

Chap 3. *Dynamic Programming*

1. The Binomial Coefficient
2. Floyd's Algorithm for Shortest Paths
3. Dynamic Programming & Optimization Problems
4. Chained Matrix Multiplication
5. Optimal Binary Search Trees
6. The Traveling Salesperson Problem
7. Sequence Alignment

Dynamic programming

- Dynamic programming
 - Similar to divide-and-conquer
 - an instance of a problem is divided into smaller instances
 - Solve small instances first, store the results, and later, whenever we need a result, look it up instead of recomputing it
 - Term “Dynamic programming” comes from control theory
 - Programming
 - use of an array (table) in which solution is constructed
- Divide-and-conquer 알고리즘 설계법은 하향식 해결법으로서, 나누어진 부분들 사이에 서로 상관관계가 없는 문제를 해결하는데 적합
 - 피보나찌 알고리즘의 경우에는 나누어진 부분들이 서로 연관이 있다.
 - 즉, **divide-and-conquer** 방법을 적용하여 알고리즘을 설계하면 같은 항을 한 번 이상 계산하는 결과를 초래하게 되므로 효율적이지 않다. 따라서 이 경우에는 **divide-and-conquer** 방법은 적합하지 않다.

Dynamic programming

- The steps in the development of Dynamic programming
 - *Establish* a recursive property that gives the solution to an instance of the problem
 - Solve an instance of the problem in a *bottom-up* fashion by solving smaller instances first

Binomial Coefficient

- 이항계수 구하는 공식

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

- 계산량이 많은 $n!$ 이나 $k!$ 을 계산하지 않고
이항계수 (**binomial coefficient**)를 구하기 위해서
통상 다음식을 사용한다.

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

Binomial Coefficient

□ 알고리즘: Using Divide-and-Conquer

- 문제: 이항계수를 계산한다.
 - 입력: 음수가 아닌 정수 n 과 k , 여기서 $k \leq n$
 - 출력: $\text{bin}, \begin{bmatrix} n \\ k \end{bmatrix}$
- 알고리즘:

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

Binomial Coefficient

□ 시간복잡도 분석:

- 분할정복 알고리즘은 작성하기는 간단하지만, 효율적이지 않다.
 - 이유? : 알고리즘을 재귀호출(**recursive call**)할 때 같은 계산을 반복해서 수행하기 때문이다.
 - 예를 들면, **bin(n-1,k-1)**과 **bin(n-1,k)**는 둘 다 **bin(n-2,k-1)**의 결과가 필요한데, 따로 중복 계산됨

- $\begin{bmatrix} n \\ k \end{bmatrix}$ 을 구하기 위해서 이 알고리즘이 계산하는 항(**term**)의 개수는 $2^{\begin{bmatrix} n \\ k \end{bmatrix}} - 1$ 이다. (증명을 해보자.)

Binomial Coefficient

□ 증명: (n 에 대한 수학적귀납법으로 증명)

- 귀납출발점: 항의 개수 n 이 1일 때 $2^{\binom{n}{k}} - 1 = 2 \times 1 - 1 = 1$ 이 됨을 보이면 된다.
 $\binom{1}{k}$ 는 $k = 0$ 이나 1 일 때 1이므로 항의 개수는 항상 1이다.

- 귀납가정: $\binom{n}{k}$ 을 계산하기 위한 항의 개수는 $2^{\binom{n}{k}} - 1$ 이라고 가정한다.

- 귀납절차: $\binom{n+1}{k}$ 을 계산하기 위한 항의 개수가 $2^{\binom{n+1}{k}} - 1$ 임을 보이면 된다.

알고리즘에 의해서 $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$ 이므로, $\binom{n+1}{k}$ 를 계산하기 위한 항의 총 개수는 $\binom{n}{k-1}$ 을 계산하기 위한 총 개수와 $\binom{n}{k}$ 를 계산하기 위한 항의 총 개수에다가 이 둘을 더하기 위한 항 1을 더한 수가 된다.

그런데 $\binom{n}{k-1}$ 을 계산하기 위한 항의 개수는 가정에 의해서 $2^{\binom{n}{k-1}} - 1$ 이고, $\binom{n}{k}$ 를 계산하기 위한 항의 개수는 가정에 의해서 $2^{\binom{n}{k}} - 1$ 이다.

Binomial Coefficient

- 따라서 항의 총 개수는

$$\begin{aligned} & 2 \left[\begin{matrix} n \\ k-1 \end{matrix} \right] - 1 + 2 \left[\begin{matrix} n \\ k \end{matrix} \right] - 1 + 1 \\ &= 2 \left(\frac{n!}{(k-1)!(n-k+1)!} + \frac{n!}{k!(n-k)!} \right) - 1 \\ &= 2 \left(\frac{n!(k+n+1-k)}{k!(n+1-k)!} \right) - 1 \\ &= 2 \left(\frac{n!(n+1)}{k!(n+1-k)!} \right) - 1 \\ &= 2 \left(\frac{(n+1)!}{k!(n+1-k)!} \right) - 1 \\ &= 2 \left[\begin{matrix} n+1 \\ k \end{matrix} \right] - 1 \end{aligned}$$

Binomial Coefficient

□ 동적계획식 알고리즘 설계전략

1. Establish a recursive property (재귀 관계식을 정립):

- 2차원 배열 B 를 만들고, 각 $B[i][j]$ 에는 $\begin{bmatrix} i \\ j \end{bmatrix}$ 값을 저장하도록 하면, 그 값은 다음과 같은 관계식으로 계산할 수 있다.

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

Binomial Coefficient

2. Solve an instance of the problem in a bottom-up fashion:

- $\begin{bmatrix} n \\ k \end{bmatrix}$ 를 구하기 위해서는 다음과 같이 $B[0][0]$ 부터 시작하여 위에서 아래로 재귀 관계식을 적용하여 배열을 채워 나가면 된다. 결국 값은 $B[n][k]$ 에 저장된다.

	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							

	$B[i-1, j-1]$	$B[i-1, j]$
	↓	↓
	$B[i, j]$	

Binomial Coefficient

□ 동적 계획 알고리즘

- 문제: 이항계수를 계산한다.
 - 입력: 음수가 아닌 정수 n 과 k , 여기서 $k \leq n$
 - 출력: **bin**, $\begin{bmatrix} n \\ k \end{bmatrix}$
- 알고리즘 (3-2):

```
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 <= minimum(i,k); j++)  
            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];  
}
```

Binomial Coefficient

□ 동적 계획 알고리즘의 분석

- 단위연산: **for-j** 루프 안의 문장
- 입력의 크기: n, k

$i = 0$ 일 때 j-루프 수행 횟수	: 1
$i = 1$ 일 때 j-루프 수행 횟수	: 2
$i = 2$ 일 때 j-루프 수행 횟수	: 3
.....	
$i = k-1$ 일 때 j-루프 수행 횟수	: k
$i = k$ 일 때 j-루프 수행 횟수	: $k + 1$
$i = k+1$ 일 때 j-루프 수행 횟수	: $k + 1$
.....	
$i = n$ 일 때 j-루프 수행 횟수	: $k + 1$

따라서 총 수행횟수는:

$$\begin{aligned}
 & 1 + 2 + 3 + \dots + k + (k+1) + \dots + (k+1) \overset{n-k+1 \text{ times}}{=} \frac{k(k+1)}{2} + (n-k+1)(k+1) \\
 & \qquad \qquad \qquad = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk)
 \end{aligned}$$

Binomial Coefficient

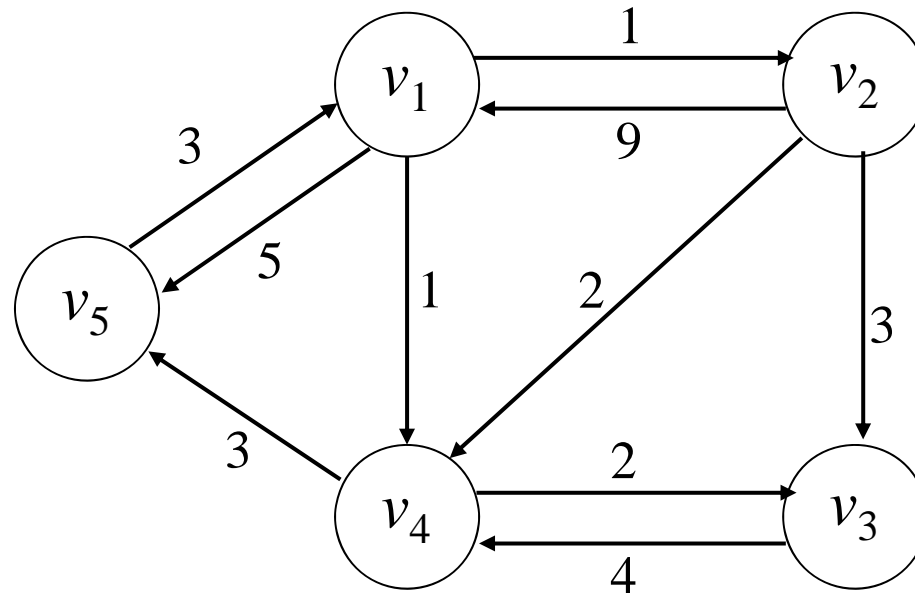
- Possible improvement of Algorithm 3.2
 - Create the entire 2-D array
 - Once a row is computed,
we no longer need the values in the row that precedes it
→ with only 1-D array indexed from 0 to k

- Take advantage of the fact that
$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n \\ n - k \end{bmatrix}$$

그래프 용어

- 정점(vertex, node), 이음선(edge, arc)
- 방향 그래프(directed graph, or digraph)
- 가중치(weight), 가중치 포함 그래프(weighted graph)
- 경로(path) – 두 정점사이에 edge가 있는 정점들의 나열
- 단순경로(simple path) – 같은 정점을 두 번 지나지 않음
- 순환(cycle) – 한 정점에서 다시 그 정점으로 돌아오는 경로
- 순환 그래프(cyclic graph) vs 비순환 그래프 (acyclic graph)
- 길이(length) : the sum of weights on the path (weighted graph)
the number of edges on the path (unweighted graph)

가중치 포함 방향 그래프의 예



- weighted digraph
- vertices, edges, weights, path, cycle, length

Shortest Path

- **Shortest Path:** 한 도시에서 다른 도시로 직항로가 없는 경우 가장 빨리 갈 수 있는 항로를 찾는 문제
- 문제: 가중치 포함, 방향성 그래프에서 최단경로 찾기
- **Optimization problem** (최적화 문제)
 - 주어진 문제에 대하여 하나 이상의 많은 해답이 존재할 때, 이 가운데에서 가장 최적인 해답(**optimal solution**)을 찾아야 하는 문제를 최적화문제(**optimization problem**)라고 한다.
- 최단경로 찾기 문제는 최적화문제에 속한다.

Shortest Path

□ Brute-force algorithm (무작정 알고리즘)

- 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 가정하자.
 - 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 계산해 보자.
 - 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 $(n-2)(n-3)...1 = (n-2)!$ 이 된다.
 - 이 경로의 개수 만 보아도 지수보다 훨씬 크므로, 이 알고리즘은 절대적으로 비효율적이다!

Shortest Path

□ 동적계획식 설계전략 - 자료구조

- 그래프의 인접행렬 (adjacent matrix) 식 표현: W

$$W[i][j] = \begin{cases} \text{이음선의 가중치} & v_i \text{에서 } v_j \text{로의 이음선이 있다면} \\ \infty & v_i \text{에서 } v_j \text{로의 이음선이 없다면} \\ 0 & i = j \text{ 이면} \end{cases}$$

- 그래프에서 최단경로의 길이의 표현: $0 \leq k \leq n$ 인, $D^{(k)}$

$D^{(k)}[i][j] = \{v_1, v_2, \dots, v_k\}$ 의 정점들 만을 통해서
 v_i 에서 v_j 로 가는 최단경로의 길이

동적계획식 설계전략 - 자료구조

□ 보기:

- W : 슬라이드 p.16에 있는 그래프의 인접행렬식 표현
- D : 각 정점들 사이의 최단 거리

$W[i][j]$	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

$D[i][j]$	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

여기서, $0 \leq k \leq 5$ 일 때, $D^{(k)}[2][5]$ 를 구해보자. 예제3.2

- $D^{(0)} = W$ 이고, $D^{(n)} = D$ 임은 분명하다. 따라서 D 를 구하기 위해서는 $D^{(0)}$ 를 가지고 $D^{(n)}$ 을 구할 수 있는 방법을 고안해 내어야 한다.

동적계획식 설계절차

1. Establish a recursive property

- $D^{(k-1)}$ 을 가지고 $D^{(k)}$ 를 계산할 수 있는 재귀 관계식을 정립

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

경우1

경우2

경우 1: $\{v_1, v_2, \dots, v_k\}$ 의 정점들 만을 통해서 v_i 에서 v_j 로 가는 최단경로가 v_k 를 거치지 않는 경우.

$$\text{보기: } D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

경우 2: $\{v_1, v_2, \dots, v_k\}$ 의 정점들 만을 통해서 v_i 에서 v_j 로 가는 최단경로가 v_k 를 거치는 경우.

$$\text{보기: } D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$$

$$\text{보기: } D^{(2)}[5][4]$$

2. 상향식으로 $k = 1$ 부터 n 까지 다음과 같이 이 과정을 반복하여 해를 구한다.

$$D^{(0)}, D^{(1)}, \dots, D^{(n)}$$

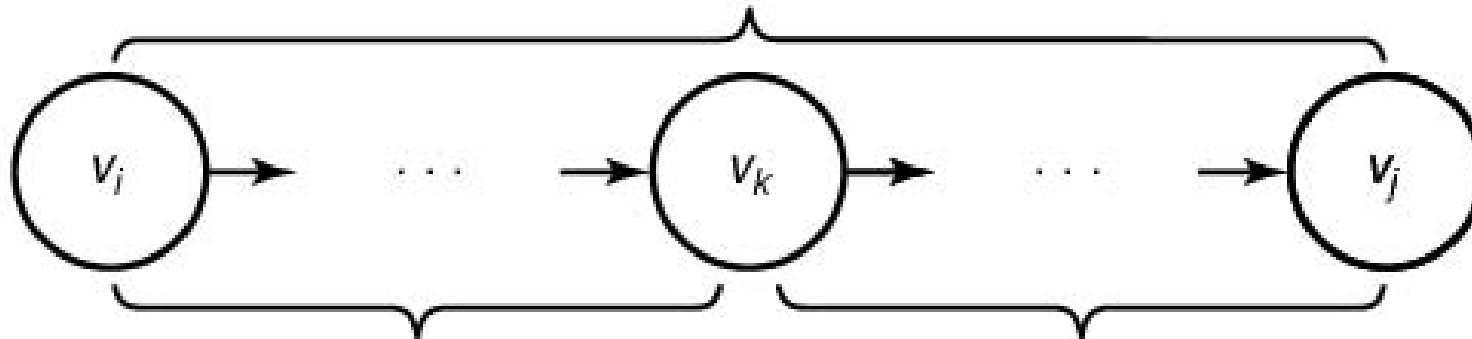
동적계획식 설계절차

$$D^{(k)}[i][j] = \underset{\text{경우1}}{\text{minimum}}(D^{(k-1)}[i][j], \underset{\text{경우2}}{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]})$$

경우1

경우2

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



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

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

Floyd's Algorithm I

□ 문제

- 가중치 포함 그래프의 각 정점에서 다른 모든 정점까지의 최단거리를 계산하라.
- 입력
 - 가중치 포함, 방향성 그래프 W 와 그 그래프에서의 정점의 수 n
- 출력
 - 최단거리의 길이가 포함된 배열 D

Floyd's Algorithm I

- 알고리즘:

```
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]);  
}
```

- 모든 경우를 고려한 분석:
 - ✓ 단위연산: for-j 루프안의 지정문
 - ✓ 입력크기: 그래프에서의 정점의 수 n

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

Floyd's Algorithm II

□ 문제

- 가중치 포함 그래프의 각 정점에서 다른 모든 정점까지의 최단거리를 계산하고, 각각의 최단경로를 구하라.
- 입력
 - 가중치 포함 방향성 그래프 W 와 그 그래프에서의 정점의 수 n
- 출력
 - 최단경로의 길이가 포함된 배열 D , 그리고 다음을 만족하는 배열 P

$$P[i][j] = \begin{cases} v_i \text{에서 } v_j \text{ 까지 가는 최단경로의 중간에 놓여 있는 정점이 최소한} \\ \text{하나는 있는 경우} \rightarrow \text{그 놓여 있는 정점 중에서 가장 큰 인덱스} \\ \\ \text{최단경로의 중간에 놓여 있는 정점이 없는 경우} \rightarrow 0 \end{cases}$$

Floyd's Algorithm II

□ 알고리즘:

```
void floyd2(int n, const number W[][], number D[][], index 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];  
                }  
}
```

Floyd's Algorithm II

- 앞의 예를 가지고 D 와 P 를 구해 보시오.

	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

최단경로의 출력

- 문제: 최단경로 상에 놓여 있는 정점을 출력하라.
- 알고리즘:

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

- 위의 P를 가지고 path(5,3)을 구해 보시오.

```
path(5,3) = 4  
path(5,4) = 1  
path(5,1) = 0  
v1  
path(1,4) = 0  
v4  
path(4,3) = 0
```

결과: v1 v4.

즉, v_5 에서 v_3 으로 가는 최단경로는 v_5, v_1, v_4, v_3 이다.

동적계획법에 의한 설계 절차

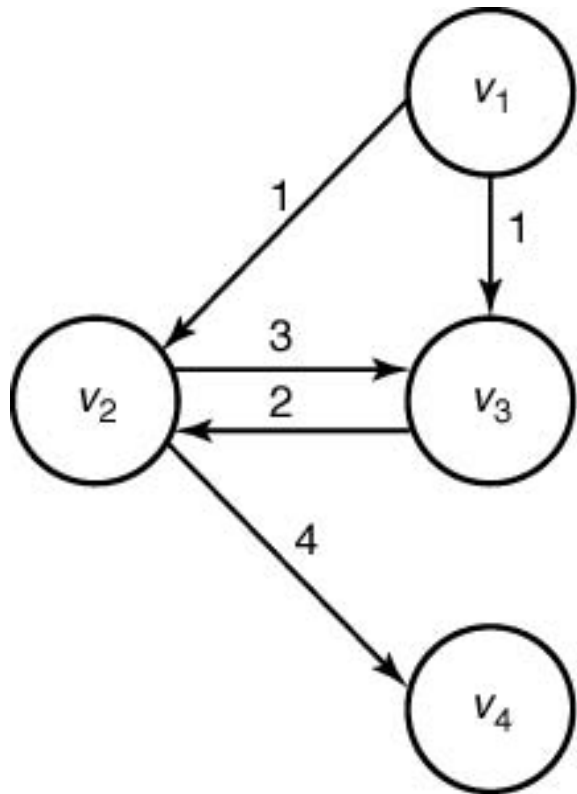
- 문제의 입력에 대해서 최적(**optimal**)의 해답을 주는 재귀 관계식 (**recursive property**)을 설정
- 상향적으로 최적의 해답을 계산
- 상향적으로 최적의 해답을 구축

최적의 원칙

- 어떤 문제의 입력에 대한 최적 해가 그 입력을 나누어 쪼갬 여러 부분에 대한 최적 해를 항상 포함하고 있으면, 그 문제는 최적의 원칙(the principle of optimality)이 적용된다 라고 한다.
- 보기:
 - 최단경로를 구하는 문제에서,
 v_k 를 v_i 에서 v_j 로 가는 최적 경로 상의 정점이라고 하면,
 v_i 에서 v_k 로 가는 부분경로와 v_k 에서 v_j 로 가는 부분경로도 반드시 최적이어야 한다.
 - 이렇게 되면 최적의 원칙을 준수하게 되므로 동적계획법을 사용하여 이 문제를 풀 수 있다.

최적의 원칙이 적용되지 않는 예:

최장경로(Longest Path) 문제



- v_1 에서 v_4 로의 최장경로는 $[v_1, v_3, v_2, v_4]$ 가 된다.
- 그러나 이 경로의 부분 경로인 v_1 에서 v_3 으로의 최장경로는 $[v_1, v_3]$ 이 아니고, $[v_1, v_2, v_3]$ 이다.
- 따라서 최적의 원칙이 적용되지 않는다.
- 주의: 여기서는 단순경로 (simple path), 즉 순환 (cycle)이 없는 경로만 고려한다.

Chained Matrix Multiplication (연쇄 행렬곱셈)

- $i \times j$ 행렬과 $j \times k$ 행렬을 곱하기 위해서는 일반적으로 $i \times j \times k$ 번 만큼의 기본적인 곱셈이 필요하다.
- 연쇄적으로 행렬을 곱할 때, 어떤 행렬곱셈을 먼저 수행하느냐에 따라서 필요한 기본적인 곱셈의 횟수가 달라지게 된다.
- 예를 들어서, 다음 연쇄행렬곱셈을 생각해 보자:
 - $A_1 \times A_2 \times A_3$.
 - $A_1: 10 \times 100, A_2: 100 \times 5, A_3: 5 \times 50$
 - $(A_1 \times A_2) \times A_3$: 기본적인 곱셈의 총 횟수는 **7,500**회
 - $A_1 \times (A_2 \times A_3)$: 기본적인 곱셈의 총 횟수는 **75,000**회
 - 따라서, 연쇄적으로 행렬을 곱할 때 기본적인 곱셈의 횟수가 가장 적게 되는 최적의 순서를 결정하는 알고리즘을 개발하는 것이 목표

Chained Matrix Multiplication

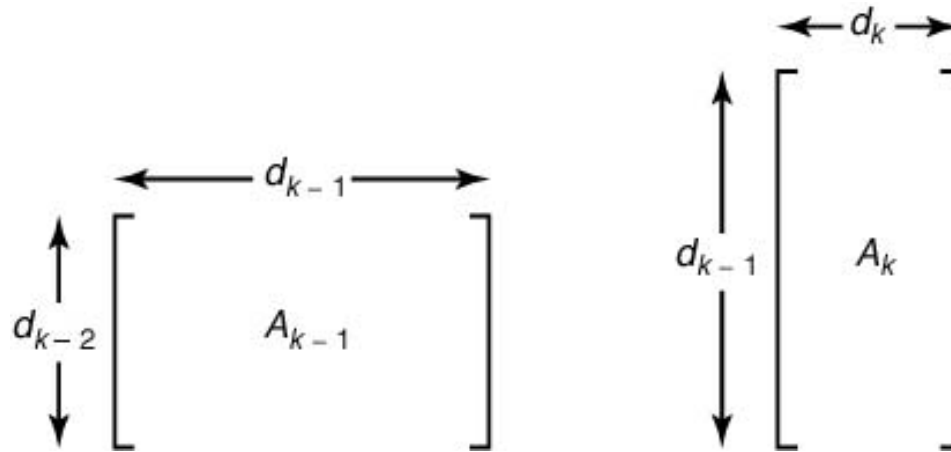
$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

$A(B(CD))$	$30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680$
$(AB)(CD)$	$20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880$
$A((BC)D)$	$2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232$
$((AB)C)D$	$20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320$
$(A(BC))D$	$2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120$

Chained Matrix Multiplication

- 무작정 알고리즘: 가능한 모든 순서를 모두 고려해 보고, 그 가운데에서 가장 최소를 택한다.
- 시간복잡도 분석: 최소한 지수(exponential-time) 시간
- 증명:
 - n 개의 행렬(A_1, A_2, \dots, A_n)을 곱할 수 있는 모든 순서의 가지 수를 t_n 이라고 하자.
 - 만약 A_1 이 마지막으로 곱하는 행렬이라고 하면, 행렬 A_2, \dots, A_n 을 곱하는 데는 t_{n-1} 개의 가지수가 있을 것이다.
 - A_n 이 마지막으로 곱하는 행렬이라고 하면, 행렬 A_1, \dots, A_{n-1} 을 곱하는 데는 또한 t_{n-1} 개의 가지수가 있을 것이다.
 - 그러면, $t_n \geq t_{n-1} + t_{n-1} = 2 t_{n-1}$ 이고 $t_2 = 1$ 이라는 사실은 쉽게 알 수 있다.
 - 따라서 $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \dots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$

Chained Matrix Multiplication



□ d_k 를 행렬 A_k 의 열(column)의 수라고 하자

● 자연히 A_k 의 행(row)의 수는 d_{k-1} ; A_1 의 행의 수는 d_0 라고 하자.

● For $1 \leq i \leq j \leq n$, let

$M[i][j] = i < j$ 일 때 A_i 부터 A_j 까지의 행렬을 곱하는데 필요한 곱셈의 최소 횟수

$$= \text{minimum}_{i \leq k \leq j-1} (M[i][k] + M[k+1][j] + d_{i-1}d_kd_j)$$

$$M[i][j] = 0 \quad \text{if } i < j$$

Chained Matrix Multiplication

□ Ex 3.5

$$\begin{array}{cccccc}
 A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\
 5 \times 2 & 2 \times 3 & 3 \times 4 & 4 \times 6 & 6 \times 7 & 7 \times 8
 \end{array}$$

$$A_4(A_5A_6)$$

$$(A_4A_5)A_6$$

$$\begin{aligned}
 M[4][6] &= \text{minimum}(M[4][4] + M[5][6] + 4 \times 6 \times 8, M[4][5] + M[6][6] + 4 \times 7 \times 8) \\
 &= \text{minimum}(0 + 6 \times 7 \times 8 + 4 \times 6 \times 8, 4 \times 6 \times 7 + 0 + 4 \times 7 \times 8) \\
 &= \text{minimum}(528, 392) = 392
 \end{aligned}$$

$M[i][j]$	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Chained Matrix Multiplication

□ Ex 3.6

- For diagonal 0: $M[i][i] = 0$ for $1 \leq i \leq 6$
- For diagonal 1: $M[1][2] = \min(M[1][k] + M[k+1][2] + d_0 d_k d_2)$
 $= M[1][1] + M[2][2] + d_0 d_1 d_2 = 30$

Compute $M[2][3]$, $M[3][4]$, $M[4][5]$, $M[5][6]$

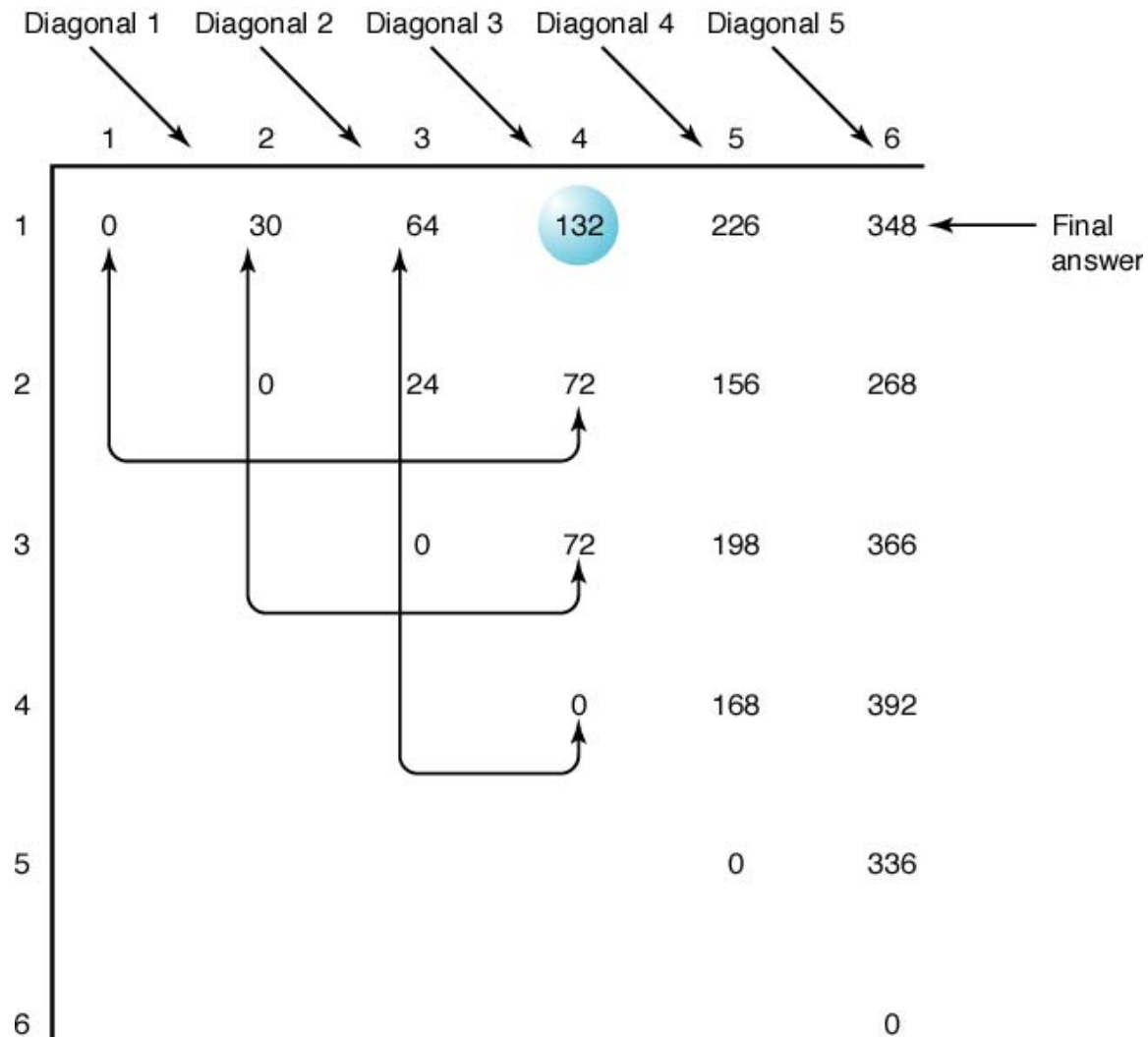
- For diagonal 2: $M[1][3] = \min(M[1][k] + M[k+1][3] + d_0 d_k d_3)$
 $= \min(M[1][1] + M[2][3] + d_0 d_1 d_3,$
 $M[1][2] + M[3][3] + d_0 d_2 d_3)$
 $= \min(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64$

Compute $M[2][4]$, $M[3][5]$, $M[4][6]$

- For diagonal 3: $M[1][4] = \min(M[1][k] + M[k+1][4] + d_0 d_k d_4)$
 $= \min(M[1][1] + M[2][4] + d_0 d_1 d_4,$
 $M[1][2] + M[3][4] + d_0 d_2 d_4,$
 $M[1][3] + M[4][4] + d_0 d_3 d_4)$
 $= \min(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, 64 + 0 + 5 \times 4 \times 6) = 132$

Compute $M[2][5]$, $M[3][6]$

Chained Matrix Multiplication



Chained Matrix Multiplication

- 최소 곱셈 (Minimum Multiplication) 알고리즘
 - 문제
 - n 개의 행렬을 곱하는데 필요한 기본적인 곱셈의 횟수의 최소치를 결정하고, 그 최소치를 구하는 순서를 결정하라.
 - 입력
 - 행렬의 수 n 와 배열 $d[0..n]$,
 $d[i-1] \times d[i]$ 는 i 번째 행렬의 규모를 나타낸다.
 - 출력
 - 기본적인 곱셈의 횟수의 최소치를 나타내는 *minmult*,
최적의 순서를 얻을 수 있는 배열 P ,
여기서 $P[i][j]$ 는 행렬 i 부터 j 까지가 최적의 순서로 갈라지는 기점

Chained Matrix Multiplication

□ 알고리즘:

```
int minmult(int n, const int d[], index P[][][]) {
    index i, j, k, diagonal;
    int M[1..n, 1..n];
    for(i=1; i <= n; i++)
        M[i][i] = 0;
    for(diagonal = 1; diagonal <= n-1; diagonal++)
        for(i=1; i <= n-diagonal; i++) {
            j = i + diagonal;
            M[i][j] = minimum(M[i][k]+M[k+1][j]+d[i-1]*d[k]*d[j]);
                           i <= k <= j-1
            P[i][j] = 최소치를 주는 k의 값
        }
    return M[1][n];
}
```

Chained Matrix Multiplication

□ 최소 곱셈 알고리즘의 모든 경우 분석

- 단위연산: 각 k 값에 대하여 실행된 명령문 (instruction), 여기서 최소값인 k 를 알아보는 비교문도 포함한다.

- 입력크기: 곱할 행렬의 수 n

- 분석: $j = i + diagonal$ 이므로,

- i -루프를 수행하는 횟수 = $n - diagonal$

- k -루프를 수행하는 횟수 =

$$(j - 1) - i + 1 = ((i + diagonal) - 1) - i + 1 = diagonal$$

- 따라서

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Chained Matrix Multiplication

□ 최적 순서의 구축

- 최적 순서를 얻기 위해서는 $M[i][j]$ 를 계산할 때 최소값을 주는 k 값을 $P[i][j]$ 에 기억한다.
- 예: $P[2][5] = 4$ 인 경우의 최적 순서는 $(A_2 A_3 A_4) A_5$ 이다.

$P[i][j]$	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

- $P[1][6] = 1$; $A_1(A_2 A_3 A_4 A_5 A_6)$
- $P[2][6] = 5$; $A_1((A_2 A_3 A_4 A_5) A_6)$
- 따라서 최적 분해는 $(A_1(((A_2 A_3) A_4) A_5) A_6))$.

Chained Matrix Multiplication

- 최적의 해를 주는 순서의 출력
 - 문제: n 개의 행렬을 곱하는 최적의 순서를 출력하시오
 - 입력: n 과 P
 - 출력: 최적의 순서
 - 알고리즘:

```
void order(index i, index j) {  
    if (i == j) cout << "A" << i;  
    else {  
        k = P[i][j];  
        cout << "(";  
        order(i,k);  
        order(k+1,j);  
        cout << ")";  
    }  
}
```

Chained Matrix Multiplication

- 최적의 해를 주는 순서의 출력
 - $\text{order}(i,j)$ 의 의미: $A_i \times \dots \times A_j$ 의 계산을 수행하는데 기본적인 곱셈의 수가 가장 적게 드는 순서대로 괄호를 쳐서 출력하시오.
 - 분석: $T(n) \in \Theta(n)$. 어떻게?

- Chained matrix multiplication
 - $\Theta(n^3)$ – Godbole (1973)
 - $\Theta(n^2)$ – Yao (1982)
 - $\Theta(n \lg n)$ – Hu and Shing (1982, 1984)

Optimal Binary Search Trees

□ Definition

- *binary search tree*

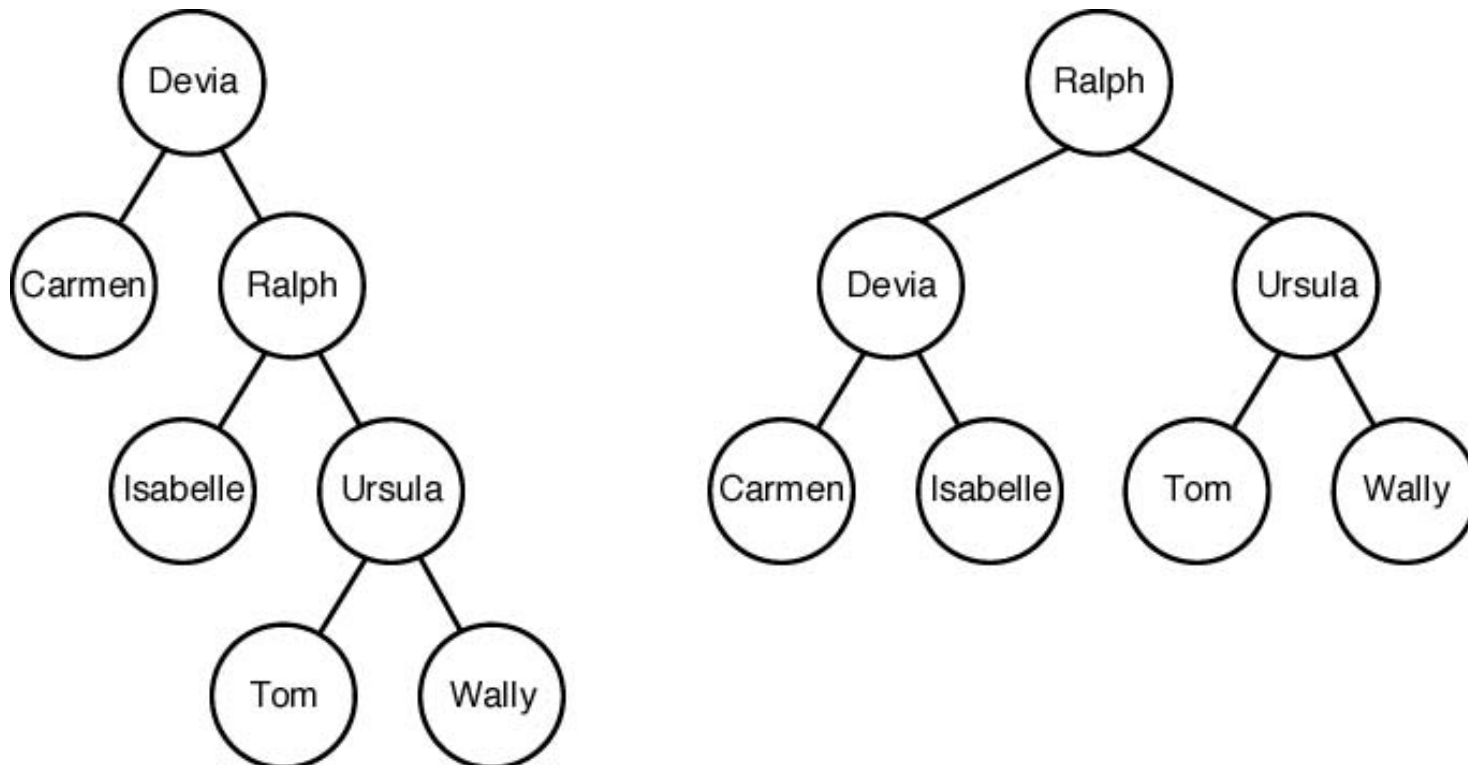
- A binary tree of items (keys), that come from an ordered set
- Each node contain one key
- The keys in the left subtree of a given node are less than or equal to the key in that tree
- The keys in the right subtree of a given node are greater than or equal to the key in that tree

- *depth* – number of edges from the root (level of the node)

- *balanced* – if the depth of the 2 subtrees of every node never differ by more than 1

- *optimal* – the average time it takes to locate a key is minimized

Optimal Binary Search Trees



Two binary search trees

Optimal Binary Search Trees

□ Data types

```
struct nodetype {  
    keytype key  
    nodetype* left  
    nodetype* right  
}  
Typedef nodetype* node_pointer;
```

Optimal Binary Search Trees

- ❑ Problem: determine the node containing a key in a binary search tree
(assume that key is in the tree)
 - Inputs: a pointer tree & a key keyin
 - Outputs: a pointer p to the node containing the key

```
void search (node_pointer tree, keytype keyin,
            node_pointer& p) {
    bool found;
    p = tree;
    found = false;
    while (!found)
        if (p->key == keyin)
            found = true;
        else if (keyin < p->key)
            p = p->left;           // Advance to left child
        else
            p = p->right;          // Advance to right child
}
```

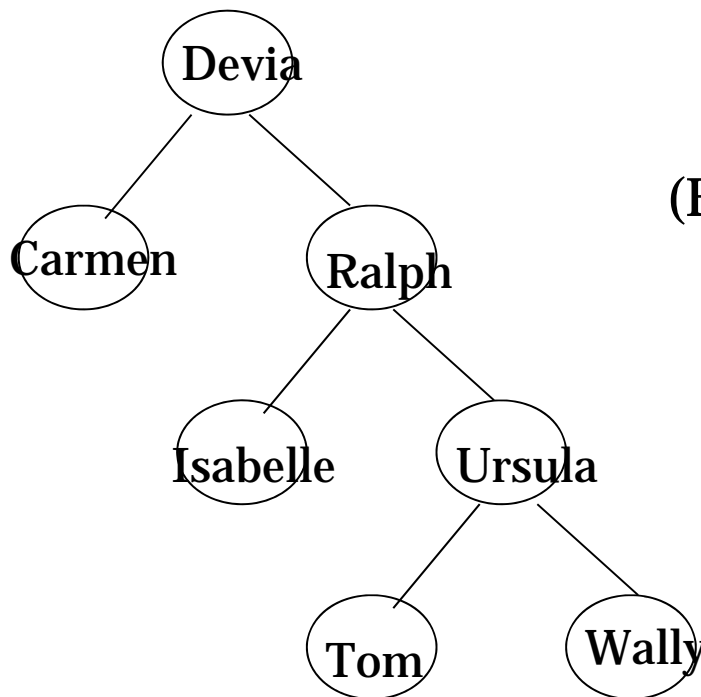
Optimal Binary Search Trees

□ Analysis

- Search time – the number of comparisons to locate a key

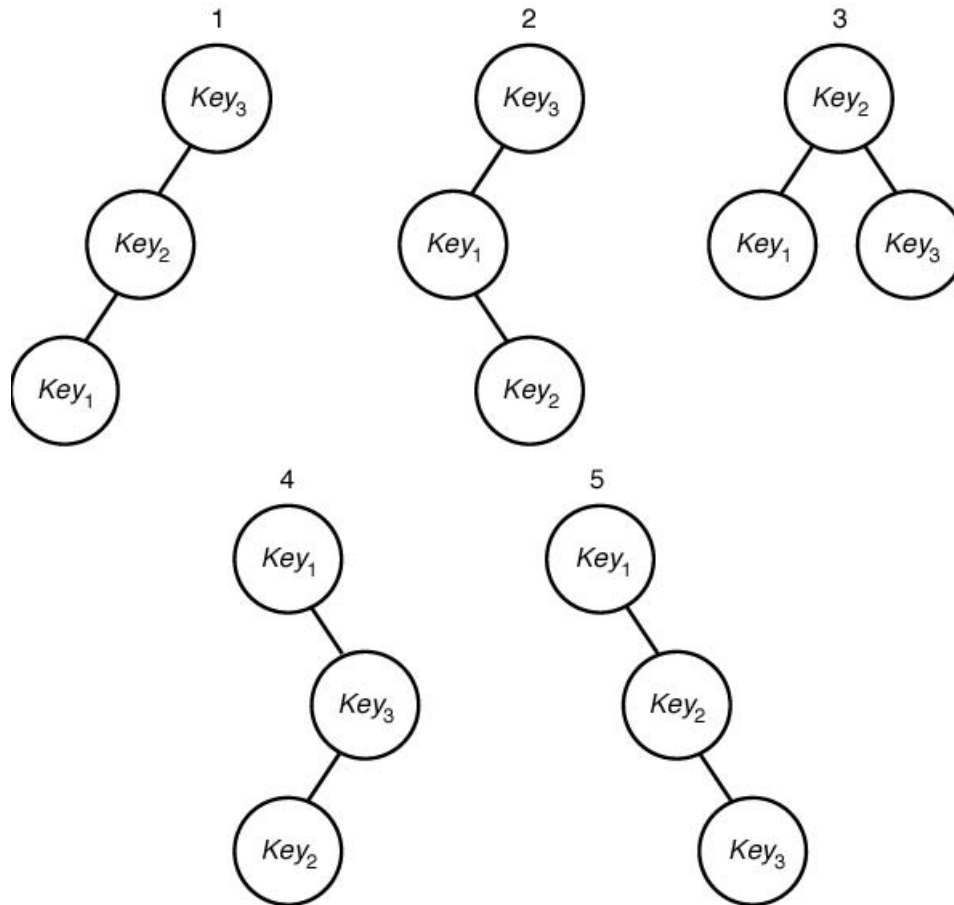
- Search time a given key $\text{depth}(\text{key}) + 1$

where $\text{depth}(\text{key})$ is the depth of the node containing the key



(Ex) Search time for “Ursula”
 $\text{depth}(\text{Ursula}) + 1 = 2 + 1 = 3$

Optimal Binary Search Trees



p_i – the probability that Key_i is the search key

$$p_1 = 0.7, p_2 = 0.2, p_3 = 0.1$$

$$1) 3(0.7) + 2(0.2) + 1(0.1) = 2.6$$

$$2) 2(0.7) + 3(0.2) + 1(0.1) = 2.1$$

$$3) 2(0.7) + 1(0.2) + 2(0.1) = 1.8$$

$$4) 1(0.7) + 3(0.2) + 2(0.1) = 1.5$$

$$5) 1(0.7) + 2(0.2) + 3(0.1) = 1.4$$

Optimal Binary Search Trees

□ Dynamic programming

- Suppose that Key_i through Key_j are arranged in a tree that

minimizes
$$\sum_{m=i}^j C_m p_m$$

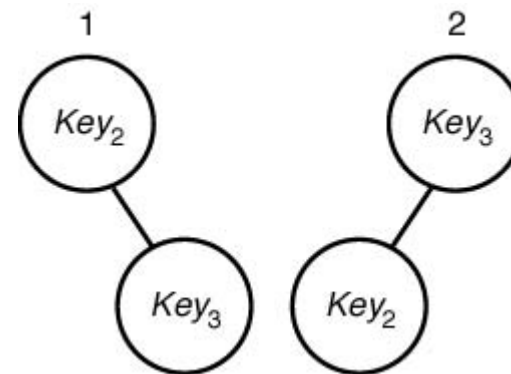
- C_m – the number of comparisons needed to locate Key_m in the tree

- p_m – the probability that Key_m is the search key

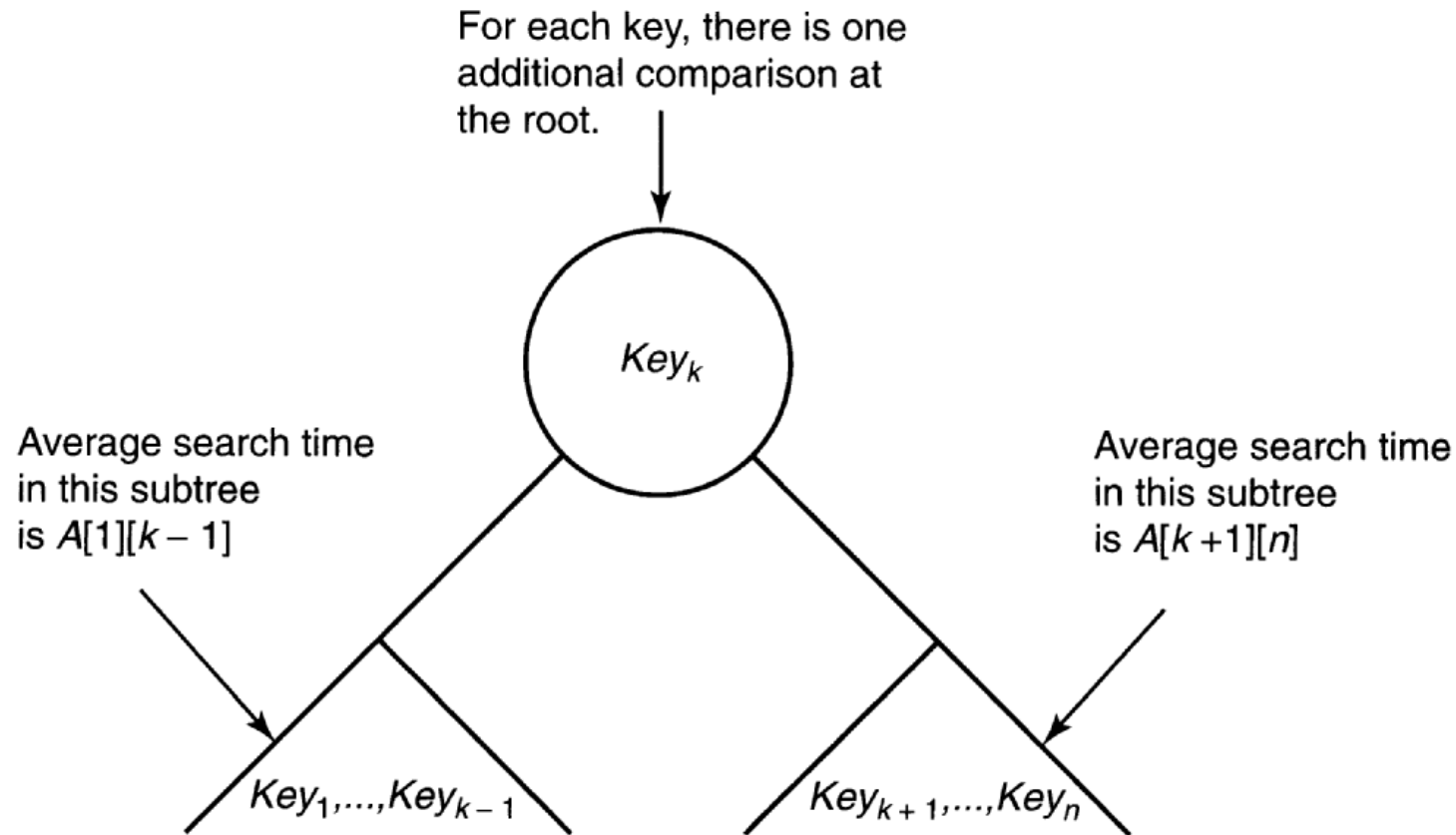
- $A[i][j]$ – the optimal value that minimized the tree

- $A[i][i] = p_i$

Ex 3.8) Determine $A[2][3]$



Optimal Binary Search Trees



Average time for tree k , $A[1][n] =$

$$\begin{array}{ccccccc}
 A[1][k-1] & + & p_1 + \dots + p_{k-1} & + & p_k & + & A[k+1][n] + p_{k+1} + \dots + p_n \\
 \text{Average time} & & \text{Additional time} & & \text{Average time} & & \text{Average time} \\
 \text{in left subtree} & & \text{comparing at root} & & \text{searching at root} & & \text{in right subtree} \\
 & & & & & & \text{Additional time} \\
 & & & & & & \text{comparing at root}
 \end{array}$$

Optimal Binary Search Trees

$$A[1][k-1] + p_1 + \dots + p_{k-1} + p_k + A[k+1][n] + p_{k+1} + \dots + p_n$$

$$= A[1][k-1] + A[k+1][n] + \sum_{m=1}^n p_m$$

$$A[1][n] = \min_{1 \leq k \leq n} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^n p_m$$

$$A[i][j] = \min_{i \leq k \leq j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^j p_m \quad (i < j)$$

$$A[i][i] = p_i$$

$$A[i][i-1] = 0 \quad \text{and} \quad A[j+1][j] = 0$$

Optimal Binary Search Trees

□ Optimal Binary Search Tree Algorithm

```
void optsearchtree(int n, const float p[], float& minavg,
                  index R[][]) {
    index i, j, k, diagonal;
    float A[1..n+1][0..n];
    for(i=1; i<=n; i++) {
        A[i][i-1] = 0; A[i][i] = p[i]; R[i][i] = i; R[i][i-1] = 0;
    }
    A[n+1][n] = 0; R[n+1][n] = 0;
    for(diagonal=1; diagonal<= n-1; diagonal++)
        for(i=1; i <= n-diagonal; i++) {
            j = i + diagonal;
            A[i][j] = min(A[i][k-1]+A[k+1][j]) +  $\sum_{m=i}^j p_m$ 
            R[i][j] = a value of k that gave the minimum;
        }
    minavg = A[1][n];
}
```

Optimal Binary Search Trees

□ Every-case Time Complexity Analysis

- Basic operation: addition & comparison for each value of k
- Input size: n , the number of keys
- 분석: $j = i + diagonal$ 이므로, (Algorithm 3.6 참조)
 - i -루프를 수행하는 횟수 = $n - diagonal$
 - k -루프를 수행하는 횟수 = $j - i + 1 = (i + diagonal) - i + 1 = diagonal + 1$
 - 따라서

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times (diagonal + 1)] = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

Optimal Binary Search Trees

□ Build Binary Search Tree Algorithm

Output: a pointer tree to an optimal binary search tree containing the n keys

```
nodepointer tree(index i, j) {  
    index k;  
    node_pointer p;  
    k = R[i][j];  
    if(k==0)  
        return NULL;  
    else{  
        p = new nodetype;  
        p -> key = key[k];  
        p -> left = tree(i, k-1);  
        p -> right = tree(k+1, j);  
        return p;  
    }  
}
```

Optimal Binary Search Trees

□ Example 3.9

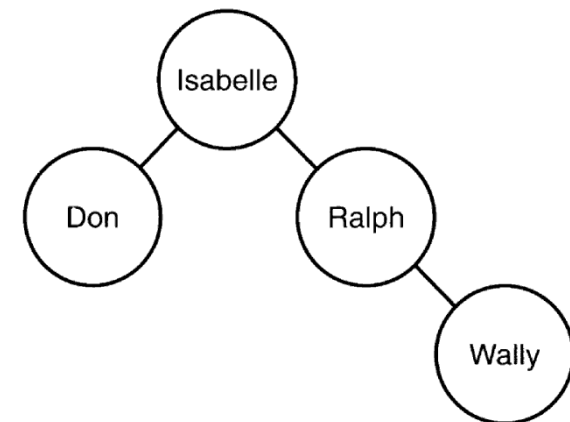
● Don Isabelle Ralph Wally
 Key[1] Key[2] Key[3] Key[4]
 $p_1=3/8$ $p_2=3/8$ $p_3=1/8$ $p_4=1/8$

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

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

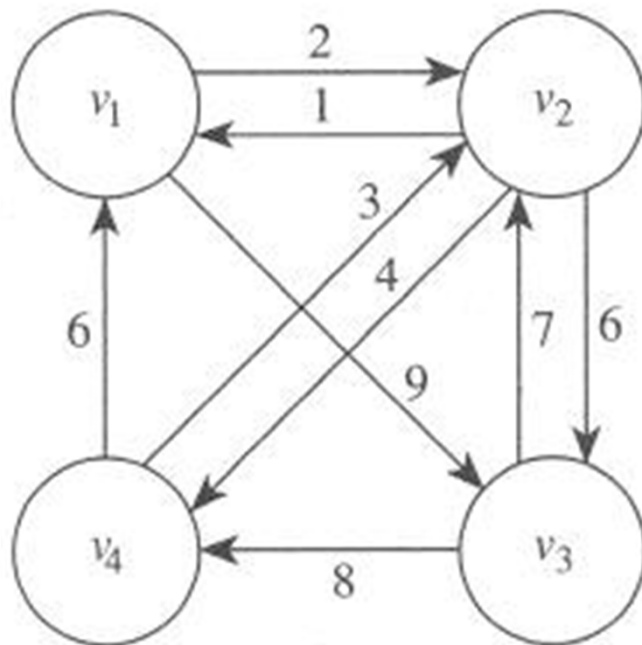
R



The Traveling Salesperson Problem

□ Problem Definition

- Determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city



$$\text{length}[v_1, v_2, v_3, v_4, v_1] = 22$$

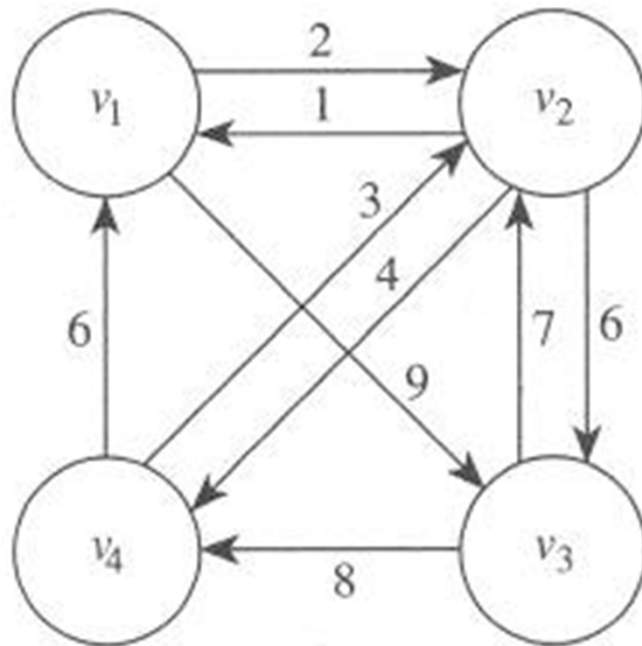
$$\text{length}[v_1, v_3, v_2, v_4, v_1] = 26$$

$$\text{length}[v_1, v_3, v_4, v_2, v_1] = 21$$

$$(n - 1)(n - 2) \cdots 1 = (n - 1)!$$

The Traveling Salesperson Problem

▣ Adjacent matrix W



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

- V = set of all the vertices
- A = a subset of V
- $D[v_i][A]$ = length of a shortest path from v_i to v_1
passing through each vertex in A exactly once

The Traveling Salesperson Problem

Ex 3.10

- $V = \{v_1, v_2, v_3, v_4\}$ represent a set
- $[v_1, v_2, v_3, v_4]$ represent a path
- If $A = \{v_3\}$, then $D[v_2][A] = \text{length}[v_2, v_3, v_1] = \infty$
- If $A = \{v_3, v_4\}$,

$$\begin{aligned}\text{then } D[v_2][A] &= \min(\text{length}[v_2, v_3, v_4, v_1], \text{length}[v_2, v_4, v_3, v_1]) \\ &= \min(20, \infty) = 20\end{aligned}$$

- Length of an optimal tour = $\underset{2 \leq j \leq n}{\text{minimum}}(W[1][j] + D[v_j][V - \{v_1, v_j\}])$
- In general for $i \neq 1$ and i not in A

$$\begin{aligned}D[v_i][A] &= \underset{v_j \in A}{\text{minimum}}(W[i][j] + D[v_j][A - \{v_j\}]) \quad \text{if } A \neq \emptyset \\ D[v_i][\emptyset] &= W[i][1].\end{aligned}$$

The Traveling Salesperson Problem

□ Ex 3.11 Determine an optimal tour in Fig 3.17

- $D[v_2][\emptyset] = 1$; $D[v_3][\emptyset] = \infty$; $D[v_4][\emptyset] = 6$
- $D[v_3][\{v_2\}] = \min_{j:v_j \in \{v_2\}} (W[3][j] + D[v_j][\{v_2\}-\{v_j\}])$
 $= W[3][2] + D[v_2][\emptyset] = 7 + 1 = 8$

$$D[v_4][\{v_2\}] = 3+1 = 4;$$

$$D[v_2][\{v_3\}] = 6 + \infty = \infty; D[v_4][\{v_3\}] = \infty + \infty = \infty;$$

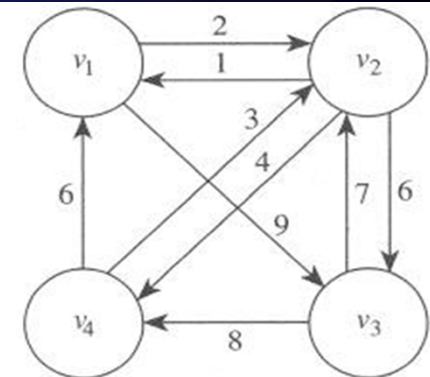
$$D[v_2][\{v_4\}] = 4+6 = 10; D[v_3][\{v_4\}] = 8+6 = 14;$$

- $D[v_4][\{v_2, v_3\}] = \min_{j:v_j \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\}-\{v_j\}])$
 $= \min(W[4][2]+D[v_2][\{v_3\}], W[4][3]+D[v_3][\{v_2\}])$
 $= \min(3+\infty, \infty+8) = \infty$

$$D[v_3][\{v_2, v_4\}] = \min(7+10, 8+4) = 12$$

$$D[v_2][\{v_3, v_4\}] = \min(6+14, 4+\infty) = 20$$

- $D[v_1][\{v_2, v_3, v_4\}] = \min_{j:v_j \in \{v_2, v_3, v_4\}} (W[1][j] + D[v_j][\{v_2, v_3, v_4\}-\{v_j\}])$
 $= \min(W[1][2]+D[v_2][\{v_3, v_4\}], W[1][3]+D[v_3][\{v_2, v_4\}], W[1][4]+D[v_4][\{v_2, v_3\}])$
 $= \min(2+20, 9+12, \infty+\infty) = 21$



	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

The Traveling Salesperson Problem

Algorithm for the Traveling Salesperson Problem

```
void travel(int n, const number W[], index P[][], number& minlength){
    index i, j, k;
    number D[1..n][subset of  $V - \{v_1\}$ ];

    for(i=2; i<=n; i++)
        D[i][ $\emptyset$ ] = W[i][1];

    for(k=1; k<= n-2; k++)
        for(all subsets  $A \subseteq V - \{v_1\}$  containing k vertices)
            for(i such that  $i \neq 1$  and  $v_i$  is not in A){
                D[i][A] =  $\min_{(j: v_j \in A)} (W[i][j] + D[j][A - \{v_j\}]);$ 
                P[i][A] = value of j that gave the minimum;
            }

    D[1][ $V - \{v_1\}$ ] =  $\min_{(2 \leq j \leq n)} (W[1][j] + D[j][A - \{v_1, v_j\}]);$ 
    P[1][ $V - \{v_1\}$ ] = value of j that gave the minimum;
    minlength = D[1][ $V - \{v_1\}$ ]
}
```

The Traveling Salesperson Problem

□ Theorem 3.1

For all $n \geq 1$

$$\sum_{k=1}^n k \cdot \binom{n}{k} = n \cdot 2^{n-1}$$

(Prove it!!)

(Hint) For all $n \geq 1$, $\sum_{k=0}^n \binom{n}{k} = 2^n$ from $(x+1)^n = \sum_{k=0}^n \binom{n}{k} \cdot x^k$

The Traveling Salesperson Problem

□ Every-case Time Complexity Analysis

- Basic operation: addition for each value of v_j (ignore the first & last loops)
- Input size: n , the number of vertices in the graph

- Analysis

- The number of subsets A of $V - \{v_1\}$ containing k vertices is $\binom{n-1}{k}$
- for each set A containing k vertices,

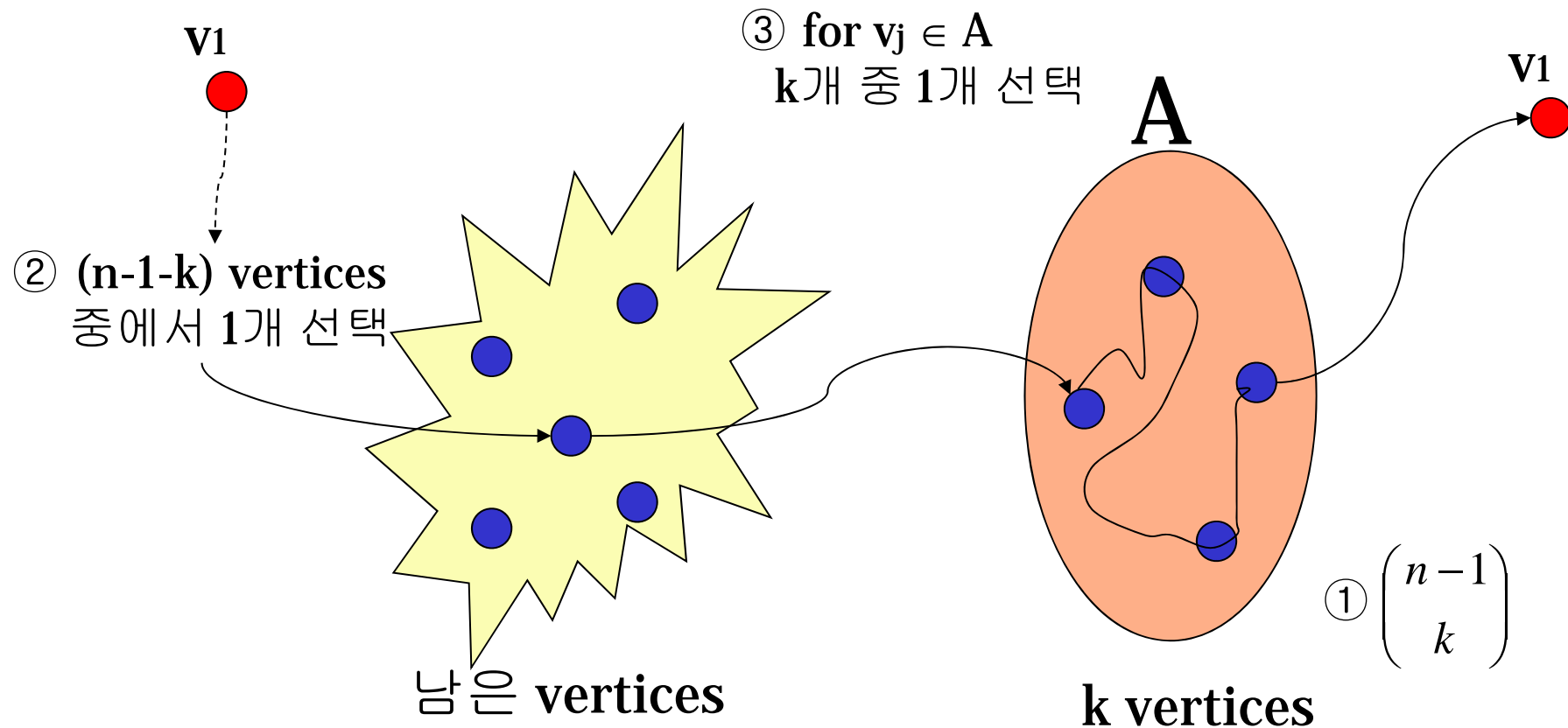
$n-1-k$ vertices, and k operations for each vertices

- The total number is

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-2} (n-1-k) \cdot k \cdot \binom{n-1}{k} \\ &= \sum_{k=1}^{n-2} (n-1) \cdot k \cdot \binom{n-2}{k} \\ &= (n-1) \cdot (n-2) \cdot 2^{n-3} \in \Theta(n^2 2^n) \end{aligned}$$

The Traveling Salesperson Problem

□ Every-case Time Complexity Analysis



The Traveling Salesperson Problem

□ Every-case Space Complexity Analysis

- Memory to store the arrays $D[v_i][A]$ and $P[v_i][A]$ is dominant
→ determine how large these arrays must be

- Because $V - \{v_1\}$ contains $n-1$ vertices, it has 2^{n-1} subsets A

$$\left(\sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}\right)$$

- The first index of the array D & P ranges in value $1 \sim n$
- Therefore,

$$M(n) = 2 \cdot n2^{n-1} = n2^n \in \Theta(n2^n)$$

The Traveling Salesperson Problem

□ Ex 3.12

● 20-city territory

- Brute-force algorithm: $19! \text{ usecs} = 3857 \text{ yrs}$
- Dynamic programming: $(20-1) (20-2) 2^{20-3} \text{ usecs} = 45 \text{ secs}$
- $M(n) = 20 \cdot 2^{20} = 20,971,520 \text{ array slots}$

The Traveling Salesperson Problem

- Retrieve an optimal tour from array P
 - $P[1][\{v_2, v_3, v_4\}] = 3$
 $P[3][\{v_2, v_4\}] = 4$
 $P[4][\{v_2\}] = 2$
 - Therefore, the optimal tour is $[v_1, v_3, v_4, v_2, v_1]$

- No one has ever found an algorithm for the Traveling Salesperson Problem whose worst-case time complexity is better than exponential. Yet no one has ever proved that such an algorithm is not possible.