

왜 효율적인 알고리즘이 필요한가?

- 10억 개의 숫자를 정렬하는데 PC에서 $O(n^2)$ 알고리즘은 300여년이 걸리는 반면에 $O(n\log n)$ 알고리즘은 5분 만에 정렬한다.

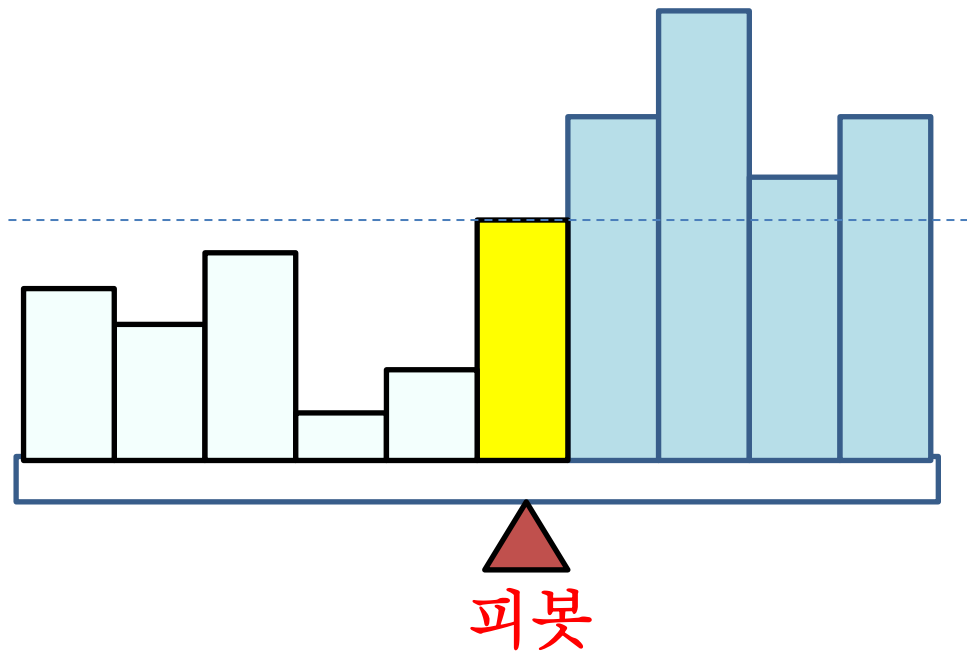
$O(n^2)$	1,000	1백만	10억
PC	< 1초	2시간	300년
슈퍼컴	< 1초	1초	1주일

$O(n\log n)$	1,000	1백만	10억
PC	< 1초	< 1초	5분
슈퍼컴	< 1초	< 1초	< 1초

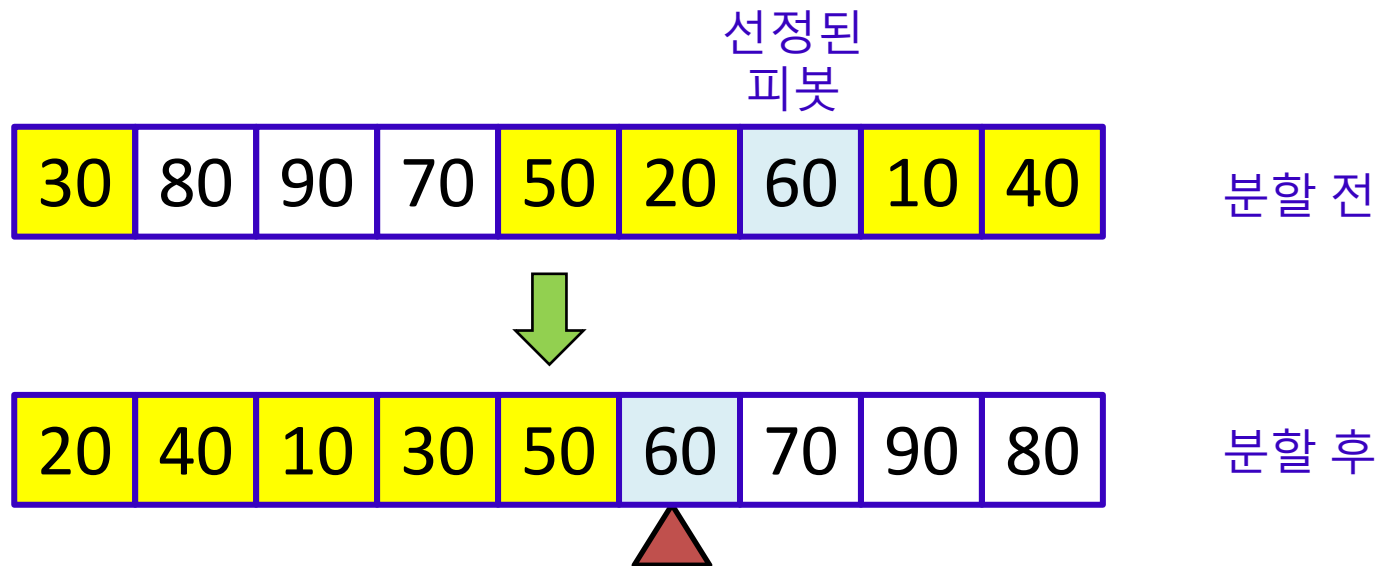
퀵 정렬

- 퀵 정렬 (Quick Sort)은 분할 정복 알고리즘으로 분류되나, 사실 알고리즘이 수행되는 과정을 살펴보면 **정복 후 분할**하는 알고리즘이다.
- 퀵 정렬 알고리즘은 문제를 2개의 부분문제로 분할하는데, 각 부분문제의 크기가 일정하지 않은 형태의 분할 정복 알고리즘

- 퀵 정렬은 **피벗 (pivot)**이라 일컫는 배열의 원소(숫자)를 기준으로 피벗보다 작은 숫자들은 왼편으로, 피벗보다 큰 숫자들은 오른편에 위치하도록 분할하고, 피벗을 그 사이에 놓는다.
- 퀵 정렬은 분할된 부분문제들에 대하여서도 위와 동일한 과정을 재귀적으로 수행하여 정렬



- 피벗은 분할된 왼편이나 오른편 부분에 포함되지 않는다. 피벗이 60이라면, 60은 [20 40 10 30 50]과 [70 90 80] 사이에 위치한다.



퀵 정렬 알고리즘

QuickSort(A, left, right)

입력: 배열 $A[\text{left}] \sim A[\text{right}]$

출력: 정렬된 배열 $A[\text{left}] \sim A[\text{right}]$

1. if (left < right) {
2. 피벗을 $A[\text{left}] \sim A[\text{right}]$ 중에서 선택하고, 피벗을 $A[\text{left}]$ 와 자리를 바꾼 후, 피벗과 배열의 각 원소를 비교하여 피벗보다 작은 숫자들은 $A[\text{left}] \sim A[p-1]$ 로 옮기고, 피벗보다 큰 숫자들은 $A[p+1] \sim A[\text{right}]$ 로 옮기며, 피벗은 $A[p]$ 에 놓는다.
3. **QuickSort**(A, left, p-1) // 피벗보다 작은 그룹
4. **QuickSort**(A, p+1, right) // 피벗보다 큰 그룹
- }

- Line 1에서는 배열 A 의 가장 왼쪽 원소의 인덱스 ($left$)가 가장 오른쪽 원소의 인덱스($right$)보다 작으면, line 2~4에서 정렬을 수행한다. 만일 그렇지 않으면 1개의 원소를 정렬하는 경우. 1개의 원소는 그 자체가 이미 정렬되어 있으므로, line 2~4의 정렬 과정을 수행할 필요 없이 그대로 호출을 마친다.
- Line 2에서는 $A[left] \sim A[right]$ 에서 피벗을 선택하고, 배열 $A[left+1] \sim A[right]$ 의 원소들을 피벗과 비교하여, 피벗보다 작은 그룹인 $A[left] \sim A[p-1]$ 과 피벗보다 큰 그룹인 $A[p+1] \sim A[right]$ 로 분할하고 $A[p]$ 에 피벗을 위치시킨다. 즉, p 는 피벗이 위치한 배열 A 의 인덱스이다.

- Line 3에서는 피벗보다 작은 그룹인 $A[\text{left}] \sim A[p-1]$ 을 재귀적으로 호출한다.
- Line 4에서는 피벗보다 큰 숫자들은 $A[p+1] \sim A[\text{right}]$ 를 재귀적으로 호출한다.

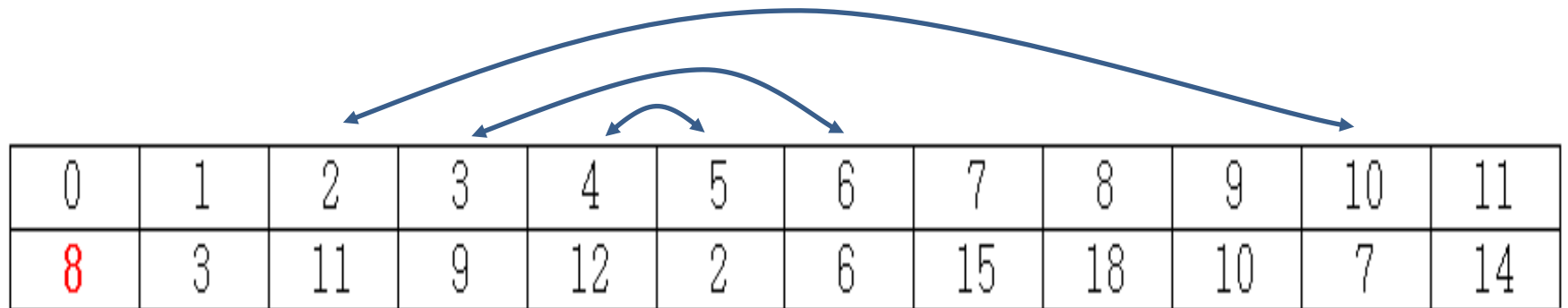
QuickSort(A,0,11) 호출

0	1	2	3	4	5	6	7	8	9	10	11
6	3	11	9	12	2	8	15	18	10	7	14

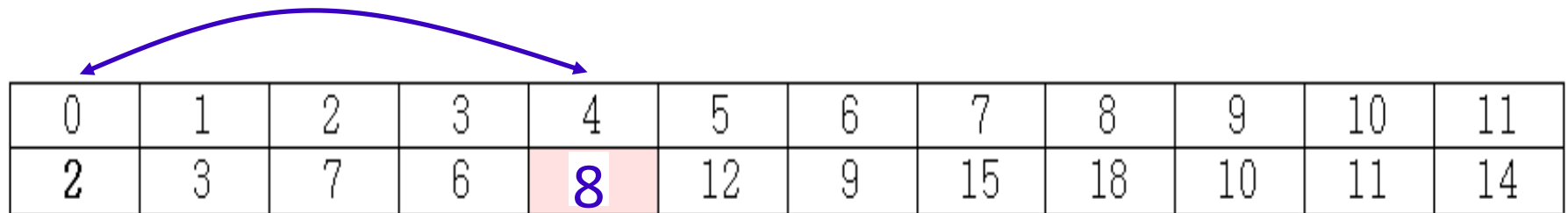
피벗 $A[6]=8$ 이라면, line 2에서 아래와 같이 차례로 원소들의 자리를 바꾼다. 먼저 피벗을 가장 왼쪽으로 이동시킨다.

0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14

- 그 다음엔 피벗보다 큰 수와 피벗보다 작은 수를 다음과 같이 각각 교환



0	1	2	3	4	5	6	7	8	9	10	11
8	3	11	9	12	2	6	15	18	10	7	14



0	1	2	3	4	5	6	7	8	9	10	11
2	3	7	6	8	12	9	15	18	10	11	14

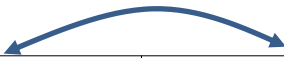
- line 3에서 $\text{QuickSort}(A, 0, 4-1) = \text{QuickSort}(A, 0, 3)$ 이 호출되고, 그 다음 line 4에서 $\text{QuickSort}(A, 4+1, 11) = \text{QuickSort}(A, 5, 11)$ 이 호출

QuickSort(A,0,3) 호출


0	1	2	3
2	3	7	6

- 피벗 $A[3]=6$ 이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.

0	1	2	3
6	3	7	2



0	1	2	3
2	3	6	7



- line 3에서 $\text{QuickSort}(A, 0, 2-1) = \text{QuickSort}(A, 0, 1)$ 이 호출되고, 그 다음 line 4에서 $\text{QuickSort}(A, 2+1, 3) = \text{QuickSort}(A, 3, 3)$ 이 호출


QuickSort(A, 0, 1) 호출

0	1
2	3

- 피벗 $A[1]=3$ 이라면, line 2에서 아래와 같이 원소들의 자리를 바꾼다.

0	1
3	2

0	1
2	3



- line 3에서 QuickSort(A,0,1-1) = QuickSort(A,0,0)이 호출되고, line 4에서 QuickSort(A,1+1,1) = QuickSort(A,2,1)이 호출
- QuickSort(A,0,0) 호출: Line 1의 if-조건이 '거짓'이 되어서 알고리즘을 더 이상 수행하지 않는다.
- QuickSort(A,2,1) 호출: Line 1의 if-조건이 '거짓'이므로 알고리즘을 수행하지 않는다.
- 위의 과정과 유사하게 A[2]~A[3]도 정렬되며, QuickSort(A,0,3)이 아래와 같이 완성된다.

0	1	2	3
2	3	7	6

시간복잡도

- 퀵 정렬의 성능은 피벗 선택이 좌우한다. 피벗으로 가장 작은 숫자 또는 가장 큰 숫자가 선택되면, 한 부분으로 치우치는 분할을 야기한다.

피벗

1	17	42	9	18	23	31	11	26
---	----	----	---	----	----	----	----	----



1	9	42	17	18	23	31	11	26
---	---	----	----	----	----	----	----	----



...

1	9	11	17	18	23	26	31	42
---	---	----	----	----	----	----	----	----



- 피벗=1일 때: 8회 - [17 42 9 18 23 31 11 26]과 각각 1회씩 비교
- 피벗=9일 때: 7회 - [42 17 18 23 31 11 26]과 각각 1회씩 비교
- 피벗=11일 때: 6회 - [17 18 23 31 42 26]과 각각 1회씩 비교
- ...
- 피벗=31일 때: 1회 - [42]와 1회 비교
- 총 비교 횟수는 $8+7+6+\dots+1 = 36$ 이다.
- 입력의 크기가 n 이라면, 퀵 정렬의 최악 경우 시간 복잡도 = $(n-1)+(n-2)+(n-3)+\dots+2+1 = n(n-1)/2 = O(n^2)$

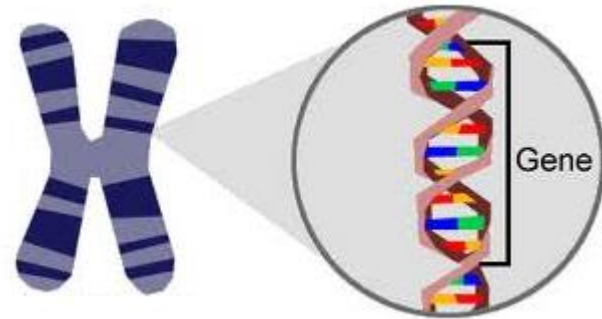
피벗 선정 방법

- 랜덤하게 선정하는 방법
- 3 숫자의 중앙값으로 선정하는 방법: 가장 왼쪽 숫자, 중간 숫자, 가장 오른쪽 숫자 중에서 중앙값으로 피벗을 정한다. 아래의 예제를 보면, 31, 1, 26 중에서 중앙값인 26을 피벗으로 사용한다.

31	17	42	9	1	23	18	11	26
----	----	----	---	---	----	----	----	----

응 용

- 켤 정렬은 커다란 크기의 입력에 대해서 가장 좋은 성능을 보이는 정렬 알고리즘이다.
- 켤 정렬은 실질적으로 어느 정렬 알고리즘보다 좋은 성능을 보인다.
- 생물 정보 공학(Bioinformatics)에서 특정 유전자를 효율적으로 찾는데 접미 배열(suffix array)과 함께 켤 정렬이 활용된다.

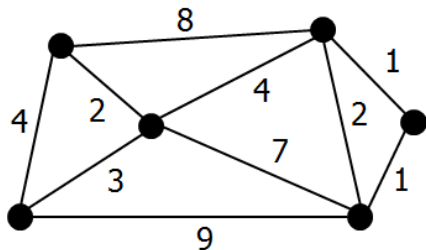


그리디 (Greedy) 알고리즘

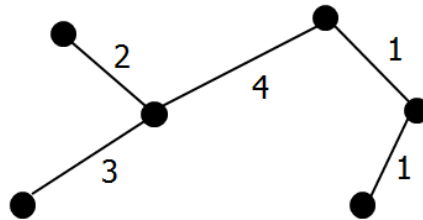
- 그리디 알고리즘은 최적화 문제를 해결한다.
- 최적화 (optimization) 문제: 가능한 해들 중에서 가장 좋은 (최대 또는 최소) 해를 찾는 문제
- 욕심쟁이 방법, 탐욕적 방법, 탐욕 알고리즘 등으로 불리기도 한다.
- 그리디 알고리즘은 (입력) 데이터 간의 관계를 고려하지 않고 수행 과정에서 '**욕심내어**' 최소값 또는 최대값을 가진 데이터를 선택한다.
- 이러한 선택을 '**근시안적**'인 선택이라고 말하기도 한다.

최소 신장 트리

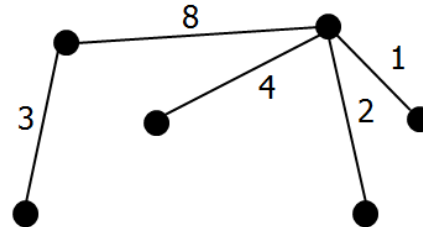
- 최소 신장 트리 (Minimum Spanning Tree): 주어진 가중치 그래프에서 사이클이 없이 모든 점들을 연결시킨 트리들 중 선분들의 가중치 합이 최소인 트리
- (a)주어진 가중치 그래프, (b) 최소 신장 트리 (c),(d)는 최소 신장 트리 아님
 - (c)는 가중치의 합이 (b)보다 크고, (d)는 트리가 주어진 그래프의 모든 노드를 포함하지 않고 있다.



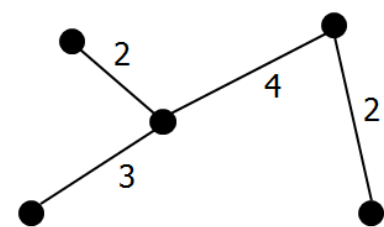
(a)



(b)

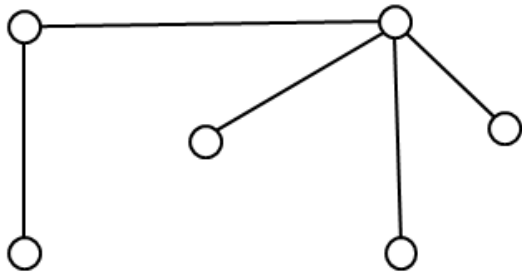


(c)

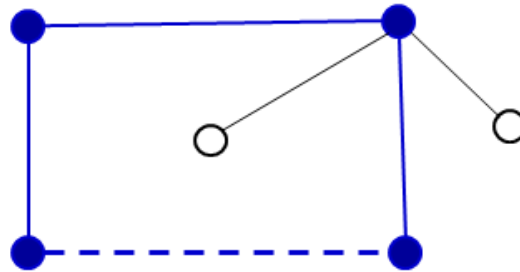


(d)

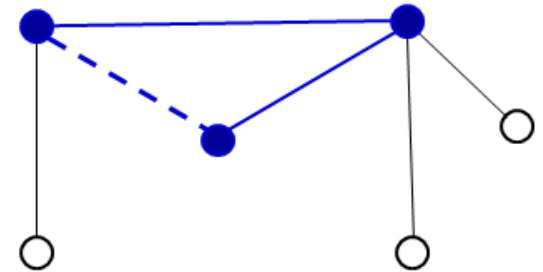
- 주어진 그래프의 신장 트리를 찾으려면 사이클이 없도록 모든 점을 연결시키면 된다. 그래프의 점의 수가 n 이면, 신장 트리에는 정확히 $(n-1)$ 개의 선분이 있다.
- 트리에 선분을 하나 추가시키면, 반드시 사이클이 만들어진다.



트리



점선으로 된 선분을 추가하여 만들어진 사이클



- 최소 신장 트리를 찾는 대표적인 그리디 알고리즘으로는 **크루스컬 (Kruskal)**과 **프림 (Prim)** 알고리즘이 있다. 알고리즘의 입력은 1개의 연결요소 (connected component)로 된 가중치 그래프이다.
- 크루스컬 알고리즘은 가중치가 가장 작은 선분이 사이클을 만들지 않을 때에만 '욕심내어' 그 선분을 추가시킨다. 다음은 크루스컬의 최소 신장 트리 알고리즘이다.

KruskalMST(G)

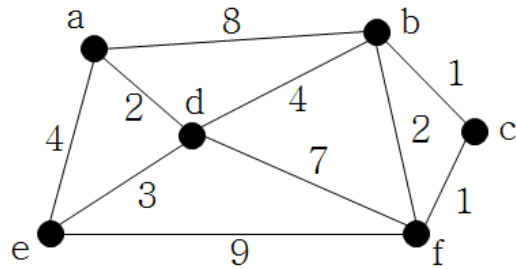
입력: 가중치 그래프 $G=(V,E)$, $|V|=n$, $|E|=m$

출력: 최소 신장 트리 T

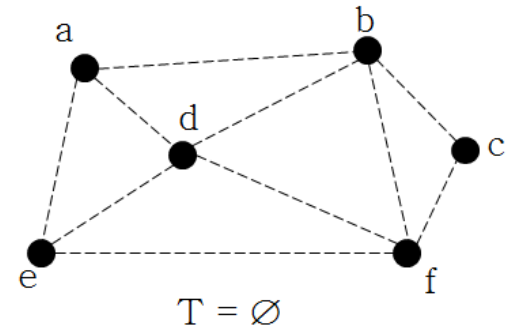
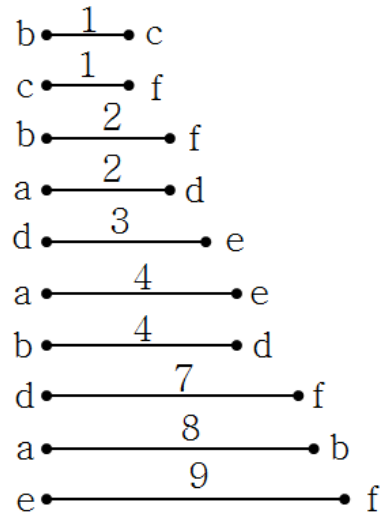
1. 가중치의 오름차순으로 선분들을 정렬한다. 정렬된 선분 리스트를 L 이라고 하자.
2. $T=\emptyset$ *// 트리 T 를 초기화시킨다.*
3. while (T 의 선분 수 $< n-1$) {
4. L 에서 가장 작은 가중치를 가진 선분 e 를 가져오고, e 를 L 에서 제거한다.
5. if (선분 e 가 T 에 추가되어 사이클을 만들지 않으면)
6. e 를 T 에 추가시킨다.
7. else *// e 가 T 에 추가되어 사이클이 만들어지는 경우*
8. e 를 버린다.
- }
9. return 트리 T *// T 는 최소 신장 트리이다.*

- Line 1: 모든 선분들을 가중치의 오름차순으로 정렬한다. 정렬된 선분들의 리스트를 L 이라고 하자.
- Line 2: T 를 초기화시킨다. 즉, T 에는 아무 선분도 없는 상태에서 시작된다.
- Line 3~8의 while-루프는 T 의 선분 수가 $(n-1)$ 이 될 때까지 수행되는데 1번 수행될 때마다 L 에서 가중치가 가장 작은 선분 e 를 가져온다. 단, 가져온 선분 e 는 L 에서 삭제되어 다시는 고려되지 않는다.
- Line 5~8: 가져온 선분 e 를 T 에 추가되어 사이클을 만들지 않으면 e 를 T 에 추가시키고, 사이클을 만들면 선분 e 를 버린다. 왜냐하면 모든 노드들이 연결되어 있으면서 사이클이 없는 그래프가 신장 트리이기 때문이다.

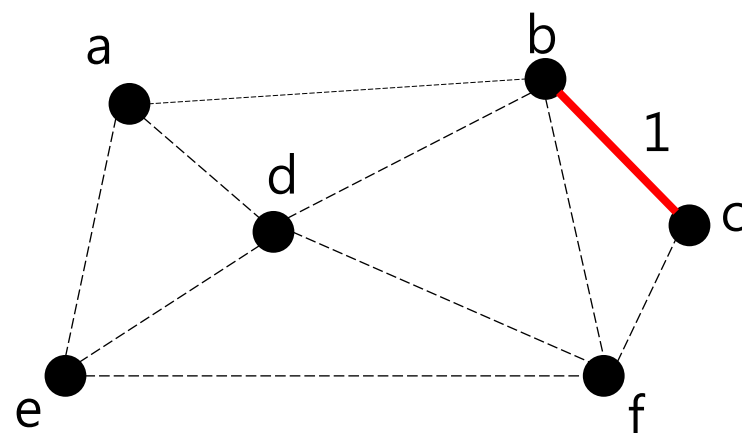
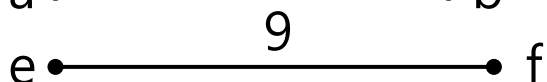
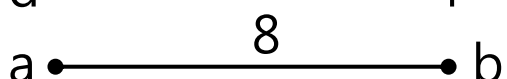
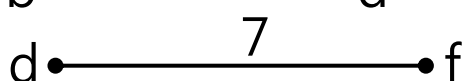
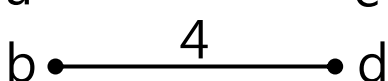
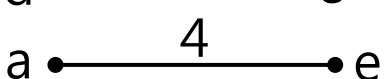
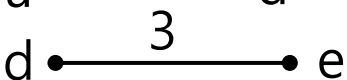
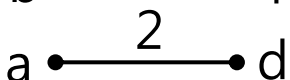
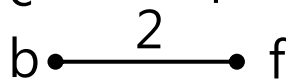
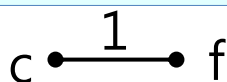
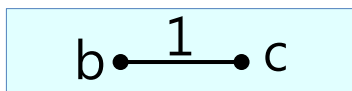
- 다음의 그래프에서 KruskalMST 알고리즘이 최소 신장 트리를 찾는 과정을 살펴보자.



정렬된
리스트
L

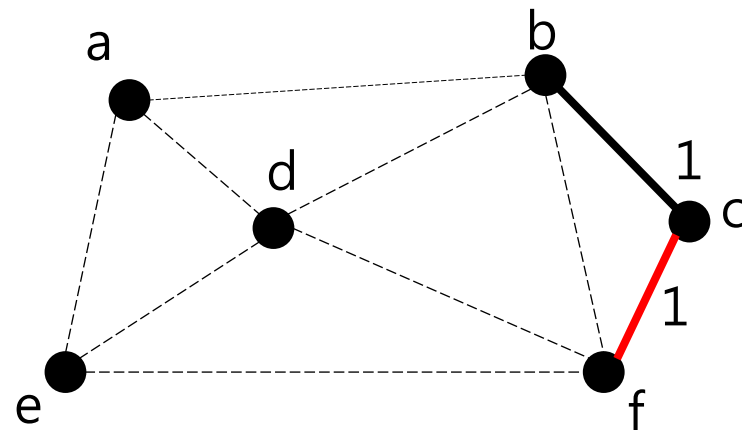
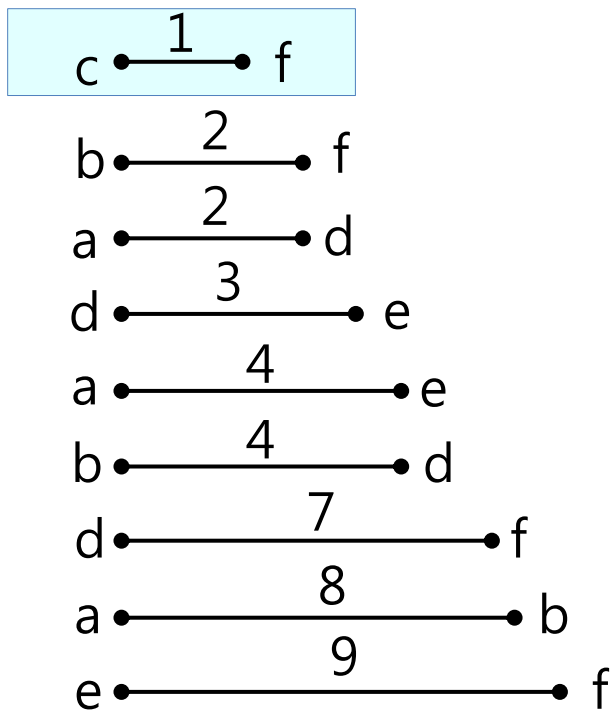


리스트
L



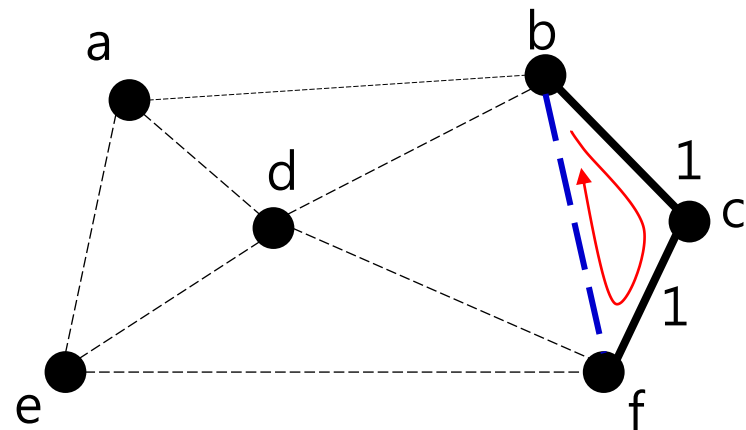
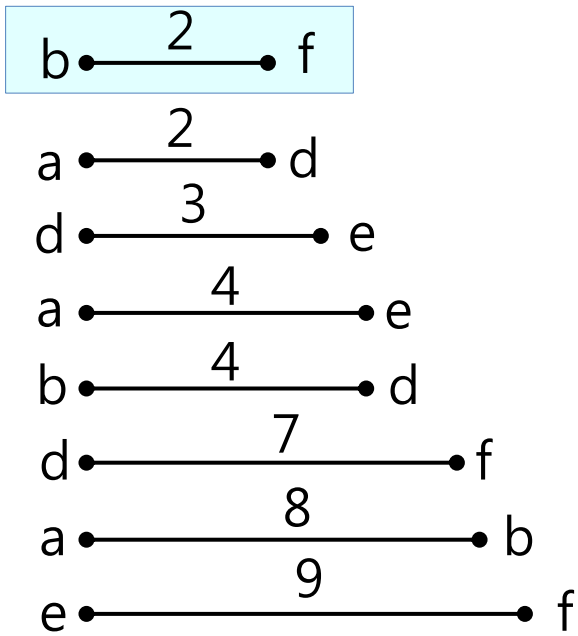
선분 (b,c) 추가

리스트
L

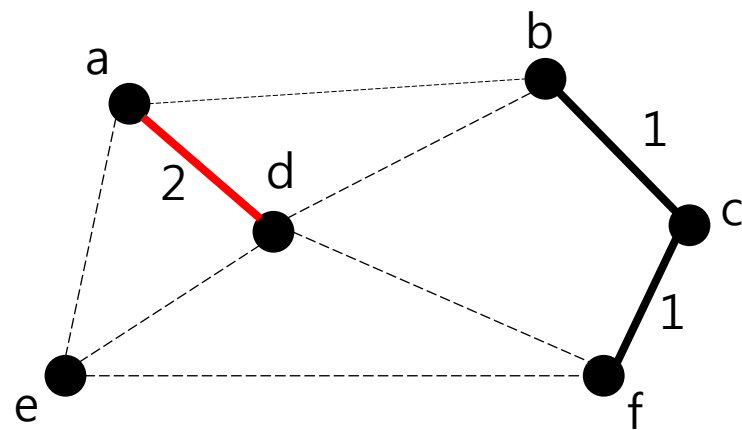
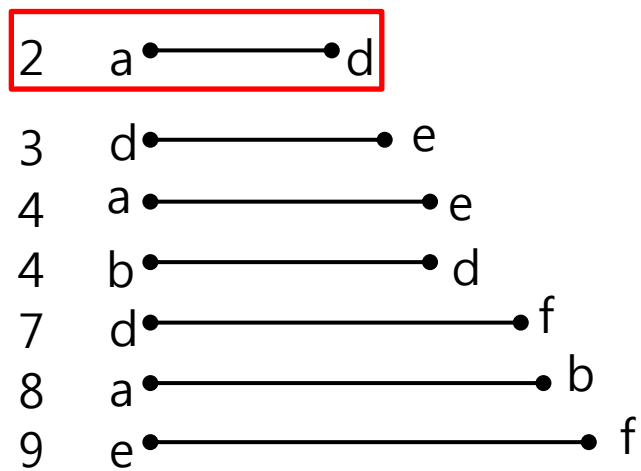


선분 (c,f) 추가

리스트
L



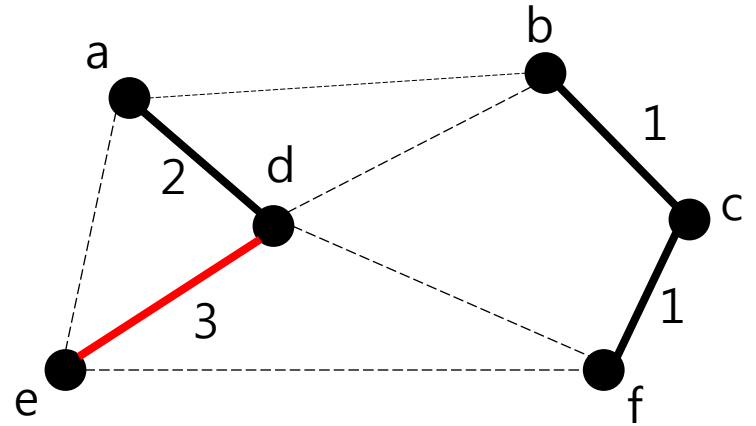
정렬된
리스트
L



선분 (a,d) 추가

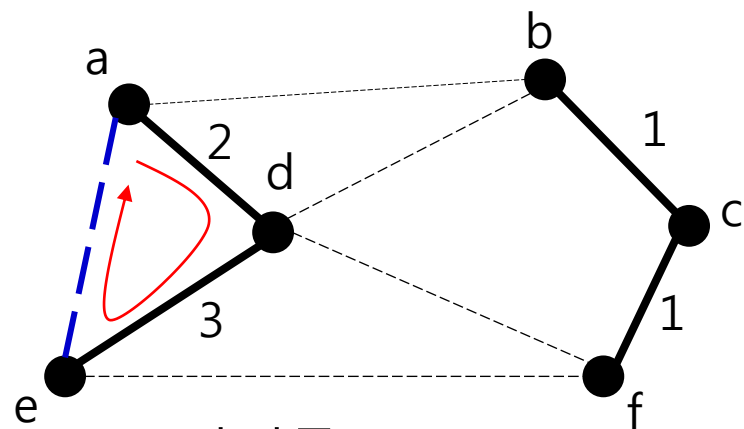
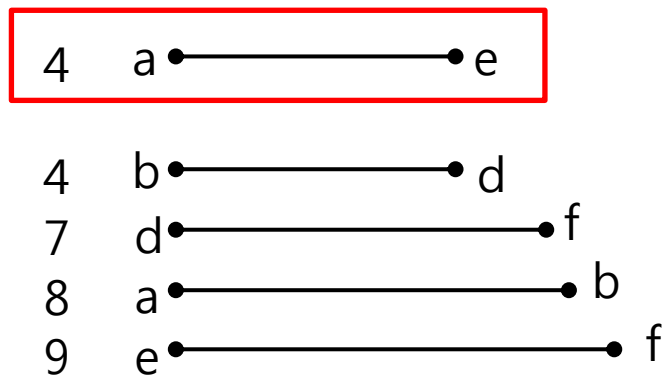
정렬된
리스트
L

3	d	—	e
4	a	—	e
4	b	—	d
7	d	—	f
8	a	—	b
9	e	—	f



선분 (d,e) 추가

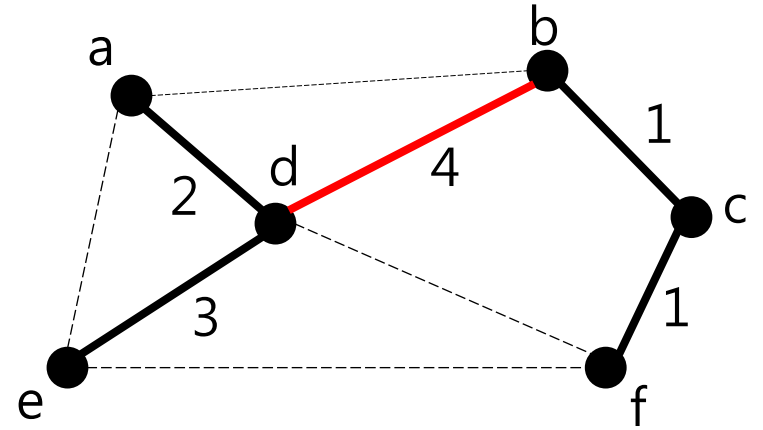
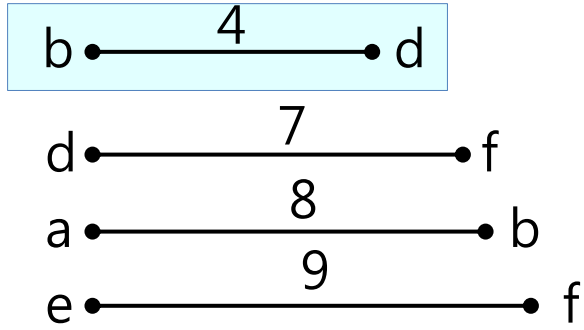
정렬된
리스트
L



사이클 a-d-e-a

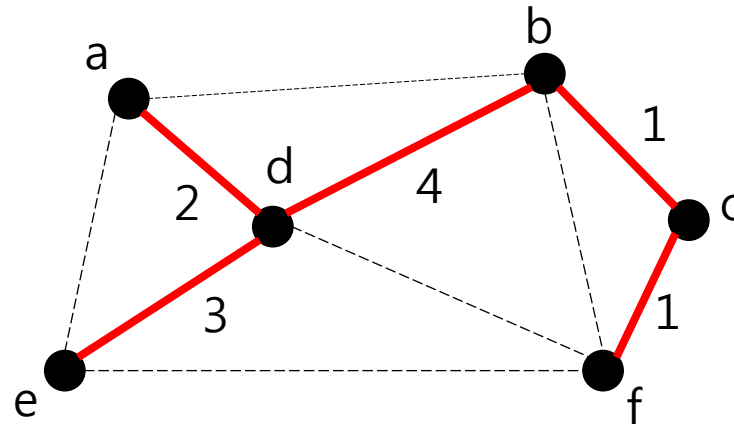
선분 (a,e) 버림

리스트
L



선분 (b,d) 추가

최종해



프림 (Prim)의 최소 신장 트리 알고리즘

- 주어진 가중치 그래프에서 임의의 점 하나를 선택한 후, $(n-1)$ 개의 선분을 하나씩 추가시켜 트리를 만든다.
- 추가되는 선분은 현재까지 만들어진 트리에 연결시킬 때 '욕심을 내어서' 항상 최소의 가중치로 연결되는 선분이다.

PrimMST(G)

입력: 가중치 그래프 $G=(V,E)$, $|V|=n$, $|E|=m$

출력: 최소 신장 트리 T

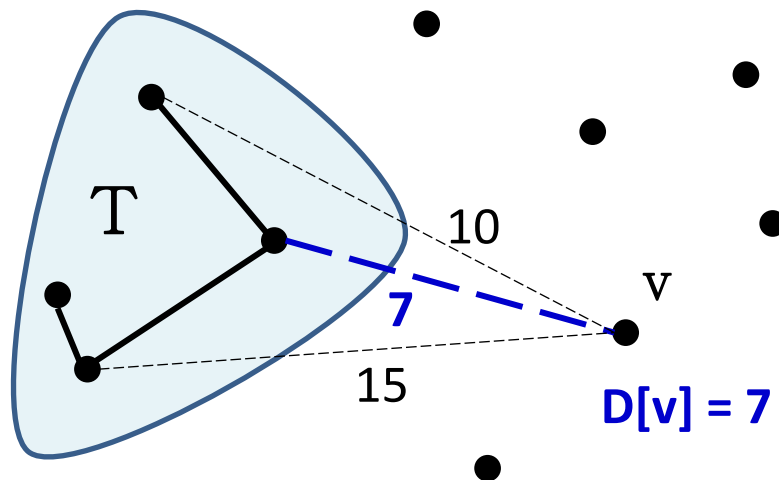
1. 그래프 G 에서 임의의 점 p 를 시작점으로 선택하고,
 $D[p]=0$ 으로 놓는다.

// 배열 $D[v]$ 는 T 에 있는 점 u 와 v 를 연결하는 선분의 최소
가중치를 저장하기 위한 원소이다.

2. for (점 p 가 아닌 각 점 v 에 대하여) { // 배열 D 의 초기화
3. if (선분 (p,v) 가 그래프에 있으면)
4. $D[v] =$ 선분 (p,v) 의 가중치
5. else
6. $D[v]=\infty$
- }

7. $T = \{p\}$ // 초기에 트리 T 는 점 p 만을 가진다.
8. while (T 에 있는 점의 수 $< n$) {
9. T 에 속하지 않은 각 점 v 에 대하여, $D[v]$ 가 최소인 점 v_{\min} 과 연결된 선분 (u, v_{\min}) 을 T 에 추가한다. 단, u 는 T 에 속한 점이고, 점 v_{\min} 도 T 에 추가된다.
10. for (T 에 속하지 않은 각 점 w 에 대해서) {
11. if (선분 (v_{\min}, w) 의 가중치 $< D[w]$)
12. $D[w] =$ 선분 (v_{\min}, w) 의 가중치 // $D[w]$ 를 갱신
- }
- }
13. return T // T 는 최소 신장 트리이다.

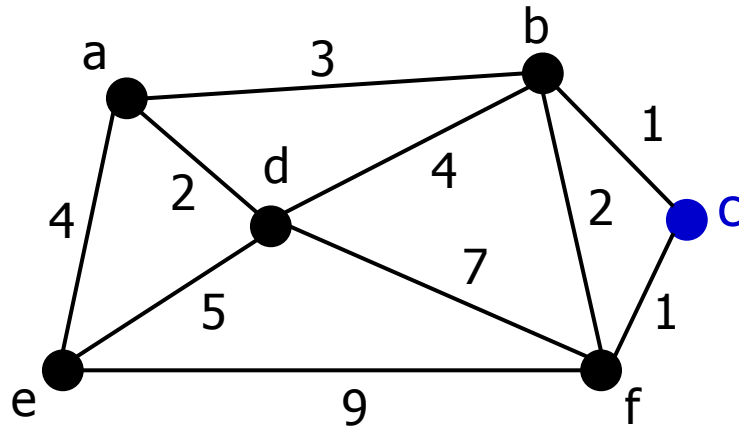
- Line 1: 임의로 점 p 를 선택하고, $D[p]=0$ 으로 놓는다.
여기서 배열 $D[v]$ 에는 점 v 와 T 에 속한 점들을
연결하는 선분들 중에서 최소 가중치를 가진
선분의 가중치를 저장한다.
- 다음그림에서 $D[v]$ 에는 10, 7, 15 중에서 최소
가중치인 **7**이 저장된다.



- Line 2~6: 시작점 p 와 선분으로 연결된 점 v 의 $D[v]$ 를 선분 (p,v) 의 가중치로 초기화시키고, 점 p 와 선분으로 연결되지 않은 점 v 에 대해서 $D[v]=\infty$ 로 놓는다.
- Line 7: $T = \{p\}$ 로 초기화시킨다.
- Line 8~12의 while-루프는 T 의 점의 수가 n 이 될 때까지 수행된다. T 에 속한 점의 수가 n 이 되면, T 는 신장 트리이다.
- Line 9: T 에 속하지 않은 각 점 v 에 대하여, $D[v]$ 가 최소인 점 v_{\min} 을 찾는다. 그리고 점 v_{\min} 과 연결된 선분 (u, v_{\min}) 을 T 에 추가한다. 단, u 는 T 에 속한 점이고, 선분 (u, v_{\min}) 이 T 에 추가된다는 것은 점 v_{\min} 도 T 에 추가되는 것이다.

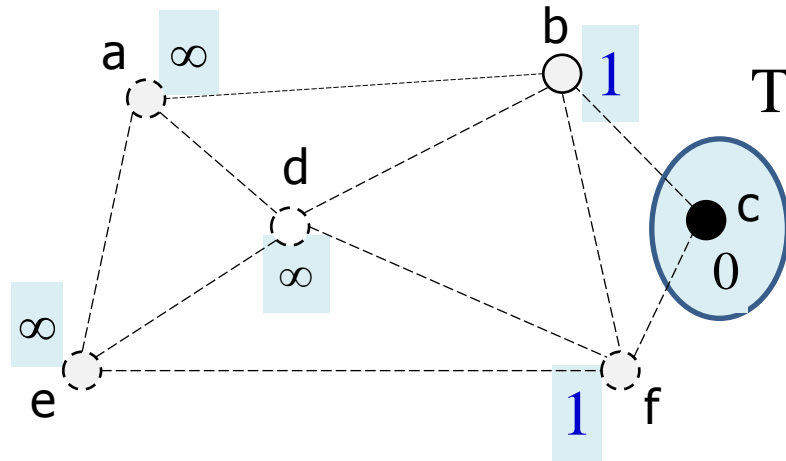
- Line 10~12의 for-루프에서는 line 9에서 새로이 추가된 점 v_{\min} 에 연결되어 있으면서 T 에 속하지 않은 각 점 w 의 $D[w]$ 를 선분 (v_{\min}, w) 의 가중치가 $D[w]$ 보다 작으면 (if-조건), $D[w]$ 를 선분 (v_{\min}, w) 의 가중치로 갱신한다.
- 마지막으로 line 13에서는 최소 신장 트리 T 를 리턴한다.

PrimMST 알고리즘 수행 과정

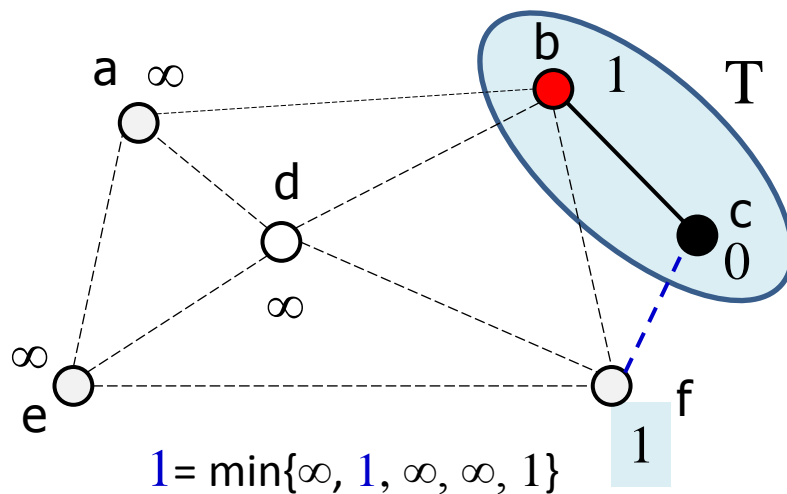


- Line 1: 임의의 시작점으로 **점 c**가 선택되었다고 가정하자. 그리고 $D[c]=0$ 으로 초기화시킨다.

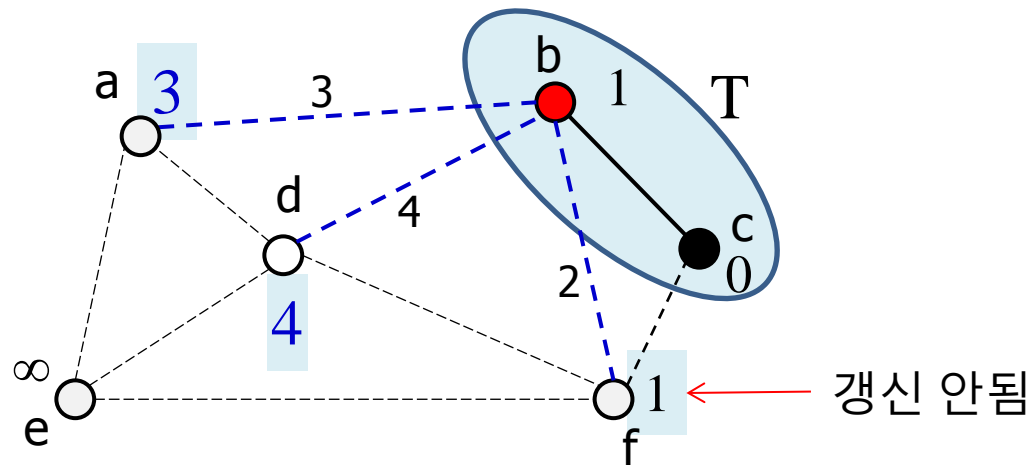
- Line 2~6: 시작점 c 와 선분으로 연결된 각 점 v 에 대해서, $D[v]$ 를 각 선분의 가중치로 초기화시키고, 나머지 각 점 w 에 대해서, $D[w]$ 는 ∞ 로 초기화시킨다.



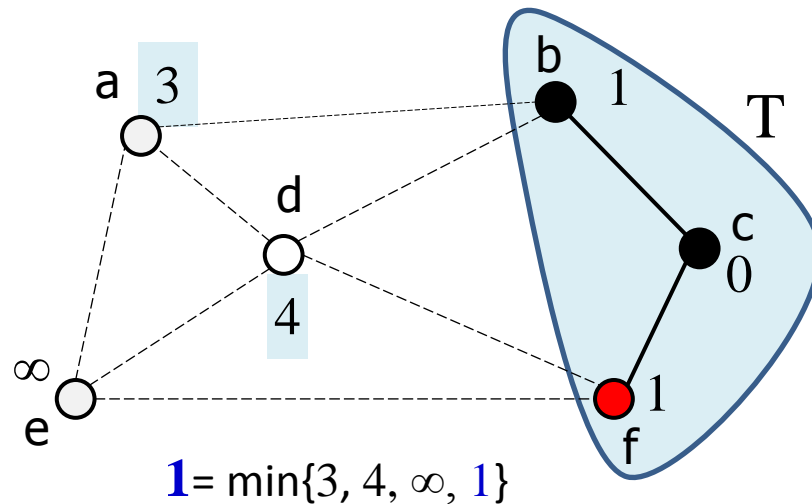
- Line 7: $T=\{c\}$ 로 초기화한다.
- Line 8의 while-루프의 조건이 '참'이다. 즉, 현재 T 에는 점 c 만이 있다. 따라서 line 9에서 T 에 속하지 않은 각 점 v 에 대하여, $D[v]$ 가 최소인 점 v_{\min} 을 선택한다. $D[b]=D[f]=1$ 로서 최소값이므로 점 b 나 점 f 중 택일. 여기서는 점 b 를 선택하자. 따라서 점 b 와 선분 (c,b) 가 T 에 추가된다.



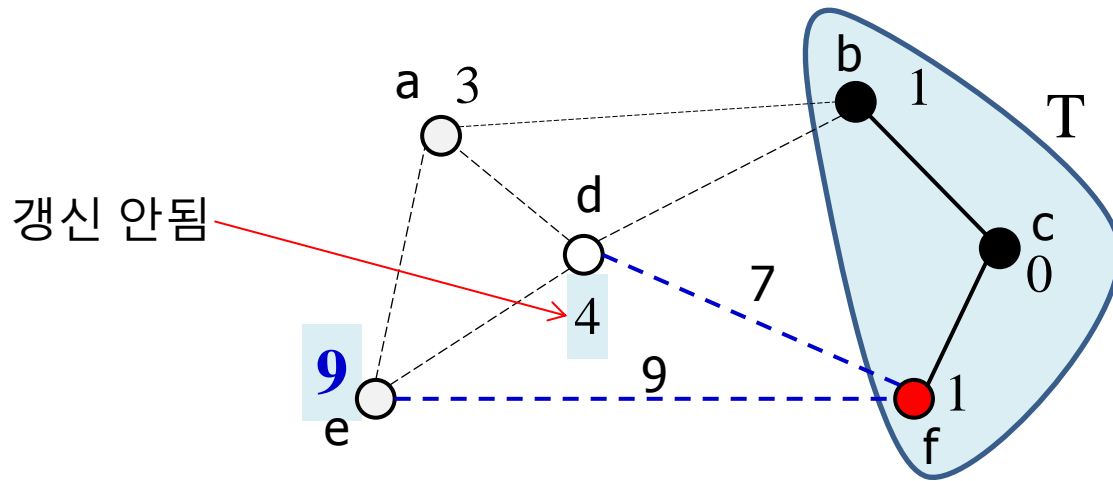
- Line 10~12: 점 b에 연결된 점 a와 d의 $D[a]$ 와 $D[b]$ 를 각각 3과 4로 갱신한다. 점 f는 점 b와 선분으로 연결은 되어있으나, 선분 (b,f)의 가중치인 2가 현재 $D[f]$ 보다 크므로 $D[f]$ 를 갱신안됨



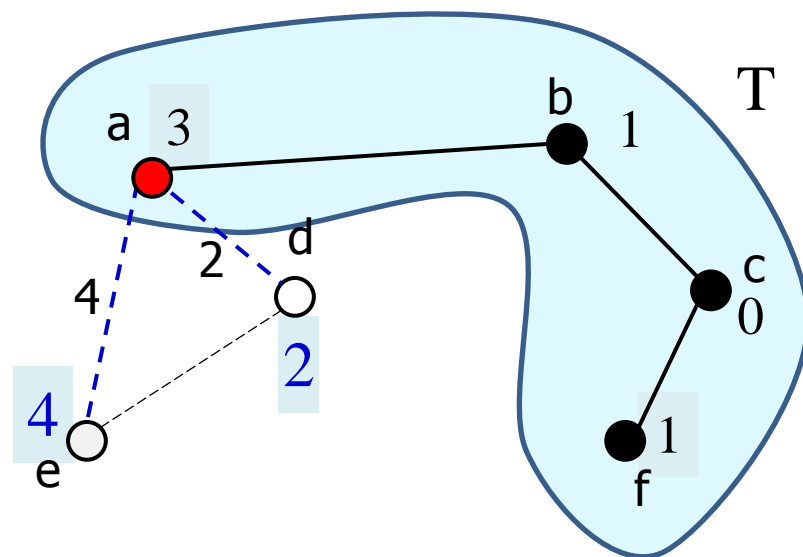
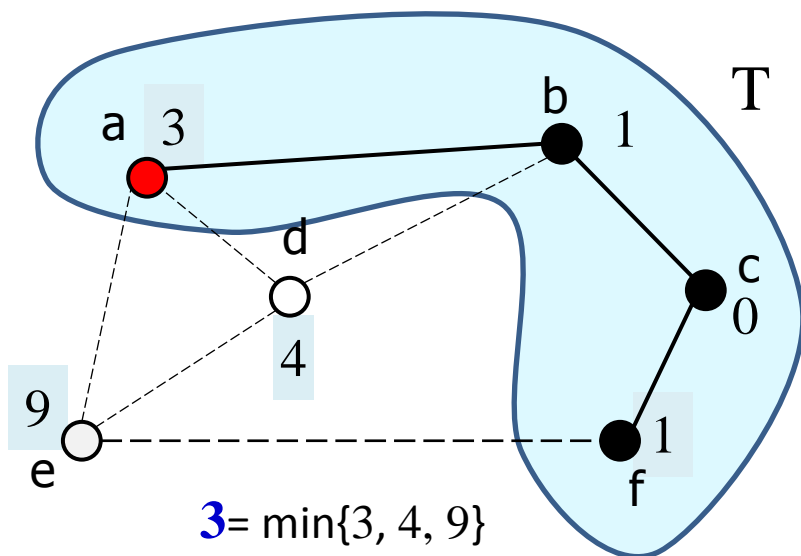
- Line 8의 while-루프의 조건이 '참'이므로, line 9에서 T에 속하지 않은 각 점 v에 대하여, v_{\min} 인 점 f를 찾고, 점 f와 선분 (c,f)를 T에 추가시킨다.

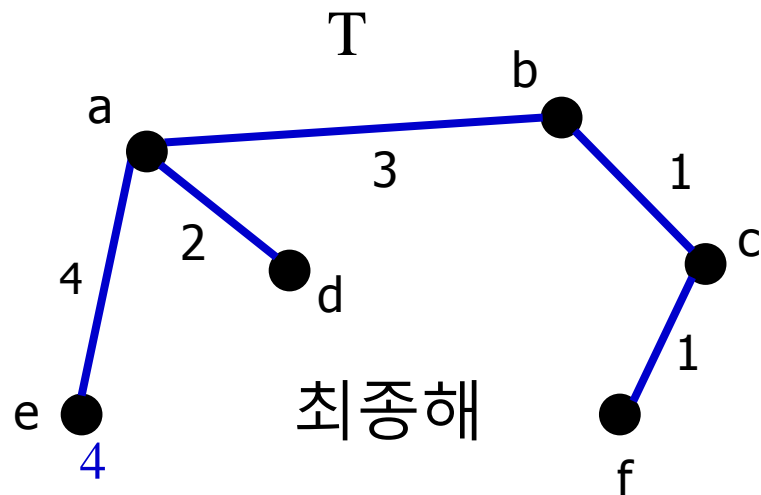
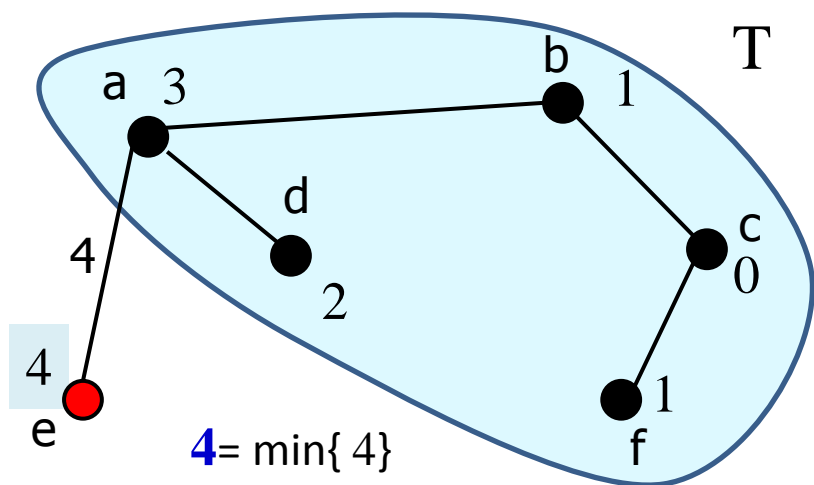
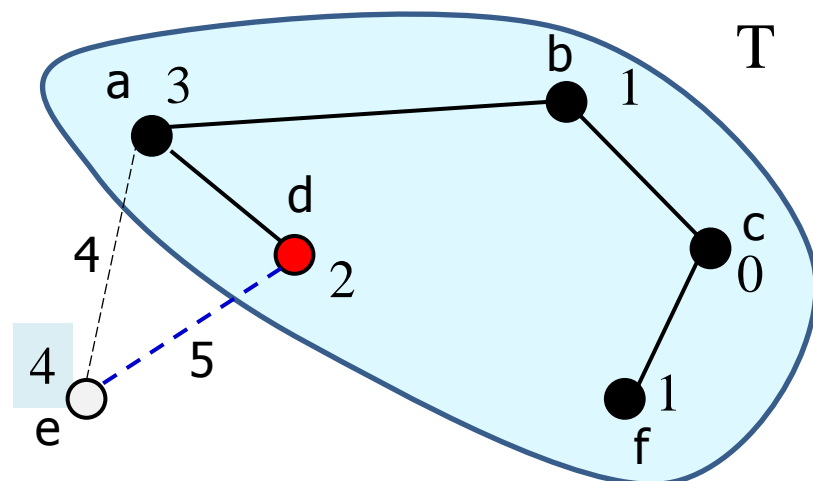
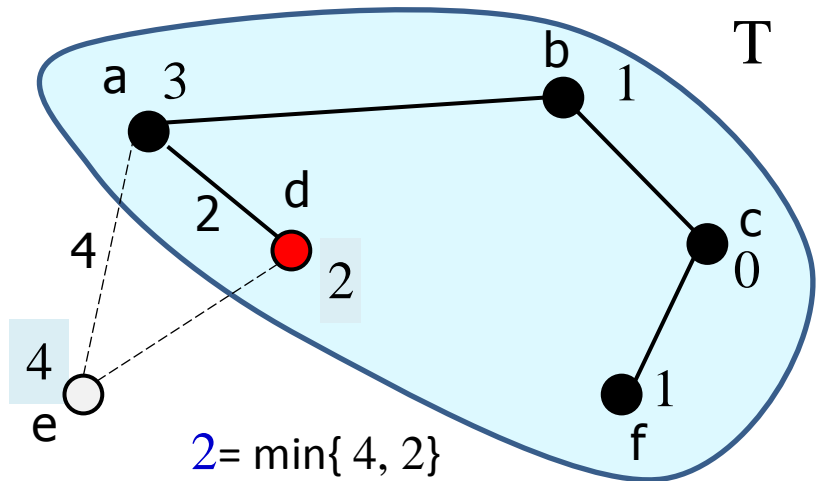


- Line 10~12: 점 f에 연결된 점 e의 $D[e]$ 를 9로 갱신한다. $D[d]$ 는 선분 (f,d) 의 가중치인 7보다 작기 때문에 갱신안됨

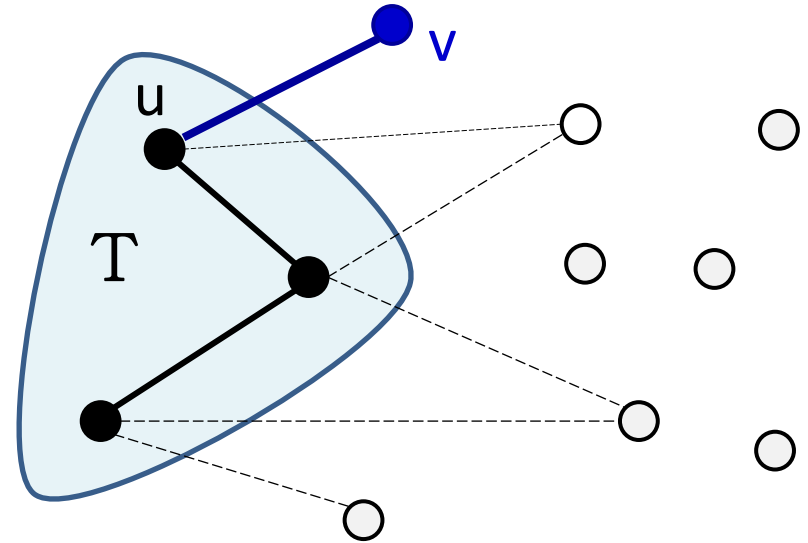
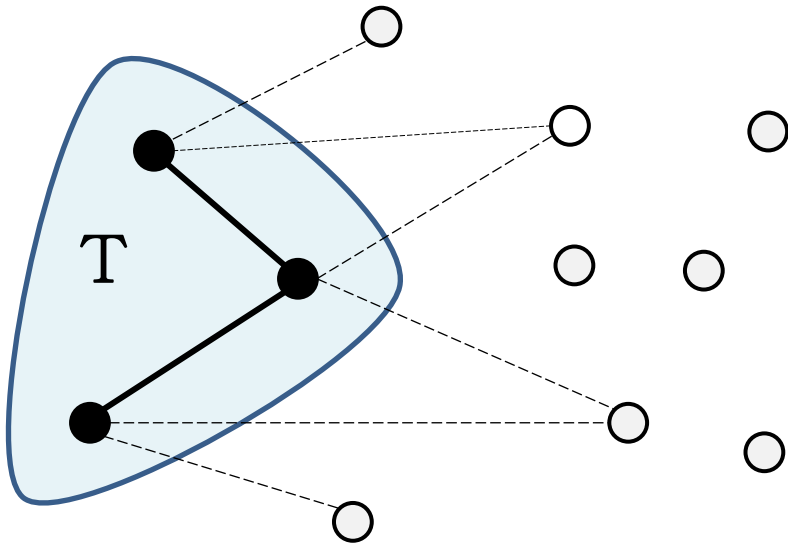


- 그 다음부터는 점 a와 선분 (b,a), 점 d와 선분 (a,d)가 차례로 T에 추가되고, 최종적으로 점 e와 선분 (a,e)가 추가되면서, 최소 신장 트리 T가 완성된다. Line 13에서는 T를 리턴하고, 알고리즘을 마친다. 다음의 그림들이 위의 과정을 차례로 보여준다.





- PrimMST 알고리즘이 최종적으로 리턴하는 T 에는 왜 사이클이 없을까?
- 프림 알고리즘은 T 밖에 있는 점을 항상 추가하므로 사이클이 안 만들어진다.



크러스컬과 프림 알고리즘의 수행 과정 비교

- 크러스컬 알고리즘에서는 선분이 1개씩 T에 추가 되는데, 이는 마치 n개의 점들이 각각의 트리인 상태에서 선분이 추가되면 2개의 트리가 1개의 트리로 합쳐지는 것과 같다. 크러스컬 알고리즘은 이를 반복하여 1개의 트리인 T를 만든다. 즉, n개의 트리들이 점차 합쳐져서 1개의 신장 트리가 만들어진 다.
- 프림 알고리즘에서는 T가 점 1개인 트리에서 시작 되어 선분을 1개씩 추가시킨다. 즉, 1개의 트리가 자라나서 신장 트리가 된다.

응 용

- 최소 비용으로 선로 또는 파이프 네트워크 (인터넷 광 케이블 선로, 케이블 TV선로, 전화선로, 송유관로, 가스관로, 배수로 등)를 설치하는데 활용
- 여행자 문제 (Traveling Salesman Problem)를 근사적으로 해결하는데 이용

그래프의 활용

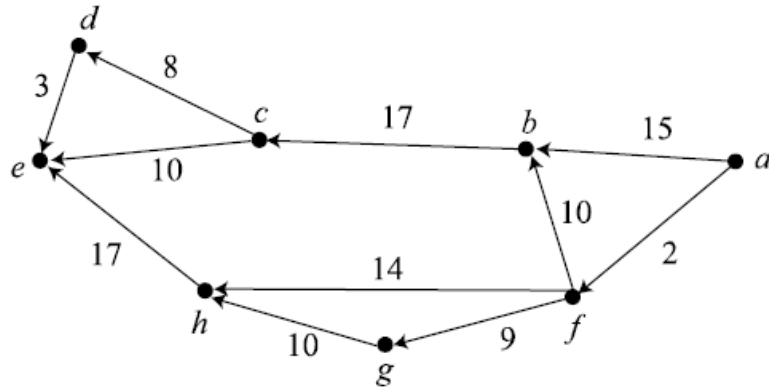
❖ 최단경로 문제

정의 9-29 최단경로 문제(Shortest Path Problem)

$|E| > 0$ 인 그래프 $G = (V, E)$ 에서 꼭짓점 $v_1, v_2 \in V$ 간의 가장 짧은 거리의 경로를 찾는 문제

- 출발점^{Source}: 경로의 시작점
- 도착점^{Destination}: 경로의 목적지

그래프의 활용



[그림 9-15] 최단경로 문제의 예

- 가중치 방향 그래프
- 가중치가 부여되지 않은 그래프 : 경로의 길이(경로에 포함되는 모서리의 수)로 최단경로
- 가중치가 부여된 경우 : 가중치를 계산하여 가중치에 의해 최단거리가 결정
- 가중치가 비용이라면 가중치의 합이 가장 작은 경로가 최단거리이고, 효과라고 하면 가중치의 합이 큰 경로가 최단거리가 됨
- [그림 9-15]의 그래프에 부여된 가중치가 꼭짓점에서 꼭짓점으로 이동하는 데 소비되는 비용이라고 했을 때, 꼭짓점 a에서 꼭짓점 e로 가는 경로와 거리를 구해 보기

그래프의 활용

① $a - b - c - d - e: 15 + 17 + 8 + 3 = 43$

② $a - b - c - e: 15 + 17 + 10 = 42$

③ $a - f - b - c - d - e: 2 + 10 + 17 + 8 + 3 = 40$

④ $a - f - b - c - e: 2 + 10 + 17 + 10 = 39$

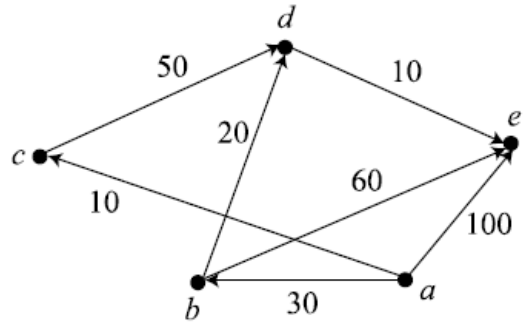
⑤ $a - f - h - e: 2 + 14 + 17 = 33$

⑥ $a - f - g - h - e: 2 + 9 + 10 + 17 = 38$

- 그래프의 최단경로는 ⑤가 됨
- 그래프 $G = \langle V, E \rangle$ 가 있을 때, $V = \{v_1, v_2, \dots, v_n\}$ 이고 시작점이 v_1 라고 가정했을 때 다익스트라 알고리즘에 사용되는 기호와 가정
 - $C[v_i, v_j]$: 꼭짓점 v_i 에서 v_j 로 가는 가중치를 의미한다.
 - $D[v_i]$: 시작점 v_1 에서부터 v_i 까지 이르는 최단경로의 가중치를 의미하는데, v_i 에서 v_j 로 가는 경로가 없으면 그 가중치는 ∞ 로 나타낸다.
 - 집합 S : 최단경로에 선택된 꼭짓점은 집합 S 의 원소로 포함시키고, 그래프 G 의 꼭짓점 집합 V 와 상등($S = V$)이 되면 알고리즘이 종료된다.

그래프의 활용

■ 가중치 그래프 예. 가중치는 비용



$$G = \langle V, E \rangle$$

$$V = \{a, b, c, d, e\}$$

$$E = \{ \langle a, b \rangle, \langle a, c \rangle, \langle a, e \rangle, \langle b, d \rangle, \langle c, d \rangle, \langle d, e \rangle \}$$

[그림 9-16] 다익스트라 알고리즘의 예

- 가장 초기 단계는 집합 $S = \emptyset$ 이고, 시작점이 a이므로 집합 $S = \{a\}$ 로 시작
- 무한대(∞)로 초기화 $D[b] = \infty, D[c] = \infty, D[d] = \infty, D[e] = \infty$
- (1) 최단경로의 거리를 알아보기 위해 시작점 a에서부터 그래프 G내의 다른 꼭짓점과의 가중치를 정리

$$D[b] = C[a, b] = 30$$

$$D[c] = C[a, c] = 10$$

$$D[d] = C[a, d] = \infty$$

$$D[e] = C[a, e] = 100$$

그래프의 활용

- (2) 앞의 단계에서 탐색된 꼭짓점은 $S=\{a,c\}$ 이고 시작점 a 와 추가된 꼭짓점 c 에서의 다른 꼭짓점까지의 가중치

$$D[b] = C[a, b] = 30$$

$$D[c] = C[a, c] = 10$$

$$D[d] = C[a, c] + C[c, d] = 60$$

$$D[e] = C[a, e] = 100$$

- (3) 집합 $S=\{a,c,b\}$ 에서의 다른 꼭짓점까지의 가중치

$$D[b] = C[a, b] = 30$$

$$D[c] = C[a, c] = 10$$

$$D[d] = C[a, b] + C[b, d] = 50$$

$$D[e] = C[a, b] + C[b, e] = 90$$

- (4) 집합 $S=\{a,c,b,d\}$ 에서부터 남은 꼭짓점 e 까지의 가중치

$$D[b] = C[a, b] = 30$$

$$D[c] = C[a, c] = 10$$

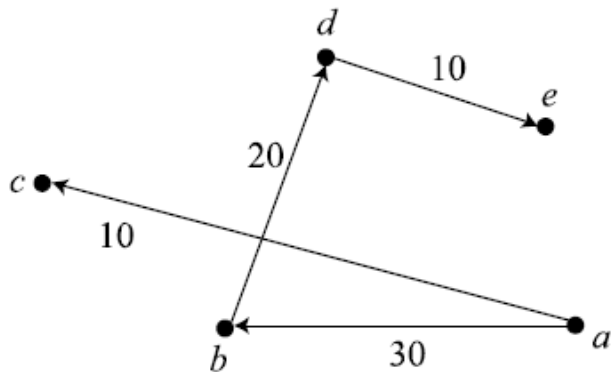
$$D[d] = C[a, b] + C[b, d] = 50$$

$$D[e] = C[a, b] + C[b, d] + C[d, e] = 60$$

그래프의 활용

[표 9-1] 다익스트라 알고리즘을 이용한 최단경로 탐색 과정

단계	탐색 전 최단경로 집합 (S)과 남은 꼭짓점 (V)	$D[b]$	$D[c]$	$D[d]$	$D[e]$	최단경로 꼭짓점	탐색 후 최단경로 집합 (S)과 남은 꼭짓점 (V)
(1)	$S = \{a\}$ $V = \{b, c, d, e\}$	30	10	∞	100	c	$S = \{a, c\}$ $V = \{b, d, e\}$
(2)	$S = \{a, c\}$ $V = \{b, d, e\}$	30	10	60	100	b	$S = \{a, c, b\}$ $V = \{d, e\}$
(3)	$S = \{a, c, b\}$ $V = \{d, e\}$	30	10	50	90	d	$S = \{a, c, b, d\}$ $V = \{e\}$
(4)	$S = \{a, c, b, d\}$ $V = \{e\}$	30	10	50	60	e	$S = \{a, c, b, d, e\}$ $V = \{ \}$

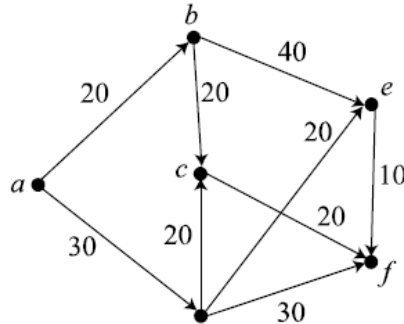


[그림 9-17] [그림 9-16]의 결과

그래프의 활용

예제 9-23

다음 그래프에서 시작점이 a 인 최단경로를 다익스트라 알고리즘을 이용해 구하라.



풀이

단계	탐색 전 최단경로 집합 (S)와 남은 꼭짓점 (V)	$D[b]$	$D[c]$	$D[d]$	$D[e]$	$D[f]$	최단경로 꼭짓점	탐색 후 최단경로 집합 (S)와 남은 꼭짓점 (V)
(1)	$S = \{a\}$ $V = \{b, c, d, e, f\}$	20	∞	30	∞	∞	b	$S = \{a, b\}$ $V = \{c, d, e, f\}$
(2)	$S = \{a, b\}$ $V = \{c, d, e, f\}$	20	40	30	60	∞	d	$S = \{a, b, d\}$ $V = \{c, e, f\}$
(3)	$S = \{a, b, d\}$ $V = \{c, e, f\}$	20	40	30	50	60	c	$S = \{a, b, d, c\}$ $V = \{e, f\}$
(4)	$S = \{a, b, d, c\}$ $V = \{e, f\}$	20	40	30	50	60	e	$S = \{a, b, d, c, e\}$ $V = \{f\}$
(5)	$S = \{a, c, b, d, e\}$ $V = \{f\}$	20	40	30	50	60	f	$S = \{a, b, d, c, e, f\}$ $V = \{ \}$

허프만 코드 알고리즘

HuffmanCoding

입력: 입력 파일의 n개의 문자에 대한 각각의 빈도수

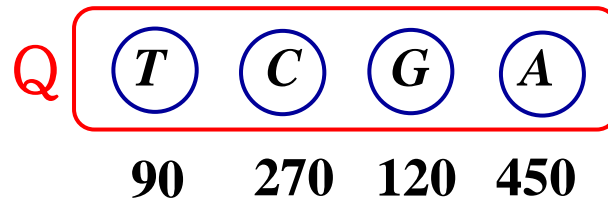
출력: 허프만 트리

1. 각 문자 당 노드를 만들고, 그 문자의 빈도수를 노드에 저장
2. n개의 노드의 빈도수에 대해 우선순위 큐 Q를 만든다.
3. while (Q에 있는 노드 수 ≥ 2) {
4. 빈도수가 가장 작은 2개의 노드 (A와 B)를 Q에서 제거
5. 새 노드 N을 만들고, A와 B를 N의 자식 노드로 만든다.
6. N의 빈도수 = A의 빈도수 + B의 빈도수
7. 노드 N을 Q에 삽입한다.
- }
8. return Q // 허프만 트리의 루트를 리턴하는 것이다.

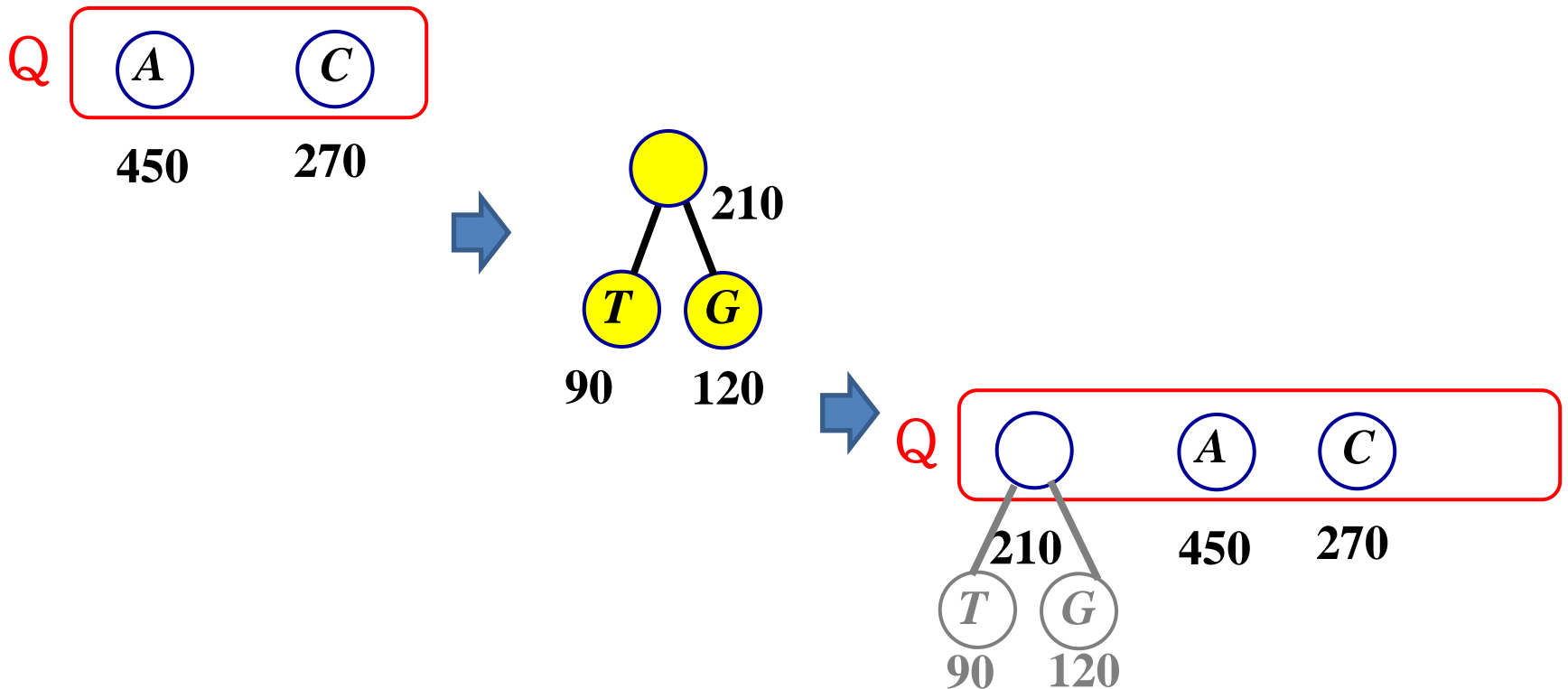
- HuffmanCoding 알고리즘의 수행 과정
- 입력 파일은 4개의 문자로 되어 있고, 각 문자의 빈도수는 다음과 같다.

A: 450 T: 90 G: 120 C: 270

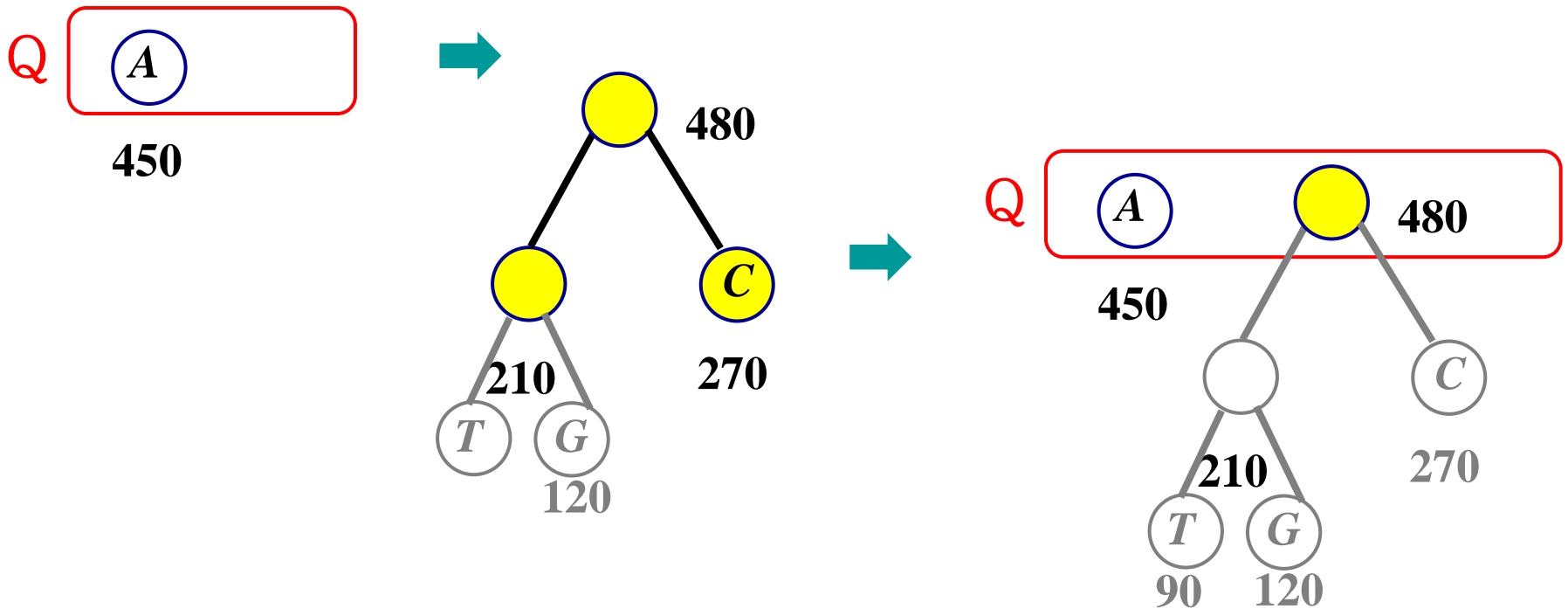
- Line 2를 수행한 후의 Q:



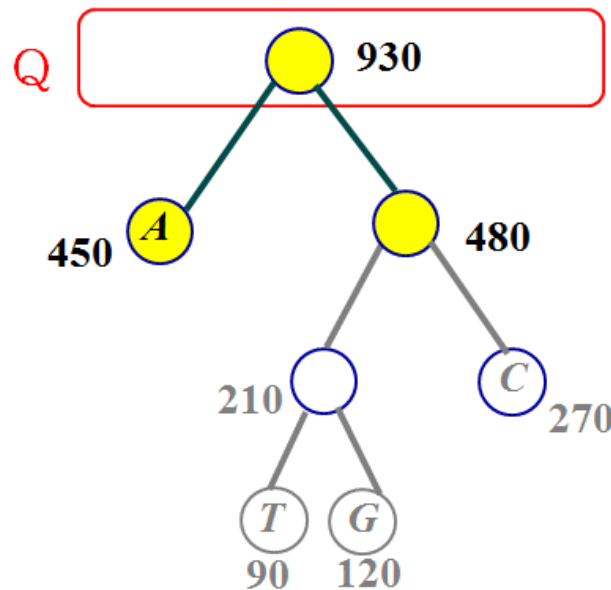
- Line 3의 while-루프 조건이 '참'이므로, line 4~7을 수행한다. 즉, Q에서 'T'와 'G'를 제거한 후, 새 부모 노드를 Q에 삽입한다.



- Line 3의 while-루프 조건이 '참'이므로, line 4~7을 수행한다. 즉, Q에서 'T'와 'G'의 부모 노드와 'C'를 제거한 후, 새 부모 노드를 Q에 삽입

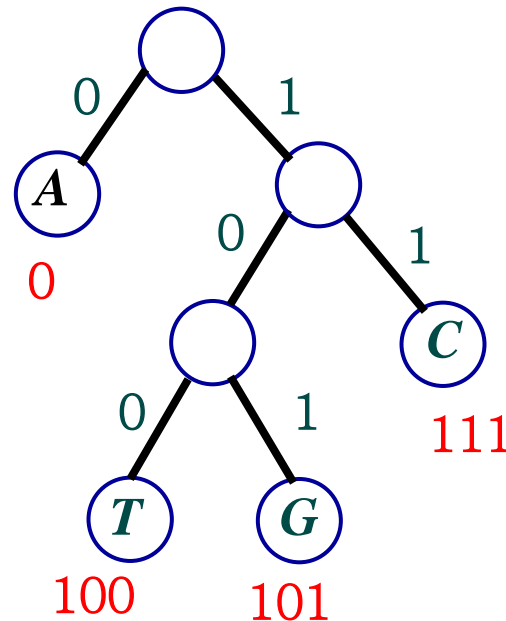


- Line 3의 while-루프 조건이 '참'이므로, line 4~7을 수행한다. 즉, Q에서 'C'의 부모 노드와 'A'를 제거한 후, 새 부모 노드 Q에 삽입한다.



- Line 3의 while-루프 조건이 '거짓'이므로, line 8에서 Q에 있는 노드를 리턴한다. 즉, 허프만 트리의 루트를 리턴한다.

- 리턴된 트리를 살펴보면 각 이파리 (단말) 노드에
만 문자가 있다. 따라서 루트로부터 왼쪽 자식 노
드로 내려가면 '0'을, 오른쪽 자식 노드로 내려가면
'1'을 부여하면서, 각 이파리에 도달할 때까지의 이
진수를 추출하여 문자의 이진 코드를 구한다.



- 위의 예제에서 'A'는 '0', 'T'는 '100', 'G'는 '101', 'C'는 '11'의 코드가 각각 할당된다.
- 할당된 코드들을 보면, 가장 빈도수가 높은 'A'가 가장 짧은 코드를 가지고, 따라서 루트의 자식이 되어 있고, 빈도수가 낮은 문자는 루트에서 멀리 떨어지게 되어 긴 코드를 가진다.
- 또한 이렇게 얻은 코드가 접두부 특성을 가지고 있음을 쉽게 확인할 수 있다.
- 위의 예제에서 압축된 파일의 크기의 bit 수는 $(450 \times 1) + (90 \times 3) + (120 \times 3) + (270 \times 2) = 1,620$ 이다.
- 반면에 아스키 코드로 된 파일 크기는 $(450 + 90 + 120 + 270) \times 8 = 7,440$ bit이다.
- 따라서 파일 압축률은 $(1,620 / 7,440) \times 100 = 21.8\%$ 이며, 원래의 약 1/5 크기로 압축되었다.

- 위의 예제에서 얻은 허프만 코드로 아래의 압축된 부분에 대해서 압축을 해제하여보면 다음과 같다.

10110010001110101010100

101 / 100 / 100 / 0 / 11 / 101 / 0 / 101 / 0 / 100

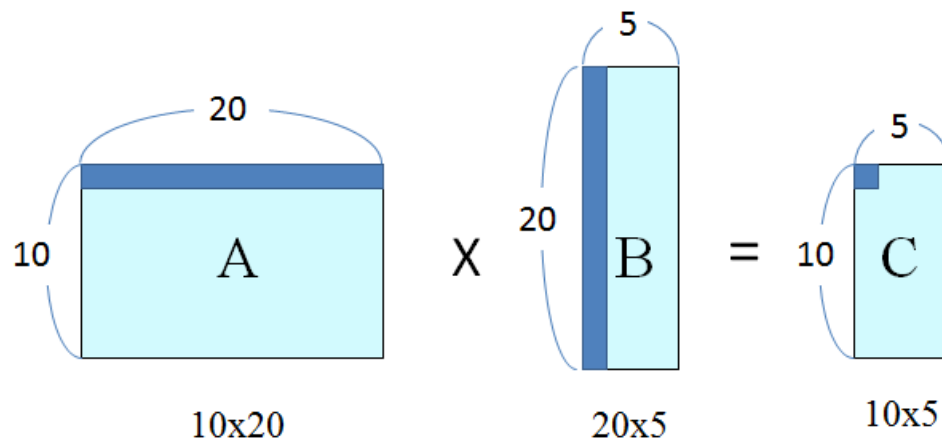
G T T A C G A G A T

응 용

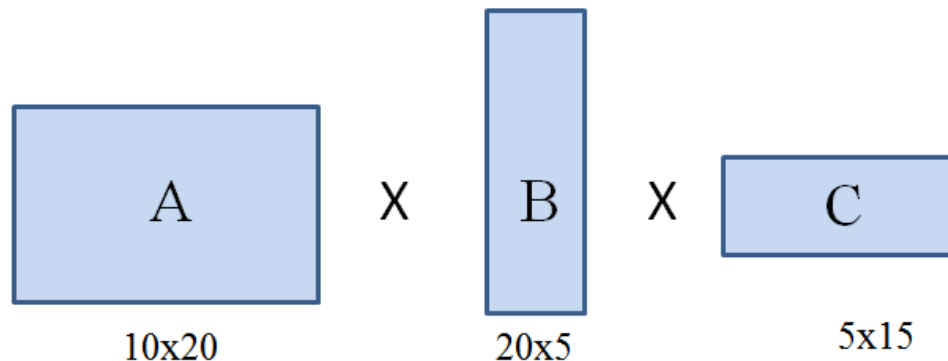
- 팩스(FAX), 대용량 데이터 저장, 멀티미디어 (Multimedia), MP3 압축 등에 활용된다.
- 또한 정보 이론 (Information Theory) 분야에서 엔트로피 (Entropy)를 계산하는데 활용되며, 이는 자료의 불특정성을 분석하고 예측하는데 이용된다.

연속 행렬 곱셈

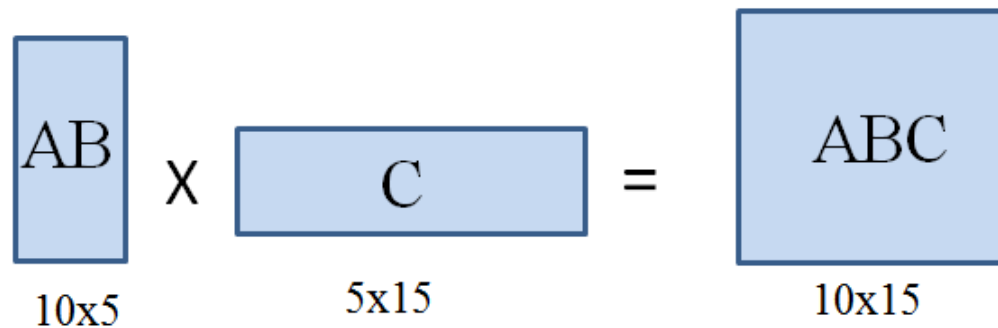
- 연속 행렬 곱셈 (Chained Matrix Multiplications) 문제는 연속된 행렬들의 곱셈에 필요한 원소간의 최소 곱셈 횟수를 찾는 문제이다.
- 10×20 행렬 A와 20×5 행렬 B를 곱하는데 원소간의 곱셈 횟수는 $10 \times 20 \times 5 = 1,000$ 이다. 그리고 두 행렬을 곱한 결과 행렬 C는 10×5 이다.



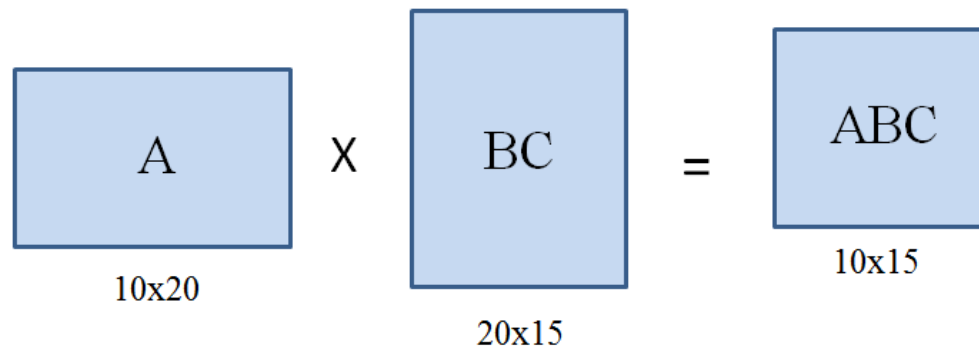
- 행렬 C의 1개의 원소를 위해서 행렬 A의 1개의 행에 있는 20개 원소와 행렬 B의 1개의 열에 있는 20개의 원소를 각각 곱한 값을 더해야 하므로 20회의 곱셈이 필요하다.
- 3개의 행렬을 곱해야 하는 경우
- 연속된 행렬의 곱셈에는 **결합 법칙이 허용**된다.
- 즉, $A \times B \times C = (A \times B) \times C = A \times (B \times C)$ 이다.
- 다음과 같이 행렬 A가 10x20, 행렬 B가 20x5, 행렬 C가 5x15라고 하자.



- 먼저 $A \times B$ 를 계산한 후에 그 결과 행렬과 행렬 C 를 곱하기 위한 원소간의 곱셈 횟수를 세어 보면, $A \times B$ 를 계산하는데 $10 \times 20 \times 5 = 1,000$ 번의 곱셈이 필요하고, 그 결과 행렬의 크기가 10×5 이므로, 이에 행렬 C 를 곱하는데 $10 \times 5 \times 15 = 750$ 번의 곱셈이 필요하다.
- 총 $1,000 + 750 = 1,750$ 회의 원소의 곱셈이 필요하다.



- 이번엔 BxC를 먼저 계산한 후에 행렬 A를 그 결과 행렬과 곱하면, BxC를 계산하는데 $20 \times 5 \times 15 = 1,500$ 번의 곱셈이 필요하고, 그 결과 20×15 행렬이 만들어지므로, 이를 행렬 A와 곱하는데 $10 \times 20 \times 15 = 3,000$ 번의 곱셈이 필요하다.
- 따라서 총 $1,500 + 3,000 = 4,500$ 회의 곱셈이 필요하다.



- 동일한 결과를 얻음에도 불구하고 원소간의 곱셈 횟수가 $4,500 - 1,700 = 2,800$ 이나 차이 난다.
- 따라서 연속된 행렬을 곱하는데 필요한 원소간의 곱셈 횟수를 최소화시키기 위해서는 적절한 행렬의 곱셈 순서를 찾아야 한다.
- 주어진 행렬의 순서를 지켜서 이웃하는 행렬끼리 반드시 곱해야 하기 때문
- 예를 들어, $A \times B \times C \times D \times E$ 를 계산하려는데, B를 건너뛰어서 $A \times C$ 를 수행한다든지, B와 C를 건너뛰어서 $A \times D$ 를 먼저 수행할 수 없다.
- 따라서 다음과 같은 부분문제들이 만들어진다.

부분 문제 크기						부분 문제 개수
1	A	B	C	D	E	5개
2	AxB	BxC	CxD	DxE		4개

- 맨 윗줄의 가장 작은 부분문제들은 입력으로 주어진 각각의 행렬 그 자체이고, 크기가 2인 부분문제는 2개의 이웃하는 행렬의 곱셈으로 이루어진 4개이다.
- 여기서 눈여겨보아야 할 것은 **부분 문제들이 겹쳐져 있다**는 것이다.
- 만일 AxB, CxD와 같이 부분문제가 서로 안 겹치면 AxBxCxD를 계산할 때 BxC에 대한 해가 없으므로 새로이 계산해야 한다.
- 이러한 경우를 대비하여 이웃하여 서로 겹치는 부분 문제들의 해도 미리 구하여 놓는다.

부분 문제
크기

부분 문제
개수

3

$A \times B \times C \quad B \times C \times D \quad C \times D \times E$

3개

4

$A \times B \times C \times D \quad B \times C \times D \times E$

2개

5

$A \times B \times C \times D \times E$

1개

- 그 다음은 크기가 3인 부분문제가 3개이고, 이들 역시 서로 이웃하는 부분문제들끼리 겹치어 있음을 확인할 수 있다.
- 다음 줄에는 크기가 4인 부분문제가 2개이고, 마지막에는 1개의 문제로서 입력으로 주어진 문제이다.

알고리즘

MatrixChain

입력: 연속된 행렬 $A_1 \times A_2 \times \dots \times A_n$, 단, A_1 은 $d_0 \times d_1$, A_2 는 $d_1 \times d_2$, ..., A_n 은 $d_{n-1} \times d_n$ 이다.

출력: 입력의 행렬 곱셈에 필요한 원소의 최소 곱셈 횟수

```
1. for i = 1 to n
2.     C[i,i] = 0
3. for L = 1 to n-1 { // L은 부분 문제의 크기를 조절하는 인덱스이다.
4.     for i = 1 to n-L {
5.         j = i + L
6.         C[i,j] = ∞
7.         for k = i to j-1 {
8.             temp = C[i,k] + C[k+1,j] + di-1dkdj
9.             if (temp < C[i,j])
10.                C[i,j] = temp
11.         }
12.     }
13. }
```

11. return C[1,n]

- Line 1~2: 배열의 대각선 원소들, 즉, $C[1,1]$, $C[2,2]$, ..., $C[n,n]$ 을 0으로 각각 초기화시킨다.
- 그 의미는 행렬 $A_1 \times A_1$, $A_2 \times A_2$, ..., $A_n \times A_n$ 을 각각 계산하는데 필요한 원소간의 곱셈 횟수가 0이란 뜻이다.
- 즉, 실제로는 아무런 계산도 필요 없다는 뜻이다. 이렇게 초기화하는 이유는 $C[i,i]$ 가 가장 작은 부분 문제의 해이기 때문이다.

- Line 3의 for-루프의 L 은 $1 \sim (n-1)$ 까지 변하는데, L 은 부분문제의 크기를 $2 \sim n$ 까지 조절하기 위한 변수이다. 즉, 이를 위해 line 4의 for-루프의 i 가 $1 \sim (n-L)$ 까지 변한다.
- $L=1$ 일 때, i 는 $1 \sim (n-1)$ 까지 변하므로, 크기가 2인 부분문제의 수가 $(n-1)$ 개이다.
- $L=2$ 일 때, i 는 $1 \sim (n-2)$ 까지 변하므로, 크기가 3인 부분문제의 수가 $(n-2)$ 개이다.
- $L=3$ 일 때, i 는 $1 \sim (n-3)$ 까지 변하므로, 크기가 4인 부분문제의 수가 $(n-3)$ 개이다.

L = 1

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-1)개

L = 2

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-2)개

L = 3

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

(n-3)개

- $L=n-2$ 일 때, i 는 $1 \sim n-(n-2)=2$ 까지 변하므로, 크기가 $(n-1)$ 인 부분 문제의 수는 2개이다.
- $L=n-1$ 일 때, i 는 $1 \sim n-(n-1)=1$ 까지 변하므로, 크기가 n 인 부분 문제의 수는 1개이다.

$L = n-1$

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

$L = n$

C	1	2	3	.	.	n-1	n
1	0						
2		0					
3			0				
.				0			
.					0		
n-1						0	
n							0

- Line 5에서는 $j=i+L$ 인데, 이는 행렬 $A_i \times \cdots \times A_j$ 에 대한 원소간의 최소 곱셈 횟수, 즉, $C[i,j]$ 를 계산하기 위한 것이다. 따라서

- $L=1$ 일 때,

- $i=1: j=1+1=2$ ($A_1 \times A_2$ 를 계산하기 위하여),

- $i=2: j=2+1=3$ ($A_2 \times A_3$ 을 계산하기 위하여),

- $i=3: j=3+1=4$ ($A_3 \times A_4$ 를 계산하기 위하여),

- ...

- $i=n-L=n-1: j=(n-1)+1=n$ ($A_{n-1} \times A_n$ 을 계산하기 위하여)

크기 2인 부분문제의 수: $(n-1)$ 개

- $L=2$ 일 때,
 - $i=1: j=1+2=3$ ($A_1 \times A_2 \times A_3$ 을 계산하기 위하여),
 - $i=2: j=2+2=4$ ($A_2 \times A_3 \times A_4$ 를 계산하기 위하여),
 - $i=3: j=3+2=5$ ($A_3 \times A_4 \times A_5$ 를 계산하기 위하여),
 - ...
 - $i=n-L=n-2: j=(n-2)+2=n$ ($A_{n-2} \times A_{n-1} \times A_n$ 을 계산하기 위해)
- 크기 3인 부분문제의 수: $(n-2)$ 개

- $L=3$ 일 때, $A_1 \times A_2 \times A_3 \times A_4, A_2 \times A_3 \times A_4 \times A_5, \dots,$
 $A_{n-3} \times A_{n-2} \times A_{n-1} \times A_n$ 을 계산한다.
 크기가 4인 부분문제의 수가 $(n-3)$ 개
 ...
- $L=n-2$ 일 때, 2개의 부분문제 $A_1 \times A_2 \times \dots \times A_{n-1},$
 $A_2 \times A_3 \times \dots \times A_n$ 을 계산한다.
- $L=n-1$ 일 때, $i=1$ 이면 $j=1+(n-1)=n$ 이고, 주어진 문제
 $A_1 \times A_2 \times \dots \times A_n$ 을 계산한다.

$$\boxed{A_1 \times A_2} \quad \boxed{A_2 \times A_3} \quad \boxed{A_3 \times A_4} \quad \cdots \quad \boxed{A_{n-2} \times A_{n-1}} \quad \boxed{A_{n-1} \times A_n} \quad n-1 \text{ 개}$$

$$\boxed{A_1 \times A_2 \times A_3} \quad \boxed{A_2 \times A_3 \times A_4} \quad \boxed{A_3 \times A_4 \times A_5} \quad \cdots \quad \boxed{A_{n-2} \times A_{n-1} \times A_n} \quad n-2 \text{ 개}$$

:

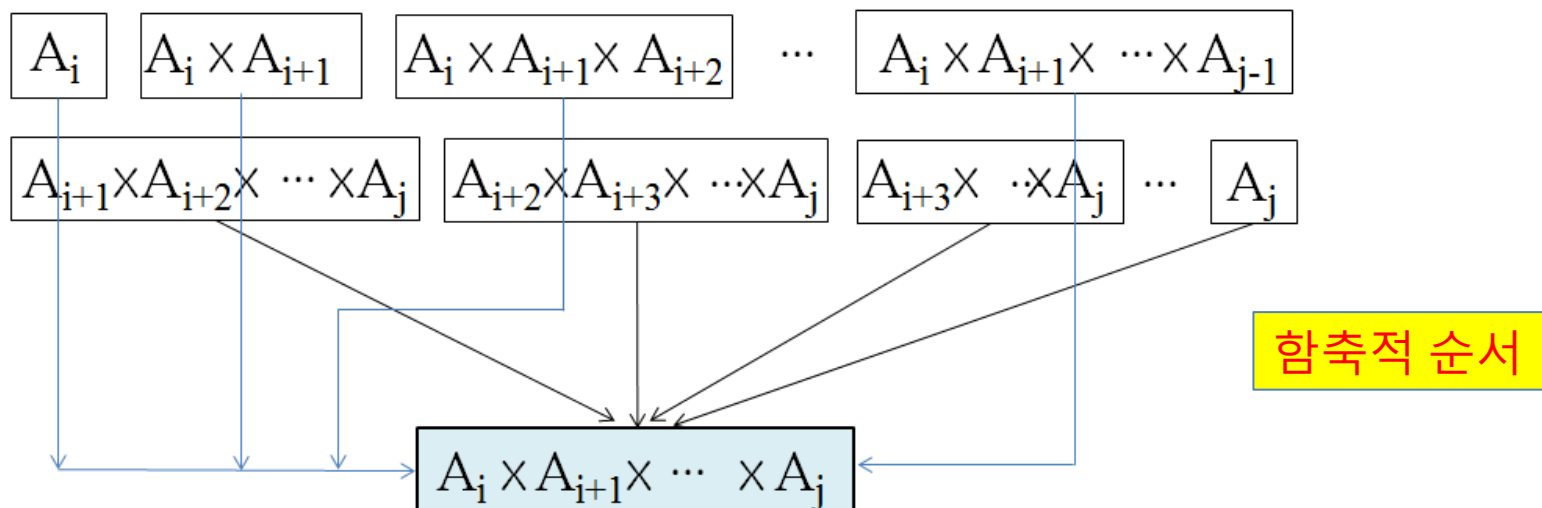
:

$$\boxed{A_1 \times A_2 \times A_3 \times \cdots \times A_{n-1}} \quad \boxed{A_2 \times A_3 \times A_4 \times \cdots \times A_n} \quad 2 \text{ 개}$$

$$\boxed{A_1 \times A_2 \times A_3 \times \cdots \times A_{n-1} \times A_n} \quad 1 \text{ 개}$$

- Line 6에서는 최소 곱셈 횟수를 찾기 위해 $C[i,j]=\infty$ 로 초기화시킨다.
- Line 7~10의 for-루프는 k 가 $i \sim (j-1)$ 까지 변하면서 어떤 부분 문제를 먼저 계산하면 곱셈 횟수가 최소인지를 찾아서 최종적으로 $C[i,j]$ 에 그 값을 저장한다.
- 즉, k 가 $A_i \times A_{i+1} \times \cdots \times A_j$ 를 2개의 부분문제로 나누어 어떤 경우에 곱셈 횟수가 최소인지를 찾는데, 여기서 부분문제간의 함축적 순서가 존재함을 알 수 있다.

$$\begin{array}{ll}
 (A_i) \times (A_{i+1} \times A_{i+2} \times \cdots \times A_j) & k=i \text{ 일때} \\
 (A_i \times A_{i+1}) \times (A_{i+2} \times A_{i+3} \times \cdots \times A_j) & k=i+1 \text{ 일때} \\
 (A_i \times A_{i+1} \times A_{i+2}) \times (A_{i+3} \times \cdots \times A_j) & k=i+2 \text{ 일때} \\
 \vdots & \vdots \\
 (A_i \times A_{i+1} \times \cdots \times A_{j-1}) \times (A_j) & k=j-1 \text{ 일때}
 \end{array}$$



- Line 8: 2개의 부분문제로 나뉜 각각의 경우에 대한 곱셈 횟수를 계산한다.
 - 첫 번째 부분문제의 해는 $C[i,k]$ 에 있고,
 - 두 번째 부분문제의 해는 $C[k+1,j]$ 에 있으며,
 - 2개의 해를 합하고, 이에 $d_{i-1}d_kd_j$ 를 더한다.
 - 여기서 $d_{i-1}d_kd_j$ 를 더하는 이유는 두 부분문제들이 각각 $d_{i-1} \times d_k$ 행렬과 $d_k \times d_j$ 행렬이고,
 - 두 행렬을 곱하는데 필요한 원소간의 곱셈 횟수가 $d_{i-1}d_kd_j$ 이기 때문이다.

- 다음은 k값의 변화에 따른 2개의 부분문제에 해당하는 행렬을 각각 보여주고 있다.

$$\begin{array}{ccc} (A_i) & \times & (A_{i+1} \times A_{i+2} \times \cdots \times A_j) \\ d_{i-1} \times d_i & & d_i \times d_j \end{array} \quad k=i \text{ 일때}$$

$$\begin{array}{ccc} (A_i \times A_{i+1}) & \times & (A_{i+2} \times A_{i+3} \times \cdots \times A_j) \\ d_{i-1} \times d_{i+1} & & d_{i+1} \times d_j \end{array} \quad k=i+1 \text{ 일때}$$

$$\begin{array}{ccc} (A_i \times A_{i+1} \times A_{i+2}) & \times & (A_{i+3} \times \cdots \times A_j) \\ d_{i-1} \times d_{i+2} & & d_{i+2} \times d_j \end{array} \quad k=i+2 \text{ 일때}$$

:

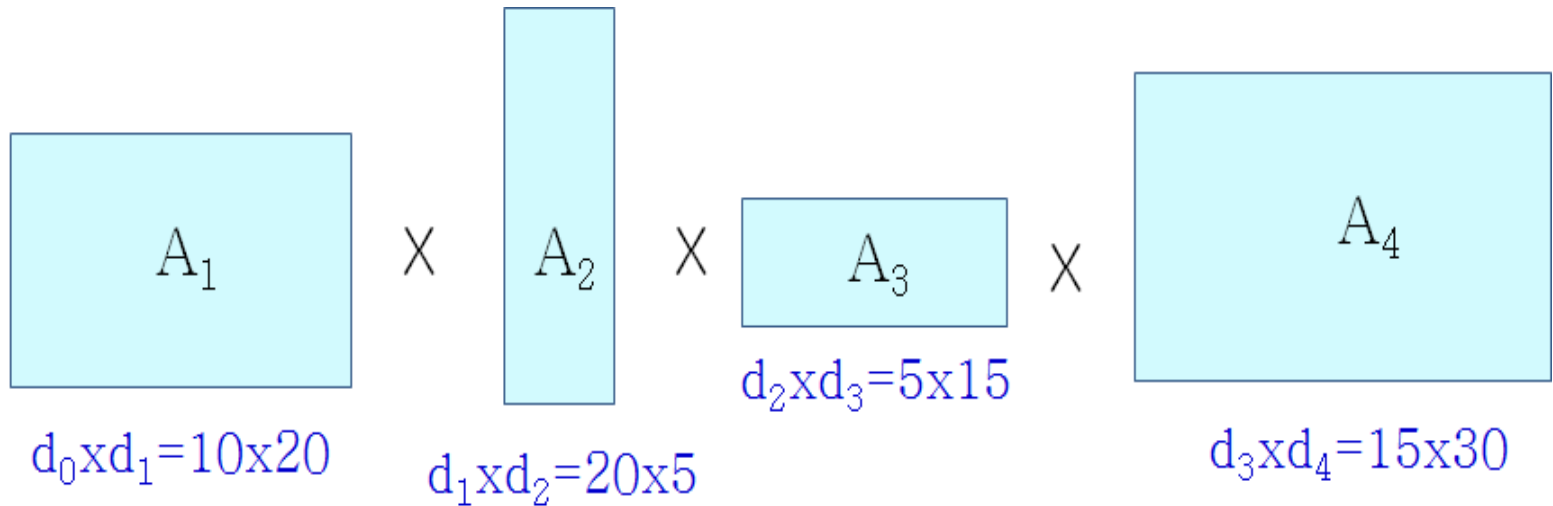
:

$$\begin{array}{ccc} (A_i \times A_{i+1} \times \cdots \times A_{j-1}) & \times & (A_j) \\ d_{i-1} \times d_{j-1} & & d_{j-1} \times d_j \end{array} \quad k=j-1 \text{ 일때}$$

- Line 9~10: line 8에서 계산된 곱셈 횟수가 바로 직전까지 계산되어 있는 $C[i,j]$ 보다 작으면 그 값으로 $C[i,j]$ 를 갱신하며, $k=(j-1)$ 일 때까지 수행되면 최종적으로 가장 작은 값이 $C[i,j]$ 에 저장된다.
- Line 11: 주어진 문제의 해가 있는 $C[1,n]$ 을 리턴

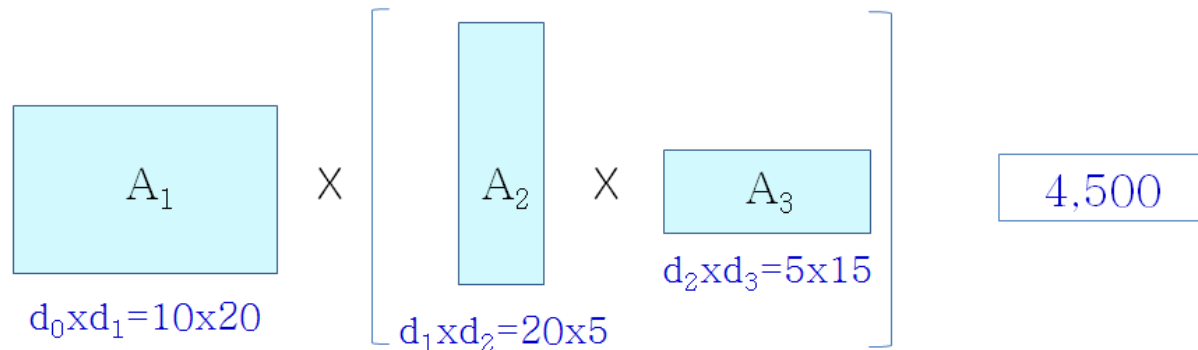
MatrixChain 알고리즘의 수행 과정

- A_1 이 10x20, A_2 가 20x5, A_3 이 5x15, A_4 가 15x30이다.

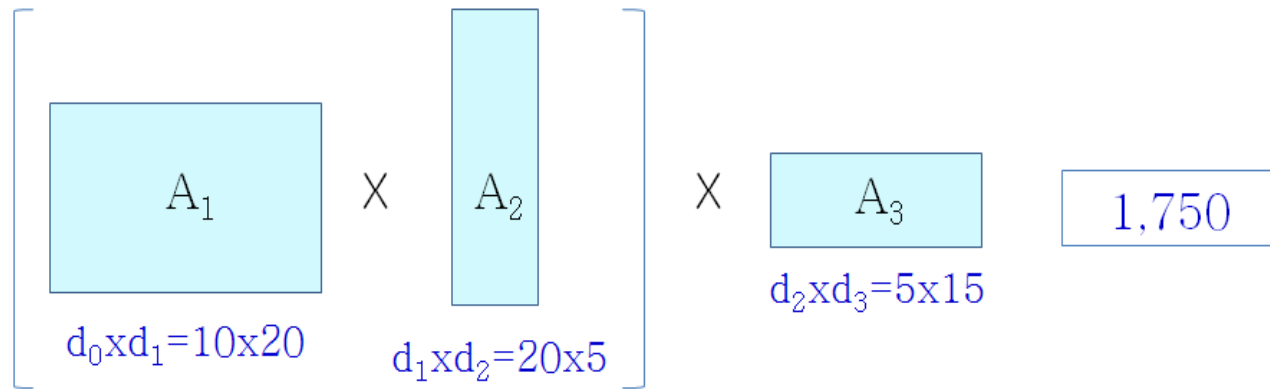


- Line 1~2에서 $C[1,1]=C[2,2]=C[3,3]=C[4,4]=0$ 으로 초기화
- Line 3에서 L 이 $1\sim(4-1)=3$ 까지 변하고, 각각의 L 값에 대하여, i 가 변화하며 $C[i,j]$ 를 계산한다.
 - $L=1$ 일 때, i 는 $1\sim(n-L)=4-1=3$ 까지 변한다.
 - $i=1$ 이면 $j=1+1=2$ 이므로, $A_1 \times A_2$ 를 위해 line 6에서 $C[1,2]=\infty$ 로 초기화하고, line 7의 k 는 $1\sim(j-1)=2-1=1$ 까지 변하므로 사실 $k=1$ 일 때 1번만 수행된다. Line 8에서 $\text{temp} = C[1,1] + C[2,2] + d_0d_1d_2 = 0+0+(10 \times 20 \times 5) = 1,000$ 이 되고, line 9에서 현재 $C[1,2]=\infty$ 가 temp 보다 크므로, $C[1,2]=1,000$ 이 된다.
 - $i=2$ 이면 $j=2+1=3$ 이므로, $A_2 \times A_3$ 을 위해 line 6에서 $C[2,3]=\infty$ 로 초기화하고, line 7의 k 는 $2\sim(j-1)=3-1=2$ 까지 변하므로 $k=2$ 일 때 역시 1번만 수행된다. Line 8에서 $\text{temp} = C[2,2] + C[3,3] + d_1d_2d_3 = 0+0+(20 \times 5 \times 15) = 1,500$ 이 되고, line 9에서 현재 $C[2,3]=\infty$ 가 temp 보다 크므로, $C[2,3]=1,500$ 이 된다.
 - $i=3$ 이면 $A_3 \times A_4$ 에 대해 $C[3,4] = 2,250$ 이 된다.

- $L=2$ 일 때 i 는 $1 \sim (n-L)=4-2=2$ 까지 변한다.
 - $i=1$ 이면 $j=1+2=3$ 이므로, $A_1 \times A_2 \times A_3$ 을 계산하기 위해 line 6에서 $C[1,3]=\infty$ 로 초기화하고, line 7의 k 는 $1 \sim (j-1)=3-1=2$ 까지 변하므로, $k=1$ 과 $k=2$ 일 때 2번 수행된다.
 - $k=1$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[1,1] + C[2,3] + d_0d_1d_3 = 0 + 1,500 + (10 \times 20 \times 15) = 4,500$ 이 되고, line 9에서 현재 $C[1,3]=\infty$ 이고 temp 보다 크므로, **$C[1,3]=4,500$** 이 된다.

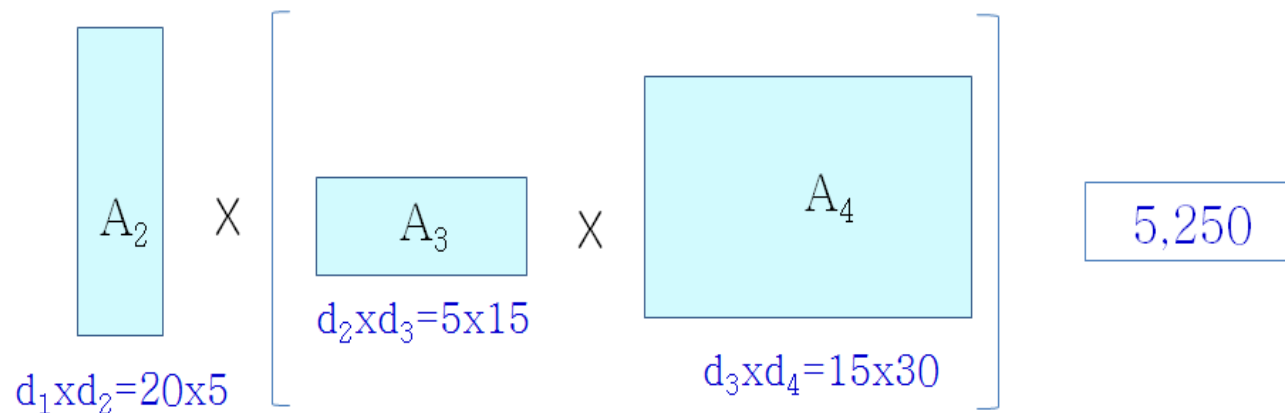


- $k=2$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[1,2] + C[3,3] + d_0d_2d_3 = 1,000 + 0 + (10 \times 5 \times 15) = 1,750$ | 되고, line 9에서 현재 $C[1,3] = 4,500$ 인데 temp보다 크므로, $C[1,3] = 1,750$ 으로 갱신된다.

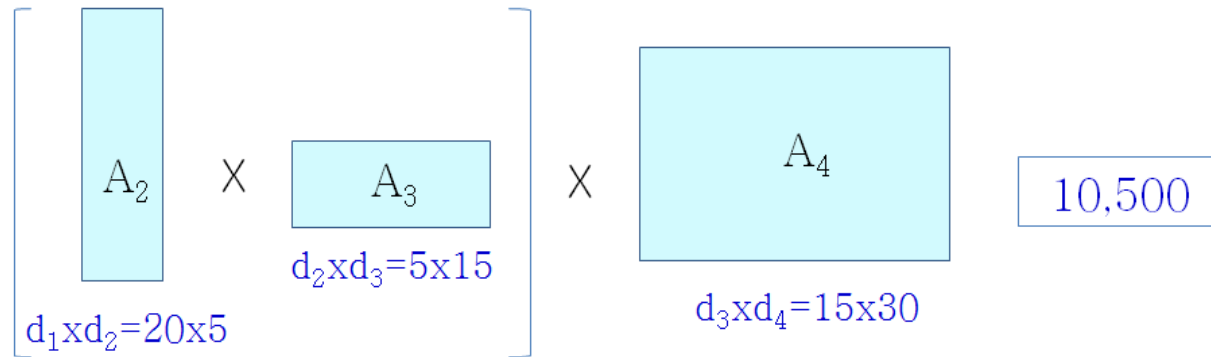


– $i=20$ 이면 $j=2+2=40$ 이므로, $A_2 \times A_3 \times A_4$ 를 계산하기 위해 line 6에서 $C[2,4]=\infty$ 로 초기화하고, line 7의 k 는 $2 \sim (j-1)=3$ 까지 변하므로, $k=2$ 와 $k=3$ 일 때 2번 수행된다.

- $k=2$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[2,2] + C[3,4] + d_1d_2d_4 = 0 + 2,250 + (20 \times 5 \times 30) = 5,250$ 이 되고, line 9에서 현재 $C[2,4]=\infty$ 가 temp 보다 크므로, **$C[2,4] = 5,250$** 이 된다.

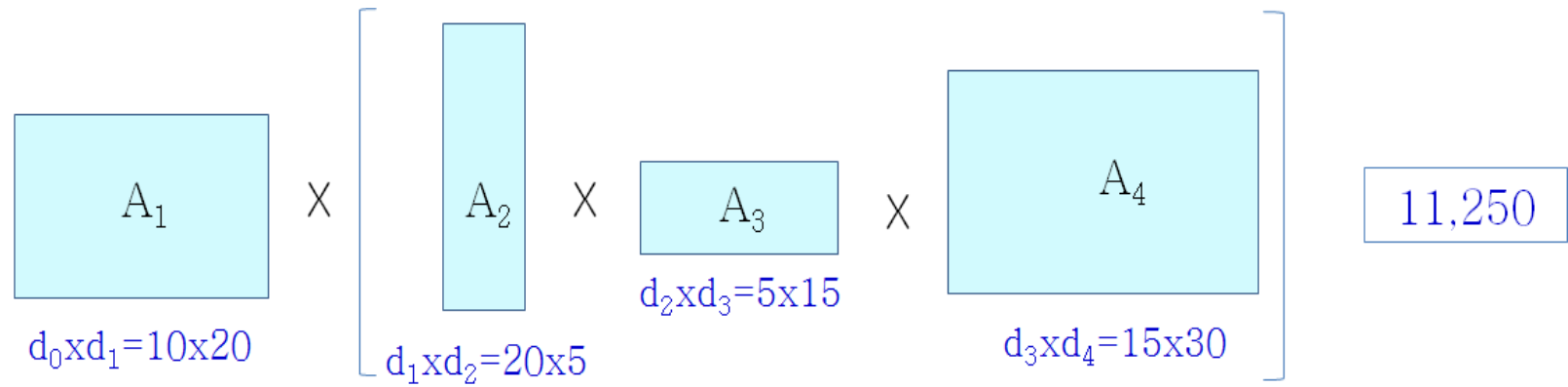


- $k=3$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[2,3] + C[4,4] + d_1d_3d_4 = 1,500 + 0 + (20 \times 15 \times 30) = 10,500$ 이 된다.

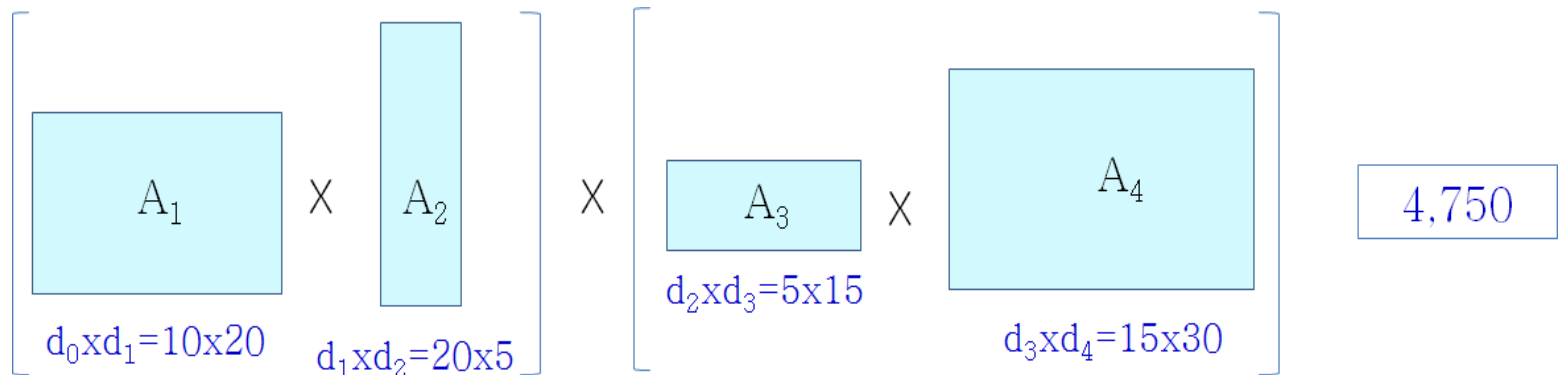


그러나 line 9에서 현재 $C[2,4]=5,250$ 이 temp 보다 작으므로, 그대로 $C[2,4]=5,250$ 이다.

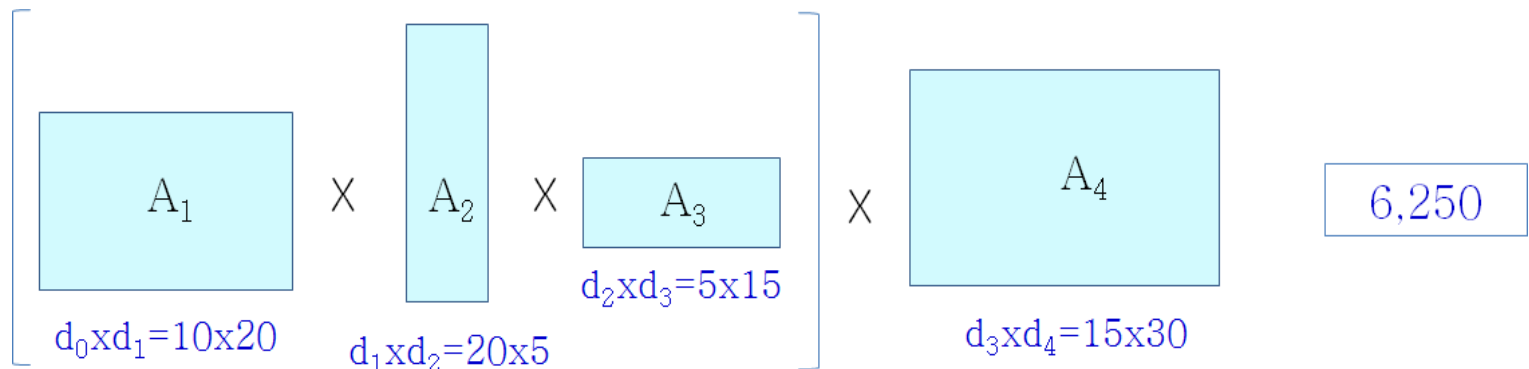
- $L=3$ 일 때 i 는 $1 \sim (n-L)=4-3=1$ 까지 이므로 $i=1$ 일 때만 수행된다.
 - $i=1$ 이면 $j=1+3=4$ 이므로, $A_1 \times A_2 \times A_3 \times A_4$ 를 계산하기 위해 line 6에서 $C[1,4]=\infty$ 로 초기화하고, line 7의 k 는 $1 \sim (j-1)=4-1=3$ 까지 변하므로, $k=1, k=2, k=3$ 일 때 각각 수행된다.
 - $k=1$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[1,1] + C[2,4] + d_0d_1d_4 = 0 + 5,250 + (10 \times 20 \times 30) = 11,250$ 이 되고, line 9에서 현재 $C[1,4]=\infty$ 가 temp 보다 크므로, $C[1,4]=11,250$ 이 된다.



- $k=2$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[1,2] + C[3,4] + d_0d_2d_4 = 1,000 + 2,250 + (10 \times 5 \times 30) = 4,750$ 이 되고, line 9에서 현재 $C[1,4] = 11,250$ 이 temp보다 크므로, $C[1,4] = 4,750$ 으로 갱신된다.



- $k=3$ 일 때, line 8에서 $\text{temp} = C[i,k] + C[k+1,j] + d_{i-1}d_kd_j = C[1,3] + C[4,4] + d_0d_3d_4 = 1,750 + 0 + (10 \times 15 \times 30) = 6,250$ 이 된다.



- 그러나 line 9에서 현재 $C[1,4] = 4,750$ 이 temp보다 작으므로, 그대로 $C[1,4]=4,750$ 이다.
- 따라서 최종해는 4,750번이다. 먼저 $A_1 \times A_2$ 를 계산하고, 그 다음엔 $A_3 \times A_4$ 를 계산하여, 각각의 결과를 곱하는 것이 가장 효율적이다. 다음은 알고리즘이 수행된 후의 배열 c이다.

C	1	2	3	4
1	0	1,000	1,750	4,750
2		0	1,500	5,250
3			0	2,250
4				0

셸 정렬

- 버블 정렬이나 삽입 정렬이 수행되는 과정을 살펴보면, 이웃하는 원소끼리의 자리이동으로 원소들이 정렬된다.
- 버블 정렬이 오름차순으로 정렬하는 과정을 살펴보면, 작은 (가벼운) 숫자가 배열의 앞부분으로 매우 느리게 이동하는 것을 알 수 있다.
- 예를 들어, 삽입 정렬의 경우 만일 배열의 마지막 원소가 입력에서 가장 작은 숫자라면, 그 숫자가 배열의 맨 앞으로 이동해야 하므로, 모든 다른 숫자들이 1칸씩 오른쪽으로 이동해야 한다.

- 셸 정렬 (Shell sort)은 이러한 단점을 보완하기 위해서 삽입 정렬을 이용하여 배열 뒷부분의 작은 숫자를 앞부분으로 '빠르게' 이동시키고, 동시에 앞부분의 큰 숫자는 뒷부분으로 이동시키고, 가장 마지막에는 삽입 정렬을 수행하는 알고리즘이다.

- 다음의 예제를 통해 쉘 정렬의 아이디어를 이해해보자.

30 60 90 10 40 80 40 20 10 60 50 30 40 90 80

- 먼저 간격 (gap)이 5가 되는 숫자끼리 그룹을 만든다.
- 총 15개의 숫자가 있으므로, 첫 번째 그룹은 첫 숫자인 30, 첫 숫자에서 간격이 5되는 숫자인 80, 그리고 80에서 간격이 5인 50으로 구성된다.

- 즉, 첫 번째 그룹은 [30, 80, 50]이다. 2 번째 그룹은 [60, 40, 30]이고, 나머지 그룹은 각각 [90, 20, 40], [10, 10, 90], [40, 60, 80]이다.

h=5

A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	30					80					50				
2		60					40					30			
3			90					20					40		
4				10					10					90	
5					40					60					80

- 각 그룹 별로 삽입 정렬을 수행한 결과를 1줄에 나열해보면 다음과 같다.

30 30 20 10 40 50 40 40 10 60 80 60 90 90 80

A 그룹	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	30	30				50	40				80	60			
			20	10				40					90		
					40				10	60				90	80

그룹별 정렬 후

- 간격이 5인 그룹 별로 정렬한 결과를 살펴보면, 80과 90같은 큰 숫자가 뒷부분으로 이동하였고, 20과 30같은 작은 숫자가 앞부분으로 이동한 것을 관찰할 수 있다.

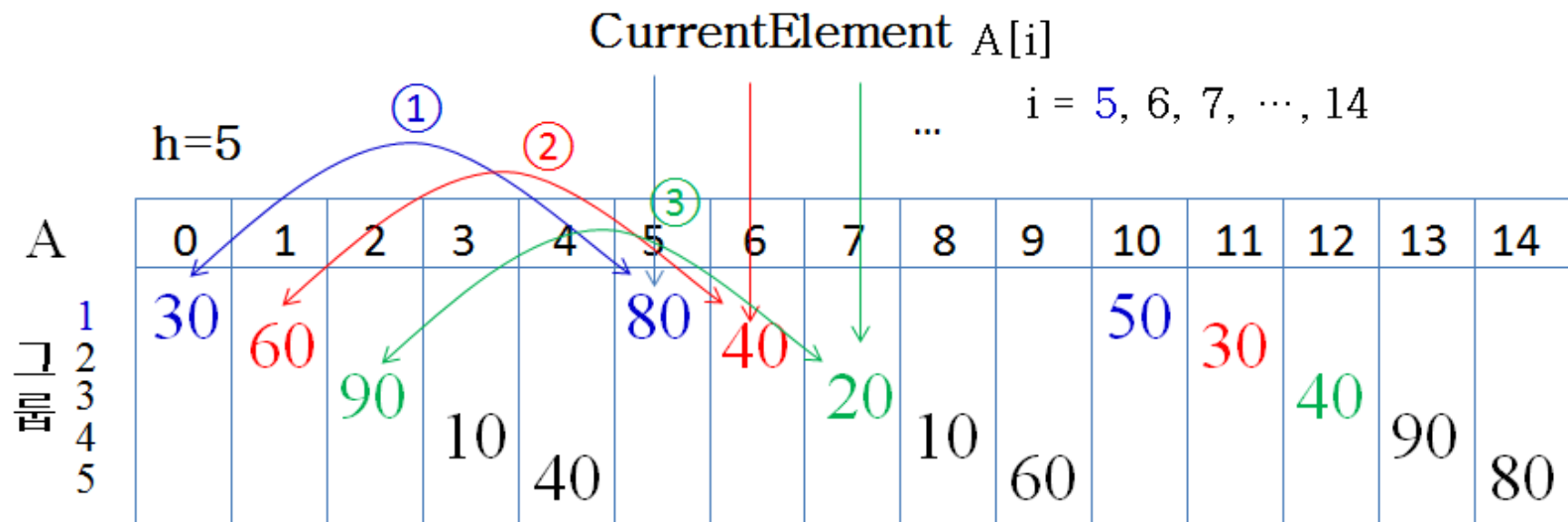
- 그 다음엔 간격을 5보다 작게 하여, 예를 들어, 3으로 하여, 3개의 그룹으로 나누어 각 그룹별로 삽입 정렬을 수행한다.
- 이때에는 각 그룹에 5개의 숫자가 있다. 마지막에는 반드시 간격을 1로 놓고 수행해야 한다.
- 왜냐하면 다른 그룹에 속해 서로 비교되지 않은 숫자가 있을 수 있기 때문이다.
- 즉, 모든 원소를 1개의 그룹으로 여기는 것이고, 이는 삽입 정렬 그 자체이다.

ShellSort 알고리즘

1. for each gap $h = [h_0 > h_1 > \dots > h_k = 1]$ // 큰 gap부터 차례로
2. for $i = h$ to $n-1$ {
3. CurrentElement=A[i];
4. $j = i$;
5. while ($j \geq h$) and ($A[j-h] > \text{CurrentElement}$) {
6. $A[j] = A[j-h]$;
7. $j = j - h$;
8. }
9. $A[j] = \text{CurrentElement}$;
10. }
11. return 배열 A

- 셀 정렬은 간격 [$h_0 > h_1 > \dots > h_k=1$]이 미리 정해져야 한다.
- 가장 큰 간격 h_0 부터 차례로 간격에 따른 삽입 정렬이 line 2~8에서 수행된다.
- 마지막 간격 h_k 는 반드시 1이어야 한다.
 - 이는 간격에 대해서 그룹별로 삽입 정렬을 수행하였기 때문에, 아직 비교 되지 않은 다른 그룹의 숫자가 있을 수 있기 때문이다.

- Line 2~8의 for-루프에서는 간격 h 에 대하여 삽입 정렬이 수행되는데, 핵심 아이디어에서 설명한대로 그룹 별로 삽입 정렬을 수행하지만 자리바꿈을 위한 원소간의 비교는 다음과 같은 순서로 진행된다.



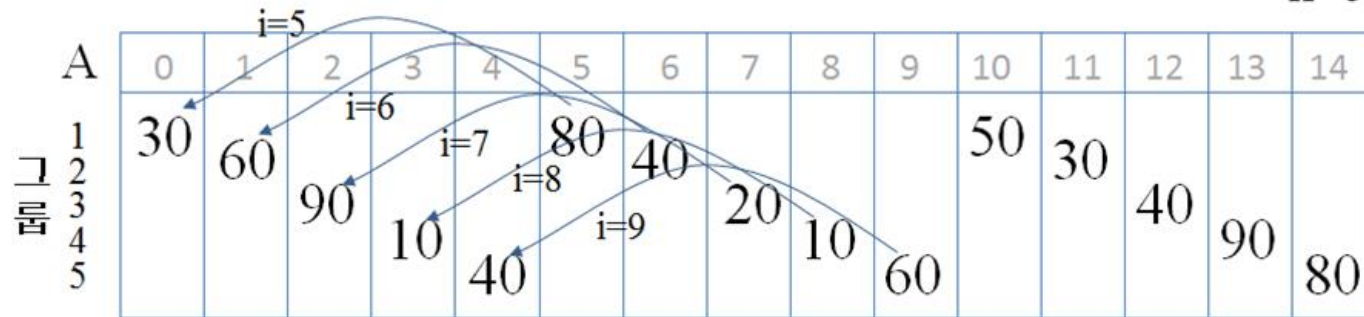
- 즉, line 2의 for-루프가 i 를 h 부터 1씩 증가시켜서 CurrentElement $A[i]$ 를 자신의 그룹에 속한 원소끼리 비교하도록 조절한다.
- Line 5~7의 while-루프에서 CurrentElement를 정렬이 되도록 앞부분에 삽입한다.
 - while-루프의 첫 번째 조건 ($j \geq h$)는 j 가 h 보다 작으면 배열 인덱스 ($j-h$)가 음수가 되어, 배열의 범위를 벗어나는 것을 검사하기 위한 것이다.
 - 2번째 조건인 ($A[j-h] > \text{CurrentElement}$)는 ($j-h$)가 음수가 아니면 CurrentElement를 자신의 그룹 원소인 $A[j-h]$ 와 비교하여 크면 line 6에서 $A[j-h]$ 를 h 만큼 뒤로 이동 (즉, $A[j]=A[j-h]$)시킨다.

- while-루프의 조건이 '거짓'이 되면 line 8에서 CurrentElement를 $A[j]$ 에 저장한다. 여기서 while-루프의 조건이 '거짓'이 될 때
 - 첫 번째 경우는 $(j-h)$ 가 음수인 경우인데, 이는 $A[j]$ 앞에 같은 그룹의 원소가 없다는 뜻이다.
 - 두 번째 경우는 $A[j-h]$ 가 CurrentElement와 같거나 작은 경우이다.
- 따라서 두 경우 모두 line 8에서 CurrentElement를 $A[j]$ 에 저장하면 알맞게 삽입된다.

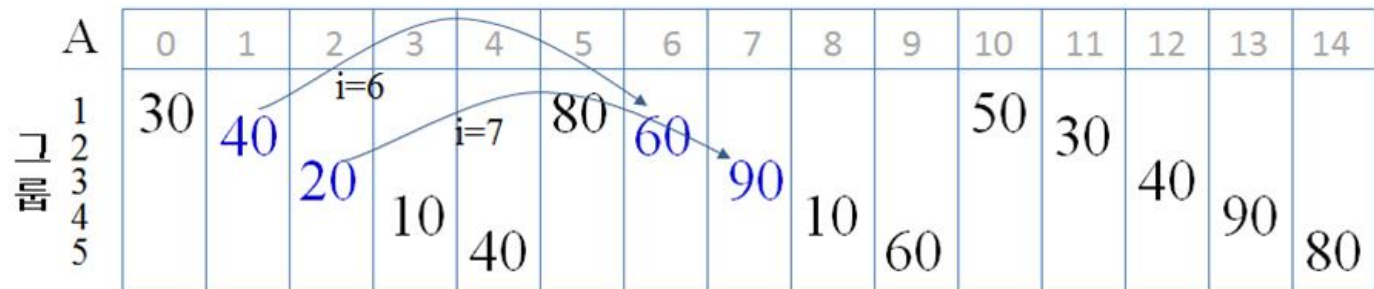
- 간격이 5일 때, 셀 정렬의 수행 과정

$i = 5, 6, 7, 8, 9$ 일 때

$h = 5$




이동 결과



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
A	1	30															
	2		30														
	3			20													
	4				10												
	5					40											

Diagram illustrating the insertion of element 30 into a binary search tree. The tree structure is shown with nodes and their values. The root node is 30. Its left child is 20, and its right child is 40. Node 20 has a left child 10, and node 40 has a left child 30. Node 10 has a right child 60. Node 60 has a right child 80. Node 80 has a right child 90. Node 90 has a right child 80. The diagram shows the insertion of element 30 into the tree, with the new node 30 being added as the left child of node 40. The diagram also shows the traversal of the tree to find the insertion point, with the path 30 → 40 → 30 being highlighted. The diagram includes labels for the nodes and their values, and arrows indicating the traversal path.

- $h=5$ 일 때의 결과를 한 줄에 나열해보면 다음과 같다.
30 30 20 10 40 50 40 40 10 60 80 60 90 90 80
- 이제 간격을 줄여서 $h=3$ 이 되면, 배열의 원소가 3개의 그룹으로 나누어진다.
 - 그룹1은 0번째, 3번째, 6번째, 9번째, 12번째 숫자이고,
 - 그룹2는 1번째, 4번째, 7번째, 10번째, 13번째 숫자로 구성되고,
 - 마지막으로 그룹3은 2번째, 5번째, 8번째, 11번째, 14번째 숫자이다.
- 각 그룹 별로 삽입 정렬하면,

그룹1	그룹2	그룹3		그룹1	그룹2	그룹3
30	30	20		10	30	10
10	40	50		30	40	20
40	40	10		40	40	50
60	80	60		60	80	60
90	90	80		90	90	80

- 각 그룹 별로 정렬한 결과를 한 줄에 나열해보면 다음과 같다. 즉, 이것이 $h=3$ 일 때의 결과이다.

10 30 10 30 40 20 40 40 50 60 80 60 90 90 80

- 마지막으로 위의 배열에 대해 $h=1$ 일 때 알고리즘을 수행하면 아래와 같이 정렬된 결과를 얻는다.
- $h=1$ 일 때는 간격이 1 (즉, 그룹이 1개)이므로, 삽입 정렬과 동일하다.

10 10 20 30 30 40 40 40 50 60 60 80 80 90 90

- 쉘 정렬의 수행 속도는 **간격 선정에 따라 좌우**된다.
- 지금까지 알려진 **가장 좋은 성능**을 보인 간격:
 - 1, 4, 10, 23, 57, 132, 301, 701
 - 701 이후는 아직 밝혀지지 않았다.

응 용

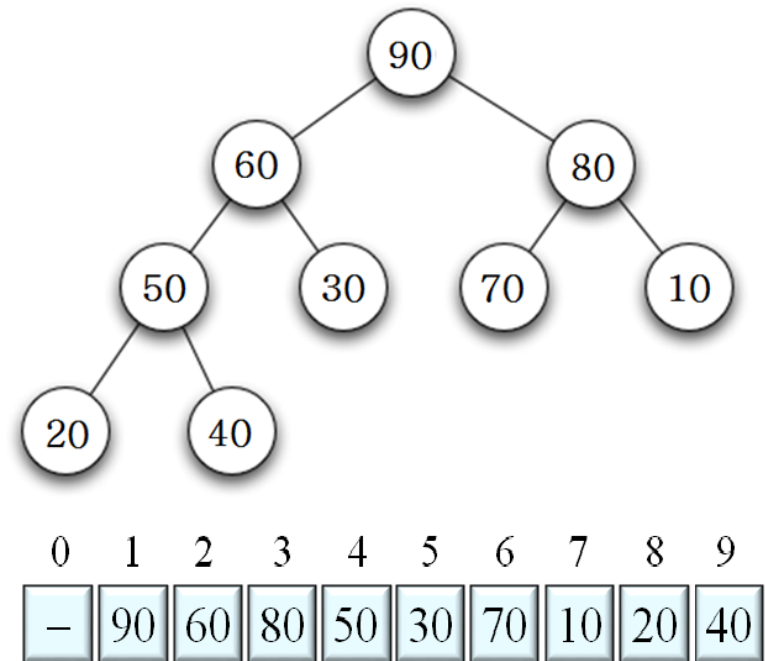
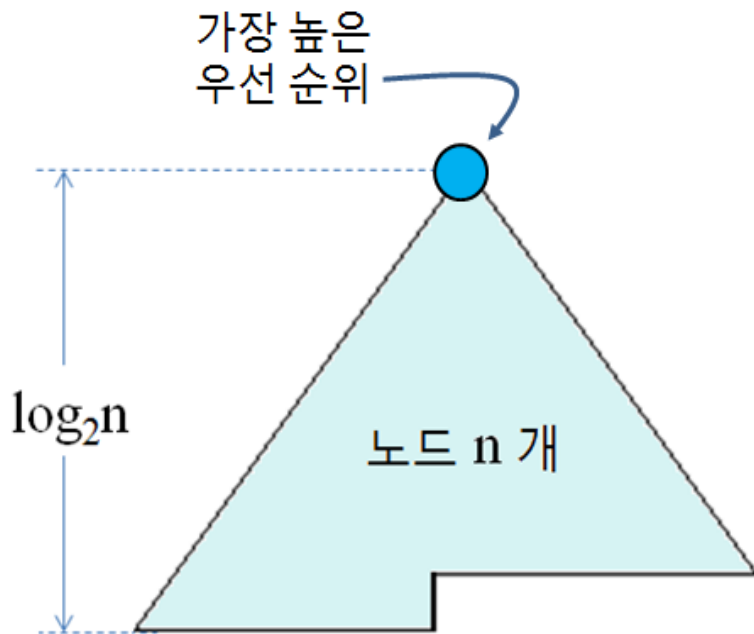
- 쉘 정렬은 입력 크기가 매우 크지 않은 경우에 매우 좋은 성능을 보인다.
- 쉘 정렬은 임베디드(Embedded) 시스템에서 주로 사용되는데, 쉘 정렬의 특징인 **간격에 따른 그룹 별 정렬 방식**이 H/W로 정렬 알고리즘을 구현하는데 매우 적합하기 때문이다.



힙 정렬

- 힙 (heap)은 힙 조건을 만족하는 완전 이진트리 (Complete Binary Tree)이다.
- 힙 조건이란 각 노드의 값이 자식 노드의 값보다 커야 한다는 것을 말한다.
- 노드의 값은 우선순위 (priority)라고 일컫는다.
- 힙의 루트에는 가장 높은 우선순위 (가장 큰 값)가 저장되어 있다.
- 값이 작을수록 우선순위가 높은 경우에는 가장 작은 값이 루트에 저장된다.

- n 개의 노드를 가진 힙은 완전 이진트리이므로, **힙의 높이**가 $\log_2 n$ 이며, 노드들을 빈 공간 없이 배열에 저장할 수 있다.
- 아래의 그림은 힙의 노드들이 배열에 저장된 모습을 보여주고 있다.



- 배열 A에 힙을 저장한다면, A[0]은 비워 두고, A[1]~A[n]까지에 힙 노드들을 트리의 층별로 좌우로 저장한다.
 - 루트의 90이 A[1]에 저장되고,
 - 그 다음 층의 60과 80이 각각 A[2]와 A[3]에 저장되며,
 - 그 다음 층의 50, 30, 70, 10이 A[4]에서 A[7]에 각각 저장되고,
 - 마지막으로 20과 40이 A[8]과 A[9]에 저장된다.

- 힙에서 부모 노드와 자식노드의 관계를 배열의 인덱스로
- $A[i]$ 의 부모 노드 = $A[i/2]$
 - 단, i 가 홀수일 때, $i/2$ 에서 정수 부분만을 취한다. 예를 들어, $A[7]$ 의 부모 노드는 $A[7/2] = A[3]$ 이다.
- $A[i]$ 의 왼쪽 자식 노드 = $A[2i]$
- $A[i]$ 의 오른쪽 자식 노드 = $A[2i+1]$
 - $A[4]$ 의 왼쪽 자식 노드는 $A[2i] = A[2 \times 4] = A[8]$ 이고, 오른쪽 자식 노드는 $A[2i+1] = A[2 \times 4 + 1] = A[9]$ 이다.

- 힙 정렬(Heap Sort)은 힙 자료 구조를 이용하는 정렬 알고리즘이다.
- 오름차순의 정렬을 위해 최대힙(maximum heap) 구성한다.
- 힙의 루트에는 가장 큰 수가 저장되므로, 루트의 숫자를 힙의 가장 마지막 노드에 있는 숫자와 바꾼다.
- 즉, 가장 큰 수를 배열의 가장 끝으로 이동시킨 것이다.
- 그리고 루트에 새로 저장된 숫자로 인해 위배된 힙 조건을 해결하여 힙 조건을 만족시키고, 힙 크기를 1개 줄인다.
- 그리고 이 과정을 반복하여 나머지 숫자를 정렬한다.
다음은 이러한 과정에 따른 힙 정렬 알고리즘이다.

HeapSort 알고리즘

입력: 입력이 A[1]부터 A[n]까지 저장된 배열 A

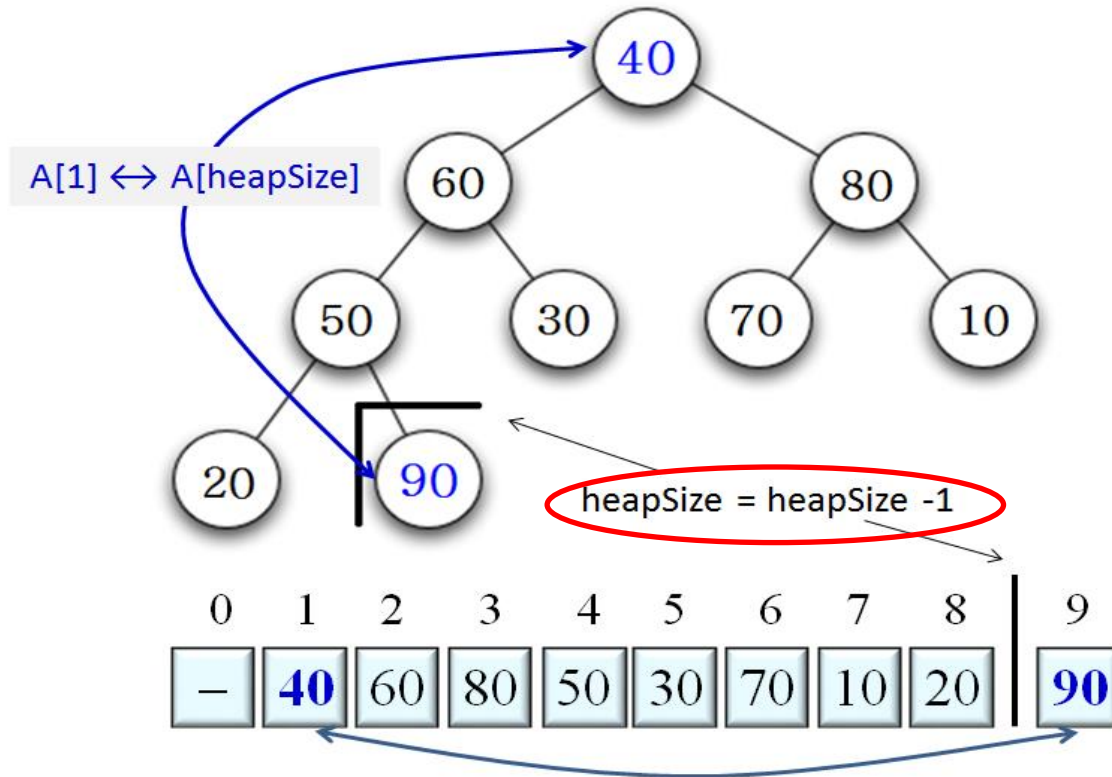
출력: 정렬된 배열 A

1. 배열 A의 숫자에 대해서 힙 자료 구조를 만든다.
2. heapSize = n // 힙의 크기를 조절하는 변수
3. for i = 1 to n-1
4. $A[1] \leftrightarrow A[\text{heapSize}]$ // 루트와 힙의 마지막 노드 교환
5. heapSize = heapSize - 1 // 힙의 크기를 1 감소
6. DownHeap() // 위배된 힙 조건을 만족시킨다.
7. return 배열 A

- Line 1: 배열 $A[1..n]$ 을 힙으로 만든다.
- Line 2: 현재의 힙의 크기를 나타내는 변수인 `heapSize`를 n 으로 초기화시킨다.
- Line 3~6의 for-루프는 $(n-1)$ 번 수행되는데, 이는 루프가 종료된 후에는 루트인 $A[1]$ 홀로 힙을 구성하고 있고, 또 가장 작은 수이므로 루프를 수행할 필요가 없기 때문이다.
- Line 4: 루트와 힙의 마지막 노드와 교환한다.
 - 즉, 현재의 힙에서 가장 큰 수와 현재 힙의 맨 마지막 노드에 있는 숫자와 교환하는 것이다.
- Line 5: 힙의 크기를 1 줄인다.

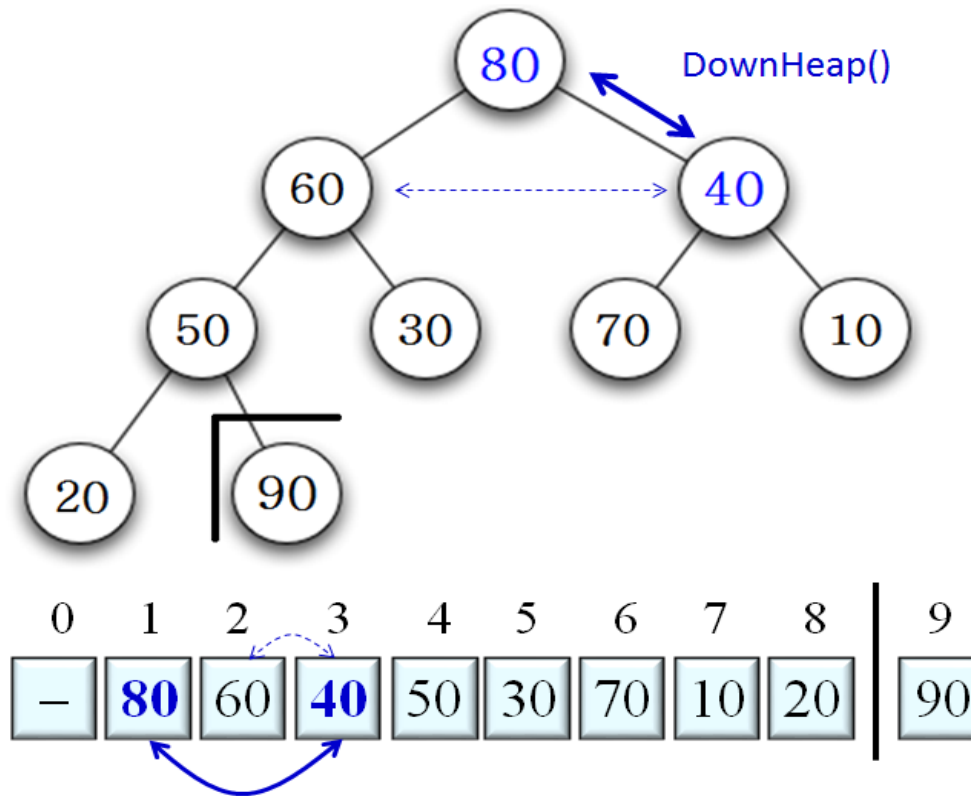
- Line 4에서 힙의 마지막 노드와 힙의 루트를 바꾸어 놓았기 때문에 새로이 루트에 저장된 값이 루트의 자식 노드의 값보다 작아서 힙 조건이 위배된다.
- Line 6에서는 위반된 힙 조건을 DownHeap을 수행시켜서 해결한다.
- DownHeap: 루트가 r 을 가지고 있다고 가정하자.
 - 먼저 루트의 r 과 자식 노드들 중에서 큰 것을 비교하여 큰 것과 r 을 바꾼다.
 - 다시 r 을 자식 노드들 중에서 큰 것과 비교하여 힙 조건이 위배되면, 앞서 수행한 대로 큰 것과 r 을 교환한다.
 - 힙 조건이 만족될 때까지 이 과정을 반복한다.

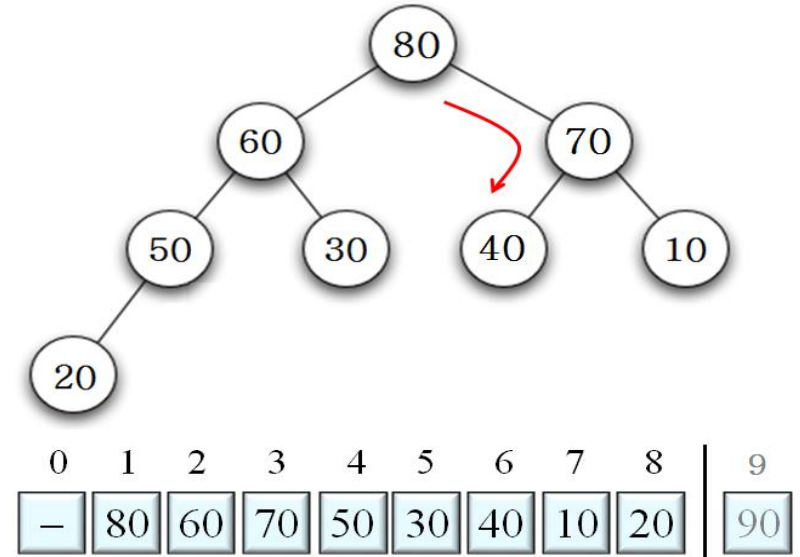
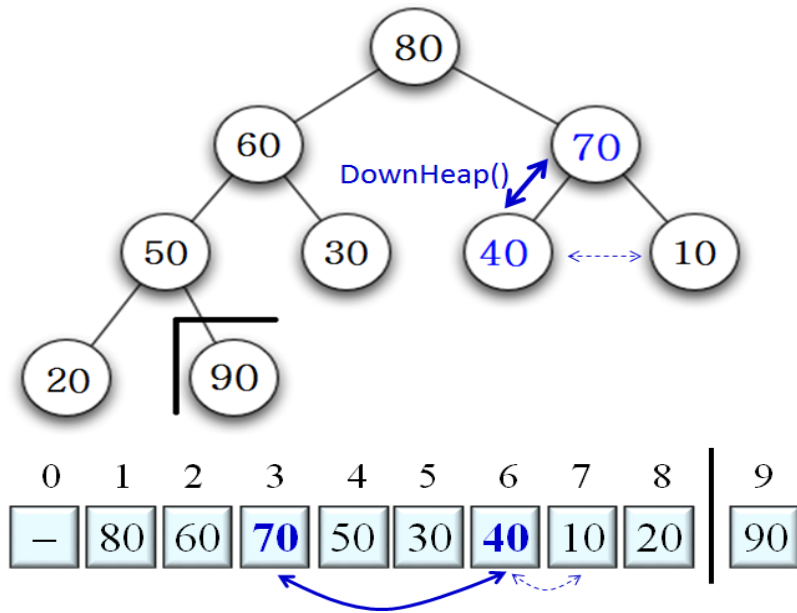
- DownHeap 실행 과정



- Line 4에서 힙의 마지막 노드 40과 루트 90을 바꾸고, 힙의 노드 수(heapsize)가 1개 줄어든 것을 보이고 있다.

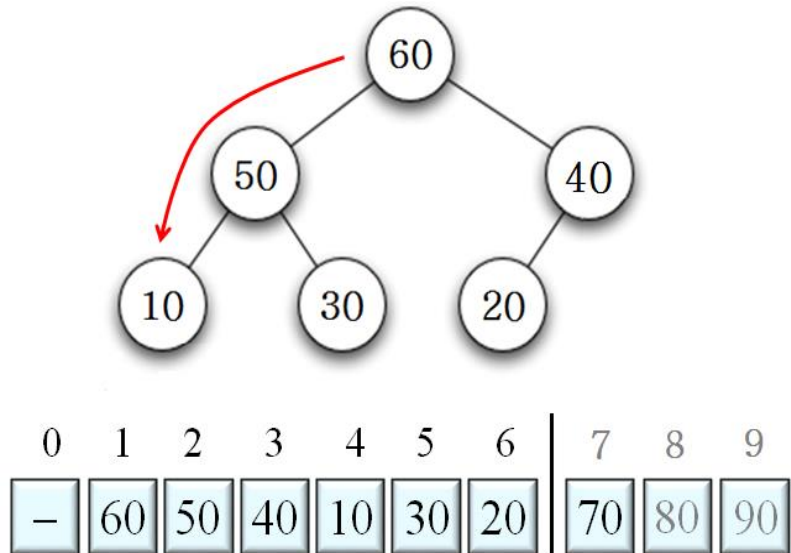
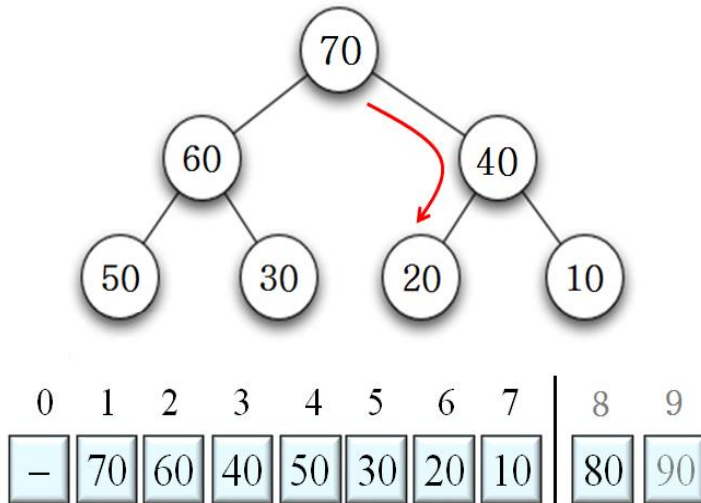
- 다음 그림은 새로이 루트에 저장된 40이 루트의 자식 노드들 (60과 80)보다 작아서 힙 조건이 위배되므로 자식 노드들 중에서 큰 자식 노드 80과 루트 40이 교환된 것을 보여준다.



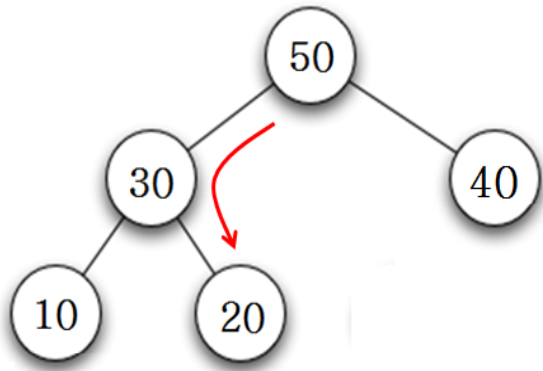


- 40은 다시 자식 노드들 (70과 10) 중에서 큰 자식 70과 비교하여, 힙 조건이 위배되므로 70과 40을 서로 바꾼다.
- 그 다음엔 더 이상 자식 노드가 없으므로 힙 조건이 만족되므로 `DownHeap`을 종료한다.

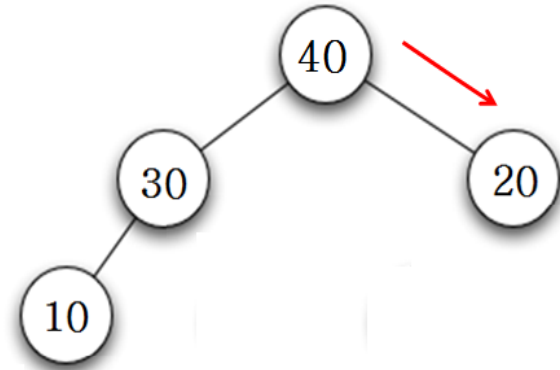
- DownHeap 예제 이은 HeapSort 수행



- 80이 20과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 70이 10과 교환된 후 DownHeap을 수행한 결과 (오른쪽)

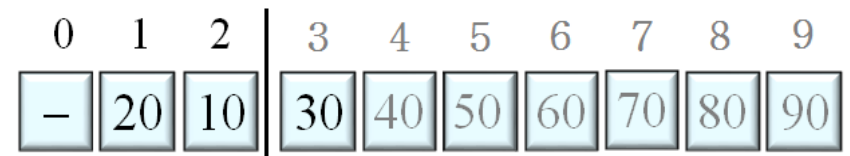
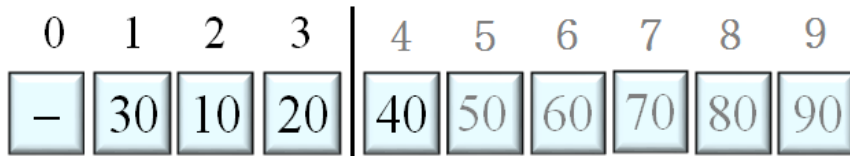
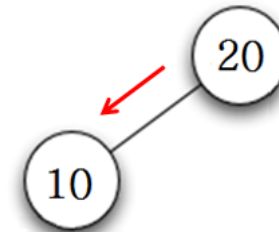
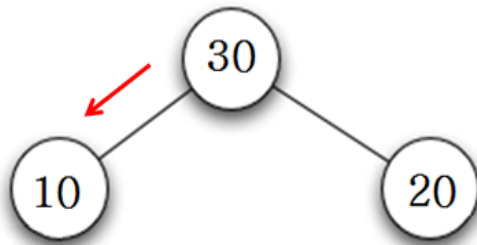


0	1	2	3	4	5	6	7	8	9
-	50	30	40	10	20	60	70	80	90



0	1	2	3	4	5	6	7	8	9
-	40	30	20	10	50	60	70	80	90

- 60이 20과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 50이 20과 교환된 후 DownHeap을 수행한 결과 (오른쪽)



- 40이 10과 교환된 후 DownHeap을 수행한 결과 (왼쪽)
- 30이 10과 교환된 후 DownHeap을 수행한 결과 (오른쪽)

0	1	2	3	4	5	6	7	8	9
-	10	20	30	40	50	60	70	80	90

- 마지막에 힙의 크기가 1이 되면 힙 정렬을 마친다.
- for-루프를 반복할 때마다 힙에서 가장 큰 수를 힙의 마지막 노드에 있는 수와 교환하고, 힙 크기를 1개 줄임으로써 힙에 속하지 않는 배열의 뒷부분에는 가장 큰 수부터 차례로 왼쪽 방향으로 저장된다.

기수 정렬

- 기수 정렬 (Radix Sort)이란 비교정렬이 아니고, 숫자를 부분적으로 비교하는 정렬 방법이다.
- 기 (radix)는 특정 진수를 나타내는 숫자들이다.
 - 예를 들어, 10진수의 기는 0, 1, 2, ..., 9이고, 2진수의 기는 0, 1이다.
- 기수 정렬은 제한적인 범위 내에 있는 숫자에 대해서 각 자릿수 별로 정렬하는 알고리즘이다.
- 기수 정렬의 가장 큰 장점은 어느 비교정렬 알고리즘보다 빠르다.

입력

1의 자리

10의 자리

100의 자리

089

07**0**

9**1**0

035

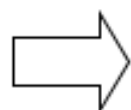
070

91**0**

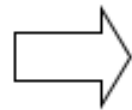
1**3**1

070

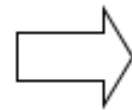
035



13**1**



0**3**5



089

131

03**5**

07**0**

131

910

08**9**

0**8**9

910

- 앞의 예제와 같이 5개의 3자리 십진수가 입력으로 주어지면, 가장 먼저 각 숫자의 1의 자리만 비교하여 작은 수부터 큰 수를 정렬한다.
- 그 다음엔 10의 자리만을 각각 비교하여 정렬한다. 이때 반드시 지켜야 할 순서가 있다.
- 예제에서 10의 자리가 3인 131과 035가 있는데, 10의 자리에 대해 정렬될 때 131이 반드시 035 위에 위치하여야 한다.
- 10의 자리가 같은데 왜 035가 131 위에 위치하면 안 되는 것일까?
 - 그 답은 1의 자리에 대해 정렬해 놓은 것이 아무 소용이 없게 되기 때문이다.

- 입력에 중복된 숫자가 있을 때, 정렬된 후에도 중복된 숫자의 순서가 입력에서의 순서와 동일하면 정렬 알고리즘이 **안정성 (stability)**을 가진다고 한다.
- 아래의 예에서 2개의 10이 입력에 있을 때, **안정한 정렬 (stable sort)** 알고리즘은 중복된 숫자에 대해 입력에서 앞서 있던 숫자가 정렬 후에도 앞서 있고, 불안정한 정렬 알고리즘은 정렬 후에도 그 순서가 반드시 지켜지지 않는다.

정렬 전	90 A	10 B	35 C	13 D	10 E	35 F	31 G	08H
안정한 정렬	08H	10 B	10 E	13 D	31 G	35 C	35 F	90 A
불안정한 정렬	08H	10 E	10 B	13 D	31 G	35 F	35 C	90 A

RadixSort 알고리즘

입력: n 개의 r 진수의 k 자리 숫자

출력: 정렬된 숫자

1. for $i = 1$ to k
2. 각 숫자의 i 자리 숫자에 대해 안정적인 정렬을 수행한다.
3. return 배열 A

- Line 1의 for-루프에서는 1의 자리~ k자리까지 차례로 안정한 정렬을 반복한다.
- Line 2에서는 각 숫자의 i자리 수만에 대해 다음과 같이 정렬한다. 입력 숫자가 r진수라면, i자리가
 - '0'인 수의 개수
 - '1'인 수의 개수
 - ...
 - '(r-1)'인 수의 개수를 각각 계산하여
- i자리가 '0'인 숫자로부터 '(r-1)'인 숫자까지 차례로 안정성에 기반을 두어 정렬한다.

- 예제에서 $i=1$ 일 때, 입력의 각 숫자의 1의 자리 수만을 보면, 9, 0, 5, 1, 0이므로, 1의 자리가 '0'인 숫자가 2개, '1'인 숫자가 1개, '5'인 숫자가 1개, '9'인 숫자가 1개이다.
- 따라서 1의 자리가 '0'인 숫자 070과 910, '1'인 숫자 131, '5'인 숫자 035, 마지막으로 '9'인 숫자 089 순으로 입력의 숫자들이 정렬된다.

