Chapter 4. 그리디 알고리즘

분할정복 - 나누는건 시간 별로.. cost ↓ , Basecase 부분문제해, 전체해

합병 ↑

퀵소트 - 나누는거에 집중

그리디(Greedy) 알고리즘 지금당장 최선을 다하는

- 그리디 알고리즘은 최적화 문제를 해결한다.
  - 최적화(optimization) 문제는 가능한 해들 중에서 가장 좋은(최대 또는 최소) 해를 찾는 문제이다 서울->부산 가장 Best 찾기
  - 그리디 알고리즘은 욕심쟁이 방법, 탐욕적 방법, 탐욕 알고리즘 등으로 불리기도 한다.
- 그리디 알고리즘은 데이터 간의 관계를 고려하지 않고 '욕심내어' 최솟값 또는 최댓값을 가진 데이터를 선택한다.
  - 이러한 선택을 '근시안적'인 선택이라고 한다. 지금 당장 최선.
  - 그리디 알고리즘은 근시안적인 선택으로 부분적인 최적해를 찾고, 이들을 모아서 문제의 최적해를 얻는다.
  - 그리디 알고리즘은 일단 한번 선택하면, 이를 번복하지 않는다. 즉, 선택한 데이터를 버리고 다른 것을 취하지 않는다.

4.1 동전 거스름돈

- 동전 거스름돈(Coin Change) 문제를 해결하는 가장 간단하고 효율적인 방법은 남은 액수를 초과하지 않는 조건하에 '욕심내어' 가장 큰 액면의 동전을 취하는 것이다.

- 다음은 동전 거스름돈 문제의 최소 동전 수를 찾는 그리디 알고리즘이다. 단, 동전의 액면은 500원, 100원, 50원, 10원, 1원이다.

동전 거스름돈 문제 알고리즘

CoinChange(W)	
입력: 거스름돈 액수 W 출력: 거스름돈 액수에 대한 최소 동전 수	
1	change=W, n500=n100=n50=n10=n1=0 // n500, n100, n50, n10, n1은 각각의 동전 수를 저장하는 변수
2	while (change ≥ 500) change = change–500, n500++ // 500원짜리 동전 수를 1 증가
3	while (change ≥ 100) change = change–100, n100++ // 100원짜리 동전 수를 1 증가
4	while (change ≥ 50) change = change–50, n50++ // 50원짜리 동전 수를 1 증가
5	while (change ≥ 10) change = change–10, n10++ // 10원짜리 동전 수를 1 증가
6	while (change ≥ 1) change = change–1, n1++ // 1원짜리 동전 수를 1 증가
7	return (n500+n100+n50+n10+n1) // 총 동전 수를 리턴한다.

동전 거스름돈 문제 알고리즘

Line 1	change를 입력인 거스름돈 액수 W로 놓고, 각 동전 수를 저장하는 변수(동전 카운트)를 n500=n100=n50=n10=n1=0 으로 초기화한다.
Line 2~6	차례로 500원, 100원, 50원, 10원, 1원짜리 동전을 각각의 while-루프를 통해 현재 남은 거스름돈 액수인 change를 넘지 않는 한 계속해서 같은 동전으로 거슬러 주고, 그 때마다 각각의 동전 카운트를 1 증가시킨다.
Line 7	동전 카운트들의 합을 리턴한다.

- CoinChange 알고리즘은 남아있는 거스름돈인 change에 대해 가장 높은 액면의 동전을 거스르며, 500원짜리 동전을 처리하는 line 2에서는 100원짜리, 50원짜리, 10원짜리, 1원짜리 동전을 몇 개씩 거슬러 주어야 할 것인지에 대해서는 전혀 고려하지 않는다. 이것이 바로 그리디 알고리즘의 근시안적인 특성이다.

CoinChange 알고리즘 예제

- 거스름돈 760원에 대해 CoinChange 알고리즘이 수행되는 과정을 살펴보자.
  - Line 1에서는 change=760, n500=n100=n50=n10=n1=0으로 초기화 된다.
  - Line 2에서는 change가 500보다 크므로 while-조건이 '참'이어서 change = change–500 = 760–500 = 260이 되고, n500=1이 된다. 다음은 change가 500보다 작으므로 line 2의 while-루프는 더 이상 수행되지 않는다.
  - Line 3에서는 change가 100보다 크므로 while-조건이 '참'이 되어서 change = change–100 = 260–100 = 160이 되고, n100=1이 된다. 다음도 change가 100보다 크므로 change = change–100 = 160–100 = 60이 되고, n100=2가 된다. 그러나 그 다음엔 change가 60이므로 100보다 작아서 while-루프는 수행되지 않는다.

CoinChange 알고리즘 예제

- Line 4에서는 change가 50보다 크므로 while-조건이 '참'이라서 change = change–50 = 60–50 = 10이 되고, n50=1이 된다. 다음은 change가 50보다 작으므로 while-루프는 수행되지 않는다.
- Line 5에서는 change가 10보다 크므로 while-조건이 '참'이라서 change = change–10 = 10–10 = 0이 되고, n10=1이 된다. 그 다음엔 change가 10보다 작으므로 while-루프는 수행되지 않는다.
- Line 6에서는 change가 0이므로 while-조건이 '거짓'이 되어 while-루프는 수행되지 않는다.
- Line 7에서는 n500+n100+n50+n10+n1 = 1+2+1+1+0 = 5를 리턴한다.

큰동전은 작은 동전에 항상 배수여서 사용가능

캐노니컬 알고리즘

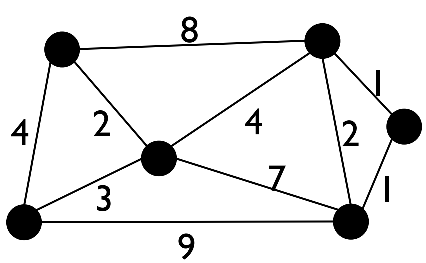
동전 거스름돈 문제

- 만일 한국은행에서 160원짜리 동전을 추가로 발행한다면, CoinChange 알고리즘이 항상 최소 동전 수를 계산할 수 있을까?
  - 거스름돈이 200원이라면, CoinChange 알고리즘은 160원짜리 동전 1개와 10원짜리 동전 4개로서 총 5개를 리턴한다.
  - 그러나 200원에 대한 최소 동전 수는 100원짜리 동전 2개이다.
  - 따라서 CoinChange 알고리즘은 항상 최적의 답을 주지는 못한다.

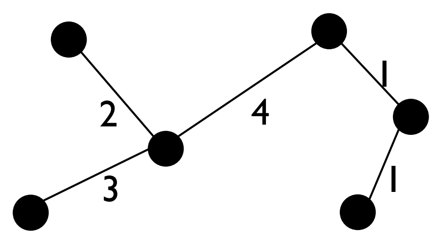
- 5장에서는 어떤 경우에도 최적해를 찾는 동전 거스름돈을 위한 동적 계획 알고리즘을 소개한다.

4.2 최소 신장 트리

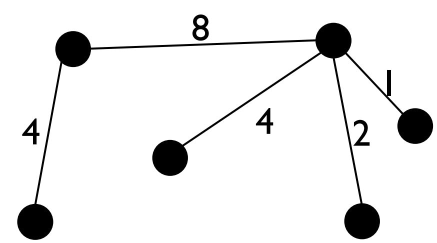
- 최소 신장 트리(Minimum Spanning Tree)란 주어진 가중치 그래프에서 사이클이 없이 모든 점들을 연결시킨 트리들 중 선분들의 가중치 합이 최소인 트리이다.
  - (a)는 주어진 가중치 그래프, (b)는 최소 신장 트리
  - (c),(d)는 최소 신장 트리 아님.
  - (c)는 신장 트리, (d)는 부분그래프



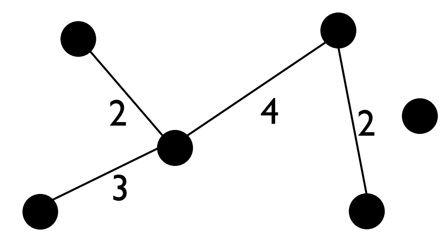
(a) 가중치 그래프



(b) 최소 신장 트리 (가중치의 합 = 11)



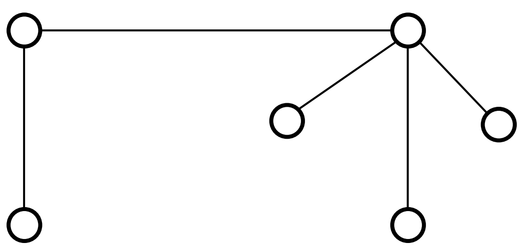
(c) 신장 트리 (가중치의 합 = 19)



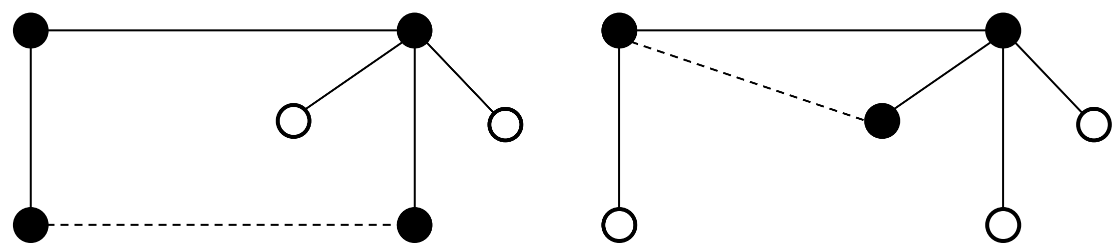
(d) 부분그래프 (한점이 연결안됨)

신장 트리

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



트리



점선으로 된 선분을 추가하여 만들어진 사이클

최소 신장 트리

- 최소 신장 트리를 찾는 대표적인 그리디 알고리즘으로는 크러스컬(Kruskal)과 프림(Prim) 알고리즘이 있다.
  - 알고리즘 입력은 1개의 연결요소(connected component)로 된 가중치 그래프

- 크러스컬 알고리즘은 가중치가 가장 작은 선분이 사이클을 만들지 않을 때에만 '욕심내어' 그 선분을 추가시킨다. 다음은 크러스컬의 최소 신장 트리 알고리즘이다.

음수로 바꾸면 최대 신장트리로 가능

크러스컬 알고리즘 Edge

KruskalMST(G)

입력: 가중치 그래프 G=(V, E), |V|=n (점의 수), |E|=m (선분의 수)
출력: 최소 신장 트리 T

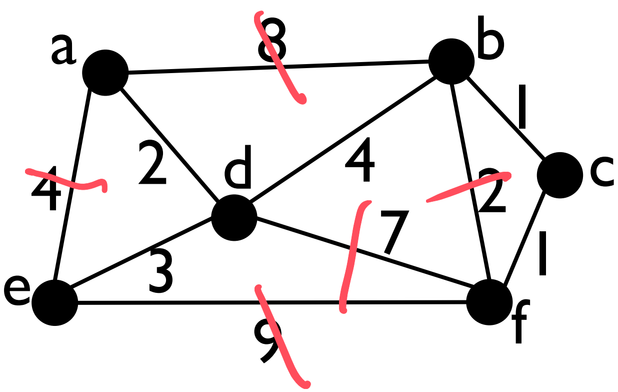
```
가중치의 오름차순으로 선분들을 정렬한다. 정렬된 선분 리스트를 L이라고
하자.                                                         mlogm
T=Ø  공집합   // 트리 T를 초기화시킨다.
while ( T의 선분 수 < n–1 ) {
      L에서 가장 작은 가중치를 가진 선분 e를 가져오고, e를 L에서 제거한다.
      if (선분 e가 T에 추가되어 사이클을 만들지 않으면)
            e를 T에 추가시킨다.
      else          // e가 T에 추가되어 사이클이 만들어지는 경우
            e를 버린다.
}
return 트리 T //T는 최소 신장 트리이다.      사이클이 생기면 안돼
```

크러스컬 알고리즘

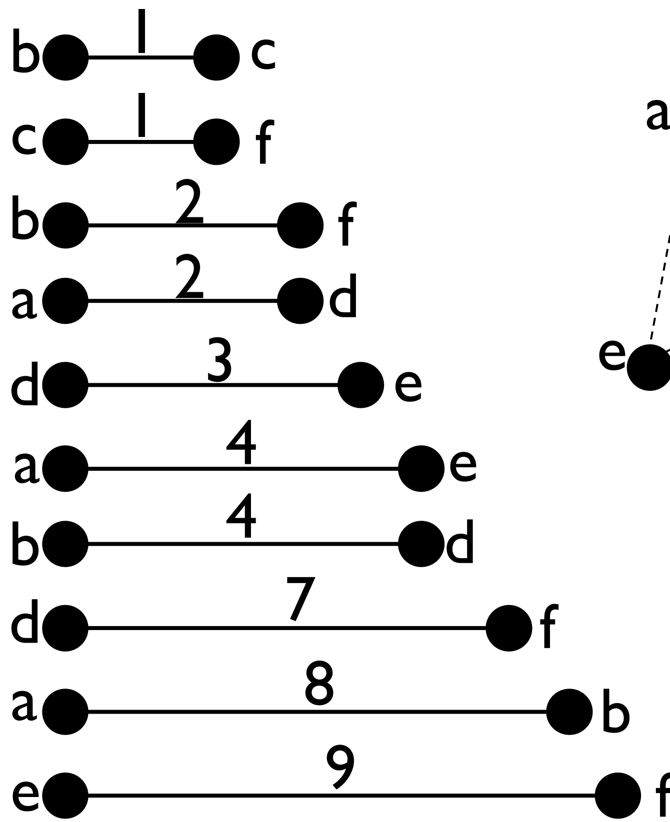
Line 1	모든 선분들을 가중치의 오름차순으로 정렬한다. 정렬된 선분들의 리스트를 L이라고 하자.
Line 2	T를 초기화시킨다. 즉, T에는 아무 선분도 없는 상태에서 시작된다.
Line 3~8	while-루프는 T의 선분 수가 (n–1)이 될 때까지 수행되는데 1번 수행될 때마다 L에서 가중치가 가장 작은 선분 e를 가져온다. 단, 가져온 선분 e는 L에서 삭제되어 다시는 고려되지 않는다.
Line 5~8	가져온 선분 e를 T에 추가하여 사이클을 만들지 않으면 e를 T에 추가시키고, 사이클을 만들면 선분 e를 버린다. 왜냐하면 모든 노드들이 연결되어 있으면서 사이클이 없는 그래프가 신장 트리이기 때문이다.

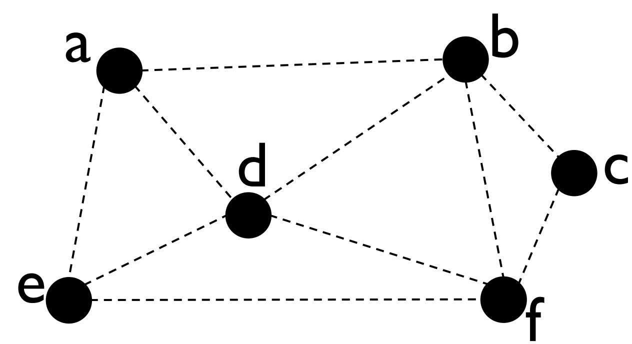
KruskalMST 예제

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



정렬된 리스트 L



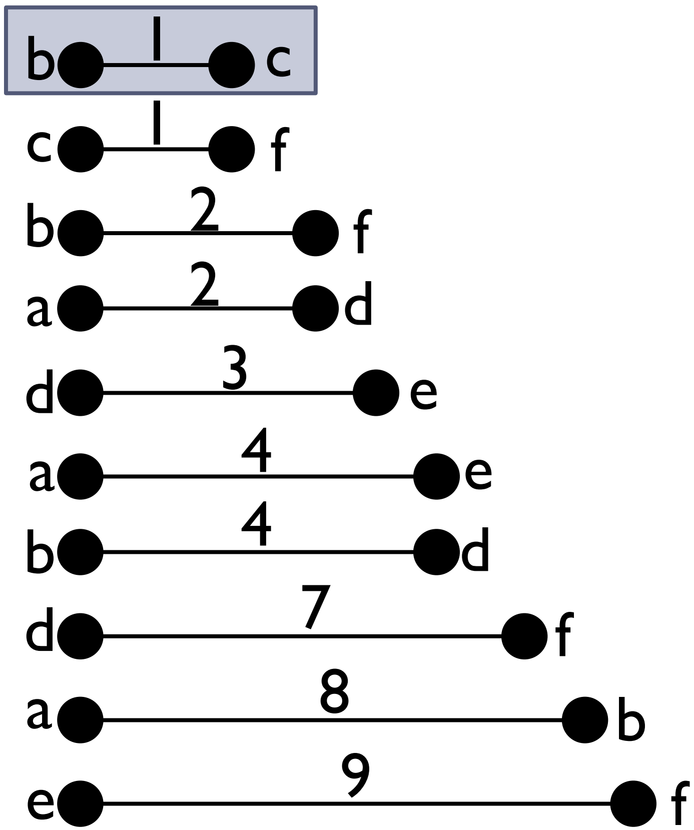


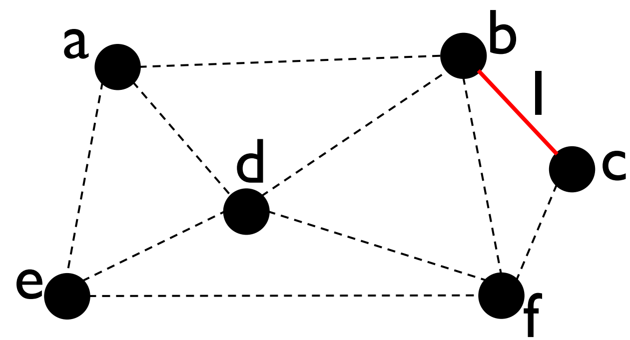
T= ∅

KruskalMST 예제

다 연결되어있는 그래프 상태에서
선분을 하나씩 빼서 트리 찾기
가장 큰거부터 빼기, 트리는 유지해야함

리스트 L

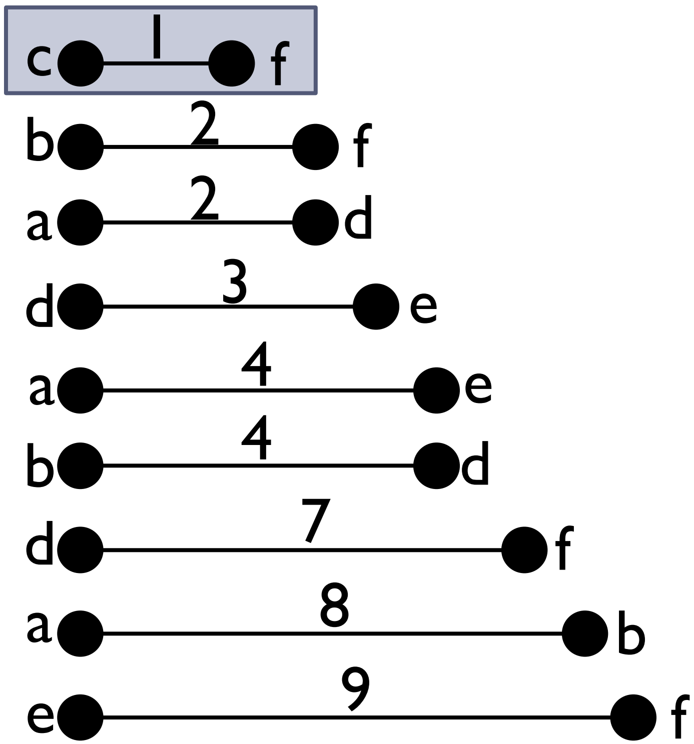


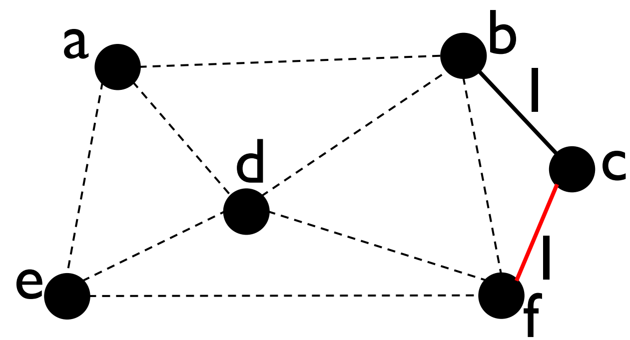


선분 (b, c) 추가

KruskalMST 예제

리스트 L

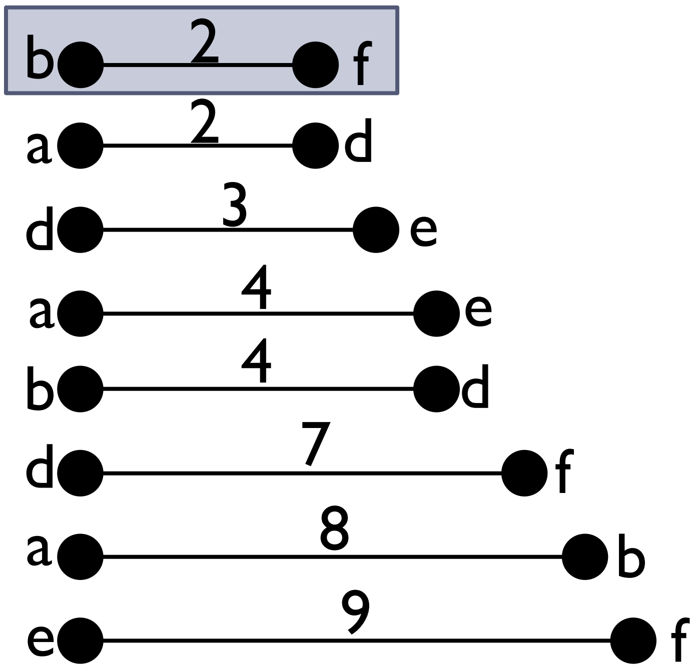


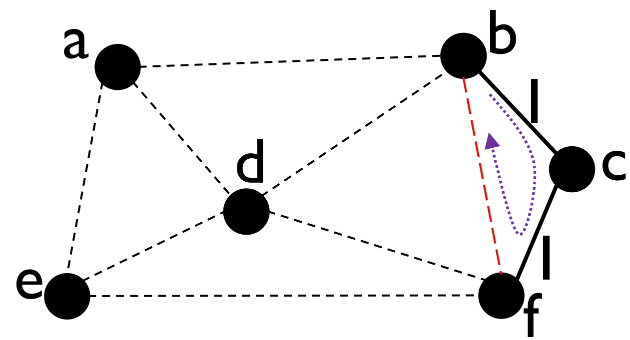


선분 (c, f) 추가

KruskalMST 예제

리스트 L



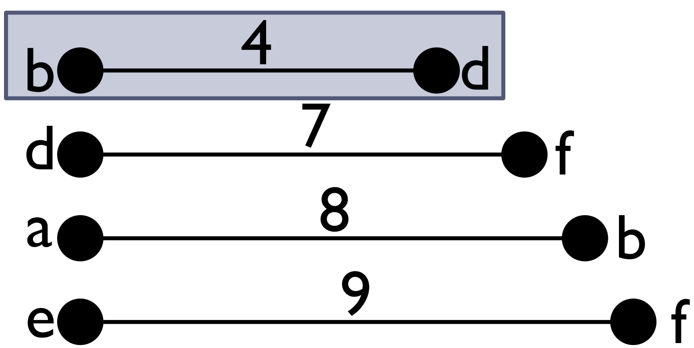


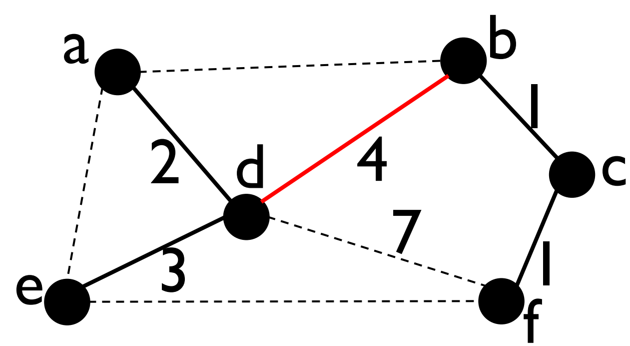
사이클 b-c-f-b
선분 (b, f) 버림

KruskalMST 예제

- 앞과 같은 과정이 반복되면서 선분 (a, d)와 (d, e)가 추가되고 (a, e)가 버려진 후, 선분 (b, d)가 다음과 같이 T에 마지막으로 추가된다.
- T의 선분 수가 n–1 = 6–1 = 5 이므로 알고리즘이 종료된다.

리스트 L





선분 (b, d) 추가

시간복잡도

적을수록 효율↑

선분의 수가 알고리즘 결정

- Line 1에서는 선분들을 가중치로 정렬하는데 O(mlogm) 시간이 걸린다. 단, m은 입력 그래프에 있는 선분의 수이다.
- Line 2에서는 T를 초기화하는 것이므로 O(1) 시간이 걸린다.
- Line 3~8의 while-루프는 최악의 경우 m번 수행된다. 즉, 그래프의 모든 선분이 while-루프 내에서 처리되는 경우이다. 그리고 while-루프 내에서는 L로부터 가져온 선분 e가 사이클을 만드는지를 검사하는데, 이는 O(log* m) 시간이 걸린다. 여기에서 log*m은 logm 보다 느리게 증가하는 함수이다.
- 따라서 크러스컬 알고리즘의 시간복잡도는 O(mlogm) + O(mlog* m) = O(mlogm)이다.

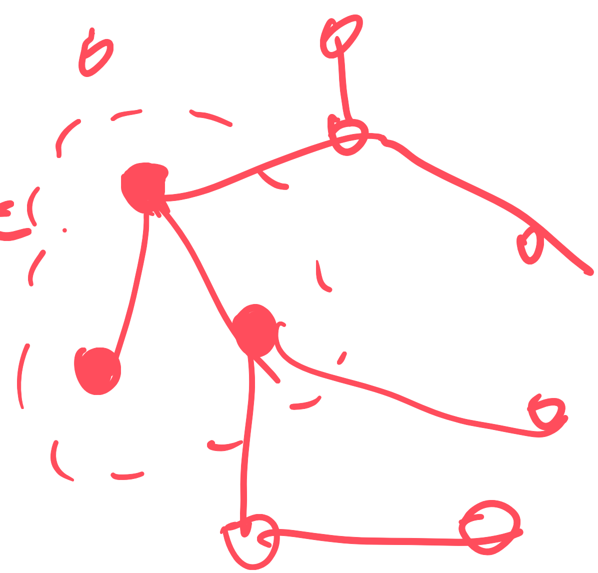
log보다 작은함수

정렬 시간 오래걸림

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.

이미 선택된 점들이
가중치 크기중 유리한걸 고르는
것



프림 알고리즘

```
PrimMST(G)
입력: 가중치 그래프 G=(V, E), |V|=n (점의 수), |E|=m (선분의 수)
출력: 최소 신장 트리 T

그래프 G에서 임의의 점 p를 시작점으로 선택하고, D[p]=0으로 놓는다.
// D[v]는 T에 있는 점과 v를 연결하는 선분의 최소 가중치를 저장한다.
for (점 p가 아닌 각 점 v에 대하여) {    // 배열 D의 초기화
    if ( 선분 (p, v)가 그래프에 있으면 )
        D[v] = 선분 (p, v)의 가중치
    else
        D[v] = ∞
}
T= {p}    // 초기에 트리 T는 점 p만을 가진다.
                    ⋮
```

p와 연결할수 있는것을 두고, 그연결선분의 가중치를 점에준다.

추가하려는 점과 기존의 선택된 점

프림 알고리즘

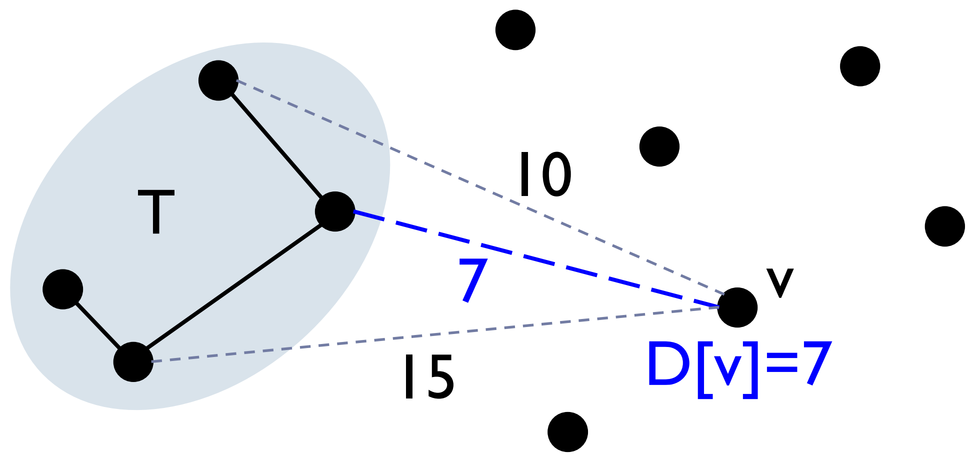
점을 다 선택하면

PrimMST(G)

while (T에 있는 점의 수 < n) {
    T에 속하지 않은 각 점 v에 대해 D[v]가 최소인 점 v_{min}과 연결된 선분 (u, v_{min})을 T에 추가한다. 단, u는 T에 속한 점이고, v_{min}은 T에 추가된다.
    for (T에 속하지 않은 각 점 w에 대해서) {
        if (선분 (v_{min}, w)의 가중치 < D[w])
            D[w] = 선분 (v_{min}, w)의 가중치 // D[w]를 갱신한다.
    }
}
return T //T는 최소 신장 트리이다.

프림 알고리즘

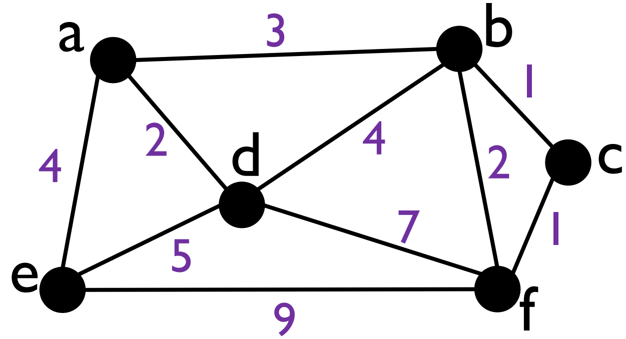
이미 선택되어져있는

Line 1	임의로 점 p를 선택하고, D[p]=0으로 놓는다. 여기서 배열 D[v]에는 점 v와 T에 속한 점들을 연결하는 선분들 중에서 최소 가중치를 가진 선분의 가중치를 저장한다. 다음 그림에서 D[v]에는 10, 7, 15 중에서 최소 가중치인 7이 저장된다. 

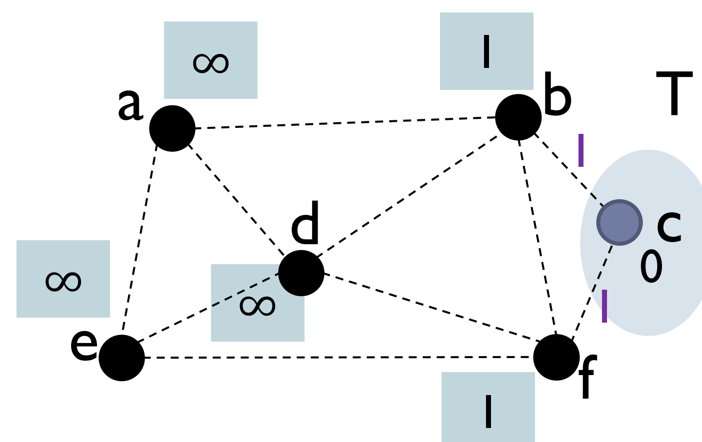
프림 알고리즘

Line 2~6	시작점 p와 선분으로 연결된 점 v의 D[v]를 선분 (p, v)의 가중치로 초기화시키고, 점 p와 선분으로 연결되지 않은 점 v에 대해서 D[v]=∞로 놓는다.
Line 7	T = {p}로 초기화시킨다.
Line 8~12	while-루프는 T의 점의 수가 n이 될 때까지 수행된다. T에 속한 점의 수가 n이 되면, T는 신장 트리이다.
Line 9	T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 v_{min}을 찾는다. 그리고 점 v_{min}과 연결된 선분 (u, v_{min})을 T에 추가한다. 단, u는 T에 속한 점이고, 선분 (u, v_{min})이 T에 추가된다는 것은 점 v_{min}도 T에 추가되는 것이다. 기준
Line 10~12	for-루프에서는 line 9에서 새로 추가된 점 v_{min}에 연결되어 있으면서 T에 속하지 않은 각 점 w의 D[w]에 대해서, 선분 (v_{min}, w)의 가중치가 D[w]보다 작으면 D[w]를 선분 (v_{min}, w)의 가중치로 갱신한다.
Line 13	최소 신장 트리 T를 리턴한다.

PrimMST 예제

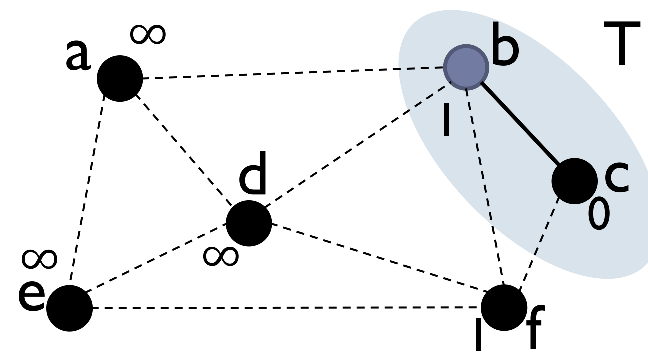


- Line 1에서 임의의 시작점으로 점 c가 선택되었다고 가정하자. 그리고 D[c]=0으로 초기화시킨다.
- Line 2~6에서는 시작점 c와 선분으로 연결된 각 점 v에 대해서, D[v]를 각 선분의 가중치로 초기화시키고, 나머지 각 점 v에 대해서, D[v]는 ∞로 초기화시킨다.



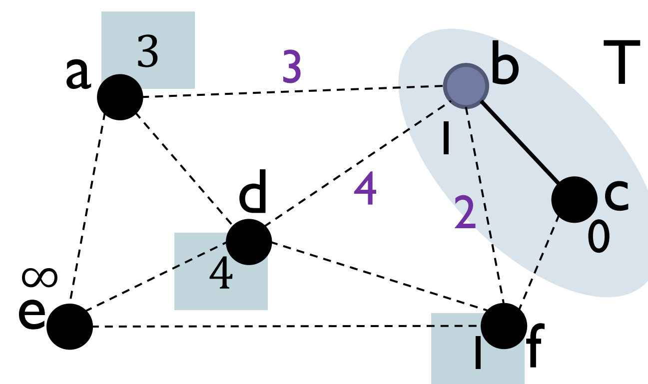
PrimMST 예제

- Line 7에서는 T={c}로 초기화한다.
- Line 8의 while-루프의 조건이 '참'이다. 즉, 현재 T에는 점 c만이 있다. 따라서 line 9에서 T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 v_{min}을 선택한다. D[b]=D[f]=1로서 최솟값이므로 점 b를 임의로 선택하자. 따라서 점 b와 선분 (b, c)가 T 에 추가된다.



$1 = \min\{\infty, 1, \infty, \infty, 1\}$

- Line 10~12에서 점 b에 연결된 점 a와 d의 D[a]와 D[d]를 각각 3과 4로 갱신한다. 점 f는 점 b와 선분으로 연결되어 있으나, 선분 (b, f)의 가중치인 2가 현재 D[f]=1 보다 크므로 D[f]는 갱신되지 않는다.

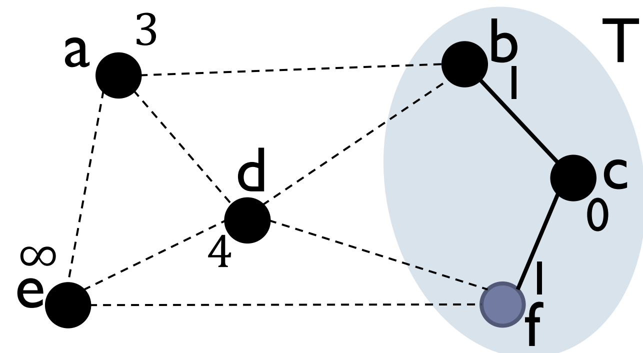


b 입장으로 갱신 필요

원래의 가중치가 작으므로 유지

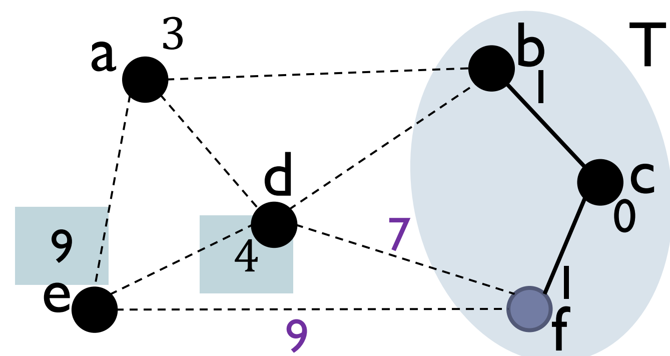
PrimMST 예제

- 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.



$1 = \min\{3, 4, \infty, 1\}$

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

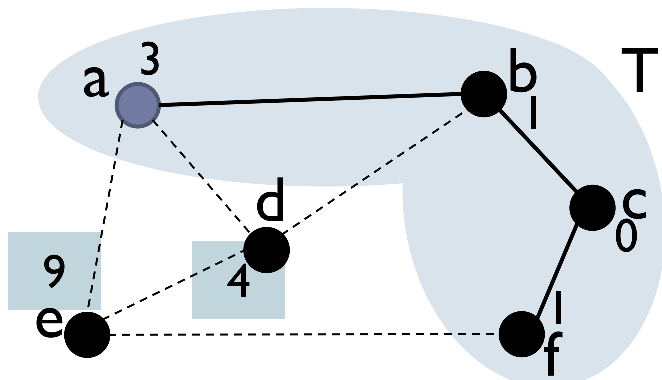


$e = \infty \to 9$

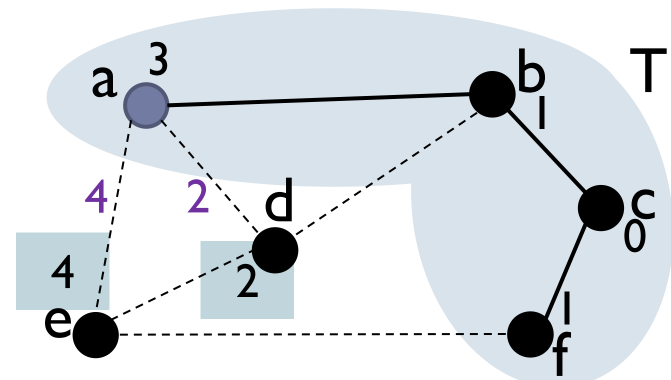
d입장에서 아까 b와 4였으니
7이아닌 4로 유지

PrimMST 예제

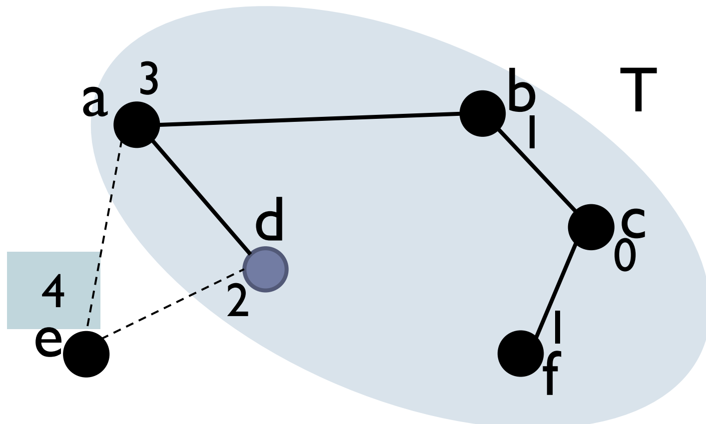
- 그 다음부터는 점 a와 선분 (a, b), 점 d와 선분 (a, d)가 차례로 T에 추가되고, 최종적으로 점 e와 선분 (a, e)가 추가되면서, 최소 신장 트리 T가 완성된다.
- Line 13에서는 T를 리턴하고, 알고리즘을 마친다. 다음의 그림들이 위의 과정을 차례로 보여준다.



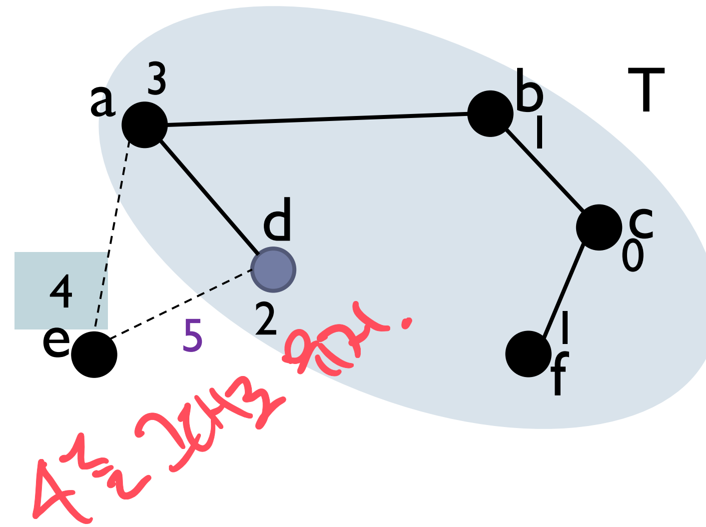
3= min{3, 4, 9}

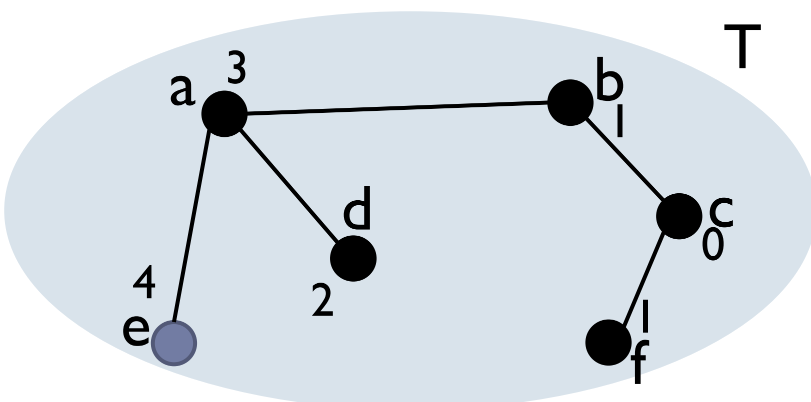


PrimMST 예제

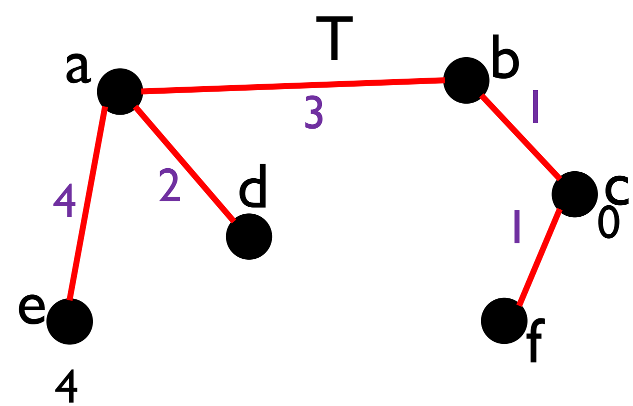


2= min{4, 2}



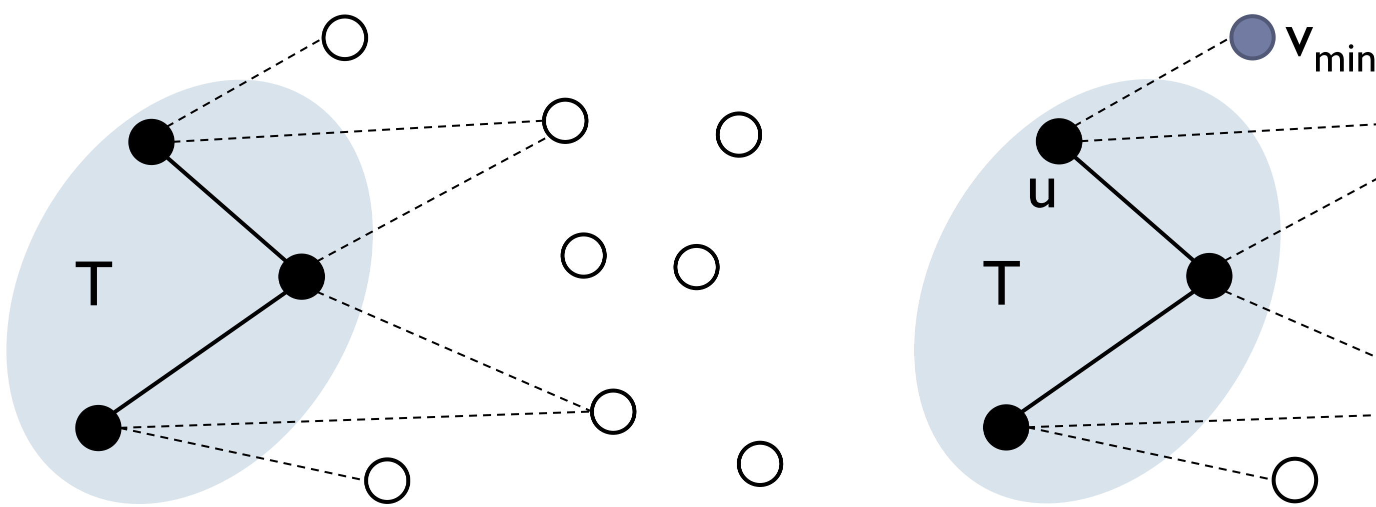


4= min{4}



프림 알고리즘 수행과정

- PrimMST 알고리즘이 최종적으로 리턴하는 T에는 왜 사이클이 없을까?
  - 프림 알고리즘은 T 외부에 있는 점을 항상 추가하므로 사이클이 안 만들어진다.
  - 선분 (u, v_{min})이 최소 가중치를 가지고 있어서 T에 추가되면, 점 v_{min}은 T 외부의 점이므로 사이클이 만들어질 수 없다.



시간복잡도

- while-루프가 (n–1)번 반복되고, 1회 반복될 때 line 9에서 T에 속하지 않은 각 점 v에 대하여, D[v]가 최소인 점 v_{min}을 찾는데 O(n) 시간이 걸린다.
- 왜냐하면 1차원 배열 D에서 (현재 T에 속하지 않은 점들에 대해서) 최솟값을 찾는 것이고, 배열의 크기는 그래프의 점의 수인 n이기 때문이다.
- 프림 알고리즘의 시간복잡도는 (n–1)×O(n) = $O(n^2)$이다.

Edge 영향 X

Edge 개수에 따라 다름

크러스컬 알고리즘과 프림 알고리즘

- 수행과정
  - 크러스컬 알고리즘에서는 선분이 1개씩 T에 추가되는데, 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 만들어진다.

선분

  - 프림 알고리즘에서는 T가 점 1개인 트리에서 시작되어 선분을 1개씩 추가시킨다. 즉, 1개의 트리가 자라나서 신장 트리가 된다.

정점

- 응용
  - 최소 비용으로 선로 또는 파이프 네트워크(인터넷 광 케이블 선로, 케이블 TV선로, 전화선로, 송유관로, 가스관로, 배수로 등)를 설치하는데 활용

4.3 최단 경로 찾기

출발지가 정해져있다.

- 최단 경로(Shortest Path) 문제는 주어진 가중치 그래프에서 어느 한 출발점에서 또 다른 도착점까지의 최단 경로를 찾는 문제이다.
- 최단 경로를 찾는 가장 대표적인 알고리즘은 다익스트라 (Dijkstra) 최단 경로 알고리즘이며, 이 또한 그리디 알고리즘이다.

다익스트라 알고리즘

- 프림의 최소 신장 트리 알고리즘과 거의 흡사한 과정으로 진행된다. 2가지 차이점은 다음과 같다.
  - 프림 알고리즘은 임의의 점에서 시작하나, 다익스트라 알고리즘은 주어진 출발점에서 시작한다.
  - 프림 알고리즘은 트리에 하나의 점(선분)을 추가시킬 때 현재 상태의 트리에서 가장 가까운 점을 추가시킨다. 그러나 다익스트라의 알고리즘은 출발점으로부터 최단 거리가 확정되지 않은 점들 중에서 출발점으로부터 가장 가까운 점을 추가하고, 그 점의 최단 거리를 확정한다.

다익스트라 알고리즘

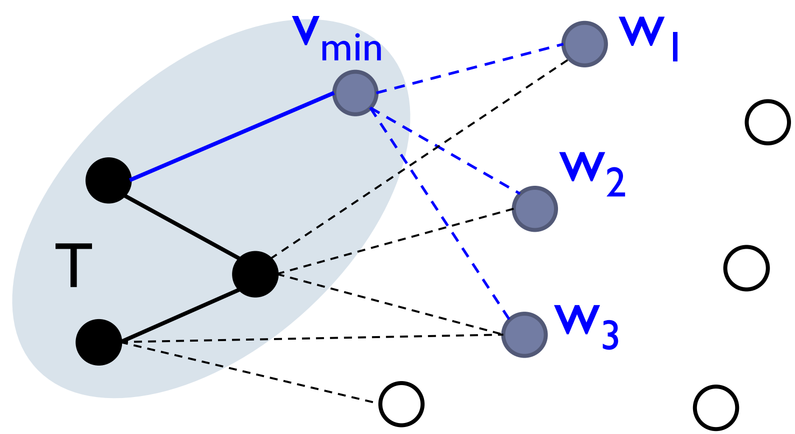
도착지점도 정했었는데 바뀜

ShortestPath(G, s)	
입력: 가중치 그래프 G=(V, E), |V|=n (점의 수), |E|=m (선분의 수), 출발점 s 출력: 출발점 s로부터 (n–1)개의 점까지 각각 최단 거리를 저장한 배열 D	
1	배열 D를 ∞로 초기화시킨다. 단, D[s]=0으로 초기화한다. // 배열 D[v]에는 출발점 s로부터 점 v까지의 거리가 저장된다.
2	while (s로부터의 최단 거리가 확정되지 않은 점이 있으면) {
3	현재까지 s로부터 최단 거리가 확정되지 않은 각 점 v에 대해서 최소의 D[v]의 값을 가진 점 v_{min}을 선택하고, 출발점 s로부터 점 v_{min}까지의 최단 거리 D[v_{min}]을 확정시킨다.
4	s로부터 현재보다 짧은 거리로 점 v_{min}을 통해 우회 가능한 각 점 w에 대해서 D[w]를 갱신한다. }
5	return D

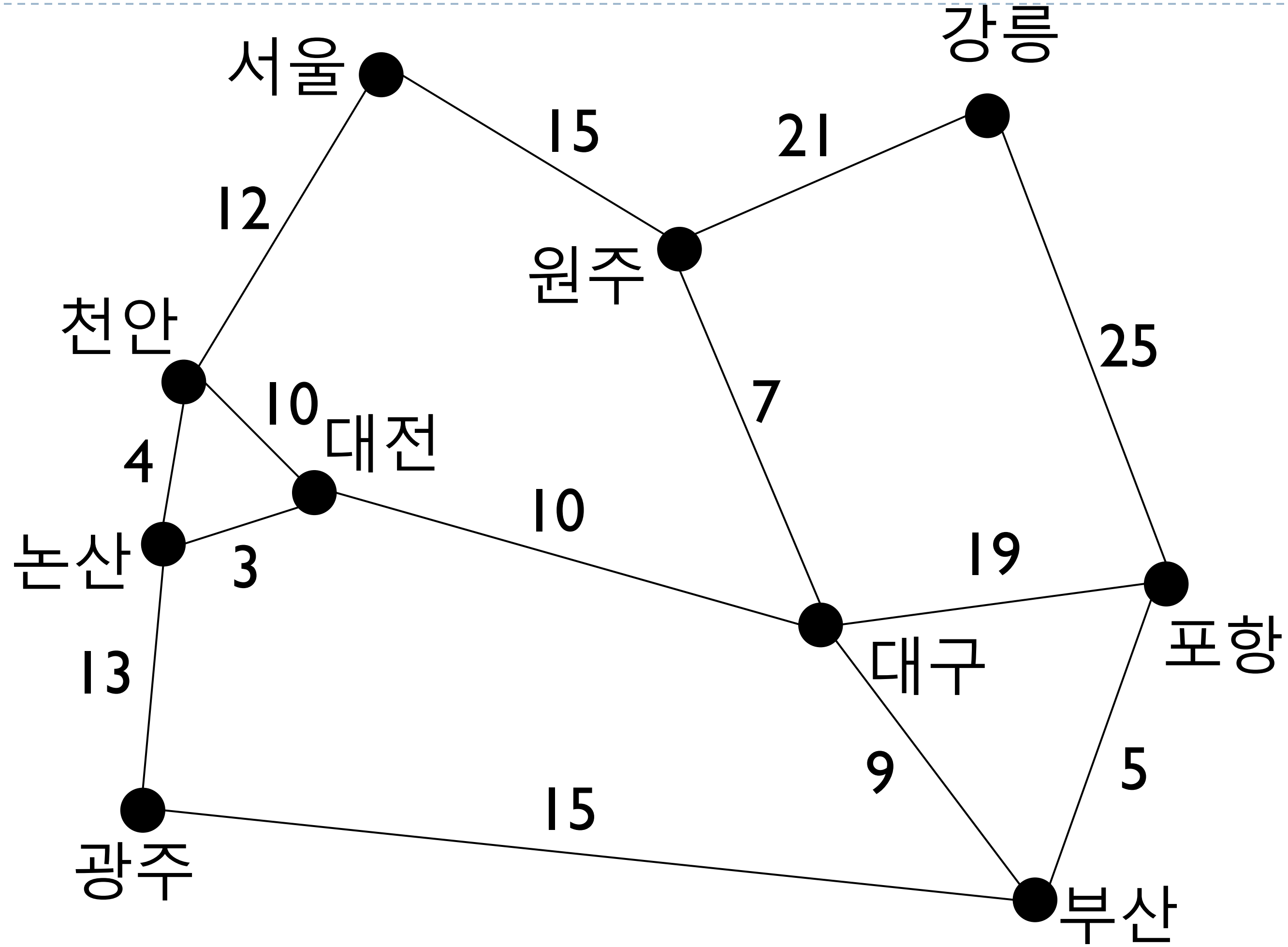
다익스트라 알고리즘

D[v]	알고리즘에서 배열 D[v]는 출발점 s로부터 점 v까지의 거리를 저장하는데 사용하고, 최종적으로는 출발점 s로부터 점 v까지의 최단 거리를 저장하게 된다.
Line 1	출발점 s의 D[s]=0으로, 또 다른 각 점 v에 대해서 D[v]=∞로 초기화시킨다.
Line 2~4	while-루프는 (n–1)회 수행된다. 현재까지 s로부터 최단 거리가 확정된 점들의 집합을 T라고 놓으면, V–T는 현재까지 s로부터 최단 거리가 확정되지 않은 점들의 집합이다. 따라서 V–T에 속한 각 점 v에 대해서 D[v]가 최소인 점 v_{min}을 선택하고, v_{min}의 최단 거리를 확정시킨다. 즉, $D[v_{min}] \le D[v]$, $v \in V–T$이다. '확정한다'는 것은 2가지의 의미를 갖는다. I. $D[v_{min}]$이 확정된 후에는 다시 변하지 않는다. II. 점 v_{min}이 T에 포함된다.

다익스트라 알고리즘

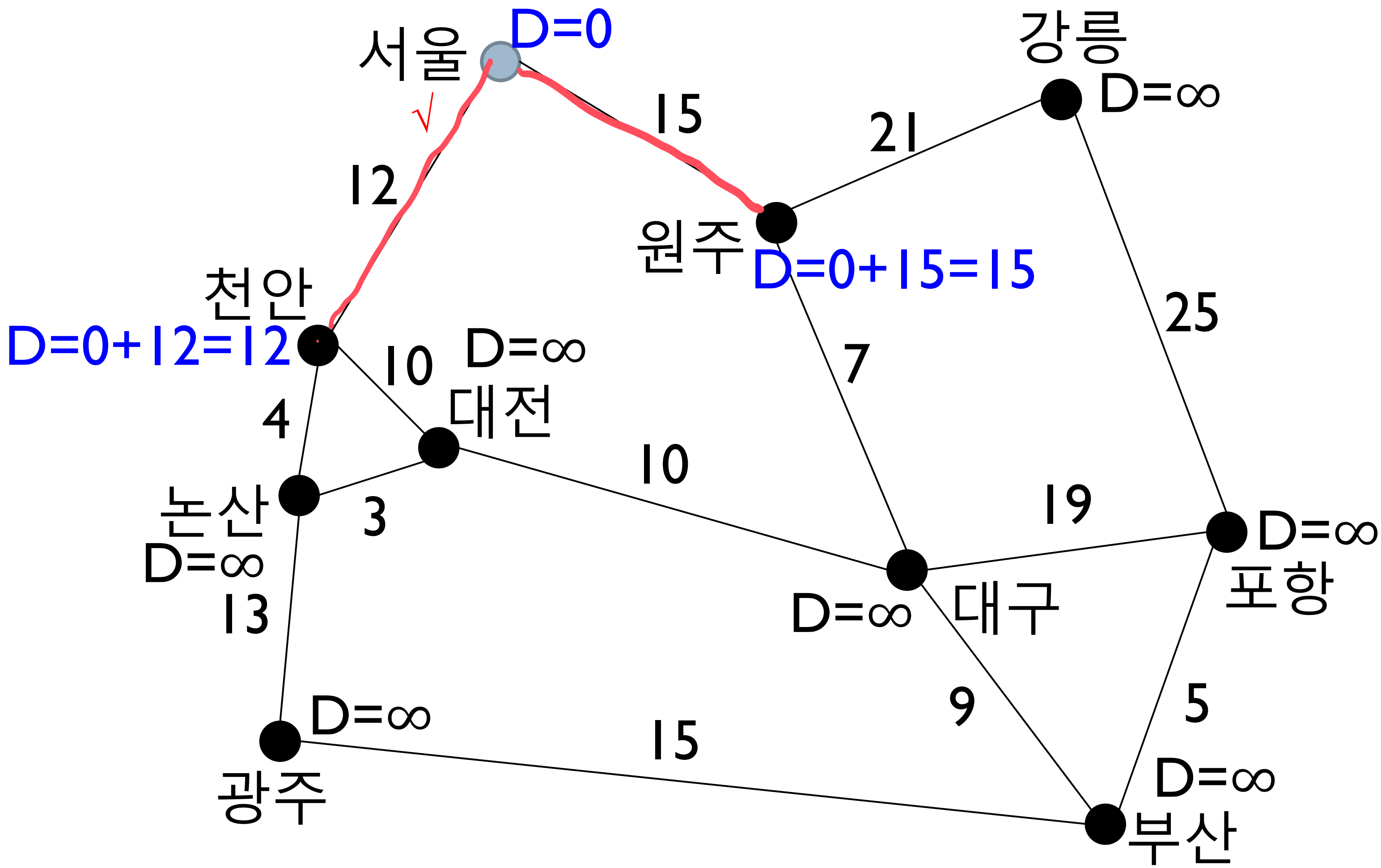
Line 4	V–T에 속한 점들 중 v_{min}을 거쳐 감(경유함)으로써 s로부터의 거리가 현재보다 더 짧아지는 점 w가 있으면, 그 점의 D[w]를 갱신한다. 다음 그림은 v_{min}이 T에 포함된 상태를 보이고 있는데, v_{min}에 인접한 점 w_1, w_2, w_3 각각에 대해서 만일 (D[v_{min}] + 선분 (v_{min}, w_i)의 가중치) < D[w_i]이면, D[w_i] = (D[v_{min}] + 선분 (v_{min}, w_i)의 가중치)로 갱신한다. 
Line 5	배열 D를 리턴한다.

ShortestPath 예제



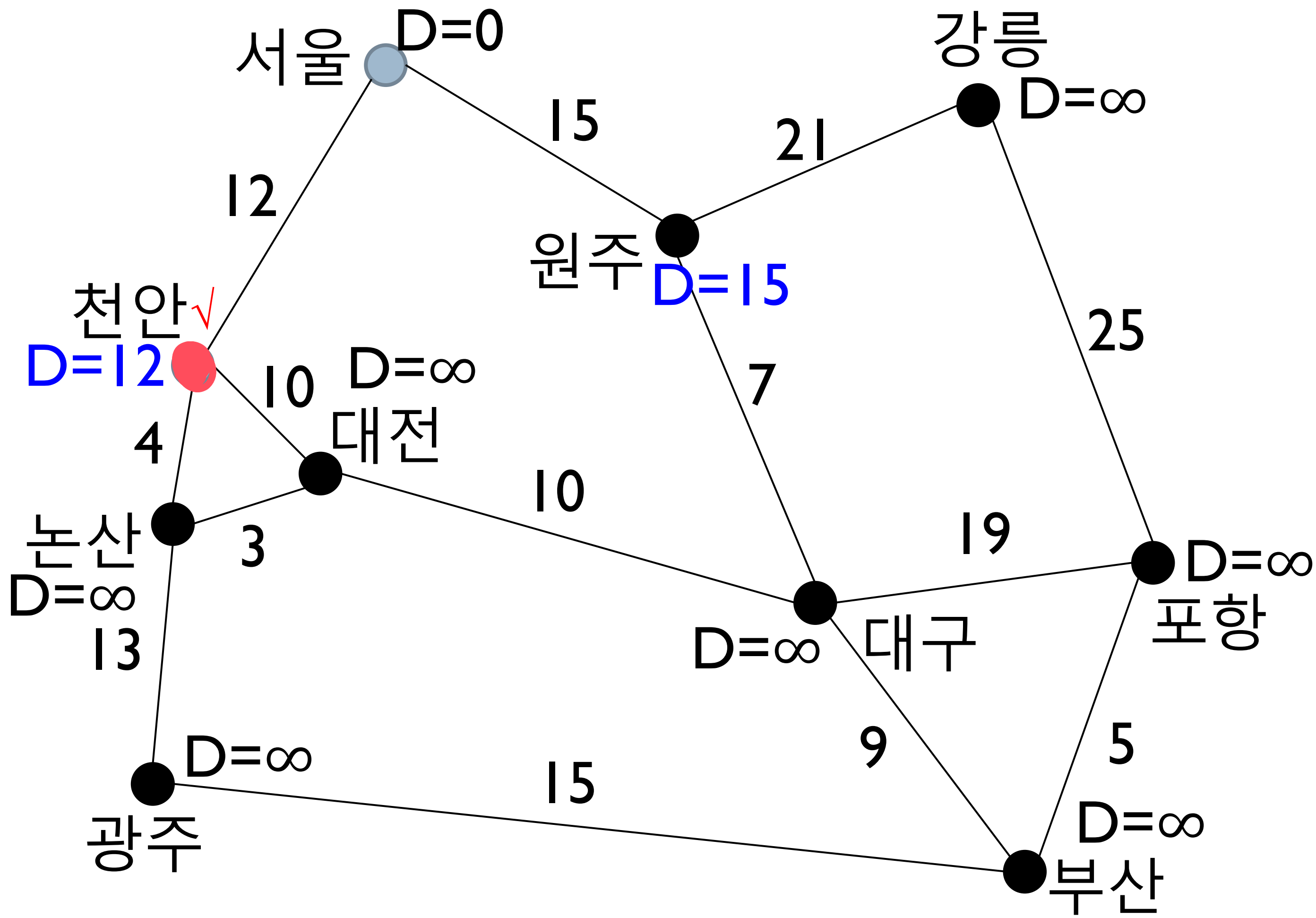
입력 그래프

ShortestPath 예제



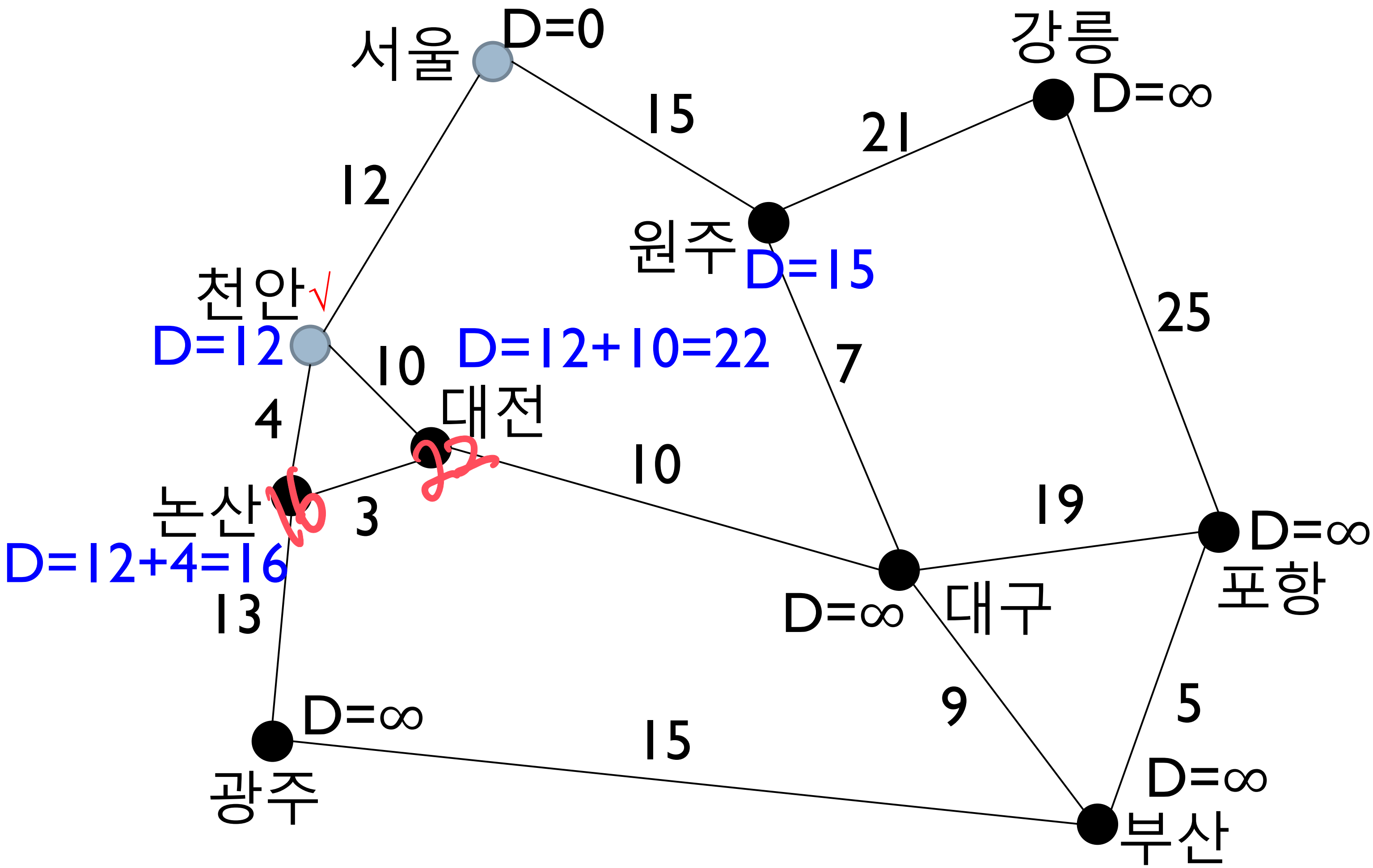
출발점=서울, D[천안], D[원주]를 갱신

ShortestPath 예제



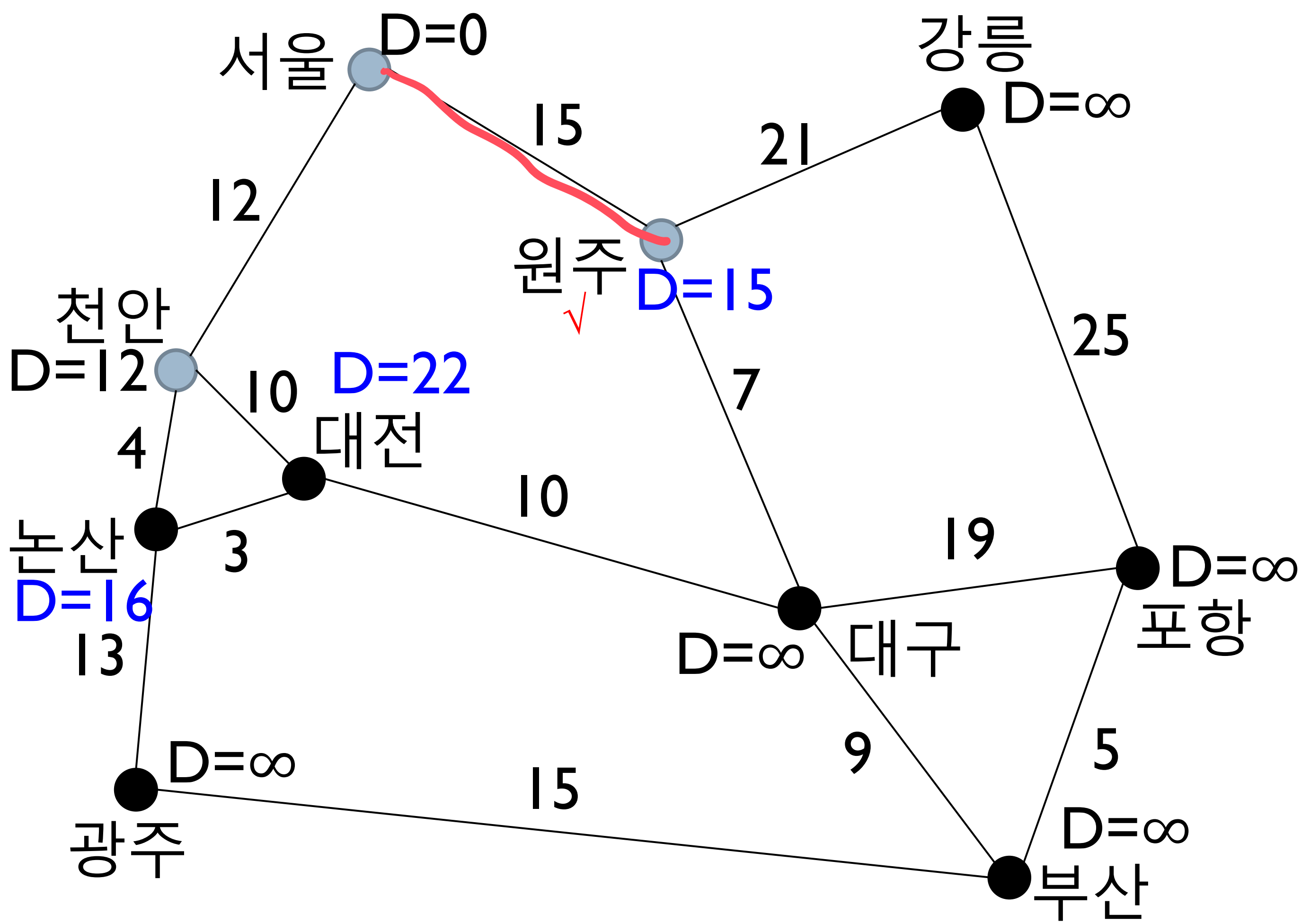
D[천안]이 최소

ShortestPath 예제



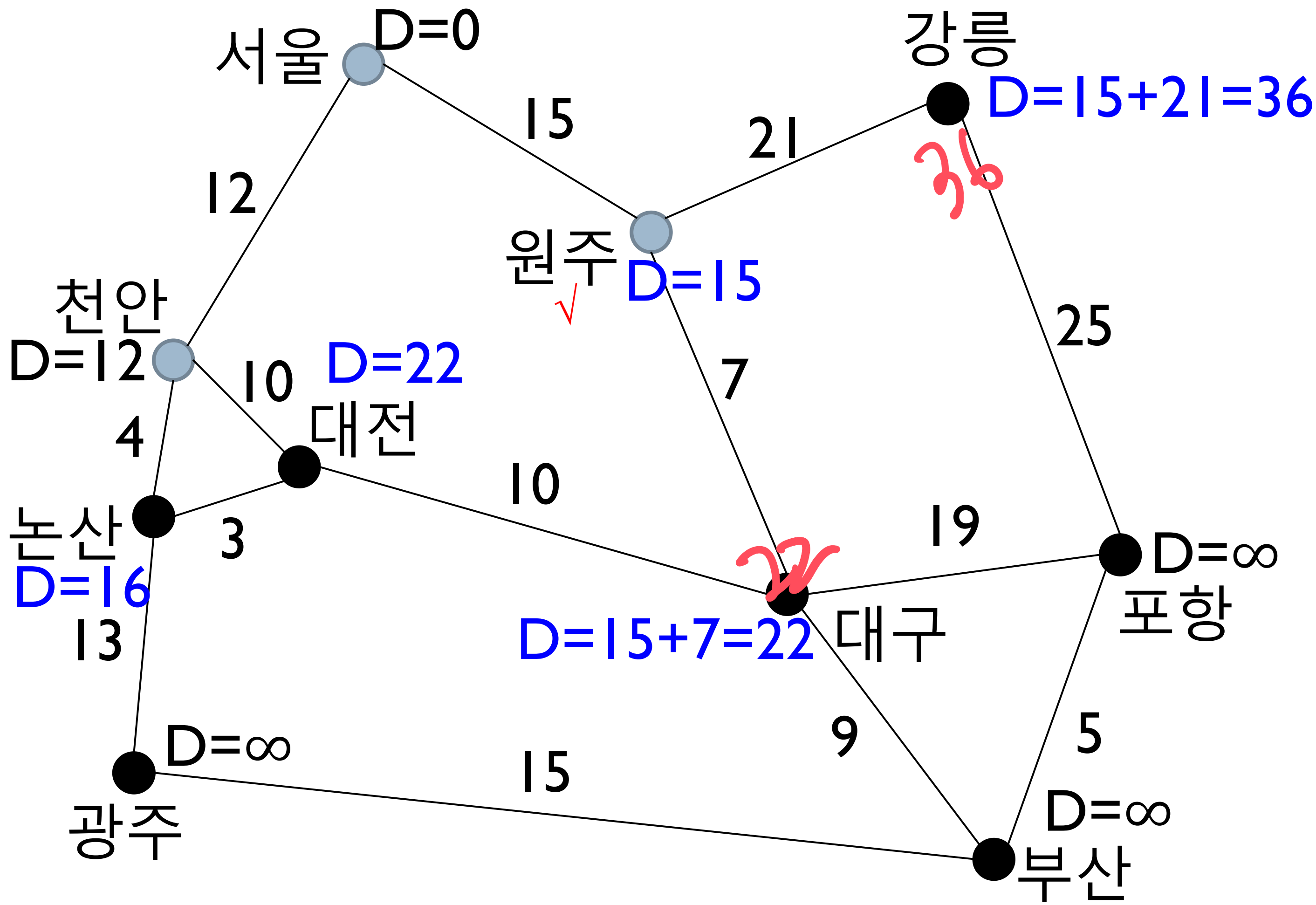
D[논산], D[대전]을 갱신

ShortestPath 예제



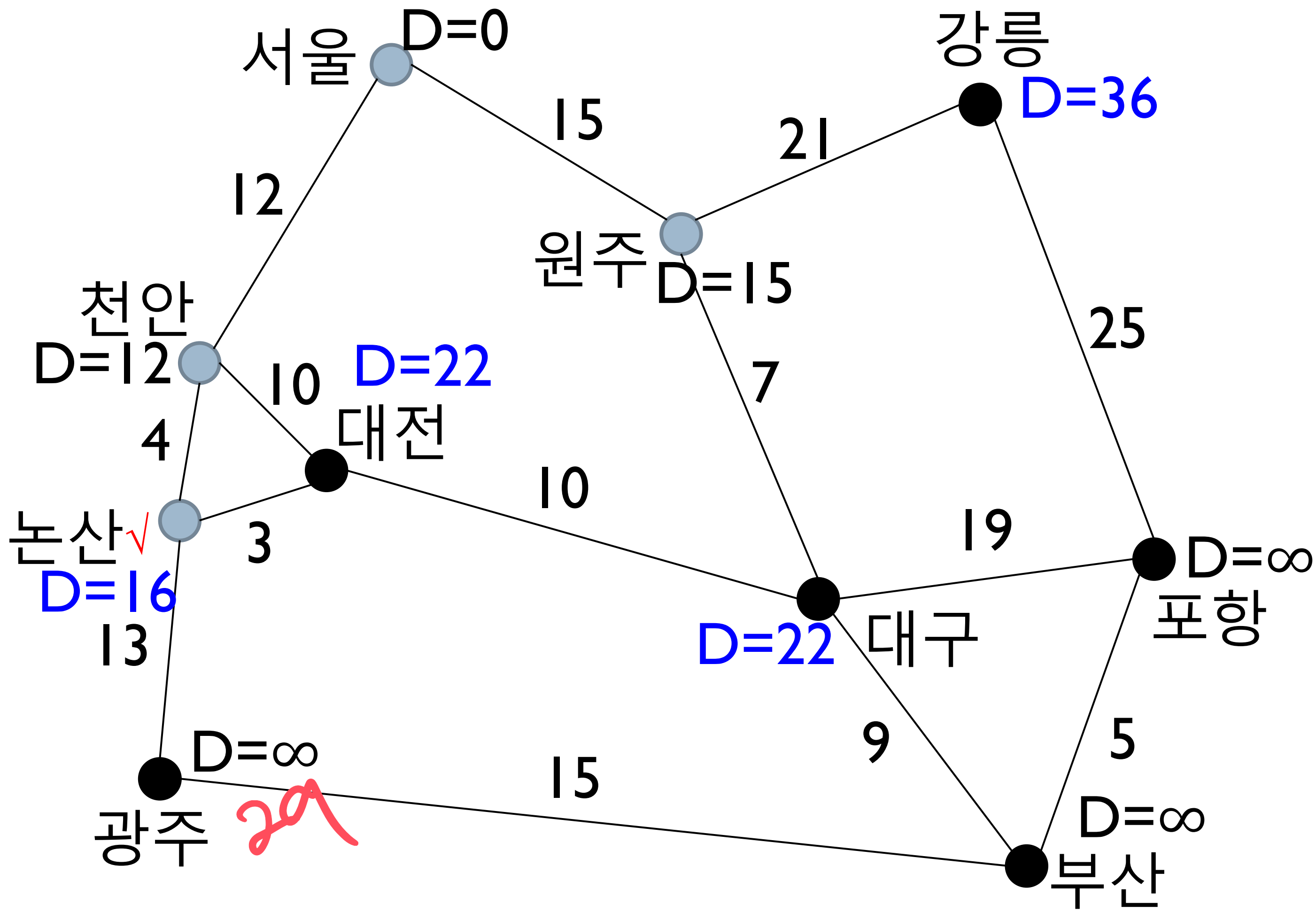
D[원주]가 최소

ShortestPath 예제



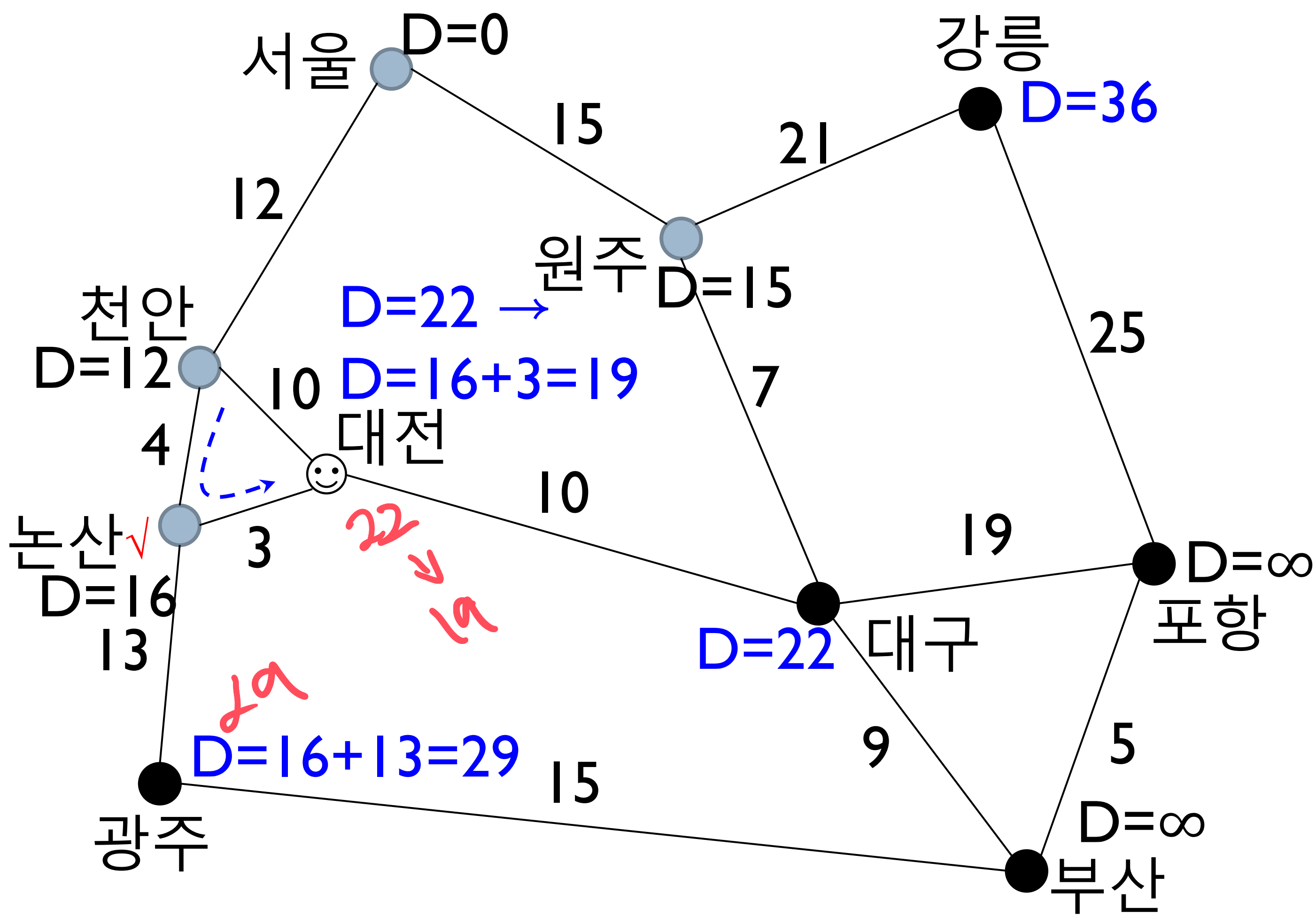
D[강릉], D[대구]를 갱신

ShortestPath 예제



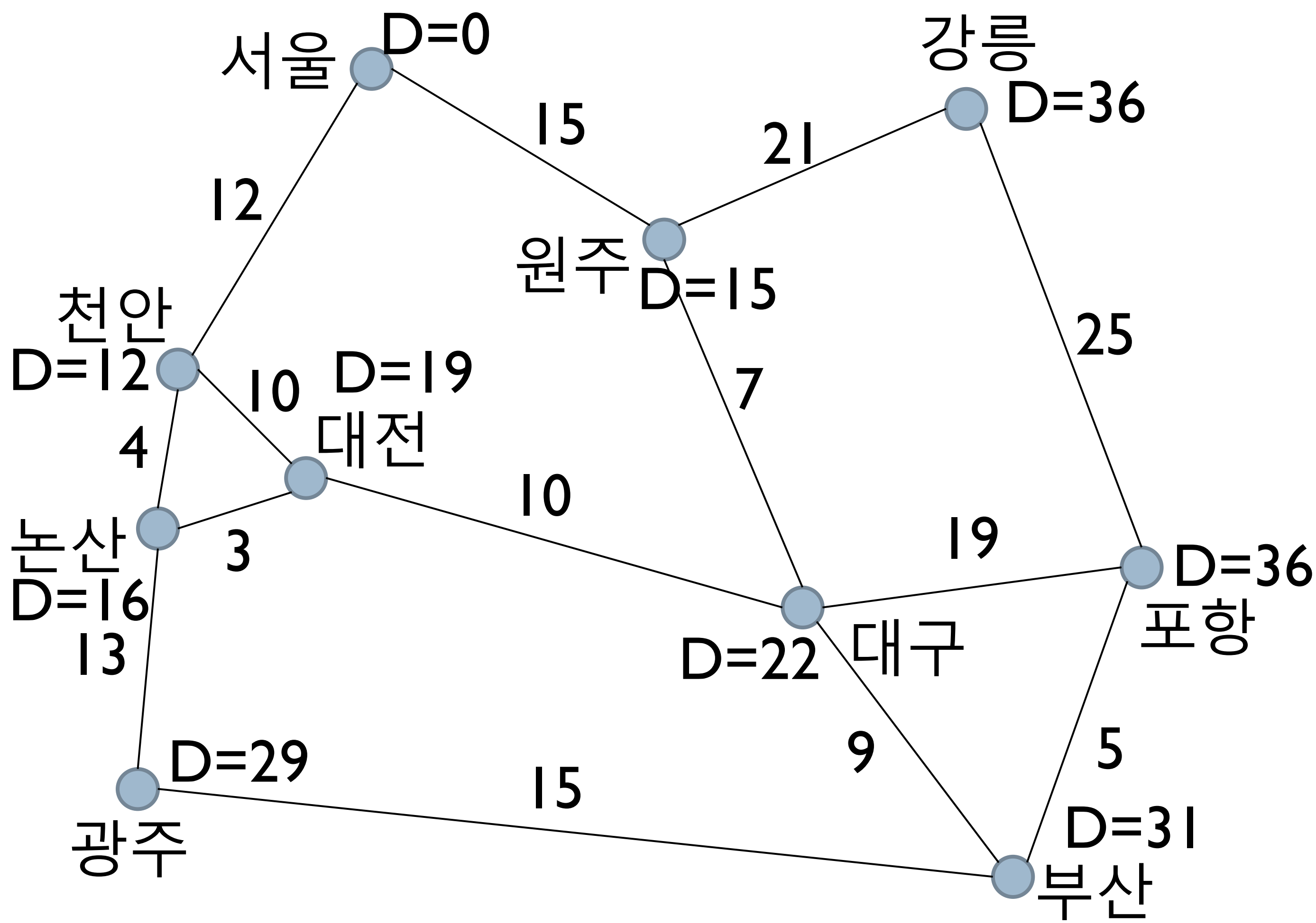
D[논산]이 최소

ShortestPath 예제



D[광주], D[대전]을 갱신
(다음은 대전이 최소, 이후 과정 생략)

ShortestPath 예제



최종 결과

시간복잡도

- while-루프가 (n–1)번 반복된다.
  - 1회 반복될 때 line 3에서 최소의 D[v]를 가진 점 v_{min}을 찾는데 O(n) 시간이 걸린다. 왜냐하면 배열 D에서 최솟값을 찾는 것이기 때문이다.
  - line 4에서도 v_{min}에 연결된 점의 수가 최대 (n–1)개이므로, 각 D[w]를 갱신하는데 걸리는 시간은 O(n)이다.

- 따라서, 시간복잡도는 (n–1) × {O(n) + O(n)} = $O(n^2)$이다.

Prim과 같다.

응용

- 구글(Google) 등 웹사이트의 지도 서비스
- 자동차 네비게이션
- 네트워크와 통신 분야
- 모바일 네트워크
- 산업 공학과 경영 공학의 운영 연구(Operation Research)
- 로봇 공학
- 교통 공학
- VLSI 디자인 분야 등

4.4 부분 배낭 문제

- 배낭(Knapsack) 문제는 n개의 물건이 있고, 각 물건은 무게와 가치를 가지고 있으며, 배낭이 한정된 무게의 물건들을 담을 수 있을 때, 최대의 가치를 갖도록 배낭에 넣을 물건들을 정하는 문제이다.
- 원래 배낭 문제는 물건을 통째로 배낭에 넣어야 되지만, 부분 배낭(Fractional Knapsack) 문제는 물건을 부분적으로 담는 것이 허용된다. 가루, 액체
- 부분 배낭 문제에서는 물건을 부분적으로 배낭에 담을 수 있으므로, 최적해를 위해서 '욕심을 내어' 단위 무게당 가장 값나가는 물건을 배낭에 넣고, 계속해서 그 다음으로 값나가는 물건을 넣는다.
- 만일 그 다음으로 값나가는 물건을 '통째로' 배낭에 넣을 수 없으면, 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 넣는다.

부분배낭 문제 알고리즘

FractionalKnapsack	w: 배낭물건 무게합
입력: n개의 물건, 각 물건의 무게와 가치, 배낭의 용량 C 출력: 배낭에 담은 물건 리스트 L과 배낭에 담은 물건의 가치 합 v	
1	각 물건에 대해 단위 무게 당 가치를 계산한다.
2	물건들을 단위 무게 당 가치를 기준으로 내림차순으로 정렬하고, 정렬된 물건 리스트를 S라고 하자. 비싼거부터
3	L=∅, w=0, v=0 // L은 배낭에 담을 물건 리스트, w는 배낭에 담긴 물건들의 무게의 합, v는 배낭에 담긴 물건들의 가치의 합이다.
4	S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져온다. ⋮

부분배낭 문제 알고리즘

FractionalKnapsack

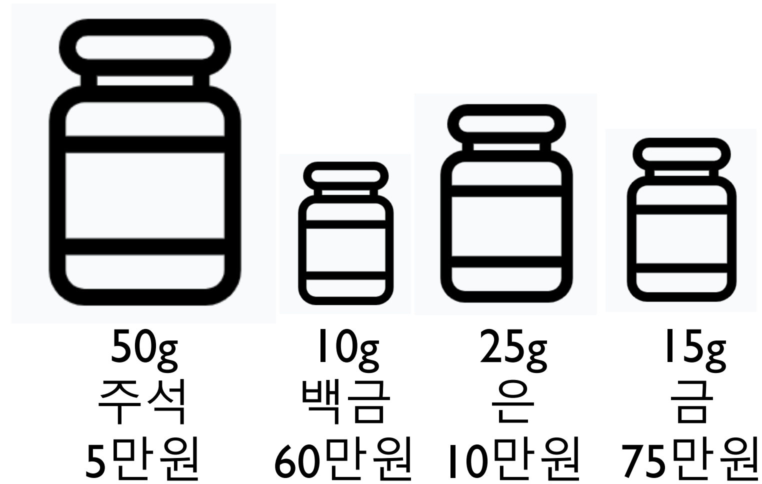
```
while ( w + (x의 무게) ≤ C ) {
        x를 L에 추가시킨다.
        w = w + (x의 무게)
        v = v + (x의 가치)
        x를 S에서 제거한다.  — 정렬 후보리스트
        S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져온다.
}   남은무게
if ( (C–w) > 0 ) { // 배낭에 물건을 부분적으로 담을 여유가 있으면
        물건 x를 (C–w)만큼만 L에 추가한다.
        v = v + (C–w)만큼의 x의 가치
}
return L, v
```

부분배낭 문제 알고리즘

Line 1~2	각 물건의 단위 무게 당 가치를 계산하여, 이를 기준으로 물건들을 내림차순으로 정렬한다.
Line 5~10	while-루프를 통해서 다음으로 단위 무게당 값나가는 물건을 가져다 배낭에 담고, 만일 가져온 물건을 배낭에 담을 경우 배낭의 용량이 초과되면 (즉, while-루프의 조건이 '거짓'이 되면) 가져온 물건을 '통째로' 담을 수 없게 되어 루프를 종료한다. 부분적으로 추가
Line 11	현재까지 배낭에 담은 물건들의 무게 w가 배낭의 용량 C 보다 작으면 (즉, if-조건이 '참'이면) line 12~13에서 해당 물건을 (C–w)만큼만 배낭에 담고, (C–w)만큼의 x의 가치만큼 v를 증가시킨다.
Line 14	최종적으로 배낭에 담긴 물건들의 리스트 L과 배낭에 담긴 물건들의 가치의 합 v를 리턴한다.

FractionalKnapsack 예제

- 4개의 금속 분말이 오른쪽 그림과 같이 있다. 배낭 용량이 40그램일 때, 알고리즘 수행 과정을 살펴보자.



- Line 1~2의 결과
  - S=[백금, 금, 은, 주석]

물건	단위 그램당 가치
백금	6만원
금	5만원
은	4천원
주석	1천원

FractionalKnapsack 예제

- Line 3: L=∅, w=0, v=0로 각각 초기화한다.
- Line 4: S = [백금, 금, 은, 주석]로부터 백금을 가져온다.
- Line 5: while-루프의 조건 (w + 백금의 무게 ≤ C) = (0 + 10 < 40)이 '참'이다. 통째로 가능
- Line 6: 백금을 배낭 L에 추가시킨다. 즉, L = [백금]이 된다.
- Line 7: w = w + (백금의 무게) = 0 + 10g = 10g
- Line 8: v = v + (백금의 가치) = 0 + 60만원 = 60만원
- Line 9: S에서 백금을 제거한다. S = [금, 은, 주석]
- Line 10: S에서 금을 가져온다.
- Line 5: while-루프의 조건 (w + 금의 무게 ≤ C) = (10 + 15 < 40)이 '참'이다. 10 15 40
- Line 6: 금을 배낭 L에 추가시킨다. L = [백금, 금]

FractionalKnapsack 예제

- Line 7: w = w + (금의 무게) = 10g + 15g = 25g
- Line 8: v = v + (금의 가치) = 60만원 + 75만원 = 135만 원
- Line 9: S에서 금을 제거한다. S = [은, 주석]
- Line 10: S에서 은을 가져온다. 25g
- Line 5: while-루프의 조건 (w + 은의 무게 ≤ C) = (25 + 25 < 40)이 '거짓'이므로 루프를 종료한다. 25 25 통째로 불가능
- Line 11: if-조건 ((C – w) > 0)이 '참'이다. 즉, 40 – 25 = 15 > 0 이기 때문이다. 40-25 = 15, 부분시간
- Line 12: 은을 C – w = 40 – 25 = 15g만큼만 배낭 L에 추가시킨다.
- Line 13: v = v + (15g × 4천원/g) = 135만원 + 6만원 = 141만원
- Line 14: 배낭 L=[백금 10g, 금 15g, 은 15g]과 가치의 합 v = 141만원을 리턴한다. ||

시간복잡도

물건의 개수: 입력 n

- Line 1에서 n개의 물건 각각의 단위 무게당 가치를 계산하는 데는 O(n) 시간이 걸리고, line 2에서 물건의 단위 무게당 가치에 대해서 내림차순으로 정렬하기 위해 O(nlogn) 시간이 걸린다.

  머지소트

- Line 5~10의 while-루프 수행은 n번을 넘지 않으며, 루프 내부의 수행은 O(1) 시간이 걸린다. 또한 line 11~14도 각각 O(1) 시간 걸린다.

  이미 정렬되어 있기때문에 그냥 갖고 오는 거임

- 따라서 알고리즘의 시간복잡도는 O(n)+O(nlogn)+n×O(1)+O(1) = O(nlogn)이다.

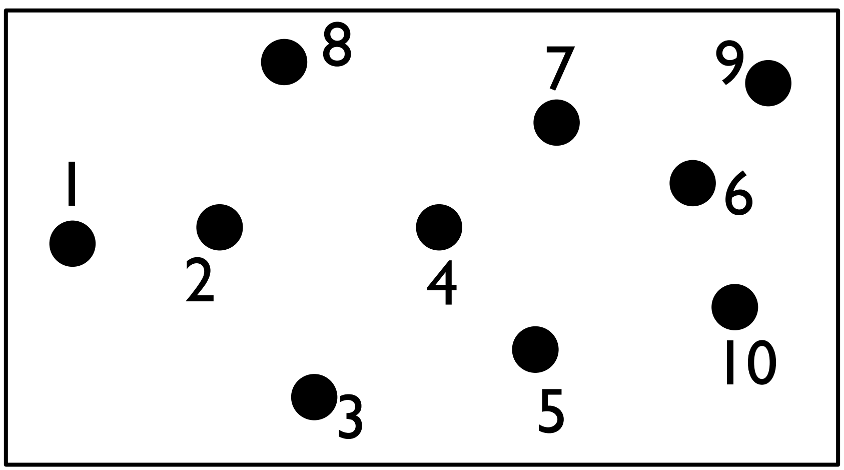
정렬이 최고 믕

4.5 집합 커버 문제

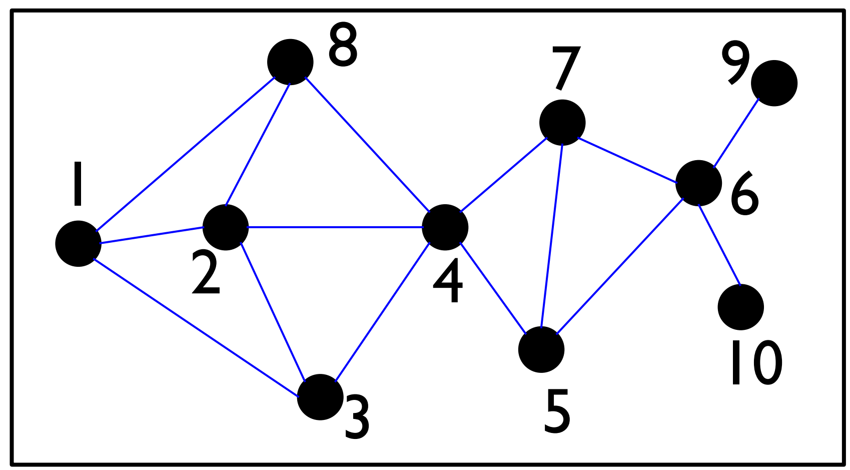
- n개의 원소를 가진 집합인 U가 있고, U의 부분 집합들을 원소로 하는 집합 F가 주어질 때, F의 원소들인 집합들 중에서 어떤 집합들을 선택하여 합집합하면 U와 같게 되는가?
- 집합 커버(Set Cover) 문제는 F에서 선택하는 집합들의 수를 최소화하는 문제이다.

집합 커버 문제

- (예) 신도시 계획 학교 배치
- 10개의 마을이 신도시에 있고, 아래의 2가지 조건이 만족되도록 학교의 위치를 선정하여야 한다고 가정하자.
  - 학교는 마을에 위치해야 한다.
  - 등교 거리는 걸어서 15분 이내이어야 한다.



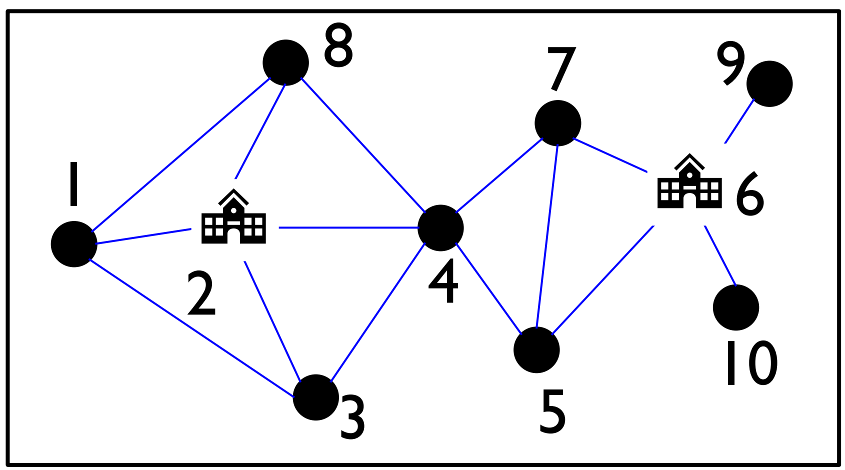
10개의 마을의 위치



등교 거리 15분 이내 마을 간의 관계

집합 커버 문제

- 어느 마을에 학교를 신설해야 학교의 수가 최소로 되는가?
  - 2번 마을에 학교를 만들면 마을 1, 2, 3, 4, 8의 학생들이 15분 이내에 등교할 수 있다 (즉, 마을 1, 2, 3, 4, 8 이 '커버'된다).
  - 6번 마을에 학교를 만들면 마을 5, 6, 7, 9, 10이 커버된다.
  - 즉, 2번과 6번 마을에 학교를 배치하면 모든 마을이 커버된다.



집합 커버 문제

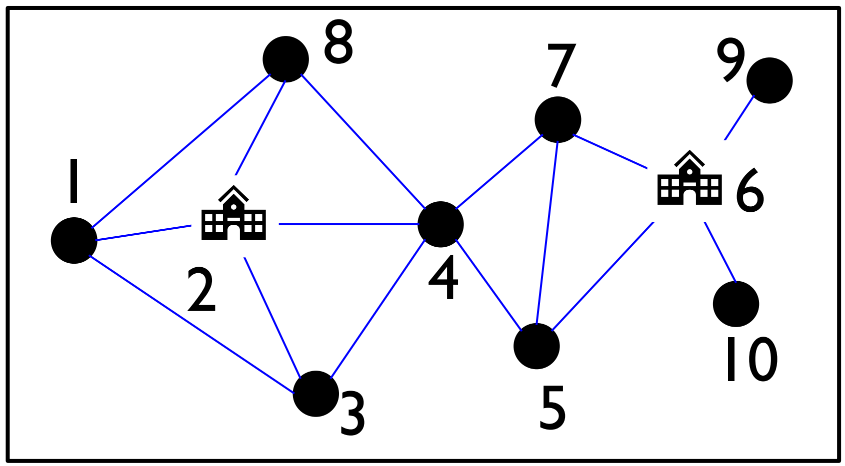
- 신도시 계획 문제를 집합 커버 문제로 변환시키면 다음과 같다. 여기서 S_i는 마을 i에 학교를 배치했을 때 커버되는 마을의 집합이다.
- U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} // 신도시의 마을 10개
- $F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

$S_1=\{1, 2, 3, 8\}$	$S_5=\{4, 5, 6, 7\}$	$S_9=\{6, 9\}$
$S_2=\{1, 2, 3, 4, 8\}$	$S_6=\{5, 6, 7, 9, 10\}$	$S_{10}=\{6, 10\}$
$S_3=\{1, 2, 3, 4\}$	$S_7=\{4, 5, 6, 7\}$	
$S_4=\{2, 3, 4, 5, 7, 8\}$	$S_8=\{1, 2, 4, 8\}$	

- S_i 집합들 중에서 어떤 집합들을 선택하여야 그들의 합집합이 U와 같은가? 단, 선택된 집합의 수는 최소이어야 한다.

집합 커버 문제

- 이 문제의 답은 $S_2 \cup S_6$ = {1, 2, 3, 4, 8}∪{5, 6, 7, 9, 10} = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = U이다.



집합 커버 문제

- 집합 커버 문제의 최적해는 어떻게 찾아야 할까?
  - F에 n개의 집합들이 있다고 가정해보자.
  - 가장 단순한 방법은 F에 있는 집합들의 모든 조합을 1개씩 합집합하여 U가 되는지 확인하고, U가 되는 조합의 집합 수가 최소인 것을 찾는 것이다.
  - 예를 들면, $F = \{S_1, S_2, S_3\}$일 경우, 모든 조합이란, S_1, S_2, S_3, $S_1 \cup S_2$, $S_1 \cup S_3$, $S_2 \cup S_3$, $S_1 \cup S_2 \cup S_3$ 이며 총 $3+3+1 = 7 = 2^3-1$개이다.
  - F에 n개의 원소가 있으면 (2^n-1)개를 다 검사하여야 하고, n이 커지면 최적해를 찾는 것은 실질적으로 불가능하다.

- 따라서, 최적해를 찾는 대신에 최적해에 근접한 근사해 (approximation solution)를 찾는다.

집합 커버 문제 알고리즘

SetCover

입력: U, F={S_i}, i=1,…,n
출력: 집합 커버 C

C = ∅ 커버가 다 끝나면
while (U ≠ ∅) do { F의 원소를 갖고온다
    U의 원소들을 가장 많이 포함하고 있는 집합 S_i를 F에서 선택한다.
    U = U – S_i 커버되지않은 집합.
    S_i를 F에서 제거하고, S_i를 C에 추가한다.
}
return C

Si가 들어간 집합

집합 커버 문제 알고리즘

Line 1	C를 공집합으로 초기화시킨다.
Line 2~5	while-루프에서는 집합 U가 공집합이 될 때까지 수행된다.
Line 3	'그리디'하게 U와 가장 많은 수의 원소들을 공유하는 집합 S_i 를 선택한다.
Line 4	S_i의 원소들을 U에서 제거한다. 왜냐하면 S_i의 원소들은 커버된 것이기 때문이다. 따라서 U는 아직 커버되지 않은 원소들의 집합이다.
Line 5	S_i를 F로부터 제거하여, S_i가 line 3에서 더 이상 고려되지 않도록 하며, S_i를 집합 커버 C에 추가한다.
Line 6	C를 리턴한다.

집합 커버 문제 알고리즘 수행과정

- 도시 계획 문제에 대해서 SetCover 알고리즘이 수행되는 과정을 살펴보자

$U=\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

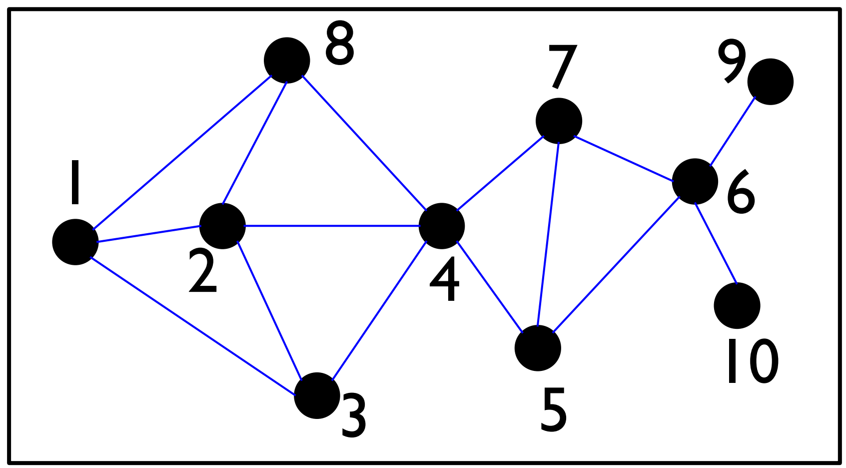
$F=\{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

$S_1=\{1, 2, 3, 8\}$ $S_5=\{4, 5, 6, 7\}$ $S_9=\{6, 9\}$

$S_2=\{1, 2, 3, 4, 8\}$ $S_6=\{5, 6, 7, 9, 10\}$ $S_{10}=\{6, 10\}$

$S_3=\{1, 2, 3, 4\}$ $S_7=\{4, 5, 6, 7\}$

$S_4=\{2, 3, 4, 5, 7, 8\}$ $S_8=\{1, 2, 4, 8\}$



SetCover 예제

- Line 1: C = ∅로 초기화한다.
- Line 2: while-조건 (U ≠ ∅) = ({1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ≠ ∅)이 '참'이다.
- Line 3: U의 원소를 가장 많이 커버하는 집합인 S_4 = {2, 3, 4, 5, 7, 8}을 F에서 선택한다.
- Line 4: U = U – S_4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} – {2, 3, 4, 5, 7, 8} = {1, 6, 9, 10}
- Line 5: S_4를 F에서 제거하고, 즉, F = {S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}} – {S_4} = {S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}}가 되고, S_4를 C에 추가한다. 즉, C = {S_4}이다.

SetCover 예제

- Line 2: while-조건 ($U \neq \emptyset$) = ($\{1, 6, 9, 10\} \neq \emptyset$)이 '참'이다.
- Line 3: U의 원소를 가장 많이 커버하는 집합인 $S_6 = \{5, 6, 7, 9, 10\}$을 F에서 선택한다.
- Line 4: $U = U - S_6 = \{1, \cancel{6}, \cancel{9}, \cancel{10}\} - \{5, 6, 7, 9, 10\} = \{1\}$
- Line 5: S_6을 F에서 제거하고, 즉, $F = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_6\} = \{S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$이 되고, S_6을 C에 추가한다. 즉, $C = \{S_4, S_6\}$이다.

SetCover 예제

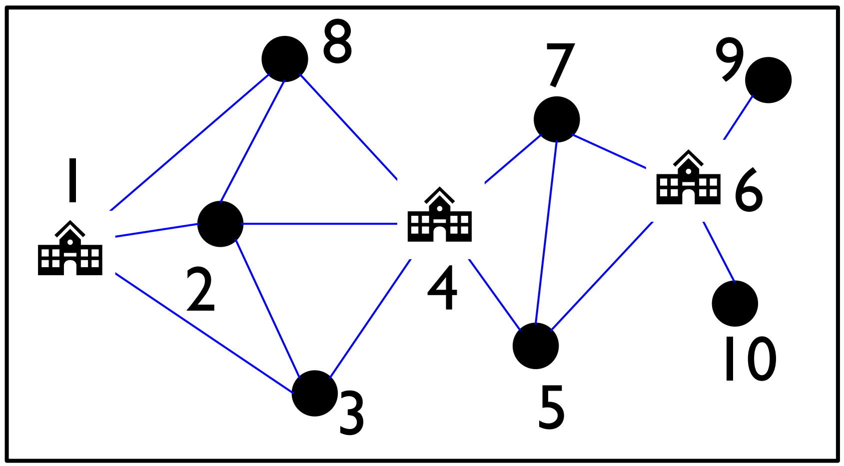
- Line 2: while-조건 ($U \neq \emptyset$) = ($\{1\} \neq \emptyset$)이 ‘참’이다.
- Line 3: U의 원소를 가장 많이 커버하는 집합인 $S_1 = \{1, 2, 3, 8\}$을 F에서 선택한다. S_1 대신 S_2, S_3, S_8 중에서 어느 하나를 선택해도 무방하다.
- Line 4: $U = U - S_1 = \{1\} - \{1, 2, 3, 8\} = \emptyset$
- Line 5: S_1을 F에서 제거하고, 즉, $F = \{S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\} - \{S_1\} = \{S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$이 되고, S_1을 C에 추가한다. 즉, $C = \{S_1, S_4, S_6\}$이다.
- Line 2: while-조건 ($U \neq \emptyset$) = ($\emptyset \neq \emptyset$)이 ‘거짓’이므로, 루프를 끝낸다.

4, 6, 1 cover만 차선

- Line 6: $C = \{S_1, S_4, S_6\}$을 리턴한다.

SetCover 예제

- SetCover 알고리즘의 최종해



시간복잡도

- 먼저 while-루프가 수행되는 횟수는 최대 n번이다.
  - 루프가 1번 수행될 때마다 집합 U의 원소 1개씩만 커버된다면, 최악의 경우 루프가 n번 수행되어야 하기 때문이다.
- 루프가 1번 수행될 때의 시간복잡도를 살펴보자.
  - Line 2의 while-루프 조건 (U ≠ ∅)을 검사는 O(1) 시간이 걸린다. 왜냐하면 U의 현재 원소 수를 위한 변수를 두고 그 값이 0인지를 검사하면 되기 때문이다.
  - Line 3에서 U의 원소들을 가장 많이 포함하고 있는 집합 S를 찾으려면, 현재 남아있는 S_i들 각각을 U와 비교하여야 한다. 따라서 S_i들의 수는 최대 n이고, 각 S_i와 U의 비교는 O(n) 시간이 걸리므로, line 3은 O(n^2) 시간이 걸린다.

시간복잡도

- Line 4에서는 집합 U에서 집합 S_i의 원소를 제거하는 것이므로 O(n) 시간이 걸린다.
- Line 5에서는 S_i를 F에서 제거하고, S_i를 C에 추가하는 것이므로 O(1) 시간이 걸린다.
- 따라서 루프 1회의 시간복잡도는 $O(1)+O(n^2)+O(n)+O(1) = O(n^2)$이다.

- SetCover 알고리즘 시간복잡도는 $O(n)\times O(n^2) = O(n^3)$이다.

근사해

K→최적해 Klogn →근사해

- 근사 알고리즘은 근사해가 최적해에 얼마나 근사한지(즉, 최적해에 얼마나 가까운지)를 나타내는 근사 비율(approximation ratio)을 알고리즘과 함께 제시하여야 한다.
- SetCover 알고리즘의 근사 비율은 Klogn 이며, 그 의미는 SetCover 알고리즘 최악 경우 해일지라도 그 집합 수가 Klogn개를 넘지 않는다는 뜻이다. 여기서 K는 최적해의 집합의 수이다.
  - 신도시 계획 예제에서는 최적해가 집합 2개로 모든 마을을 커버했으므로, Klogn = 2×log10 < 2×4 = 8이다. 가장 최악의 경우
  - 즉, SetCover 알고리즘이 찾은 근사해의 집합 수는 8개를 초과하지 않는다는 것이다.
- 집합 문제의 최적해를 찾는 데는 지수 시간이 걸리나, SetCover 알고리즘은 $O(n^3)$ 시간에 근사해를 찾으며 그 해도 실질적으로 최적해와 비슷하다.

응용

- 도시 계획(City Planning)에서 공공 기관 배치하기
- 경비 시스템: 미술관, 박물관, 기타 철저한 경비가 요구되는 장소의 CCTV 카메라의 최적 배치
- 컴퓨터 바이러스 찾기: 알려진 바이러스들을 '커버'하는 부분 스트링의 집합 - IBM에서 5000개의 알려진 바이러스들에서 9000개의 부분 스트링을 추출하였고, 이 부분 스트링의 집합 커버를 찾았는데, 총 180개의 부분 스트링이었다. 이 180개로 컴퓨터 바이러스의 존재를 확인하는데 성공하였다.
- 대기업의 구매 업체 선정: 각 업체가 제시하는 여러 종류의 부품들과 가격에 대해, 최소의 비용으로 구입하려고 하는 부품들을 모두 '커버'하는 업체를 찾기 위해 집합 문제의 해를 사용하였다.
- 기업의 경력 직원 고용: 각 지원자들이 여러 개의 기술을 보유하고 있을 때 회사가 필요로 하는 모든 기술을 커버하는 최소 인원을 찾으려면, 집합 문제의 해를 사용하면 된다.

4.6 작업 스케줄링

입력의 수: n

- 기계에서 수행되는 n개의 작업 $t_1, t_2, \ldots, t_n$이 있고, 각 작업은 시작시간과 종료시간이 있다.
- 작업 스케줄링(Task Scheduling) 문제는 작업의 수행 시간이 중복되지 않도록 모든 작업을 가장 적은 수의 기계에 배정하는 문제이다.
- 작업 스케줄링 문제는 대학에서 수업들을 강의실에 배정하는 문제와 같다.
  - 수업은 '작업'이고, 강의실은 '기계'로 생각할 수 있다.

작업 스케줄링

- 작업 스케줄링 문제에 주어진 문제 요소들은 다음과 같다.
  - 작업의 수
  - 각 작업의 시작시간과 종료시간
  - 작업의 시작시간과 종료시간은 정해져 있으므로 작업의 길이도 주어진 것이다.
- 여기서 작업의 수는 입력의 크기이므로 알고리즘을 고안하기 위해 고려되어야 하는 직접적인 요소는 아니다.

작업 스케줄링

- 시작시간, 종료시간, 작업 길이에 대해 다음과 같은 그리디 알고리즘들을 생각해볼 수 있다.
  - 빠른 시작시간 작업 우선(Earliest start time first) 배정
  - 빠른 종료시간 작업 우선(Earliest finish time first) 배정
  - 짧은 작업 우선(Shortest job first) 배정
  - 긴 작업 우선(Longest job first) 배정
- 위의 4가지 중 첫 번째 알고리즘을 제외하고 나머지 3가지는 항상 최적해를 찾지 못한다.

작업 스케줄링 알고리즘

JobScheduling

입력: n개의 작업 $t_1, t_2, \ldots, t_n$
출력: 각 기계에 배정된 작업 순서

```
시작시간의 오름차순으로 정렬한 작업 리스트를 L 이라고 하자.
while ( L ≠ Ø ) {
        L에서 가장 이른 시작시간 작업 t_i를 가져온다.
        if (t_i를 수행할 기계가 있으면)
                t_i를 수행할 수 있는 기계에 배정한다.
        else
                새로운 기계에 t_i를 배정한다.
        t_i를 L에서 제거한다.
}
return 각 기계에 배정된 작업 순서
```

작업 스케줄링 알고리즘

Line 1	시작시간에 대해 작업을 오름차순으로 정렬한다.
Line 2~8	while-루프는 L에 있는 작업이 다 배정될 때까지 수행된다.
Line 3	L에서 가장 이른 시작시간을 가진 작업 t_i를 가져온다.
Line 4~5	작업 t_i를 수행 시간이 중복되지 않게 수행할 기계를 찾아서, 그러한 기계가 있으면 t_i를 그 기계에 배정한다.
Line 6~7	기존의 기계들에 t_i를 배정할 수 없는 경우에는 새로운 기계에 t_i를 배정한다.
Line 8	작업 t_i를 L에서 제거하여, 더 이상 t_i가 작업 배정에 고려되지 않도록 한다.
Line 9	마지막으로 각 기계에 배정된 작업 순서를 리턴한다.

JobScheduling 예제

(입력)

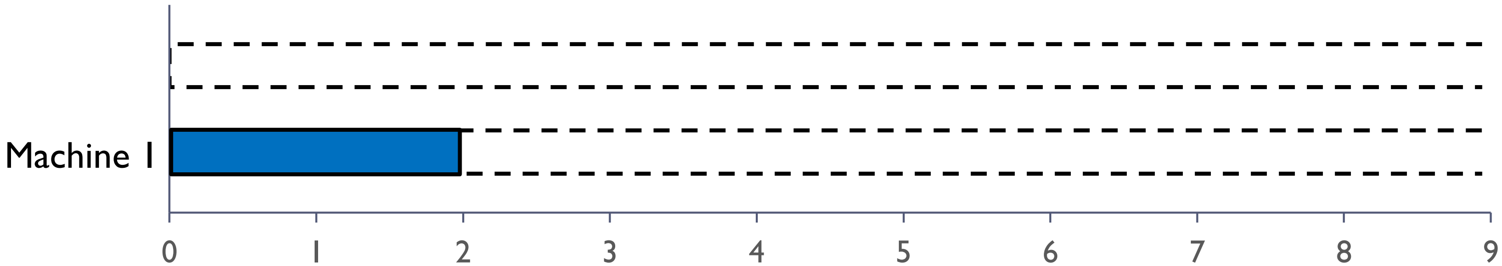
- t_1=[7, 8], t_2=[3, 7], t_3=[1, 5], t_4=[5, 9], t_5=[0, 2], t_6=[6, 8], t_7=[1, 6]
- 단, [s, f]에서 s는 작업의 시작시간, f는 작업의 종료시간이다.

(수행과정)

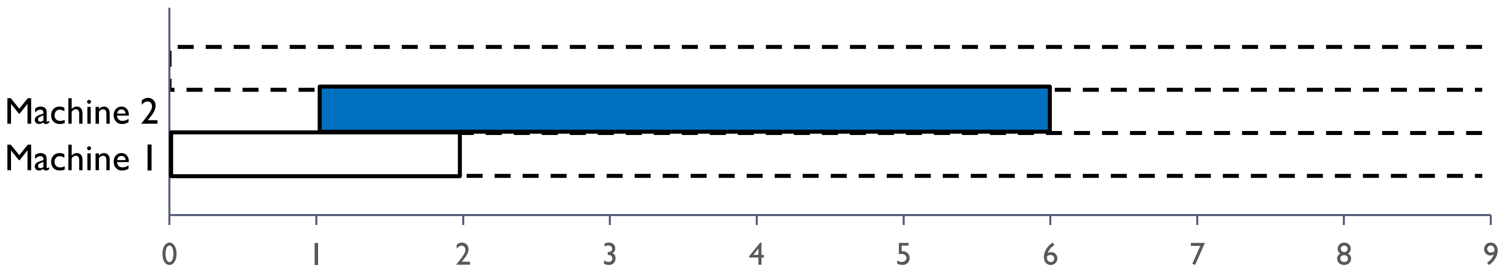
- Line 1: 시작시간의 오름차순으로 정렬한다.
  L = {[0, 2], [1, 6], [1, 5], [3, 7], [5, 9], [6, 8], [7, 8]} 이다.
- 다음 그림은 line 2~8까지의 while-루프가 수행되면서, 각 작업이 적절한 기계에 배정되는 것을 차례로 보이고 있다.

JobScheduling 예제

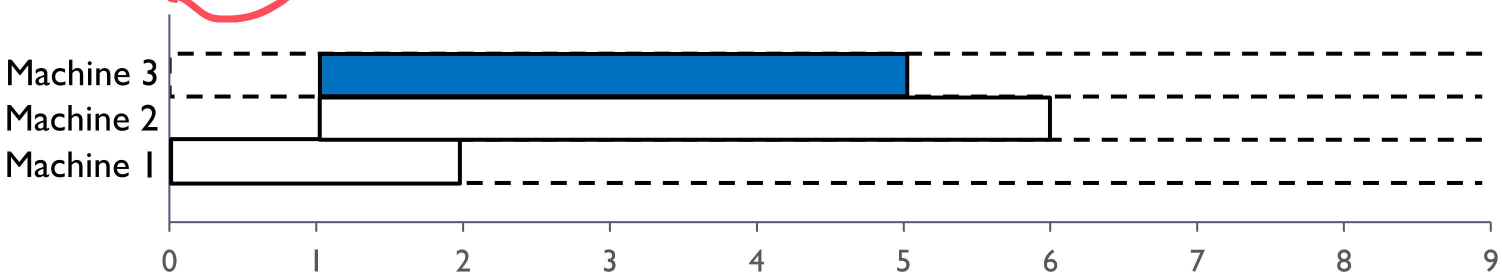
[0, 2]



[0, 2], [1, 6]

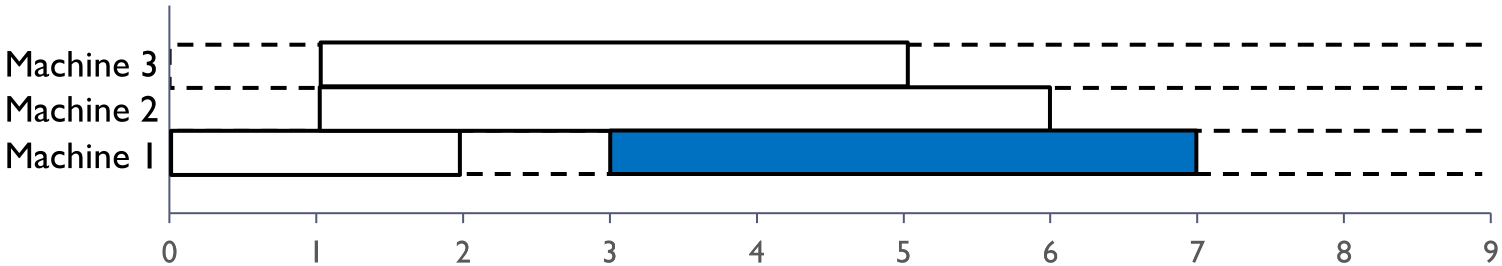


[0, 2], [1, 6], [1, 5] 아무거나

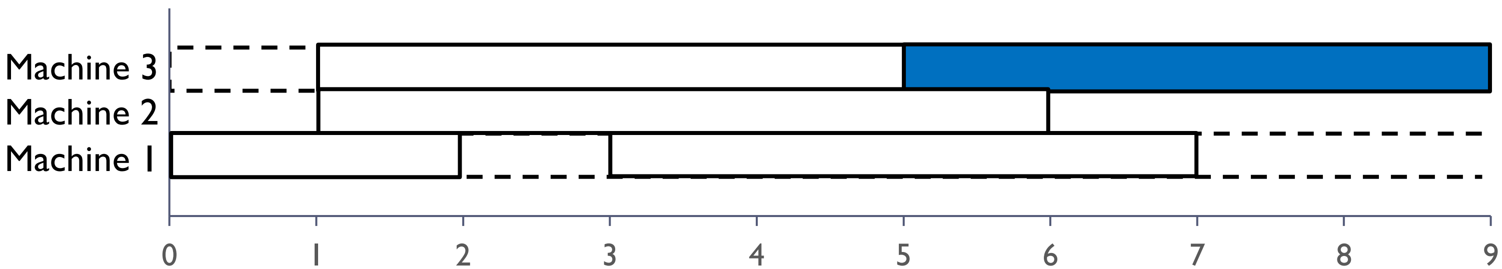


JobScheduling 예제

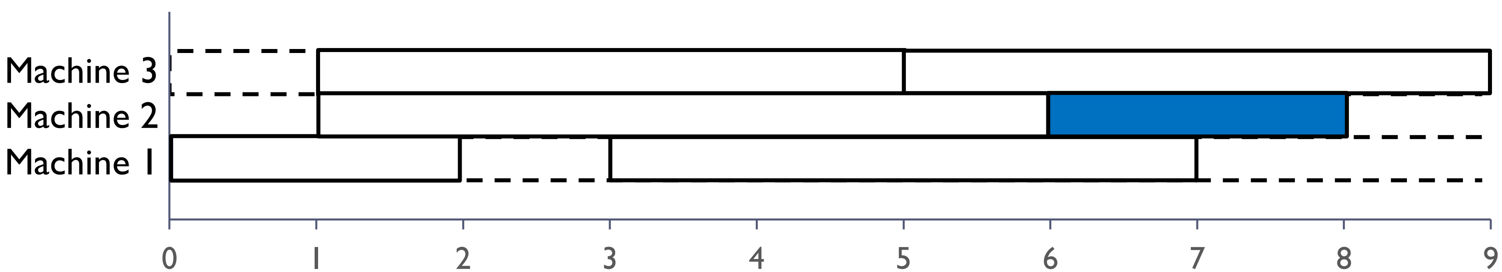
[0, 2], [1, 6], [1, 5], [3, 7]



[0, 2], [1, 6], [1, 5], [3, 7], [5, 9]

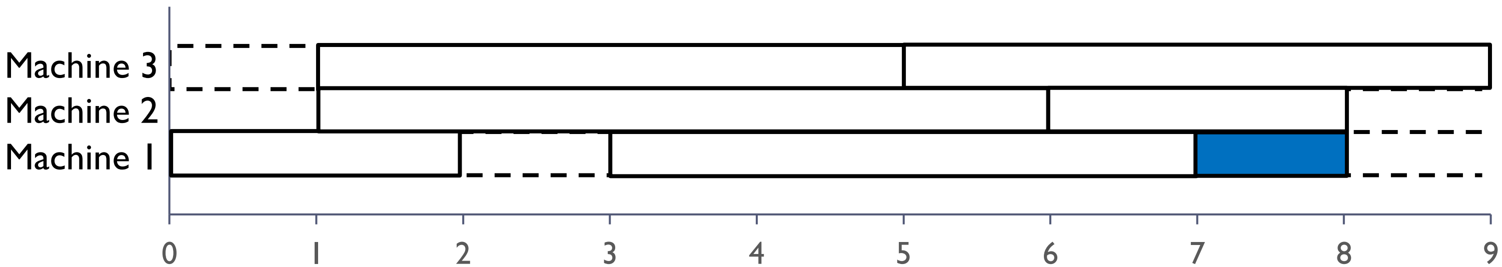


[0, 2], [1, 6], [1, 5], [3, 7], [5, 9], [6, 8]



JobScheduling 예제

[0, 2], [1, 6], [1, 5], [3, 7], [5, 9], [6, 8], [7, 8]



최종 결과

기계를 가동하면 최대한의 효율, 쉬지않고 일을 시킴

시간복잡도

- Line 1에서 n개의 작업을 정렬하는데 O(nlogn) 시간이 걸린다.
- while-루프에서는 작업을 L에서 가져다가 수행 가능한 기계를 찾아서 배정하므로 O(m) 시간이 걸린다. 단, m은 사용된 기계의 수이다.
- while-루프가 수행된 총 횟수는 n번이므로, line 2~9까지는 O(m)×n = O(mn) 시간이 걸린다.
- 따라서 JobScheduling 알고리즘의 시간복잡도는 O(nlogn)+O(mn)이다.

응용

기계가

가장 많은

하나밖에 없을 때 Task 사용

끝난시간을 생각. 뒤로 남은 공간↑

- 비즈니스 프로세싱
- 공장 생산 공정
- 강의실/세미나룸 배정
- 컴퓨터 태스크 스케줄링

많이 나오는 문자는 01, 적으면 긴 비트

4.7 허프만 압축

- 파일의 각 문자가 8bit 아스키(ASCII) 코드로 저장되면, 그 파일의 bit 수는 8×(파일의 문자 수)이다.
- 이와 같이 파일의 각 문자는 일반적으로 고정된 크기의 코드로 표현된다.
- 이러한 고정된 크기의 코드로 구성된 파일을 저장하거나 전송할 때 파일의 크기를 줄이고, 필요시 원래의 파일로 변환할 수 있으면, 메모리 공간을 효율적으로 사용할 수 있고, 파일 전송 시간을 단축시킬 수 있다.
- 주어진 파일의 크기를 줄이는 방법을 파일 압축(file compression)이라고 한다.

허프만 압축

- 허프만(Huffman) 압축은 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고, 드물게 나타나는 문자에는 긴 이진 코드를 할당한다.
- 허프만 압축 방법으로 변환시킨 문자 코드들 사이에는 접두부 특성(prefix property)이 존재한다. 자르는 코드
  - (의미) 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 접두부(prefix)가 되지 않는다.
  - (예) 문자 'a'에 할당된 코드가 '101'이라면, 모든 다른 문자의 코드는 '1' 또는 '10' 또는 '101'으로 시작되지 않는다.
  - (장점) 코드와 코드 사이를 구분할 특별한 코드가 필요 없다. 예를 들어, 101#10#1#111#0#…에서 '#'가 인접한 코드를 구분 짓고 있는데, 이러한 특별한 코드 없이 파일을 압축 및 해제할 수 있다.

허프만 압축 알고리즘

HuffmanCoding
입력: 입력 파일의 n개의 문자에 대한 각각의 빈도수 출력: 허프만 트리

```
각 문자에 대해 노드를 만들고, 그 문자의 빈도수를 노드에 저장한다.
n개의 노드의 빈도수에 대해 우선순위 큐 Q를 만든다.
while ( Q에 있는 노드 수 ≥ 2 ) {
      빈도수가 가장 작은 2개의 노드(A와 B)를 Q에서 제거한다.
      새 노드 N을 만들고, A와 B를 N의 자식 노드로 만든다.
      N의 빈도수 ← A의 빈도수 + B의 빈도수
      노드 N을 Q에 삽입한다.
}
return Q // 허프만 트리의 루트를 리턴하는 것이다.
```

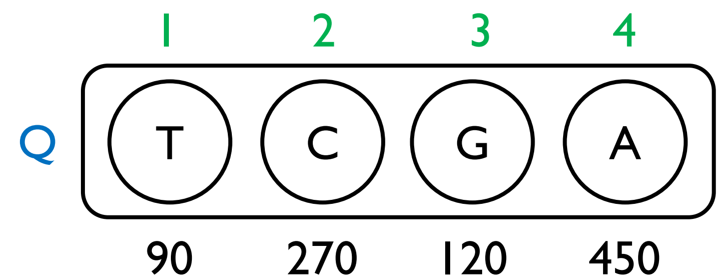
- 허프만 압축은 입력 파일에 대해 각 문자의 출현 빈도수에 기반을 둔 이진 트리를 만들어서, 각 문자에 이진 코드를 할당한다.

HuffmanCoding 예제

- 입력 파일은 4개의 문자로 되어 있고, 각 문자의 빈도수는 다음과 같다.
  - A: 450 T: 90 G: 120 C: 270

우선순위 큐를 만듦.

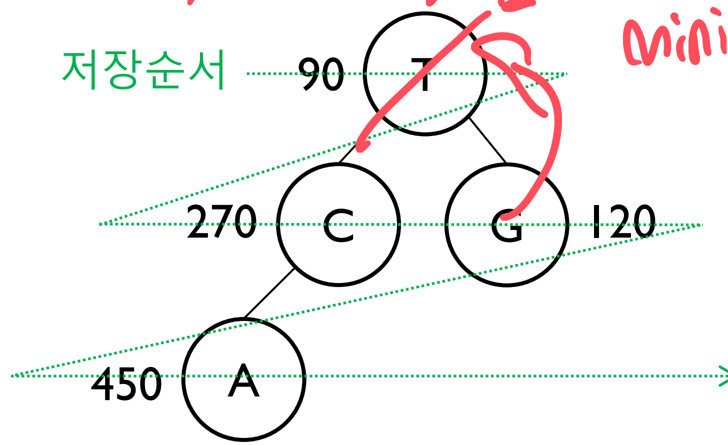
- Line 2를 수행한 후의 Q:



2개씩 낮은 거 갖고옴

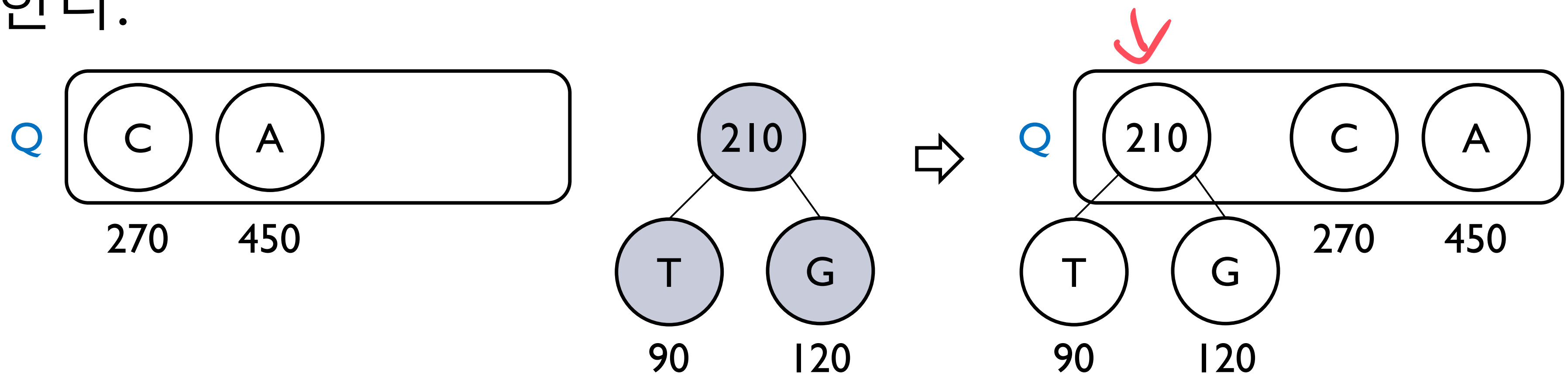
minimum

- 최소 힙 기반 우선순위 큐
  - 최소 힙은 각 노드의 값이 자식 노드의 값보다 작은 완전 이진 트리
  - 생성: O(n), 삽입/삭제: O(logn)
  - 부록 II 참고



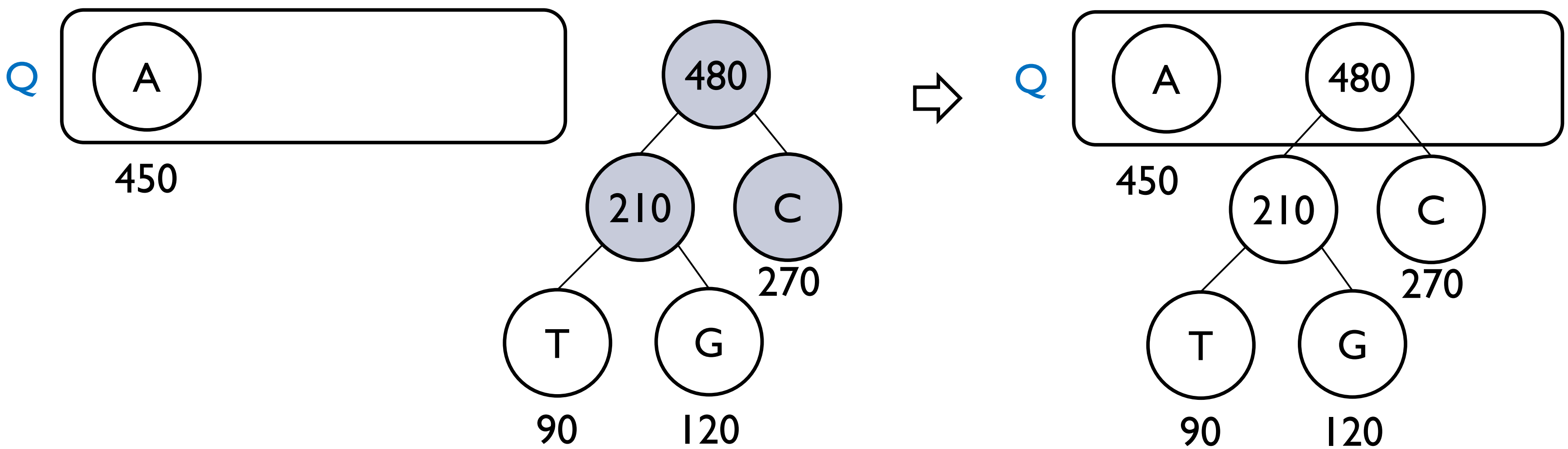
HuffmanCoding 예제

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



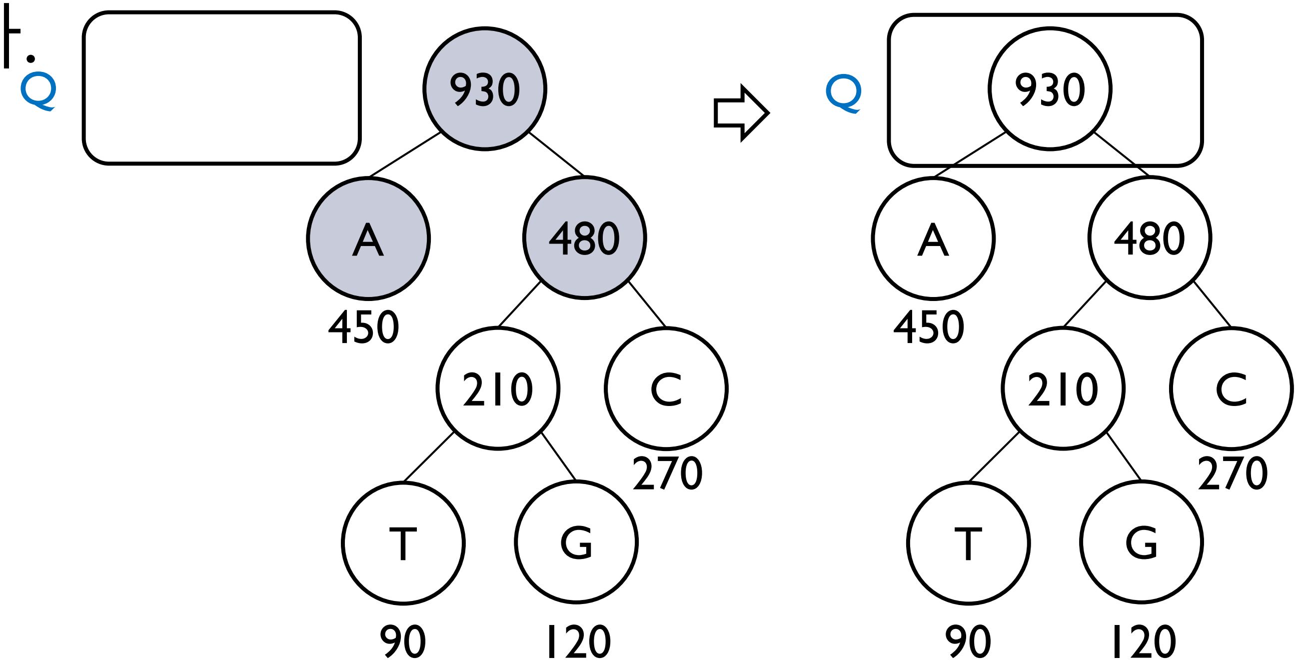
HuffmanCoding 예제

- 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 삽입한다.



HuffmanCoding 예제

- 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.



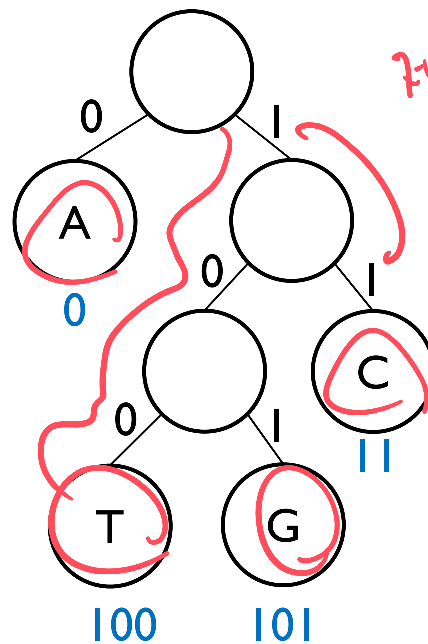
- Line 3의 while-루프 조건이 '거짓'이므로, line 8에서 Q에 있는 노드를 리턴한다. 즉, 허프만 트리의 루트를 리턴한다.

HuffmanCoding 예제

- 리턴된 트리를 살펴보면 각 단말(leaf) 노드에만 문자가 있다. 따라서 루트로부터 왼쪽 자식(child) 노드로 내려가면 '0'을, 오른쪽 자식 노드로 내려가면 '1'을 부여하면서, 각 단말에 도달할 때까지의 이진수를 추출하여 문자의 이진 코드를 구한다.
  - A: 0
  - T: 100
  - G: 101
  - C: 11

  → 접두부 특성 확인 가능

제일 많이 나오므로 0



중간에 끼워넣음 X, 위는 합이므로

접두부 특성을 가짐

HuffmanCoding 예제

- 할당된 코드들을 보면, 가장 빈도수가 높은 'A'가 가장 짧은 코드를 가지고, 따라서 루트의 자식 노드가 되어 있고, 빈도수가 낮은 문자는 루트에서 멀리 떨어지게 되어 긴 코드를 가진다.

- 앞의 예제에서 압축된 파일의 크기는
  $(450\times1)+(90\times3)+(120\times3)+(270\times2) = 1620$ bit이다.

- 아스키 코드 원본 파일 크기는
  $(450+90+120+270)\times8 = 7440$ bit이다.

- 따라서 파일 압축률은 $(1620/7440)\times100 = 21.8\%$이며, 원래의 약 1/5 크기로 압축되었다.

HuffmanCoding 예제

- 예제에서 얻은 허프만 코드로 아래의 압축된 부분에 대해서 압축을 해제하여 보면 다음과 같다.
  - A: 0
  - T: 100
  - G: 101
  - C: 11

1011001000111010101010100

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

G T T A C G A G A T

시간복잡도

- Line 1에서는 n개의 노드를 만들고, 각 빈도수를 노드에 저장하므로 O(n) 시간이 걸린다.
- Line 2에서는 n개의 노드로 우선순위 큐 Q를 만든다. 여기서 우선순위 큐로서 힙(heap) 자료구조를 사용하면 O(n) 시간이 걸린다.
- Line 3~7은 최소 빈도수를 가진 노드 2개를 Q에서 제거하는 힙의 삭제 연산과 새 노드를 Q에 삽입하는 연산을 수행하므로 O(logn) 시간이 걸린다. 그런데 while-루프는 (n–1)번 반복된다. 왜냐하면 루프가 1번 수행될 때마다 Q에서 2개의 노드를 제거하고 1개를 Q에 추가하기 때문이다. 따라서 line 3~7은 (n–1) ×O(logn) = O(nlogn)이 걸린다.
- Line 8은 트리의 루트를 리턴하는 것이므로 O(1) 시간이 걸린다.
- 따라서 시간복잡도는 O(n)+O(n)+O(nlogn)+O(1) = O(nlogn)이다.

우선순위 안쓰면 n^2

응용

- 데이터 압축
  - 팩스 (FAX), 대용량 데이터, 멀티미디어(Multimedia) 데이터 압축 등
- 정보 이론(Information Theory) 분야에서 엔트로피(Entropy)를 계산하는데 활용