

Algorithms

CHAP. 3

동적 프로그래밍

Dynamic Programming

배경사용 : 저장해둔다는 뜻

Overview

복합정복과 동적정. 차이점 비교!

- 동적 프로그래밍(Dynamic Programming)
- 이항계수 계산 알고리즘 *binomial coefficient*
- 이진 검색 트리 개수 구하기
- 최적 이진 검색 트리
- Floyd의 최단 경로 알고리즘 *프loyd와샬*
- 최적의 원칙

Dynamic Programming

동적 프로그래밍

Dynamic Programming 동적 프로그래밍

- 분할정복법은 하향식 해결법으로서, 나누어진 부분들 사이에 서로 상관관계가 없는 문제를 해결하는데 적합.
 - 분할정복법으로 피보나치 수를 계산하면 중복된 계산을 많이 하게 되므로 적합하지 않음.
 - 피보나치 수를 계산하는 알고리즘 경우에는 나누어진 부분들 사이에 서로 연관됨.
 - F_5 를 계산하기 위해서는 F_4 와 F_3 을 계산해야 되며, 이 둘은 모두 F_2 의 계산이 필요.
- 동적프로그래밍(*dynamic programming*)은 분할 정복법과 정반대되는 접근 방법으로서, 상향식(bottom-up) 해결법.
 - 분할정복법과 마찬가지로 문제를 작은 여러 개의 사례로 나누어 이들 사례를 먼저 해결.
 - 하지만 해결한 값을 보관하여, 나중에 이 값이 또 필요하면 다시 계산하지 않고 보관된 결과를 사용.

Binomial's Algorithm

이항계수 계산 알고리즘

- 조합
- 이항정리, 이항계수
- 이항계수 – 분할정복법 접근
- 이항계수 – 동적 프로그래밍 접근

RECAP: 조합(Combination)

서로 다른 n 개의 원소 중에서 순서를 생각하지 않고 r 개를 택할 때, 이것을 n 개의 원소에서 r 개를 택하는 조합(combination)이라 하고, 그 조합의 수를 ${}_nC_r$ 과 같이 나타낸다.

$$\begin{aligned} {}_nC_r &= \frac{{}_nP_r}{r!} = \frac{n \times (n-1) \times (n-2) \times \cdots \times (n-r+1)}{r \times (r-1) \times (r-2) \times \cdots \times 3 \times 2 \times 1} \\ &= \frac{n!}{r!(n-r)!} \quad (\text{단, } 0 < r \leq n) \end{aligned}$$

예를 들면, a, b, c, d로부터 3개의 문자를 순서에 관계없이 선택한다면 다음과 같은 4개의 경우가 있다. 이것을 순열 기호로 나타내면 ${}_4C_3 = 4$ 가 된다.

abc, abd, acd, bcd

한 번에 r 개를 취하는 n 개 대상의 조합의 수는 ${}_nC_r$ 로 표시되며, 같은 기호로서 $C(n, r)$, $C_{n, r}$ 등이 사용되고 있다.

RECAP: 조합(combination)

남자 5명과 여자 4명이 있을 때, 이 중에서 남자 3명, 여자 2명을 뽑는 경우의 수는 몇 가지가 있는지 살펴보자.

풀이 남자 5명 중 3명을 뽑는 경우의 수는 ${}_5C_3$ 가지이고, 여자 4명 중 2명을 뽑는 경우의 수는 ${}_4C_2$ 이므로 ${}_5C_3 \times {}_4C_2 = 10 \times 6 = 60$ 가지이다.

주머니에 크기가 서로 다른 3개의 빨간 공과 4개의 흰 공이 들어 있을 때, 다음을 구해보자.

(1) 이 주머니에서 3개의 공을 뽑는 경우의 수 ${}_7C_3$

(2) 빨간 공 2개와 흰 공 3개를 뽑는 경우의 수 ${}_3C_2 \times {}_4C_3$

RECAP: 이항정리 (Binomial Theorem)

$(a + b)^n$ 을 전개하면 다음과 같은 식이 나오는데, 이것을 이항 정리(binomial theorem)라고 하고, 이때 ${}_nC_r$ 을 이항 계수(binomial coefficient)라고 한다.

$$(a + b)^n = \sum_{k=0}^n {}_nC_k a^{n-k} b^k$$

ex) $(a+b)^2 = {}_2C_0 a^2 + {}_2C_1 ab + {}_2C_2 b^2$

이항 계수 Binomial Coefficient

- 이항계수(binomial coefficient)는 다음과 같이 정의.

$$\begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k!(n-k)!} \text{ for } 0 \leq k \leq n$$

- n 과 k 가 작지 않으면 이항계수를 직접 계산하는 것은 어려움.

- 이항계수는 다음과 같이 재귀적으로 정의.

ex) 1 2 3 4 5 6

$nC2$ 의 경우

① $2C1$

(1, 3), (2, 3) 생각!

② $2C2$

(1, 2)

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + \begin{bmatrix} n-1 \\ k \end{bmatrix} \quad \text{if } 0 < k < n$$
$$1 \quad \text{if } k = 0 \text{ or } k = n$$

- 이 정의로부터 분할정복 방식의 알고리즘을 구현가능. 이 알고리즘은 $n!$ 이나 $k!$ 을 계산하지 않고 이항계수를 계산가능.

분할 정복법을 이용한 이항계수 알고리즘

- 문제: 이항계수를 계산하기
- 입력: 음이 아닌 정수 n 과 k , 여기서 $k \leq n$
- 출력: 이항계수 ${}_nC_k$ $\begin{bmatrix} n \\ k \end{bmatrix}$

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

- 이 알고리즘은 피보나치 수를 계산하는 알고리즘과 마찬가지로 같은 계산을 반복적으로 수행.
- 실제 $2 \cdot {}_nC_k - 1$ (= $2 \binom{n}{k} - 1$) 만큼의 항을 계산해야 ${}_nC_k$ 를 계산가능.

분할정복법을 이용한 이항계수 알고리즘

$2 \cdot {}_nC_k - 1$ 만큼의 항을 계산해야 ${}_nC_k$ 를 계산할 수 있다.

$$= T(n, k)$$

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + \begin{bmatrix} n-1 \\ k \end{bmatrix} \quad \begin{array}{l} \text{if } 0 < k < n \\ 1 \quad \text{if } k=0 \text{ or } k=n \end{array}$$

(증명) 수학적귀납법으로 증명

- basis) $n=1$ 일때 $k=0$ or $k=1 \rightarrow T(1,0) = 2 \cdot {}_1C_0 - 1 = 1$
 $T(1,1) = 2 \cdot {}_1C_1 - 1 = 1$

hypothesis) $n > m > 1$ 인 경우 $T(m, k) = 2 \cdot {}_mC_k - 1$ 이 참이라고 가정

induction) n 에 대해서도 $T(n, k) = 2 \cdot {}_nC_k - 1$ 이 성립함을 보인다 \rightarrow 호환된 수

$$T(n, k) = T(n-1, k-1) + T(n-1, k) + 1 \rightarrow \text{by 재귀 알고리즘} \quad ?? \text{ 이해안됨...}$$

$$= 2 \cdot {}_{n-1}C_{k-1} - 1 + 2 \cdot {}_{n-1}C_k - 1 + 1 \rightarrow \text{by 가정}$$

$$= 2 \cdot ({}_{n-1}C_{k-1} + {}_{n-1}C_k) - 1$$

$$= 2 \cdot \left(\frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \right) - 1 \rightarrow \text{by 조합의 정의}$$

$$= 2 \cdot \left(\frac{k(n-1)!}{k!(n-k)!} + \frac{(n-k)(n-1)!}{k!(n-k)!} \right) - 1$$

$$= 2 \cdot \frac{(k+n-k)(n-1)!}{k!(n-k)!} - 1$$

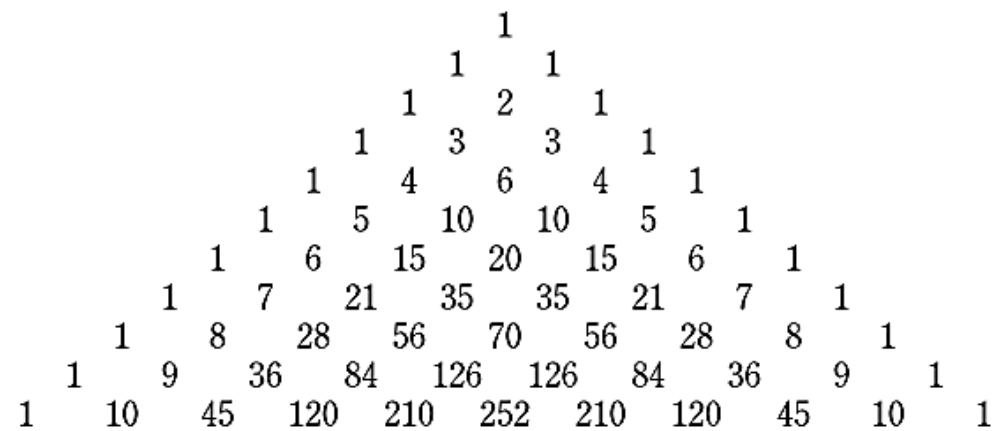
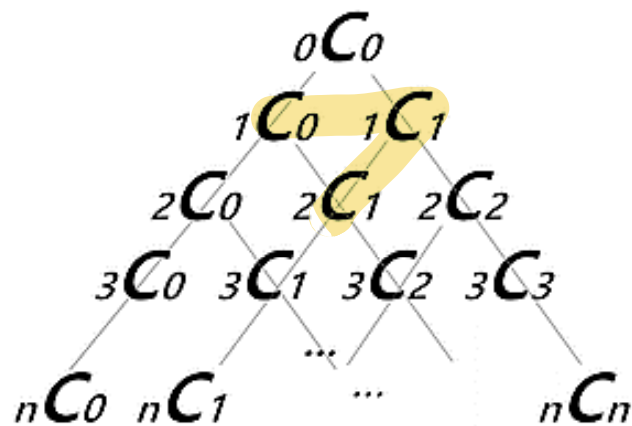
$$= 2 \cdot \frac{n!}{k!(n-k)!} - 1$$

$$= 2 \cdot {}_nC_k - 1$$

Pascal's Triangle

파스칼의 삼각형(Pascal's triangle)에서는 이항 계수가 다음과 같이 만들어진다.

$${}^{n-1}C_{r-1} + {}^{n-1}C_r = {}^nC_r$$



동적 프로그래밍을 이용한 이항계수 알고리즘

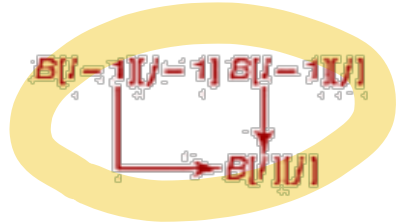
■ 알고리즘 절차

- 단계1. 재귀관계식을 정립한다.

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0, j = i \end{cases}$$

- 단계2. 문제의 사례를 상향식 방식으로 해결한다.

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
i							
n							



$$B[0][0] = 1$$

$$B[1][0] = 1$$

$$B[1][1] = 1$$

$$B[2][0] = 1$$

$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$

$$B[2][2] = 1$$

$$B[3][0] = 1$$

$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$

...

동적 프로그래밍을 이용한 이항계수 알고리즘

- 문제: 이항계수를 계산하기
- 입력: 음이 아닌 정수 n 과 k , 여기서 $k \leq n$
- 출력: 이항계수 ${}_nC_k$
- 알고리즘:

```
int bin2(int n, int k){  
    index i, j;  
    int B[0..n][0..k];  
    for(i = 0; i ≤ n; i++)  
        for(j = 0; j ≤ min(i,k); j++)  
            기보연산  
            (항등) (if(j == 0 || j == i) B[i][j] = 1;  
                    else B[i][j] = B[i-1][j-1] + B[i-1][j];  
    return B[n][k];  
}
```

동적 프로그래밍을 이용한 이항계수 알고리즘

- 주어진 n 과 k 에 대해 for 루프가 반복하는 횟수는 다음과 같다.

i	0	1	2	3	...	k	$k+1$...	n
반복횟수	1	2	3	4	...	$k+1$	$k+1$...	$k+1$

- 전체 반복 횟수는 다음과 같다.

$$1 + 2 + 3 + 4 + \cdots + k + \underbrace{(k+1) + (k+1) + \cdots + (k+1)}_{n-k+1 \text{ 번}}$$

- 위 표현식은 다음과 같다.

$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(k+1)(2n-k+2)}{2} \in \Theta(nk)$$

Binary Search Tree

이진 검색 트리 개수 구하기

- 트리의 기초
- 이진 트리의 특성
- 이진 검색 트리
 - 이진 검색트리에서의 키 검색
 - 이진 검색 트리에서의 최소, 최대 값 찾기
 - 이진 검색 트리의 개수 구하기
 - 최적 이진 검색 트리

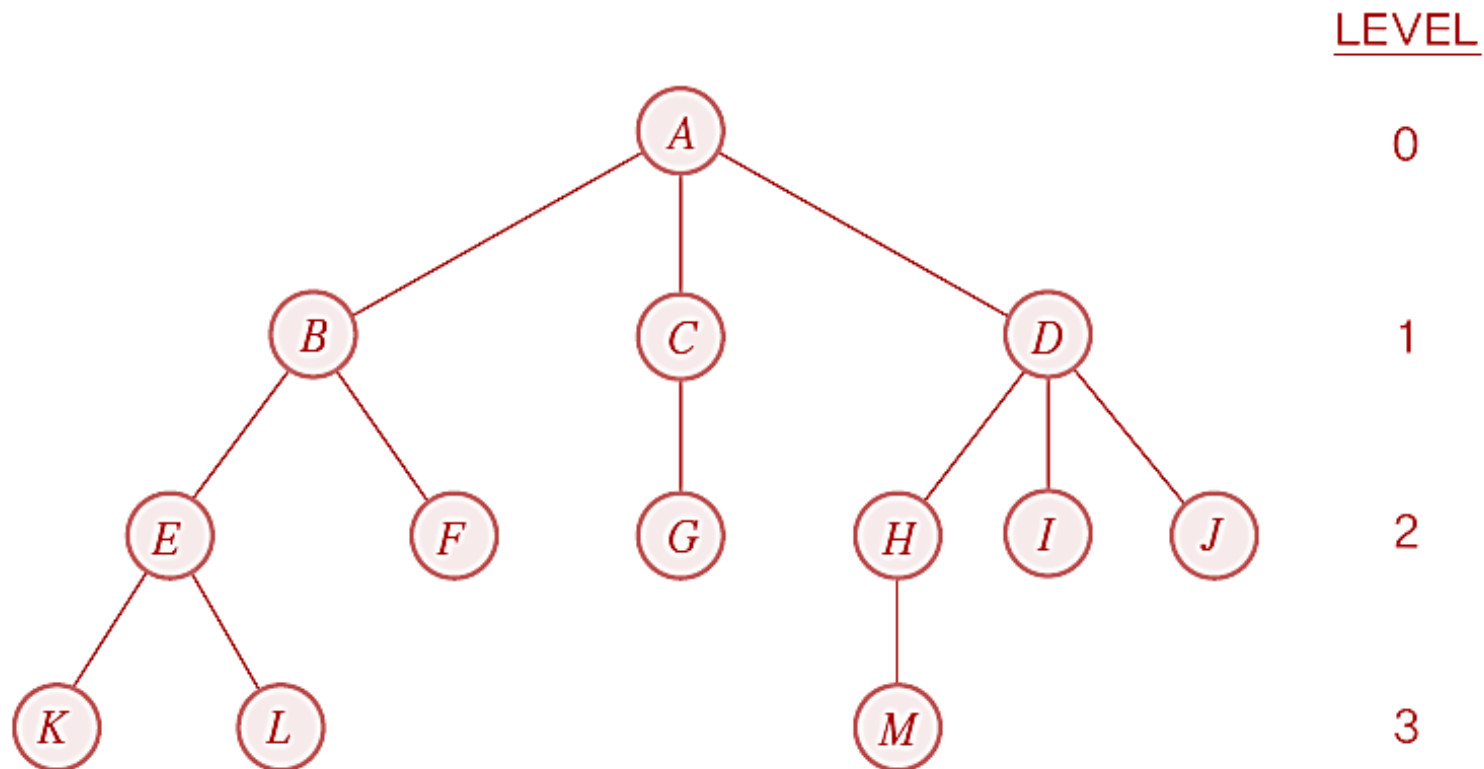
Tree 의 기초 1/3

■ 트리(tree)

- 연결된 비순환 무방향 그래프 (정의, 자유트리)
- 루트(root)라고 하는 유일한 시작 노드를 가지며,
각 노드는 여러 개의 자식 노드를 가질 수 있는 구조로서,
루트에서 각 노드까지의 경로가 유일한 구조. (rooted tree)
- 각 노드를 기준으로 그 노드가 루트가 되는 부분트리 (subtree)를 만들 수 있음 . *→개체개체*

Tree 의 기초 2/3

- **노드의 깊이(depth):** 루트 노드부터 그 노드까지 간선의 수
= **노드의 레벨(level).**
※ 높이 (바닥노드에서 루트까지)



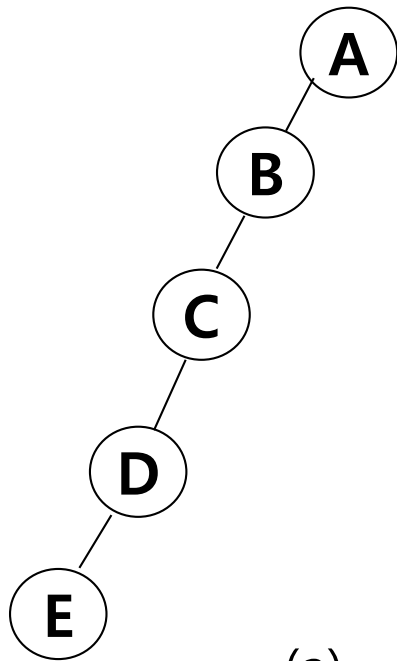
Tree 의 기초 3/3

- **이진 트리(binary tree):** 각 노드가 최대 두 개의 자식 노드만을 가질 수 있는 트리
 - 트리는 재귀 구조(recursive structure).
 - 루트, 왼쪽 서브트리, 오른쪽 서브트리 세부분으로 구분
- **균형 이진 트리(balanced binary tree):** 트리의 모든 노드의 왼쪽 부분트리와 오른쪽 부분트리의 깊이가 최대 하나 차이가 나는 트리
- **순서트리(ordered tree):** 각 노드의 자식이 순서화된 루트트리.
- **완전 이진 트리:** 노드의 차수가 2거나 리프노드.
 - (깊이가 k 일 때 레벨 1부터 $k-1$ 까지는 모두 차 있고 레벨 k 에서는 왼쪽 노드부터 차례로 차 있는 이진 트리임

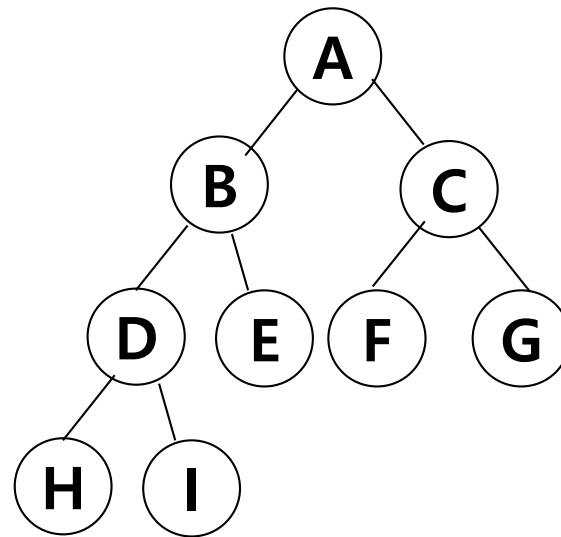
Binary Tree 예

Skewed tree (a) and *complete* binary tree (b)

$$\left. \begin{matrix} 2^0 \\ 2^1 \\ 2^2 \\ \vdots \\ 2^k \end{matrix} \right\} \rightarrow \frac{1(2^{k+1}-1)}{2-1}$$



(a)



(b)

level

0

1

2

3

4

완전이진트리 → 배열 관리
연속 메모리 주소로 접근 가능

Binary Tree 의 특성

- 최대 노드의 개수
 - Maximum number of nodes on level i of a binary tree = 2^i
 - Maximum number of nodes in a binary tree of depth $k = 2^{k+1}-1$
 -
- 이진 트리의 표현 (앞 페이지 그림)

tree

A	B	-	C	-	-	-	D	-	0 0 0	E
---	---	---	---	---	---	---	---	---	-------	---

인덱스 번호
1부터 시작

tree

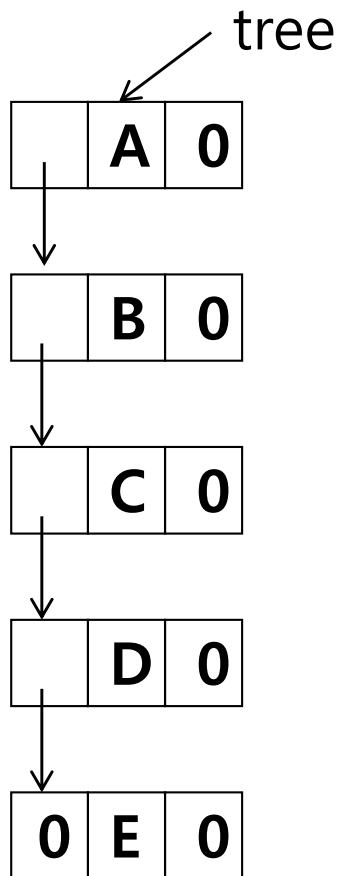
A	B	C	D	E	F	G	H	I
---	---	---	---	---	---	---	---	---

(1) (2) (3) (4) (5) (6) (7) (8) (9) 0 0 0 (16)

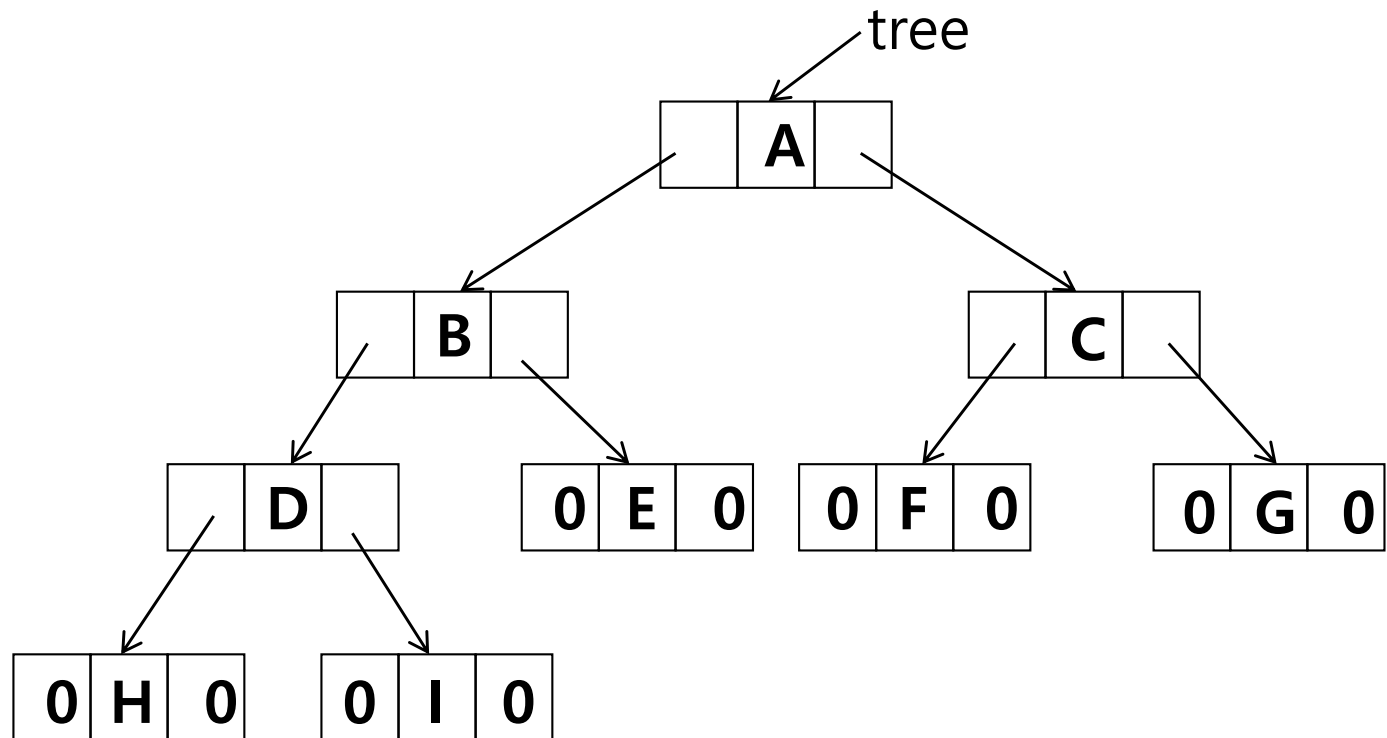
- 부모 자식 노드 관계
 - $\text{parent}(i)$ is at $\text{floor}(i/2)$ if $i \neq 1$
 - $\text{lchild}(i)$ is at $2i$ if $2i \leq n$
 - $\text{rchild}(i)$ is at $2i+1$ if $2i+1 \leq n$

Binary Tree 이진 트리의 표현

- Linked representation of binary tree
 - Three fields (*lchild*, *data*, *rchild*)



(a)



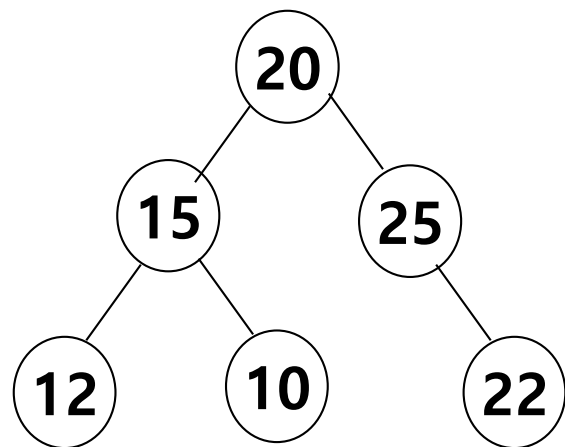
(b)

Binary Search Tree 이진 검색 트리

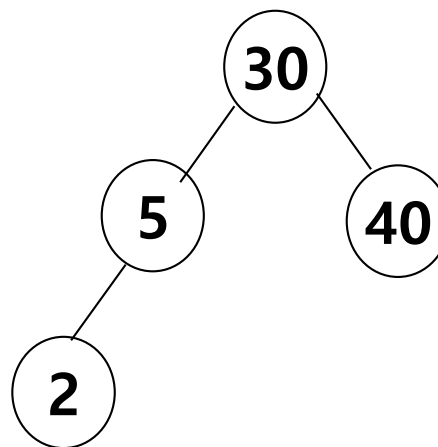
- 이진 검색 트리의 정의: 순서가 정의되어 있는 항(노드)들로 구성된 이진 트리
- 이진 검색 트리의 특성:
 - 각 노드는 키를 가지고 있다.
 - 한 노드의 키는 그 노드의 왼쪽 부분트리에 있는 노드들의 모든 키보다 크다.
 - 한 노드의 키는 그 노드의 오른쪽 부분트리에 있는 노드들의 모든 키보다 작다.

Binary Search Tree ?

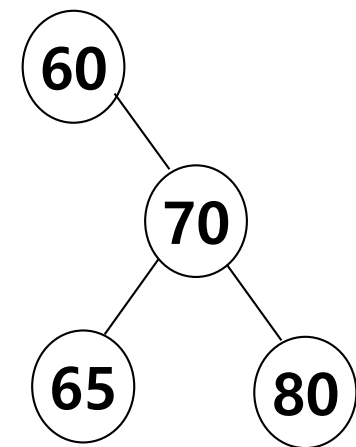
- Example: binary trees
 - Which are the binary search trees ?



(a)



(b)



(c)

Binary Search Tree 에서의 키 검색

- 문제: 주어진 키를 이진 검색 트리에서 찾아라.
- 입력: 이진 검색 트리의 루트의 포인터 x, 찾는 키 k
- 출력: 이진 검색 트리에서 k 가 있는 노드의 포인터.

```
nodetype TreeSearch(x, k) {  
    if ( x==NIL or k==x.key)  
        return x;  
    if( k < x.key )  
        return TreeSearch(x.left, k)  
    else return TreeSearch(x.right, k)  
}
```

```
Struct nodetype{  
    keytype key;  
    nodetype left;  
    nodetype right;  
}
```

```
nodetype TreeSearch2(x, k) {  
    while ( x!=NIL and k!=x.key)  
        if (k < x.key) x=x.left;  
        else          x=x.right;  
    return x }
```

Binary Search Tree 에서의 최대 최소 검색

```
nodetype TreeMin(x){  
    while(x.left != NIL)  
        x=x.left;  
    return x  
}
```

```
nodetype TreeMax(x){  
    while(x.right != NIL)  
        x=x.right;  
    return x  
}
```

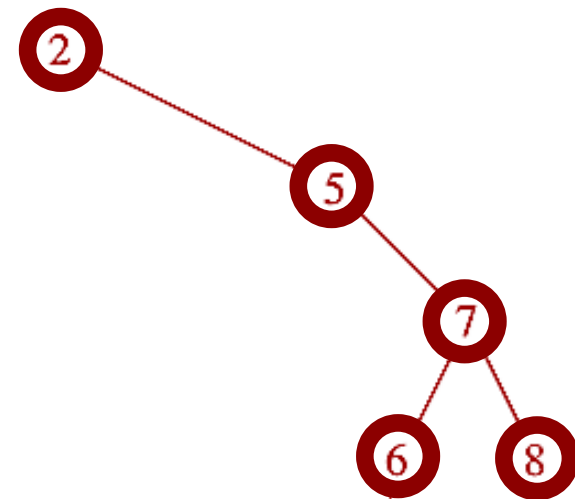
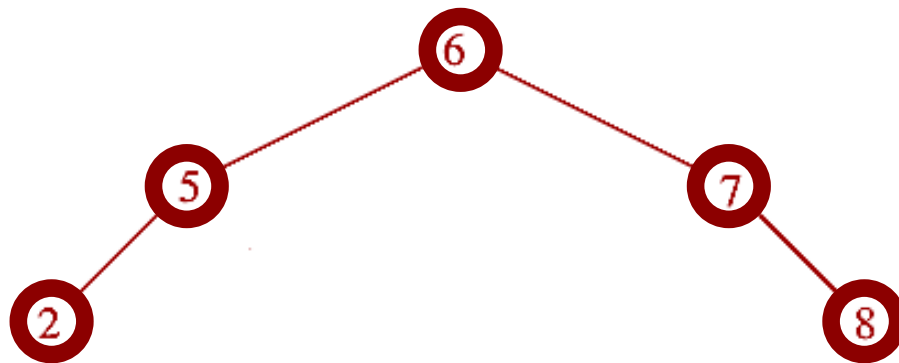
Binary Search Tree 의 개수 구하기

$a_1 < a_2 < a_3 < \dots a_k < \dots a_n$: 이진 검색 트리의 노드(키)

B_n : n 개의 노드로 이루어진 이진 검색 트리의 개수

$$B_0 = 1$$

$$B_n = \sum_{k=0}^{n-1} B_k B_{n-1-k} \quad (n \geq 1)$$



$$B_0 = 1$$

$$B_n = \sum_{k=0}^{n-1} B_k B_{n-1-k} \quad (n \geq 1)$$

$$B_1 = B_0 B_0 = 1$$

$$B_2 = B_0 B_1 + B_1 B_0$$

$$B_3 = B_0 B_2 + B_1 B_1 + B_2 B_0$$

$$B_4 = B_0 B_3 + B_1 B_2 + B_2 B_1 + B_3 B_0$$

$$\vdots$$


Binary Search Tree 의 개수 구하기(분할정복)

- 문제: n개의 노드로 이루어진 이진 검색 트리의 개수를 구하라
- 입력: 노드의 개수
- 출력: n개의 노드로 이루어진 이진 검색 트리의 개수

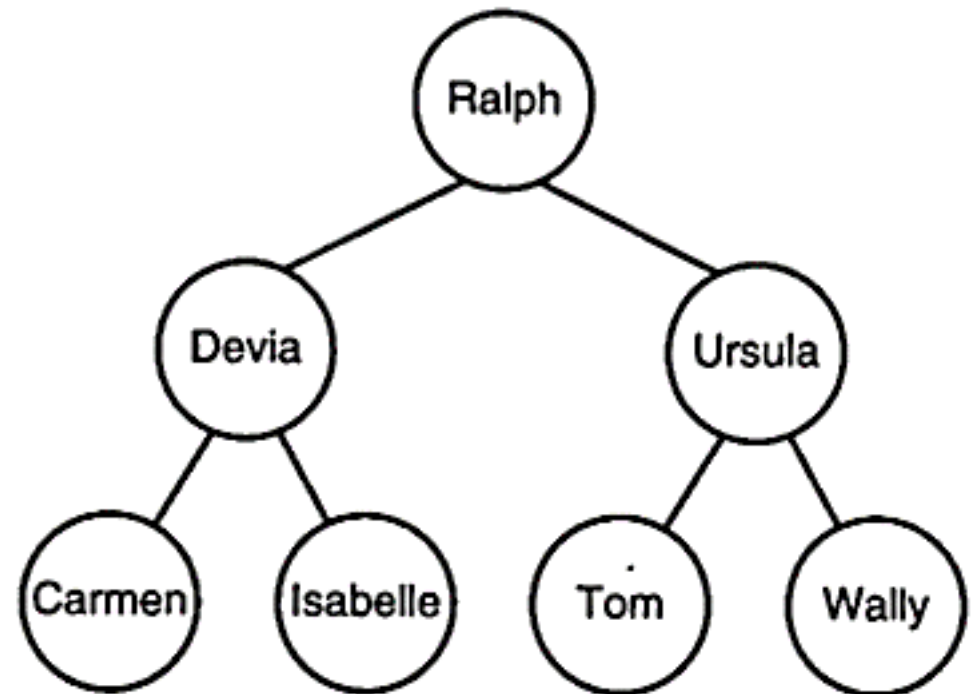
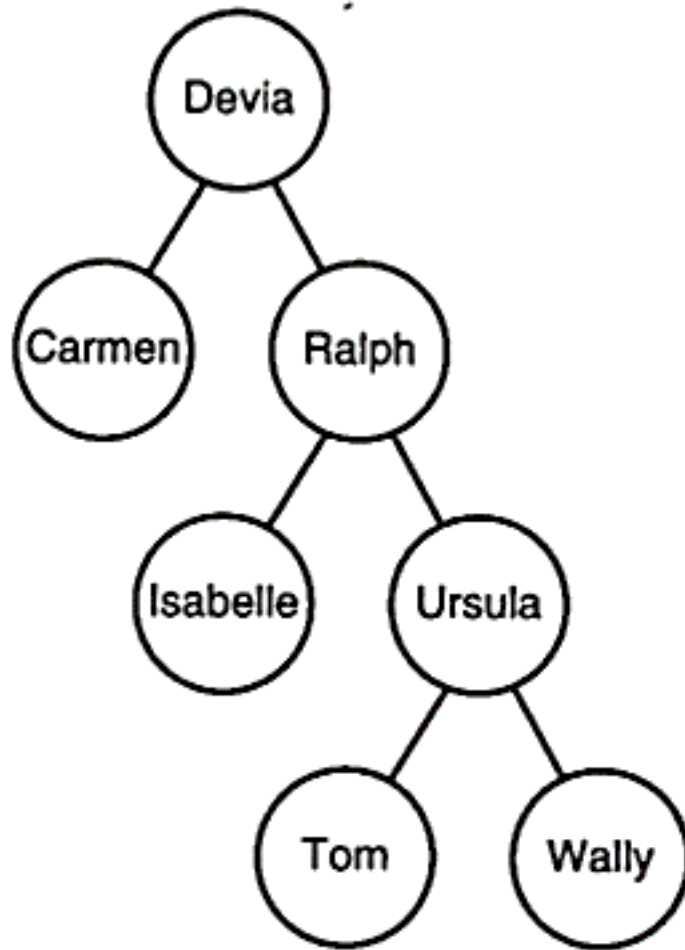
```
Int countTree(int N){  
    if(N==0 || N==1)  
        return 1;  
    else{  
        int sum=0, left=0; right=0;  
        for(k=1;k<=N;k++){  
            left=countTree(k-1);  
            right=countTree(N-k);  
            sum+=left*right;  
        }  
        return sum;  
    }  
}
```

Binary Search Tree 의 개수 구하기(동적프로그래밍)

- 문제: n개의 노드로 이루어진 이진 검색 트리의 개수를 구하라
- 입력: 노드의 개수
- 출력: n개의 노드로 이루어진 이진 검색 트리의 개수

```
int countNumberOfTree(int n){  
    index i, k;  
    int B[n]; // 노드가 n개일 때, 가능한 모든 이진트리의 개수  
    B[0] = 1;  
    B[1] = 1;   
    if(n==0||n==1) return 1;  
    else{  
        for (i = 2; i <= n ; i++)  
            for (k = 0; k <= i-1; k++)  
                B[i] = B[i] + B[k] * B[i - 1 - k] ;  
    }  
}
```

Optimal Binary Search Tree 1/3



Optimal Binary Search Tree 2/3

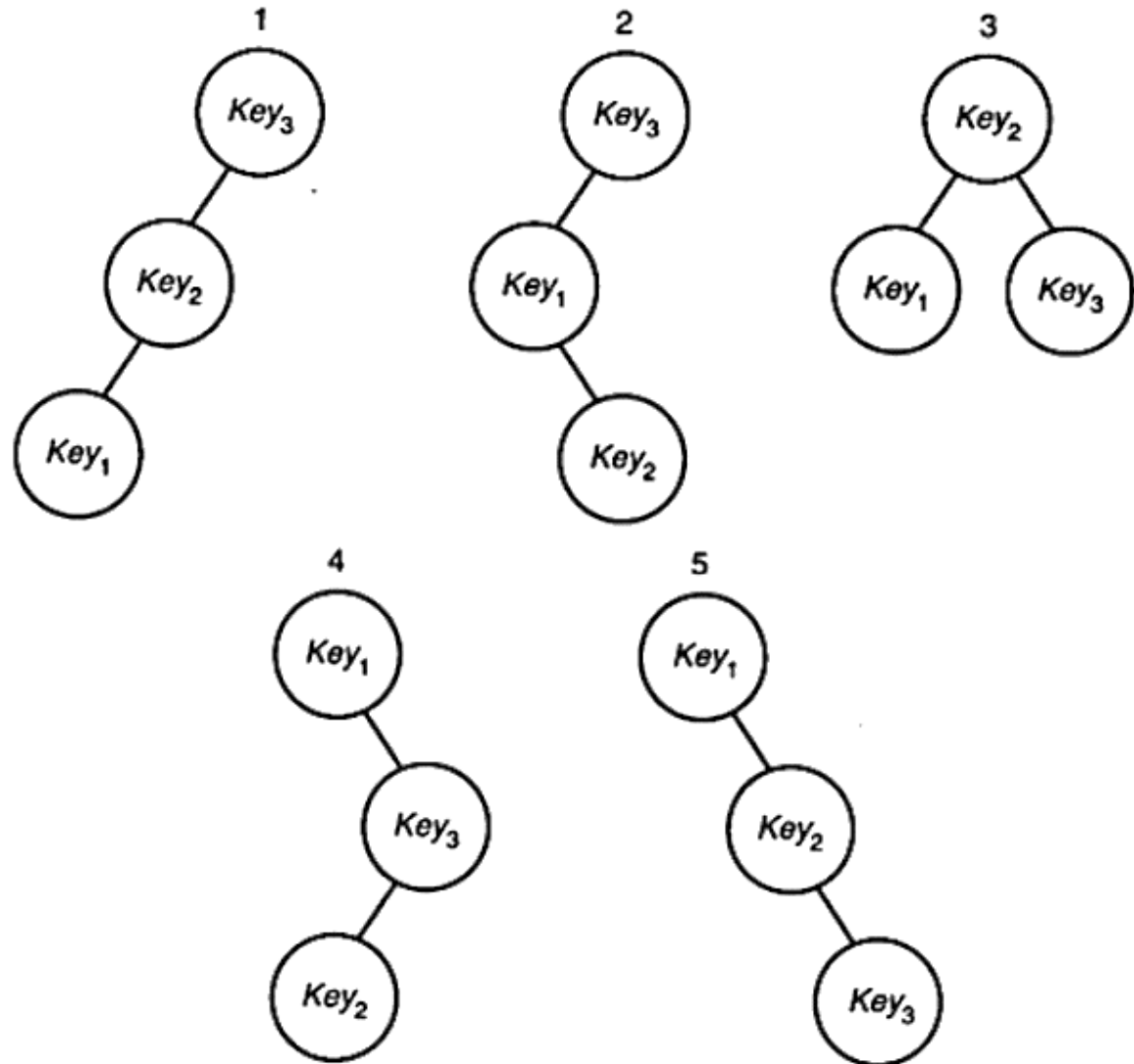
- 이진 검색 트리에서 검색키와 같은 원소를 찾는데 걸리는 평균시간이 최소가 되는 트리를 최적 이진 검색 트리라고 한다.
- 검색시간(Search time) : 어떤 키를 찾기 위해 필요한 시간, 비교 횟수.
 - $\text{Depth}(\text{key}) + 1$
 - 예) Ralph를 찾는 시간: 각각 2, 1
- 트리의 평균 검색 시간
 - (p_i : key_i 를 검색할 확률, c_i : key_i 를 찾는데 필요한 비교횟수)

$$\sum_{i=1}^n c_i p_i$$

Optimal Binary Search Tree 3/3

the average search time

- $P_1 = 0.7,$
 $P_2 = 0.2,$
 $P_3 = 0.1$



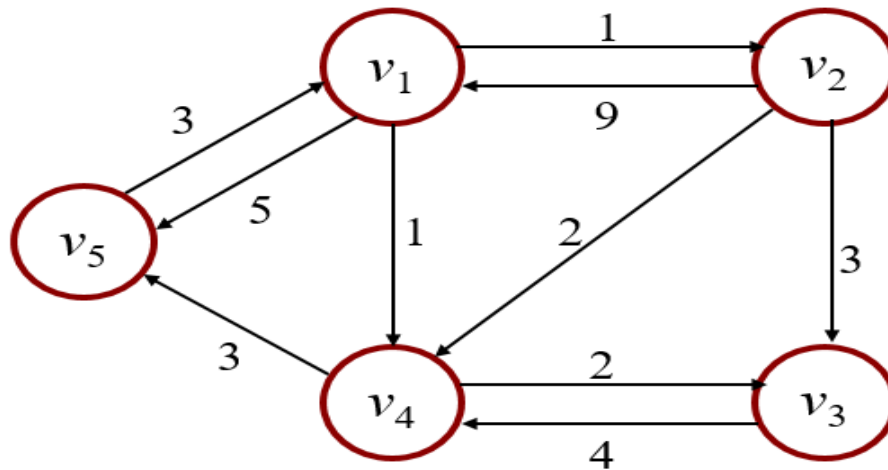
Floyd's Shortest Path Algorithm

FLOYD의 최단 경로 알고리즘

- 그래프의 기초
- 최단경로 찾기
 - Brute -Force Algorithm
 - 플로이드의 최단 경로 알고리즘 I - 최단 거리
 - 플로이드의 최단 경로 알고리즘 II - 경유 노드 구하기

Graph 기초 1/2

- 다음은 가중치 방향 그래프이다.



- 방향그래프(**directed graph, digraph**): 간선에 방향이 있는 그래프
- 가중치 그래프(**weighted graph**): 간선과 연관된 값이 있는 그래프
- 경로(**path**): 한 정점에서 다른 정점까지 도달하기 위해 지나는 일련의 정점(정점 사이에는 간선이 있어야 함)을 말한다.
- 단순경로(**simple path**): 같은 정점을 두 번 지나지 않는 경로

Graph 기초 2/2

- **순환 (cycle):** 한 정점에서 다시 그 정점으로 돌아오는 경로
- **순환 그래프 (cyclic graph):** 순환이 존재하는 그래프
 - 비교. 비순환그래프 (acyclic graph)
- **경로의 길이 (length):** 가중치 그래프에서는 경로를 구성하는 간선들의 가중치의 합을 말하며, 비가중치 그래프에서는 경로를 구성하는 간선의 수를 말한다.
- **인접정점 (adjacent vertex):** 정점 v_i 와 v_j 를 잇는 간선이 있으면 v_i 와 v_j 는 인접 정점이라 한다.

최단 경로 찾기: Brute-force Algorithm

- 무작정 알고리즘(brute-force algorithm): 한 정점에서 다른 정점으로의 모든 경로의 길이를 구한 뒤, 그들 중에서 최소길이를 찾는다.
- 분석:
 - 그래프가 n 개의 정점을 가지고 있고, 모든 정점들 사이에 이음선이 존재한다고 가정.
 - 그러면 한 정점 v_i 에서 어떤 정점 v_j 로 가는 경로들을 다 모아 보면, 그 경로들 중에서 나머지 모든 정점을 한번씩은 꼭 거쳐서 가는 경로들도 포함되어 있는데, 그 경로들의 수만 우선 계산.
 - v_i 에서 출발하여 처음에 도착할 수 있는 정점의 가지 수는 $n-2$ 개 이고, 그 중에 하나를 선택하면, 그 다음에 도착할 수 있는 정점의 가지 수는 $n-3$ 개 이고, 이렇게 계속하여 계산해 보면, 총 경로의 개수는 $(n-2)(n-3)\dots 1 = (n-2)!$
 - 이 경로의 개수만 보아도 지수보다 훨씬 크므로, 절대적으로 비효율적!

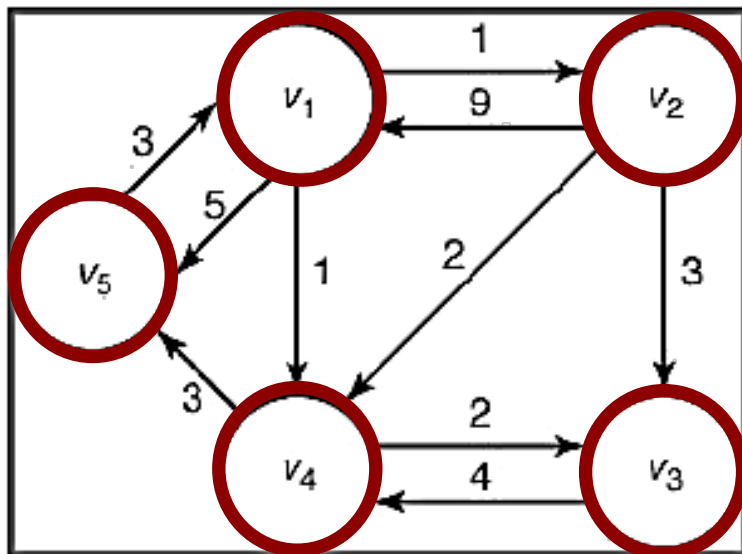
최단 경로 찾기 문제

- 최단 경로 찾기 문제는 최적화 문제(optimization problem) 중 하나.
 - 최적화 문제란 후보 답이 여러 개 존재할 수 있는 문제.
 - 이 문제의 해답은 후보 답 중에 가장 최적의 값을 가지는 답을 하나 찾는 문제.
 - 최적 값이란 보통 최대 또는 최소값.
- 한 정점에서 다른 정점으로 하나 이상의 최단 경로가 존재할 수 있으므로 이 중 하나만 찾으면 된다.
 - 방법1. 모든 경로를 찾은 후에 그 중에서 최단 경로를 찾는다.
 - 완전 연결 그래프이면 한 정점에서 다른 정점으로 모든 정점을 지나는 경로만 총 $(n-2)!$ 개의 존재한다.
 - 이것은 지수시간 보다 더 나쁘다.
 - 방법2. 동적 프로그래밍을 이용하면 차수가 n^3 인 알고리즘을 만들 수 있다.

Floyd의 최단 경로 알고리즘 I 1/6

- 가중치 그래프를 다음과 같이 2차원 배열로 나타낼 수 있다.

$$W[i][j] = \begin{cases} \text{edge weight} & \text{adjacent vertex} \\ \infty & \text{no adjacent vertex} \\ 0 & i = j \end{cases}$$



	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

$W[i][j]$

Floyd의 최단 경로 알고리즘 I 2/6

예) V_2 에서 V_5 까지의 최단경로를 구하는 방법

- $D(k)[i][j]$: 집합 $\{V_1, \dots, V_k\}$ 에 속한 정점만을 중간 정점으로 사용하는 V_i 에서 V_j 로 가는 최단 경로의 길이

- $D(0)[2][5] = \infty$

- $D(1)[2][5]$

$= \min(\text{len}[V_2, V_5], \text{len}[V_2, V_1, V_5]) = 14$

- $D(2)[2][5] = D(1)[2][5] = 14$

- $D(3)[2][5] = D(2)[2][5] = 14$

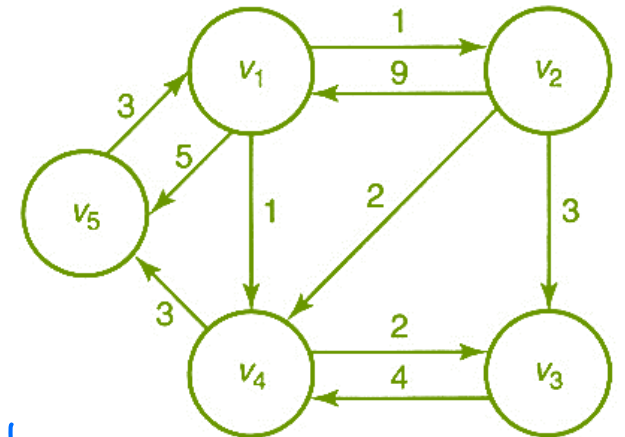
- $D(4)[2][5]$

$= \min(\text{len}[V_2, V_1, V_5], \text{len}[V_2, V_4, V_5],$

$\text{len}[V_2, V_1, V_4, V_5], \text{len}[V_2, V_3, V_4, V_5])$

$= \min(14, 5, 13, 10) = 5$

- $D(4)[2][5] = D(5)[2][5] = 5$



가장 먼저
자신과 같은게 없으면 ∞

$D[i][j]$	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

	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

$D[i][j]$

Floyd의 최단 경로 알고리즘 I 3/6

- 앞 슬라이드 예를 통해 다음이 성립.

기본정보
 $D(0) = \underline{W}, D(n) = D(\text{최단 경로})$

- 동적 프로그래밍을 이용한 최단 경로 알고리즘의 설계 절차

- 단계1. $D(k-1)$ 로부터 $D(k)$ 를 계산할 수 있도록 재귀관계식을 설립
- 단계2. 상향식방법으로 $D(0)$ 부터 $D(n)$ 까지 차례로 구한다.

- 재귀 관계식의 설정

- 다음 두 가지 경우를 고려해야 함

- 경우1. V_i 에서 V_j 로의 하나이상의 최단경로가 V_k 를 사용하지 않으면 다음이 성립한다.

$$D(k)[i][j] = D(k-1)[i][j]$$

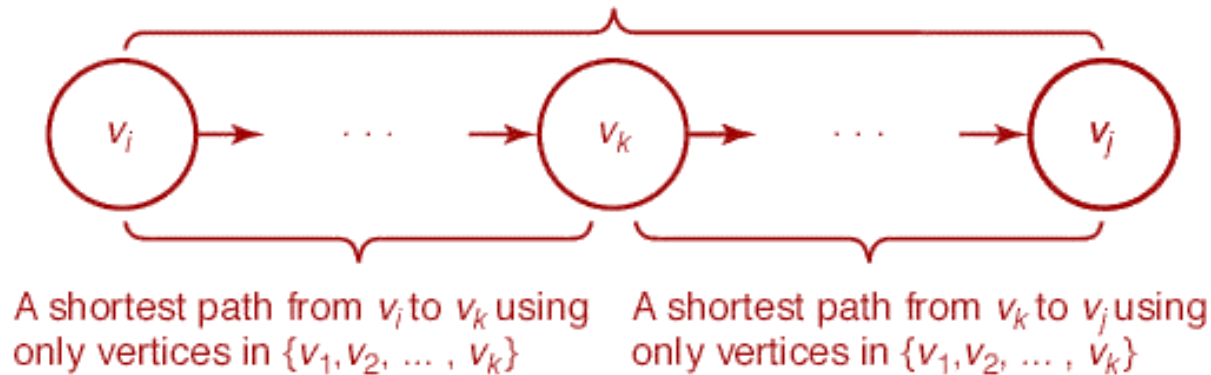
- 경우2. V_i 에서 V_j 로의 모든 최단경로가 V_k 를 사용하는 경우

$$D(k)[i][j] = D(k-1)[i][k] + D(k-1)[k][j]$$

Floyd의 최단 경로 알고리즘 I 4/6

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

■ 경우2.



- v_i 에서 v_j 로의 최단 경로가 v_k 를 중간 노드로 사용하는 경우이다.
- 따라서 이 경로는 두 구간으로 나눌 수 있다.
 - v_i 에서 v_k 로의 최단 경로(v_k 는 포함될 수 없음): $D(k-1)[i][k]$
 - v_k 에서 v_j 로의 최단 경로(v_k 는 포함될 수 없음): $D(k-1)[k][j]$
- 그러므로 다음이 성립한다.

$$D(k)[i][j] = D(k-1)[i][k] + D(k-1)[k][j]$$

Floyd의 최단 경로 알고리즘 I 5/6

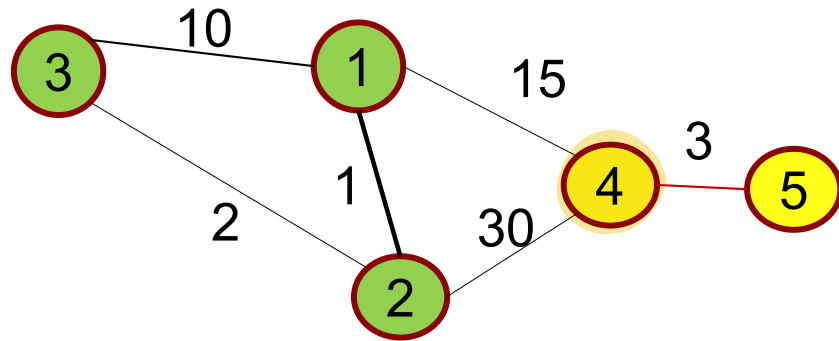
- 경우 1과 경우 2를 모두 고려하면 다음과 같다.

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

- $D[5][4]$?
- $D(0)[5][4] = \infty$
- $D(1)[5][4] = \min(D(0)[5][4], D(0)[5][1] + D(0)[1][4])$
 $= \min(\infty, 3 + 1) = 4$
- $D(2)[5][4] = \min(D(1)[5][4], D(1)[5][2] + D(1)[2][4])$
 $= \min(4, 4 + 2) = 4$

	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

Floyd의 최단 경로 찾기 예제 1:



W=D(0)

	1	2	3	4	5
1	0	1	10	15	∞
2	15	0	2	30	∞
3	10	2	0	∞	∞
4	15	30	∞	0	3
5	∞	∞	∞	3	0

D(1) *1번경유시*

	1	2	3	4	5
1	0	1	10	15	∞
2	15	0	2	16	∞
3	10	2	0	25	∞
4	15	16	25	0	3
5	∞	∞	∞	3	0

D(2)

	1	2	3	4	5
1	0	1	3	15	∞
2	15	0	2	16	∞
3	3	2	0	18	∞
4	15	16	18	0	3
5	∞	∞	∞	3	0

D(3)

	1	2	3	4	5
1	0	1	3	15	∞
2	15	0	2	16	∞
3	3	2	0	18	∞
4	15	16	18	0	3
5	∞	∞	∞	3	0

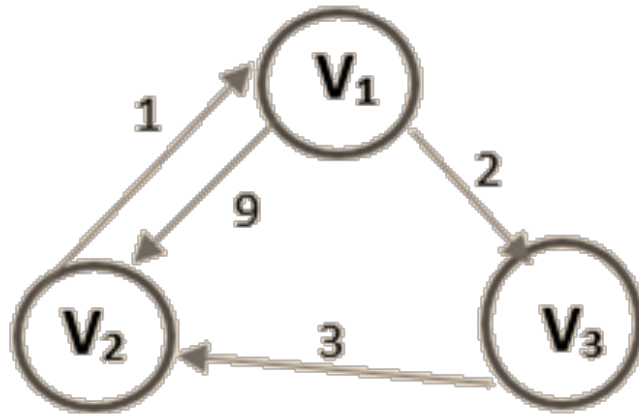
D(4)

	1	2	3	4	5
1	0	1	3	15	18
2	15	0	2	16	19
3	3	2	0	18	21
4	15	16	18	0	3
5	18	19	21	3	0

D(5)

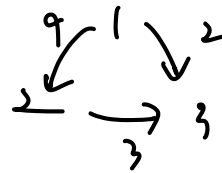
	1	2	3	4	5
1	0	1	3	15	18
2	15	0	2	16	19
3	3	2	0	18	21
4	15	16	18	0	3
5	18	19	21	3	0

Floyd의 최단 경로 찾기 예제:



$W=D(0)$

0	9	2
1	0	∞
∞	3	0



$D(1)$

	1	2	3
1	0	9	2
2	1	0	3
3	∞	3	0

1은 9보다 작게 기록할 필요가
없어지지 않음

$D(2)$

	1	2	3
1	0	9	2
2	1	0	3
3	4	3	0

$D(3)$

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

Floyd의 최단 경로 알고리즘 I 6/6

- 문제:가중치방향 그래프에서 최단 경로 찾기(가중치는 양수)
- 입력: 가중치방향 그래프 W , W 에 있는 정점의 수 n
- 출력: 2차원배열, 이 배열의 각 요소는 한 정점에서 다른 정점으로 최단 경로의 길이를 나타낸다.

```
void floyd(int n, const number W[], number D[][]) {
```

```
    int i, j, k;
```

```
    D = W;
```

```
    for(k=1; k <= n; k++)
```

```
        for(i=1; i <= n; i++)
```

```
            for(j=1; j <= n; j++)
```

```
                D[i][j] = minimum(D[i][j], D[i][k]+D[k][j]);
```

```
    }
```

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

Floyd의 최단 경로 알고리즘 II 1/2

- 문제, 입력: Floyd 알고리즘I과 동일
- 출력: Floyd 알고리즘I의 출력, 다음과 같은 2차원 배열 P

- $P[i][j]$: i 에서 j 로 최단경로 상에 중간 노드가
 - 없는 경우는 0,
 - 있으면 중간 노드 중에서 색인 값이 가장 큰 노드,

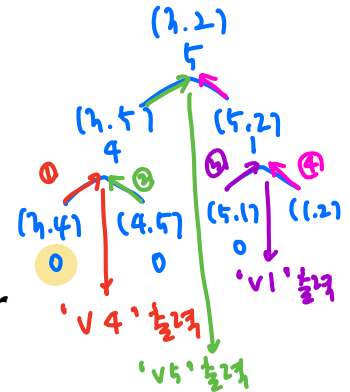
경유하는 노드 저장
: 마지막에 남는 건 경유 노드 중
인덱스가 가장 큰 노드

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

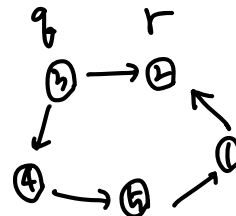
Floyd의 최단 경로 알고리즘 II 2/2

■ 최단 경로 출력하기

- 문제: 가중치방향 그래프에서 최단경로의 중간 노드들을 출력하기
- 입력: **Floyd** 알고리즘II에 의해 생성된 배열 P , 시작과 끝 정점 q 와 r
- 출력: q 와 r 까지의 최단 경로에 포함 되는 중간 노드들



```
void path(index q,r) {
    if (P[q][r] != 0) {
        path(q,P[q][r]);
        print " v" P[q][r];
        path(P[q][r],r);
    }
}
```



$P[i][j]$	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

v_3 에서 v_2 으로 가는 최단 경로:
 v_3, v_4, v_5, v_1, v_2

Principle of Optimality

최적의 원칙

최적화 문제와 동적 프로그래밍

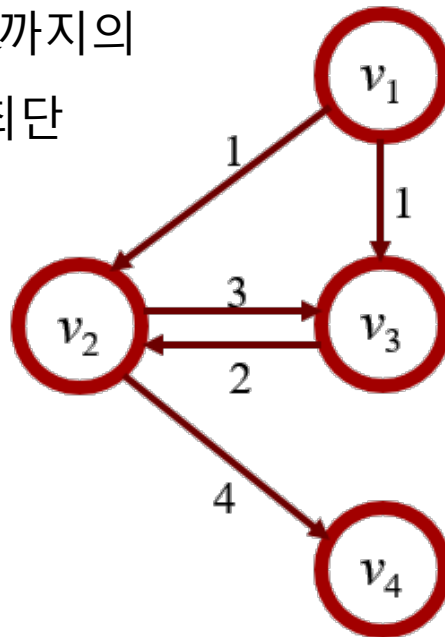
- 동적 프로그래밍을 이용한 최적화 문제를 해결하는 알고리즘 개발 과정
 - 단계1.문제의 사례에 대한 최적해를 주는 재귀관계식 정립
 - 단계2.상향식 방법으로 최적해의 값을 계산.
 - 예)최단 경로의 길이를 구한다.
 - 단계3.상향식 방법으로 최적해를 구성.
 - 예)최단 경로를 구한다.
- 모든 최적화 문제는 동적 프로그래밍으로 해결할 수 있는 것은 아니다.
- 최적원칙(*principle of optimality*): 어떤 문제의 사례에 대한 최적해가 그 사례의 모든 부분사례의 최적해를 항상 포함하고 있으면 이 원칙이 적용.
 - 이 원칙이 적용되어야 동적 프로그래밍으로 최적화 문제를 해결.

Principle Of Optimality

■ 적용되는 경우

• 최단 경로 문제

- v_i 에서 v_j 까지 최단 경로가 v_k 를 포함하면 이 경로의 부분 경로인 v_i 에서 v_k 까지의 경로는 두 정점 간의 최단 경로.



■ 적용되지 않는 경우

• 최장 경로 문제

- 문제의 대한 해를 단순 경로로 제한
- v_1 에서 v_4 까지의 최장 경로
 - $(v_1, v_3, v_2, v_4) = 7$
- v_1 에서 v_3 까지의 최장 경로
 - $(v_1, v_2, v_3) = 4$
- v_1 에서 v_4 까지의 최장 경로의 부분 경로는 항상 최장 경로가 아니다.