Problem C. Evolutionary Tree

Time limit:	please refer to DOM Judge
Memory limit:	please refer to DOM Judge

For any node u on a rooted binary tree T, we define $\mathrm{LEAF_T}(u)$ to be the set of leaves of the sub-tree rooted at u.

For a subset of nodes S on a rooted binary tree T, we define $\mathrm{LCA_T}(S)$ to be the lowest common ancestor of all the nodes in S.

Given two rooted binary tree T_1, T_2, each of size $2n-1$. The leaves are numbered $1, 2, \ldots, n$ and the other nodes are numbered $n+1, n+2, \ldots, 2n-1$ in T_1 and T_2 respectively. It is guaranteed every internal node has two children.

Your task is for each node u in T_1, finding $\mathrm{LCA_{T_2}}(\mathrm{LEAF_{T_1}}(u))$.

Input

The first line of the input contains an integer $2n-1$ — the number of nodes in T_1 and T_2 respectively.

The second line of the input contains $2n-1$ integers $p_1, p_2, \ldots, p_{2n-1}$ — p_i is the parent of node i in T_1 while $p_i = 0$ if node i is the root of T_1.

The third line of the input contains $2n-1$ integers $q_1, q_2, \ldots, q_{2n-1}$ — q_i is the parent of node i in T_2 while $q_i = 0$ if node i is the root of T_2.

- $1 \le n \le 10^6$
- $0 \le p_i \le 2n-1$ for $i = 1, 2, \ldots, 2n-1$
- $0 \le q_i \le 2n-1$ for $i = 1, 2, \ldots, 2n-1$

Output

You should print $2n-1$ integers where the i-th integer represents $\mathrm{LCA_{T_2}}(\mathrm{LEAF_{T_1}}(\mathrm{i}))$.

Examples

Standard Input	Standard Output
5 4 4 5 5 0 4 5 5 0 4	1 2 3 4 4
13 11 11 12 10 9 12 13 13 10 8 9 8 0 12 12 11 9 11 13 9 10 8 0 13 10 8	1 2 3 4 5 6 7 10 10 10 12 13 10

In example 2, $\mathrm{LEAF}_{T_1}(12) = \{3, 6\}$, $\mathrm{LCA}_{T_2}(\{3, 6\}) = 13$.

Note

Evolutionary tree is typically a rooted binary tree.

This page is intentionally left blank

Problem D. Vectors Domination

Time limit: please refer to DOM Judge
Memory limit: please refer to DOM Judge

For any two vectors $V = (v_1, v_2, \ldots, v_d)$ and $U = (u_1, u_2, \ldots, u_d)$, we define

- $\max(V, U) = (\max(v_1, u_1), \max(v_2, u_2), \ldots, \max(v_d, u_d))$
- $U \overset{k}{\preceq} V$ if there are $i_1, i_2, \ldots i_k$ such that $1 \le i_1 < i_2 < \cdots < i_k \le d$ and $u_{i_j} \le v_{i_j}$ for $j = 1, 2, \ldots, k$
- V is lexicographically larger than U if there is some j such that $1 \le j \le d$ satisfying $u_i = v_i$ for $i = 1, 2, \ldots, j-1$ and $u_j < v_j$.

Given n vectors $S_1, S_2, \ldots, S_n$ and a special vector T. All the vectors are of size d.

You can perform the following operation infinitely many times:

Choose i such that $S_i \overset{k}{\preceq} T$. Update T to $\max(T, S_i)$.

Your task is to find the lexicographically largest vector T can be.

Input

The first line of the input contains three integers n, d, k — the number of vectors, the size of each vector, and the constant of $\overset{k}{\preceq}$.

The second line contains d integers $T_1, T_2, \ldots, T_d$ — the special vector T.

The i-th of the next n lines contains d integers $S_{i_1}, S_{i_2}, \ldots, S_{i_d}$ — the given vector S_i.

- $1 \le n \le 10^5$
- $1 \le k < d \le 5$
- $1 \le T_i \le 10^2$ for $i = 1, 2, \ldots, d$
- $1 \le S_{i_j} \le 10^9$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, d$

Output

Print d integers representing the lexicographically largest vector T can be.

Examples

Standard Input	Standard Output
3 2 1 3 5 2 7 5 6 8 8	5 7
5 4 3 1 2 3 4 1 1 1 8 5 2 2 7 4 4 3 4 5 4 2 7 9 6 3 2	5 4 3 8

This page is intentionally left blank

Problem E. Learning Dynamic Programming Tech

Time limit:	please refer to DOM Judge
Memory limit:	please refer to DOM Judge

DP (dynamic programming) is a well-known technique in competitive programming. Essentially, DP is a combination of recursion and memorization. Therefore, when tackling a problem with DP, the first step is to define the recursion. Many people struggle with DP because they overlook this initial step.

Let's consider the following problem from Atcoder for practice:

There are $n(1 \leq n \leq 5 \times 10^3)$ slimes lined up in a row, the i-th slime from the left has a size of $w_i(1 \leq w_i \leq 10^9)$. Your task is to perform the following operation until there is only one slime left: Choose two adjacent slimes and combine them into a new slime. The new slime has a size equal to the sum of the sizes of the two slimes being combined. This operation incurs a cost equal to the sum of their sizes. The positions of the slimes do not change during this process. Find the minimum possible total cost incurred.

Let $f(i, j)$ be the minimum cost to combine the i-th to j-th slimes into a single slime. While many prefer to denote this as $dp(\dots)$, it's important to recognize that DP involves recursion combined with memorization, so what we have is the definition of a "function". Thus, calling it "dp" might seem odd.

Now, let's return to our function. With the definition of $f(i, j)$), we can start recursively computing it. According to the definition, $f(i, j)$ represents the result of combining the i-th to j-th slimes. Considering the last step of the combining, there exists some k such that $i \leq k < j$, allowing us to combine the i-th to k-th slimes into one big slime on the left and the $k + 1$-th to j-th slimes into another big slime on the right. Finally, we combine these two big slimes into one, completing the process. Hence, we have:

$$f(i, j) = \min_{i \leq k < j} f(i, k) + f(k + 1, j) + (\text{the cost of combining left big slime and right big slime})$$

What is the cost of combining the left and right big slimes? Note that the size of the left big slime is $\sum_{l=i}^{k} w_l$ and the size of the right big slime is $\sum_{l=k+1}^{j} w_l$. Therefore the cost is their sum, $\sum_{l=i}^{j} w_l$. Surprisingly, the cost is independent of k and we can pre-compute the prefix sum of w to quickly compute it. Let $s_i = \sum_{j=1}^{i} w_j$. We have:

$$f(i, j) = \min_{i \leq k < j} f(i, k) + f(k + 1, j) + (s_j - s_{i-1})$$

Note that the answer we seek is $f(i, j)$, which represents the minimum cost to combine the 1st to n-th slimes into one. Let's apply the memorization technique to compute the time complexity to calculate $f(1, n)$. First, computing the prefix sum of w takes $O(n)$ time. There are $n \times n$ entries of $f(i, j)$ since i and j ranges from 1 to n respectively. Each recursive takes $j - i + 1 = O(n)$ time to enumerate k and compute $f(i, k) + f(k + 1, j) + (s_j - s_{i-1})$. In total, the time complexity is:

$$O(\overset{\text{prefix sum}}{n}) + O(\overset{\#\text{function entries}}{n \times n} \times \overset{\text{recursive}}{n}) = O(n^3)$$

However, since $n \leq 10^4$, $O(n^3)$ might not be sufficient to solve this problem. We need to optimize it. Typically, there are two ways to speed up: changing the function definition or optimizing recursion. In this problem, as the definition is straightforward, let's focus on optimizing the recursion.

Prefix sum of w can be computed in the following way:

```
// remember to declare s as long long since \sum w may be greater than 2^31
for (int i = 1; i <= n; ++ i) {
    cin >> w[i];
    s[i] = s[i - 1] + w[i];
}
```

Let $opt(i,j)$ be the smallest index such that $f(i,j) = f(i,opt(i,j)) + f(opt(i,j)+1,j) + (s_j - s_{i-1})$.

Now, let's compare $f(i,j-1)$ and $f(i,j)$. There is one more slime on the rightmost in the case of $f(i,j)$. One may guess that $opt(i,j-1) \le opt(i,j)$. Indeed, it is true, and the proof is left for practice. Similarly, $opt(i+1,j) \le opt(i,j)$. Hence, we can compute $f(i,j)$ in the order of $j-i+1$ to ensure $f(i,j-1)$ and $f(i+1,j)$ are computed before $f(i,j)$. Finally,

$$f(i,j) = \min_{opt(i,j-1) \le k \le opt(i+1,j)} f(i,k) + f(k+1,j) + (s_j - s_{i-1})$$

That is,

```
// l = j - i + 1, which is the length of the interval
for (int l = 2; l <= n; ++ l) {
    for (int i = 1; i + l - 1 <= n; ++ i) {
        int j = i + l - 1;
        for (int k = opt(i, j - 1); k <= opt(i + 1, j); ++ k) {
            // compare current f(i, j) to f(i, k) + f(k + 1, j) + (s_j - s_{i-1})
            if ("strictly" better) {
                // "strictly" since opt is defined as the smallest among the best
                // update f(i, j)
                // update opt(i, j) to k
            }
        }
    }
}
// there may be bugs
```

What about the time complexity? The total time taken by the recursion is:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} opt(i+1,j) - opt(i,j-1) + 1 \tag{1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} opt(i+1,j) - \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} opt(i,j-1) + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \tag{2}$$

$$= \sum_{i=2}^{n} \sum_{j=i+1}^{n} opt(i,j) - \sum_{i=1}^{n-1} \sum_{j=i}^{n-1} opt(i,j) + O(n^2) \tag{3}$$

$$= \sum_{i=2}^{n-1} \sum_{j=i+1}^{n} opt(i,j) - \sum_{i=1}^{n-1} \sum_{j=i}^{n-1} opt(i,j) + O(n^2) \tag{4}$$

$$= \sum_{i=2}^{n-1} \sum_{j=i+1}^{n-1} opt(i,j) + \sum_{i=2}^{n} opt(i,n) - \sum_{i=2}^{n-1} \sum_{j=i+1}^{n-1} opt(i,j) - \sum_{i=2}^{n} opt(i,i) + O(n^2) \tag{5}$$

$$= \sum_{i=2}^{n} (opt(i,n) - opt(i,i)) + O(n^2) \tag{6}$$

Since $opt(i,n) = O(n)$. $\sum_{i=2}^{n} opt(i,n) = O(n^2)$. Hence the result is $O(n^2)$, which is sufficiently fast to solve this problem.

Note that we still need to deal with the base cases. According to the definition, it is reasonable to make $f(i,i) = 0$. Besides, $f(i,j) = \infty$ for $i \neq j$ initially. How about $opt(i,i)$? Since $f(i,i)$ is not computed via recursion, $opt(i,i)$ is undefined. For convenience, we can assign $opt(i,i) = i$, and rewrite the recursion to make it cover all the cases:

$$f(i,j) = \min_{opt(i,j-1) \le k \le \min(opt(i+1,j), j-1)} f(i,k) + f(k+1,j) + (s_j - s_{i-1})$$

Now that we know how to solve the problem, let's write some code to test your understanding.

Input

The first line of the input contains an integer n — the number of slimes.

The second line of the input contains n positive integers $w_1, w_2, \ldots, w_n$ — the weights of the slimes.

- $1 \leq n \leq 5 \times 10^3$
- $1 \leq w_i \leq 10^9$

Output

Print one integer representing the minimum possible total cost incurred

Examples

Standard Input	Standard Output
4 1 2 3 4	19
5 2 7 3 2 5	43

The $opt(i, j)$ of example 2 are listed below:

One can find $opt(i, j-1) \leq opt(i, j) \leq opt(i+1, j)$. That is, for each entry, it is greater than or equal to the entries on its left or above it.

For example, $opt(2, 3) = 2 \leq opt(2, 4) = 2 \leq opt(3, 4) = 3$.

$$\begin{array}{llll} opt(1,2)=1 & opt(1,3)=2 & opt(1,4)=2 & opt(1,5)=2 \\ & opt(2,3)=2 & opt(2,4)=2 & opt(2,5)=2 \\ & & opt(3,4)=3 & opt(3,5)=4 \\ & & & opt(4,5)=4 \end{array}$$

The $f(i, j)$ of example 2 are listed below.

$$\begin{array}{llll} f(1,2)=9 & f(1,3)=21 & f(1,4)=28 & f(1,5)=43 \\ & f(2,3)=10 & f(2,4)=17 & f(2,5)=32 \\ & & f(3,4)=5 & f(3,5)=15 \\ & & & f(4,5)=7 \end{array}$$

Note

By the way, the speed up used in this problem is also called the Knuth's Optimization.

This page is intentionally left blank

Problem F. White Season

Time limit: please refer to DOM Judge
Memory limit: please refer to DOM Judge

December, often referred to as the white season due to the likelihood of snowfall, carries with it a romantic ambiance that easily captivates hearts.

Normally, love is beautiful, love is fulfilling, love is wonderful. However, when it comes to a love triangle, it takes a bitter turn.

Imagine the following scenario: Suppose there are three individuals: A, B, and C. A loves B, B loves C, and C loves B as well. Finally, B and C become a blessed couple, leaving A in despair.

The situation becomes even more dire in the following instance: A loves B, B loves C, C loves B, and A "likes" C. That is, A and C share a friendship, but both harbor feelings for the same individual. The anguish experienced by A in such a predicament is beyond comprehension.

Thus, the severity of the aforementioned situations is evident, and even the problem setter of this problem acknowledges it. Hence, the problem setter aims to circumvent such scenarios within this problem.

For any two non-negative integers a and b, we define a loving b as $a \vee b = b$, where $\vee$ denotes the logical OR operation, signifying that a is entirely encompassed by b.

Given n unique non-negative integers $s_1, s_2, \ldots, s_n$, the problem setter ponders whether a love triangle is possible. Specifically, the problem setter seeks to determine if there exists distinct i, j, k such that $1 \leq i, j, k \leq n$ satisfying s_i loves s_j, s_i loves s_k, and s_j loves s_k.

Input

The first line of input contains an integer n —the number of integers.

The second line of input contains n non-negative integers, $s_1, s_2, \ldots, s_n$.

- $3 \leq n \leq 100000$
- $1 \leq s_i \leq 10^6$ for $i = 1, 2, \ldots, n$

Output

Print "love triangle"(without quotes) if there exist distinct i, j, k such that $1 \leq i, j, k \leq n$ satisfying s_i loves s_j, s_i loves s_k, and s_j loves s_k. Otherwise, print "peace"(without quotes).

Examples

Standard Input	Standard Output
3 1 3 7	love triangle
4 1 2 4 6	peace

Note

You may only need to read the last two paragraph of the statement.

This page is intentionally left blank