Chapter 3. 동적계획
Dynamic Programming

Divide and Conquer에서 중복하여
계산하는 부분을 개선하기 위해
Bottom-up 방식으로 문제를 해결한다.

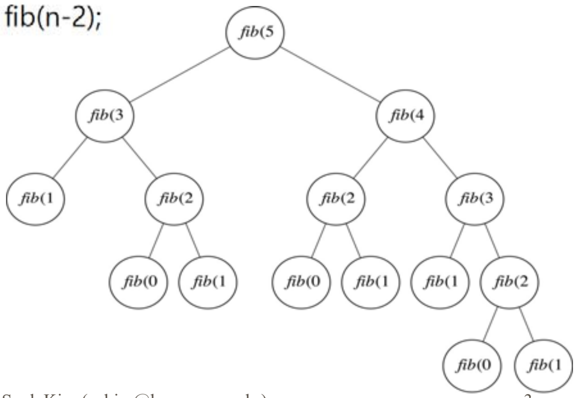
필요한 항목들을 미리 계산하여 두고 이를
활용하여 다음단계의 값을 계산한다.

- General Approach
  - divide a problem instance into small instances
  - **solve small instances first**
  - **store the results**, and later
  - **look it up instead of recomputing it**
- Development Steps
  1. Establish a recursive property
  2, Solve in a bottom-up fashion
- Examples
  - ~~binomial coefficient~~
  - shortest path (Floyd Algorithm)
  - chained matrix multiplication
  - optimal binary search tree
  - traveling salesperson problem

Fibonacci Number Revisited

- Divide and conquer algorithm

```
int fib(int n)  {          // 알고리즘 1.6
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

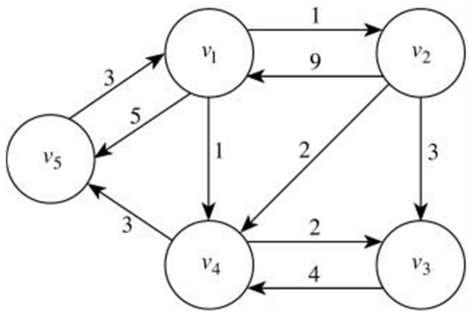


- Dynamic programming algorithm

```
int fib2(int n)     {          // 알고리즘 1.7
    index i; int f[0..n];
    f[0] = 0;
    if (n > 0) {
        f[1] = 1;
        for (i=2; i <= n; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

Graph

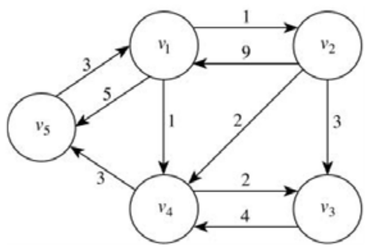
- Terminology
  - G = <V,E> where V is a set of vertices and E is a set of edges
  - directed graph (digraph)
  - weighted graph
  - path, simple path, length
  - cycle, cyclic/acyclic graph
  - example) Figure 3.2



- Shortest Paths Problem
  - find shortest simple paths on weighted graphs for **all pair of vertices**

  ⇨ optimization problem
  - Floyd's Algorithm (a dynamic programming algorithm)

  (Single source shortest path : Dijkstra's algorithm)



Representation of Graph

✓ Adjacency Matrix vs Adjacency List

	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

Adjacency Matrix

	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

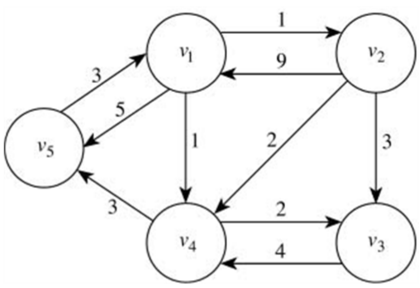
D

Distance

Floyd's Algorithm

Let $V^{(k)} = \{v_1, v_2, \ldots, v_k\}$

Let $D^{(k)}[i][j]$ be the length of the shortest path $<v_i, v_j>$ via $V^{(k)}$ (p.103 Ex. 3.2)

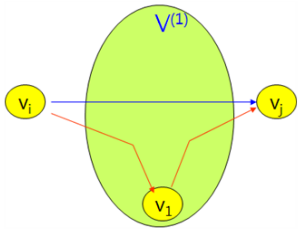


W	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D^{(0)} = W$

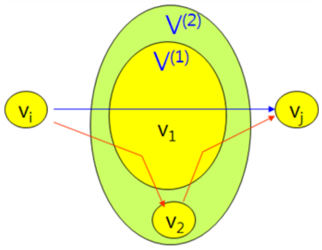
$D^{(1)}$ Dij = min(Dij, Di1+D1j)

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	14
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	4	∞	4	0



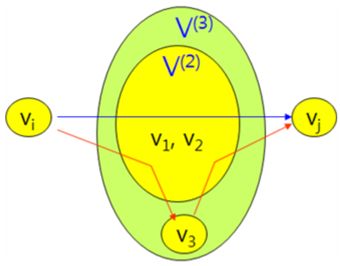
$D^{(2)}$ Dij = min(Dij, Di2+D2j)

	1	2	3	4	5
1	0	1	4	1	5
2	9	0	3	2	14
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	4	7	4	0



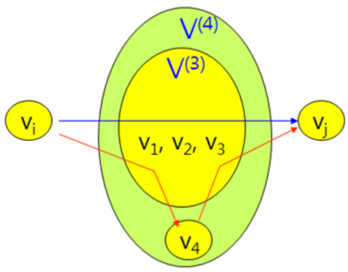
$D^{(3)}$ Dij = min(Dij, Di3+D3j)

	1	2	3	4	5
1	0	1	4	1	5
2	9	0	3	2	14
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	4	7	4	0



$D^{(4)}$ Dij = min(Dij, Di4+D4j)

	1	2	3	4	5
1	0	1	3	1	4
2	9	0	3	2	5
3	∞	∞	0	4	7
4	∞	∞	2	0	3
5	3	4	6	4	0

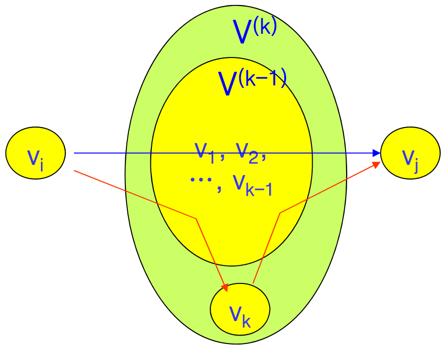


Shortest path $<v_i,v_j>$ via $V^{(k)}$: the shorter one of

(1) the shortest path $<v_i,v_j>$ via $V^{(k-1)}$ and

(2) the shortest path $<v_i,v_k>$ via $V^{(k-1)}$

+ the shortest path $<v_k,v_j>$ via $V^{(k-1)}$



작은 문제의 답 중복 사용 검토

$D^{(1)}$ Dij = min(Dij, Di1+D1j) $D^{(2)}$ Dij = min(Dij, Di2+D2j)

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	14
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	4	∞	4	0

	1	2	3	4	5
1	0	1	4	1	5
2	9	0	3	2	14
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	4	7	4	0

$D^{(1)}[2][3]$은 3번 활용됨

$D^{(2)}[1][3] = \min(D^{(1)}[1][3], D^{(1)}[1][2] + D^{(1)}[2][3])$

$D^{(2)}[4][3] = \min(D^{(1)}[4][3], D^{(1)}[4][2] + D^{(1)}[2][3])$

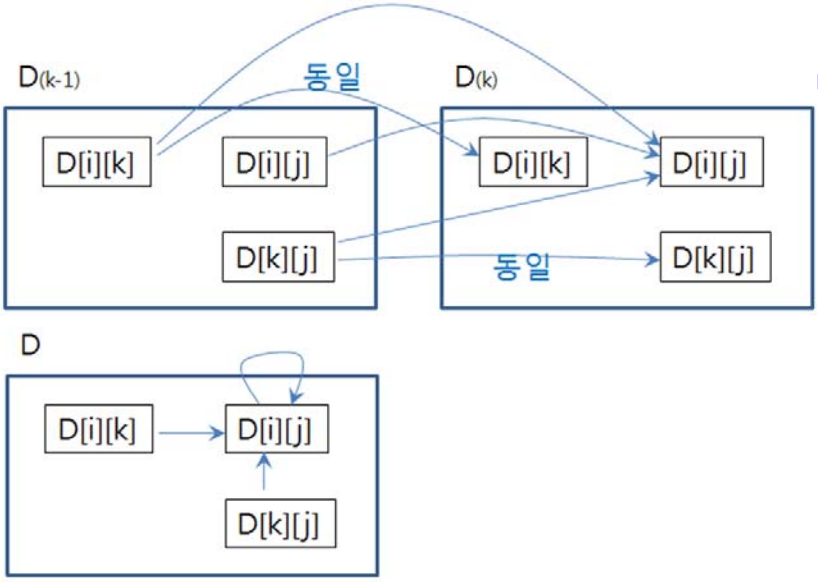
$D^{(2)}[5][3] = \min(D^{(1)}[5][3], D^{(1)}[5][2] + D^{(1)}[2][3])$

Floyd's Algorithm

- ✓ $D^{(k)}$ can be computed from $D^{(k-1)}$
  $D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j])$
- ✓ Compute $D^{(k)}[i][j]$ from k=0 to k=n (Ex. 3.3)
- ✓ Algorithm 3.3 ***floyd***

```
void floyd(int n, number W[][], number D[][])
{
  index i, j, k;
  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])
           D[i][j] = D[i][k]+D[k][j];
}
```


- ● Time Complexity (Every-Case)
- basic operation : the most inner loop
- $T(n) = n \times n \times n \in \Theta(n^3)$
- ● Space Complexity (Every-Case)
- strait-forward method : n of D arrays $\Rightarrow \Theta(n^3)$
- can use only one D array (no additional memory)
  since in the *k*th iteration
  (1) D[i][k] and D[k][j] are not changed
  Since D[k][k] = 0,
  min(D[i][k], D[i][k]+D[k][k]) = D[i][k]
  min(D[k][j], D[k][k]+D[k][j]) = D[k][j]
  (2) D[i][j] is computed from its own value
  and D[i][k] and D[k][j]



D[i][j] = min(D[i][j], D[i][k]+D[k][j])

굳이 분할정복 알고리즘을 만든다면?

```
number W[1..n][1..n], number D[1..n][1..n];

AllPaths(int n)
{
    for (i=1; i <= n; i++)
      for (j=1; j <= n; j++)
        D[i][j] = shortest(n, i, j);
}
shortest(int k, int i, int j)
{
    if (k == 0)
      return W[i][j];
    else
      return min( shortest(k-1, i, j),
            shortest(k-1, i, k) + shortest(k-1, k, j) );
}
```

Shortest Path Output

- Save the Shortest Path
  - $P[i][j]$: the highest index of an intermediate vertex on the shortest path $<v_i, v_j>$
    0 if no intermediate vertex exists
  - Algorithm 3.4 ***floyd2***
  - $T(n) \in \Theta(n^3)$


```
void floyd2(int n, 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];
        }
}
```

Print the Shortest Path

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

- path의 중간 vertex 개수가 k일 때
  path 호출 때마다 vertex 한개는 출력되므로
  총 호출 회수는 k
  따라서 T(k) = k

⇒ 임의의 vertex 간에 k ≤ n 이므로
  $T(n) \in O(n)$

For Optimization Problems

✓Development of Dynamic Programming Algorithm

1. Establish a recursive property
2. Compute the value of an optimal solution in a bottom-up fashion
3. Construct an optimal solution in a bottom-up fashion

Note) Dynamic Programming can be used only if the *principle of optimality* is applied

✓**Principle of Optimality**

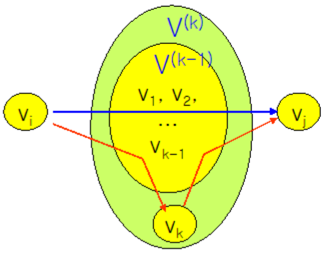
- ✓an optimal solution to an instance of a problem always contains optimal solutions to all sub-instances

Example of Principle of Optimality

Example) If an optimal path <v_i,v_j> is <v_i,v_k>+<v_k,v_j> then <v_i,v_k> and <v_k,v_j> also optimal paths

In Floyd's algorithm

D[i][j] = min(D[i][j], **D[i][k]+D[k][j]**)

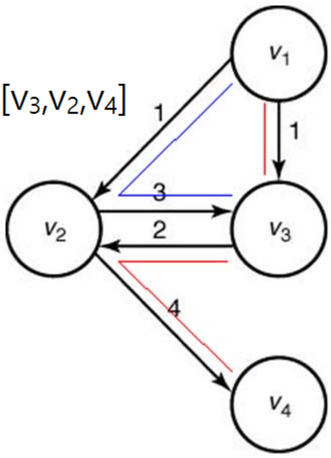


Counter Example of Principle of Optimality

Longest Path Problem (Fig. 3.6)

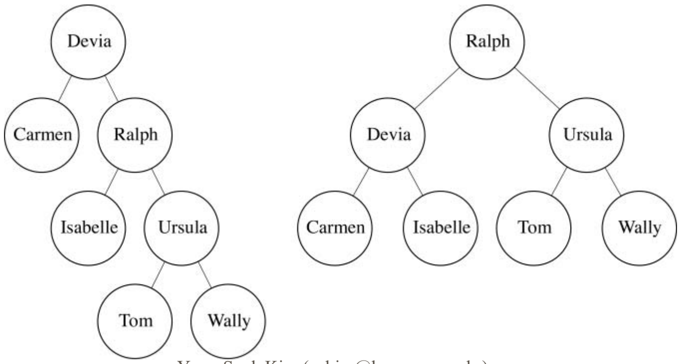
<v_1,v_4> = [v_1,**v_3**,v_2,v_4]

<v_1,v_3>+<v_3,v_4> = [v_1,v_2,v_3] + [v_3,v_2,v_4]



Optimal Binary Search Tree Problem

- ✓Binary Search Tree
  - ✓1. each node has one key
  - ✓2. the keys in the left sub-tree ≦ the key
  - ✓3. the keys in the right sub-tree ≧ the key



Binary Search Tree

- ✓Terminology
  - ✓***depth*** (level) of a node: # of edges from the root
  - ✓***balanced*** tree: for each node

    diff(depth(left subtree), depth(right subtree)) ≦ 1
  - ✓***optimal*** search tree: minimize the average time to locate keys

    Note) probabilities of keys are different
- ✓Data Structure of a Tree Node

```
struct nodetype {
    keytype key;
    nodetype *left, *right;
}
```

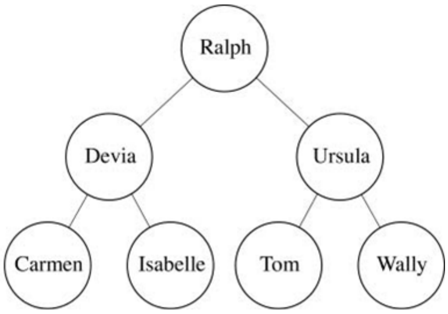
Search Time

search time (# of comparisons) = depth(key)+1

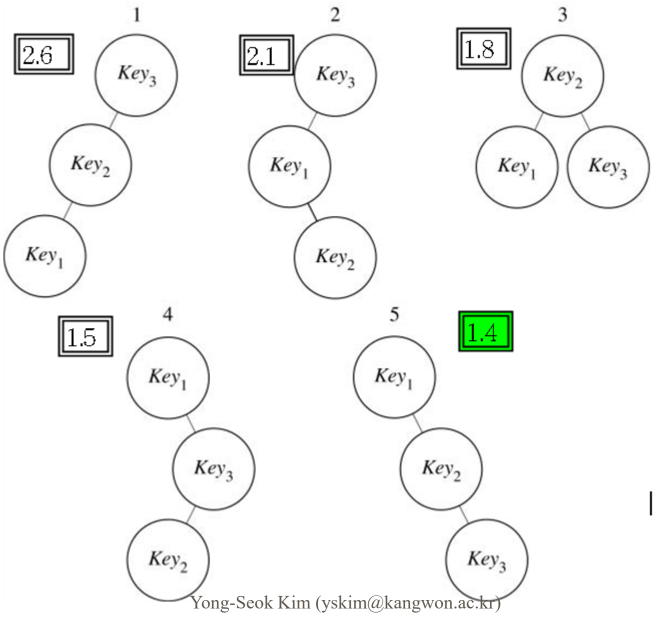
average search time = $\sum_{i=1}^{n}(d_i+1)p_i$

d_i is the depth of node i

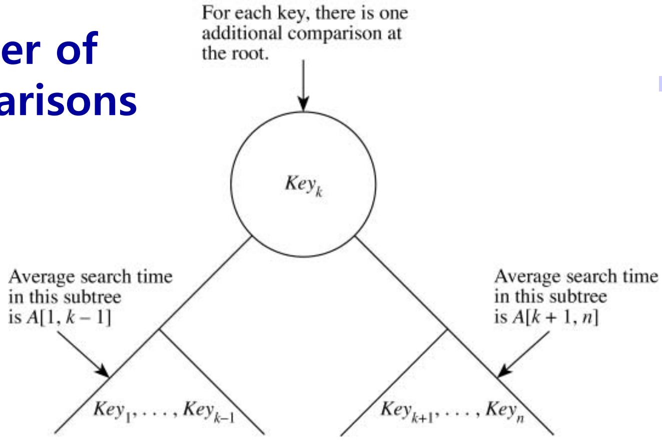
p_i : the probability of node i



$p_1 = 0.7,\ p_2 = 0.2,\ p_3 = 0.1$ (Fig. 3.11)



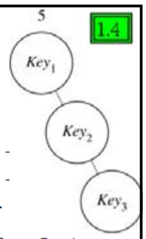
Number of Comparisons



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

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

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



Ex. 3.7) $p_1 = 0.7$, $p_2 = 0.2$, $p_3 = 0.1$

$A[1][3] = \min(\cancel{A[1][0]} + A[2][3],\ A[1][1] + A[3$
$\quad A[1][2] + \cancel{A[4][3]}) + p_1 + p_2 +$
$= \min(0 + 0.4,\ 0.7 + 0.1,\ 1.1 + 0) + 0.7 + 0.2 + 0.1$
$= 0.4 + 1.0 = 1.4$

$A[2][3] = \min(\cancel{A[2][1]} + A[3][3],\ A[2][2] + \cancel{A[4][3]})$
$\quad + p_2 + p_3$
$= \min(0 + 0.1,\ 0.2 + 0) + 0.2 + 0.1 = 0.4$

$A[1][2] = \min(\cancel{A[1][0]} + A[2][2],\ A[1][1] + \cancel{A[3][2]})$
$\quad + p_1 + p_2$
$= \min(0 + 0.2,\ 0.7 + 0) + 0.7 + 0.2 = 1.1$

Gilbert and Moore's Algorithm

- *by E. N. Gilbert and E. F. Moore, 1959)*
- A[i][j] : the minimum # of comparisons for Key_i to Key_j in a tree

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

where $A[1][0] = 0, \quad A[n+1][n] = 0$

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

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

use $A[i][i-1] = 0, \quad A[j+1][j] = 0$ for convenience

Algorithm *optsearchtree*

- Example 3.9

p: Don 3/8, Isabelle 3/8, Ralph 1/8, Wally 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

A[1][2] = min(A[1][0]+A[2][2], A[1][1]+A[3][2]) + P1+P2
= min(0+3/8, 3/8+0) + 3/8+3/8 = 9/8   k=1 or 2
A[2][3] = ...
A[3][4] = ...
A[1][3] = min(A[1][0]+A[2][3], A[1][1]+A[3][3], A[1][2]+A[4][3])
+ P1+P2+P3
= min(0+5/8, 3/8+1/8, 9/8+0) + 3/8+3/8+1/8
= 4/8+7/8 = 11/8   k=2
A[2][4] = min(A[2][1]+A[3][4], A[2][3]+A[4][4], A[2][4]+A[5][4])
+ P2+P3+P4
= ...
A[1][4] = min(A[1][0]+A[2][4], A[1][1]+A[3][4], A[1][2]+A[4][4],
A[1][3]+A[5][4]) + P1+P2+P3+P4
= min(0+1, 3/8+3/8, 9/8+1/8, 11/8+0) + 3/8+3/8+1/8+1/8
= 6/8 + 1 = 14/8 = 7/4   k=2

관찰사항: A[1][2], A[2][3] 등의 계산 결과를 중복사용

Algorithm *optsearchtree*

- ✓Find the optimal search tree:
  - ✓Alg. 3.9 **optsearchtree**
  - ✓R[i][j] : a value of k that gave the minimum
- ✓Build the tree:
  - ✓Alg. 3.10 **tree**
- ✓*전체 프로그램*

```
{
    optsearchtree(n, p, &minavg, R);
    root = tree(1, n);
}
```

Algorithm 3.9 optsearchtree

```
Void optsearchtree(int n, float p[], float& minavg, index R[][])
{  index i, j, k, diag; 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-1] = 0; R[i][i] = I;
   }
   A[n+1][n] = 0; R[n+1][n] = 0;
   for (diag=1; diag <= n-1; diag++) {
     for (i=1; i <= n-diag; i++) (
       j = i+diag;
       A[i][j] = min_{k=i to j} (A[i][k-1] + A[k+1][i]) + sum_{k=i to j} p[k];
       R[i][j] = min에서 선택된 k
     }
   minavg = A[1][n];
}
```


Algorithm 3.10 tree

```
node_pointer tree (index i, index 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;
   }
}
```

Complexity of optsearchtree

- Time Complexity in O Notation
  $T(n) < c \times n \times n \times n$
  $T(n) \in O(n^3)$
- Time Complexity in Θ Notation

– for minimum op.: $j - i + 1 = diag + 1$

– for each inner loop: $(n - diag) \times (diag + 1)$

– for the outer loop,

$$T(n) = \sum_{diag=1}^{n-1} (n - diag) \cdot (diag + 1) = \ldots$$

$$= n(n-1)(n+4)/6 \in \Theta(n^3)$$

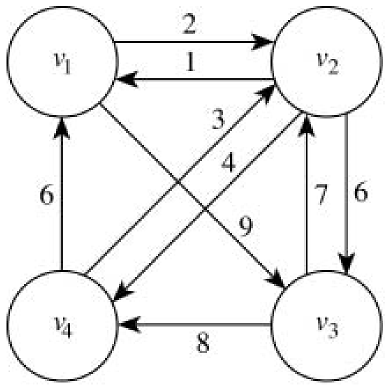
- Time Complexity of tree
  a divide-and-conquer algorithm
  T(n) = ???
- More Enhancements
  – $\Theta(n^2)$ by F. Yao, 1982

Traveling Salesperson Problem

- The Problem
  - determine a shortest route that starts at the home city, visits each city, and returns to the home city
- Terminology
  - ***tour*** (Hamiltonian circuit/cycle):
  - ***optimal tour*** : a tour with the minimum length
- Traveling Salesperson Problem
  - Find an optimal tour for a given weighted di-graph

Example

- (Fig. 3.16) the tour is [v_1, v_3, v_4, v_2, v_1]



- For a complete graph : (n-1)! tours

Dynamic Programming Approach

- Data Structure
  - V : set of the vertices
  - A : a subset of V
  - D[i][A] : length of shortest path from v_i to v_1 passing each vertex in A exactly once
  - the minimum length : D[1][V-{v_1}]
- Fig. 3.16

D[1][{v2,v3,v4}] =
min (W[1][2]+ D[2][{v3,v4}],
W[1][3]+ D[3][{v2,v4}],
W[1][4]+ D[4][{v2,v3}])
D[2][{v3,v4}] = min (W[2][3]+ D[3][{v4}],
W[2][4]+ D[4][{v3}])
D[3][{v2,v4}] = ... D[4][{v2,v3}] = ...
D[3][{v4}] = W[3][4] + D[4][{}]
D[4][{v3}] = ...

(참고) Description of subset A

- use n bit integer
- Ex) {v4} = 1000_B, {v3, v1} = 0101_B, {} = 0000_B
- D[1][{v2,v3,v4}] = min (W[1][2]+ D[2][{v3,v4}],

⇒ D[1][1110_B] = min(W[1][2] + D[2][1100_B], ...)
⇒ D[1][14] = min(W[1][2] + D[2][12], ...)

Yong-Seok Kim (yskim@kangwon.ac.kr)

Dynamic Programming Approach

Optimal tour length:

A=V−{v_1}

D[1][A] = $\min_{2 \leq j \leq n}(W[1][j] + D[j][A - \{v_j\}])$

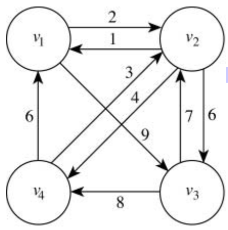
For $i \neq 1$ and $v_i \notin A$,

– $D[i][A] = \min_{v_j \in A}(W[i][j] + D[j][A - \{v_j\}])$ if $A \neq \emptyset$

– $D[i][\emptyset] = W[i][1]$

Build D[i][A] from |A|=0 to |A| = n−1

Ex. 3.11 (Fig 3.16)



$D[i][A] = \min_{v_j \in A}(W[i][j] + D[j][A - \{v_j\}])$ if $A \neq \emptyset$

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

D[2][{}] = W[2][1] = 1, D[3][{}] = ∞, D[4][{}] = 6

D[2][{3}] = W[2][3] + D[3][{}] = 6 + ∞ = ∞

D[2][{4}] = W[2][4] + D[4][{}] = 4 + 6 = 10;

D[3][{2}] = W[3][2] + D[2][{}] = 8; D[3][{4}] = W[3][4] + D[4][{}] = 14

D[4][{2}] = W[4][2] + D[2][{}] = 4; D[4][{3}] = W[4][3] + D[3][{}] = ∞

D[2][{3,4}] = min(W[2][3]+D[3][{4}], W[2][4]+D[4][{3}])
= min(6+14, 4+ ∞) = 20

D[3][{2,4}] = min(W[3][2]+D[2][{4}], W[3][4]+D[4][{2}]) = 12

D[4][{2,3}] = min(W[4][2]+D[2][{3}], W[4][3]+D[3][{2}]) = ∞

D[1][{2,3,4}] = min(W[1][2]+D[2][{3,4}], W[1][3]+D[3][{2,4}],
W[1][4]+D[4][{2,3}]) = min(2+20, 9+12, ∞+∞) = 21

Algorithm 3.11 *travel*

P[i][A] : v_i 에서 A의 모든 정점을 한번만 거쳐서 v_1 으로 가는 가장 짧은 경로의 첫번째 방문 정점

```
Void travel (int n, number W[][], indexP[][], number& minlenth)
{ 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 (V-{v1}의 부분 집합으로서 원소가 k개인 모든 A에 대하여)
      for (A에 포함되지 않으면서 1도 아닌 모든 정점 i에 대하여) {
        D[i][A] = min j∈A ( W[i][j] + D[j][A-{j}] )
        P[i][A] = min 이 되는 인덱스 j  }
  A = V – {v1};
  D[1][A] = min j∈A ( W[i][j] + D[j][A-{j}] )
  P[1][A] = min 이 되는 인덱스 j
  minlength = D[1][A]
}
```


전체 프로그램:

```
travel(n, W, P, minlength);
PrintTour();
```

● Print the optimal tour from P (the ending part of section 3.6)

```
PrintTour()
{ index k;  set A;
    A = V - {v1};
    k = 1;
    cout << "v1 ";
    while (A is not empty) {
        k = P[k][A];
        cout << "v" << k << " ";
        A = A - {vk};
    }
    cout << "v1 ";
}
```

Complexity of Alg. 3.11

- Simple Analysis of Time Complexity

Since, count of 1^{st} level for loop = n−2 < n
count of 2^{nd} level for loop < 2^n
count of 3^{rd} level for loop < n
and count for minimum = k < n,
$T(n) < n \cdot 2^n \cdot n \cdot n$ and therefore, $T(n) \in O(n^3 2^n)$

- Time Complexity in O Notation
  - The triply nested loop is most significant
  - # of loops for the outer 2 loop nesting:

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

  - count of the inner most loop = n−1−k < n
  - count for minimum = k < n

Therefore, $T(n) < 2^n \times n \times n$, and $T(n) \in O(n^2 2^n)$

- Time Complexity in Θ Notation

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

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

$$= (n-1)\sum_{k=1}^{n-2} k \binom{n-2}{k} \quad \Leftarrow \text{Thm. 3.1 (p.135)}$$

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

⇨ Much better than (n−1)!

- Space Complexity

Space for D and P: $S(n) \in \Theta(n 2^n)$

Characteristics of TSP

- $(n-1)(n-2)2^{n-3}$ is much better than (n−1)!
  - (Ex. 3.12) TSP for 20 cities
  - assume 1 microsec for each path
  - Brute force algorithm

    $T_{BF}(20)$ = (20−1)! microsec = 3,857 years
  - Dynamic programming approach

    $T_{DP}(20) = (20-1)(20-2)2^{20-3}$ microsec = 45 sec

    Note) not practical for more than 50 cities
- 후속 장에서 TSP 다시 보기
  - Chapter 6 Branch-and-Bound

DNA Sequence Alignment Problem

- Find corresponding DNA sequence alignment
  - Base: **A, G, C, T**
  - Base pair: **A-T, G-C**
- Ex 3.13
  - x: **A A C A G T T A C C**
  - y: **T A A G G T C A**

Possible alignments

① **- A A C A G T T A C C**

T A A - G G T - - C A

② **A A C A G T T A C C**

T A - A G G T - C A

- Penalty: Mismatch 1, Gap 2: Cost ① 10, ② 7
- Penalty: Mismatch 3, Gap 1: Cost ① 10, ② 11

Divide and conquer approach

- ✓ x[0..m-1] and y[0..n-1]
- ✓ opt(i,j): minimum cost for x[i..m-1] and y[j..n-1]
- ✓ opt(0,0): minimum cost for x and y
- ✓ opt(i,j) = min(opt(i+1, j+1) + penalty,
  opt(i+1, j) + 2, ⇦ gap on x[i]
  opt(i, j+1) + 2) ⇦ gap on y[j]

  where penalty is 0 if x[i] == y[j], 1 otherwise

Divide and Conquer Version *opt* (Alg. 3.12)

```
void opt(int i, int j)
{
   if (i == m)                // no more base on x
      return 2*(n-j);
   if (j == n)                // no more base on y
      return 2*(m-i);
   if (x[i] == y[j]) penalty = 0;
   else penalty = 1;
   return min(opt(i+1, j+1) + penalty, opt(i+1, j) + 2,
              opt(i, j+1) + 2);
}
```

✓문제의 답: optimal_cost = opt(0,0);

(참고) Time Complexity Analysis

$T(m,n) = T(m-1,n-1) + T(m-1,n) + T(m,n-1) + c$

Assume $m > n$

$T(m,n) > T(n,n)$

$\quad > T(n-1,n-1) + T(n-1,n) + T(n,n-1)$

$\quad > 3T(n-1,n-1)$

Let $U(n) = T(n,n)$

$U(n) > 3U(n-1) > 3^2U(n-2) > \ldots > 3^{n-1}U(1)$

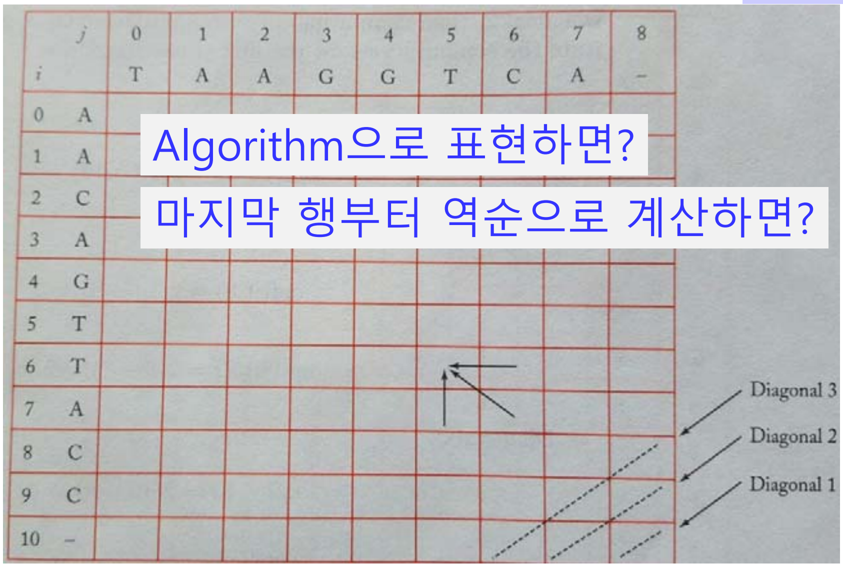
$\therefore U(n) > 3^{n-1}$ and $U(n) \in$???

$\therefore T(m,n) \in$???

Dynamic Programming Algorithm

opt[i][j] = 2(n-j), if (i == m)
2(m-i), if (j == n)
min(opt[i+1][j+1] + penalty,
opt[i+1][j] + 2,
opt[i][j+1] + 2)
where penalty is 0 if x[i] == y[j], 1 otherwise

Calculate opt[i][j] in diagonal order



Algorithm으로 표현하면?

마지막 행부터 역순으로 계산하면?

Example: Fig. 3.20

x : A A C A G T T A C C
y : T A A G G T C A

i \ j		0	1	2	3	4	5	6	7	8
		T	A	A	G	G	T	C	A	-
0	A	7	8	10	12	13	15	16	18	20
1	A	6	6	8	10	11	13	14	16	18
2	C	6	5	6	8	9	11	12	14	16
3	A	7	5	4	6	7	9	11	12	14
4	G	9	7	5	4	5	7	9	10	12
5	T	8	8	6	4	4	5	7	8	10
6	T	9	8	7	5	3	3	5	6	8
7	A	11	9	7	6	4	2	3	4	6
8	C	13	11	9	7	5	3	1	3	4
9	C	14	12	10	8	6	4	2	1	2
10	-	16	14	12	10	8	6	4	2	0

Example: Fig. 3.20

Input

x: A A C A G T T A C C

y: T A A G G T C A

Result (Cost = 7)

x: A A C A G T T A C C

y: T A - A G G T - C A

(참고) 대상이 반대쪽의 서열일 수도 있음 (A-T, G-C)

x: A A C A G T T A C C

y: A T T C C A G T

→ 2가지 각각 cost를 계산하여 작은 쪽을 적용

(참고) Dynamic Programming Version

```
int optDP(int x[], int y[])
{ index i, j; int opt[0..m][0..n];
   for(i=0; i <= n; i++) opt[i][n] = 2*(m-i);
   for(j=0; j <= m; j++) opt[m][j] = 2*(n-j);
   for(i=m-1; i>=0; i--)
      for(j=n-1; j>=0; j--) {
         if(x[i] == y[j]) penalty = 0;
         else             penalty = 1;
         opt[i][j] = min(opt[i+1][j+1] + penalty,
                         opt[i+1][j] + 2, opt[i][j+1] + 2);
      }
   return opt[0][0];
}
```