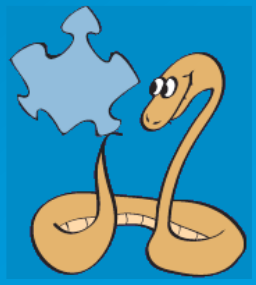
파이썬 알고리즘

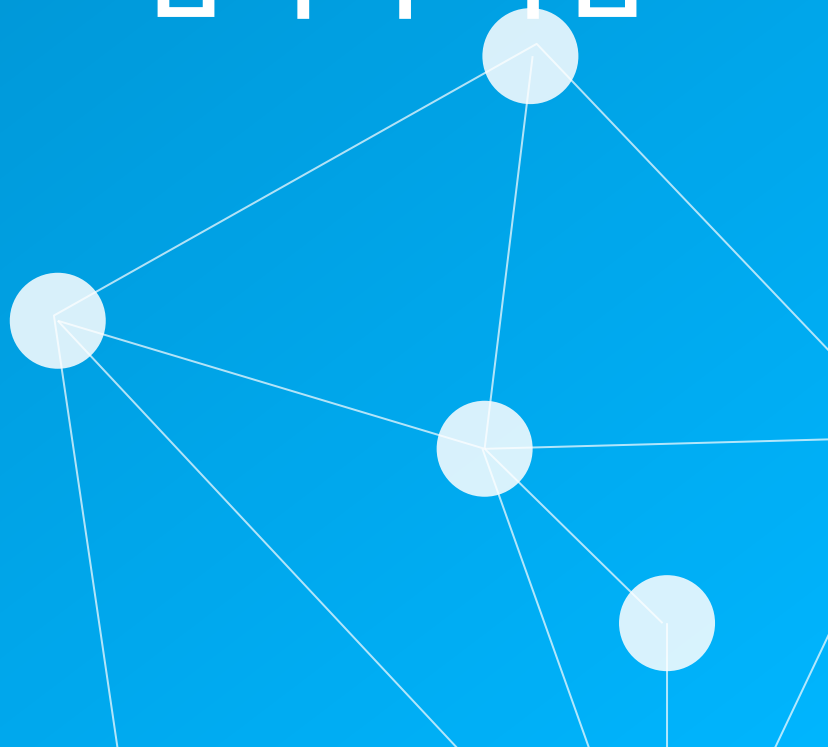
ALGORITHMS IN PYTHON 최영규 지음



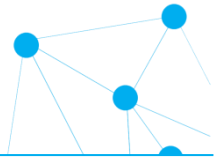
CHAPTER

08

탐욕적 기법

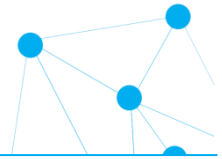


알고리즘의 설계 기법들



- 억지(brute-force) 기법과 완전 탐색: 3장
  - 문제 정의를 가장 직접 사용, 원하는 답 구할때까지 모든 경우 테스트
- 축소 정복(decrease-and-conquer): 4장
  - 주어진 문제를 하나의 좀 더 작은 문제로 축소하여 해결
- 분할 정복(divide-and-conquer): 5장
  - 주어진 문제를 여러 개의 더 작은 문제로 반복적으로 분할하여 해결 가능한 충분히 작은 문제로 분할 후 해결
- 공간을 이용해 시간을 버는 전략: 6장
  - 추가적인 공간을 사용하여 처리시간 줄이는 전략
- 동적 계획법(divide-and-conquer): 7장
  - 더 작은 문제로 나누는 분할정복과 유사하지만, 작은분제 먼저해결 저장하고 다음에 더 큰 문제 해결
- **탐욕적(greedy) 기법: 8장**
  - **단순하고 직관적인 방법으로 모든 경우 고려하여 가장 좋은 답을 찾는 것이 아니라 "그 순간에 최적"이라고 생각되는 것을 선택**
- 백트래킹과 분기 한정 기법: 9장
  - 상태공간에서 단계적 해 찾기, 현재의 최종 해가 않된다면 더 이상 탐색하지 않고 백트래킹(되돌아가서)해서 다른 후보 해 탐색

학습 내용



8.1 거스름돈 동전 최소화

8.2 분할 가능한 배낭 채우기

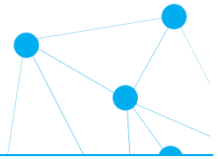
8.3 최소비용 신장트리: Prim 알고리즘

8.4 최소비용 신장트리: Kruskal 알고리즘

8.5 Dijkstra의 최단경로 알고리즘

8.6 허프만 코드

탐욕적 기법(greedy method)



- Greedy method
  - 어떤 결정을 해야 할 때 마다 "그 순간에 최적"이라고 생각되는 것을 선택 ("근시안적" 알고리즘)
  - 최적의 해답을 주는지 검증해야 함
- 탐욕적 기법이 사용될 수 있는 경우
  - 탐욕적 기법을 사용해도 항상 최적해를 구할 수 있는 문제
  - 최적해를 구하는 것이 현실적으로 불가능한 경우
    - 예) 분기 한정 기법(9장)에서 좋은 한계값를 구하기 위해 사용
- 알고리즘 설계 과정

① 현재 주어진 문제에 대한 가장 탐욕적인 해를 선택한다.
② 이 해가 가능한 해인지를 검사한다. 만약 가능하지 않다면 ①로 돌아가 다음으로 가장 탐욕적인 해를 선택한다.
③ 만약 가능한 해라면, 문제가 해결되었는지를 검사한다. 만약 아직 문제가 해결되지 않았으면 이 해를 적용해 크기가 줄어든 부분 문제에 대해 다시 ①로 돌아가 해를 구한다.

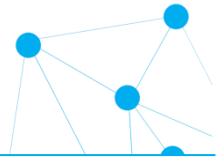
8.1 거스름돈 동전 최소화

액면가가 서로 다른 m가지의 동전 $\{C_1, C_2, \cdots, C_m\}$이 있다. 거스름돈으로 V원을 동전으로만 돌려주어야 한다면 최소 몇 개의 동전이 필요한지를 구하라. 단, 모든 동전은 무한히 사용할 수 있고, 동전들은 액수가 큰 것부터 내림차순으로 순서대로 정렬되어 있다.

- 우리나라 동전

  - 620원: 500원 + 100원 + 10원×2 → 동전 4개
  - 345원: 100원×3 + 10원×4 + 5원 → 동전 8개
  - 572원: 500원 + 50원 + 10원×2 + 1원×2 → 동전 6개
  - 탐욕적 전략
    - 가장 액수가 큰 동전부터 최대한 사용하면서 거스름돈을 맞춤
  - 현재까지는 최적해. 만약 60원 동전이 생긴다면?
- 항상 최적해를 구하는 방법
  - 방법 1: 완전 탐색 → 지수함수
  - 방법 2: 동적 계획법 → $O(mV)$

탐욕적 전략의 동전 최소화 알고리즘



- **탐욕적인 해**: 액면가가 가장 큰 동전을 사용한다. 남은 잔돈을 가장 작게 만들어 전체 동전의 수를 최소화시킬 가능성이 가장 크기 때문이다.
- **가능한 해인지 검사**: 사용한 동전이 잔돈을 초과하지 않아야 한다.

  - 예: 580원 → 5개
    - 500 * 1 + 100 * 0 + 50 * 1 + 10 * 3

- 알고리즘

알고리즘 8.1 거스름돈 최소화 알고리즘

```
01  def min_coins_greedy( coins, V ):
02      count = []
03      for i in range(len(coins)) :
04          cnt = V // coins[i]
05          count.append(cnt)
06          V -= cnt * coins[i]
07      return count
```

알고리즘 테스트 탐욕적 전략의 거스름돈 동전 최소화

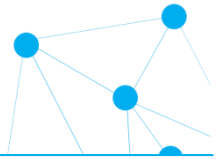
```
coins = [500 , 100 , 50 , 10 , 5 , 1]
V = 580
print("잔돈 액수 = ", V)
print("동전 종류 = ", coins)
print("동전 개수 = ", min_coins_greedy(coins, V ))
```

```
C:₩WINDOWS₩system32₩cmd.exe
잔돈 액수 =  580
동전 종류 =  [500, 100, 50, 10, 5, 1]
동전 개수 =  [1, 0, 1, 3, 0, 0]
```

각 동전의 개수

- 복잡도: $O(m)$

최적해를 보장할까?



- 동전 체계에 따라 다름(항상 보장하지는 않음)

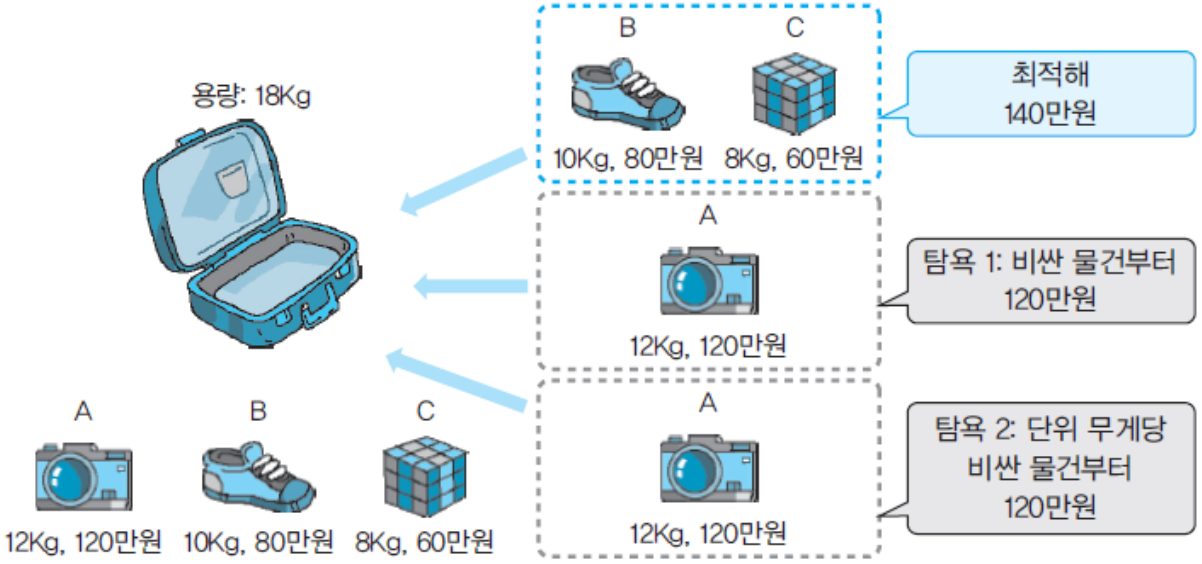
> 동전의 액면가 중에서 어떤 두 개를 고르더라도 큰 액면가를 작은 액면가로 나누어 떨어지는 동전 체계를 갖는다면 최적해를 보장한다. 작은 액면가를 여러 개 모으면 반드시 큰 액면가를 만들 수 있기 때문이다.

  - 예: 우리나라에 60원 동전이 생긴다면?
    - 120원? 100원*1 + 10원 * 2 → 3개
    - 120원? 60원*2 → 2개

8.2 분할 가능한 배낭 채우기

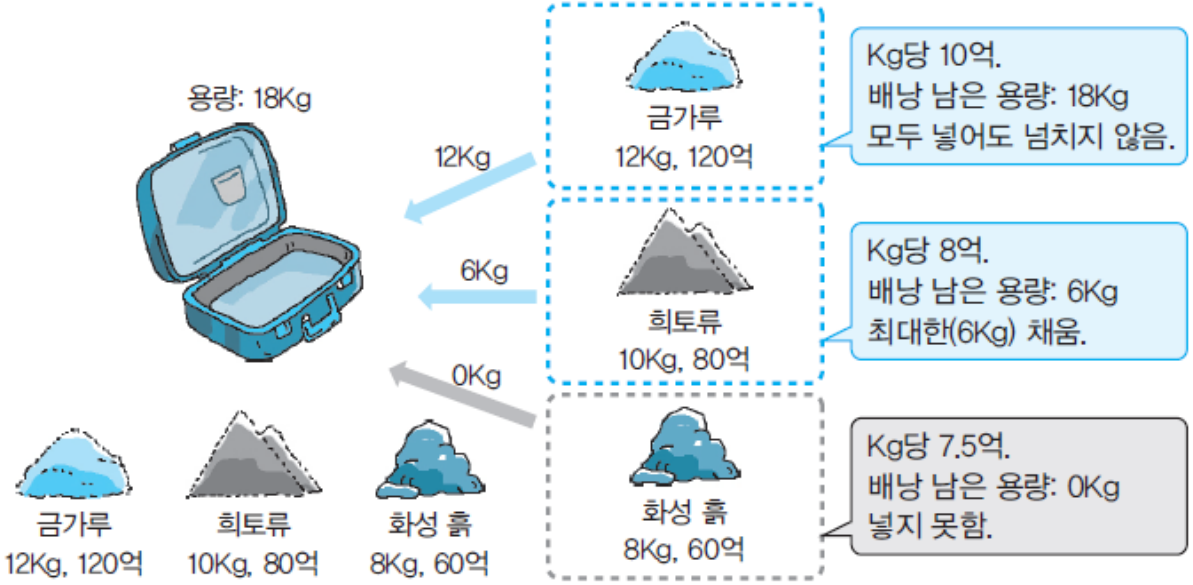
- 0-1 배낭 채우기 탐욕적 전략
  - 탐욕 1: 무게와 상관없이 가장 비싼 물건부터 넣어보는 방법
  - 탐욕 2: 단위 무게당 가격이 가장 높은 물건부터 넣어보는 방법
  - 탐욕 3: 가장 가벼운 물건부터 넣어보는 방법

- 0-1 배낭 채우기: 탐욕적 전략으로 최적해를 구하지 못함



분할 가능한 배낭 채우기 문제

- 0-1 배낭 채우기 문제(문제 3.6)을 변형
  - 물건들은 분할해서 일부분만을 넣을 수도 있다.
- 탐욕적 기법
  - 탐욕적인 해: 무게당 가격이 비싼 물건부터 순서대로 배낭의 남은 용량을 넘지 않는 최대한으로 채움
  - 가능한 해 검사: 배낭의 용량을 넘치지만 않으면 항상 가능한 해



알고리즘

알고리즘 8.2 분할 가능한 배낭 채우기(탐욕적 기법)

```
01  def knapSack_fractional_greedy(obj, W):
02      obj.sort(key = lambda o: o[2]/o[1], reverse=True)
03
04      totalValue = 0
05      for o in obj :
06          if W <= 0 : break
07          if W - o[1] >= 0:
08              W -= o[1]
09              totalValue += o[2]
10          else:
11              fraction = W / o[1]
12              totalValue += o[2] * fraction
13              W = int(W - (o[1] * fraction))
14
15      return totalValue
```

알고리즘 테스트 분할 가능한 배낭 채우기(탐욕적 기법)

```
obj = [ ("A", 10, 80), ("B", 12, 120), ("C", 8, 60)]    # (물건,무게,가치)
print("W = 18 ", obj)
print("부분적인배낭(18): ", knapSack_fractional_greedy(obj,18), end='\n\n')

obj = [ ("A", 10, 60), ("B", 40, 40), ("C", 20, 100), ("D", 30, 120) ]
print("W = 50 ", obj)
print("부분적인배낭(50): ", knapSack_fractional_greedy(obj, 50))
```

```
C:₩WINDOWS₩system32₩cmd.exe
W = 18  [('A', 10, 80), ('B', 12, 120), ('C', 8, 60)]
부분적인배낭(18):  168.0
W = 50  [('A', 10, 60), ('B', 40, 40), ('C', 20, 100), ('D', 30, 120)]
부분적인배낭(50):  240.0
```

120(B) + 6*80/10(A) + 0(C)

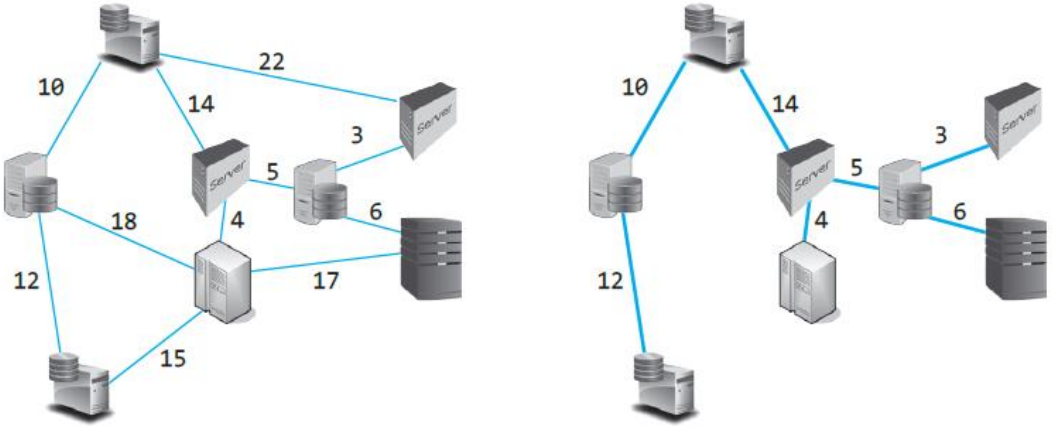
60(A) + 100(C) + 20*120/30(D) + 0(B)

- 시간 복잡도: 정렬에 좌우 → $O(nlog_2n)$
- 항상 최적해를 구함

8.3 최소비용 신장트리: Prim

가중치 그래프 $G=(V, E)$가 주어졌다. G의 모든 신장트리 중에서 간선들의 가중치 합이 최소인 신장트리를 찾아라. 이러한 트리를 **최소비용 신장트리**(MST: minimum spanning tree)라고 한다.

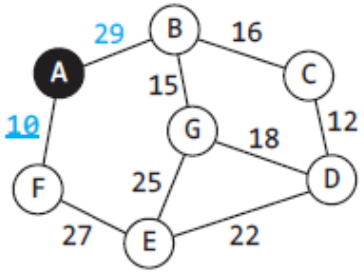
- 신장트리(spanning tree)
  - 그래프 내의 모든 정점을 포함하는 트리
- 최소비용 신장트리(MST)
  - "간선들의 비용 최소화"하는 신장트리



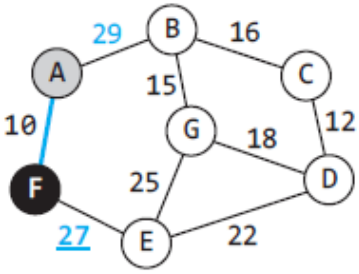
(a) 사이트 사이의 연결 비용 (b) 최소비용 신장트리의 예

Prim의 탐욕

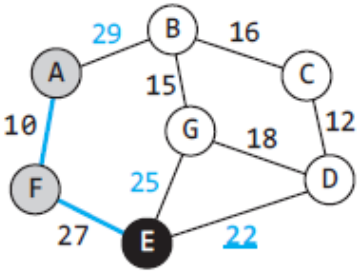
- 현재까지 만들어진 MST의 인접 정점들 중에서 간선 가중치가 최소인 정점을 반복적으로 선택해 트리를 확장



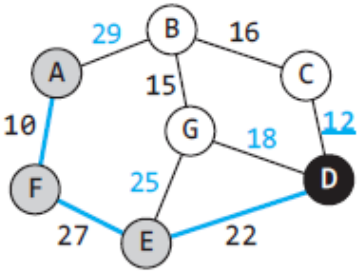
(a) 시작 정점 A 추가



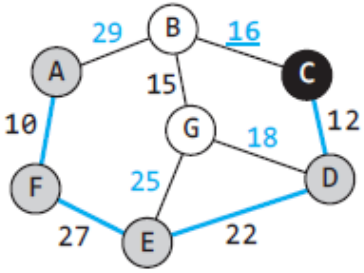
(b) 최소 간선 (A,F) 선택
정점 F 추가



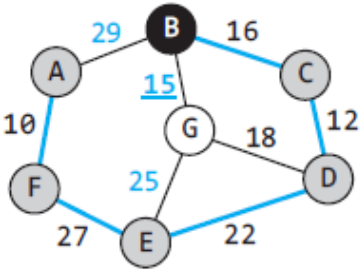
(c) 최소 간선 (E,F) 선택
정점 E 추가



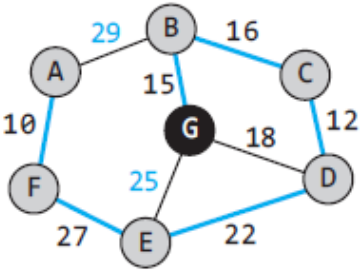
(d) 최소 간선 (E,D) 선택
정점 D 추가



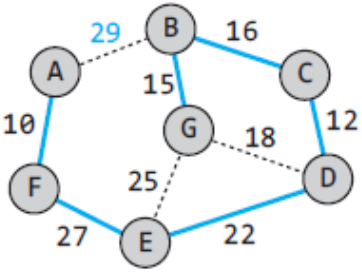
(e) 최소 간선 (C,D) 선택
정점 C 추가



(f) 최소 간선 (B,C) 선택
정점 B 추가

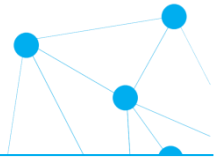


(g) 최소 간선 (B,G) 선택
정점 G 추가



(h) 최종 MST 완성

알고리즘 정확성

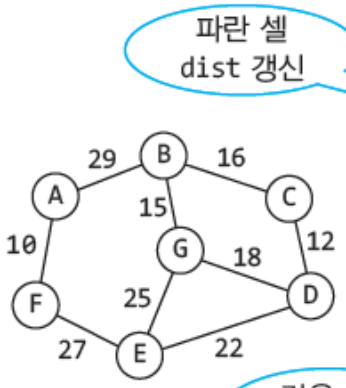


알고리즘 8.3 Prim의 최소비용 신장트리 알고리즘(자연어)

```
Prim()
1. 그래프에서 시작 정점을 선택하여 초기 트리를 만든다.
2. 현재 트리의 정점들과 인접한 정점들 중에서 간선의 가중치가 가장 작은 정점 v를 선택한다.
3. 이 정점 v와 이때의 간선을 트리에 추가한다.
4. 모든 정점이 삽입될 때 까지 2번으로 이동한다.
```

- Prim 알고리즘의 정확성(Correctness)
  - 심화학습: 333~334쪽

알고리즘: 가중치가 가장 작은 간선의 선택



파란 셀 dist 갱신

검은 셀 확정된 정점

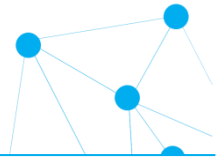
인덱스	0	1	2	3	4	5	6	
	A	B	C	D	E	F	G	
dist	0	INF	INF	INF	INF	INF	INF	맨 처음: 시작 정점 A → dist[A]=0
	0	29	INF	INF	INF	10	INF	최소 정점 A 선택 → dist[F, B] 갱신
	0	29	INF	INF	27	10	INF	최소 정점 F 선택 → dist[E] 갱신
	0	29	INF	22	27	10	25	최소 정점 E 선택 → dist[D, G] 갱신
	0	29	12	22	27	10	18	최소 정점 D 선택 → dist[C, G] 갱신
	0	16	12	22	27	10	18	최소 정점 C 선택 → dist[B] 갱신
	0	16	12	22	27	10	15	최소 정점 B 선택 → dist[G] 갱신
	0	16	12	22	27	10	15	최소 정점 G 선택 → MST 완성

[그림 8.13] 최소 간선 선택을 위한 dist 배열 변화 과정

- getMinVertex()
  - dist[]
  - selected[]

```
01  def getMinVertex(dist, selected) :
02      minv = -1
03      mindist = INF
04      for v in range(len(dist)) :
05          if not selected[v] and dist[v]<mindist :
06              mindist = dist[v]
07              minv = v
08      return minv
```

알고리즘: Prim의 MST



알고리즘 8.5 Prim의 최소비용 신장트리 알고리즘

```
01  def MSTPrim(vertex, adj) :
02      vsize = len(vertex)
03      dist = [INF] * vsize
04      dist[0] = 0
05      selected = [False] * vsize
06
07      for i in range(vsize) :
08          u = getMinVertex(dist, selected)
09          selected[u] = True
10          print(vertex[u], end=':')
11          print(dist)
12
13          for v in range(vsize) :
14              if (adj[u][v]  != None):
15                  if selected[v]==False and adj[u][v]< dist[v] :
16                      dist[v] = adj[u][v]
```

알고리즘 테스트 Prim의 MST 알고리즘

```
vertex = ['A',... ]              # 그림 8.12의 vertex
weight = [ ... ]                 # 그림 8.12의 weight
print("MST By Prim's Algorithm")
MSTPrim(vertex, weight)
```

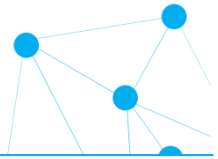
정점이 선택되는 과정

```
md.exe
MST By Prim's Algorithm
A: [0, 29, 9999, 9999, 9999, 10, 9999]
F: [0, 29, 9999, 9999, 27, 10, 9999]
E: [0, 29, 9999, 22, 27, 10, 25]
D: [0, 29, 12, 22, 27, 10, 18]
C: [0, 16, 12, 22, 27, 10, 18]
B: [0, 16, 12, 22, 27, 10, 15]
G: [0, 16, 12, 22, 27, 10, 15]
```

dist 배열 변화

- 시간 복잡도: $O(n^2)$

8.4 최소비용 신장트리: Kruskal



- 현재 선택할 수 있는 가장 가중치가 작은 간선을 선택
- 문제: 만약 MST에 사이클이 생긴다면?
  - 그 간선을 넣지 않아야 함.

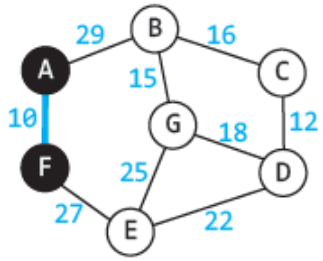
알고리즘 8.6 Kruskal의 최소비용 신장트리 알고리즘(자연어)

kruskal()

1. 그래프의 모든 간선을 가중치에 따라 오름차순으로 정렬한다.
2. 가장 가중치가 작은 간선 e를 뽑는다.
3. e를 신장트리에 넣었을 때 사이클이 생기면 넣지 않고 2번으로 이동한다.
4. 사이클이 생기지 않으면 최소 신장트리에 삽입한다.
5. n-1개의 간선이 삽입될 때 까지 2번으로 이동한다.

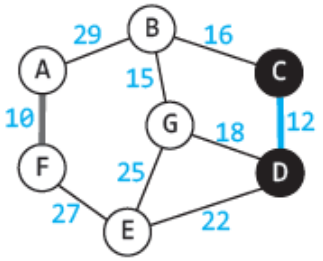
Kruskal 의 탐욕

- 최소 가중치 간선



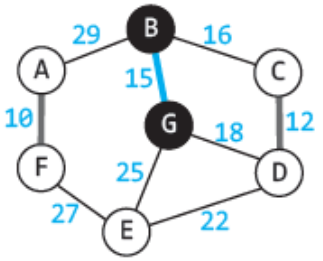
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(a) AF 선택 → MST 추가



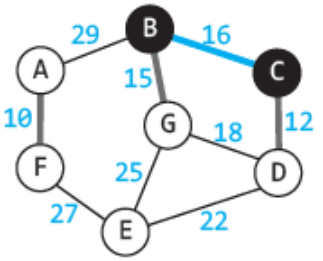
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(b) CD 선택 → MST 추가



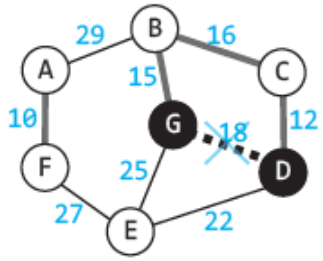
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(c) BG 선택 → MST 추가



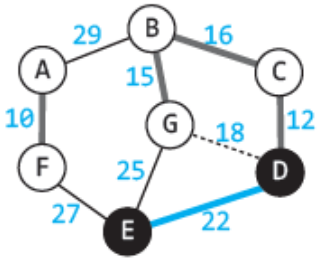
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(d) BC 선택 → MST 추가



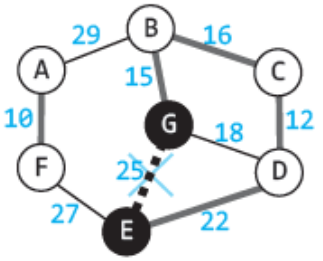
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(e) CD 선택 → 사이클 발생



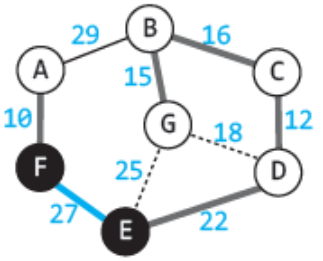
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(f) DE 선택 → MST 추가



AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

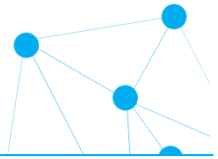
(g) EG 선택 → 사이클 발생



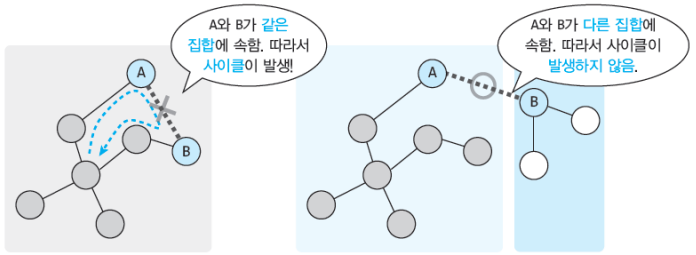
AF	CD	BG	BC	DG	DE	EG	EF	AB
10	12	15	16	18	22	25	27	29

(h) EF 선택 → MST 추가
MST 완성

사이클 검사

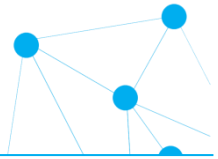


- 새로운 간선이 사이클을 만드는 경우?
  - 이미 다른 경로를 통해 연결된 정점들 사이의 간선인 경우



- 해결 방안
  - 서로소 부분집합과 Union-Find 알고리즘
    - find(x): 원소 x가 속한 집합 → 보통은 "대표원소" 반환
    - union(x,y): 원소 x와 y가 속한 집합을 하나로 뭉침

트리를 이용한 Union-Find 구현

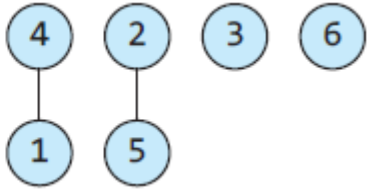


- S={1, 2, 3, 4, 5, 6}

  {1}, {2}, {3}, {4}, {5}, {6}

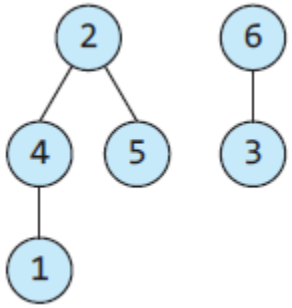
- union(1, 4)와 union(5, 2)

  {1, 4}, {5, 2}, {3}, {6}



- union(4, 5)와 union(3, 6)

  {1, 4, 5, 2}, {3, 6}

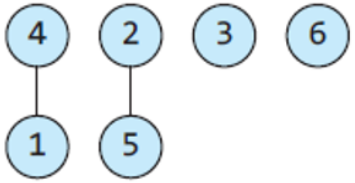


- S={1, 2, 3, 4, 5, 6}

  {1}, {2}, {3}, {4}, {5}, {6}

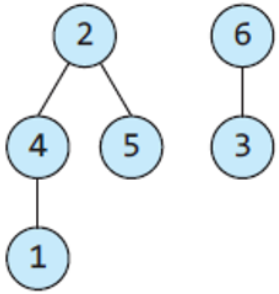
- union(1, 4)와 union(5, 2)

  {1, 4}, {5, 2}, {3}, {6}

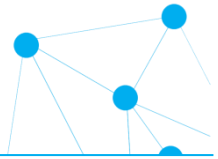


- union(4, 5)와 union(3, 6)

  {1, 4, 5, 2}, {3, 6}



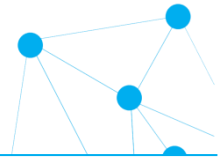
클래스 구현: disjointSets(서로소 집합)



클래스 8.1 disjointSets(서로소 집합)

```
01  class disjointSets:
02      def __init__(self, n):
03          self.parent = [-1]*n
04          self.set_size = n
05
06      def find(self, id) :
07          while (self.parent[id] >= 0) :
08              id = self.parent[id]
09          return id
10
11      def union(self, s1, s2) :
12          self.parent[s1] = s2
13          self.set_size = self.set_size-1
```

알고리즘: Kruskal의 MST



알고리즘 8.7 Kruskal의 최소비용 신장트리

```
01  def MSTKruskal(V, adj):
02      n = len(V)
03      dsets = disjointSets(n)
04      E = []
05      for i in range(n-1) :
06          for j in range(i+1, n) :
07              if adj[i][j] != None :
08                  E.append((i,j,adj[i][j]))
09
10      E.sort(key= lambda e : e[2])
11
12      ecount = 0
13      for i in range(len(E)):
14          e = E[i]
15          uset = dsets.find(e[0])
16          vset = dsets.find(e[1])
17
18          if uset != vset :
19              dsets.union(uset, vset)
20              print("간선 추가 : (%s, %s,
21              ecount += 1
22              if ecount == n-1 :
23                  break
```

알고리즘 테스트 Kruskal의 MST 알고리즘

```
vertex = ['A',... ]              # 그림 8.12의 vertex
weight = [ ... ]                 # 그림 8.12의 weight
print("MST By Kruskal's Algorithm")
MSTKruskal1(vertex, weight)
```

```
C:₩WINDOWS₩system32₩cmd.exe
MST By Kruskal's Algorithm
간선 추가 : (A, F, 10)
간선 추가 : (C, D, 12)
간선 추가 : (B, G, 15)
간선 추가 : (B, C, 16)
간선 추가 : (D, E, 22)
간선 추가 : (E, F, 27)
```

간선이 선택되는 과정

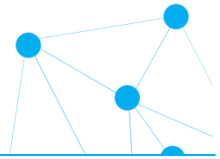
- 시간 복잡도(인접행렬)
  - 3-5행: $O(n^2)$
  - 10행: $O(elog_2e)$

  → $O(n^2 + elog_2e)$

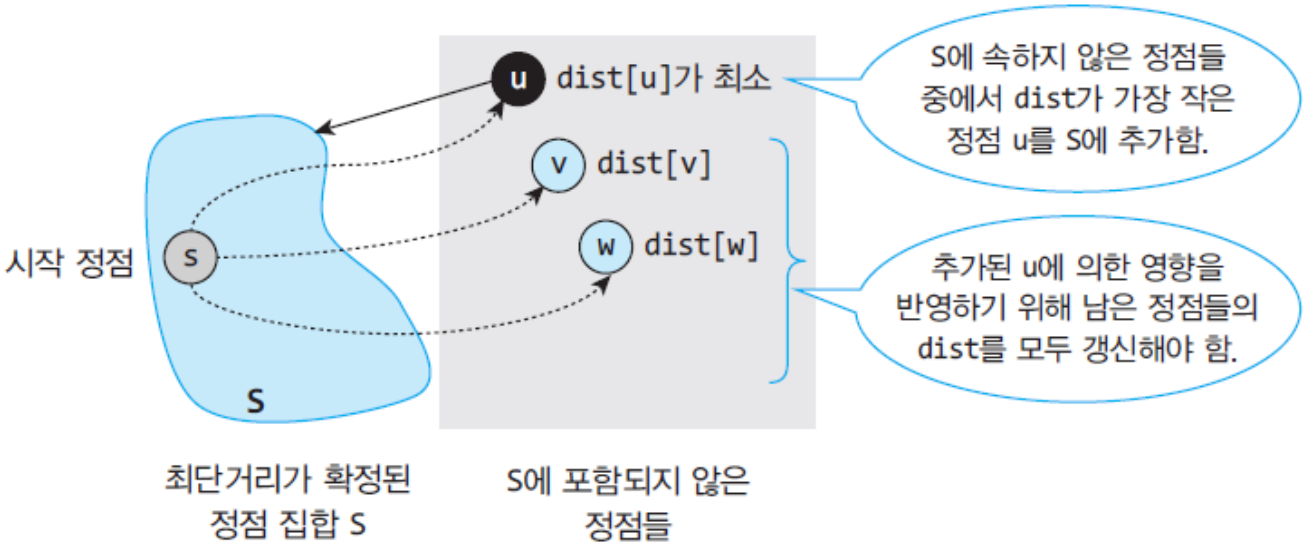
- 인접 리스트 표현:

  → $O(elog_2e)$

8.5 Dijkstra의 최단경로 알고리즘

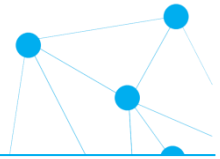


- 한 정점에서 다른 모든 정점까지의 최단 경로 거리
- Dijkstra가 선택한 탐욕
  - 최단거리가 확정된 정점들과 간선으로 직접 연결된 정점들 중에서 가장 가까운 정점을 선택한다.



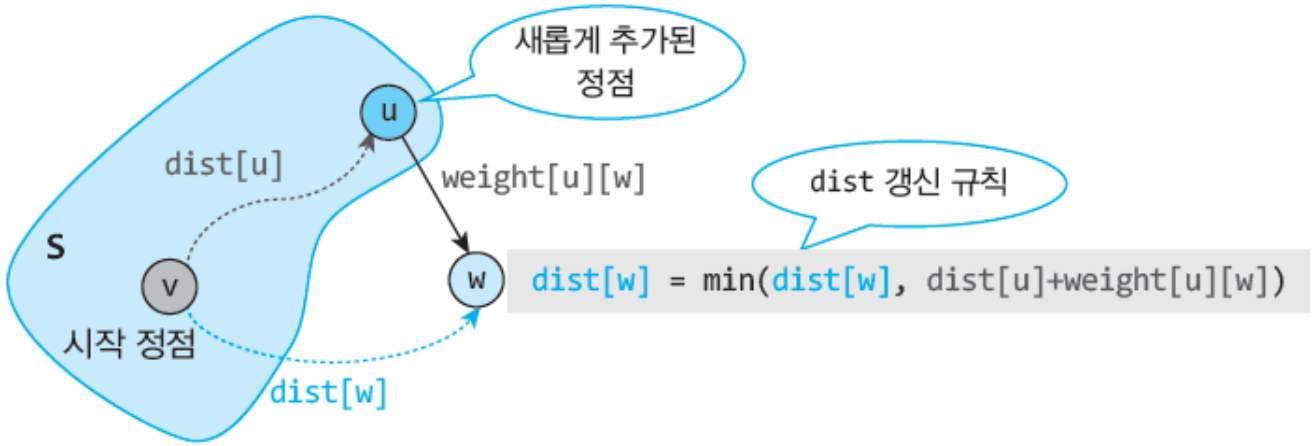
[그림 8.17] Dijkstra 최단경로 알고리즘의 기본 아이디어

Distance 갱신



- 각 정점 w에 대해
  - u를 거쳐서 w로 가는 새로운 경로가 더 짧다면 dist[w] 갱신

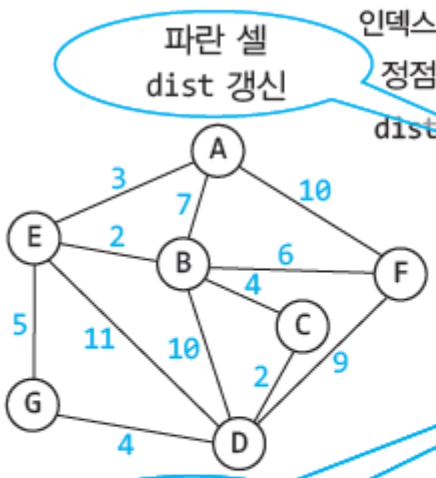
    dist[w] = dist[v] + weight[u][w]
  - 그렇지 않다면 dist[w] 유지



[그림 8.18] 최단경로 알고리즘에서의 distance 갱신

Dijkstra 알고리즘 진행 과정

- A가 시작 정점인 경우



파란 셀 dist 갱신

검은 셀 거리가 확정된 정점

인덱스	0	1	2	3	4	5	6
정점	A	B	C	D	E	F	G
dist	0	7	INF	INF	3	10	INF
	0	5	INF	14	3	10	8
	0	5	9	14	3	10	8
	0	5	9	12	3	10	8
	0	5	9	11	3	10	8
	0	5	9	11	3	10	8
	0	5	9	11	3	10	8
최단 거리	0	5	9	11	3	10	8

❶ 시작 정점 A → 모든 dist 초기화
❷ 최소 정점 E 거리 확정 → dist[B,D,G] 갱신
❸ 최소 정점 B 선택 → dist[C] 갱신
❹ 최소 정점 G 선택 → dist[D] 갱신
❺ 최소 정점 C 선택 → 갱신할 노드 없음
❻ 최소 정점 F 선택 → 갱신할 노드 없음
❼ 최소 정점 d 선택 → 전체 노드 거리 확정

A부터 각 정점까지의 최단 거리

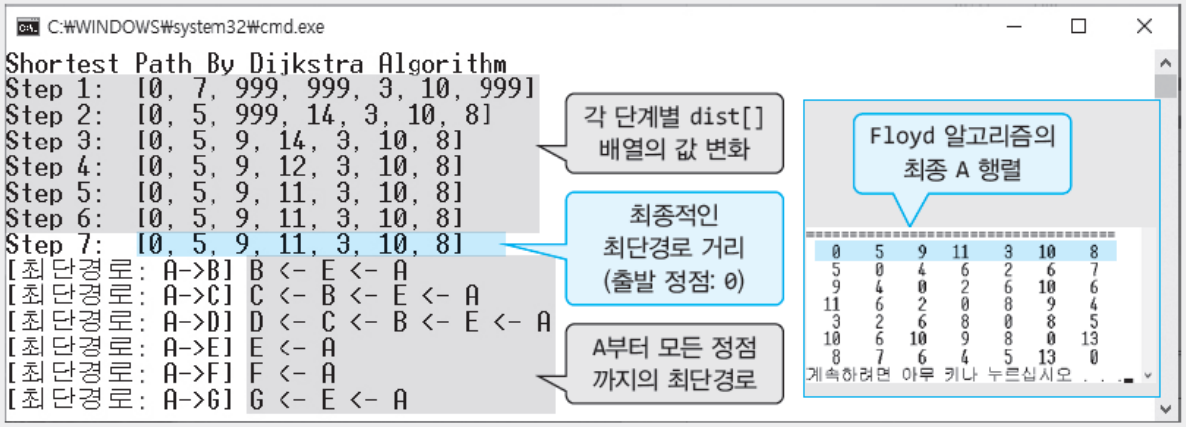
[그림 8.19] Dijkstra 알고리즘 진행 과정

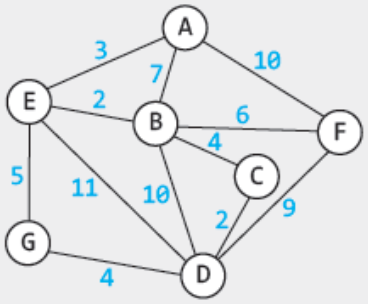
- 탐욕적 선택 단계의 타당성
  - 심화학습: p351

알고리즘

알고리즘 8.8 Dijkstra의 최단경로 알고리즘

```
01  def shortest_path_dijkstra(vtx, adj, start) :
02      vsize = len(vtx)
03      dist = list(adj[start])
04      dist[start] = 0
05      path = [start] * vsize
06      found= [False] * vsize
07      found[start] = True
08
09      for i in range(vsize) :
10          print("Step%2d:  "%(i+1),dist)
11          u = getMinVertex(dist, found)
12          found[u] = True
13
14          for w in range(vsize) :
15              if not found[w] :
16                  if dist[u] + adj[u][w] < dist[w] :
17                      dist[w] = dist[u] + adj[u][w]
18                      path[w] = u
19
20      return path
```

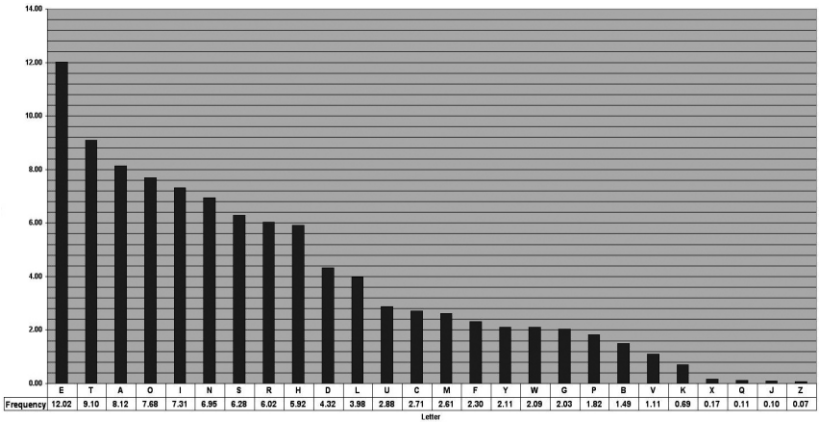




- 시간 복잡도: $O(n^2)$

8.6 허프만 코드

- 영어 알파벳별 빈도수 예



- 가변 길이 코드

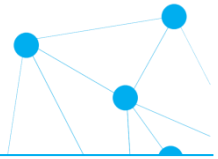
빈도수 테이블	
문자	빈도수
A	24
B	3
C	8
D	10
E	33
F	6
G	4
H	12
	100

고정길이 코드		
코드	문자의 비트수	전체 비트수
000	3	72
001	3	9
010	3	24
011	3	30
100	3	99
101	3	18
110	3	12
111	3	36
	24	300

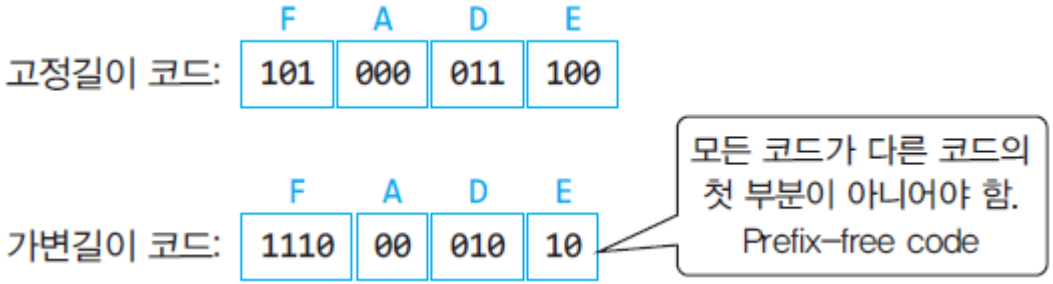
가변길이 코드		
코드	문자의 비트수	전체 비트수
00	2	48
11110	5	15
011	3	24
010	3	30
10	2	66
1110	4	24
11111	5	20
110	3	36
	27	263

[그림 8.21] 빈도수가 알려진 문자에 대한 고정길이 코드와 가변길이 코드의 비교

문자의 표현과 해독



- "FADE"의 인코딩

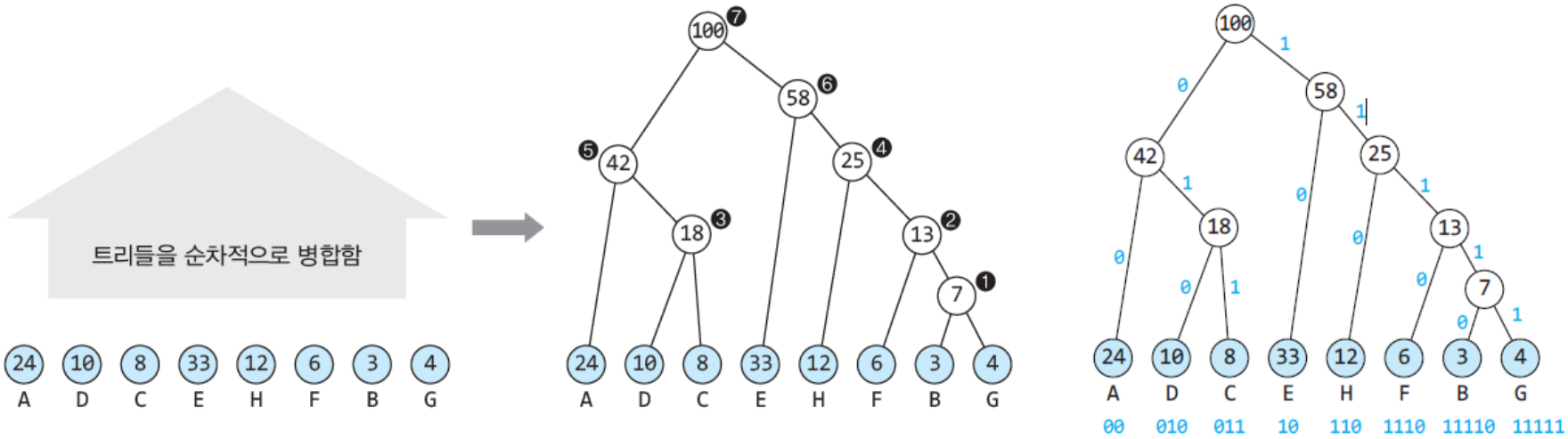


- 가변길이 코드 디코딩은?
  - 접두어 없는(prefix-free) 코드
  - 비트열을 왼쪽에서 오른쪽으로 조사하면 정확히 하나의 코드만 일치
- 빈도표 → 가변길이코드
  - 허프만 코드(Huffman codes)
    - 탐욕적 전략 사용

허프만 코드 생성

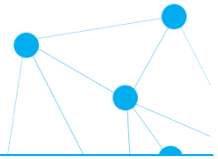
- 허프만 트리(Huffman tree)
  - 탐욕적인 해: 트리 중에서 루트의 값이 가장 두 트리 선택

- 허프만 트리 생성 과정 → 코드 부여



문자	A	B	C	D	E	F	G	H
빈도	24	3	8	10	33	6	4	12
코드	00	11110	011	010	10	1110	11111	110

알고리즘: 허프만 트리 생성



알고리즘 8.9 허프만 트리 생성 알고리즘

```
01  import heapq
02  def make_heap_tree(freq, label):
03      n = len(freq)
04      h=[]
05      for i in range(n):
06          heapq.heappush(h, (freq[i], label[i]))
07
08      for i in range(1, n) :
09          e1 = heapq.heappop(h)
10          e2 = heapq.heappop(h)
11          heapq.heappush(h, (e1[0]+e2[0], e1[1]+e2[1]))
12          print(e1, "+", e2)
13
14      print(heapq.heappop(h))
```

알고리즘 테스트 허프만 트리 알고리즘

```
label = [ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H' ]
freq  = [24,    3,    8,    10,   33,   6,    4,    12]
make_heap_tree(freq, label)
```

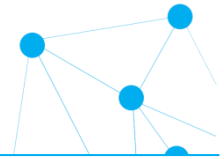
```
C:₩WINDOWS₩system32₩cmd.exe
(3, 'B') + (4, 'G')
(6, 'F') + (7, 'BG')
(8, 'C') + (10, 'D')
(12, 'H') + (13, 'FBG')
(18, 'CD') + (24, 'A')
(25, 'HFBG') + (33, 'E')
(42, 'CDA') + (58, 'HFBGE')
(100, 'CDAHFBGE')
```

트리가 합해지는 과정

최종 트리의 루트

- 시간 복잡도(우선순위 큐 사용): → $O(nlog_2n)$

실습 과제



감사합니다!