KNU KYUNGPOOK NATIONAL UNIVERSITY

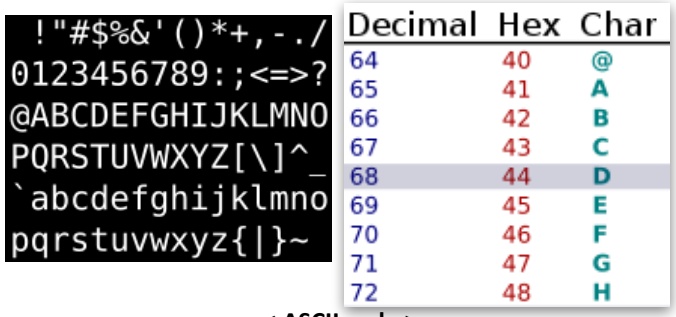
Lecture #18: Greedy Algorithm

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam

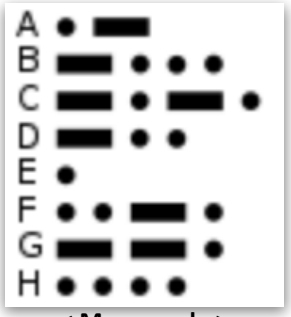
문자코드화 문제

- Coding problem
- Coding: assignment of bit strings to alphabet characters
- Codewords: bit strings assigned for characters of alphabet
- Two types of codes:
- fixed-length encoding (e.g., ASCII)
- variable-length encoding (e.g., Morse code)
- 압축할 데이터의 특성에 따라 **20~90%**까지 절약 가능하다



Decimal	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H

< ASCII code >



< Morse code >

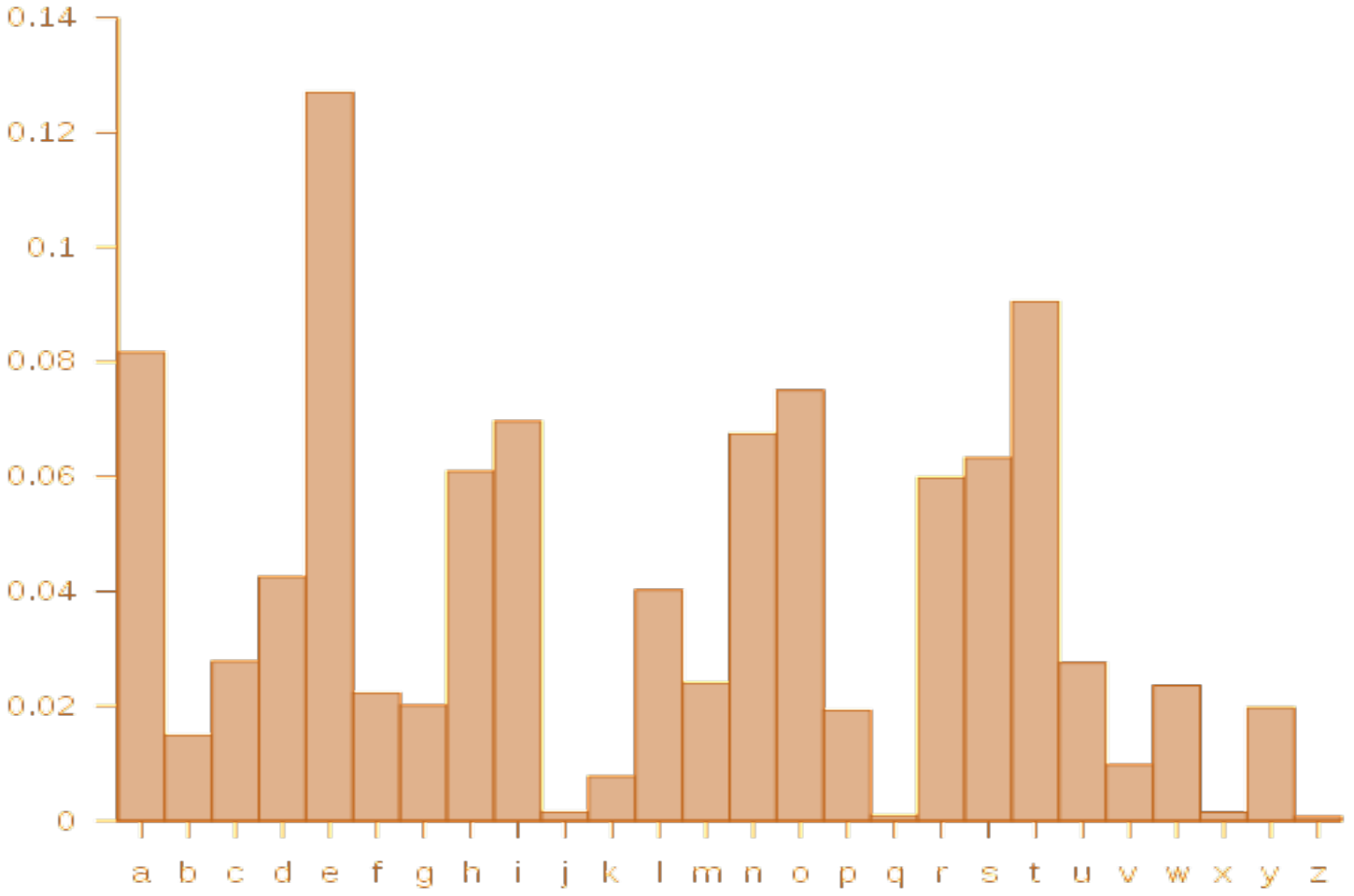
문자코드화 문제

- **Example: Character-coding problem for representing 6 characters**
- **100,000**개의 문자를 가진 데이터 파일 존재
- **Fixed-length code: 300,000 bits**
- **Variable-length code: bits (a savings of approximately 25%)**
- 가변길이를 통해 **224,000**비트로 코드화 가능
- **(15*1+13*3+12*3+16*3+9*4+5*4)*1000 = 224000**비트

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

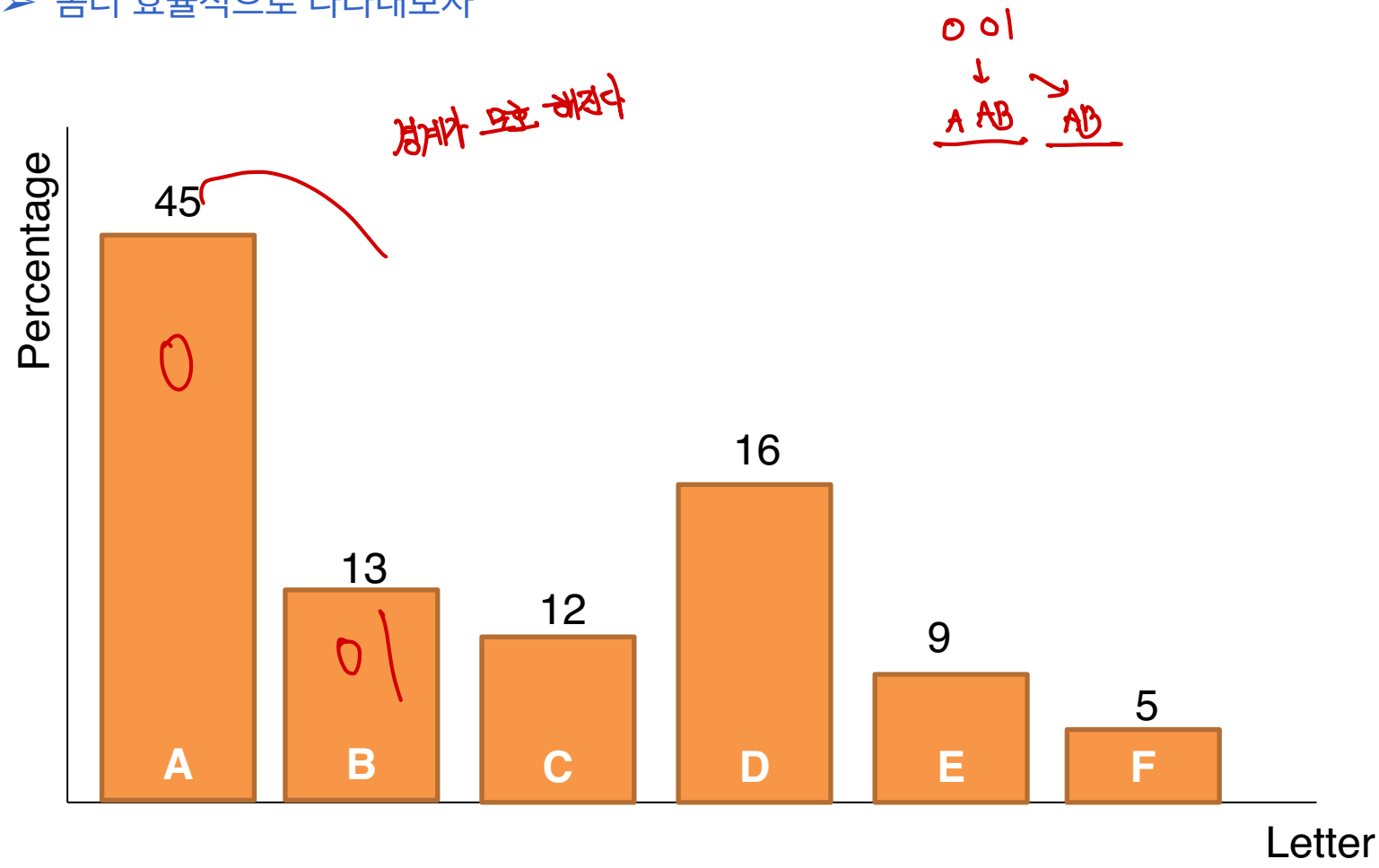
허프만 코딩 (Huffman coding)

- 알파벳의 빈도수의 분포를 확인해보자.



허프만 코딩 (Huffman coding)

- 알파벳의 빈도수의 분포를 확인해보자.
  - ➢ 좀더 효율적으로 나타내보자

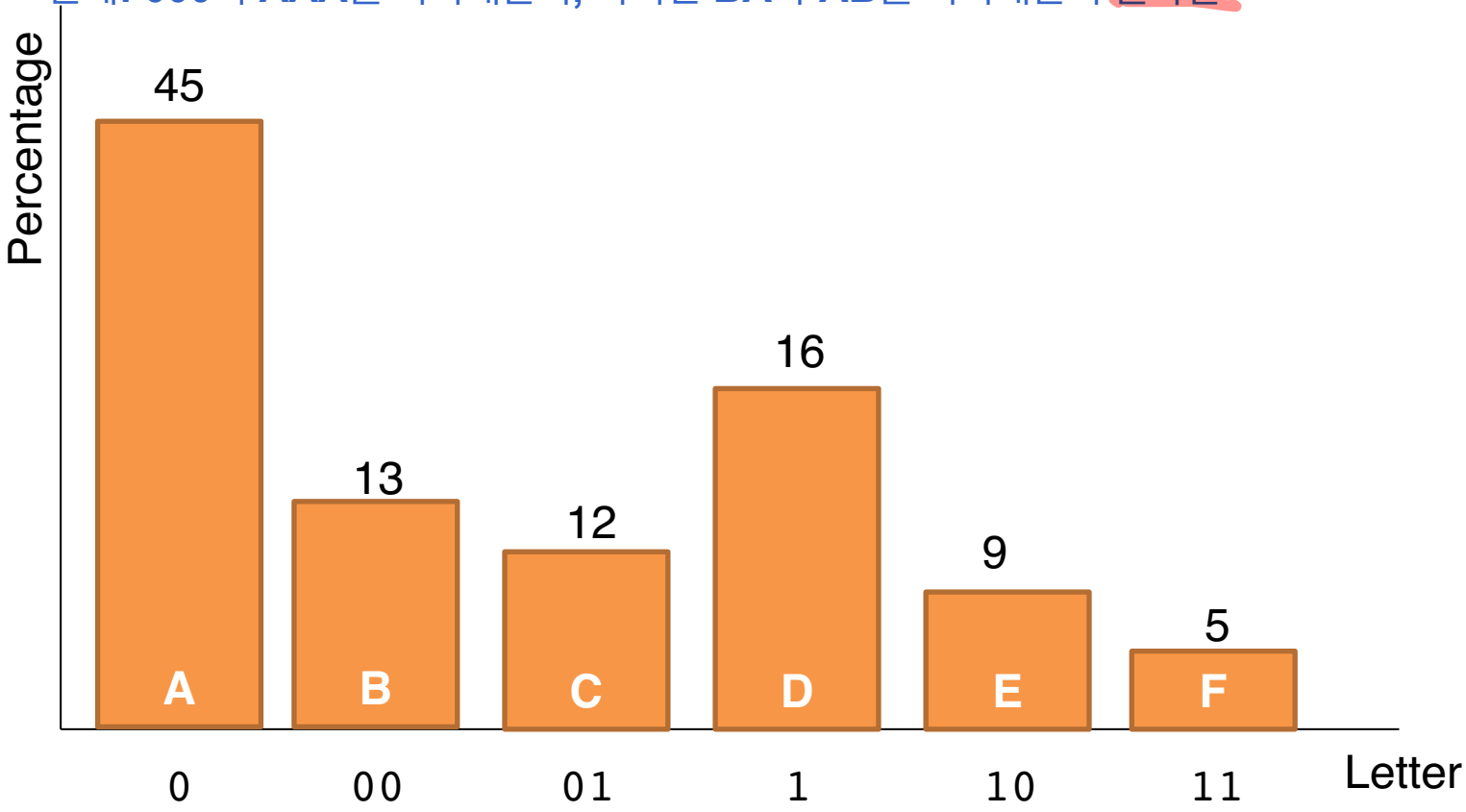


허프만 코딩 (Huffman coding)

- 방법 **#0 (ASCII** 코드와 비슷한 접근법)
  - ➢ 모든 알파벳들을 **3**자리 비트로 표현
  - ➢ 문제점:
    - **110**과 **111**은 아예 할당이 되지도 않았음
    - **A**를 표현할 때 좀 더 짧게 표현할 수 있는 방식이 있을지도…

허프만 코딩 (Huffman coding)

- 방법 **#1**
  - 모든 글자들을 **0~2**자리 비트의 이진 문자열로 할당
  - 자주 나타나는 글자들은 더 짧은 길이를 갖도록 함
  - 문제: **000**이 **AAA**를 나타내는지, 아니면 **BA**나 **AB**를 나타내는지 불확실



Prefix codes (접두사코드)

- **Prefix codes (Prefix-free codes)**
  - 어떤 문자열의 코드가 다른 문자열의 코드의 접두사**(prefix)**가 되지 않는 유형의 코드
  - 다시 말해, 어떤 코드도 다른 코드의 시작 부분이 되지 않습니다
- 예시
  - **A: 0, B: 10, C: 110, D: 111**
  - **'A'**의 코드인 **'0'**는 **'B', 'C', 'D'**의 코드의 시작 부분이 아니다
  - **'B'**의 코드인 **'10'**은 **'C'** 또는 **'D'**의 코드의 시작 부분이 아니다
  - **ABCD**을 인코딩하면**? 010110111**
  - 해독시 **ABCD**를 얻을 수 있다
  - 만약 접두사가 아니라면**?**
  - 명확히 해독할 수 없다

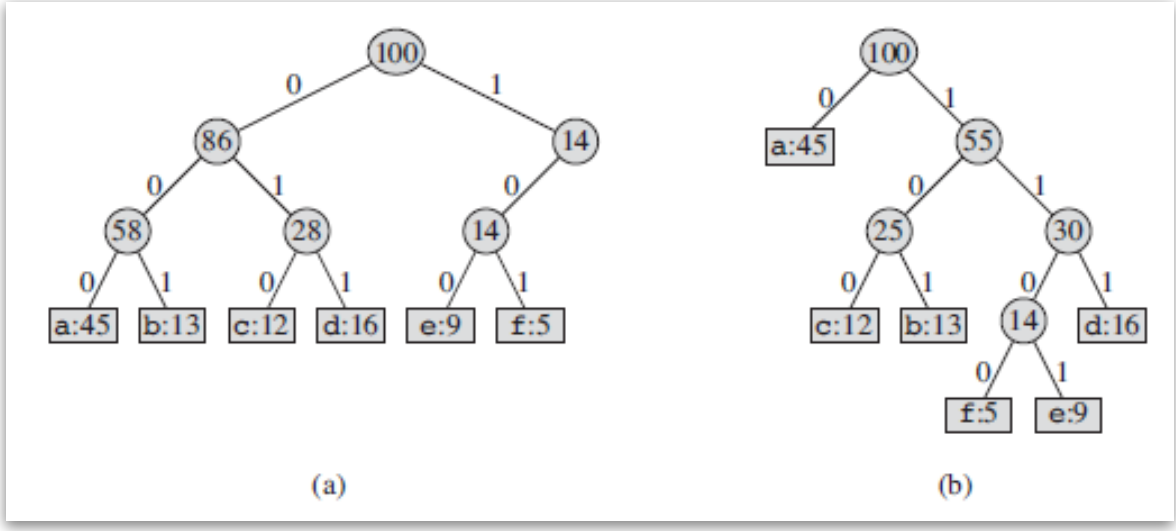
010110111

A·B·C·D

Convenient Representation for the Prefix Code

- **Prefix codes (Prefix-free codes)**
  - 해독과정에서 원래 코드단어를 쉽게 뽑아내기 위해 프리픽스 코드에 대한 편리한 표현방법이 필요
  - 이진 트리로 쉽게 표현 (이진검색트리 **x**)
  - **0:** 왼쪽 자식으로 가라
  - **1:** 오른쪽 자식으로 가라
  - **Cost** $B(t) = \sum_{c} c.freq * d_t(c)$
  - $d_t(c)$ 는 트리안에 있는 **c**의 리프노드 깊이

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



Huffman Codes

- 압축하고자 하는 문자열 : **ABBCCCDDDDEEEEEFFFFFF**
- 고정 길이 코드 : **A ~F. 6**개의 문자를 구분하기 위해 **3bit** 필요
- 가변 길이 코드 : 허프만 코드를 이용해서 나온 값.

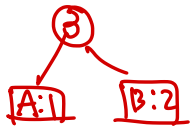
	A	B	C	D	E	F
고정 길이 코드	000	001	010	011	100	101
가변 길이 코드	1000	1001	101	00	01	11

- 압축결과
  - ➢ 고정 길이 코드 : **000001001010010010011011011011100100100100100101101101101101**
  - ➢ 가변 길이 코드 : **1000100110011011011010000000001010101011111111111111**
  - ➢ 알파벳 별 빈도수

A : 1	B : 2	C : 3	D : 4	E : 5	F : 6

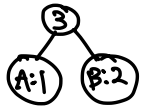
Huffman Codes

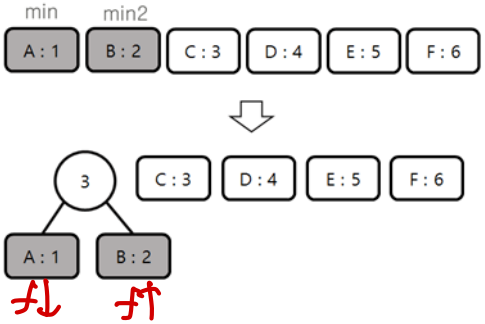
- 알파벳 별 빈도수



- 규칙
  - ➢ **Characters in leaves**
  - ➢ **Codeword is path from root to leaf**

- 빈도수를 비교하여 가장 작은 빈도수를 가진 노드와 두 번째로 작은 빈도수를 가진 것을 찾아서 두 개의 빈도수를 합친 수로 노드를 하나 만들어 줌
- 만들어준 노드의 왼쪽 자식에는 가장 작은 빈도 수의 노드를 연결하고 오른쪽 자식에는 두 번째로 작은 빈도수를 가진 노드를 연결

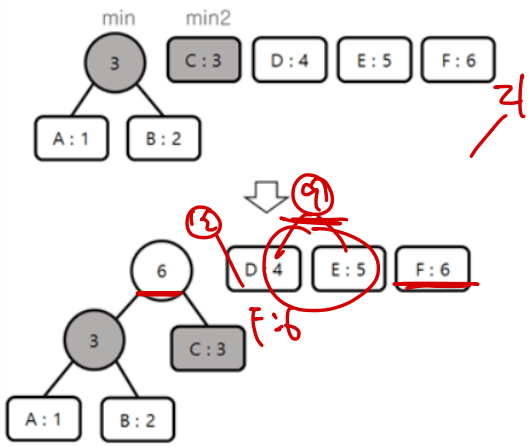


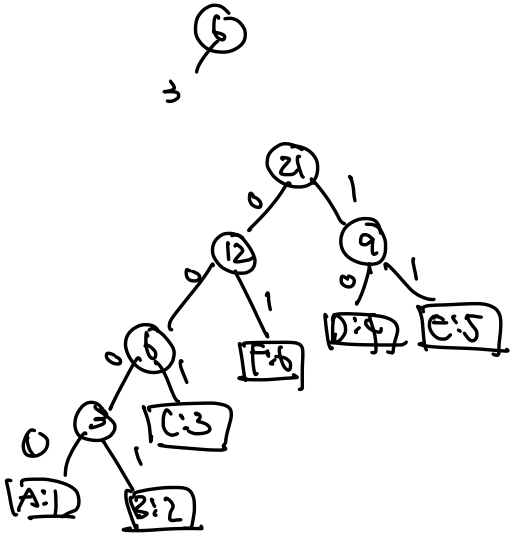


Huffman Codes

- 알파벳 별 빈도수

- 빈도수를 비교하여 가장 작은 빈도수를 가진 노드와 두 번째로 작은 빈도수를 가진 것을 찾아서 두 개의 빈도수를 합친 수로 노드를 하나 만들어 줌
- 만들어준 노드의 왼쪽 자식에는 가장 작은 빈도 수의 노드를 연결하고 오른쪽 자식에는 두 번째로 작은 빈도수를 가진 노드를 연결



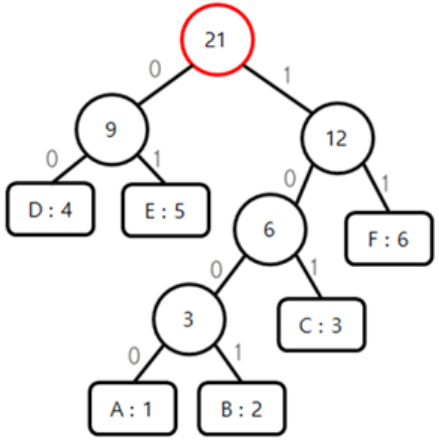


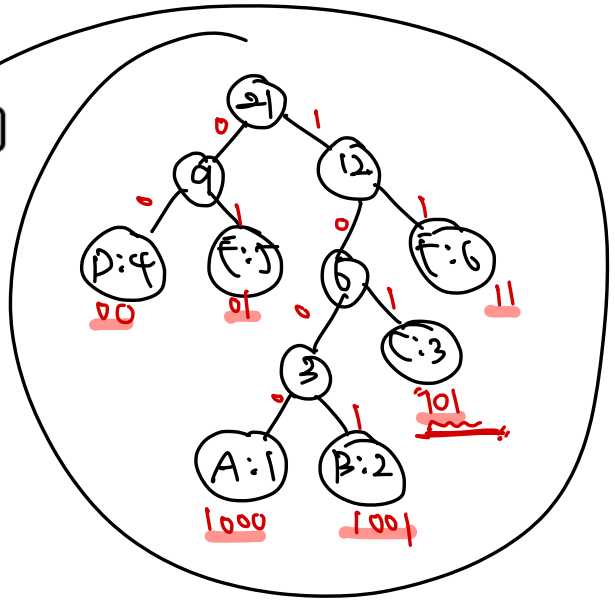
Huffman Codes

- 알파벳 별 빈도수

A : 1	B : 2	C : 3	D : 4	E : 5	F : 6

- 비교할 노드가 한 개가 남을 때까지 반복



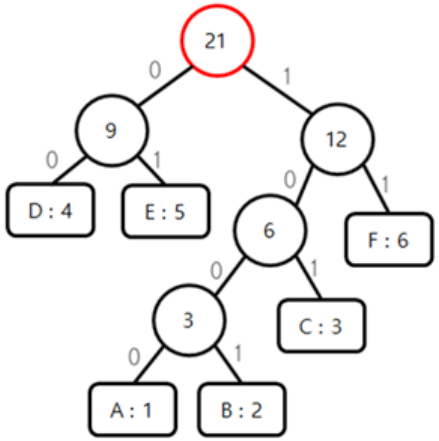


Huffman Codes

- 알파벳 별 빈도수

A : 1	B : 2	C : 3	D : 4	E : 5	F : 6

- 왼쪽 간선에는 **0,** 오른쪽 간선에는 **1** 가중치
- 트리들의 단 노드가 압축하고자 하는 문자가 되며 그 문자들을 루트로부터 탐색했을 때 지난 간선들의 가중치들의 합이 허프만 코드(가변 길이 코드)가 됨.
- **EX) A : 1-0-0-0 / E : 0-1**

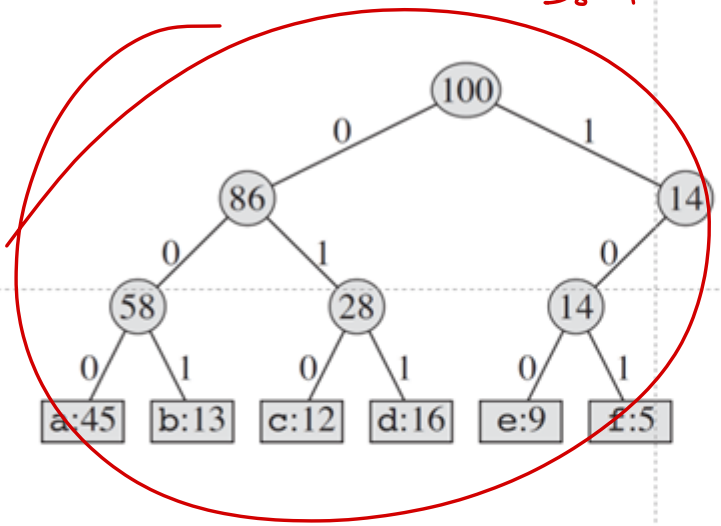


Constructing Huffman Code(교재)

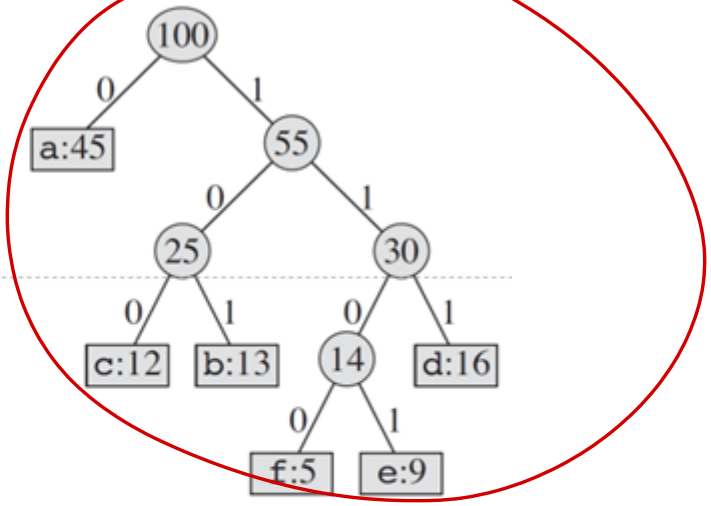
- 알파벳 별 빈도수

f:5 e:9 c:12 b:13 d:16 a:45

고정 길이



가변 길이

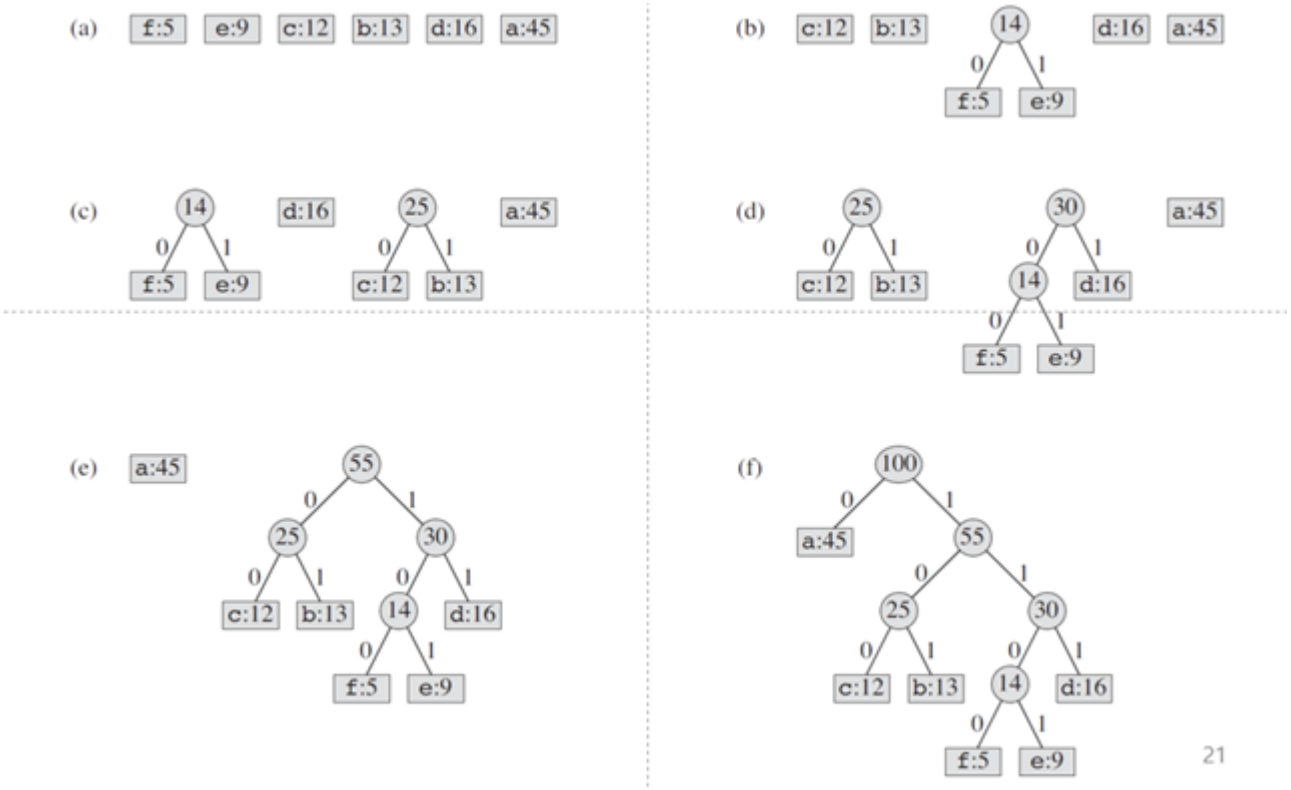


	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c$$

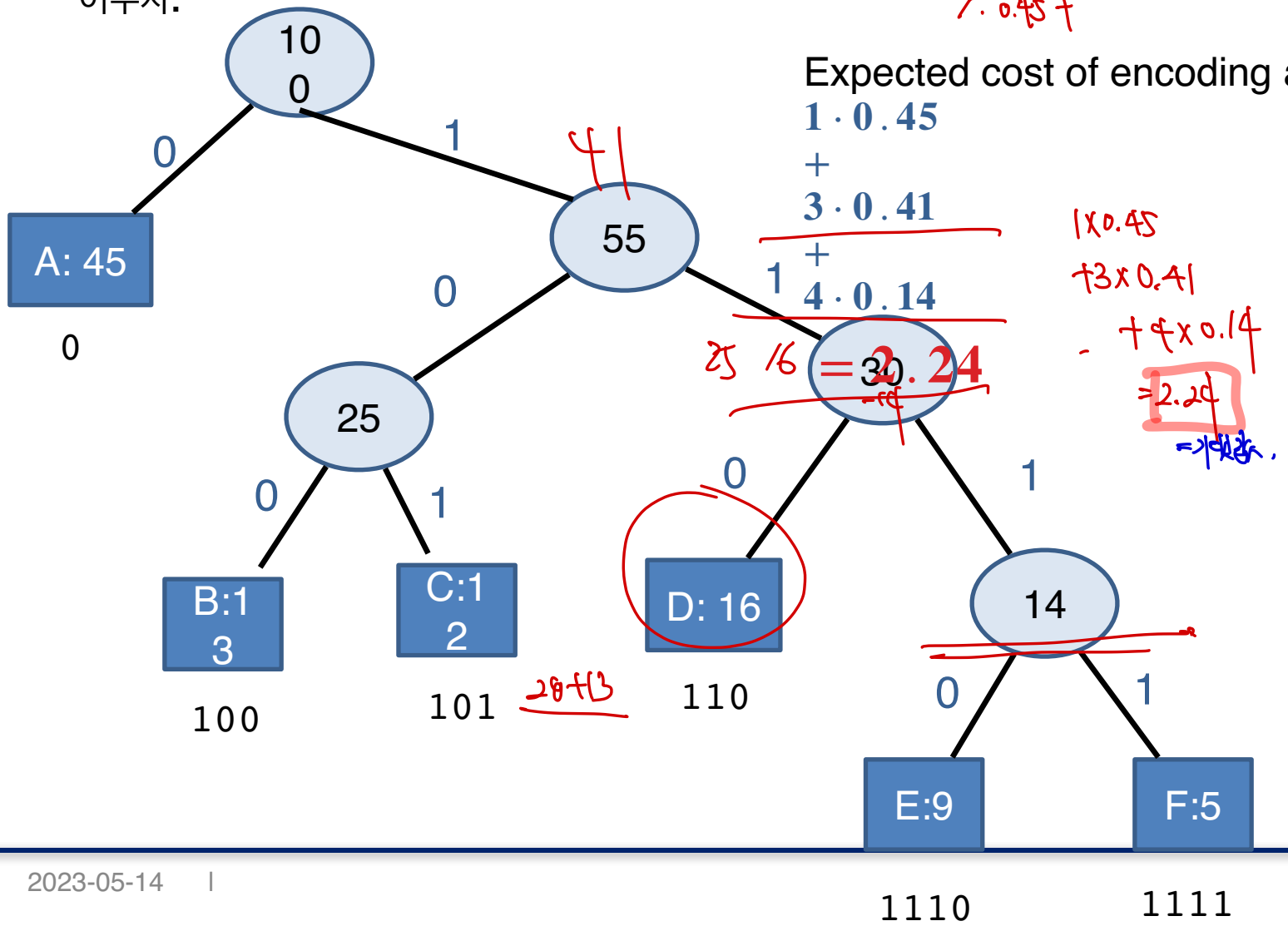
Constructing Huffman Code(교재)

- Huffman's algorithm



허프만 코딩 (Huffman coding)

- 적용 전략: 그리디 메소드를 통해서 트리를 만들되, 적은 빈도로 등장하는 문자부터 먼저 트리로 엮어주자.



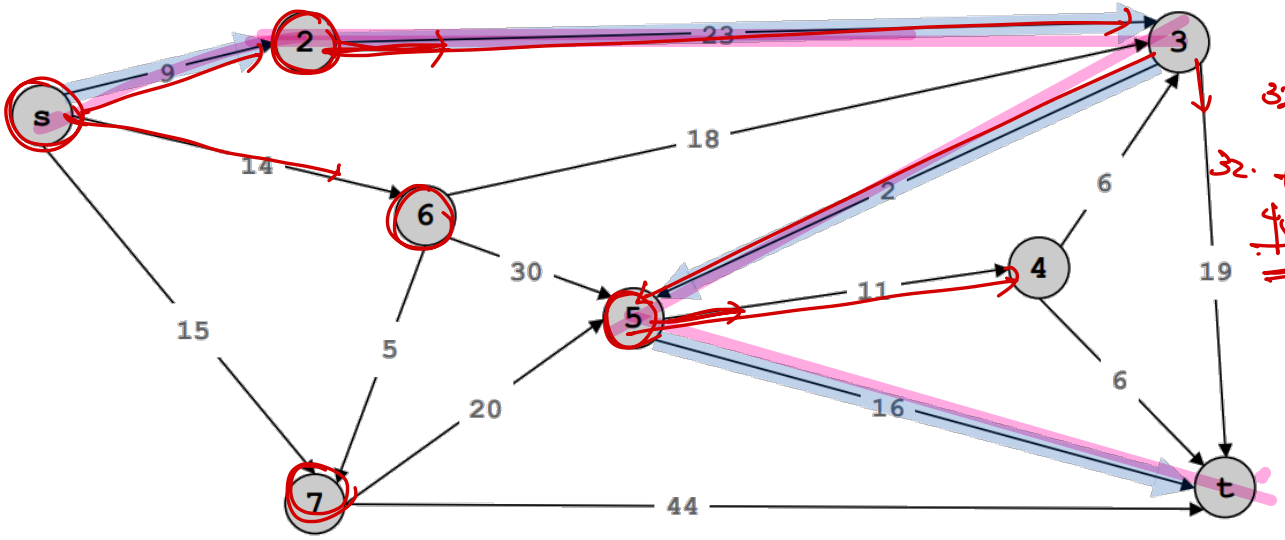
Expected cost of encoding a letter:

$$1 \cdot 0.45 + 3 \cdot 0.41 + 4 \cdot 0.14 = 2.24$$

최단 경로 구하기

- **Shortest path network**
  - ➢ 유향 그래프(directed graph) G = (V, E)를 대상
    - 시작점: **s** ~ 도착점: **t**
    - $l_e = (length\ of\ edge\ e)$

- **최단 경로 문제 (Shortest path problem)**
  - ➢ **s**부터 **t**에 이르는 최단 경로 구하기
    - 경로 상에 있는 모든 **edge**들의 **cost(length)**의 합이 최소화 되는 경로



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

최단 경로 구하기

- 다익스트라 알고리즘 **(Dijkstra's Algorithm)** Greedy

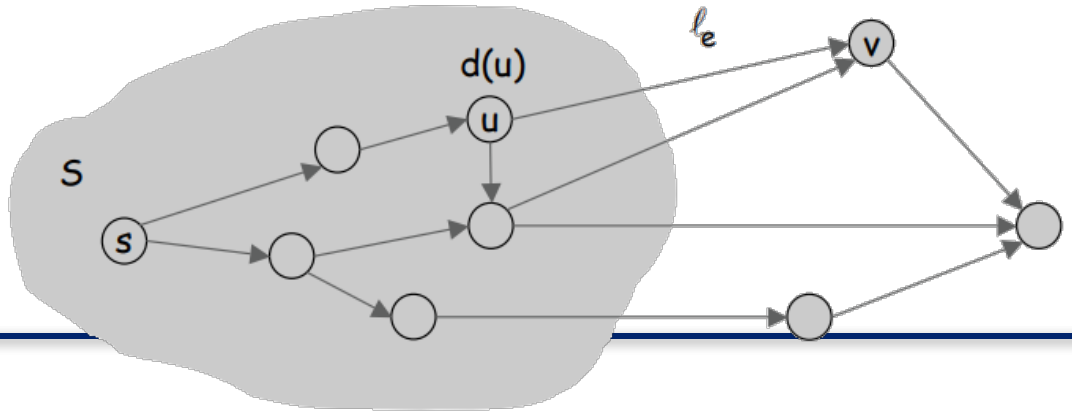
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



최단 경로 구하기

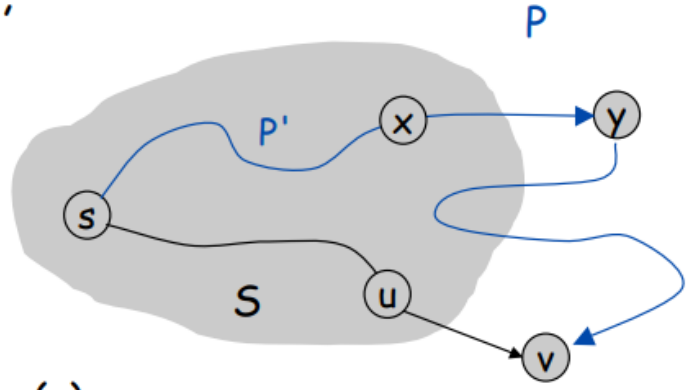
Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.

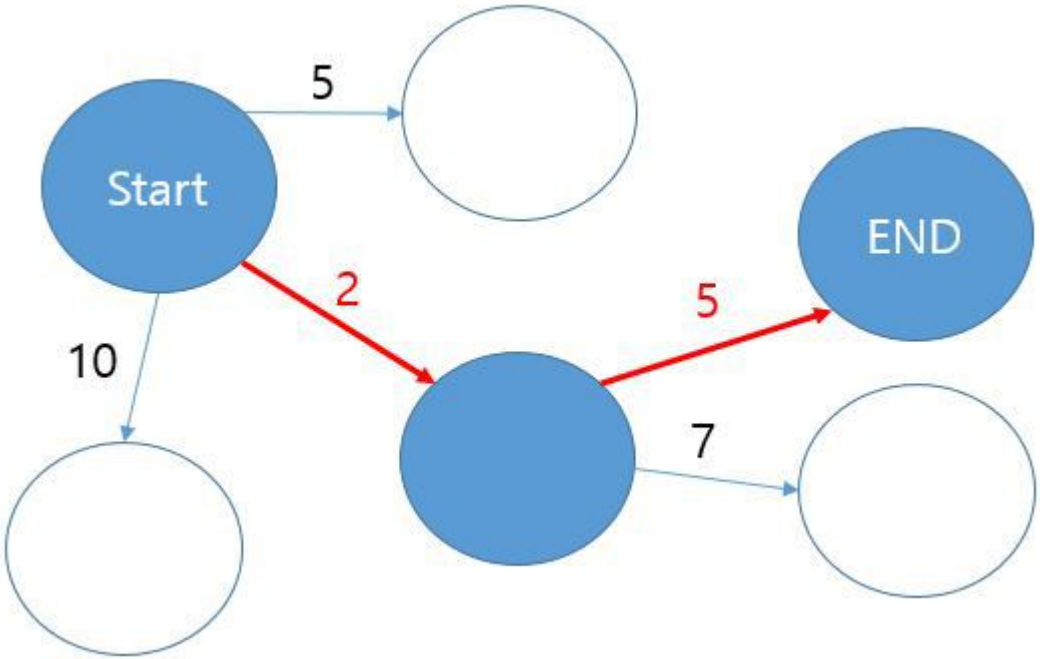


$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- nonnegative weights
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose v instead of y

최단 경로 구하기

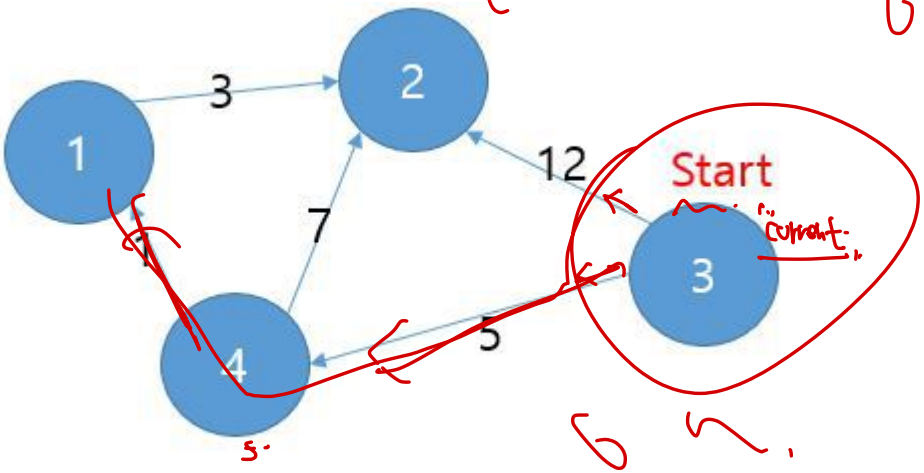
- 다익스트라 알고리즘 **(Dijkstra's Algorithm)**
  - ➢ 매 주어진 상황에서 가장 좋은 경로를 선택함으로써 만들어가는 탐색 방법



최단 경로 구하기

- 동작 과정
  - ➢ 먼저 가장 처음 각각의 노드에서 이동할 수 있는 비용은 모두 **Infinity**로 설정

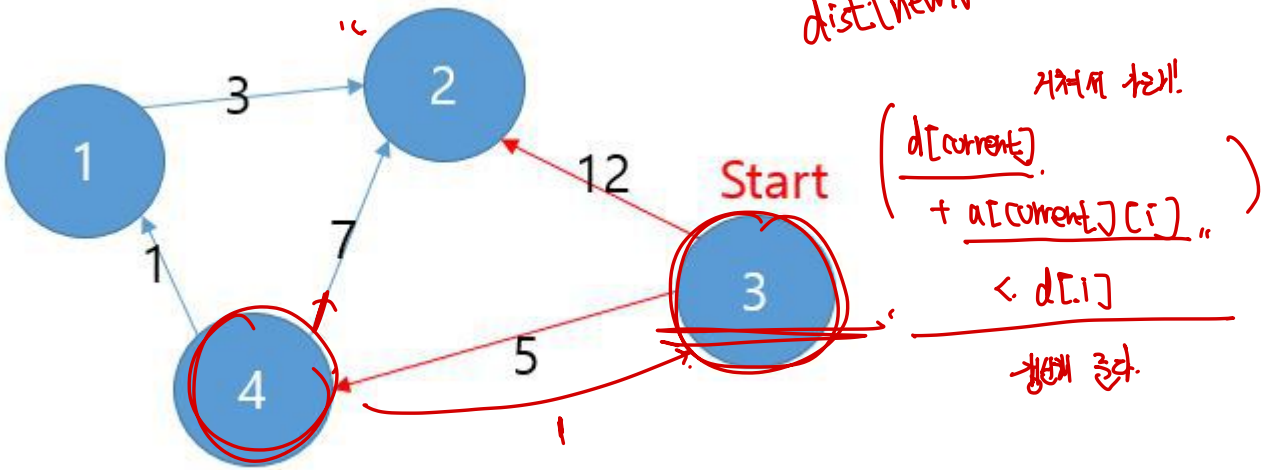
노드	1	2	3	4
비용	Infinity	Infinity	Infinity	Infinity



최단 경로 구하기

- 동작 과정
  - ➢ **3**번 노드에서 이동 가능한 **2**번노드와 **4**번노드의 비용을 각각 현재 배열에 저장된 값과 비교하여 최솟값을 선택

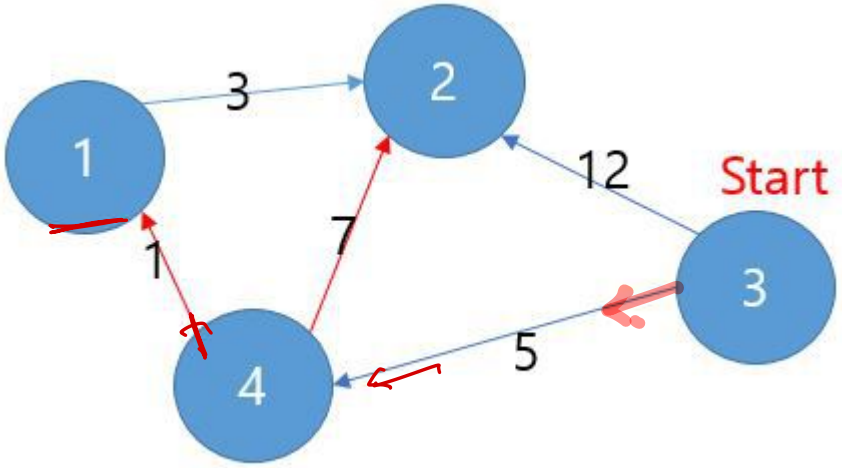
노드	1	2	3	4
비용	Infinity	12	0	5



최단 경로 구하기

- 동작 과정
  - **2**번노드와 **4**번노드 중 최소 비용이 걸리는 **4**번노드로 이동 후 같은 작업을 반복
  - **min(arr[1],arr[4]+1)**과 **min(arr[2],arr[4]+7)**을 통해 배열을 수정

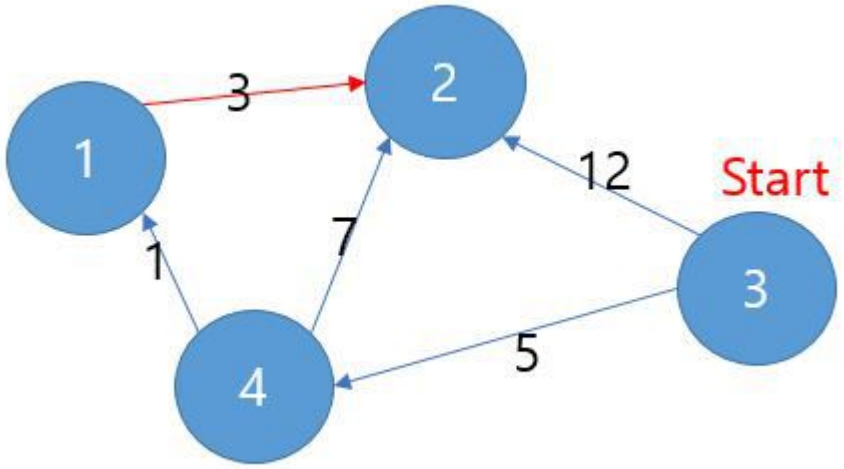
노드	1	2	3	4
비용	6	12	0	5



최단 경로 구하기

- 동작 과정
  - ➢ **2**번 노드로 이동하는 비용은 **12**로 같기 때문에 변화가 없고 **1**번노드로 가는 비용은 무한대보다 **6**이 더 최솟값이기 때문에 위 배열 값을 수정
  - ➢ **1**번노드로 이동 후 작업을 반복. **min(arr[1]+3,arr[2])**를 비교

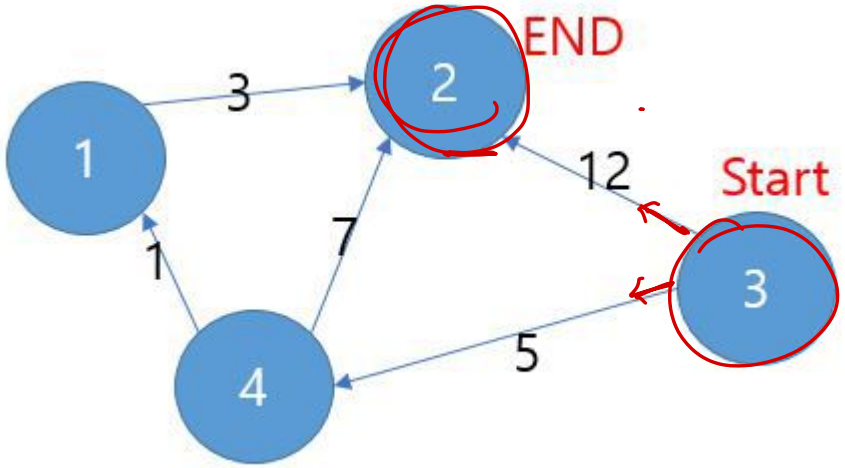
노드	1	2	3	4
비용	6	9	0	5



최단 경로 구하기

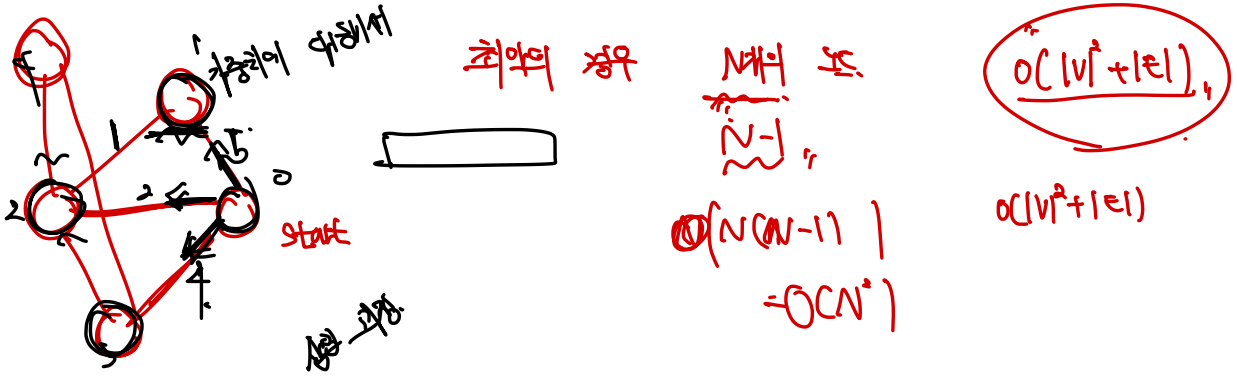
- 동작 과정
  - ➢ **2**번노드에서 이동할 수 없는 화살표가 없기 때문에 작업이 일어나지 않는다
  - ➢ 모든 노드를 방문하면 다익스트라 알고리즘이 끝나기 때문에 위 정리된 최소비용배열을 구함으로써 다익스트라 알고리즘이 종료

노드	1	2	3	4
비용	6	9	0	5

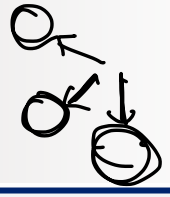


다익스트라 time complexity

- 먼저 최악의 경우 **N**개의 노드가 있을 때 첫 번 째 노드에서 **N-1**번의 연산이 일어나고 **2**번째 노드에서도 **N-1**번의 연산이 일어납니다.**(**화살표가 양방향으로 있는 경우 고려**)**
  - ➢ 즉 결과적으로 **N*(N-1)**의 연산이 일어나고 최악의 경우 **O(N^2)**의 효율성을 가지게 됩니다.
- 우선순위 큐를 이용했다면 효율성을 조금 더 높일 수 있음
  - ➢ 다익스트라 알고리즘의 경우 매 순간 최솟값을 비교를 통해 구하고 최솟값을 바탕으로 갱신하며 진행되기 때문에 우선순위 큐를 이용하여 구현하게 되면
  - ➢ 최솟값을 선택하는 비용이 상수 비용이 되기 때문에 효율성을 높일 수 있습니다.



다익스트라 time complexity



- 결과적으로 노드가 **N**개 있고 간선의 갯수가 **E**개 있다면 우선순위 큐 높이가 **logN**이 되고 이 때 우선순위큐에 새로운 데이터를 삽입 및 삭제하는 과정은 최악의 경우 **E**개수 만큼진행되므로 **O(ElogN)**의 효율성을 가지게 됩니다.
  - ➢ 단, 이 경우 역시 최악의 경우 **E**의 개수는 **O(N^2)**과 같으므로 최악의 경우 **O(N^2logN)**의 극악 효율성을 가짐
- 마지막으로 다익스트라의 가장 큰 단점은 비용이 양수가 아닌 음수의 경우 무한 루프에 빠지는 등 결과 값을 제대로 구할 수 없는 치명적인 단점