Greedy Algorithms

(CLRS Chapter 16)

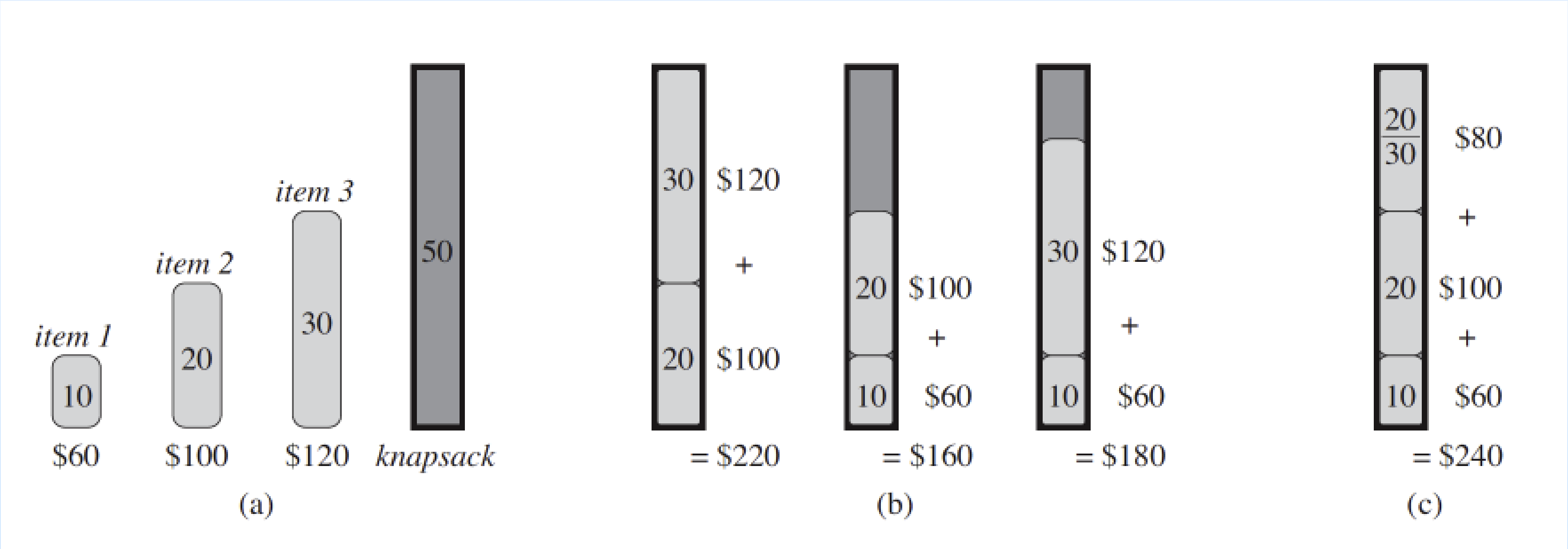
김동진
(NHN NEXT)

Knapsack problem

- 상황 설명
  - 보물섬에서 보물을 찾았다.
  - 가지고 있는 배낭에 보물을 넣어서 나가려고 한다.
  - 배낭에는 W Kg까지 넣을 수 있다. 그 이상 넣으면 찢어져서 사용할 수가 없다.
  - 찾은 보물은 개수는 n개이다.
  - 보물 i의 가치는 v_i 원이다.
  - 보물 i의 무게는 w_i Kg이다.
- Fractional knapsack problem
  - 물건을 분할해서 넣을 수 있다. 예를 들어, 금 가루인 경우 일정 무게만 넣을 수 있다.
  - 가치가 최대가 되도록 knapsack에 채우는 방법을 제시하시오.

Fractional Knapsack Problem (1)

- 아이디어
  - 가치/무게 비율이 큰 것부터 채우자.
  - 예제
    - b)는 0/1 knapsack
    - c)는 fractional knapsack: 최대 가치 240



Fractional Knapsack Problem (2)

- 연습 문제 16.2-1
  - Prove that the fractional knapsack problem has the greedy-choice property.
  - 증명
    - 모순에 의한 증명 방법 사용
    - Greedy-choice로 구한 결과를 S_g라 하자.
    - S_g에는 가치/무게 비율이 큰 아이템부터 차례로 채워져 있다.
    - S_g와 다른 optimal solution 결과가 S_s가 있다고 가정하자.
    - 그러면 S_g에는 없지만 S_s에는 있는 item s가 차지하고 있는 무게가 있다. 이 무게를 w_s라고 하자.
    - 또한, S_g에는 있지만 S_s에는 없는 item g이 차지하고 있는 무게가 있다. 이 무게를 w_g라고 하자.
    - 아이템 s의 가치/무게는 아이템 g의 가치/무게보다 작다.
    - 따라서, w_s를 꺼내고 w_g를 채우면 optimal solution S_s보다 가치가 더 커지게 된다.
    - 따라서, optimal solution 결과가 S_s가 optimal solution이라는 가정에 위배된다.
    - 모순이므로 S_g와 다른 optimal solution 결과가 S_s는 존재하지 않는다.

Fractional Knapsack Problem (3)

- 알고리즘
  - 각 아이템의 가치/무게 비율을 구한다.
  - 가치/무게 내림차순으로 정렬한다.
  - 정렬된 순서로 아래 사항을 진행한다.
    - 아이템을 채울 수 있는 만큼 채운다.
    - Knapsack 용량을 꽉 채웠거나 채울 아이템이 없을 때까지 반복한다.
- 손코딩

Fractional Knapsack Problem (4)

- Fractional knapsack code

```
flaot   fracKnapsack(item_t *items, int num, int capa)
{
    flaot       value = 0;

    for(int id = 0; id < num; id++)
        items[id].density = items[id].value / items[id].weight;

    mySort(items, num);

    for(int id = 0; id < num; id++) {
        if(capa > items[id].weight) {
            value += items[id].value;
            capa -= items[id].weight;
        } else {
            value += items[id].density * capa;
            break;
        }
    }

    return value;
}
```

Fractional Knapsack Problem (5)

- 보조 함수

```
typedef struct item {
    int     weight;
    int     value;
    float   density;
} item_t;

bool myfunction(item_t a, item_t b)
{
    return a.density > b.density;
}

void    mySort(item_t *items, int num)
{
    vector<item_t> itemVector (items, items + num);

    sort(itemVector.begin(), itemVector.end(), myfunction);

    for(unsigned int id = 0; id < num; id++) {
        items[id].weight = itemVector[id].weight;
        items[id].value = itemVector[id].value;
        items[id].density = itemVector[id].density;
    }
}
```

Greedy Strategy의 구성 요소

- Optimal substructure
  - 주어진 문제의 optimal solution은 sub-problem의 optimal solution들을 포함하는 구조
- Greedy-choice property
  - 전체적인 optimal solution은 locally optimal (greedy) choice에 의해서 구성될 수 있다.
  - 매 step에서 locally optimal choice를 할 경우 globally optimal solution을 구한다는 것을 증명해야만 한다.

Greedy verse Dynamic Programming

- 공통점
  - Optimal substructure를 사용
- 차이점
  - Dynamic programming은 매 step에서 여러 개의 sub-problem 해결
  - Greedy strategy는 매 step에서 한 개의 sub-problem만 처리
- 차이점 비교 예제
  - 0/1 knapsack problem
  - Fractional knapsack problem

An Activity-Selection 문제 (1)

- 문제 설명
  - 1개의 resource가 주어지고 resource를 사용하려고 하는 n개의 activity(일)이 있다.
    - 예) 강의실 1개와 n개의 강의 일정
  - 각각의 activity마다 시작 시점과 종료 시점이 주어져 있다.
    - [시작 시점, 끝나는 시점)
  - 1개의 resource를 두 개의 activity가 동시에 사용할 수 없다.
  - 끝나는 시점의 오름차순으로 activities가 주어져 있다.
  - 예제
    - s_i: activity a_i의 시작 시점
    - f_i: activity a_i의 종료 시점

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- 문제
  - Resource를 사용하는 activity의 개수를 최대로 하는 방법을 찾아라.
  - 위 예제
    - Resource 사용 가능한 경우: $\{a_3, a_9, a_{11}\}$
    - 개수가 최대인 경우: $\{a_1, a_4, a_8, a_{11}\}$, $\{a_2, a_4, a_9, a_{11}\}$

An Activity-Selection 문제 (2)

- Recursive Solution 아이디어
  - 주어진 activities를 종료 시간 증가순으로 정렬
  - activity a_i에 대해서
    - a_i 선택 (a_i는 resource 사용 가능. 단, 주어진 시간 범위 내에 a_i의 시작 시점과 종료 시점이 포함되어야 함)
    - a_i 시작 시점 이전에 사용 가능한 activity 최대 개수
    - a_i 종료 시점 이후에 사용 가능한 activity 최대 개수
  - Resource를 사용하는 activity가 최대가 되는 a_i 선택

```c
typedef struct act {
        int startTime;
        int endTime;
} act_t;
```

```c
int aspRecursion(act_t *acts, int sId, int eId, int startTime, int endTime)
{
    int     maxNumOfAct = 0;

    if(sId > eId) return 0;
    if(sId == eId) {
        if(acts[sId].startTime >= startTime && acts[sId].endTime <= endTime)
            return 1;
        else return 0;
    }

    for(int id = sId; id <= eId; id++) {
        maxNumOfAct = max(maxNumOfAct,
                            aspRecursion(acts, sId, id - 1, startTime, acts[id].startTime)
                          + aspRecursion(acts, id + 1, eId, acts[id].endTime, endTime)
                          + aspRecursion(acts, id, id, startTime, endTime));
    }

    return maxNumOfAct;
}
```

An Activity-Selection 문제 (3)

- 최적화 아이디어
  - 만약, 각 activity를 선택하는 방안 중 최대값을 찾는 대신에 한번에 최대값이 되는 activity를 구하는 방법이 있다면?
    - Greedy algorithms

- Greedy algorithm 적용을 위한 아이디어
  - 주어진 activity들은 종료 시점 기준으로 오름차순으로 정렬되어 있음
  - 만약, 첫 번째 종료되는 activity를 무조건 선택한다면 어떻게 될까?

An Activity-Selection 문제 (4)

- ***Theorem 16.1***
  Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

  Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

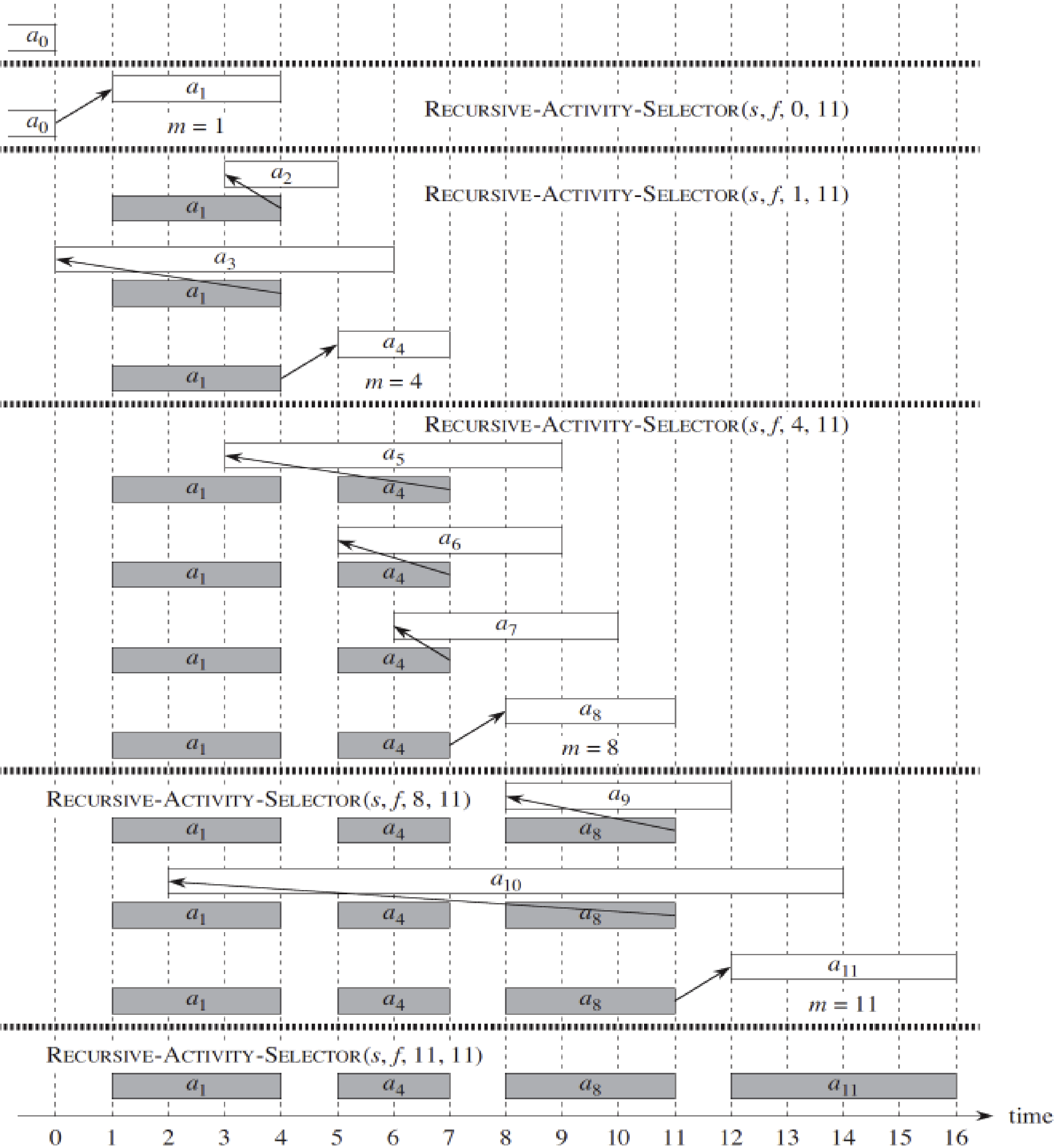
An Activity-Selection 문제 (5)

- Theorem 16.1 증명
  - Activity 개수가 최대가 되는 결과 set A_k를 구했다고 가정.
  - 종료 시점이 제일 빠른 activity a_m이 A_k에 포함되는지 여부 파악하자.
  - A_k에서 종료 시점이 제일 빠른 activity를 a_j라 하자.
  - $a_m = a_j$인 경우 포함됨
  - a_m이 a_j가 아닌 경우
    - set A_k에서 a_j를 빼고 a_m을 추가해도 시간이 겹치는 activity가 없고 set A_k의 크기도 변하지 않음.
  - 따라서, activity a_m는 activity 개수가 최대가 되는 결과에 포함될 수 있음.

An Activity-Selection 문제 (6)

- Recursive greedy algorithm 아이디어
  - 종료 시점 기준으로 activity 정렬
    - 주어진 문제에서는 이미 정렬되어 있음
  - 종료 시점이 가장 빠른 activity 추가
  - 추가된 activity의 종료 시점 이후에 나머지 activity를 할당
    - 작아진 문제에서 activity를 채울 수 있는 가장 빠른 시점은 추가된 activity의 종료 시점임
    - 따라서, 추가된 activity의 직후 activity는 추가된 activity의 종료 시점보다 시작 시점이 같거나 뒤인 activity 중 종료 시점이 가장 빠른 activity임
  - 위 과정을 반복

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



An Activity-Selection 문제 (7)

- Recursive greedy algorithm
  - Input
    - Activity 배열
    - 배열 크기
    - 고려할 첫 번째 activity id
    - 직전 추가된 activity의 종료 시점
  - Output
    - 겹치지 않고 수행할 수 있는 최대 개수
  - 종료 조건
    - 처리할 activity가 없으면 종료
  - Inductive step
    - 처리 가능한 첫 번째 activity 추출
    - 나머지 activity를 대상으로 추출된 activity의 종료 시점 기준을 사용하는 sub-problem 해결

An Activity-Selection 문제 (8)

- 손코딩

- Recursive greedy algorithm code

```
int aspGreedyRecursion(act_t *acts, int num, int firstId, int endTimeOfLastAct)
{
    if(firstId >= num) return 0;

    int nextId = firstId;

    while(nextId < num && acts[nextId].startTime < endTimeOfLastAct) nextId++;

    if(nextId == num)
        return 0;
    else
        return (1 + aspGreedyRecursion(acts, num, nextId + 1, acts[nextId].endTime));
}
```

- 호출 코드
  - asp(acts, num, 0, 0);

An Activity-Selection 문제 (9)

- Iterative greedy algorithm
  - Input
    - Activity 배열
    - 배열 크기
  - Output
    - 겹치지 않고 수행할 수 있는 최대 개수

- 손코딩

An Activity-Selection 문제 (10)

- Iterative greedy algorithm code

```c
int aspGreedyIter(act_t *acts, int num)
{
    int maxNum, lastInsertedActId;

    if(num <= 0) return 0;

    maxNum = 1;
    lastInsertedActId = 0;
    for(int id = 1; id < num; id++) {
        if(acts[id].startTime >= acts[lastInsertedActId].endTime) {
            maxNum++;
            lastInsertedActId = id;
        }
    }

    return maxNum;
}
```

HW.C5

- HW.C5
  - Activity selection problem의 greedy algorithm 코딩

- 마감: 다음 수업 시작 전

HW.P

- 문제 풀이 과제
  - HW.P1: 연습 문제 16.1-2
  - HW.P2: 연습 문제 16.2-3
- 마감: 다음 수업 시작 전

데이터 압축 (1)

- 주어진 상황
  - 문자열로 이루어진 1MB file을 인터넷으로 전송하려고 한다.
  - 데이터 누락없이 용량을 최소화로 줄여서 전달하고 싶다.

- 문제
  - 데이터 누락없이 용량을 최소화하는 방법은?

- 아이디어
  - 문자는 1byte(ascii code) 혹은 2~3bytes(unicode)로 표현된다.
    - codeword라고 함.
  - File에 있는 문자의 종류가 6개라면 0 ~ 7로 각 문자를 대응 가능
    - 0~7는 3bits로 표현 가능

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

    - 이러한 방법을 fixed-length code라고 함.

데이터 압축 (2)

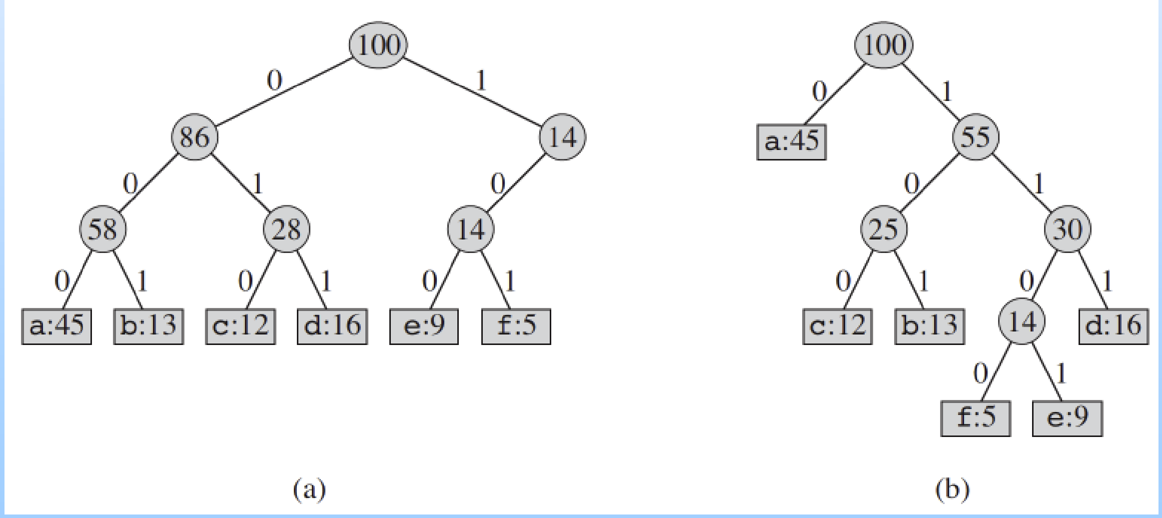
- 개선 아이디어
  - 각 문자별 사용빈도를 활용
  - 많이 사용되는 문자는 짧은 bit-stream으로 표현하자.
  - 예제) 압축할 문서 정보

	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

    - 압축 전 총 길이: (45 + 13 + 12 + 16 + 9 + 5) x 1,000 = 100 x 1, 000 = 800,000 bits
    - 압축 후 총 길이: (45x1 + 13x3 + 12x3 + 16x3 + 9x4 + 5x4) = 224,000 bits
    - 압축률 = 100 - 224 / 800 * 100 = 72%
  - Variable-length code라고 한다.
- 질문
  - Variable-length code를 만들 때 주의해야 할 사항은 없을까?

Prefix Code (1)

- 개념
  - Codeword에 있는 어떤 문자도 다른 codeword의 prefix(앞 글자)가 아니어야 한다.
    - 예제) 0, 101, 100, 111, 1101, 1100
    - 001011101 → 0, 0, 101, 1101로 분할 가능
- Tree 표현
  - 각 문자는 root로부터 leaf까지의 경로상의 값에 대응



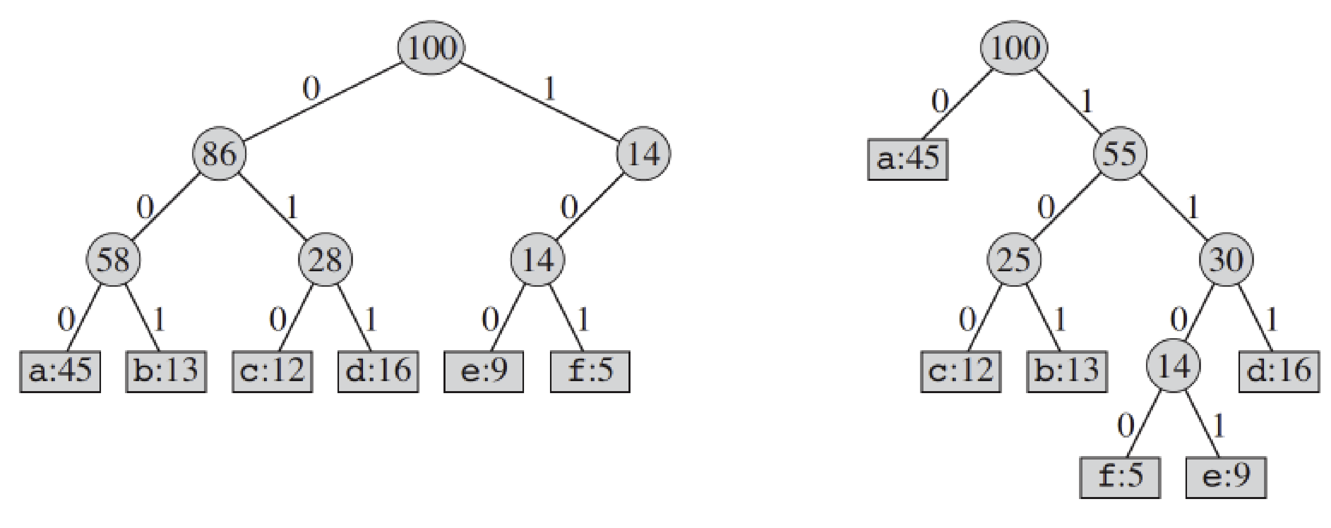
  - a) 000, 001, 010, 011, 100, 101
  - b) 0, 100, 101, 1100, 1101, 111

Prefix Codes (2)

- Prefix code로 저장한 결과의 길이
  - "각 단어의 빈도수" x "tree에서 단어의 depth"의 총합

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

    - c.freq : 단어의 빈도
    - $d_T(c)$: 단어 c의 tree 내 depth
- 예제



(a) (b)

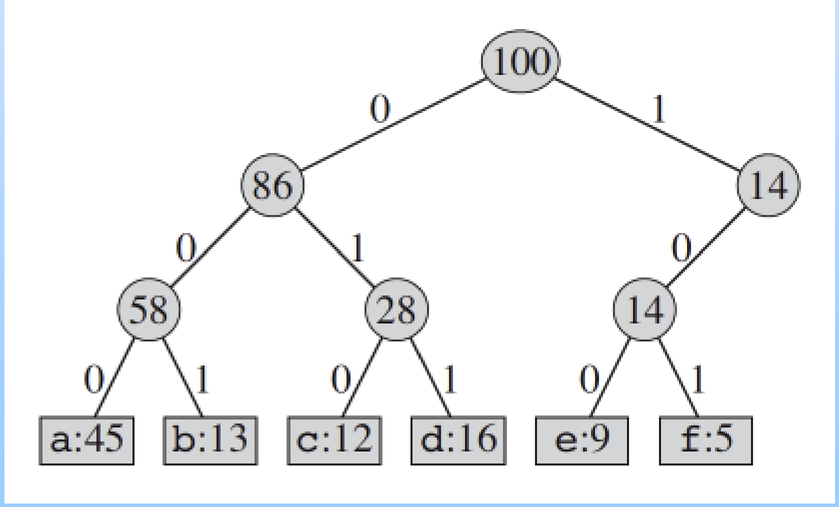
    - a) 3x45 + 3x13 + 3x12 + 3x16 + 3x9 + 3x5 = 303
    - b) 1x45 + 3x12 + 3x13 + 5x4 + 4x9 + 3x16 = 224

Prefix Codes (3)

- Optimal prefix code
  - Prefix code 중 압축률이 제일 높은 방법
  - 문서 길이를 제일 작게 만드는 codes

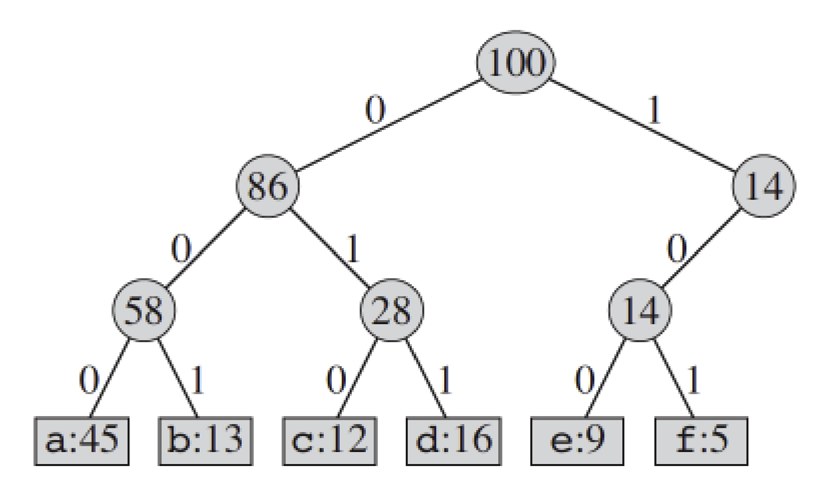
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

  - Optimal prefix code를 tree로 표현하면 full binary tree
    - 모든 internal node가 두 개의 child를 갖는 트리
    - 아래 tree는 optimal이 아님



Prefix Codes (4)

- 연습문제 16.3-2
  - Full binary tree가 아닌 경우는 optimal prefix code가 될 수 없음을 보여라.



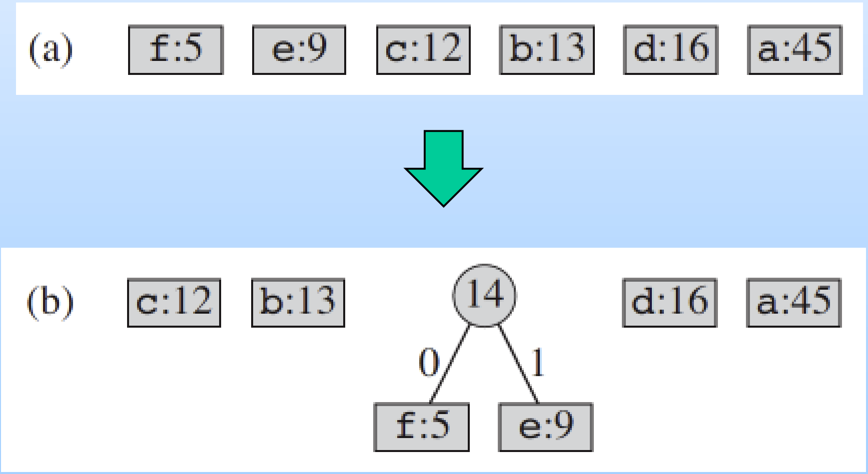
- 증명
  - Full binary가 아닌 위의 tree가 prefix code가 optimal prefix code라고 가정하자.
  - Child가 하나인 internal node를 제거해도 prefix code가 유지된다.
  - 더 작은 값의 prefix tree가 존재하므로 위의 tree가 optimal prefix code라는 가정에 위배. 모순 발생
  - 따라서, full binary tree가 아닌 경우는 optimal prefix code가 될 수 없다.

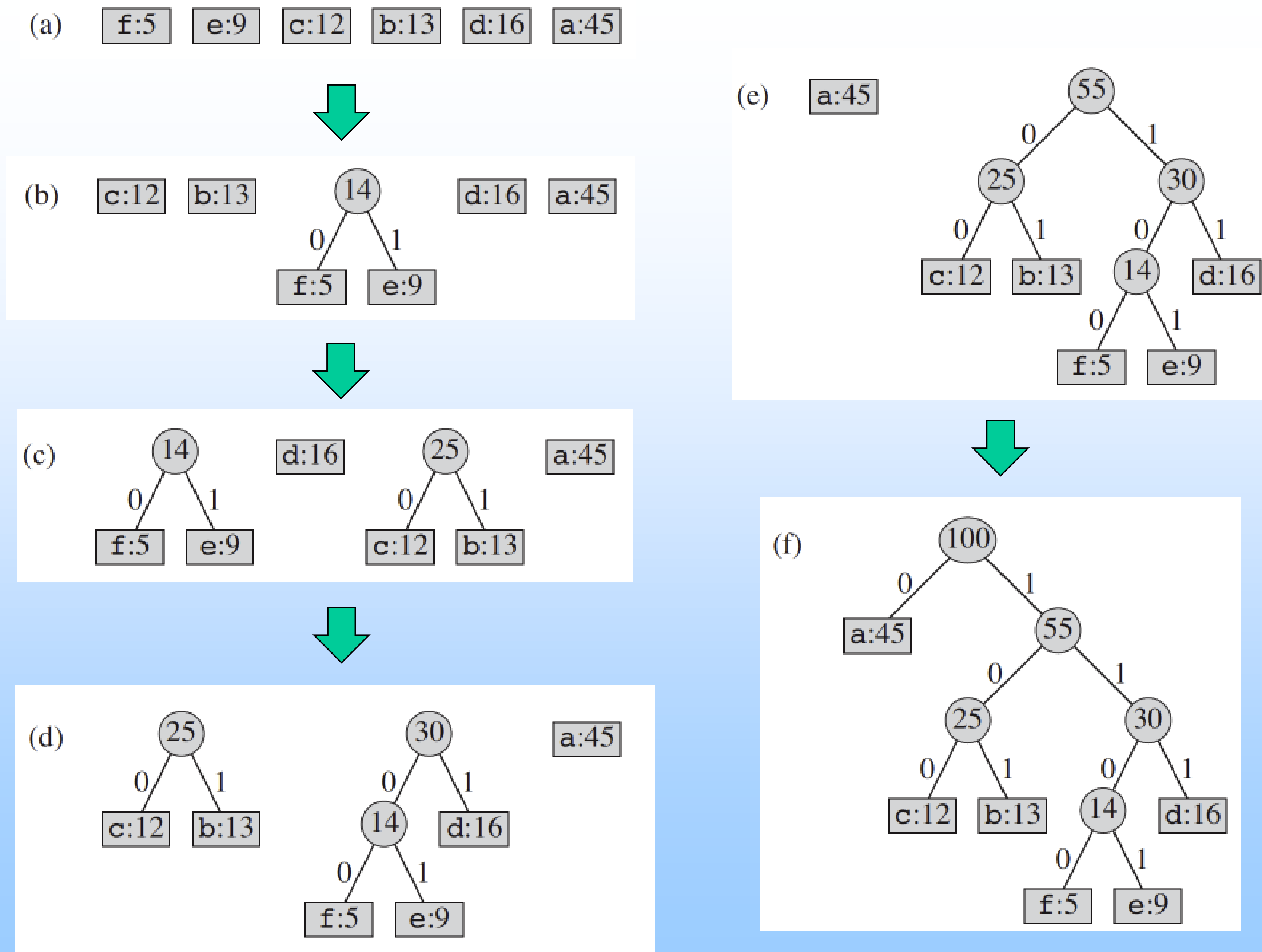
Huffman codes (1)

- Optimal prefix code
- Huffman이 optimal prefix code를 만드는 greedy algorithm 개발
- 효율
  - 데이터의 특성에 따라서 20% ~ 90% 압축
- Lossless 압축(compression) 방법
  - MP3, MPEG4 등 audio/video 압축 시의 중간 단계에서도 사용됨
- Lossy vs Lossless
  - Lossy
    - 정보 손실이 있는 압축 방법
    - MP3, MPEG4 등 Audio/video 압축에서 사람이 느끼지 못하는 수준에서 데이터를 제거하는 방법
  - Lossless
    - 정보 손실이 없는 압축 방법
    - 원래 데이터를 그래도 복원할 수 있는 압축 방법
    - Gzip 등

Huffman codes (2)

- 구성하는 방법
  - 단어와 단어의 사용 빈도의 pair 정보를 저장한 nodes set이 주어짐
  - 사용빈도가 제일 낮은 두 개의 단어를 leaf node로 하는 subtree 구성
    - Subtree의 root의 사용빈도는 children의 사용빈도의 합
  - 사용된 단어를 set에서 제거하고 생성된 subtree의 root를 set에 추가
  - 위 과정을 set에 하나의 node가 남을 때까지 반복




(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9

Huffman codes (4)

- Pseudo code 작성
  - Min priority heap Q 사용

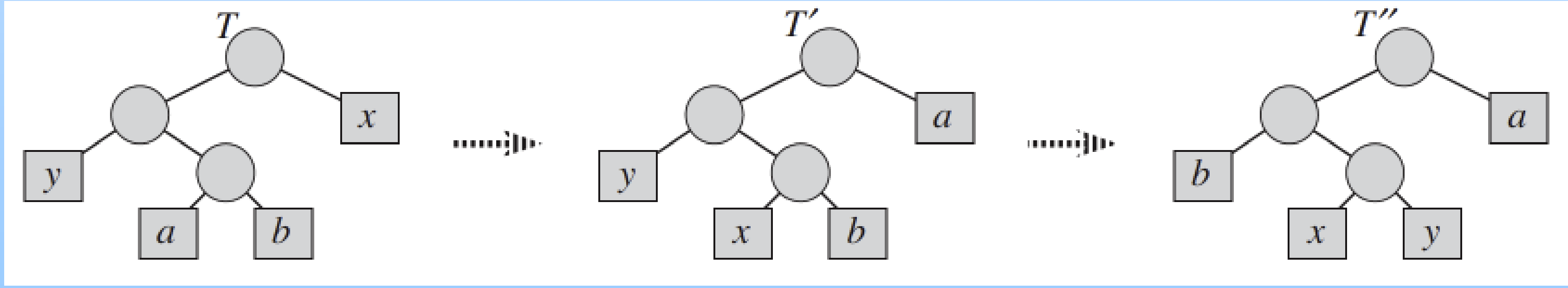
- Pseudo code

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

Huffman codes correctness (1)

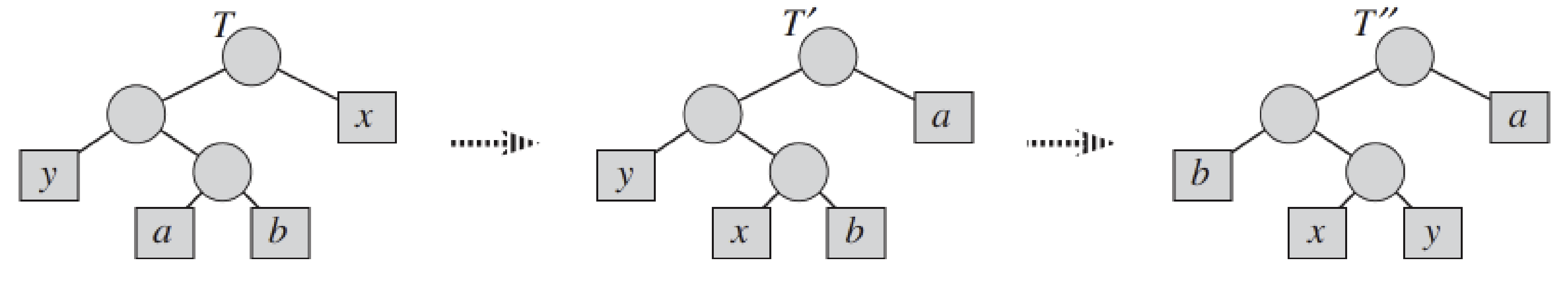
- ***Lemma 16.2***
  Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

- 증명
  - 아이디어
    - 임의의 optimal prefix code를 나타내는 tree T를 고려
    - Tree T를 변형해서 다른 optimal prefix code를 나타내는 tree T′을 구성. 이 때 tree T′에서는 x와 y가 tree에서 가장 depth가 크고 서로 인접하도록 구성함.



Huffman codes correctness (2)

- 증명 (cont.)

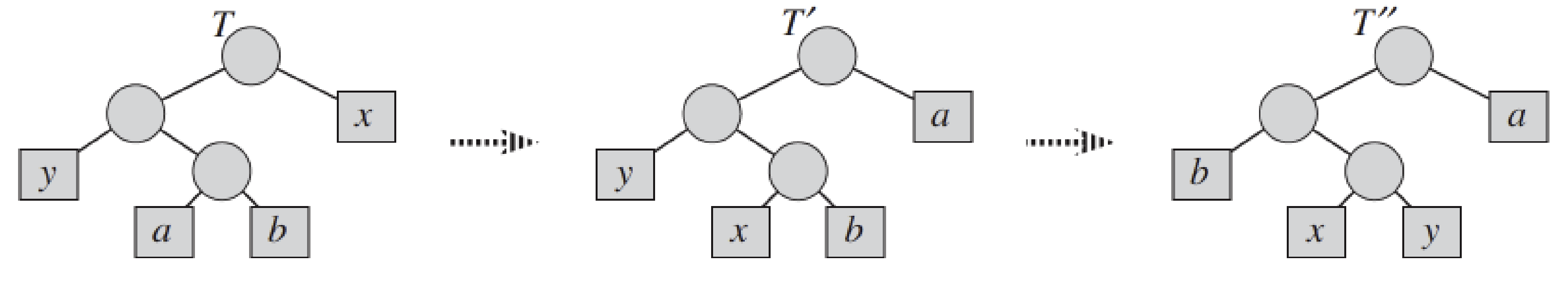


- T는 optimal prefix code를 나타내는 tree이다.
- x와 y가 전체에서 제일 적게 사용되는 문자라는 점이 주어져 있음.
- x가 a이면 x와 a의 교체 과정 생략
- x가 a가 아닌 경우
- x의 depth가 a의 depth 보다 작다면 T는 optimal이 아님 (T와 T′의 비용 차이에 대한 상세 수식은 다음 페이지...)
- 모순이므로 x.depth == a.depth이어야 함.
- x.depth == a.depth이면 x와 a를 교체해도 전체 비용은 변함 없음
- y와 b의 관계도 마찬가지.
- 따라서, optimal인 경우 T″처럼 구성할 수 있음.
- x와 y가 Sibling이고 1bit만 다른 optimal prefix code가 존재한다.

Huffman codes correctness (3)

- 증명 (cont.)
  - T와 T′의 비용 차이

$$
\begin{aligned}
B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&= (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}
$$



- x의 depth가 a의 depth 보다 작다면, B(T) > B(T′)이므로 B(T)가 optimal이 아님.

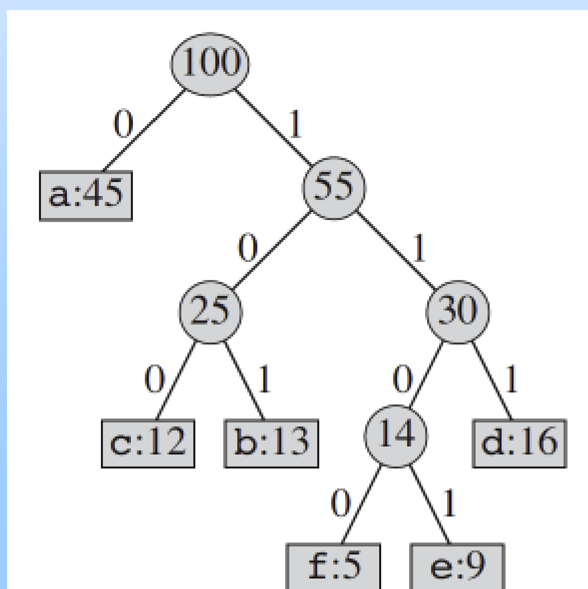
Huffman codes correctness (4)

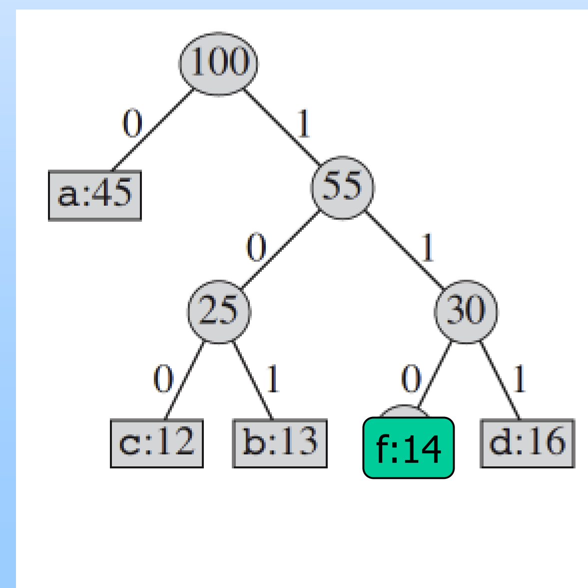
- ***Lemma 16.3***

  Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

  - 의미: T′이 optimal prefix code이면 T도 optimal prefix code

T





T′

Huffman codes correctness (5)

- 증명
  - 모순에 의한 증명
  - 노드를 대체하면 B(T) = B(T′) + x.freq + y.freq
  - T가 optimal tree가 아니라고 가정하자
  - 그러면 optimal tree인 T_o가 존재한다.
  - Lemma 16.2에 의해서 x와 y는 T_o에서 sibling이다.
  - T_o에서 x와 y의 parent 대신 x.freq + y.freq에 해당하는 새로운 node z를 추가한 결과를 T_o′라 하자.
  - B(T_o′) = B(T_o) - x.freq – y.freq

    < B(T) – x.freq – y. freq (T가 optimal이 아니라는 가정에 의해서)

    = B(T′)
  - B(T′)은 optimal이라고 가정했다. 여기서, B(T_o′) < B(T′)이라는 모순 발생
    - 즉, B(T′)보다 비용이 더 작은 Tree 존재하는 모순 발생
  - 따라서, T가 optimal tree가 아니라는 가정이 잘못됨
  - 즉, T는 optimal tree이다.

Huffman codes correctness (6)

- 손코딩
  - 입력
    - (문자, 빈도)로 구성된 n개의 pair
  - 출력
    - Huffman code에 해당되는 tree

Task Scheduling Problem (1)

- 문제 설명
  - 하나의 프로세스에서 여러 개의 task를 처리하려고 한다.
  - 각 task 처리 소요 시간은 1이다.
  - 각 task마다 deadline이 있고 deadline안에 끝내지 못하면 penalty가 주어진다.
  - 총 penalty의 합이 최소가 되는 schedule을 정하라.

- 문제 정의
  - Input
    - A set S = $\{a_1, a_2, \ldots, a_n\}$ of n unit-time tasks
    - Task a_i의 deadline d_i, $1 <= d_i <= n$
    - Task a_i의 non-negative penalty w_i
  - Output
    - Penalty가 최소가 되는 schedule

Task Scheduling Problem (2)

- Solution
  - Penalty 내림차순으로 정렬
  - 정렬된 순서대로 아래 과정 진행
    - 추가했을 때 각 시간 t별로 deadline이 늦지 않은 task 개수 측정.
    - 각 시간마다 늦지 않은 task 개수가 t보다 작으면 task 추가.
    - 예) task의 deadline이 2이면 T[2]~T[n]까지 각각을 1씩 증가시킴. 만약 T[i] > i인 경우가 있으면 추가하지 않음
  - 결과에 추가된 task들을 deadline 증가순으로 정렬
  - $O(n^2)$ 알고리즘

동전 교환

- 문제 설명 (교재 Problems 16-1)
  - 지폐를 동전으로 환전할 때 동전 개수가 최소가 되도록 환전하고자 한다.
- 문제 1
  - 동전 종류가 10원, 50원, 100원, 500원일 때 동전 개수를 최소화하는 방법을 제시하라.
- 문제 2
  - 주어진 동전 종류에 따라서 greedy algorithm으로 해를 구할 수 없는 경우가 있다. 이러한 경우에 해당하는 동전 종류를 예시하라.
- 문제 3
  - 임의의 동전 종류가 주어졌을 때 최적의 해를 구하는 방법을 제시하라.

HW.C6

- HW.C6
  Huffman codes 구현

- 마감: 다음 주 수업 시작 전