챕터 9. Greedy Approach (탐욕 접근법) - (2) Huffman's Code

과제 정보:

Input

첫째 줄에 문서에 포함된 문자의 개수 N이 주어진다.

둘째 줄에 N개의 문자(characters)가 주어진다.(문자는 알파벳 문자임이 보장되고, 대소문자는 구분한다.)

셋째 줄에 각 문자의 출현 빈도(frequecies)가 주어진다.

단, 출현 빈도는 오름차순으로 정렬되어 있다고 보장할 수 없다. (알고리즘 실행 전에 정렬부터 할 것!)

이후 허프만 코드로 인코딩할 문자의 개수 M이 주어진다.

이후 M개의 문자열이 주어진다.

이후 문자열로 디코딩할 허프만 코드의 개수 K가 주어진다.

이후 K개의 허프만 코드가 주어진다.

Output

첫째 줄에 허프만 트리의 전위(preorder) 순회 결과를 출력한다.

둘째 줄에 허프만 트리의 중위(inorder) 순회 결과를 출력한다.

(단, 순회 결과를 출력할 때 줄 끝에 공백 문자를 출력하지 않도록 주의해야 한다.)

(단, 각 노드의 표현은 '문자:빈도' 형태로 출력한다. 리프 노드가 아닌 내부 노드의 경우 문자를 '+'로 출력한다.)

이후 M개의 줄에 해당 문자열을 인코딩한 허프만 코드를 출력한다.

이후 K개의 줄에 해당 허프만 코드를 디코딩한 문자열을 출력한다.

(이상 모든 줄 끝에는 공백 문자가 출력되지 않도록 주의한다.)

Sample Input 1

```
6
b e c a d f
5 10 12 16 17 25
5
cab
dec
fad
becadf
fdaceb
5
110001110
011111110
100001
11101111110000110
10010011011111110
```

Sample Output 1

```
+:85 +:33 a:16 d:17 +:52 f:25 +:27 c:12 +:15 b:5 e:10
a:16 +:33 d:17 +:85 f:25 +:52 c:12 +:27 b:5 +:15 e:10
110001110
011111110
100001
11101111110000110
10010011011111110
cab
dec
fad
becadf
fdaceb
```

Sample Input 2

```
7
A B I M S X Z
12 7 18 10 9 5 1
8
A
B
I
M
S
X
Z
ABIMSXZ
4
01100010101010
1000100001010
11100100111101
1000010011100
```

Sample Output 2

```
+:62 +:25 A:12 +:13 +:6 Z:1 X:5 B:7 +:37 I:18 +:19 S:9
M:10
A:12 +:25 Z:1 +:6 X:5 +:13 B:7 +:62 I:18 +:37 S:9 +:19
M:10
00
011
10
111
110
0101
0100
000111011111001010100
BAXX
IAIAX
MAIBS
IAZMA
```

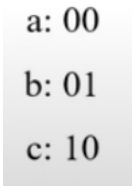
허프만 코드를 공부하기 전에 필요한 배경지식

1. 이진 코드 (binary code)

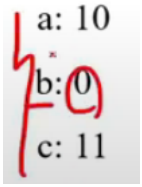
데이터 파일을 이진 코드로 인코딩하여 저장 (압축 파일 등)

- 길이가 고정된 (fixed-length) 이진 코드 (ex: ASCII / UNICODE)
- 길이가 변하는 (variable-length) 이진 코드 <- 압축 효율이 좋음

만약, 사용하는 문자의 집합이 {a, b, c}라면?



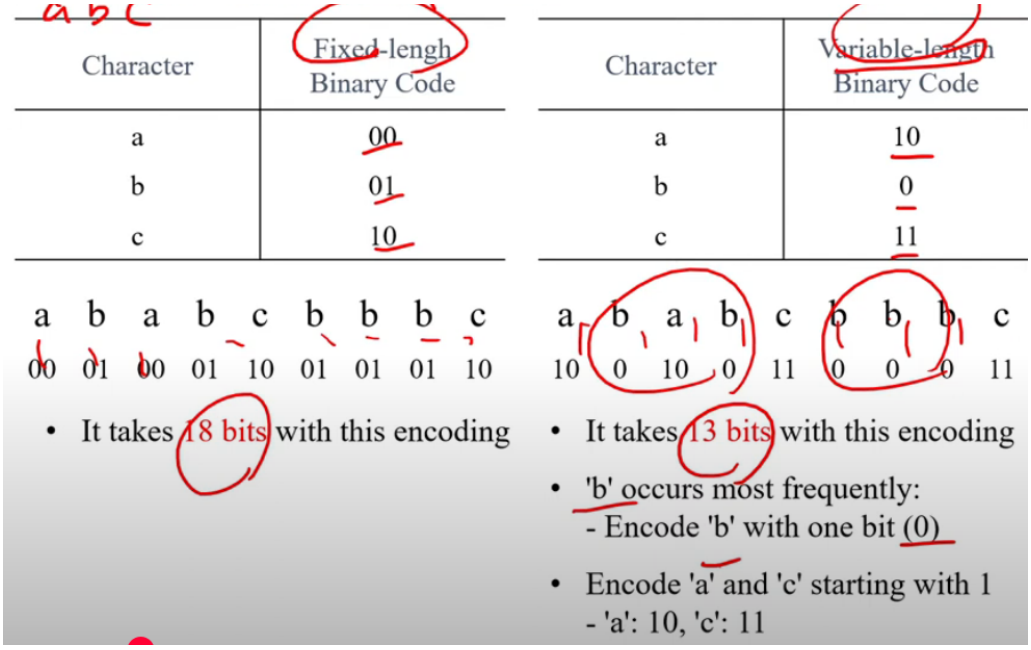
abcc : 00011010 으로 읽을 수 있다. 하지만 압축 효율 면에서는 별로 좋지 않다.



abcc : 1001111 으로 읽을 수 있다. 이는 한 비트가 줄음으로써 압축 효율이 좋아졌다.

2. EX:

File = "ababcbbbc"와 S = {a, b, c}인 abc로 구성되어있는 파일과 S집합이 있다.



첫번째 Fixed-length 이진 코드는 18bits 인코딩을 가지지만,

두번째 Variable-length 이진 코드는 13bits 인코딩을 가진다.

따라서 가장 자주 나타나는 문자의 비트 길이를 가장 짧도록 초기화해주면 효율이 올라간다는것을 알 수 있다.

최적 이진코드 문제 (optimal binary code)

주어진 파일에 있는 문자들을 이진코드로 표현할 때

필요한 비트의 개수가 최소가 되는 이진코드를 찾아라.

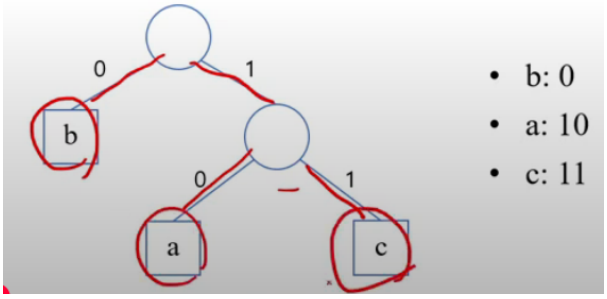
전치 코드 (prefix code)

길이가 변하는 이진코드의 특수한 형태

한 문자의 코드워드가 다른 문자의 코드워드의 앞부분이 될 수 없다.

- 예) 01이 'a'의 코드워드라면 011은 'b'의 코드워드가 될 수 없다.

전치코드의 표현: 모든 전치코드는 리프노드가 코드문자인 이진트리로 표현 가능



가장 자주 나타나는 문자인 'b'를 제일 짧은 곳에 둔다

전치 코드의 장점: 파일을 파성(디코딩)할 때 뒷부분을 미리 볼 필요가 없다.

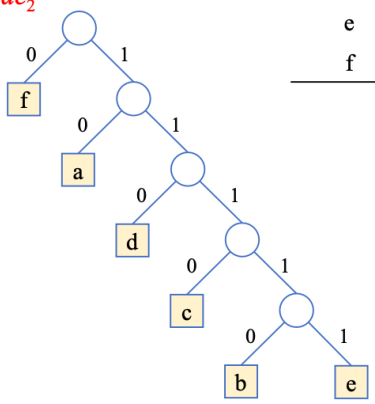
예시로, 빈도수가 주어져있는 character들의 표가 있다.

$S = \{a, b, c, d, e, f\}$

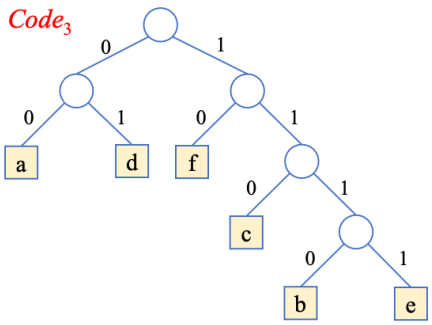
Character	Frequency	$Code_1$	$Code_2$	$Code_3$
a	**16**	000	10	00
b	**5**	001	11110	1110
c	**12**	010	1110	110
d	**17**	011	110	01
e	**10**	100	11111	1111
f	**25**	101	0	10

- $Code_1$: Fixed-Length
- $Code_3$: Huffman code

$Code_2$



$Code_3$



code1은 Fixed-length이며 기본 값이다.

code2는 frequency기준으로 가장 많은 character의 이진표현을 줄인 것, 그리고 단일트리형태를 보여준다.

code3은 균형잡힌 이진트리형태를 보여주며, 허프만은 이것이 가장 optimal하다라는 것을 증명하였다.

-> 그럼 code3은 어떻게 찾을까?

허프만 코드 (Huffman's code)

허프만 알고리즘에 의해 생성된 최적 이진코드

허프만 알고리즘 (Huffman;s algorithm)

허프만 코드에 해당하는 최적 이진트리를 구축하는 탐욕 알고리즘(Greedy Approach)

최적 이진코드를 위한 이진트리의 구축

데이터 파일을 인코딩하는 데 필요한 비트의 수 계산

주어진 이진트리를 T라 하자.

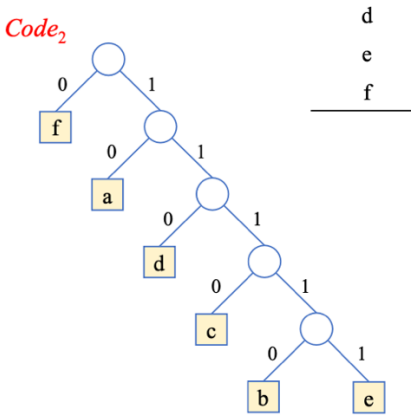
{v1, v2, ---, vn} : 데이터 파일 내의 문자들의 집합

frequency(vi) : 문자 vi가 파일 내에서 나타나는 빈도수

depth(vi) : 이진 트리 T에서 문자 vi 노드의 깊이

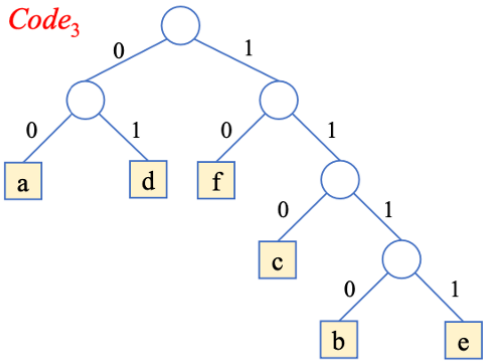
$$bits(T) = \sum_{i=1}^{n} frequency(v_i)depth(v_i)$$

- bits(T2) = 231



d
e
f

- bits(T3) = 212 (optimal)



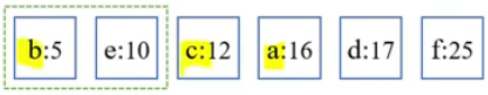
허프만 알고리즘 : 탐욕 알고리즘 (The Greedy Approach)

n : 주어진 데이터 파일내 문자의 개수

PQ : 빈도수가 낮은 노드를 먼저 리턴하는 우선순위큐(min-heap) priority-queue

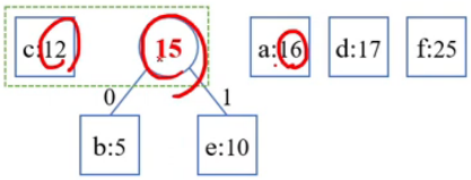
```
for i in [1..n-1]:
    remove(PQ, p)
    remove(PQ, q)
    r = new node
    r->left = p
    r->right = q
    r->frequency = q->frequency + q->frequency
    insert(PQ, r)
remove(PQ, r)
return r
```

1. 앞은 character, 뒤는 frequency, 먼저 frequency기준 비내림차순으로 정렬하였다.

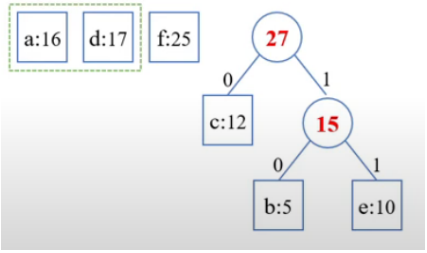


2. 꺼낼 수 있는 방식이 고급 technic이다.

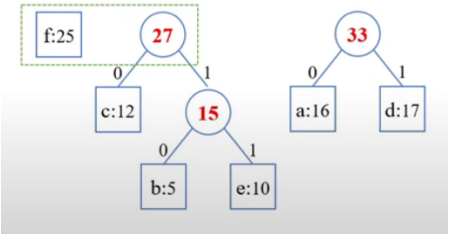
-> b와 e의 frequency를 더하여서 r 을 구한 뒤, 이 r이 15니까 12인 c와 16인 a 사이에 넣는다.



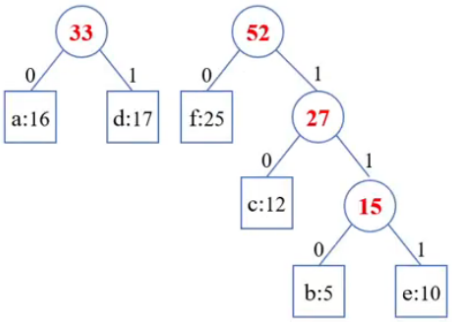
3. 이 상태에서 priority-queue를 remove하면 c와 15가 return된다. 따라서 c는 p, 15는 q로, 둘을 subtree를 만드는 r이 있고, 12 + 15니까 이번에 r은 27이 된다. 이 r은 27이니까 25인 f 뒤에 넣는다.



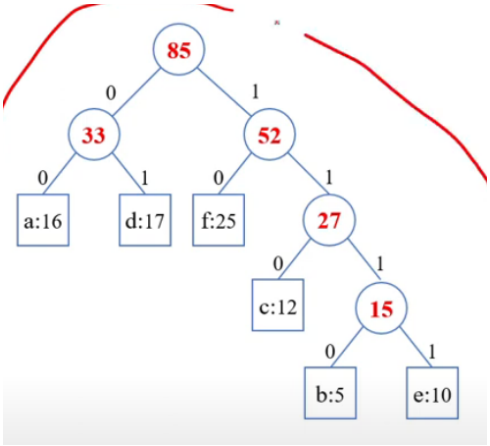
4. 이 상태에서 다시 remove하면 a는 p, d는 q로, 둘을 subtree를 만드는 r이 있고, 16 + 17이니까 이번에 r은 33이 된다. 이 r은 33이니까 27인 트리의 뒤에 넣는다.



5. 이 상태에서 다시 remove하면 f는 p, 27은 q로, 둘을 subtree로 만드는 r이 있고, 25 + 27이니까 이번에 r은 52가 된다. 이 r은 52이니까 33인 트리의 뒤에 넣는다.



6. 이 상태에서 다시 remove하면 33은 p, 52는 q로, 둘을 subtree로 만드는 r이 있고, 33 + 52이니까 이번에 r은 85가 된다. 이 r은 85니까 root노드에 그대로 넣는다.



따라서 이렇게 bits(T3) = 212 (optimal)인 Code3를 만들었다.

이상 알고리즘을 만들기까지의 과정이며 아래는 코드들을 나열할 것.

Codes

```
struct HuffNode
{
    char symbol;     // 심볼
    int freq;        // 빈도수
    HuffNode *left;  // 왼쪽 자식 노드
    HuffNode *right; // 오른쪽 자식 노드

    // 생성자
    HuffNode(char s, int f)
    {
        symbol = s;
        freq = f;
        left = nullptr;  // 초기값은 nullptr
        right = nullptr; // 초기값은 nullptr
    }
```

```cpp
// 전위 순회
    void preorder(int &count)
    {
        if (count > 0)
            cout << " ";                                        // 첫 번째 노드가 아닐 경우 공백 출력
        cout << (symbol == ' ' ? '+' : symbol) << ":" << freq; // 빈도수 출력
        count++;
        if (left != nullptr)
        {
            left->preorder(count); // 왼쪽 자식 노드 전위 순회
        }
        if (right != nullptr)
        {
            right->preorder(count); // 오른쪽 자식 노드 전위 순회
        }
    }

    // 중위 순회
    void inorder(int &count)
    {
        if (left != nullptr)
        {
            left->inorder(count); // 왼쪽 자식 노드 중위 순회
        }
        if (count > 0)
            cout << " ";                                        // 첫 번째 노드가 아닐 경우 공백 출력
        cout << (symbol == ' ' ? '+' : symbol) << ":" << freq; // 빈도수 출력
        count++;
        if (right != nullptr)
        {
            right->inorder(count); // 오른쪽 자식 노드 중위 순회
        }
    }
```

```cpp
// 비교 함수 (우선순위 큐에서 사용)
struct Compare
{
    bool operator()(HuffNode *a, HuffNode *b)
    {
        return a->freq > b->freq; // 빈도수가 낮은 순서로 정렬
    }
};

```

```cpp
// 허프만 알고리즘
HuffNode *huffman(int n, priority_queue<HuffNode *, vector<HuffNode *>, Compare> &PQ)
{
    for (int i = 0; i < n - 1; ++i)
    {
        HuffNode *p = PQ.top();
        PQ.pop(); // 가장 낮은 빈도수의 노드
        HuffNode *q = PQ.top();
        PQ.pop(); // 두 번째로 낮은 빈도수의 노드

        HuffNode *r = new HuffNode(' ', p->freq + q->freq); // 새로운 노드 생성
        r->left = p;                                        // 왼쪽 자식
        r->right = q;                                       // 오른쪽 자식

        PQ.push(r); // 새로운 노드를 우선순위 큐에 추가
    }
    return PQ.top(); // 최종 루트 노드 반환
}
```

```cpp
// 허프만 코드 생성
void generateHuffmanCodes(HuffNode *root, const string &code, unordered_map<char, string> &huffmanCodes)
{
    if (!root)
        return;

    if (root->symbol != ' ')
    {
        huffmanCodes[root->symbol] = code; // 리프 노드일 경우 코드 저장
    }

    generateHuffmanCodes(root->left, code + "0", huffmanCodes);  // 왼쪽 자식
    generateHuffmanCodes(root->right, code + "1", huffmanCodes); // 오른쪽 자식
}
```

```cpp
int main()
{
    // 예시로 우선순위 큐 생성 및 허프만 알고리즘 실행
    priority_queue<HuffNode *, vector<HuffNode *>, Compare> PQ;

    int N;
    cin >> N; // 문자 개수 입력

    vector<pair<char, int>> input(N); // 문자와 빈도수를 저장할 벡터
    for (int i = 0; i < N; i++)
        cin >> input[i].first; // 문자 입력
    for (int i = 0; i < N; i++)
        cin >> input[i].second; // 빈도수 입력

    // 빈도수 기준으로 정렬
    sort(input.begin(), input.end(), [](const pair<char, int> &a, const pair<char, int> &b)
         { return a.second < b.second; });

    // 데이터 추가
    for (int i = 0; i < N; i++)
    {
        PQ.push(new HuffNode(input[i].first, input[i].second)); // 우선순위 큐에 노드 추가
    }

    HuffNode *root = huffman(N, PQ); // 허프만 알고리즘 실행

    // 결과 출력 (루트 노드의 빈도수)
    int preCount = 0;
    root->preorder(preCount);
    cout << endl;

    // 중위 순회 출력
    int inCount = 0;
    root->inorder(inCount);
    cout << endl;

    // 허프만 코드 생성
    unordered_map<char, string> huffmanCodes;
    generateHuffmanCodes(root, "", huffmanCodes);

    int M;
    cin >> M; // 인코딩할 문자의 개수
    for (int i = 0; i < M; i++)
    {
        string str;
        cin >> str; // 문자열 입력
        string encoded = "";
        for (char c : str)
        {
            encoded += huffmanCodes[c]; // 허프만 코드로 인코딩
        }
        cout << encoded << endl; // 인코딩 결과 출력
    }

    int K;
    cin >> K; // 디코딩할 허프만 코드의 개수
    for (int i = 0; i < K; i++)
    {
        string code;
        cin >> code; // 허프만 코드 입력
        string decoded = "";
        HuffNode *current = root;
        for (char c : code)
        {
            current = (c == '0') ? current->left : current->right; // 트리 탐색
            if (current->symbol != ' ')
            {                                // 리프 노드에 도달했을 때
                decoded += current->symbol; // 문자 추가
                current = root;             // 루트로 돌아감
            }
        }
        cout << decoded << endl; // 디코딩 결과 출력
    }

    return 0;
}
```