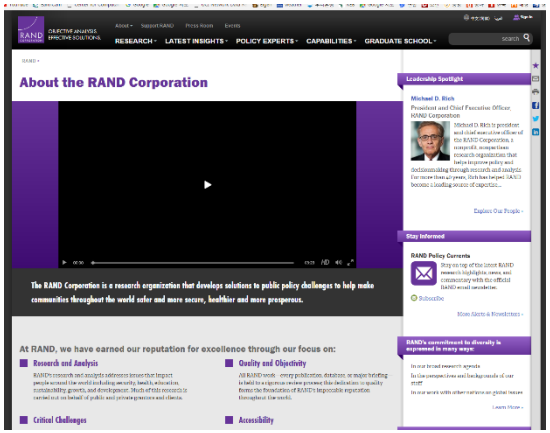
3장 동적계획
(Dynamic Programming)

Dynamic Programming 어원

- Richard Bellman (RAND corporation) 이 1950에 만듬



RAND: 공공정책연구소

- 원래의 방법은 multistage decision processes
- 이 단어를 정부기관의 관료에게 그럴듯하게 보이게 하기 위해
- Dynamic: multistage, time-varying
- Programming: 훈련과 병참지원을 위한 military schedule의 optimal program을 구하고는 방법을 program이라고 하였음.
- (예) linear programming(선형계획법), nonlinear(비선형) programming, Mathematical programming
- mathematical programming은 산업경영공학과의 핵심적인 분야

Operations Research © 2002 INFORMS
Vol. 50, No. 1, January–February 2002, pp. 48–51

RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS
University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was con-

what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very inter-

동적계획

- divide-and-conquer(분할정복식, 재귀) 알고리즘은 하향식(top-down) 해결법
  - ✓ 나누어진 부분들 사이에 서로 상관관계가 없는 문제를 해결하는데 적합
- 피보나찌 알고리즘은 나누어진 부분들이 서로 연관이 있음.
  - ✓ 같은 항 $f(i)$ 를 한 번 이상 → 비효율적
  - ✓ 분할정복식 방법은 적합하지 않음.
- 동적계획법(dynamic programming)은 상향식 해결법(bottom-up approach)
  - ✓ 분할정복식 방법과 마찬가지로 문제를 나눈 후에 나누어진 부분들을 먼저 푼다.
  - ✓ 인덱스를 효과적으로 설정하여 작은 문제들의 중복해결을 배제
  - ✓ 작은 문제 해결을 먼저 → 결과를 큰 문제의 해결로 확산
  - ✓ 개발 절차
    - (1) 재귀 관계식(recursive property) 정립
    - (2) 작은 사례를 먼저 해결하는 상향식 방법으로 진행

이항계수

$$(a+b)^n = {}_nC_0a^nb^0+{}_nC_1a^{n-1}b^1+\cdots+{}_nC_{n-1}a^1b^{n-1}+{}_nC_na^0b^n$$

$$(a+b)^2 = {}_2C_0a^2b^0+{}_2C_1a^1b^1+{}_2C_2a^0b^2= a^2+2ab+b^2$$

$$(a+b)^3 = {}_3C_0a^3b^0+{}_3C_1a^2b^1+{}_3C_2a^1b^2+{}_3C_3a^0b^3$$
$$= a^3+3a^2b^1+3a^1b^2+b^3$$

이항계수 구하기

- 이항계수(binomial coefficient) 공식

$$ {}_nC_k = \binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } 0 \le k \le n $$

- 계산량이 많은 $n!$이나 $k!$을 계산하지 않고 이항계수를 구하기 위해서 다음 식을 사용한다.(100! ?)

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

알고리즘: 분할정복식 접근방법

- 문제: 이항계수를 계산한다.
- 입력: 음수가 아닌 정수 n과 k, 여기서 $k \le n$
- 출력: bin, ${}_nC_k$
- 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)
}
```

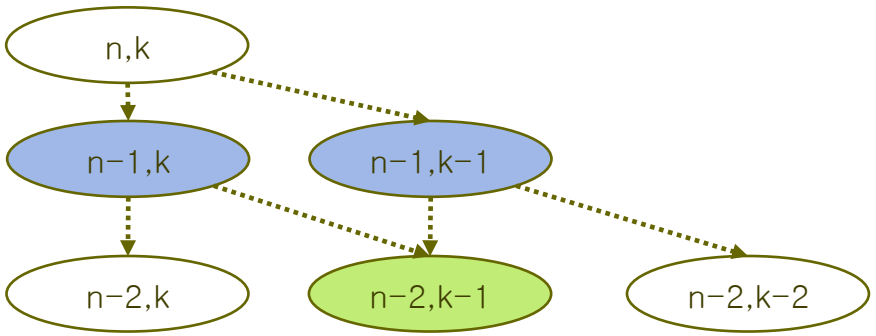
$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

알고리즘: 분할정복식 접근방법

- 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
  - ✓ ${}_nC_k$를 구하기 위해서 이 알고리즘이 계산하는 항(term)의 개수는 $2 \times {}_nC_k - 1$ 이다.

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



$_nC_k$를 구하기 위해서 이 알고리즘이 계산하는 항(term)의 개수는 $2 \times {}_nC_k - 1$ 이다.

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

- **증명**: (n에 대한 수학적귀납법으로 증명)
- **귀납출발점**: 항의 개수 n이 1일 때 $2\times{}_nC_k-1=2\times1-1=1$ 이 됨을 보이면 된다. ${}_1C_k$ 는 $k = 0$ 이나 1 일 때 1이므로 항의 개수는 항상 1이다.
- **귀납가정**: ${}_nC_k$를 계산하기 위한 항의 개수는 $2\times{}_nC_k-1$ 이라고 가정한다.
- **귀납절차**: ${}_{n+1}C_k$ 를 계산하기 위한 항의 개수가 $2\times{}_{n+1}C_k-1$임을 보이면 된다. 알고리즘에 의해서 ${}_{n+1}C_k = {}_nC_{k-1}+{}_nC_k$ 이므로, 즉 ${}_{n+1}C_k$를 계산하기 위한 항의 총 개수는 ${}_nC_{k-1}$를 계산하기 위한 총 개수와 ${}_nC_k$를 계산하기 위한 항의 총 개수에다가 이 둘을 더하기 위한 항 1을 더한 수가 된다. 그런데 ${}_nC_{k-1}$ 를 계산하기 위한 항의 개수는 가정에 의해서 $2\times{}_nC_{k-1}-1$ 이고, ${}_nC_k$를 계산하기 위한 항의 개수는 가정에 의해서 $2\times{}_nC_k-1$ 이다. 따라서 항의 총 개수는

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

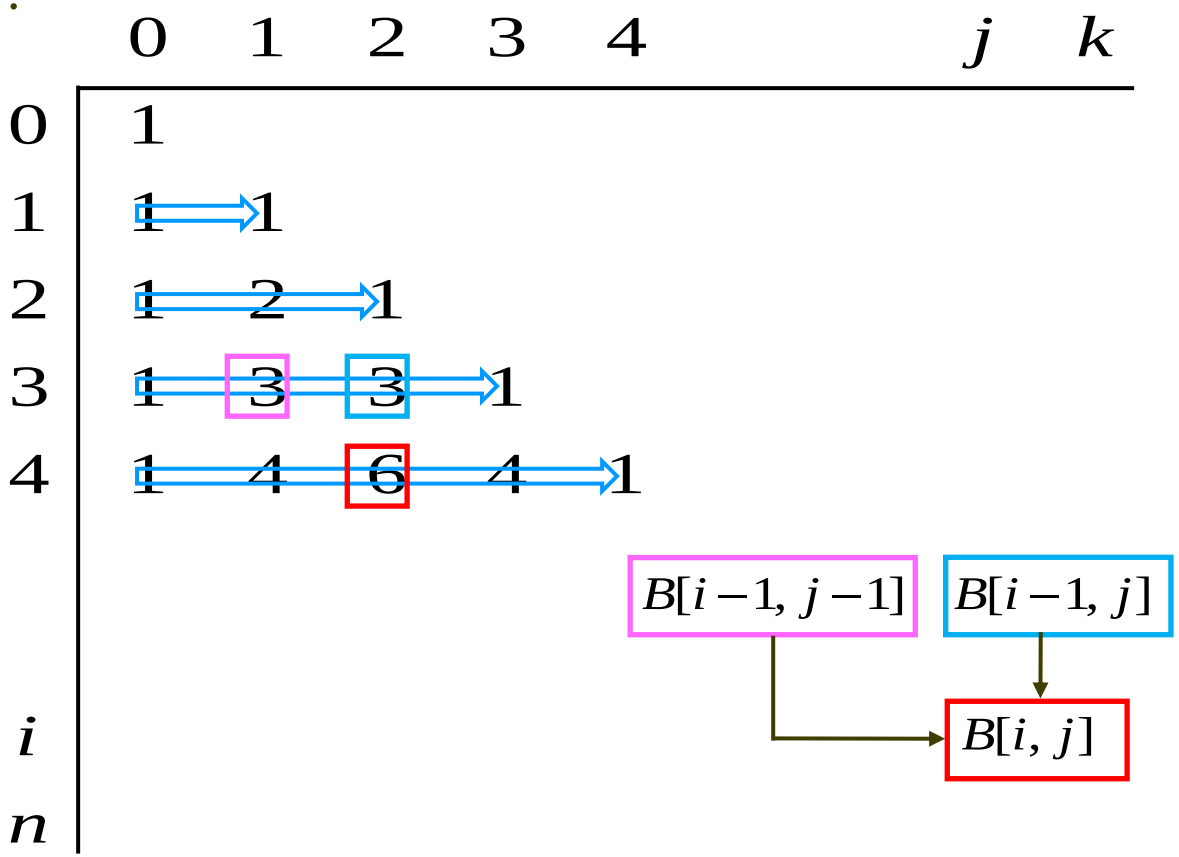
동적계획식 알고리즘 설계전략

1. 재귀 관계식(recursive property)을 정립:
   - 2차원 배열 B를 만들고, 각 $B[i][j]$에는 ${}_iC_j$ 값을 저장하도록 함
   - 그 값은 다음과 같은 관계식으로 계산

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

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

2. ${}_nC_k$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.

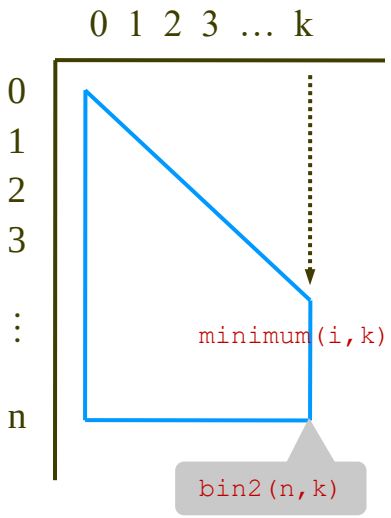


Pascal 의 삼각형

동적계획 알고리즘

- 문제: 이항계수를 계산한다.
- 입력: 음수가 아닌 정수 n과 k, 여기서 $k \leq n$
- 출력: bin2, ${}_nC_k$

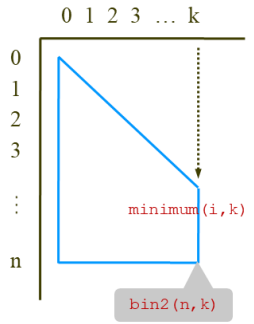
```
int bin2(int n, int k) {
  index i, j;
  int B[0..n][0..k];
  for(i=0; i <= n; i++)
    for(j=0; j <= minimum(i,k); j++)
      if (j==0 || j == i)
        B[i][j] = 1;
      else B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```



동적계획 알고리즘의 분석

- 단위연산: for-j 루프 안의 문장
- 입력의 크기: n, k

i	0	1	2	3	…	k	k+1	…	n
j루프의 수행회수	1	2	3	4	…	k+1	k+1	…	k+1



따라서 총 수행횟수는:

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

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

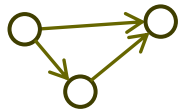
```
int bin2(int n, int k) {
index i, j; int B[0..n][0..k];

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];
}
```

동적계획 방법

- 작은 문제부터 해결
- 인덱스를 조정하여 중복 계산 회피
- 작은 문제의 해를 모아서 다음 단계의 문제 해결에 사용

그래프 용어

- 정점(vertex, node), 이음선(edge, arc)
- 방향 그래프(directed graph/digraph)



- 가중치(weight), 가중치 포함 그래프(weighted graph)
- 경로(path): 각 정점에서 다음 정점을 잇는 이음선이 존재하는 일련의 정점들.
- 단순경로(simple path) – 같은 정점을 두 번 지나지 않는 경로
- 순환(cycle) – 한 정점에서 다시 그 정점으로 돌아오는 경로
- 순환 그래프(cyclic graph) vs 비순환 그래프(acyclic graph)
  - cycle이 있는(없는) 그래프
- 길이(length): 경로상에 있는 가중치의 합(가중치포함그래프). 경로상의 이음선의 개수(가중치가 없는 그래프)

가중치 포함 방향 그래프의 예

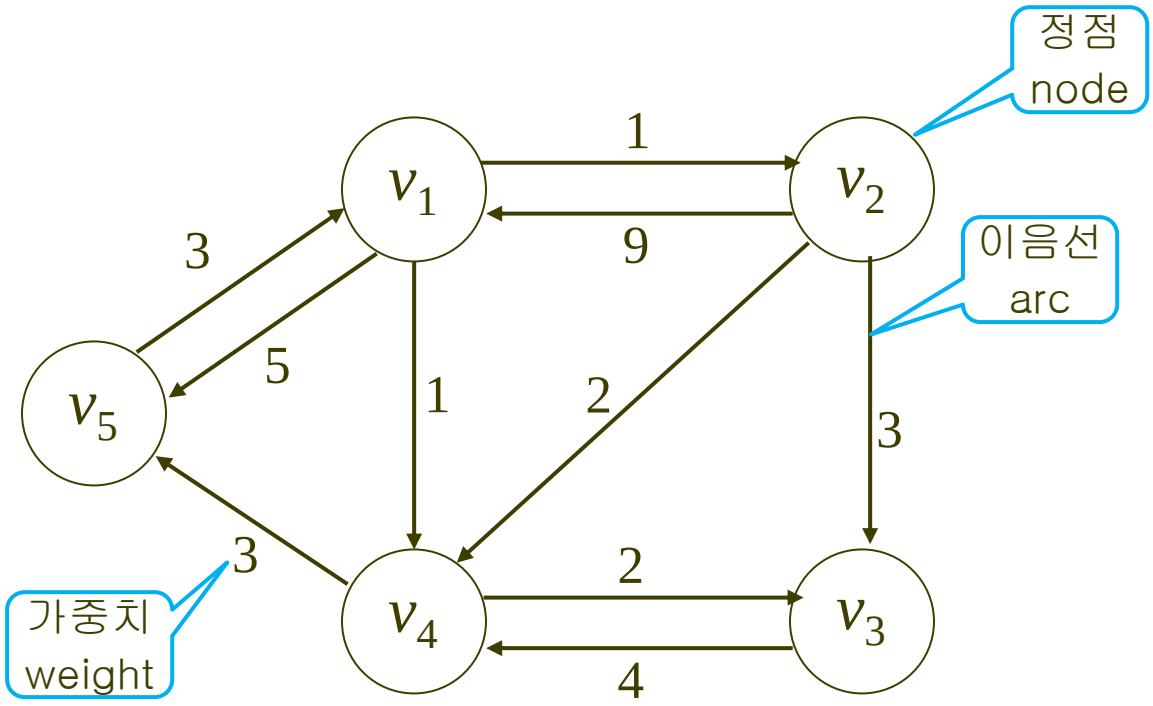


그림 3.2

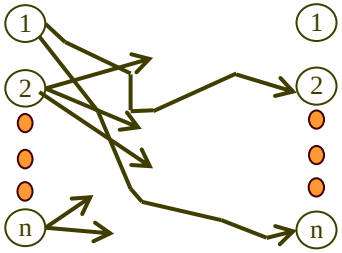
최단경로 문제
(all-pairs shortest paths problem)

- 보기 : 모든 도시에 대해, 한 도시에서 다른 도시로 갈 수 있는 가장 짧은 길을 찾는 문제

- 문제: 가중치 포함, 방향성 그래프에서 최단경로 찾기

- 최적화문제(optimization problem)
  - ✓ 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.

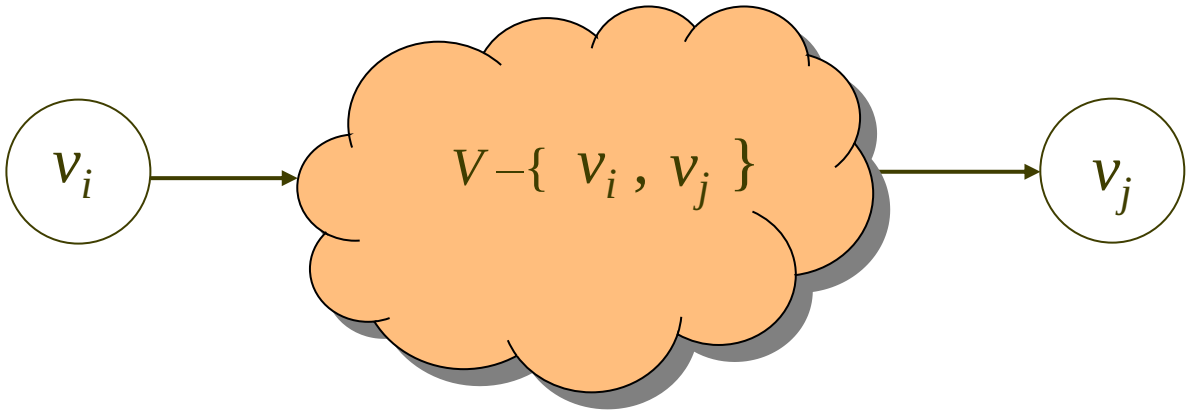


최단경로찾기: 무작정 알고리즘

- 무작정 알고리즘(brute-force algorithm)
  정점 v_i에서 다른 정점 v_j로의 가능한 모든 경로들의 길이를 구한 뒤, 그들 중에서 최소길이의 경로를 찾는다.
- 분석:
  - ✓ 주어진 그래프는 n개의 정점을 가지고 있고, 모든 정점들 사이에 이음선이 존재한다고 가정.
  - ✓ 정점 v_i에서 다른 정점 v_j로 가는 경로들을 다 모아 보면, 그 경로들 중에서 나머지 모든 정점을 한번씩은 꼭 거쳐서 가는 경로(중간에 n-2개의 정점 사용)들도 포함되어 있는데, 그 경로들의 수만 우선 계산해 보자.
  - ✓ v_i에서 출발하여 처음에 도착할 수 있는 정점의 가지 수는 n-2개. 그 중에 하나를 선택하면, 그 다음에 도착할 수 있는 정점의 가지 수는 n-3개. 이렇게 계속하여 계산해 보면, 총 경로의 개수는 $(n-2)(n-3)\ldots 1 = (n-2)!$.
  - ✓ 이 외에도 중간에 n-3, n-4, ..,1,0개의 정점을 사용하는 경로도 확인해야 함.
  - ✓ n-2개의 중간 정점을 사용하는 경로의 개수 만 보아도 지수보다 훨씬 크므로, 이 알고리즘은 절대적으로 비효율적!

$V=\{v_1, \ldots, v_n\}$

1) When n-2 nodes are used for the shortest path from v_i to v_j.



2) When n-3 nodes are used for the shortest path from v_i to v_j.

3) When n-4 nodes are used for the shortest path from v_i to v_j.

동적계획식 설계전략 - 자료구조

- 그래프의 인접행렬(adjacency matrix)식 표현: W

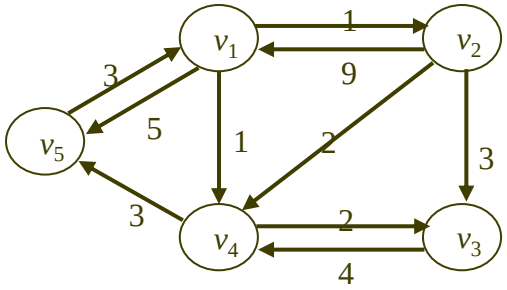
$$W[i][j] = \begin{cases} \text{이음선의 가중치}, & v_i\text{에서}v_j\text{로 가는 이음선이 있는 경우} \\ \infty, & v_i\text{에서}v_j\text{로 가는 이음선이 없는 경우} \\ 0, & i = j\text{인 경우} \end{cases}$$

- 그래프에서 최단경로의 길이의 표현:

$D^{(k)}[i][j] =$ 집합 $\{v_1, v_2, ..., v_k\}$의 정점들 만을 이용해서(이들을 모두 이용해야 하는 것은 아님. 일부만 이용 가능) v_i에서 v_j로 가는 최단경로의 길이

동적계획식 설계전략 - 자료구조

- ✓ W : 그림 3.2에 있는 그래프의 인접행렬식 표현(문제)
- ✓ D : 각 정점들 사이의 최단 거리(답)

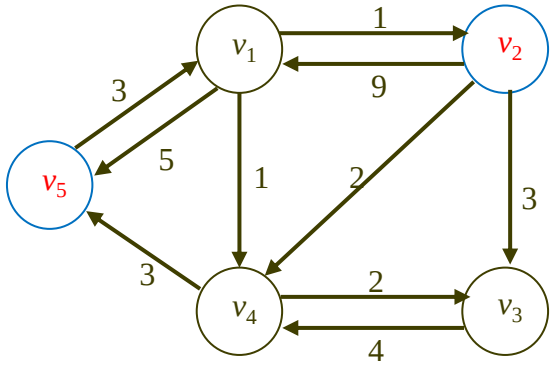


$W[i][j]$	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D[i][j]$	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

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

(예 3.2) $0 \le k \le 5$ 일 때, $D^{(k)}[2][5]$?



(1) $D^{(0)}[2][5]$ = length$[v_2, v_5]=\infty$

(2) $D^{(1)}[2][5]$ = min(length$[v_2, v_5]$, length$[v_2, v_1, v_5]$)

= min(∞ , 14) = 14

(3) $D^{(2)}[2][5] = D^{(1)}[2][5]=14$

(4) $D^{(3)}[2][5] = D^{(2)}[2][5]=14$

(5) $D^{(4)}[2][5]$ = min(length$[v_2, v_1, v_5]$, length$[v_2, v_4, v_5]$,

length$[v_2, v_1, v_4, v_5]$, length$[v_2, v_3, v_4, v_5]$),

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

(6) $D^{(5)}[2][5] = D^{(4)}[2][5]=5$

– 가능한 경로를 쉽게 확인할 수 있는 조직적인 방법 필요

동적계획식 설계절차

- 1. $D^{(k-1)}$을 가지고 $D^{(k)}$를 계산할 수 있는 재귀 관계식(recursive property)을 정립

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

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

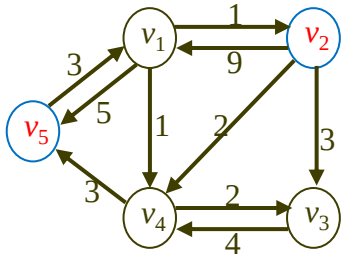
(예) $D^{(5)}[1][3] = D^{(4)}[1][3] = 3$

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

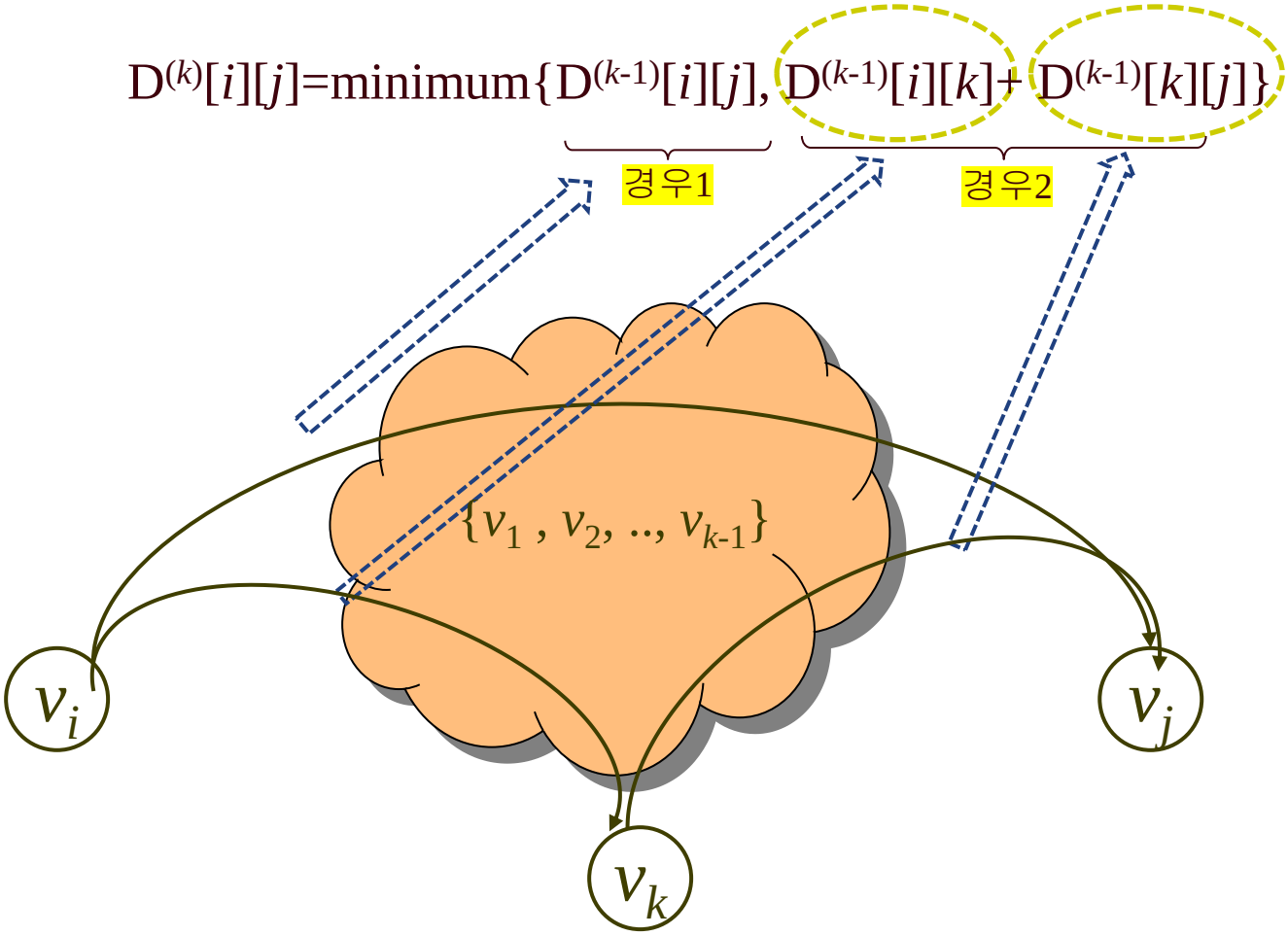
(예) $D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$

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

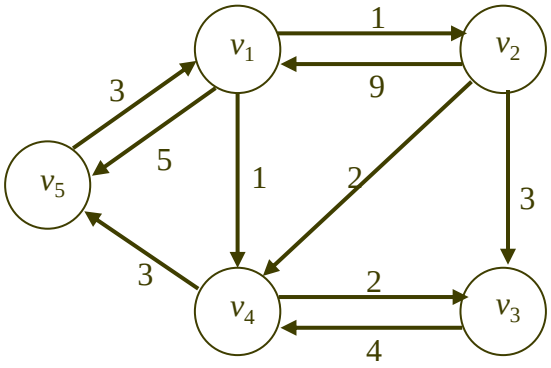
$$D^{(0)}, D^{(1)},\ldots\ldots, D^{(n)}$$



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



(예 3.3)



(1) $D^{(1)}[2][4] = \min(D^{(0)}[2][4], D^{(0)}[2][1]+D^{(0)}[1][4])$

$= \min(2, 9+1) = 2$

(2) $D^{(1)}[5][2] = \min(D^{(0)}[5][2], D^{(0)}[5][1]+D^{(0)}[1][2])$

$= \min(\infty, 3+1) = 4$

(3) $D^{(1)}[5][4] = \min(D^{(0)}[5][4], D^{(0)}[5][1]+D^{(0)}[1][4])$

$= \min(\infty, 3+1) = 4$

$D^{(1)}$을 모두 계산한 후 $D^{(2)}$를 계산한다.

$D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2]+D^{(1)}[2][4])$ /* $D^{(1)}$사용 */

$= \min(4, 4+2) = 4$

$D^{(2)}, D^{(3)}, D^{(4)}, D^{(5)}$ = D를 순차적으로 계산한다.

Floyd의 알고리즘 I

- 문제: 가중치 포함 그래프의 각 정점에서 다른 모든 정점까지의 최단 거리를 계산하라.
- 입력: 가중치 포함, 방향성 그래프 W와 그 그래프에서의 정점의 수 n.
- 출력: 최단거리의 길이가 포함된 배열 D

```
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]);
}
```

```
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

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

D[i][j] 계산 시 D[i][k], D[k][j]값이 사용됨. 만일 $D^{[k]}$ 계산 시 D[i][k], D[k][j] 값이 변경된다면, 별도의 D를 저장할 공간이 필요. 그러나 필요하지 않다.

[이유]

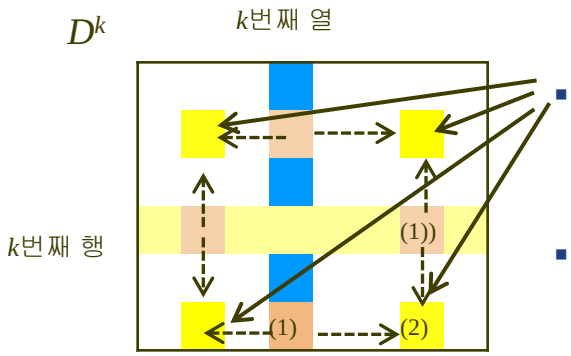
- 추가 공간 필요 없이 D만을 이용하여 데이터 저장 가능

(이유) *k*번째 행과 *k*번째 열에 있는 값들이 루프의 *k*번째 반복을 수행하는 동안 변하지 않기 때문

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

0

0



- 이 element 계산시 D^{k-1}[i,k], D^{k-1}[k,j] 필요. 그런데 이것은 D^{k-1}[i,k]= D^k[i,k], D^{k-1}[k,j] =D^k[k,j] . 따라서 overwrite 가능.
- (1)은 (2) 보다 먼저 계산. (2) 계산 시 (1)의 값 D^{k-1}이 필요. (1)의 값이 update되면 D^k 로 변경. 그러나, (1)의 값이 변동이 없으므로 연산 수행 가능

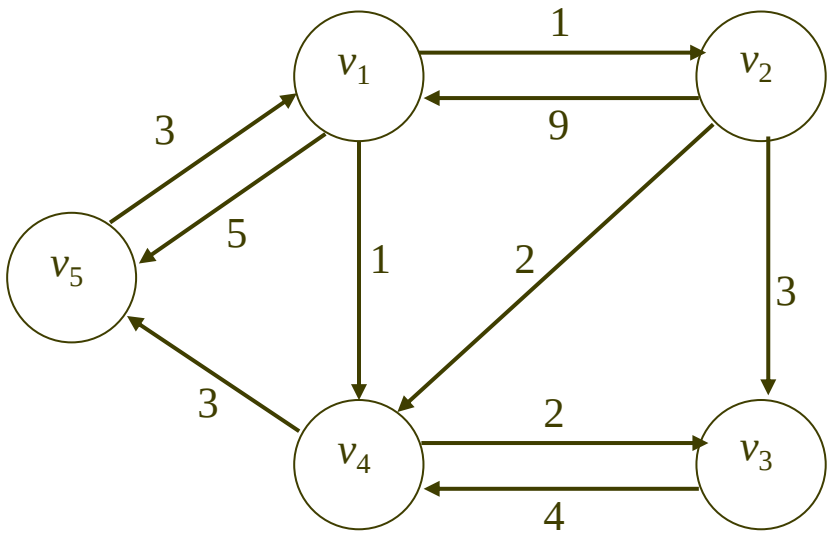
Floyd의 알고리즘 II

- 출력: 최단경로의 길이가 포함된 배열 D, 그리고 다음을 만족하는 배열 P.

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

```
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];
                }
}
```

- 앞의 예에 대한 결과 P



$P[i][j]$	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

5→3 경로 : 5→?→3

5→4→3

5→?→4→?→3

5→?→1→?→4→?→3

5→1→4→?→3

5→1→4→3

최단경로의 출력

- 문제: 최단경로 상에 놓여 있는 정점을 출력.

```
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);
         }
 }
```

$W(n) \in \Theta(n)$

$P[i][j]$	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

- 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 최단경로 v_5, v_1, v_4, v_3 이다.

동적계획법에 의한 설계 절차

(단계 1) 문제의 입력에 대해서 최적(optimal)의 해답을 주는 재귀 관계식 (recursive property)을 설정

(단계 2) 상향적으로 최적의 해답을 계산

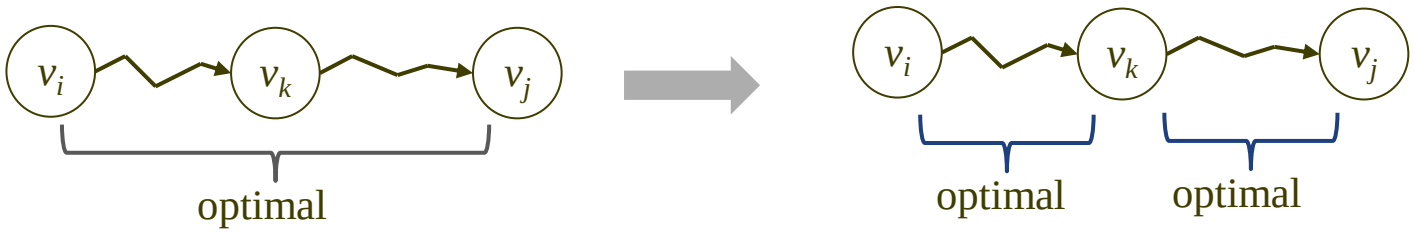
(단계 3) 상향적으로 최적의 해답을 구축

최적의 원칙

- 어떤 문제 사례에 대한 최적 해가 그 사례를 분할한 부분사례에 대한 최적 해를 항상 포함하고 있으면, 그 문제는 **최적의 원칙(the principle of optimality)**이 적용된다 라고 한다.

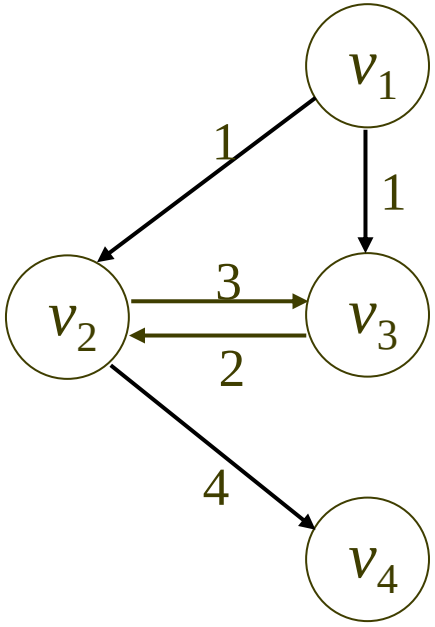
- 최적의 원칙이 적용되어야 지만 동적계획법을 사용할 수 있다.

- (예) 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.



- 모든 문제가 최적의 원칙이 적용되는 것은 아니다.

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.

  v_1-v_3 -v_2 -v_3 -v_2 -v_3 -v_2 -v_4 : 불허

$$\begin{pmatrix} x & x \\ x & x \\ x & x \end{pmatrix} \times \begin{pmatrix} y & y & y & y \\ y & y & y & y \end{pmatrix} = \begin{pmatrix} z & z & z & z \\ z & z & z & z \\ z & z & z & z \end{pmatrix}$$

3×2 2×4 3×4

- z 하나를 계산하는데 필요한 곱셈은 2회
- 행렬곱셈을 위해 총 $3\times2\times4$회의 곱셈 필요

연쇄 행렬곱셈(matrix-chain multiplication)

- $i \times j$ 행렬과 $j \times k$행렬을 곱하기 위해서는 $i \times j \times k$번 만큼의 곱셈이 필요.
- 연쇄적으로 행렬을 곱할 때, 어떤 행렬곱셈을 먼저 수행하느냐에 따라서 필요한 총 곱셈의 횟수가 달라짐.
- (예)
  - ✓ $A_1 \times A_2 \times A_3$.
  - ✓ A_1의 크기는 10×100, A_2의 크기 100×5, A_3의 크기 5×50.
  - ✓ $(A_1 \times A_2) \times A_3$ 곱셈의 총 횟수 7,500(=5,000+2,500)회
  - ✓ $A_1 \times (A_2 \times A_3)$ 곱셈의 총 횟수 75,000(=25,000+50,000)회
  - ✓ 따라서, 연쇄적으로 행렬을 곱할 때 곱셈의 횟수가 가장 적게 되는 최적의 순서를 결정하는 알고리즘을 개발하는 것이 목표.

연쇄 행렬곱셈 무작정 알고리즘

- 알고리즘: 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, $t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$이고 $t_2 = 1$이라는 사실은 쉽게 알 수 있다.
  - 따라서 $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2 = 2^{n-2} \in \Theta(2^n)$

$2^{n-2} \in \Theta(2^n)$. $t_n \in \Theta(2^n)$ 라는 것은 아님

$$A_1 \times (A_2 \times \ \ldots \ \times A_{n\text{-}1} \times A_n)$$

$t_{n\text{-}1}$

$$(A_1 \times A_2 \times \ \ldots \ \times A_{n\text{-}1}) \times A_n$$

$t_{n\text{-}1}$

$$t_n \geq t_{n\text{-}1} + t_{n\text{-}1} = 2\, t_{n\text{-}1}$$

연쇄 행렬곱셈 무작정 알고리즘

- 무작정 알고리즘의 모든 가능한 가지 수 P(n)
- P(n) = C(n-1) ····· : Catalan Number

$$C_n = \frac{1}{n+1}\binom{2n}{n} \quad : n\text{th Catalan number}$$

$$= \frac{4^n}{\sqrt{\pi} n^{3/2}}(1+\mathrm{O}(1/n))$$

$$= \frac{4n-2}{n+1}C_{n-1} \quad , (C_1 = 1)$$

- C(n)은 n개의 노드를 갖고 있는 이진트리의 모양의 경우의 수
- 2개의 matrix를 곱하는 경우의 수- A A -1가지

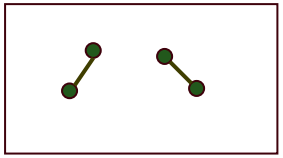
$$\mathrm{P}(2)=\mathrm{C}(1) = \frac{1}{1+1}\binom{2}{1} = \frac{1}{2}\times 2 = 1$$

- 이진트리 •

- 3개의 matrix를 곱하는 경우의 수- A(A A), (A A)A, -2가지

$$P(3)=C(2)=\frac{1}{2+1}\binom{4}{2}=\frac{1}{3}\times 6=2$$

- 이진트리

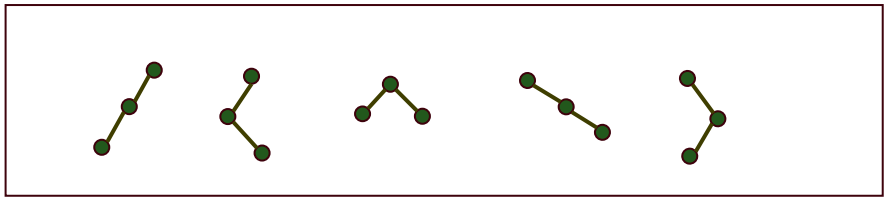


- 4개의 matrix를 곱하는 경우의 수-

A(A(A A)), (AA)(AA), A((A A)A), ((AA)A)A, (A(AA))A -5가지

$$P(4)=C(3)=\frac{1}{3+1}\binom{6}{3}=\frac{1}{4}\times 20=5$$

- 이진트리: 노드 3개로 구성



- 5개의 matrix를 곱하는 경우의 수-

  A(A-A-A-A) →5가지

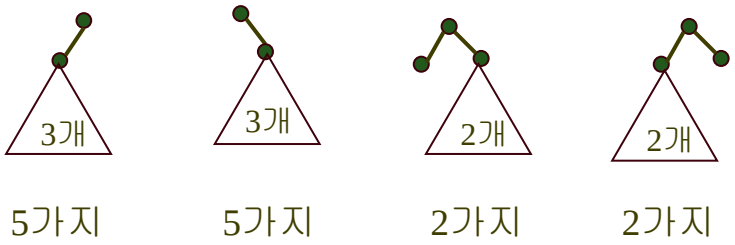
  (A-A-A-A)A →5가지

  (AA)(A-A-A) →2가지

  (A-A-A)AA → 2가지 총 14가지.

$$P(5)=C(4)=\frac{1}{4+1}\binom{8}{4}=\frac{1}{5}\times 70=14$$

- 노드 4개의 이진트리



- 최적의 원칙 적용 여부
  - n개의 행렬을 곱하는 최적의 순서는 n개 행렬의 부분집합을 곱하는 최적의 순서를 포함

  - Ex) 6 개의 행렬을 곱하는 최적의 순서가

    $A_1((((A_2A_3)A_4)A_5)A_6)$이라면 A_2에서 A_4까지 곱하는 최적의 순서는 반드시 $(A_2A_3)A_4$가 됨

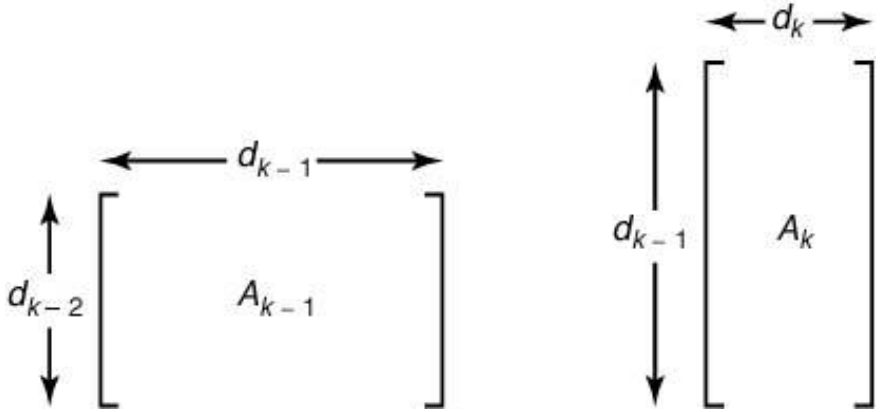
$$\begin{pmatrix} x & x \\ x & x \\ x & x \end{pmatrix} \times \begin{pmatrix} y & y & y & y \\ y & y & y & y \end{pmatrix} = \begin{pmatrix} z & z & z & z \\ z & z & z & z \\ z & z & z & z \end{pmatrix}$$

3×2 2×4 3×4

- z 하나를 계산하는데 필요한 곱셈은 2회
- 행렬곱셈을 위해 총 $3\times2\times4$회의 곱셈 필요

연쇄 행렬곱셈 동적계획식 설계전략

- A_k의 크기는 $d_{k-1} \times d_k$: d_{k-1} : 행(row)의 수, d_k: 열(column)의 수
- A_1의 행의 수는 d_0.



$M[i][j] =$ $i \le j$일 때 A_i부터 A_j까지의 행렬을 곱하는데 필요한 기본적인 곱셈의 최소 횟수 $\quad 1 \le i \le j \le n$

$$= \begin{cases} minimum_{i \le k \le j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j) & \text{if } i < j \\ M[i][i] = 0 \end{cases}$$

$$(A_i A_{i+1} \ldots A_k) \times (A_{k+1} \ldots . A_j)$$

C: $d_{i-1} \times d_k$ D: $d_k \times d_j$

$M[i][j] =$ $i \le j$일 때 A_i부터 A_j까지의 행렬을 곱하는데 필요한 기본적인 곱셈의 최소 횟수 $\quad 1 \le i \le j \le n$

$$= \begin{cases} minimum_{i \le k \le j-1}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j) & \text{if } i < j \\ M[i][i] = 0 \end{cases}$$

$(A_i A_{i+1} \ldots A_k) \times (A_{k+1} \ldots . A_j)$

C: $d_{i-1} \times d_k$ D: $d_k \times d_j$

재귀관계식의 적용 예

- 보기:

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

$$M[4][6] = \min_{4\le k\le 5}(M[4][4]+M[5][6]+4\times 6\times 8,\ M[4][5]+M[6][6]+4\times 7\times 8)$$
$$= min\,(0+6\times 7\times 8+4\times 6\times 8,\ 4\times 6\times 7+0+4\times 7\times 8)$$
$$= min\,(528,392) = 392$$

$M[i][j]$	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

대각 2

대각 1

진행순서

최종해는

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

$$M[1][6] = minimum_{1\le k\le 5}(M[1][k] + M[k+1][6] + d_0 d_k d_6)$$

$M[i][j]$	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

최종해

$$M[1][4] = minimum_{1\le k\le 3}(M[1][1] + M[2][4] + d_0 d_1 d_4,$$
$$M[1][2] + M[3][4] + d_0 d_2 d_4,$$
$$M[1][3] + M[4][4] + d_0 d_3 d_4)$$
$$= minimum(0 + 72 + 5\times 2\times 6,\ 30 + 72 + 5\times 3\times 6$$
$$64 + 0 + 5\times 4\times 6) = 132$$

최적 순서의 구축

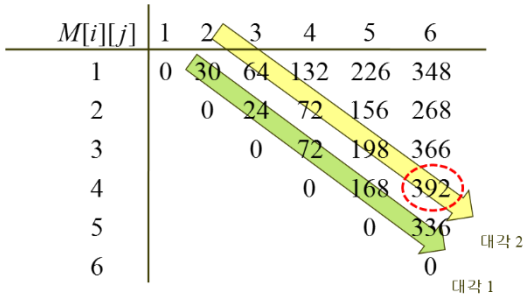
- 최적 순서를 얻기 위해서는 $M[i][j]$를 계산할 때 최소값을 주는 k값을 $P[i][j]$에 기억한다.
- 예: $P[2][5] = 4$인 경우의 최적 순서는 $((A_2 A_3)A_4) A_5$이다. 구축한 P는 다음과 같다.

$P[i][j]$	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

따라서, 최적 분해는 $(A_1((((A_2 A_3) A_4) A_5) A_6))$.

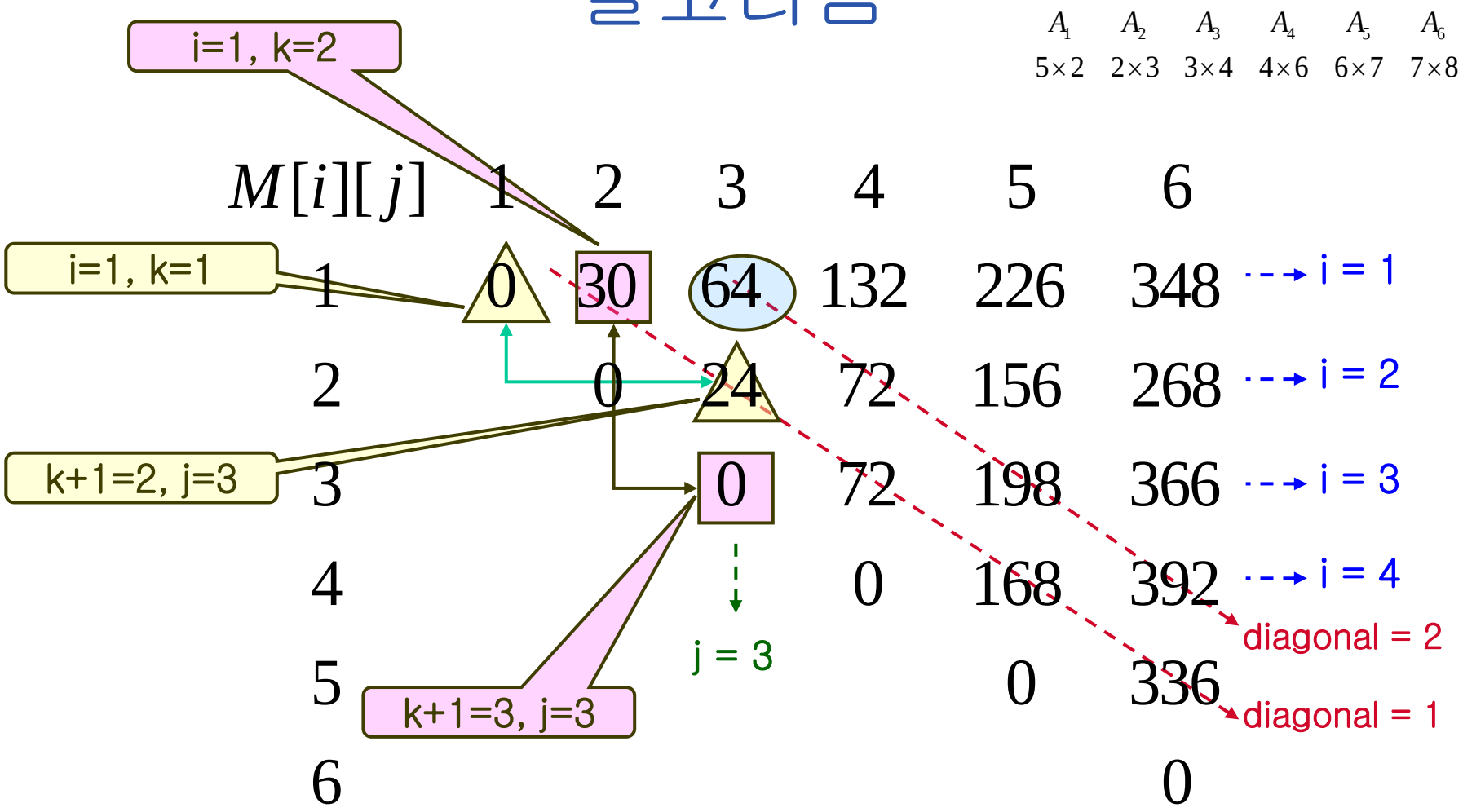
최소곱셈알고리즘

- 문제: n개의 행렬을 곱하는데 필요한 기본적인 곱셈의 횟수의 최소치를 결정하고, 그 최소치를 구하는 순서를 결정하라.
- 입력: 행렬의 개수 n, 배열 $d[i\text{-}1] \times d[i]$는 i번째 행렬의 규모를 나타낸다.
- 출력:
  - ✓ 기본적인 곱셈의 횟수의 최소치를 나타내는 *minmult*;
  - ✓ 최적의 순서를 구할 수 있는 배열 P, P 는 $1 \ldots n\text{-}1$ by $1..n$. 여기서 $P[i][j]$는 행렬 i부터 j까지가 최적의 순서로 갈라지는 기점을 나타낸다.




```
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_{i<=k<=j-1}(M[i][k]+M[k+1][j]+
                        d[i-1]*d[k]*d[j]);
            P[i][j] = 최소치를 주는 k의 값
            }
    return M[1][n];
}
```

최소곱셈(Minimum Multiplication) 알고리즘



(1,3)을 계산할 때: 64=min{0+24+5×2×4, 30+0+5×3×4}

최소곱셈 알고리즘의 모든 경우분석

- 단위연산: 각 k값에 대하여 실행된 명령문(instruction), 여기서 최소값인 지를 알아보는 비교문도 포함한다.
- 입력크기: 곱할 행렬의 개수 n
- 분석: $j = i + diagonal$ 이므로,
  - ✓ k-루프를 수행하는 횟수 = $(j-1)-i+1 = i+diagonal-1-i+1 = diagonal$
  - ✓ for-i 루프를 수행하는 횟수 = $n - diagonal$
  - ✓ 따라서

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

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

$\sum_{diagonal=1}^{n-1}$

(*n-diagonal*)

j-1-i+1=j-i
=i+diagonal-i

```
int minmult(int n, const int d[], index P[][]) {
    for(i=1; i <= n; i++)    ----
    for(diagonal = 1; diagonal <= n-1; diagonal++)
      for(i=1; i <= n-diagonal; i++) {
                j = i + diagonal;
                M[i][j]= minimum_{i<=k<=j-1}(        );
                         -----      }
    }
```

- Godbole(1973)

최적의 해를 주는 순서의 출력

- 문제: n개의 행렬을 곱하는 최적의 순서를 출력하시오.
- 입력: n과 P
- 출력: 최적의 순서

```
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 << ")";
        }
}
```

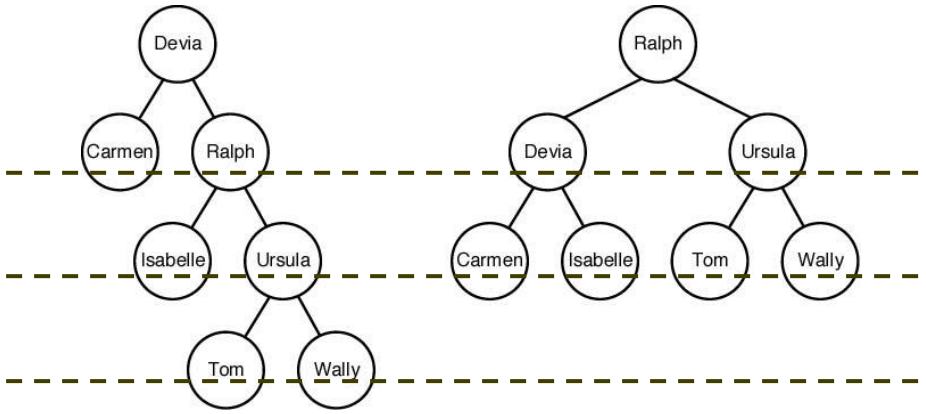
- order(1,6)은 (A1((((A2A3)A4)A5)A6)) 을 출력
- $T(n) \in \Theta(n)$.

다른 알고리즘

- Yao(1982) - $\Theta(n^2)$
- Hu and Shing(1982, 1984) - $\Theta(n \lg n)$

최적 이진검색 트리

- left(right) subtree: 이진트리에서 어떤 마디의 왼쪽(오른쪽)자식마디가 뿌리마디가 되는 부분트리
- 이진검색트리(binary search tree): 순서가능집합(ordered set)에 속한 아이템(키)으로 구성된 이진 트리
  - ✓ 각 마디는 하나의 키만 가지고 있다
  - ✓주어진 마디의 왼쪽(오른쪽) 부분트리에 있는 키는 그 마디의 키보다 작거나(크거나) 같다.



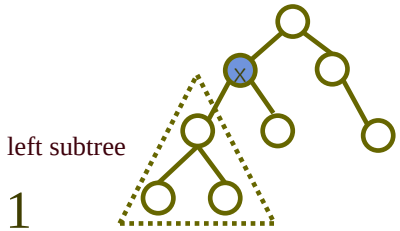
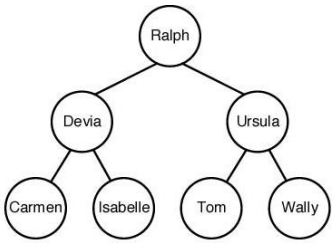


그림 3.10 두 이진검색 트리의 예

- Optimal binary search tree: 키를 찾는데 걸리는 평균시간이 최소가 되도록 구축된 트리
- Optimal binary search tree problem : 각 키를 찾을 확률이 주어져 있을 때(모두 같지는 않음을 가정) 키를 찾는데 걸리는 평균시간이 최소가 되도록 이진트리를 구축하는 문제



- data type

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

typedef nodetype* node_pointer;
```

- 이진검색트리의 검색

```
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;
          else
              p = p->right;
}
```

- 키의 검색시간

  - depth(key)+1

- notation: $Key_1, Key_2, \ldots, Key_n$: n개의 키.

  p_i : Key_i가 검색키일 확률

  c_i : 주어진 트리에서 키 Key_i를 찾는데 필요한 비교횟수

평균검색시간 --- $\sum_{i=1}^{n} c_i p_i$

(예 3.7) $p_1 = 0.7$, $p_2 = 0.2$, $p_3 = 0.1$

평균검색시간 트리 (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

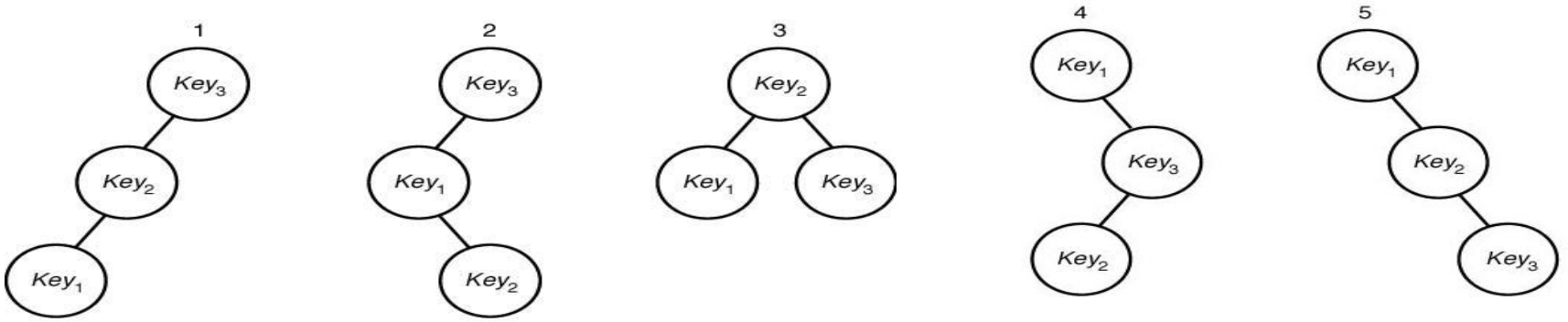
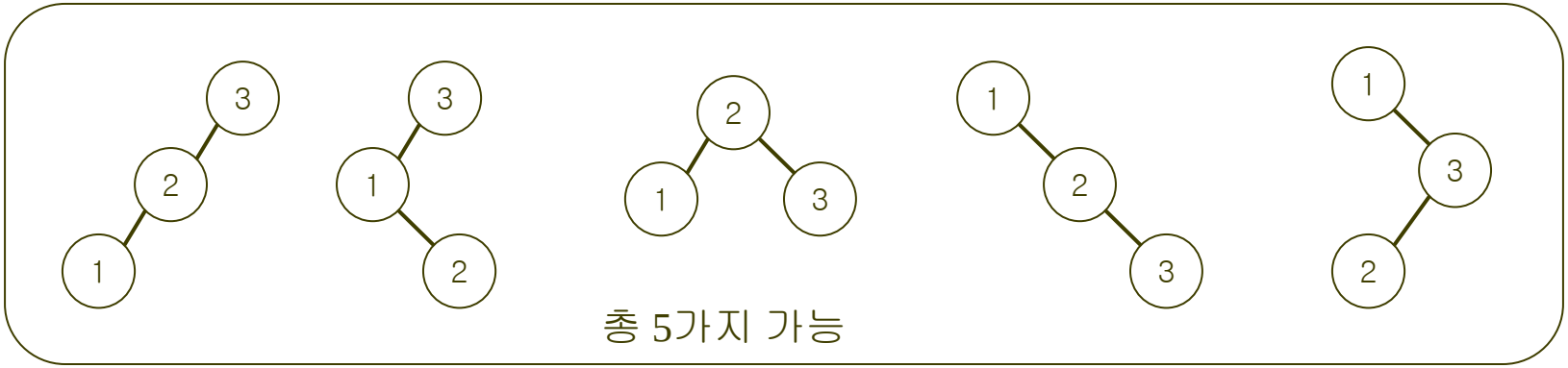


그림 3.11

- n=3, 키가 1, 2, 3 인 경우의 가능한 이진검색트리



- n개의 키가 있는 경우 이진검색트리의 개수

$$C_n = \frac{1}{n+1}\binom{2n}{n} \quad : n\text{th Catalan number}$$

$$= \frac{4^n}{\sqrt{\pi} n^{3/2}}(1+O(1/n))$$

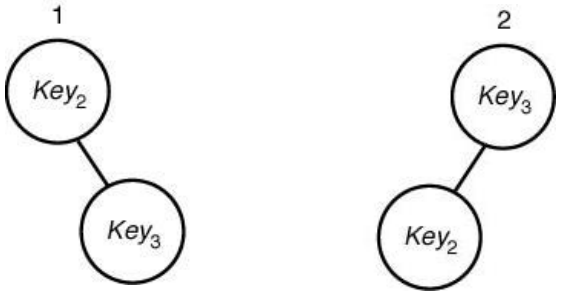
- 모든 경우의 이진검색트리를 구축하여 이 중 최적을 선택하는 방법은 비현실적 → 동적계획법을 이용한 효과적인 방법 필요

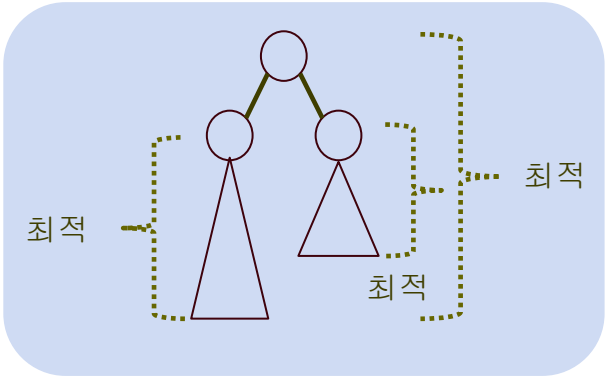
동적계획법

- Key_i 부터 Key_j 까지 키를 포함하는 최적이진검색트리는 $\sum_{m=i}^{j} c_m p_m$ 를 최소화해야 함.
- 검색시간 최적값을 A[i][j]로 표시. A[i][i]=p_i
- (예 3.8)

  p_1=0.7, p_2=0.2, p_3= 0.1일 때 A[2][3]?

  1. $1(p_2)+2(p_3) = 1(0.2) + 2(0.1) = 0.4$
  2. $2(p_2)+1(p_3) = 2(0.2) + 1(0.1) = 0.5$





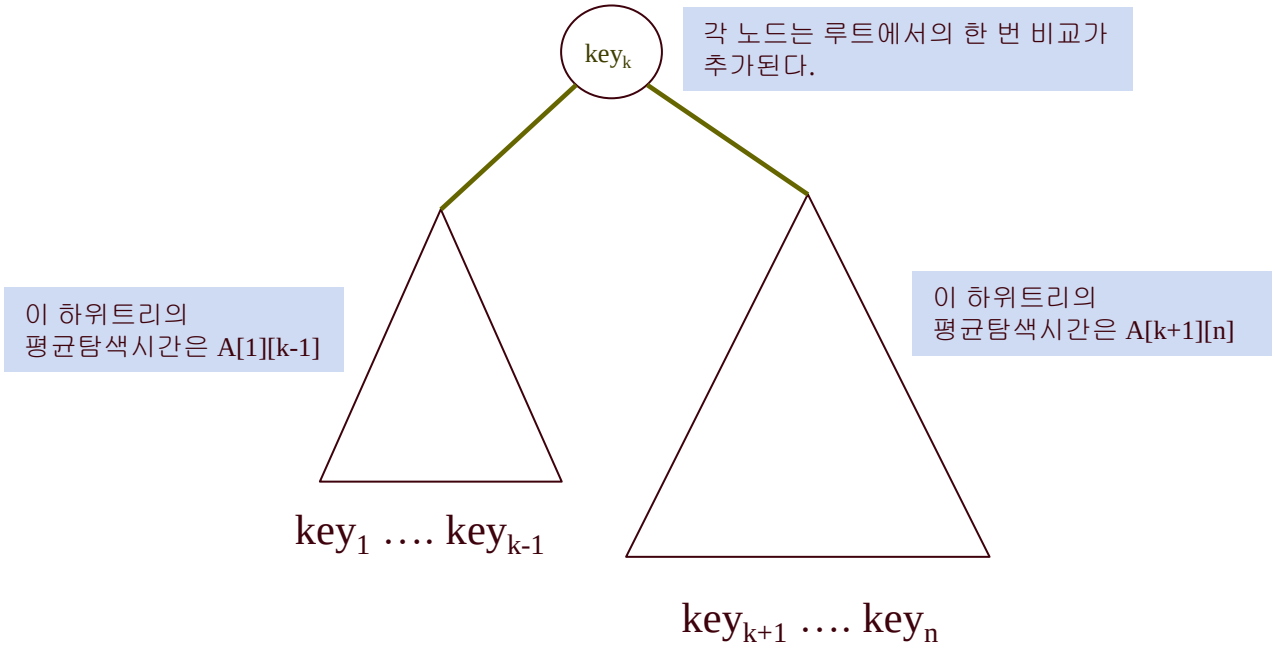
- 최적트리의 부분트리는 그 부분트리안에 있는 키들에 대해서 반드시 최적이어야 한다. → 최적의 원칙 적용

- 트리 k: n개의 키가 있을 때 이 중 k번째 키가 루트가 되는 트리
- 트리 k의 검색시간 $A[1][n]$

$$
\begin{aligned}
&= A[1][k-1] && /* \text{왼쪽 부분트리에서평균시간} \\
&\quad + p_1 + \ldots + p_{k-1} && /* \text{뿌리에서 비교하는데드는 추가시간} \\
&\quad + p_k && /* \text{뿌리를 검색하는 평균시간} \\
&\quad + A[k+1][n] && /* \text{오른쪽 부분트리에서 평균시간} \\
&\quad + p_{k+1} + \ldots + p_n && /* \text{뿌리에서 비교하는데드는 추가시간} \\
&= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m
\end{aligned}
$$

- 따라서 최적의 이진검색트리 평균검색시간은

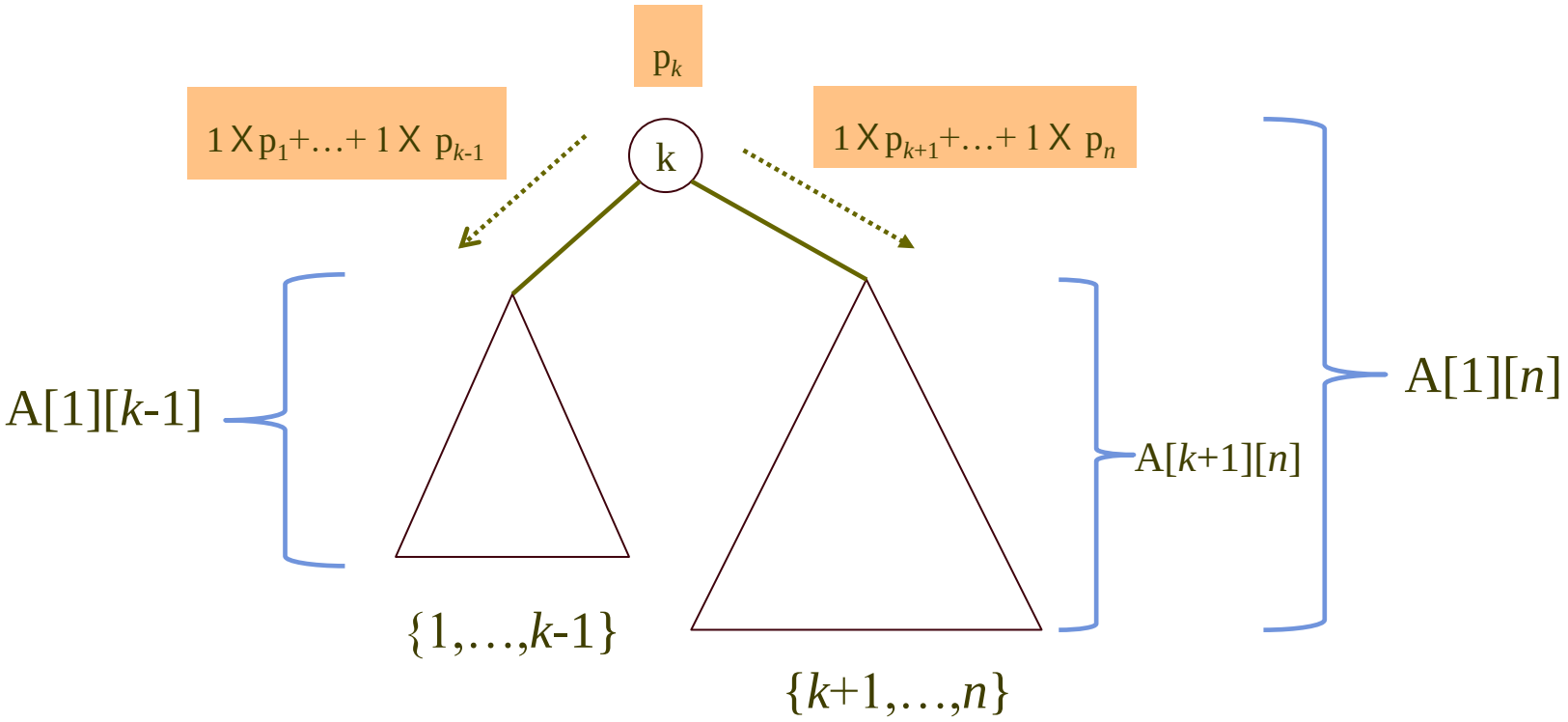
$$A[1][n] = \min_{1 \le k \le n} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$



- 일반화 시키면

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

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



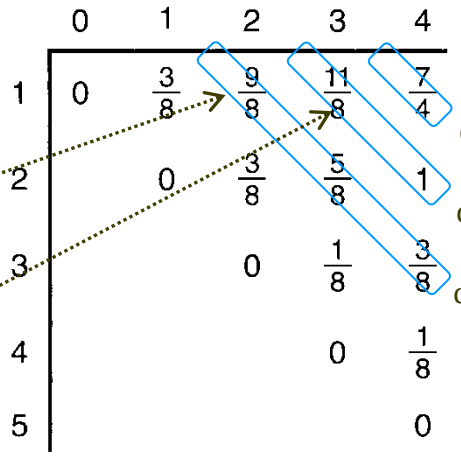
$$
\begin{aligned}
A[1][n] &= A[1][k-1] && /* \text{왼쪽 부분트리에서평균시간} \\
&+ p_1 + \ldots + p_{k-1} && /* \text{뿌리에서 비교하는데드는 추가시간} \\
&+ p_k && /* \text{뿌리를 검색하는 평균시간} \\
&+ A[k+1][n] && /* \text{오른쪽 부분트리에서 평균시간} \\
&+ p_{k+1} + \ldots + p_n && /* \text{뿌리에서 비교하는데드는 추가시간} \\
&= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m
\end{aligned}
$$

- 문제: 최적이진검색트리 구하기

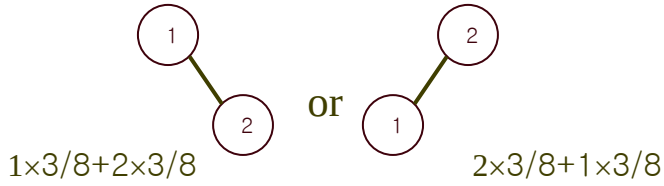
```
void optsearchtree(int n, const float p[], float& minavg
                   index R[][] ){
   index i, j, k, diagonal;
   float A[1..n+1][0..n];

   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_{i≤k≤j}(A[i][k-1]+A[k+1][j]) + Σ_{m=i}^{j} p_m ;
             R[i][j] = 최소값을 주는 k의 값
             }
   minavg = A[1][n];
}
```

- 예 3.9

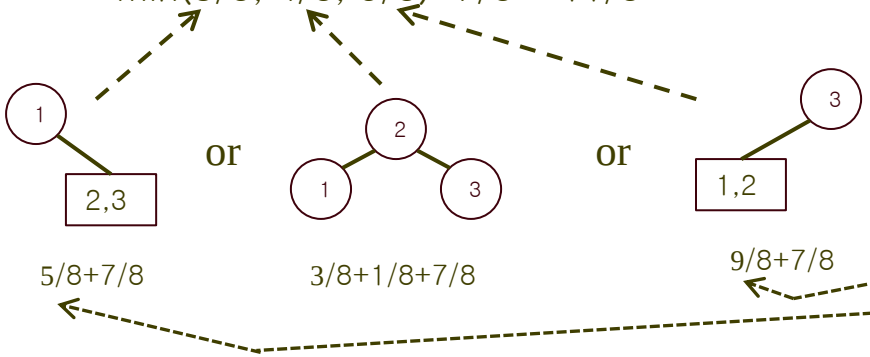
$p_1=3/8, p_2=3/8, p_3=1/8, p_4=1/8$

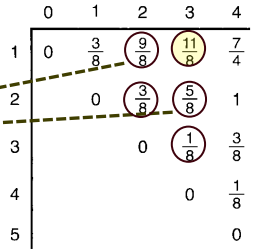


A[1][2] = min(A[1][0]+A[2][2],A[1][1]+A[3][2])+6/8
= min(3/8,3/8)+6/8 = 9/8



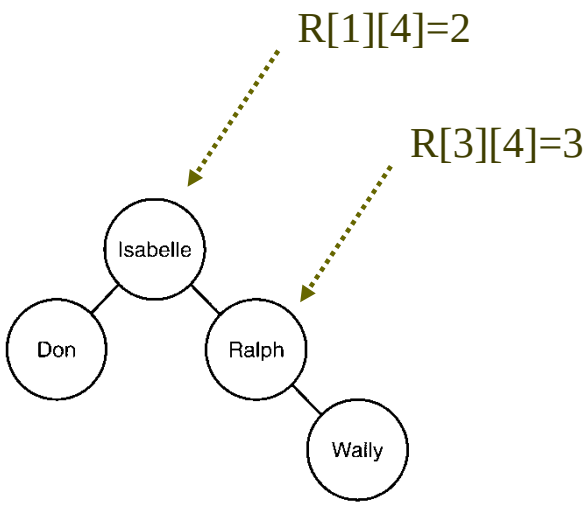
A[1][3] = min(A[1][0]+A[2][3], A[1][1]+A[3][3], A[1][2]+A[4][3])
+7/8
= min(5/8, 4/8, 9/8)+7/8 = 11/8





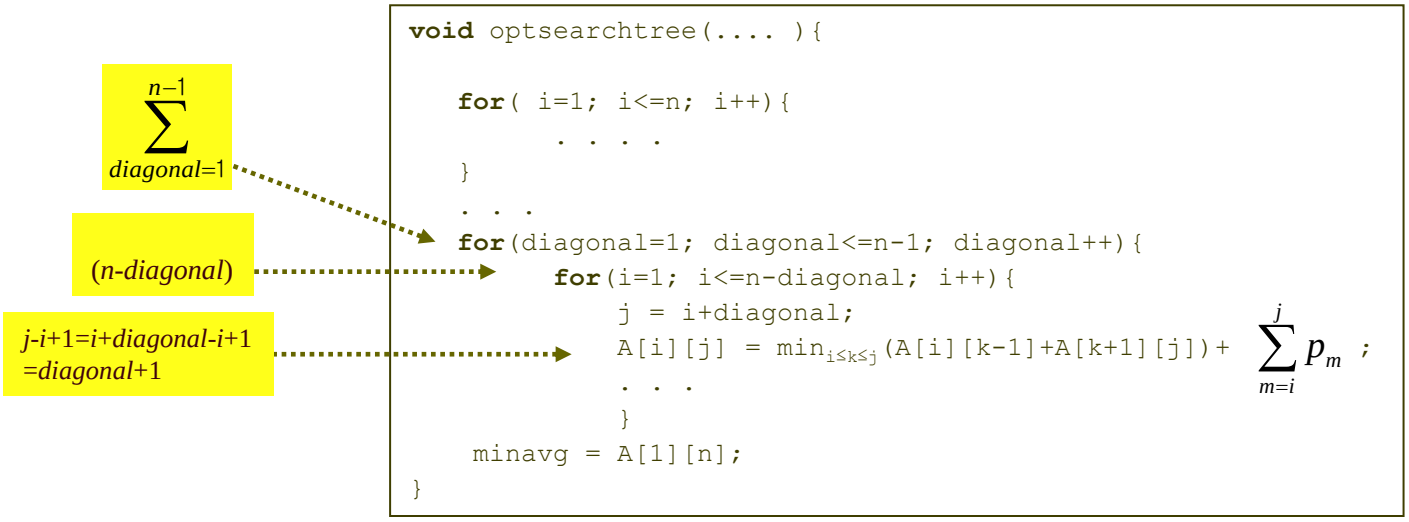
	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R



- 시간복잡도 분석
  - ✓단위연산: 첨자 k에 대한 문장
  - ✓입력크기: 키의 개수 n
  - ✓행렬의 최소곱셈의 복잡도 분석과 유사

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



- 문제: 최적이진검색트리 구축

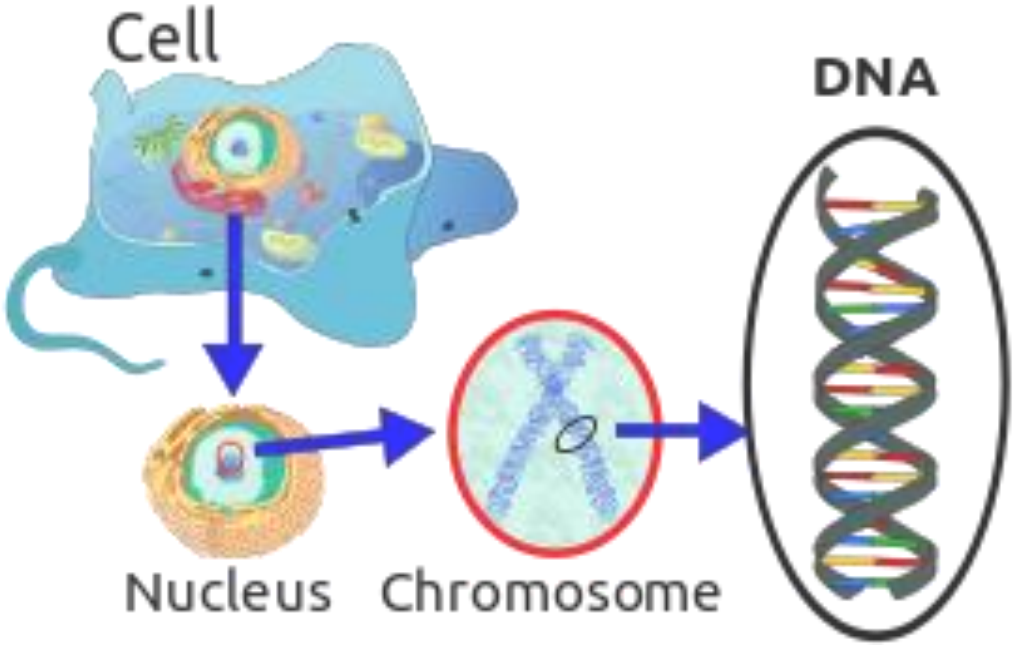
```
node_pointer tree (index i, index j) {
    index k;
    node_pointer p;

    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;
    }
}
```

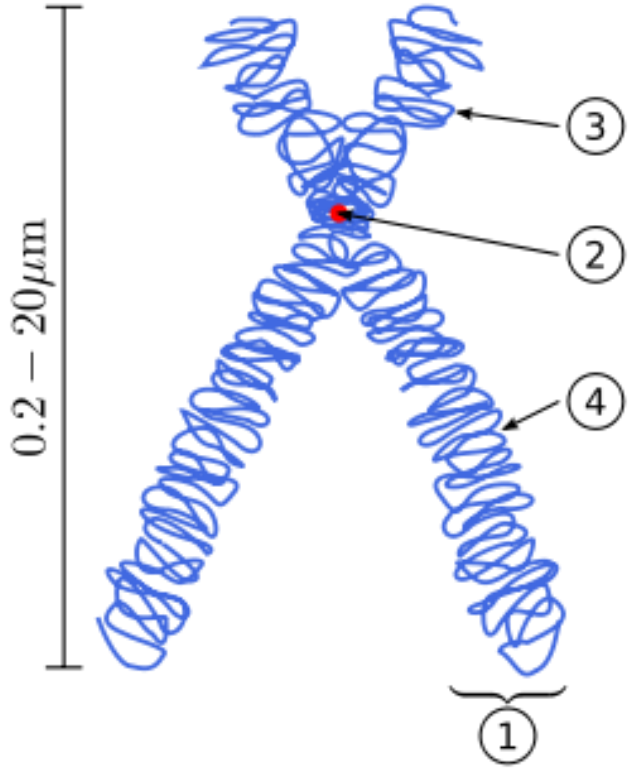
- root = tree(1, n)

- Gilbert와 Moore(1959)
- 향상된 방법으로 Yao(1982) $\Theta(n^2)$

DNA 서열 맞춤(sequence alignment)

- 분자유전학(molecular genetics)의 한 분야인 동족(homologous) DNA 서열 맞춤문제를 동적계획법으로 해결하는 방법 소개
- 먼저 divide-and-conquer 방법을 설명하고, 동적계획법 방법을 설명한다.



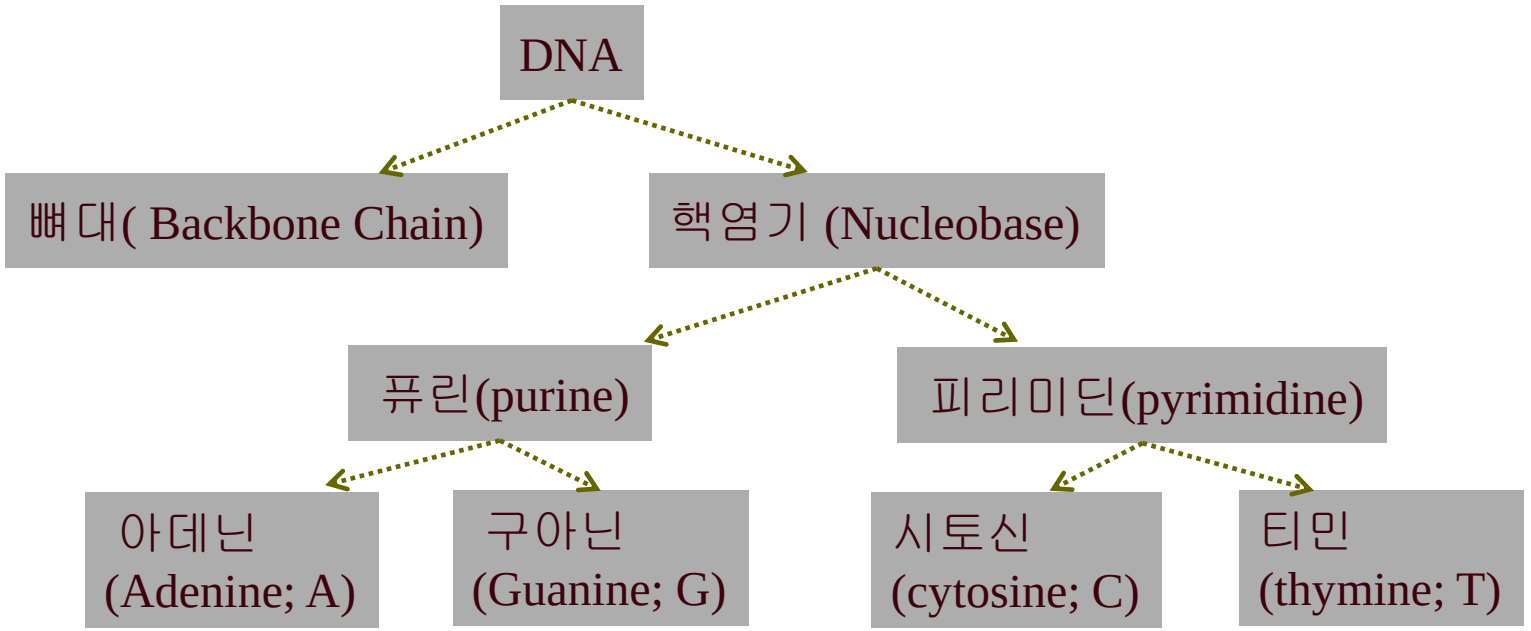


염색체(chromosome)

from wikipedia

DNA:

- 이중 나선구조를 이루는 뼈대 (Backbone chain)와 핵염기 (Nucleobase)로 구성
- 핵염기에는 퓨린(purine)과 피리미딘(pyrimidine)의 두 가지 종류가 있으며, 퓨린에는 다시 아데닌(Adenine; A)과 구아닌(Guanine; G)의 두 가지가 존재하고, 피리미딘에는 시토신(cytosine; C), 티민(thymine; T)이 존재



- A는 T와, C는 G와 연결된다.

DNA 섹션(section)을 간단히 다음의 문자열로 표시한다.

사이트(site): 염기쌍이 위치한 장소

- 돌연변이
  - ✓ 교환 돌연변이(substitution mutation): 핵염기가 서로 바뀐다
  - ✓ 삽입 돌연변이(insertion mutation): 핵염기쌍이 추가된다.
  - ✓ 제거 돌연변이(deletion mutation): 핵염기쌍이 제거된다.

- 동족서열(homologous sequence)

  - ✓ 하나의 종에서 출발하여, 교환, 삽입, 삭제돌연변이에 의해 서열이 달라지게 된다.
  - ✓ 달라진 두 종에서 추출한 염기서열을 동족서열이라고 한다.
  - ✓ 이 두 서열이 얼마나 다른 지를 확인해서, 종족간의 거리를 측정한다.
  - ✓ 두 개의 염기서열을 어떻게 맞추어야(alignment) 할까?

염기쌍의 한 쪽만 표시

(예) 동족서열

A A C A G T T A C C

T A A G G T C A

맞춤(alignment)방법1

틈

- A A C A G T T A C C

T A A - G G T - - C A

5개의 일치, 4개의 틈, 2개의 불일치

맞춤(alignment)방법2

A A C A G T T A C C

T A - A G G T - C A

5개의 일치, 2개의 틈, 3개의 불일치

- (틈, gap): 위치에 제거가 일어 났거나, 상대편 서열에 삽입이 일어났다는 것을 의미한다.

- 어느 방법이 더 좋은 방법인가?
- 틈과 불일치에 대한 손해를 정의한 후, 맞춤방법에 대한 총 비용을 계산한다.
- (예) 손해 정의 - 틈: 1, 불일치:3

방법1 4개의 틈, 2개의 불일치

총 비용: 10

방법2 2개의 틈, 3개의 불일치

총 비용: 11

- 틈과 불일치에 대한 손해를 얼마로 하는냐에 따라 총 비용이 달라진다
- 교재에서는 틈:2 불일치:1의 손해로 정의

- DNA 서열 맞춤 문제:

두 개의 서열을 최소비용으로 맞추는 방법을 찾는다.

서열을 배열로 표시

x[ ]	0	1	2	3	4	5	6	7	8	9
	A	A	C	A	G	T	T	A	C	C

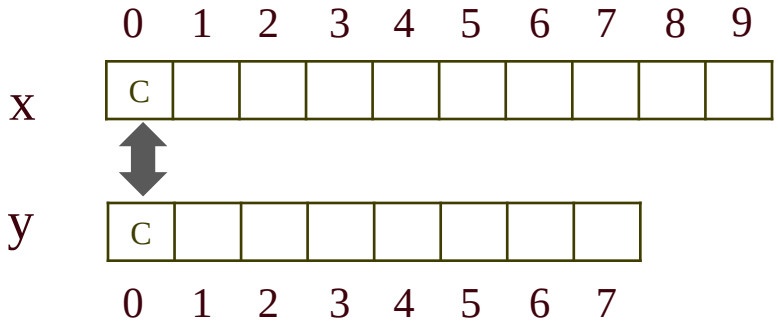
y[ ]	0	1	2	3	4	5	6	7
	T	A	A	G	G	T	C	A

- opt(i,j): 부분 서열 x[i,...,9]와 y[j,...,7]의 최적 맞춤 비용
- opt(0,0): 부분 서열 x[0,...,9]와 y[0,...,7]의 최적 맞춤 비용, 즉, 우리가 구하려는 최종 비용

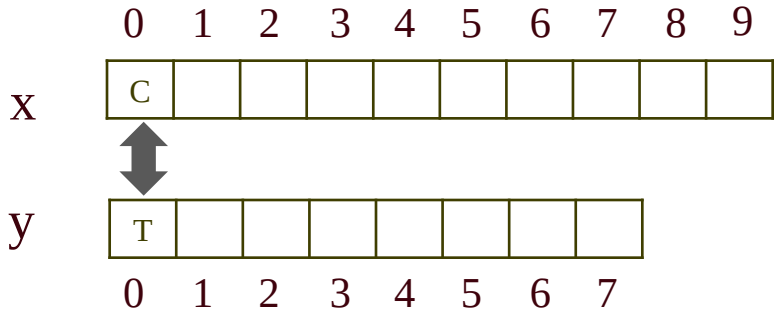
틈: 2 불일치: 1

- x[0]과 y[0]에 대해

1. x[0]과 y[0]을 맞춘다.
   - ✓ 만일 x[0]=y[0] 이면, 첫 맞춤에서는 손해가 없다.

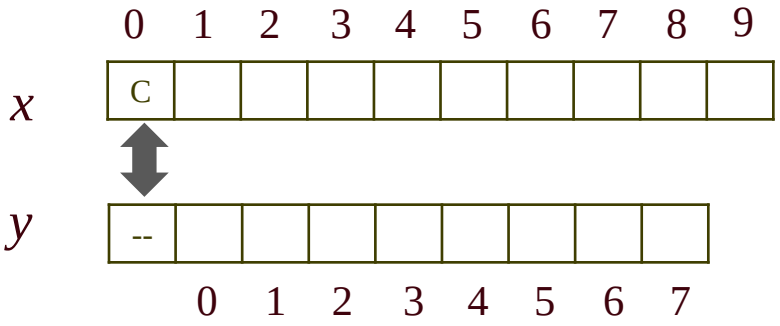


   - ✓ 만일 x[0]≠y[0] 이면, 첫 맞춤에서는 손해는 1이다.

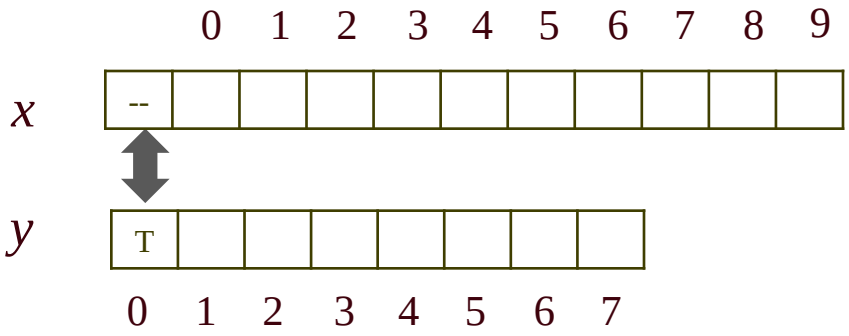


틈: 2 불일치: 1

2. x[0]는 틈과 맞춘다, 첫 맞춤 사이트의 손해는 2.

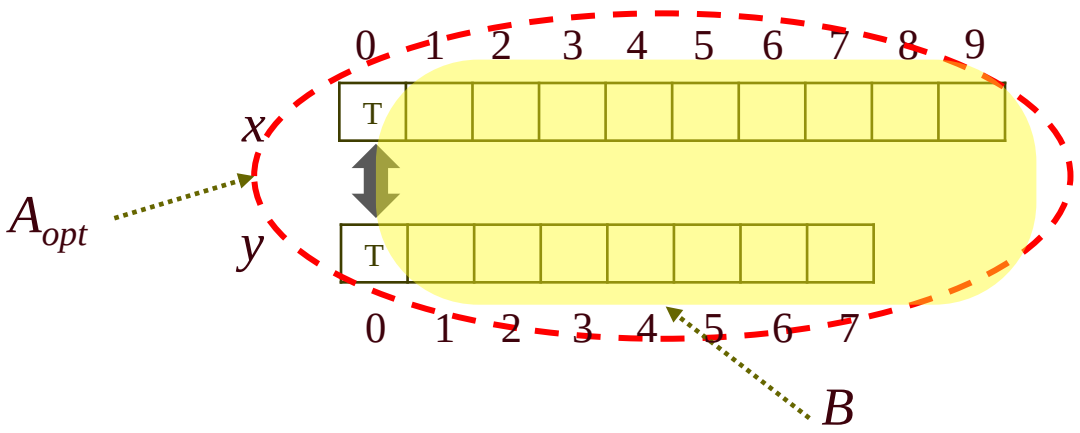


3. y[0]는 틈과 맞춘다, 첫 맞춤 사이트의 손해는 2.



Principle of Optimality

- $x[0,\ldots,9]$와 $y[0,\ldots,7]$의 최적 맞춤 A_{opt}에 $x[0]$과 $y[0]$이 맞추어져 있다고 가정.
- 이 맞춤 내부에는 $x[1,\ldots,9]$와 $y[1,\ldots,7]$에 대한 맞춤 B가 포함되어 있다.
- 만일 B가 이 두 부분 서열의 최적 맞춤이 아니라고 하자.
- 그러면, 이 두 부분서열에 대해 비용이 더 작은 맞춤 C가 있어야 한다.
- A_{opt}에 부분 맞춤 C를 연결하면, A_{opt} 보다 비용이 더 작은 $x[0,...,9]$와 $y[0,...,7]$의 맞춤 방법이 존재한다.
- 그러나 A_{opt} 가 최적이라고 하였기 때문에, C는 존재할 수 없다.
- 따라서 B는 $x[1,...,9]$와 $y[1,...,7]$에 대한 최적 맞춤이어야 한다.



Principle of Optimality

- 같은 방법으로, $x[0]$에 틈을 맞출 경우, $y[0]$에 틈을 맞출 경우도 같은 방법으로 최적의 원칙을 증명할 수 있다.

전 예의 최적의 맞춤($x[0,\ldots,9]$와 $y[0,\ldots,7]$)

x[ ]	0	1	2	3	4	5	6	7	8	9
	A	A	C	A	G	T	T	A	C	C

y[ ]	0	1		2	3	4	5		6	7
	T	A	-	A	G	G	T	-	C	A

$x[1,\ldots,9]$와 $y[1,\ldots,7]$ 최적 맞춤

x[ ]	1	2	3	4	5	6	7	8	9
	A	C	A	G	T	T	A	C	C

y[ ]	1		2	3	4	5		6	7
	A	-	A	G	G	T	-	C	A

- $x[0,\ldots,9]$와 $y[0,\ldots,7]$)의 최적맞춤은 $x[1,\ldots,9]$와 $y[1,\ldots,7]$의 최적맞춤을 포함한다.

틈: 2 불일치: 1

opt(*i*,*j*): 부분 서열 *x*[*i*,…,9]와 *y*[*j*,…,7]의 최적 맞춤 비용

$$opt(0,0)=\min(opt(1,1)+penalty,\ opt(1,0)+2,\ opt(0,1)+2)$$

- *penalty*=0 if x[0]=y[0]
- *penalty*=1 if x[0]≠y[0]

① ② ③

① *penalty*=1

x[ ]	0	1	2	3	4	5	6	7	8	9
	A	A	C	A	G	T	T	A	C	C

y[ ]	0	1	2	3	4	5	6	7
	T	A	A	G	G	T	C	A

② 틈 1개에 의해 비용 2 발생

x[ ]	0	1	2	3	4	5	6	7	8	9
	A	A	C	A	G	T	T	A	C	C

y[ ]		0	1	2	3	4	5	6	7
	-	T	A	A	G	G	T	C	A

③ 틈 1개에 의해 비용 2 발생

x[ ]		0	1	2	3	4	5	6	7	8	9
	-	A	A	C	A	G	T	T	A	C	C

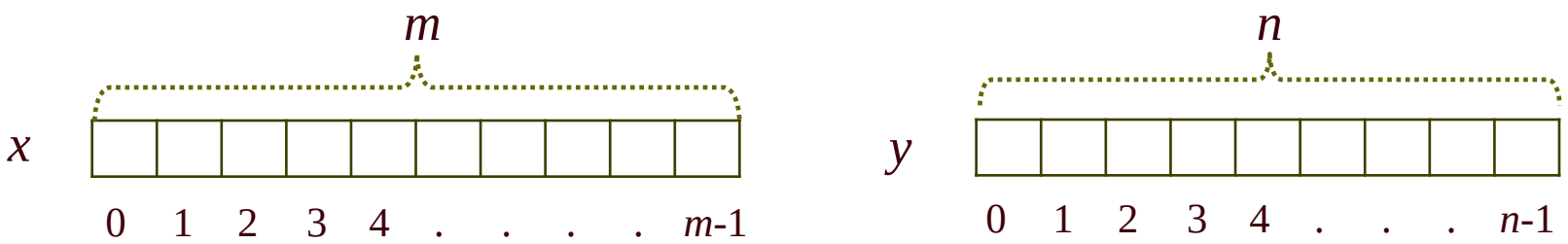
y[ ]	0	1	2	3	4	5	6	7
	T	A	A	G	G	T	C	A

- 일반식은

$$opt(i,j)=\min(opt(i+1,j+1)+penalty,\ opt(i+1,j)+2,\ opt(i,j+1)+2)$$

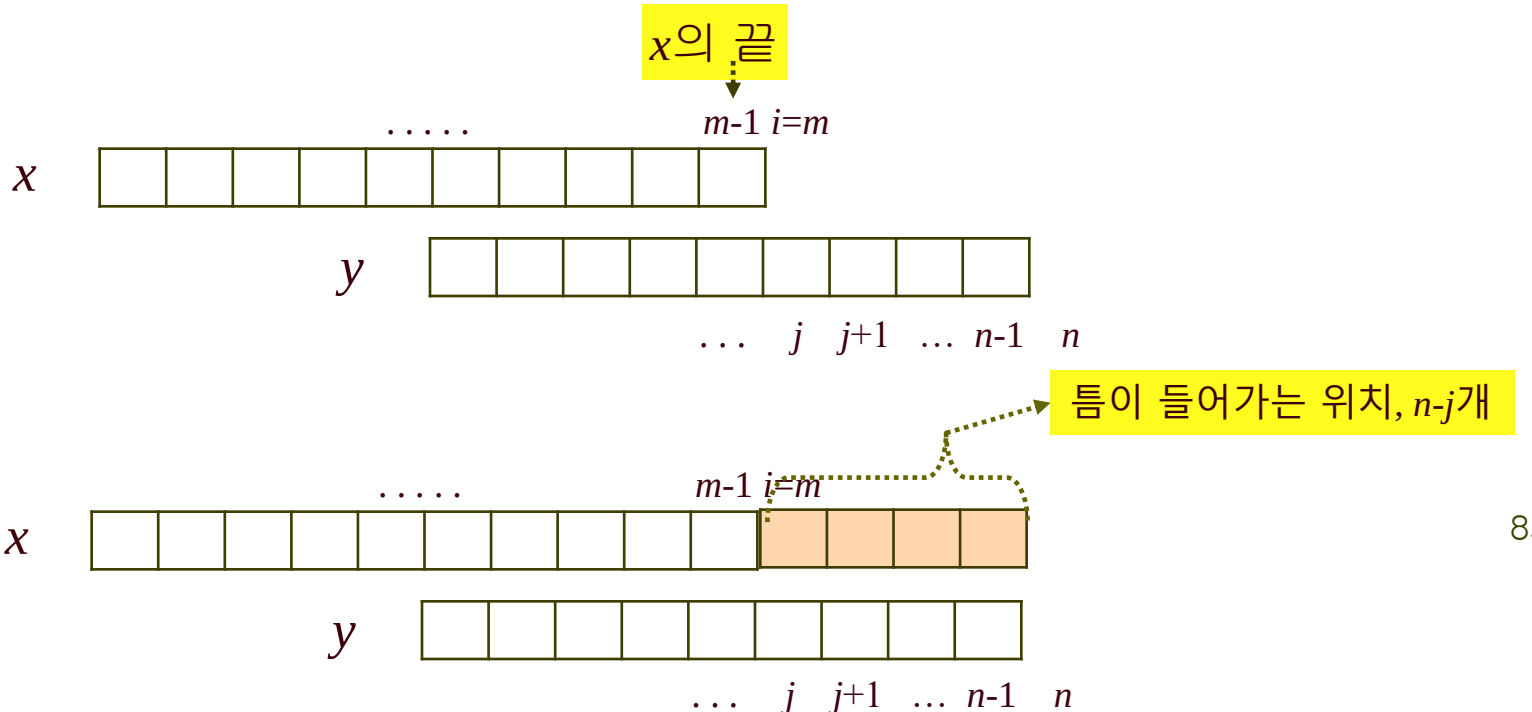
- 재귀알고리즘을 만들기 위한 종료조건

서열 x의 크기를 m, 서열 y의 크기를 n이라고 하면,

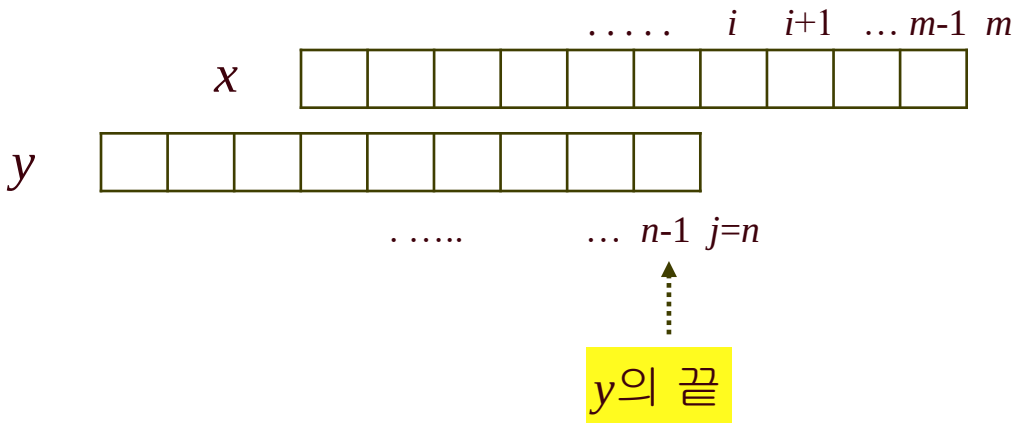


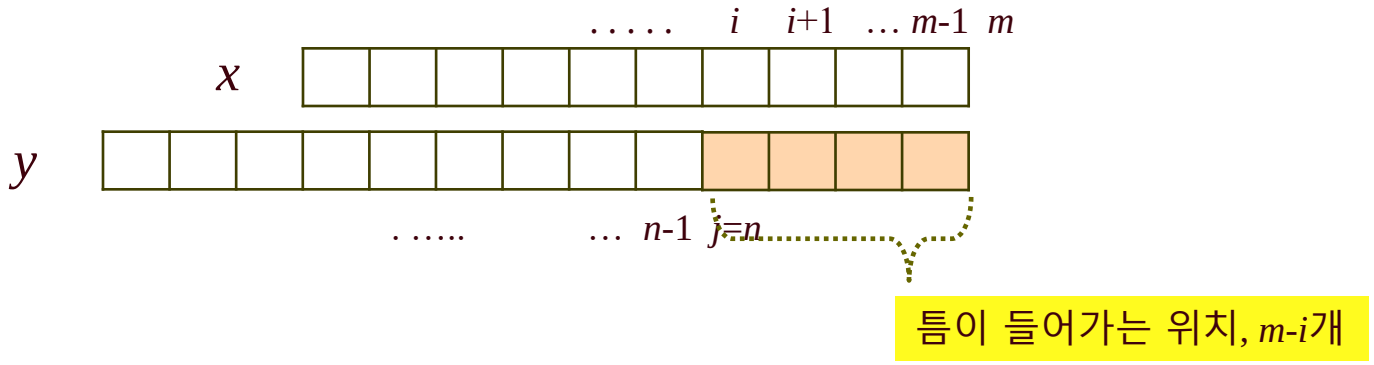
(1) 서열 x의 끝지점을 지났고($i=m$), 서열 y의 j지점($j<n$)에 있다면,

틈을 $n-j$개 삽입해야 한다. 따라서, $opt(m,j)=2(n-j)$



(2) 서열 y의 끝지점을 지났고($j=n$), 서열 x의 i지점($i<m$)에 있다면, 틈을 $m-i$개 삽입해야 한다. 따라서, $opt(i,n)=2(m-i)$





- 분할정복방법의 서열 맞춤 방법

  문제: 두 개의 동족 DNA 서열의 최적 맞춤을 구하시오

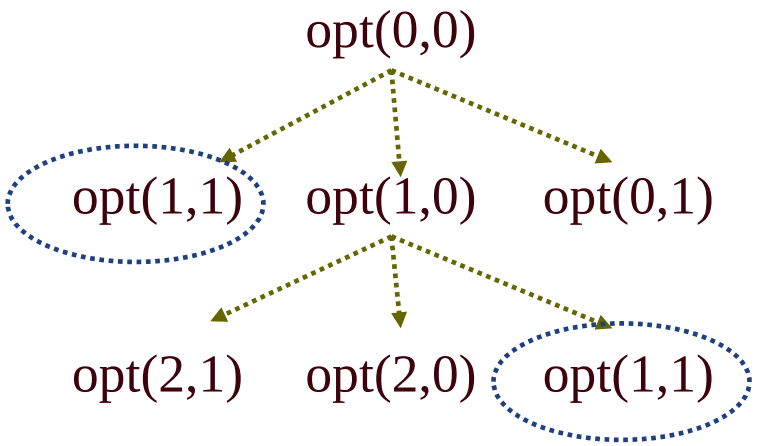
  입력: 길이 m인 DNA 서열 x와 길이 n인 DNA 서열 y, 서열은 배열로 표현

  출력: 두 서열의 최적 맞춤의 비용

```
int opt(int i, int j) {
  if(i==m)
     opt_val = 2(n-j);
  else if(j==n)
     opt_val = 2(m-i);
  else {
     if (x[i] == y[j])
        penalty = 0;
     else
        penalty = 1;
     opt_val = min(opt(i+1,j+1)+penalty, opt(i+1,j)+2, opt(i,j+1)+2)
     }
  return opt_val;
  }
```

  - optimal_cost = opt(0,0)

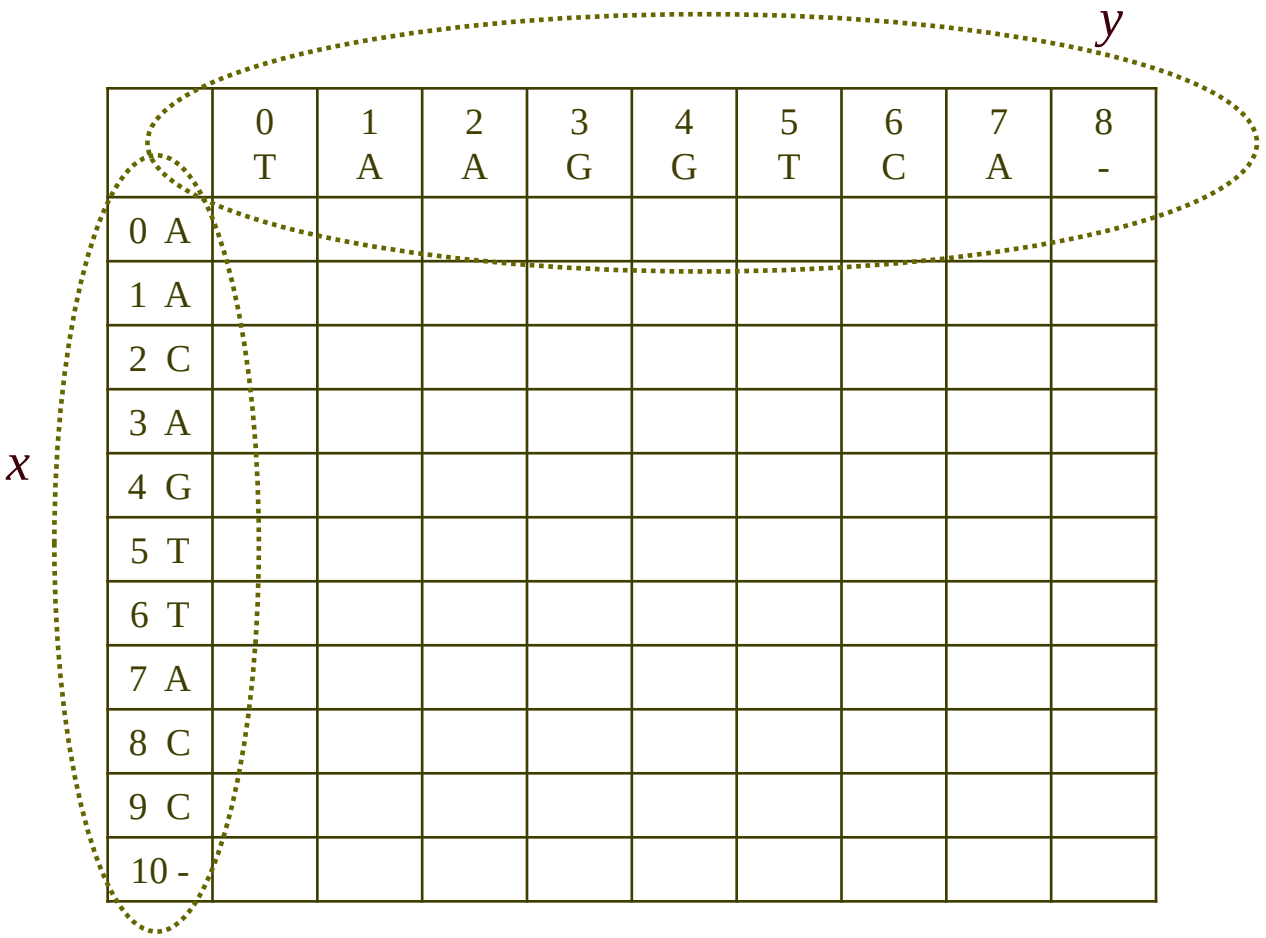
- 이 알고리즘은 매우 비효율적이다.
- 같은 함수를 중복해서 호출한다.
  - ✓ opt(0,0)은 opt(1,1), opt(1,0), opt(0,1)을 호출
  - ✓ opt(1,0)은 opt(2,1), opt(2,0), opt(1,1)을 호출



- 피보나찌수열을 구하는 분할정복법의 알고리즘 비효율성과 유사

동적계획법을 이용한 최적맞춤 방법 찾기

- m+1, n+1 크기의 2차원 배열. m=10, n=8
- 마지막 행, 마지막 열에 '– ' 틈 문자 추가

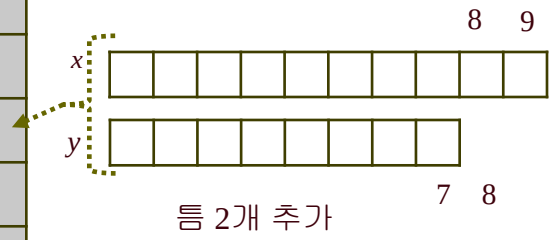


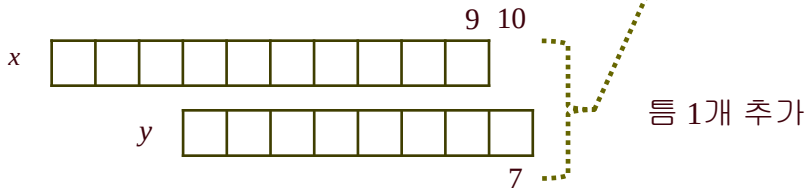
	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
0 A									
1 A									
2 C									
3 A									
4 G									
5 T									
6 T									
7 A									
8 C									
9 C									
10 -									

1. 틈 행과 틈 열을 채워 넣는다

$opt(10,j)=2(8-j)$, $opt(i,8)=2(10-i)$, 틈의 비용=2

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





$opt(i,j)=\min(opt(i+1,j+1)+penalty,\ opt(i+1,j)+2,\ opt(i,j+1)+2)$

2. 우측 아래 부터 대각원소들을 채워 넣는다

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

x[9]와 y[7]의 일치여부에 의한 penalty

① $opt(9,7)= \min\{opt(9+1,7+1)+penalty,\ opt(9+1,7)+2,\ opt(9,7+1)+2\}$

$= \min\{0+1,\ 2+2,\ 2+2\}$

$=1$

C과 A로 다르므로

- 테이블의 모든 값

j i	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
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

$opt(i,j)=\min(opt(i+1,j+1)+penalty,\ opt(i+1,j)+2,\ opt(i,j+1)+2)$

- 최적서열맞춤을 찾아가는 방법
  - ✓ opt(0,0)에서 출발하여, 3가지 가능성을 조사

i \ j	0 T	1 A
0 A	7	8
1 A	6	6

(1) [0][0] 선택

(2) 경로에 넣을 둘째 항목을 찾는다

1) [0][1]을 검사

opt(0,1)+2=8+2=10 ≠7

2) [1][0]을 검사

opt(1,0)+2=6+2=8 ≠7

3) [1][1]을 검사

opt(1,1)+1=6+1=7. 찾음

([0][0]이 A, T로 서로 다르므로 penalty=1)

i \ j	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
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

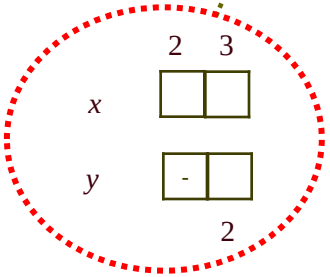
✓최적서열맞춤 해는 해당 cell을 음영으로 표시했다.

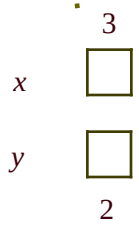
- 배열값을 채워 넣을 때 어디로 부터 min을 구했는지의 정보를 저장할 수 있다.

j / i	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
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

- $\min\{6+1, 4+2, 8+2\}$
- $\min\{4+0, 4+2, 5+2\}$

j i	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
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





x[2]와 y[2]를 맞추는 방법은 y에 – 를 넣은 것이 최적

- 경로를 찾은 후 맞춤된 서열을 구성하는 방법

1. 최적배열의 오른쪽 맨 아래 구성에서 시작하여 표시해둔 경로를 따라간다.
2. 배열의 [*i*][*j*]칸에 도달하기 위해서 대각선으로 이동할 때 마다, *i*째 행에 해당하는 문자를 *x*서열에 넣고, *j*째 열에 해당하는 문자를 *y*서열에 넣는다.
3. 배열의 [*i*][*j*]칸에 도달하기 위해서 위로 이동할 때 마다, *i*째 행에 해당하는 문자를 *x*서열에 넣고, 틈 문자를 *y*서열에 넣는다.
4. 배열의 [*i*][*j*]칸에 도달하기 위해서 왼쪽으로 이동할 때 마다, *j*째 열에 해당하는 문자를 *y*서열에 넣고, 틈 문자를 *x*서열에 넣는다.

j / i	0 T	1 A	2 A	3 G	4 G	5 T	6 C	7 A	8 -
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

- 최적해

A A C A G T T A C C

T A - A G G T - C A

1. 최적배열의 오른쪽 맨 아래 구성에서 시작하여 표시해둔 경로를 따라간다.
2. 배열의 [*i*][*j*]칸에 도달하기 위해서 대각선으로 이동할 때 마다, *i*째 행에 해당하는 문자를 *x*서열에 넣고, *j*째 열에 해당하는 문자를 *y*서열에 넣는다.
3. 배열의 [*i*][*j*]칸에 도달하기 위해서 위로 이동할 때 마다, *i*째 행에 해당하는 문자를 *x*서열에 넣고, 틈 문자를 *y*서열에 넣는다.
4. 배열의 [*i*][*j*]칸에 도달하기 위해서 왼쪽으로 이동할 때 마다, *j*째 열에 해당하는 문자를 *y*서열에 넣고, 틈 문자를 *x*서열에 넣는다.


3장 끝
수고하셨습니다.