][k] + D[k][k])$, and
    - $D[k][j] = minimum(D[k][j], D[k][k] + D[k][j])$
  - During the kth iteration, $D[i][j]$ is computed
    - from only *its own value* and *values* in the kth row and the kth column.
    - These values have *maintained* their values from the $(k-1)$st iteration.

- Time Complexity of Floyd's algorithm: $T(n) = n^3 \in \Theta(n^3)$.

3.2 Floyd's Algorithm for Shortest Paths

- Producing the *Shortest Paths*:
  - Define an array P, where

$$P[i][j] = \begin{cases} \text{highest index of } \textit{an itermediate vertex} \text{ on the shortest path} \\ \text{from } v_i \text{ to } v_j\text{, if at least one intermediate vertex exists.} \\ 0\text{, if no intermediate vertex exists.} \end{cases}$$

P	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

ALGORITHM 3.4: Floyd's Algorithm for Shortest Paths 2

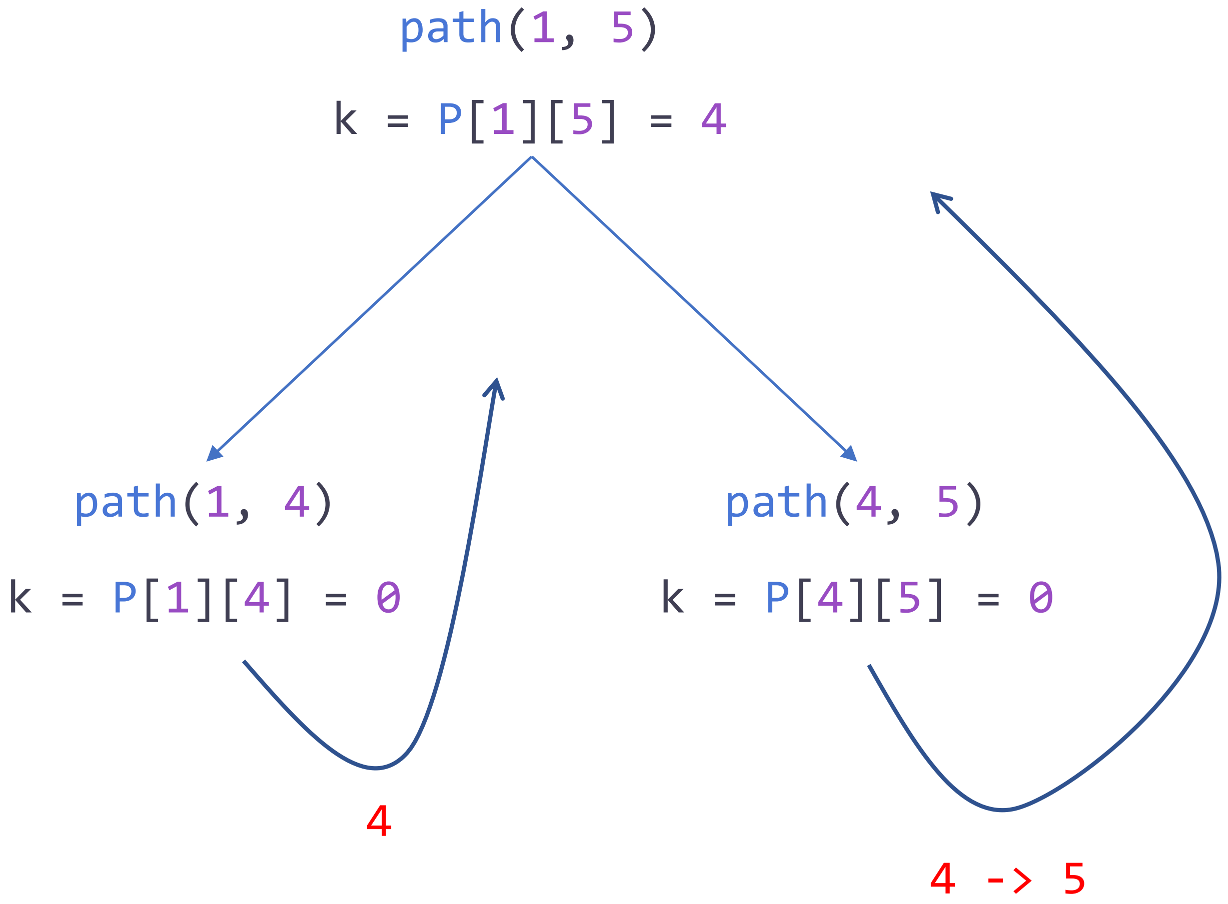
```
void floyd2(int n, matrix_t& W, matrix_t& D, matrix_t& P)
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            D[i][j] = W[i][j];
            P[i][j] = 0;
        }
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (D[i][j] > D[i][k] + D[k][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
}
```

ALGORITHM 3.5: Print Shortest Path

```
void path(matrix_t& P, int u, int v, vector<int>& p)
{
    int k = P[u][v];
    if (k != 0) {
        path(P, u, k, p);
        p.push_back(k);
        path(P, k, v, p);
    }
}
```

3.2 Floyd's Algorithm for Shortest Paths

P	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0



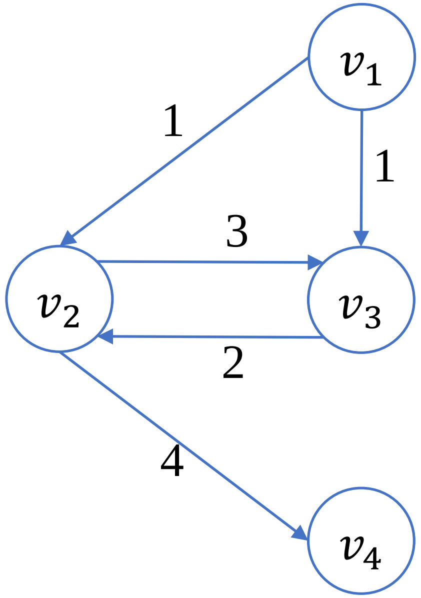
3.3 Dynamic Programming and Optimization Problems

- The design of a D.P. algorithm for an *optimization problem*:
  1. *Establish* a recursive property
     - that gives the optimal solution to an instance of the problem.
  2. *Compute* the *value* of an *optimal solution* in a bottom-up fashion.
  3. *Construct* an *optimal solution* in a bottom-up fashion.

- Note that *Steps 2* and *3* are ordinarily accomplished
  - at *about the same point* in the algorithm

- ***The Principle of Optimality***:
  - An *optimization problem* can be solved using *dynamic programming*,
    - if and only if the *principle of optimality applies* in the problem.
  - The principle of optimality is said to *apply* in a problem
    - if an *optimal solution* to an instance of a problem
      - *always contains optimal solutions to all subinstances*.

- Example: the *Longest Paths* problem
  - Finding the *longest **simple** paths* from each vertex to all other vertices.
  - We restrict the path to be *simple*,
    - because *a cycle* can always create an arbitrarily long path
      - by repeatedly passing through the cycle.
  - Then, we can show that the principle of optimality *does not apply*.



- The optimal (longest) simple path from v_1 to v_4 is $[v_1, v_3, v_2, v_4]$.
- However, the subpath $[v_1, v_3]$ is not an optimal path from v_1 to v_4
  - $length\ [v_1, v_3] = 1$ and $length[v_1, v_2, v_3] = 4$.

삼각형 위의 최대 경로

문제 아래 형태와 같이 삼각형 모양으로 배치된 자연수들이 있습니다. 맨 위의 숫자에서 시작해, 한 번에 한 칸씩 아래로 내려가 맨 아래 줄로 내려가는 경로를 만들려고 합니다. 경로는 아래 줄로 내려갈 때마다 바로 아래 숫자, 혹은 오른쪽 아래 숫자로 내려갈 수 있습니다. 이 때 모든 경로 중 포함된 숫자의 최대 합을 찾는 프로그램을 작성하세요.

```
6
1   2
3   7   4
9   4   1   7
2   7   5   9   4
```

입력 입력의 첫 줄에는 테스트 케이스의 수 C(C <= 50)가 주어집니다. 각 테스트 케이스의 첫 줄에는 삼각형의 크기 n(2 <= n <= 100)이 주어지고, 그 후 n줄에는 각 1개~n개의 숫자로 삼각형 각 가로줄에 있는 숫자가 왼쪽부터 주어집니다. 각 숫자는 1 이상 100000 이하의 자연수입니다.

출력 각 테스트 케이스마다 한 줄에 최대 경로의 숫자 합을 출력합니다.

예제 입력

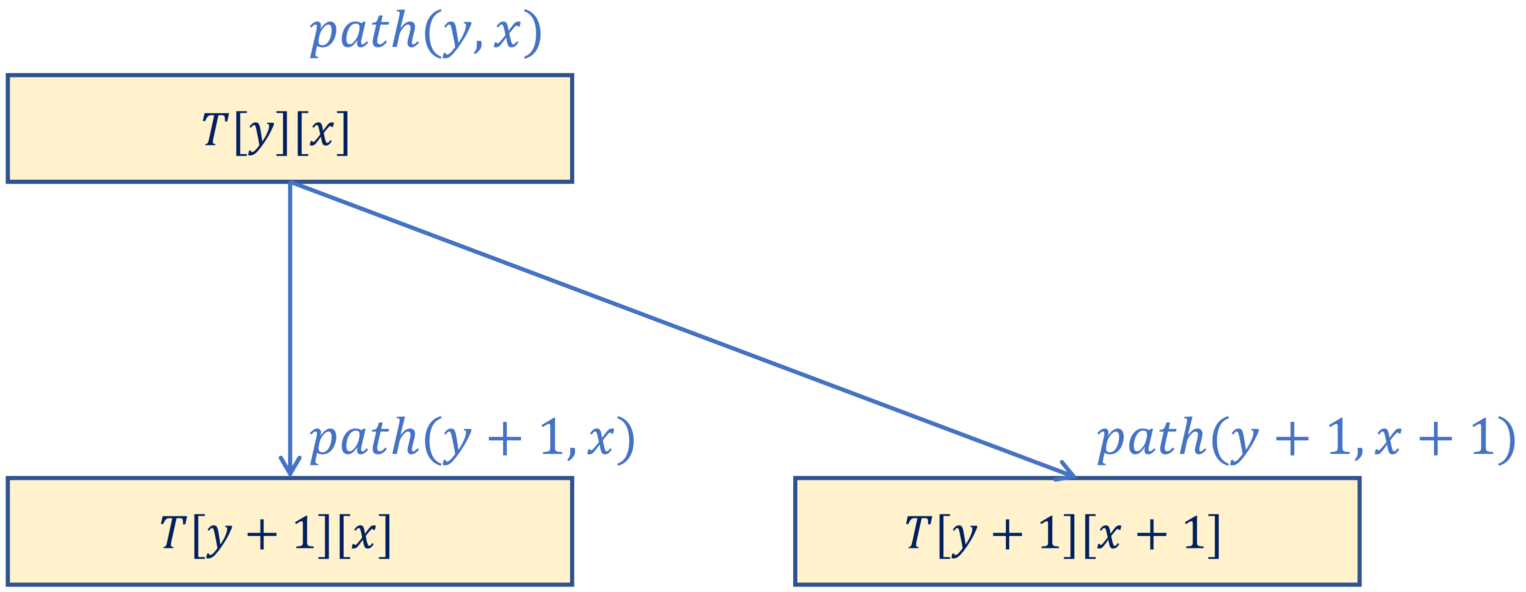
```
2
5
6
1   2
3   7   4
9   4   1   7
2   7   5   9   4
5
1
2  4
8  16  8
32  64  32  64
128  256  128  256  128
```

예제 출력

```
28
341
```

Lec. 04. 동적 계획법 (1)

$path(y, x)$: (y, x)에서 시작해서 맨 아래줄까지 내려가는 부분 경로의 최대 합



$$path(y, x) = T[y][x] + \max\big(path(y+1, x), path(y+1, x+1)\big)$$

$$path(n-1, x) = T[y][x]$$

메모이제이션으로 해결하기

$T =$

6				
1	2			
3	7	4		
9	4	1	7	
2	7	5	9	4

$cache =$

-1				
-1	-1			
-1	-1	-1		
-1	-1	-1	-1	
-1	-1	-1	-1	-1

메모이제이션으로 해결하기

$T =$

6				
1	2			
3	7	4		
9	4	1	7	
2	7	5	9	4

$cache =$

28				
20	22			
19	18	20		
16	11	10	16	
-1	-1	-1	-1	-1

태뷸레이션으로 해결하기

$T =$

6				
1	2			
3	7	4		
9	4	1	7	
2	7	5	9	4

$dp[0][0]$: 맨 위에서 맨 아래쪽까지 내려갔을 때의 최적값

$dp =$

28				
20	22			
19	18	20		
16	11	10	16	
2	7	5	9	4

bottom-up

삼각형 위의 최대 경로 문제: 최적 부분 구조가 성립하는가?

삼각형 위의 최대 경로 문제는 최대 경로의 합을 찾는 최적화 문제이다.

이 문제의 재귀적 관계에서 현재 문제는 두 개의 부분 문제로 분할한다.

각 부분 문제의 최적값을 구하면, 둘 중에서 더 큰 값을 선택하여 현재 값과 더한다.

부분 문제의 최적해로 전체 문제의 최적해를 알 수 있으므로,

최적 부분 구조가 성립한다.

태뷸레이션 + 공간 복잡도 개선: 슬라이딩 윈도우 적용하기

$n = 10{,}000$인 경우: 공간 복잡도가 $O(n^2)$이므로, 메모리 초과!

슬라이딩 윈도우를 적용하여 공간 복잡도를 한 차수 낮출 수 있다.

재귀식: $dp[i][j] = T[i][j] + \max(dp[i+1][j], dp[i+1][j+1])$

i번째 가로줄 $dp[i]$를 계산하려면, $i+1$번째 가로줄 $dp[i+1]$만 필요하다.

두 개의 윈도우를 생성하여, 번갈아 가며 사용한다.

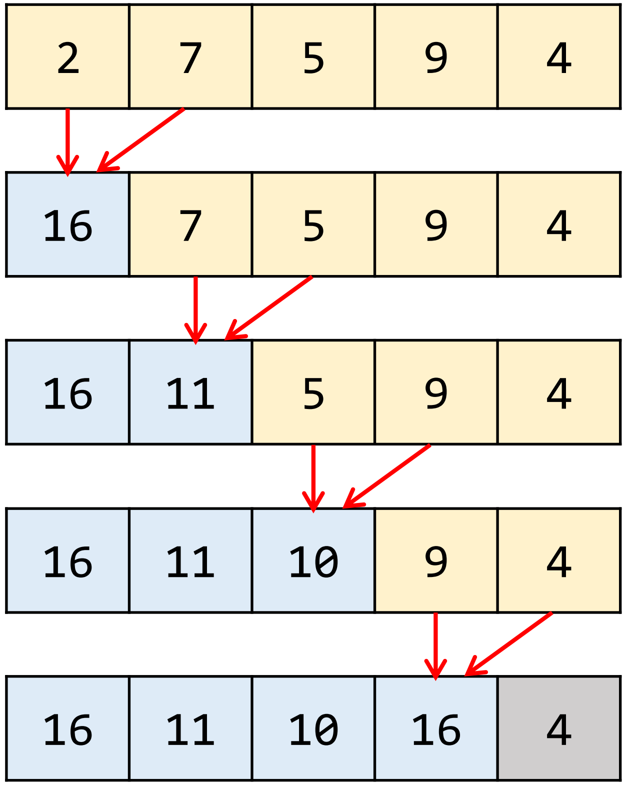
2	7	5	9	4
-1	-1	-1	-1	-1

2	7	5	9	4
16	11	10	16	-1

19	18	20	9	4
16	11	10	16	-1

그런데, 윈도우가 반드시 두 개가 필요한가?

현재 윈도우의 값을 계산할 때, 먼저 계산된 값이 다른 위치의 값에 영향을 주지 않으면?



j번째 원소의 값을 계산할 때

j번째 원소의 값과 $j+1$번째 원소의 값만 고려한다.

j번째 원소의 값이 변경된 이후의 계산에서

j번째 원소의 값은 다시 고려하지 않는다.

따라서, 한 개의 윈도우 만으로도 충분하다.

Any Questions?