15^{th} Marathon of Parallel Programming
WSCAD – 2020

Calebe Bianchini[1] **and Marcos Amaris**[2]

[1]Mackenzie Presbyterian University

[2]Federal University of Pará

Rules for Remote Contest

For all problems, read carefully the input and output session. For all problems, a sequential implementation is given, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file (*zip*) with your source code, the *Makefile* and an execution script. The script must have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule all, which will be used to compile your source code. The execution script runs your solution the way you design it – it will be inspected not to corrupt the target machine.

The execution time of your program will be measured running it with time program and taking the real CPU time given. Each program will be executed at least three times with the same input and the mean time will be taken into account. The sequential program given will be measured the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (speedup). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 6 problems; pages are numbered from 1 to 8.

General Information

Compilation

You should use **CC** or **CXX** inside your *Makefile*. Be careful when redefining them! There is a simple *Makefile* inside you problem package that you can modify. Example:

```
FLAGS=-O3
EXEC=sum
CXX=icpc

all: $(EXEC)

$(EXEC):
	$(CXX) $(FLAGS) $(EXEC).cpp -c -o $(EXEC).o
	$(CXX) $(FLAGS) $(EXEC).o -o $(EXEC)
```

Each judge machine has its group of compilers. See them below and choose well when writing your *Makefile*. The compiler that is tagged as *default* is predefined in **CC** and **CXX** variables.

machine	compiler	variables
host / tsubasa	GCC 4.8.5 20150623 (*default*)	CC=gcc CXX=g++
	GCC 10.2.0	CC=gcc-10.2 CXX=g++-10.2
	NEC Compiler 3.0.8	CC=ncc CXX=nc++
MPI	GCC 8.3.0 (*default*)	CC=gcc CXX=g++
	GCC 10.2.0	CC=gcc-10.2 CXX=g++-10.2
	Open MPI 3.1.3	CC=mpicc CXX=mpic++
knl	GCC 8.3.0 (*default*)	CC=gcc CXX=g++
	GCC 10.2.0	CC=gcc-10.2 CXX=g++-10.2
	Intel C/C++ Compiler 19.1.0.166	CC=icc CXX=icpc
gpu	GCC 8.3.0 (*default*)	CC=gcc CXX=g++
	GCC 10.2.0	CC=gcc-10.2 CXX=g++-10.2
	Cuda 10.2, V10.2.89, Driver Ver: 440.33.01	CC=nvcc CXX=nvcc

Submitting

General information

You must have an execution script that has the same name of the problem. This script runs your solution the way you design it. There is a simple script inside you problem package that should be modified. Example:

```
#!/bin/bash
# This script runs a generic Problem A
# Using 32 threads and OpenMP
export OMP_NUM_THREADS=32
OMP_NUM_THREADS=32 ./sum
```

Submitting MPI

If you are planning to submit an MPI solution, you should compile using *mpicc*/*mpic++*. The script must call *mpirun*/*mpiexec* with the correct number of processes (max: 4). It must use a file called `machines` that are generated by the *auto-judge* system - **do not** create it.

```
#!/bin/bash
# This script runs a generic Problem A
# Using MPI in the entire cluster (4 nodes)
# 'machines' file describes the nodes
mpirun -np 4 -machinefile machines ./sum
```

Comparing times & results

In your personal machine, measure the execution time of your solution using *time* program. Add input/output redirection when collecting time. Use *diff* program to compare the original and your solution results. Example:

```
$ time -p ./A < original_input.txt > my_output.txt
real 4.94
user 0.08
sys 1.56

$ diff my_output.txt original_output.txt
```

Do not measure time and **do not** add input/output redirection when submitting your solution - the *auto-judge system* is prepared to collect your time and compare the results.

Problem A
Maximum Tropical Path Problem

Let $G = (V; E)$ a simple graph, undirected, where $V = v_1, v_2, \ldots, v_n$ denote the vertices and $E = e_1, e_2, \ldots, e_m$ denotes the edges. A vertex-colored graph is such that the vertices are colored by only one of the colors that are represented by integer numbers. Here a coloring is a simple assignment of colors to the vertices of the graph. A tropical path P is a path in G, $(v_1, v_2, \ldots, v_k)$, where each color of the initial graph appears at least once in P. The problem consists of finding a path with the biggest number of colors used by the original graph. One tropical path clearly will be an optimum solution to this problem.

Input

An input represents only a test case. The first line contains two integer values N and K that represent the number of vertices and the number of colors, respectively. Each of the following N lines contains an integer value $c(0 \leq c < K)$ which represents the color assigned to the vertex. The next line has another integer A that represents the number of edges of the graph, and finally, each of the following A lines contains a pair of integer values x and y $(0 \leq x < N; 0 \leq y < N)$ that represents the ends of the respective edge. The entry must be read from the standard entry.

The input must be read from the standard input.

Output

The output contains two lines. The first line is a sequence of integer values, separated by the symbol: "-", which represent the vertices of the path. The returned path is such that it has as many colors as possible. The second line contains an integer value that represents the number of colors present in the path.

The output must be written to the standard output.

Example

Input	Output
4 3 0 1 2 1 6 0 1 0 2 0 3 1 2 1 3 2 3	0 1 2 3

Problem B
Recursive Quicksort of Positive Integers

From Wikipedia:

> Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm. Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

Your job is to create a parallel solution for the proposed quicksort algorithm.

Input

The first and unique argument for the program is the number of positive integers that should be generated to run the code. The seed of the random number generator is set to zero to generate reproducible cases for the same case size. The program will compute the MD5 checksum of the integers after quicksort to guarantee the correctness of the algorithm.

The input must be read from the standard input.

Output

The output contains just one line. The program will output the MD5 checksum of the sorted array.

The output must be written to the standard output.

Example

Input	Output
`10000000`	`0f37a269c52bc42856f5acadd51bd05a`

Problem C
Jacobi linear solver method

Several problems require a linear system solution. In doing so, some numerical methods can be used to resolve linear system. One of useful is named Jacobi method.

Thus, given a linear system:

$$\begin{cases} a_{11}\cdot x_1 + a_{12}\cdot x_2 + \cdots + a_{1n}\cdot x_n = b_1 \\ a_{21}\cdot x_1 + a_{22}\cdot x_2 + \cdots + a_{2n}\cdot x_n = b_2 \\ \vdots \quad\quad \vdots \quad\quad \ddots \quad\quad \vdots \quad\quad \vdots \\ a_{m1}\cdot x_1 + a_{m2}\cdot x_2 + \cdots + a_{mn}\cdot x_n = b_n \end{cases}$$

Jacobi method re-writes as following:

$$\begin{cases} x_1^k = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{k-1} - a_{13}x_3^{k-1} - \cdots - a_{1n}x_n^{k-1}) \\ x_2^k = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{k-1} - a_{23}x_3^{k-1} - \cdots - a_{2n}x_n^{k-1}) \\ x_3^k = \frac{1}{a_{33}}(b_3 - a_{31}x_1^{k-1} - a_{32}x_2^{k-1} - \cdots - a_{3n}x_n^{k-1}) \\ \vdots \\ x_n^k = \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{k-1} - a_{n3}x_3^{k-1} - \cdots - a_{nn-1}x_{n-1}^{k-1}) \end{cases}$$

For each step k, new values of x^{th} is found out. This procedure is repeated until the method converge. Which means, x^k is equal to x^{k-1}.

Input

The input has a binary format.

The input must be read from the standard input.

Output

Output is two integers: the first one is the number of time steps necessary to solver a linear system. The second line informs if the linear system is correct. Where correct is 1 and failure is 0.

The output must be written to the standard output.

Example

Input	Output
binary format	`6` `1`

Problem D
All Permutations, Sorted

A permutation is any arrangement of the elements of a finite multiset into a ordered sequence. *E.g.*, there are three distinct permutations of the finite multiset $\{B, O, B\}$, namely (B, B, O), (B, O, B), and (O, B, B). A given string and its anagrams are distinct permutations from the multiset formed by the string's symbols.

Two permutations are defined to be equal *iff* their elements are sorted in the same order and to be distinct otherwise. For a given finite multiset of size n the number of all possible permutations is $n!$, while the number of distinct permutations is $n!/\Pi(n_i!)$, where n_i is the multiplicity of element i in the multiset. *E.g.*, $\{B, O, B\}$ has $n = 3$ and, thus, $3! = 6$ possible permutations and $n!/n_B! \cdot n_O! = 3!/2! \cdot 1! = 3$ distinct permutations.

The problem to be solved is: given one string, to generate all its distinct permutations sorted in lexicographic (*a.k.a.* alphabetic) order.

The proposed sequential solution is an implementation of an in-place algorithm based on continuasly finding the next lexicographic permutation, as long as one is found. It can handle repeated values, for which case it generates each distinct multiset permutation only once. For a given input string $s = (s_1, \ldots, s_n)$, its description is as follows.

1. **Find the minimal lexicographic permutation.** Sort s in weakly increasing order, which gives the its lexicographically minimal permutation.
2. **Find longest non-increasing suffix.** Find the largest index k such that $s_k < s_{k+1}$. If no such index exists, the permutation is the last permutation.
3. **Find the rightmost successor.** Find the largest index l greater than index k such that $s_k < s_l$.
4. **Swap.** Swap the value of a_k with that of a_l.
5. **Reverse.** Reverse the sequence from s_{k+1} up to and including the final element s_n.

This method uses about 3 comparisons and 1.5 swaps per permutation, amortized over the whole sequence, not counting the initial sort.

Input

The input must be read from the standard input. It is a single string whose distinct permutations (ignoring whitespaces) will be generated.

The input must be read from the standard input.

Output

The input must be written to the standard output. It is a list of all permutations of the input string's symbols in lexicographic order.

The output must be written to the standard output.

Example

Input	Output
BILL	[B I L L] [B L I L] [B L L I] [I B L L] [I L B L] [I L L B] [L B I L] [L B L I] [L I B L] [L I L B] [L L B I] [L L I B]

Problem E
Yet More Primes

Prime test is a common task in many applications, specially for security, cryptography, and other interest areas. There are a number of techniques used to verify if a number is prime, and the common problem is that all techniques consumes lots of time.

The problem is easy to solve, giving the number to be tested and time enough to calculate. But a weird scientist took our numbers, and cut them in two halves, so we have first to re-arrange the numbers, looking for the primes. In this case, we have the certainty that all the numbers we have are primes, but the problem is how to re-order them

Input

In the first line, there is the number P of prime numbers $(1 \leq P \leq 1000)$. Then, $2P$ lines with parts of the numbers to be tested, with no order. Prime numbers in this problem are taken as strings, so first half and second half simply means an arbitrary cut of such string. For instance, prime number 504155039 can be cut in 5041 and 55039, or 504155 and 039

Note that second half numbers could begin with a lead zero, so you must take it into account. Of course, this doesn't happen with the first half.

The input must be read from the standard input.

Output

You have to print the list of the P prime numbers, in ascendant order.

The output must be written to the standard output.

Example

Input	Output
4 5039 50415 100 55750 01 0213 55713 5041	1000213 5575001 504155039 504155713

Problem F
Graph Search on a DAG

Several problems in Computer Science can be modeled using graphs. For example, in scheduling algorithms, dependent tasks can be expressed as nodes, while an edge represents a dependency relation between two nodes. Such a strategy enables the scheduler to easily identify independent tasks, i.e., that can execute in parallel. The given sequential code implements a graph search on a DAG (Directed Acyclic Graph). The goal is recursively computing a maximal value starting from a given node. The algorithm works as follows:

- starting at a given node, check all neighbors and continuous the search at the neighbor with the **highest** value;
- for each node, the search returns the sum of the current node value plus the visited neighbor's computed sum.

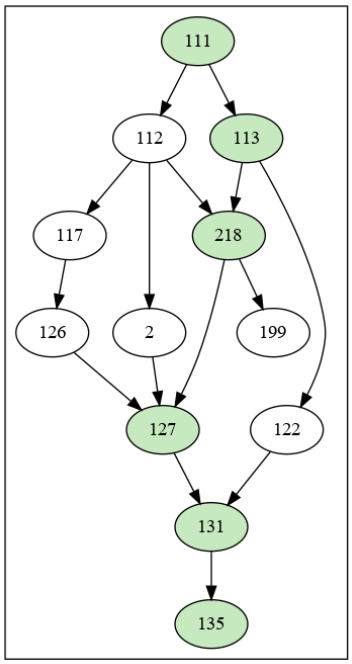

Figure F.1: when starting at node 135, the answer is 835.

Input

The first line contains the mandatory **header**, which has two parts:

- the first value is an integer describing the number of entries in the file;
- the following values are a list of values to be computed (separated with spaces).

The remaining lines have the following structure:

- the first value is the node ID (integer)
- the second value represents the node value (float)
- the third value is a list of variable size containing node neighbors IDs.

The input must be read from the standard input.

Output

The output contains one line per value in the list of values to be computed. Each line has the node ID and the calculated value for this node. The first and second values are separated by ": ".

The output must be written to the standard output.

Example

Input	Output
30 94 89 65 43.711854 19 66 56.506839 65 22 67 56.547957 65 25 68 56.500064 65 28 69 57.009560 65 31 70 73.114013 69 66 34 71 73.113959 69 67 37 72 85.821312 69 68 40 73 69.615694 65 43	94: 959.023438 89: 456.091675