Chap 3. Dynamic Programming

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

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

- 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 n과 k, 여기서 $k \le n$
    - 출력: $\begin{bmatrix} n \\ k \end{bmatrix}$
  - 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**: (n에 대한 수학적귀납법으로 증명)
  - <u>귀납출발점</u>: 항의 개수 n이 1일 때 $2\begin{bmatrix} n \\ k \end{bmatrix} - 1 = 2 \times 1 - 1 = 1$ 이 됨을 보이면 된다.
    $\begin{bmatrix} 1 \\ k \end{bmatrix}$는 k = 0이나 1 일 때 1이므로 항의 개수는 항상 1이다.
  - <u>귀납가정</u>: $\begin{bmatrix} n \\ k \end{bmatrix}$을 계산하기 위한 항의 개수는 $2\begin{bmatrix} n \\ k \end{bmatrix} - 1$ 이라고 가정한다.
  - <u>귀납절차</u>: $\begin{bmatrix} n+1 \\ k \end{bmatrix}$을 계산하기 위한 항의 개수가 $2\begin{bmatrix} n+1 \\ k \end{bmatrix} - 1$임을 보이면 된다.
    알고리즘에 의해서 $\begin{bmatrix} n+1 \\ k \end{bmatrix} = \begin{bmatrix} n \\ k-1 \end{bmatrix} + \begin{bmatrix} n \\ k \end{bmatrix}$ 이므로, $\begin{bmatrix} n+1 \\ k \end{bmatrix}$를 계산하기 위한 항의 총 개수는 $\begin{bmatrix} n \\ k-1 \end{bmatrix}$을 계산하기 위한 총 개수와 $\begin{bmatrix} n \\ k \end{bmatrix}$를 계산하기 위한 항의 총 개수에다가 이 둘을 더하기 위한 항 1을 더한 수가 된다.
    그런데 $\begin{bmatrix} n \\ k-1 \end{bmatrix}$을 계산하기 위한 항의 개수는 가정에 의해서 $2\begin{bmatrix} n \\ k-1 \end{bmatrix} - 1$ 이고, $\begin{bmatrix} n \\ k \end{bmatrix}$를 계산하기 위한 항의 개수는 가정에 의해서 $2\begin{bmatrix} n \\ k \end{bmatrix} - 1$ 이다.

Binomial Coefficient

- 따라서 항의 총 개수는

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

Binomial Coefficient

- 동적계획식 알고리즘 설계전략

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

  - 2차원 배열 B를 만들고, 각 $B[i][j]$에는 $\left[\begin{array}{c} i \\ j \end{array}\right]$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							

$$\begin{tikzcd} B[i-1, j-1] \arrow[dr] & B[i-1, j] \arrow[d] \\ & B[i, j] \end{tikzcd}$$

Binomial Coefficient

- 동적계획 알고리즘
  - 문제: 이항계수를 계산한다.
    - 입력: 음수가 아닌 정수 n과 k, 여기서 $k \le n$
    - 출력: bin, $\left[\begin{matrix} n \\ k \end{matrix}\right]$
  - 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

- 동적계획 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
………………….
$i = k$-1일 때 j-루프 수행 횟수 : k
$i = k$일 때 j-루프 수행 횟수 : $k + 1$
$i = k$+1일 때 j-루프 수행 횟수 : $k + 1$
………………….
$i = n$일 때 j-루프 수행 횟수 : $k + 1$

따라서 총 수행횟수는:

$$1+2+3+\cdots+k+\underbrace{(k+1)+\cdots+(k+1)}_{}^{n-k+1 \text{ times}} = \frac{k(k+1)}{2} + (n-k+1)(k+1)$$

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

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 $\begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n \\ n-k \end{bmatrix}$

The Knapsack Problem

- Problem:

$S = \{item_1, item_2, \ldots, item_n\},$

$w_i = item_i$의 무게

$p_i = item_i$의 가치

$W =$ 배낭에 넣을 수 있는 최대 무게

라고 할 때, $\sum_{item_i \in A} w_i \leq W$ 를 만족하면서

$\sum_{item_i \in A} p_i$ 가 최대가 되도록

$A \subseteq S$가 되는 A를 결정하는 문제이다.

The Fractional Knapsack Problem

- 물건의 일부분을 잘라서 담을 수 있는 경우

- $W = 30$ lb

품목	무게	값	값어치
$item_1$	5 lb	\$50	\$10/lb
$item_2$	10 lb	\$60	\$6/lb
$item_3$	20 lb	\$140	\$7/lb

- 탐욕적인 접근방법으로 최적해를 구할 수 있다.
- 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다!!
- $item_1 + item_3 + (5/10) * item_2 = \$50 + \$140 + (5/10)*\$60$

  $\Rightarrow \$220 \quad (30 \text{ lb})$

- Optimal!

The 0-1 Knapsack Problem

- 물건의 일부분을 잘라서 담을 수 없는 경우

- The 0-1 Knapsack Problem (1)

  - 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다?
  - 항상 최적의 해를 주지는 않는다!
  - 왜 아닌지 보기: W = 30 lb

품목	무게	값	값어치
$item_1$	5 lb	$50	$10/lb
$item_2$	10 lb	$60	$6/lb
$item_3$	20 lb	$140	$7/lb

    - 탐욕적인 방법: $item_1 + item_3 \Rightarrow$ 25 lb $\Rightarrow$ $190
    - 최적인 해답: $item_2 + item_3 \Rightarrow$ 30 lb $\Rightarrow$ $200

The 0-1 Knapsack Problem

- The 0-1 Knapsack Problem (2)
  - 가장 비싼 물건부터 우선적으로 채운다?
  - 이 방법도 항상 최적의 해를 주는 건 아니다!
  - 왜 아닌지 보기: W = 30 lb

품목	무게	값
$item_1$	25 lb	$10
$item_2$	10 lb	$9
$item_3$	10 lb	$9

    - 탐욕적인 방법: $item_1 \Rightarrow$ 25 lb $\Rightarrow$ $10
    - 최적인 해답: $item_2 + item_3 \Rightarrow$ 20 lb $\Rightarrow$ $18

The 0-1 Knapsack Problem

- The 0-1 Knapsack Problem (3)

- 무작정 알고리즘
  - n개의 물건에 대해서 모든 부분 집합을 다 고려한다.
  - 크기가 n인 집합의 부분집합의 수는 2^n개이다!!
  - 따라서 적어도 $\Omega(2^n)$ 의 시간복잡도가 필요하다.

품목	무게	값
$item_1$	25 lb	\$10
$item_2$	10 lb	\$9
$item_3$	10 lb	\$9

The 0-1 Knapsack Problem

- Dynamic Programming Approach (0-1 Knapsack Problem)
  - $i > 0$ 이고 $w > 0$일 때, 전체 무게가 w가 넘지 않도록 i번째까지의 항목 중에서 얻어진 최고의 이익(optimal profit)을 $P[i][w]$라고 하면,

$$P[i][w] = \begin{cases} maximum(P[i-1][w], p_i + P[i-1][w-w_i]) & (\text{if } w_i \le w) \\ P[i-1][w] & (\text{if } w_i > w) \\ 0 & (\text{if } i = 0 \text{ or } w = 0) \end{cases}$$

여기서 $P[i-1][w]$는 i번째 항목을 포함시키지 않는 경우의 최고 이익이고,
$p_i + P[i-1][w - w_i]$는 i번째 항목을 포함시키는 경우의 최고 이익이다.
위의 재귀 관계식이 최적화 원칙을 만족하는지는 쉽게 알 수 있다.

  - 그러면 어떻게 최대 이익 $P[n][W]$값을 구할 수 있을까?
    - int $P[0..n][0..W]$의 2차원 배열을 만든 후, 각 항을 계산하여 넣는다
    - 여기서 $P[0][w] = 0$, $P[i][0] = 0$으로 놓으면 되므로, 계산해야 할 항목의 수는 $nW \in \Theta(nW)$

The 0-1 Knapsack Problem

- Refinement of Dynamic Programming
  - 여기서 n과 W와는 아무런 상관관계가 없다.
    만일 (임의적으로) $W = n!$이라고 한다면, 수행시간은 $\Theta(n \times n!)$이 된다.
    그렇게 되면 이 알고리즘은 앞에서 얘기한 무작정 알고리즘보다도 나을게 하나도 없다.

  - 그럼 이 알고리즘을 최악의 경우에 $\Theta(2^n)$ 시간에 수행될 수 있도록, 즉 무작정 알고리즘 보다 느리지 않고, 때로는 훨씬 빠르게 수행될 수 있도록 개량할 수 있을까?
    - 착안점은 $P[n][W]$를 계산하기 위해서 $(n\text{-}1)$번째 행을 모두 계산할 필요가 없다는데 있다.

The 0-1 Knapsack Problem

- $P[n][W]$는 아래 식으로 표현할 수 있다

$$P[n][W] = \begin{cases} maximum(P[n-1][W], p_n + P[n-1][W - w_n]) & (\text{if } w_n \le W) \\ P[n-1][W] & (\text{if } w_n > W) \end{cases}$$

  - 따라서 (n-1)번째 행에서는 $P[n\text{-}1][W]$와 $P[n\text{-}1][W\text{-}w_n]$ 항만 필요
  - i-번째 행에 어떤 항목이 필요한지를 결정한 후에,
    다시 (i-1)번째 행에 필요한 항목을 결정
    - $P[\mathrm{i}][\mathrm{w}]$는 $P[i\text{-}1][w]$와 $P[i\text{-}1][w\text{-}w_i]$로 계산
  - 이런 식으로 $n = 1$이나 $w \le 0$일 때까지 계속해 나가면 된다.

The 0-1 Knapsack Problem

- Ex 4.7
  - W=30 lb

품목	무게	값
$item_1$	5 lb	$50
$item_2$	10 lb	$60
$item_3$	20 lb	$140

  - We need P[3][W] = P[3][30]
    - To compute P[3][30] -- max(P[3-1][30], p_3 + P[3-1][30-w_3])
      = max(P[2][30], p_3 + P[2][10])
    - To compute P[2][30] -- max(P[2-1][30], p_2 + P[2-1][30-w_2])
      = max(P[1][30], p_2 + P[1][20])
    - To compute P[2][10] -- max(P[2-1][10], p_2 + P[2-1][10-w_2])
      = max(P[1][10], p_2 + P[1][0])

The 0-1 Knapsack Problem

- Compute row 1
  - $P[1][w] = \begin{cases} \max(P[0][w], \$50+P[0][w-5]) & (\text{if } w_1 = 5 \leq w) \\ P[0][w] & (\text{if } w_1 = 5 > w) \end{cases}$

    $= \begin{cases} \$50 & (\text{if } w_1 = 5 \leq w) \\ \$0 & (\text{if } w_1 = 5 > w) \end{cases}$
  - Therefore

    P[1][0] = \$0; P[1][10] = \$50; P[1][20] = \$50; P[1][30] = \$50

- Compute row 2
  - $P[2][10] = \begin{cases} \max(P[1][10], \$60+P[1][0]) & (\text{if } w_2 = 10 \leq 10) \\ P[1][10] & (\text{if } w_2 = 10 > 10) \end{cases}$

    $= \$60$
  - $P[2][30] = \begin{cases} \max(P[1][30], \$60+P[1][20]) & (\text{if } w_2 = 10 \leq 30) \\ P[1][30] & (\text{if } w_2 = 10 > 30) \end{cases}$

    $= \$60 + \$30 = \$110$

The 0-1 Knapsack Problem

- Compute row 3
  - $P[3][30] = \begin{cases} \max(P[2][30], \$140+P[2][10]) & (\text{if } w_3 = 20 \leq 30) \\ P[2][10] & (\text{if } w_3 = 20 > 30) \end{cases}$

    $= \$140 + \$60 = \$200$

- The modified algorithm – compute only 7 entries
- The original algorithm - compute $3 \times 30 = 90$ entries

The 0-1 Knapsack Problem

- Efficiency in the worst case
  - Compute at most 2^i entries in the $(n - i)$-th row
    - The total number is $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$.
    - 최악의 경우 수행시간 $\Theta(2^n)$
  - The number of entries computed is in $O(nW)$
    - What about the number of the modified algorithm ?
    - If $n = W+1$, and $w_i = 1$ for all i,
      then the total number of entries is about

      $$1 + 2 + 3 + \ldots + n = n(n+1) / 2 = (W+1)(n+1) / 2$$
    - For arbitrary large values of n and W, $\Theta(nW)$
  - Combining these 2 results, the worst case is in $O(min(2^n, nW))$

The 0-1 Knapsack Problem

- 아직 아무도 최악의 경우 수행시간이 지수(exponential)보다 나은 알고리즘을 발견하지 못했고,
- 아직 아무도 그러한 알고리즘은 없다라고 증명한 사람도 없다.

  -> **NP문제**

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:
  - $A_1 \times A_2 \times A_3$을 구하라.
  - A_1: 10×100, A_2; 100×5, A_3: 5×50
  - $(A_1 \times A_2) \times A_3$: 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

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

$A(B(CD))$	$30 \times 12 \times 8 +$	$2 \times 30 \times 8 +$	$20 \times 2 \times 8 =$	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

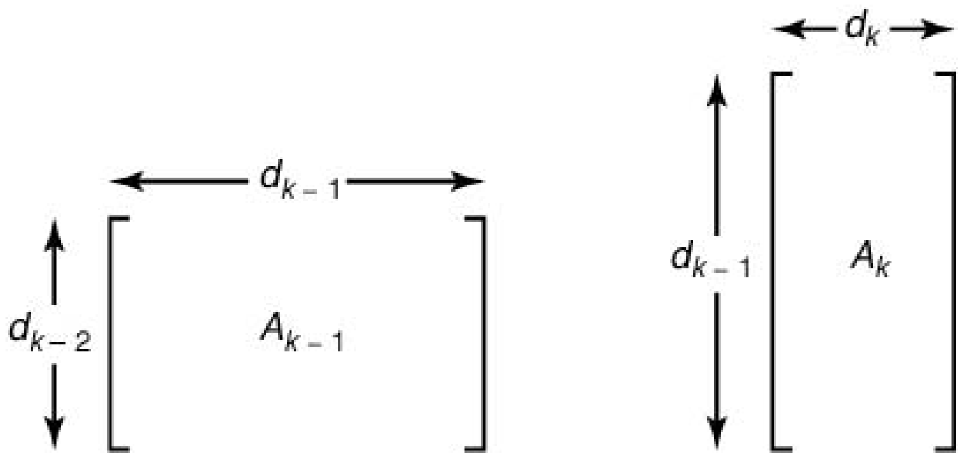
Chained Matrix Multiplication

- 무작정알고리즘: 가능한 모든 순서를 모두 고려해 보고, 그 가운데에서 가장 최소를 택한다.

- 시간복잡도 분석: 최소한 지수(exponential-time) 시간

- 증명:
  - n개의 행렬($A_1, A_2, \ldots, A_n$)을 곱할 수 있는 모든 순서의 가지 수를 t_n이라고 하자.
  - 만약 A_1이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_2, \ldots, A_n$을 곱하는 데는 t_{n-1}개의 가지수가 있을 것이다.
  - A_n이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_1, \ldots, A_{n-1}$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 $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$

Chained Matrix Multiplication



- d_k를 행렬 A_k의 열(column)의 수라고 하자
  - 자연히 A_k의 행(row)의 수는 d_{k-1}; A_1의 행의 수는 d_0라고 하자.
  - For $1 \leq i \leq j \leq n$, let

    M[i][j] = i < j일 때 A_i부터 A_j까지의 행렬을 곱하는데 필요한 곱셈의 최소 횟수

$$= minimum_{i \leq k \leq j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j)$$

M[i][j] = 0 if i = j

Chained Matrix Multiplication

- Ex 3.5

A_1	A_2	A_3	A_4	A_5	A_6
5×2	2×3	3×4	4×6	6×7	7×8

$$A_4(A_5A_6) \qquad\qquad (A_4A_5)A_6$$

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

$M[i][j]$	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 $1 \le i \le 6$
  - 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: $M[1][3] = \min(M[1][k] + M[k+1][3] + d_0 d_k d_3)$

    $= \min(M[1][1] + M[2][3] + d_0 d_1 d_3,$

    $M[1][2] + M[3][3] + d_0 d_2 d_3)$

    $= \min(0+24+5x2x4, 30+0+5x3x4) = 64$

    Compute M[2][4], M[3][5], M[4][6]
  - For diagonal 3: $M[1][4] = \min(M[1][k] + M[k+1][4] + d_0 d_k d_4)$

    $= \min(M[1][1] + M[2][4] + d_0 d_1 d_4,$

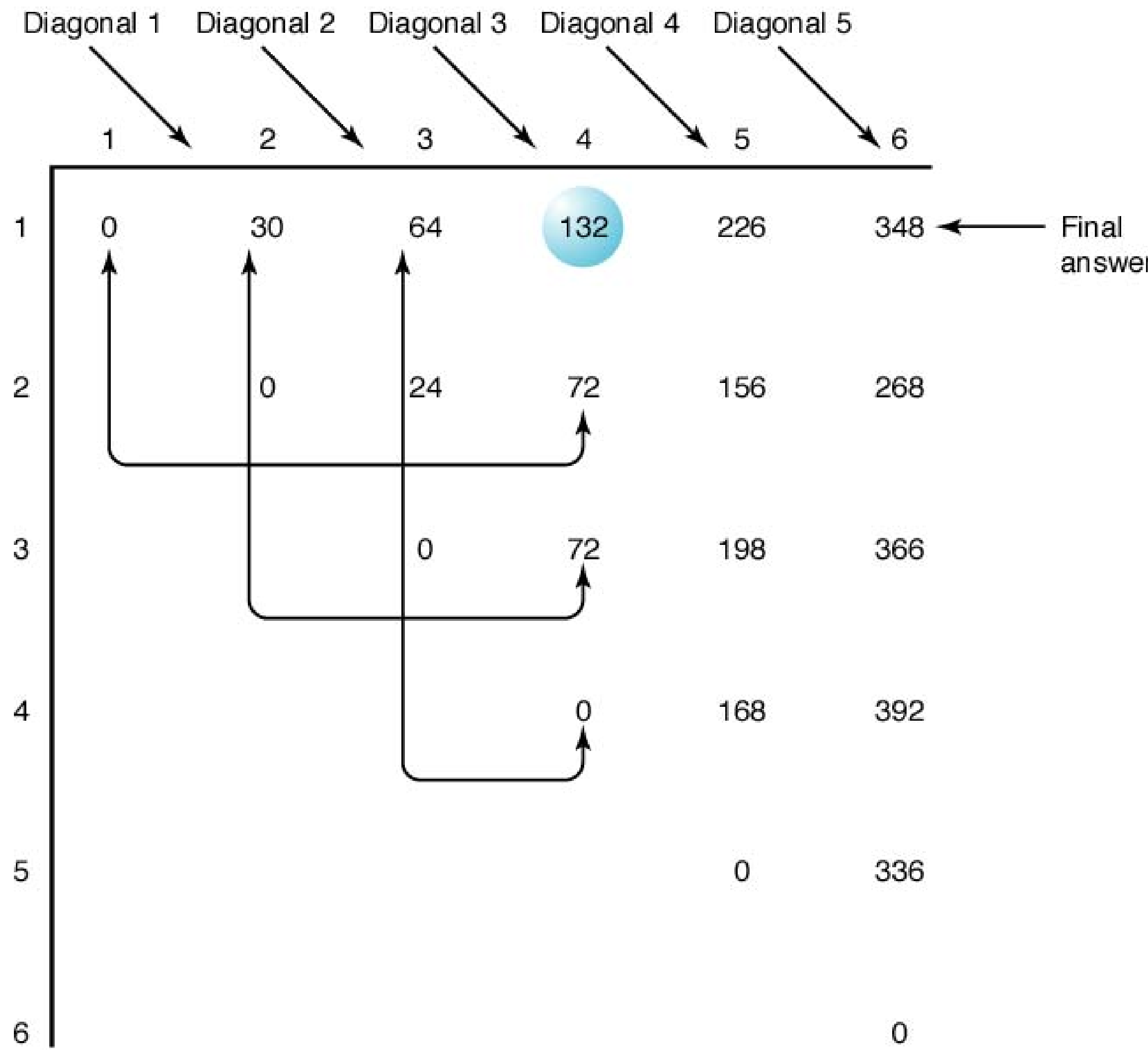
    $M[1][2] + M[3][4] + d_0 d_2 d_4,$

    $M[1][3] + M[4][4] + d_0 d_3 d_4)$

    $= \min(0+72+5x2x6, 30+72+5x3x6, 64+0+5x4x6) = 132$

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

Chained Matrix Multiplication



Chained Matrix Multiplication

- 최소곱셈(Minimum Multiplication) 알고리즘
  - 문제
    - n개의 행렬을 곱하는데 필요한 기본적인 곱셈의 횟수의 최소치를 결정하고, 그 최소치를 구하는 순서를 결정하라.
  - 입력
    - 행렬의 수 n와 배열 $d[0..n]$,
      $d[i-1]\times d[i]$는 i번째 행렬의 규모를 나타낸다.
  - 출력
    - 기본적인 곱셈의 횟수의 최소치를 나타내는 *minmult*;
      최적의 순서를 얻을 수 있는 배열 P,
      여기서 $P[i][j]$는 행렬 i부터 j까지가 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

- 최소곱셈 알고리즘의 모든 경우 분석
  - 단위연산: 각 k값에 대하여 실행된 명령문 (instruction), 여기서 최소값인 지를 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 k값을 $P[i][j]$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: n개의 행렬을 곱하는 최적의 순서를 출력하시오
    - 입력: n과 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
  - order(i,j)의 의미: $A_i \times \ldots \times A_j$ 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)

동적계획법에 의한 설계 절차

- <u>최적의 원칙**(the principle of optimality)**</u> 확인!!

입력에 대한 최적의 해가 그 입력을 나누어 쪼갠 여러 부분에 대한 최적의 해로부터 구할 수 있는가?

=> **재귀 관계식(recursive property)을 구한다.**

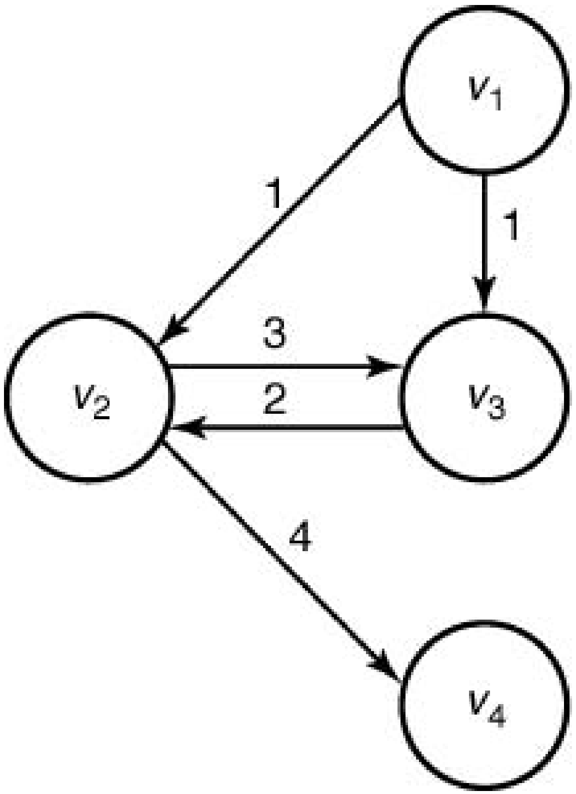
- 작은 입력의 경우부터 큰 입력으로 상향적으로 최적의 해답을 계산

- 상향적으로 최적의 해답을 구축

최적의 원칙

- 최단경로를 구하는 문제에서,
  v_k를 v_i에서 v_j로 가는 최적 경로 상의 정점이라고 하면,
  v_i에서 v_k 로 가는 부분경로와 v_k에서 v_j로 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



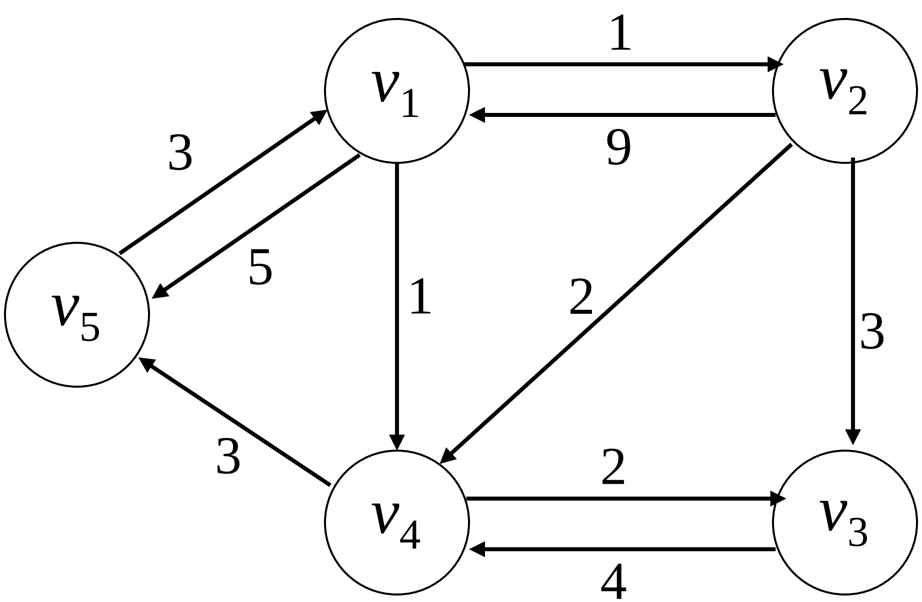
- v_1에서 v_4로의 최장경로는 $[v_1, v_3, v_2, v_4]$가 된다.
- 그러나 이 경로의 부분 경로인 v_1에서 v_3으로의 최장경로는 $[v_1, v_3]$이 아니고, $[v_1, v_2, v_3]$이다.
- 따라서 최적의 원칙이 적용되지 않는다.
- 주의: 여기서는 단순경로(simple path), 즉 순환(cycle)이 없는 경로만 고려한다.

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 n개의 정점을 가지고 있고, 모든 정점들 사이에 이음선이 존재한다고 가정하자.
    - 그러면 한 정점 v_i에서 어떤 정점 v_j로 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_i에서 출발하여 처음에 도착할 수 있는 정점의 가지 수는 n-2개 이고, 그 중에 하나를 선택하면, 그 다음에 도착할 수 있는 정점의 가지 수는 n-3개 이고, 이렇게 계속하여 계산해 보면, 총 경로의 개수는 $(n-2)(n-3)\ldots1 = (n-2)!$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: $0 \le k \le n$인, $D^{(k)}$

    $D^{(k)}[i][j] = \{v_1, v_2, \ldots, v_k\}$의 정점들 만을 통해서 v_i에서 v_j로 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, $0 \le k \le 5$ 일 때, $D^{(k)}[2][5]$를 구해보자. 예제3.2

- $D^{(0)} = W$이고, $D^{(n)} = D$임은 분명하다. 따라서 D를 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

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

경우 1: $\{v_1, v_2, \ldots, v_k\}$의 정점들 만을 통해서 v_i에서 v_j로 가는 최단경로가 <u>v_k를 거치지 않는</u> 경우.

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

경우 2: $\{v_1, v_2, \ldots, v_k\}$의 정점들 만을 통해서 v_i에서 v_j로 가는 최단경로가 <u>v_k를 거치는</u> 경우.

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

보기: $D^{(2)}[5][4]$

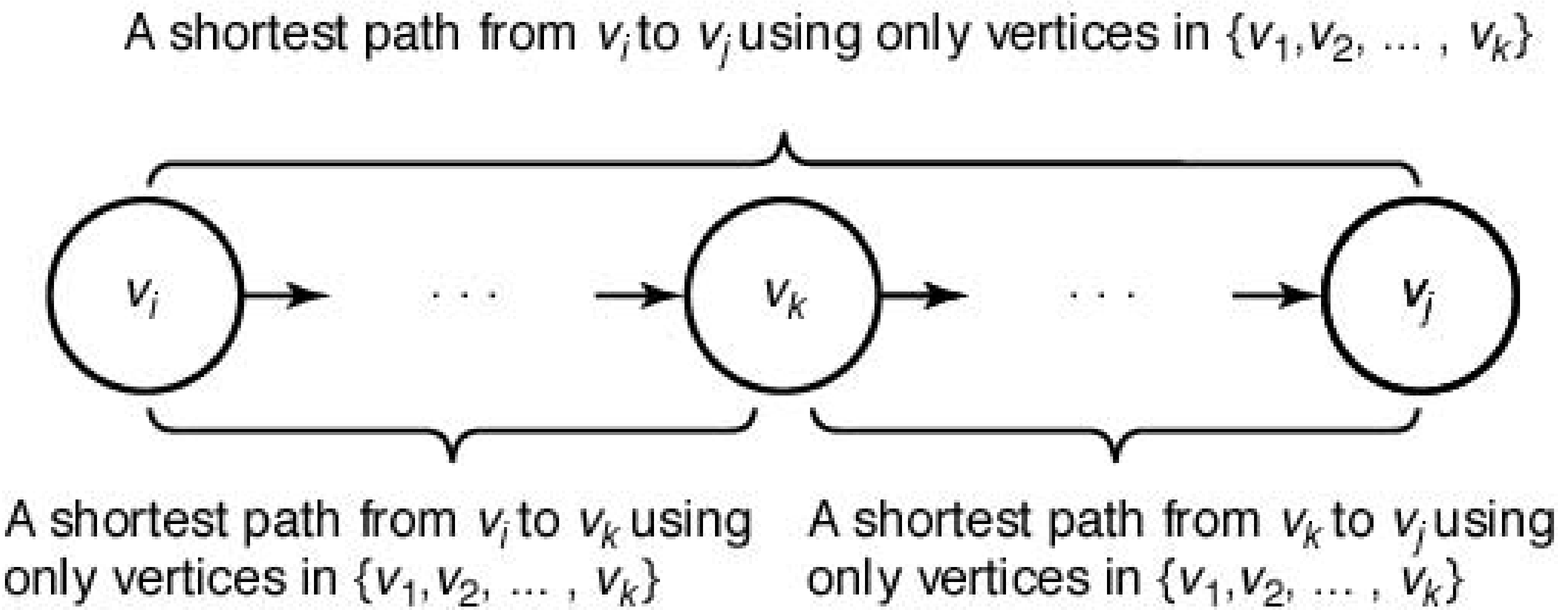
2. 상향식으로 $k = 1$부터 n까지 다음과 같이 이 과정을 반복하여 해를 구한다.

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

동적계획식 설계절차

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

경우1 경우2



Floyd's Algorithm I

- 문제
  - 가중치 포함 그래프의 각 정점에서 다른 모든 정점까지의 최단거리를 계산하라.
  - 입력
    - 가중치 포함, 방향성 그래프 W와 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 W와 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 D와 P를 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
```

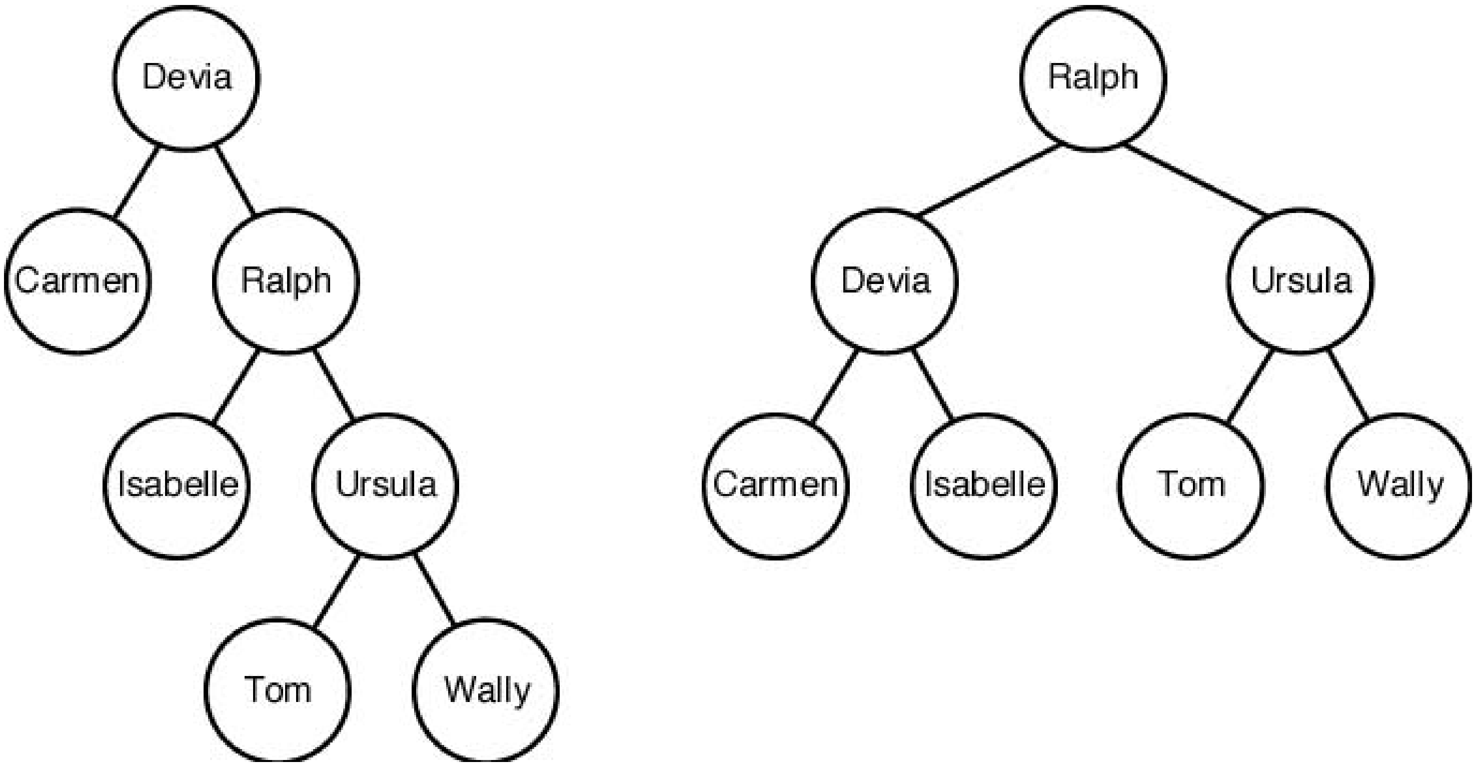
결과: v1 v4.

즉, v_5에서 v_3으로 가는 최단경로는 v_5, v_1, v_4, v_3,이다.

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

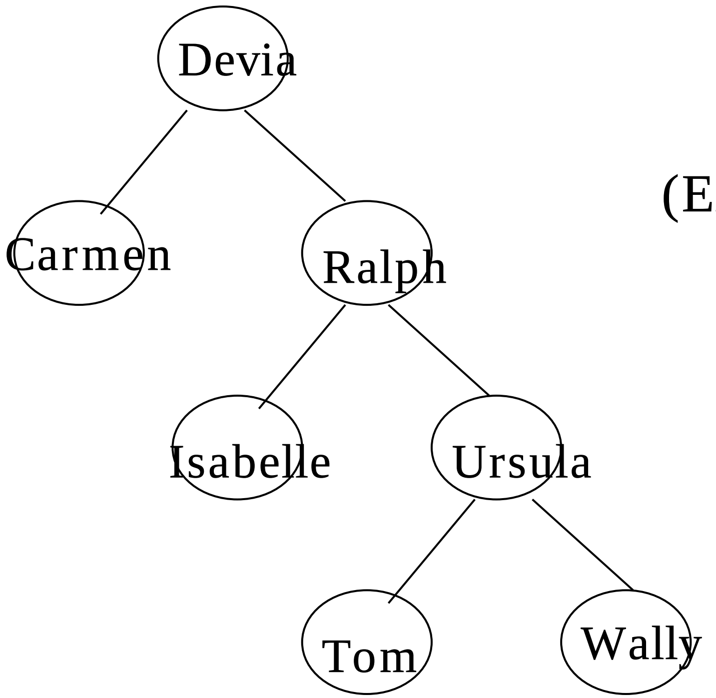
- Problem: determine the node containing a key in a binary search tree (assume that key is in the tree)
  - Inputs: a pointer tree & a key keyin
  - Outputs: a pointer p to the node containing the key

```
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

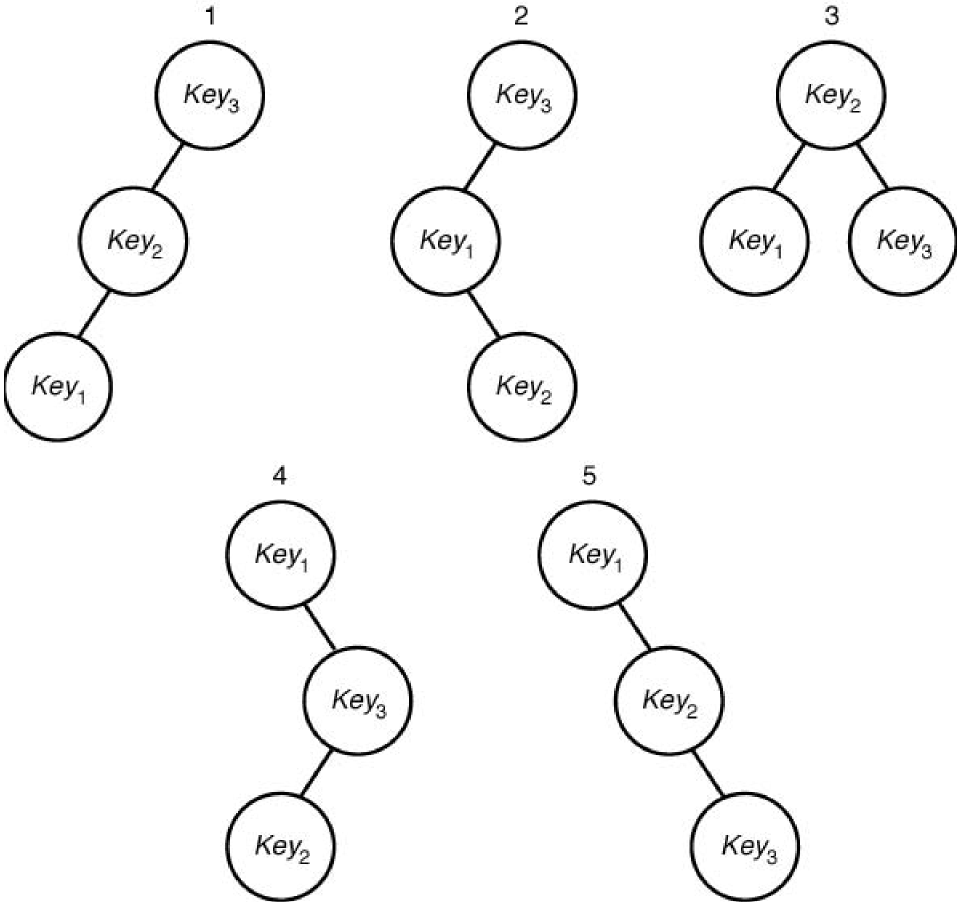
- Analysis
  - Search time – the number of comparisons to locate a key
    - Search time a given key depth(key) + 1

      where depth(key) is the depth of the node containing the key



(Ex) Search time for "Ursula"
depth(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) $2(0.7) + 1(0.2) + 2(0.1) = 1.8$

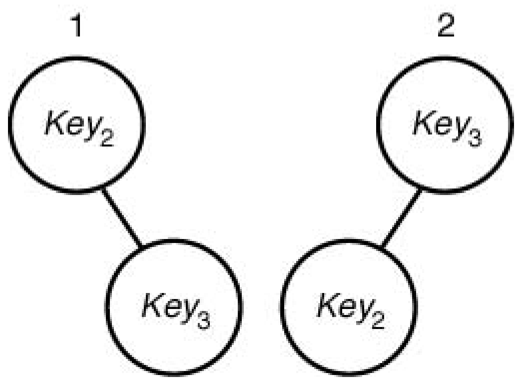
4) $1(0.7) + 3(0.2) + 2(0.1) = 1.5$

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

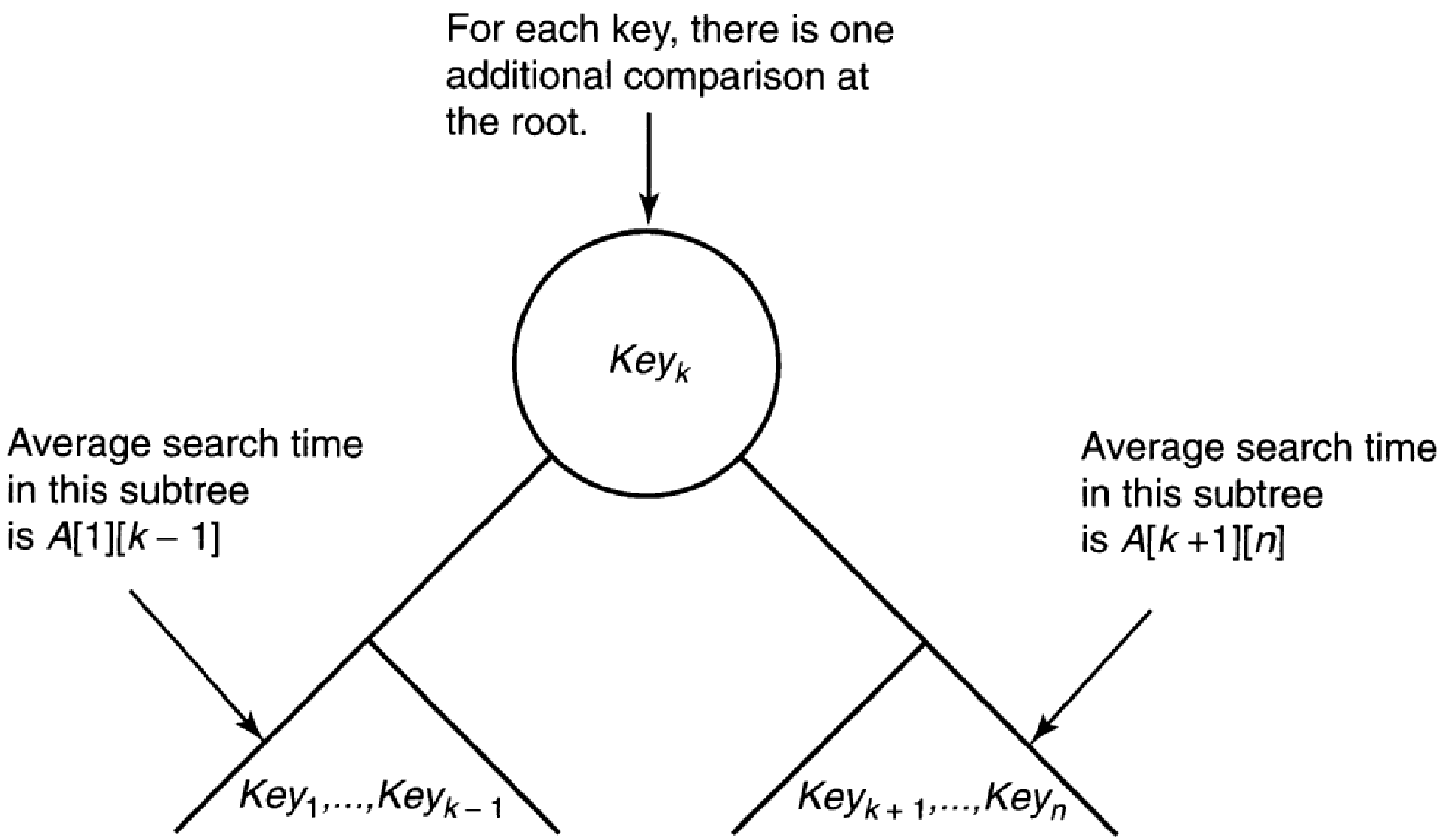
Optimal Binary Search Trees

- 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, $A[1][n]$ =

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

Average time in left subtree	Additional time comparing at root	Average time searching at root	Average time in right subtree	Additional time comparing at root

Optimal Binary Search Trees

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

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

$$A[1][n] = \underset{1 \le k \le n}{\text{minimun}}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

$$A[i][j] = \underset{i \le k \le j}{\text{minimun}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad (i < j)$$

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

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

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]) + Σ_{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

```
node_pointer 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]

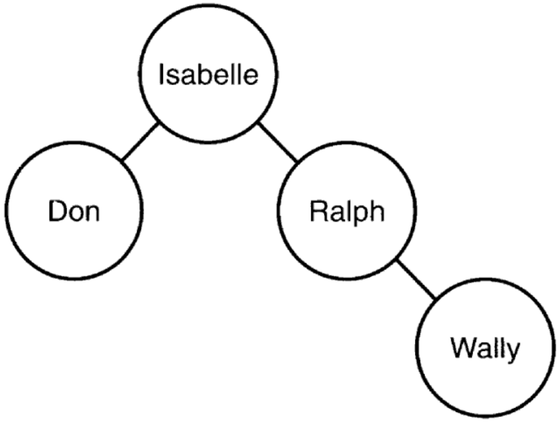
    $p_1=3/8$ $p_2=3/8$ $p_3=1/8$ $p_4=1/8$

	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

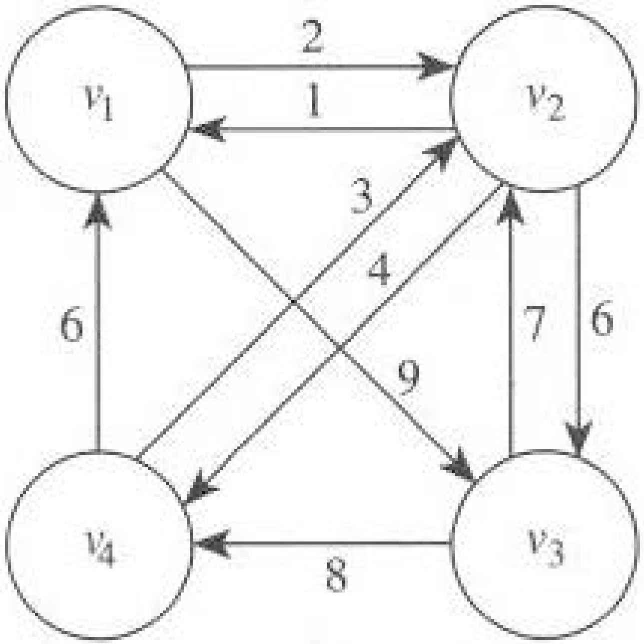
	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R



The Traveling Salesperson Problem

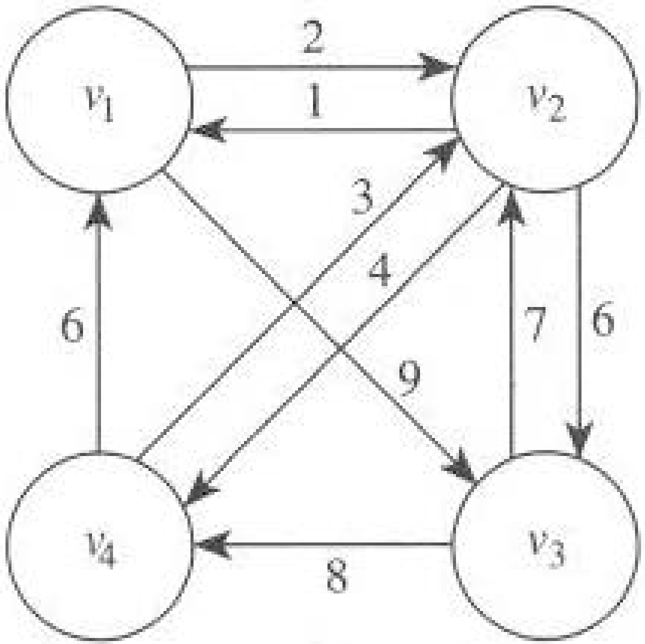
- Problem Definition
  - Determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city



$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$
$$(n-1)(n-2)\cdots 1 = (n-1)!$$

The Traveling Salesperson Problem

- Adjacent matrix W



$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 2 & 9 & \infty \\
2 & 1 & 0 & 6 & 4 \\
3 & \infty & 7 & 0 & 8 \\
4 & 6 & 3 & \infty & 0
\end{array}
$$

- 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 $D[v_2][A] = length[v_2, v_3, v_1] = \infty$
- If $A = \{v_3, v_4\}$,

  then $D[v_2][A] = \min(length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1])$

  $= \min(20, \infty) = 20$

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

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

$$D[v_i][\emptyset] = W[i][1].$$

The Traveling Salesperson Problem

- Ex 3.11 Determine an optimal tour in Fig 3.17
  - $D[v_2][\varnothing] = 1$; $D[v_3][\varnothing] = \infty$; $D[v_4][\varnothing] = 6$
  - $D[v_3][\{v_2\}] = \min_{j:v_j \in \{v_2\}} (W[3][j] + D[v_j][\{v_2\}-\{v_j\}])$

    $= W[3][2] + D[v_2][\varnothing] = 7 + 1 = 8$

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

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

    $D[v_2][\{v_4\}] = 4+6 = 10$; $D[v_3][\{v_4\}] = 8+6 = 14$;
  - $D[v_4][\{v_2, v_3\}] = \min_{j:v_j \in \{v_2, v_3\}} (W[4][j] + D[v_j][\{v_2, v_3\}-\{v_j\}])$

    $= \min(W[4][2]+D[v_2][\{v_3\}], W[4][3]+D[v_3][\{v_2\}])$

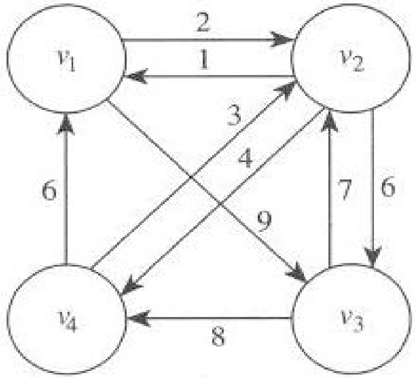
    $= \min(3+\infty , \infty+8) = \infty$

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

    $D[v_2][\{v_3, v_4\}] = \min(6+14, 4+\infty) = 20$
  - $D[v_1][\{v_2,v_3 ,v_4\}] = \min_{j:v_j \in \{v_2,v_3,v_4\}}(W[1][j] + D[v_j][\{v_2,v_3,v_4\}-\{v_j\}])$

    $= \min(W[1][2]+D[v_2][\{v_3,v_4\}], W[1][3]+D[v_3][\{v_2,v_4\}], W[1][4]+D[v_4][\{v_2,v_3\}])$

    $= \min(2+20, 9+12, \infty+\infty) = 21$



	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-{v1}];

    for(i=2; i<=n; i++)
        D[i][∅] = W[i][1];

    for(k=1; k<= n-2; k++)
        for(all subsets A⊆V-{v1} containing k vertices)
            for(i such that i≠1 and vi is not in A){
                D[i][A] = min(j:vj∈A)(W[i][j]+D[j][A-{vj}]);
                P[i][A] = value of j that gave the minimum;
            }

    D[1][V-{v1}] = min(2≤j≤n)(W[1][j]+D[j][V-{v1,vj}]);
    P[1][V-{v1}] = value of j that gave the minimum;
    minlength = D[1][V-{v1}]
}
```

The Traveling Salesperson Problem

- Theorem 3.1

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

  (Prove it!!)

  (Hint) For all n ≥ 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

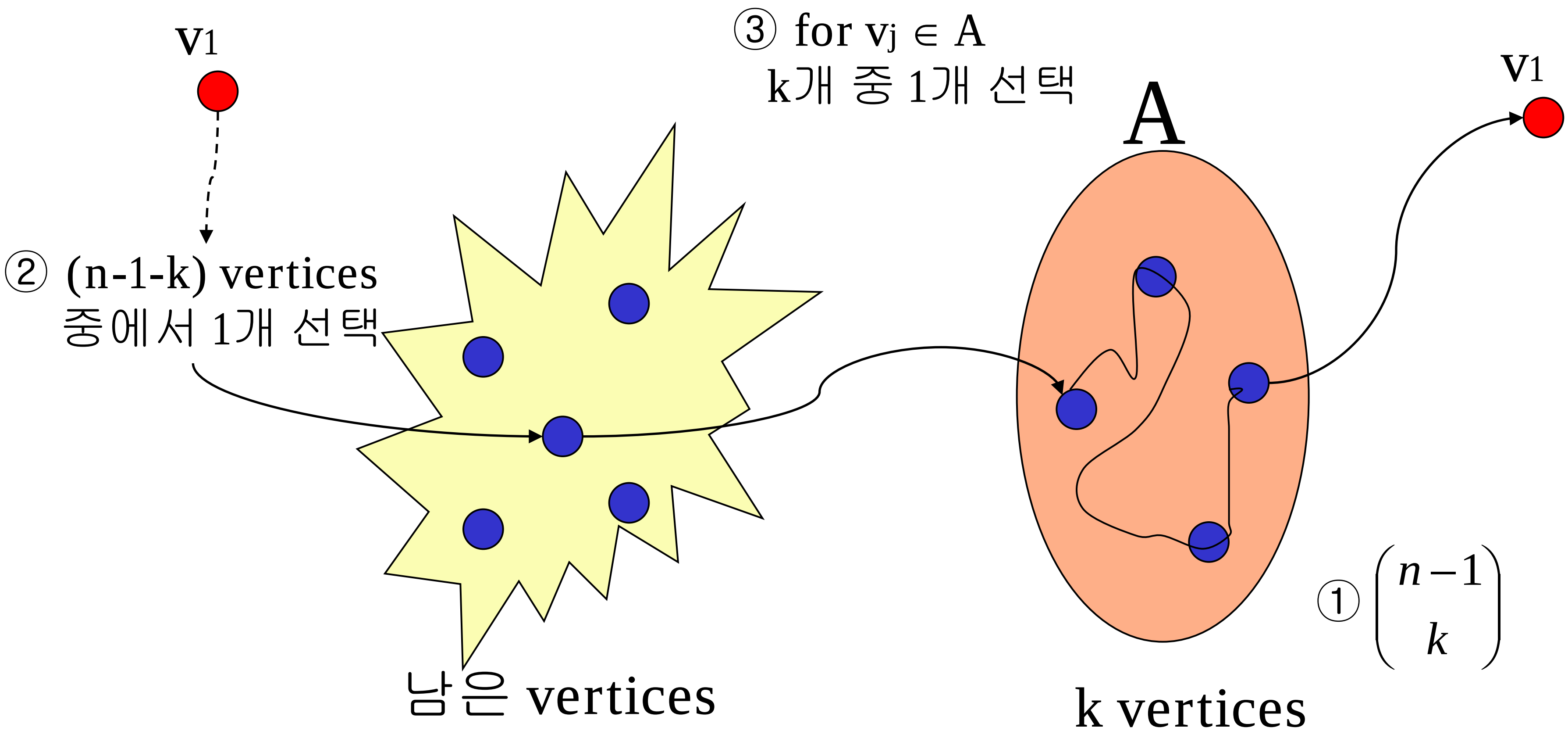
$$T(n) = \sum_{k=1}^{n-2} (n-1-k) \cdot k \cdot \binom{n-1}{k}$$

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

$$= (n-1) \cdot (n-2) \cdot 2^{n-3} \in \Theta(n^2 2^n)$$

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 ~ 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! usecs = 3857 yrs
    - Dynamic programming:

      $(20\text{-}1)\ (20\text{-}2)\ 2^{20-3}$ usecs = 45 secs

    - $M(n) = 20\ 2^{20}$ = 20,971,529 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.