Problem A

Lossless Text Compression

Translating text characters to binary prefix codes is a common way to perform lossless compression of text – *i.e.*, to produce a smaller representation of the data that can be used to reconstruct the original input without any loss. Prefix codes are codifications whose main property is that there is no code word in the system that is a prefix of any other code word in the same system. This trait ensures that the original data can be reconstructed straightforward through a "single pass" on the compressed text, without further parsing.

In this context, the proposed problem can be defined as follows. Given a set of symbols (in this case, text characters) and their weights (usually proportional the number of times it occurs on some input text), find a set of binary prefix codes (one for each symbol) whose expected codeword length is minimum. More formally, given the symbol alphabet $A = \{a_1, a_2, \ldots, a_n\}$ and the set of positive symbol weights $W = \{w_1, w_2, \ldots, w_n\}$, such that $weight(a_i) = w_i$, find a tuple of codewords $C = (c_1, c_2, \ldots, c_n)$, where $c_i = codeword(a_i)$. The goal, thus, is to obtain a C such that, for any other code T, $L(C) < T(C)$, where $L(C) = \sum_{i=1}^{n} w_i \times length(c_i)$ is the weighted path length of code C.

You shall parallelize *Huffman Coding*, an algorithm that generates minimum binary prefix codes for a list of characters and their associated frequencies on some input text. The technique works by creating a binary tree of nodes where each node can be either a leaf node (corresponding to a symbol and its associated frequency) or an internal node (corresponding to a sum of frequencies). Then, the codes are obtained by cumulatively labeling each link between nodes with 0 or 1 in a depth-first visit on the nodes of this tree. The algorithm is detailed next.

Initially, there are only n leaf nodes, one for each symbol, which contains the symbol itself and its weight (frequency). The construction algorithm uses a priority queue sorted by the symbols' weight, where the node with lowest weight is given highest priority:

1. Add all leaves to the priority queue.
2. While there is more than one node in the queue:
    a. Remove the two nodes of highest priority (lowest weight) from the queue;
    b. Create a new internal node with these two nodes as children and with weight equal to the sum of the two nodes' weights;
    c. Add the new node to the queue.

3. The remaining node is the root node and the tree is complete.

After the tree is assembled, in order to get the binary codes one shall perform a depth-first tree traversal in pre-order. To each leaf corresponds a binary code, built going from the root to it, adding a "0" at each time the path goes by a left child and an "1" when it goes by a right child.

Figure A1 shows an assembled Huffman Tree, with the paths labeled. A given code is delivered by concatenating the zeros and ones from a leaf to the root.

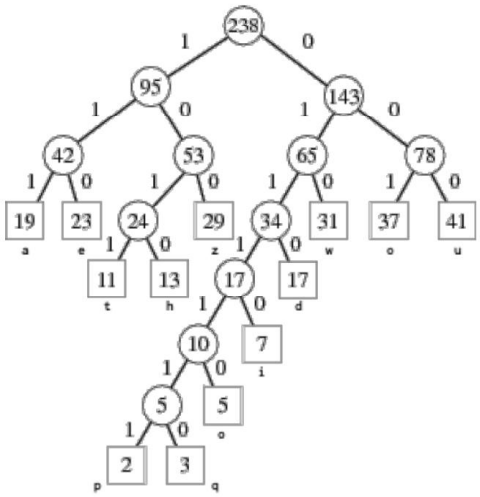

Figure A1. A Huffman Tree. The leaf nodes are represented by pink squares and the numbers inside it represent the frequency of a symbol, which is placed under it. The circles are the intermediary nodes and represent a sum of frequencies. Each edge is labeled with 0 or 1, corresponding to a left or right child (the concatenation of the labels in a given branch is the binary prefix code for a given tree)[1].

Important. Although more than one correct solution exists, your parallel version shall provide the same output as the provided sequential solution.

Input

The first line of the input contains the number of lines of the input text, say N ($0 < N \leq 10^6$). Then, the following N lines are strings of at most 256 ASCII characters that serve as symbols. The only restriction on the characters to appear is that they are readable and

[1] Figure from http://mathworld.wolfram.com/HuffmanCoding.html

no other spacing is present other than the space character.

Blank lines may appear and are valid.

The input must be read from the standard input.

Output

It is also a sequence of lines. Each line corresponds to a symbol and shows its frequency (number of time it appears) on the input text and the correspondent binary code. The format is "`X Y Z`", where `X` is the symbol, `Y` is the frequency, and `Z` is the binary prefix code. For instance, one line could be "`h 13 0110`", meaning the symbol `h` appears `13` times on the input text and has prefix code `0110`. There is only one invariant: if the input text contains the space character it must appear as the word "`space`" on the output instead of the space character.

The output must be written to the standard output.

Example

Input	Output for the input

`2`
`this is an example of a huffman tree`
`j'aime aller sur le bord de l'eau les jeudis`
	`' 2 110111`
	`a 7 1110`
	`b 1 1111000`
	`d 3 10110`
	`e 11 100`
	`f 3 10111`
	`h 2 111101`
	`i 4 11111`
	`j 2 01000`
	`l 6 1100`
	`m 3 11010`
	`n 2 01001`
	`o 2 01010`
	`p 1 1111001`
	`r 4 0110`
	`s 5 1010`
	`space 15 00`
	`t 2 01011`
	`u 4 0111`
	`x 1 110110`

```
class HuffmanTree {

    enum { left = 0, right = 1 };

  public:
    Weight weight = Weight{0};
    Symbol symbol = Symbol{};
    Vector<HuffmanTree> children;

    HuffmanTree() {}

    HuffmanTree(const HuffmanTree& e) :
        symbol   = e.symbol   ;
        weight   = e.weight   ;
        children = e.children ;
    {}

    HuffmanTree& operator=(const HuffmanTree& e) {
        symbol   = e.symbol   ;
        weight   = e.weight   ;
        children = e.children ;
        return *this ;
    }

    HuffmanTree(Symbol s, Weight w) {
        symbol = s ;
        weight = w ;
    }

    HuffmanTree(HuffmanTree a, HuffmanTree b) {
        weight = a.weight + b.weight ;
        children.push_back(a) ; children.push_back(b) ;
        symbol = a.symbol | b.symbol ;
    }

    HuffmanTree left_subtree() { return children[left] ; }
    HuffmanTree right_subtree() { return children[right] ; }
    bool is_leaf() { return children.empty() ; }

    friend bool operator==(HuffmanTree a, HuffmanTree b) {
        return a.symbol == b.symbol && a.weight == b.weight ;
    }

    friend bool operator<(HuffmanTree a, HuffmanTree b) {
        return a.weight == b.weight ? a.symbol < b.symbol : a.weight < b.weight ;
    }

    friend bool operator>(HuffmanTree a, HuffmanTree b) {
        return b < a ;
    }

    friend bool operator<=(HuffmanTree a, HuffmanTree b) {
        return !(a > b) ;
    }

    friend bool operator>=(HuffmanTree a, HuffmanTree b) {
        return !(a < b) ;
    }
} ;

HuffmanTree extract(PriorityQueue<HuffmanTree>& p) {
    HuffmanTree r = p.top() ;
    p.pop() ;
    return r ;
}

Map<Symbol, Code> walk(HuffmanTree h, Code c = Code(), Map<Symbol, Code> m = Map<Symbol, Code>()) {
    if (h.is_leaf()) {
        m.insert(h.symbol, c) ;
    } else {
        m = walk(h.left_subtree(), c + Code("0"), m) ;
        m = walk(h.right_subtree(), c + Code("1"), m) ;
    }
    return m ;
}

PriorityQueue<HuffmanTree> make_queue(Map<Symbol, Weight> m) {
    PriorityQueue<HuffmanTree> q ;
    for (auto i : m)
        q.push(HuffmanTree(i.first, i.second)) ;
    return q ;
}

HuffmanTree huffman(Map<Symbol, Weight> m) {
    assert(!m.empty()) ; // precondition
    PriorityQueue<HuffmanTree> p = make_queue(m) ;
    HuffmanTree a, b;
    do {
        a = extract(p) ;
        if (p.empty()) break ;
        b = extract(p) ;
        p.push(HuffmanTree(a, b)) ;
    } while (true) ;
    return a ;
}

template<typename OutStream>
int write(OutStream& out, Map<Symbol, Weight> w, Map<Symbol, Code> c) {
    for (auto i : w)
        out << i.first << " " << i.second << " " << c[i.first] << std::endl;
    return 0 ;
}

template<typename I, typename M> bool found(I i, const M& m) {
    return i != m.end() ;
}

template<typename Pair> Map<Symbol, Weight> insert_or_add(Map<Symbol, Weight> m, Pair p) {
    auto i = m.find(p.first) ;
    if (found(i, m)) i -> second += p.second ;
    else m.insert(p) ;
    return m ;
}

Map<Symbol, Weight> map_weighted_union(Map<Symbol, Weight> a, Map<Symbol, Weight> b) {
    for (auto e : b) a = insert_or_add(a, e) ;
    return a ;
}

template<typename Instream> String read_line(Instream& in) {
    String s ;
    std::getline(in, s) ;
    return s ;
}

Symbol to_symbol(char c) {
    if (c == ' ') return Symbol("space") ;
    return Symbol((c)) ;
}

Map<Symbol, Weight> insert_or_increment(Map<Symbol, Weight> m, Symbol s) {
    auto i = m.find(s) ; ++(i -> second) ;
    if (found(i, m)) ++(i -> second) ;
    else m.insert(s, Weight(1)) ;
    return m ;
}

template<typename Instream> Map<Symbol, Weight> read_entry(Instream& in) {
    Map<Symbol, Weight> m ;
    for (char c : read_line(in))
        m = insert_or_increment(m, to_symbol(c)) ;
    return m ;
}
```