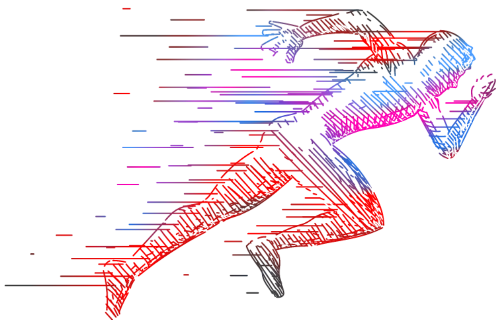
STUDENTS@SC22

Parallel Programming Marathon



SC22 Dallas, TX | hpc accelerates.

Instructions

- You have five problems to parallelize! You can choose how much effort you put into each.
- Your solution will be tested for correctness, but there might be more test cases than the sample.
- How are solutions scored? We will compute the speedup of your solution (sequential/parallel version). You can tune anything! But the output should be the same!
- For each problem, we will take the best submission to score. The final score is the sum of all the best scores. The (correct) submission time will be used to break ties (the sooner, the better)

You can fetch the pack for each solution at: `https://github.com/jeanbez/ppm`

Make sure you load any additional modules and tune your submission script based on the number of nodes and ranks you want to use. Using more nodes than the given limit will cause the submission to fail.

You are going to submit your solution as a `.tar.gz` file. It should contain the following structure:

```
B-username.tar.gz
└─problem
   ├─B
   ├─Makefile
   └─...
```

In this example for problem B, notice that the name is used in the .tar.gz file. It should be the problem letter in uppercase, a dash, and your username. It should have a first-level directory that contains a bash script with the problem letter and the Makefile provided for compilation.

```
#SBATCH --job-name=ppm
#SBATCH --qos=debug
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --constraint=haswell
#SBATCH -o ppm-%J.out
#SBATCH -e ppm-%J.err
#SBATCH --reservation=marathon
```

Cori – Haswell Compute Nodes[1]

- Each node has two sockets
- Each socket is populated with a 2.3 GHz 16-core Haswell processor Intel Xeon Processor E5-2698 v3
- Each core supports 2 hyper-threads and has two 256-bit-wide vector units
- 36.8 Gflops/core (theoretical peak)
- 1.2 TFlops/node (theoretical peak)
- 2.81 PFlops total (theoretical peak)
- Each node has 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket)

Cori – GPU Compute Nodes[2]

- Each node has two sockets of 20-core Intel Xeon Gold 6148 ('Skylake') @ 2.40 GHz
- 384 GB DDR4 memory (372 GB available to user programs)
- 1 TB on-node NVMe storage (930 GB available to user programs)
- 8 NVIDIA Tesla V100 ('Volta') GPUs, each with with 16 GB HBM2 memory
- Connected with NVLink interconnect
- 4 dual-port Mellanox MT27800 (ConnectX-5) EDR InfiniBand HCAs

[1]https://docs.nersc.gov/systems/cori/#haswell-compute-nodes
[2]https://docs-dev.nersc.gov/cgpu/hardware

A Brute-Force Password Cracking

8 nodes — Timelimit: 5 minutes

One way to crack a password is through a brute-force algorithm, which tests all possible password combinations exhaustively until it finds the correct one.

A common practice in security systems is to store the *hash* of user passwords (instead of the plain text). Hashes map a given x (the password) with variable length N to a given $y = hash(x)$ with a fixed length M, following specific security properties. One of them is: you can not obtain the value of x from the value of y. In a security system, y is stored instead of x, and the comparison is performed directly between the hash of attempted password x' and the stored $y = hash(x)$.

Examples of hash algorithms are MD5 and SHA. For MD5 algorithm, the fixed length of the $hash(x)$ is 128 bits, usually presented as 32-hexadecimal characters.

This problem considers the cracking of MD5-hashed passwords by trying all the combinations of possible characters (uppercase and lowercase letters and numeric symbols) exhaustively and comparing pairs of MD5 hashes. That is, for all combinations of characters x'_i, $hashMD5(x) = hashMD5(x'_i)$?

Input

Input consists of only one case of test. The single line contains a string with 32-hexadecimal characters representing the value of MD5 hash of the password to be cracked.

Consider that the possible passwords used to generate the hashes have the length N, with $1 \leq N \leq 10$.

The input must be read from the standard input.

Output

The output is a single line that contains the password value found.

Sample Input	**Sample Output**
7a95bf926a0333f57705aeac07a362a2	found: PASS

B Graph Isomorphism Detection

16 nodes — Timelimit: 5 minutes

Given two graphs $G(V, E)$ and $H(V, E)$, where V denotes the vertices and E the edges, the problem of finding isomorphic graphs consists of determining a bijection function $f : V(G) \rightarrow V(H)$ that maps the vertices of both graphs while preserving the adjacency condition. In other words, for the isomorphism property to exist, if any two vertices $u, v \in G$ are adjacent, then the corresponding vertices $f(u), f(v) \in H$ must also be adjacent. In this sense, the proposed algorithm detects all the possible isomorphic relations between two undirected graphs through extensive permutations, comparing all possible vertex correspondences between G and H.

Input

An input represents only a test case. Initially, the graph G is informed, followed by H. The first line contains an integer V_g $(0 < V_g < 100)$ representing the number of vertices in G, while the second line informs an integer E_g $(0 \leq E_g < (V_g * (V_g - 1))/2)$ denoting the number of edges in G. Following, E_g edges are sequentially informed, each denoted by two integers $u, v \in V_g$. The next lines repeat the same rationale for informing the graph H: an integer V_h $(0 < V_h < 100)$ representing the number of vertices in H is followed by an integer E_h $(0 \leq E_h < (V_h * (V_h - 1))/2)$ denoting the number of edges in H. Finally, E_h edges are sequentially informed, each denoted by two integers $u, v \in V_h$.

The input must be read from the standard input.

Output

The output contains N lines. If there is no possible isomorphic relation between G and H, only the message *The graphs are not isomorphic* is printed to the output. Otherwise, a number of lines equal to the amount of possible isomorphisms is printed, each containing a stream of integers $v \in V_h$, which represents a vertex correspondence between G and H, such that $f(u) = v$, with $u \in V_g$ equal to the position of each v on the stream. Thus, the order in which vertices are printed in each solution line defines the solution itself.

The output must be written to the standard output.

Sample Input	Sample Output

4
6
0 2
1 1
1 3
2 2
2 3
3 3
 	
4
6
0 0
0 1
0 2
1 1
1 3
3 3
	2 3 0 1

Sample Input	Sample Output

4
6
0 3
1 1
1 3
2 2
2 3
3 3
 	
4
6
0 0
0 1
0 2
1 1
1 3
3 3
	The graphs are not isomorphic

C Recursive Quicksort of Positive Integers

16 nodes — Timelimit: 5 minutes

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 random number generator's seed 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

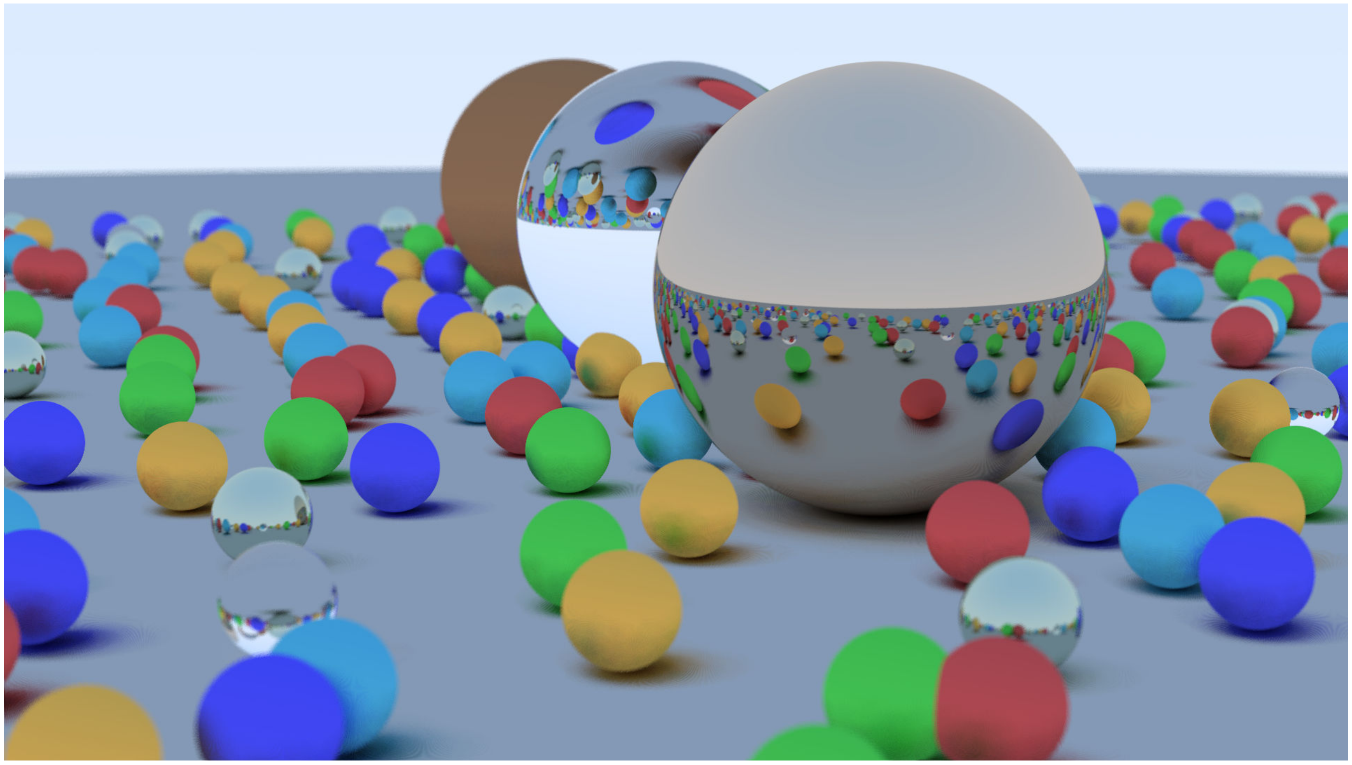
The output contains just one line. The program will output the MD5 checksum of the sorted array.

The output must be written to the standard output.

Sample Input	**Sample Output**
10000000	0f37a269c52bc42856f5acadd51bd05a

D Ray Tracing

8 CPU nodes — 1 GPU node (8 GPUs) — Timelimit: 5 minutes



The ray tracing technique models the transport of light between different objects and light sources. Its basic concept is: considering an observer (camera), one can calculate the colors of each pixel of the visualization considering the arrival of n rays of light per pixel. Each ray of light is calculated in reverse from nature (starting from the camera and arriving at a light source). When a ray strikes an object, various properties are applied considering its material and the recursive arrival of other rays of light at that point, creating a light contribution between various objects.

The source code[3] renders a scene of $N \times M$ pixels with several spheres and three different materials. The final color of each pixel is the accumulation of n rays of light that are calculated independently. In this program, when a ray strikes an object, the color of this point is given by the object's material accumulated with a new ray starting from this point in a new direction. If a ray does not hit an object, it takes on the color of ambient light (the ray came from the sun, for example). While calculating a color, if the depth of new rays reaches a certain limit, the new ray is assigned the black color (not arrived at a light emitter).

Input

The input is composed of 2 integers. The first is the horizontal size of the image in pixels N, and the second is the number of rays per pixel r to be used by Ray Tracing.

Output

The output is an image described in P3 format. The first line contains the text "P3". The second line contains two integers, the horizontal N and vertical M image size in pixels. The third line contains the number 255. The next $N \times M$ lines contain 3 integers (each between 0 and 255) describing the RGB color of the pixel. The P3 file can be opened on gnome-based distributions with the eog program.

[3]Original code developed by Peter Shirley and available at https://github.com/RayTracing/raytracing.github.io/ under CC0 license. Adapted for the Parallel Programming Marathon, removing the use of random numbers to make it deterministic across multiple platforms.

Sample A Input

```
600 1
```

Sample A Output

```
P3
600 337
255
220 235 255
220 235 255
220 235 255
...
132 153 181
132 153 181
132 153 181
```

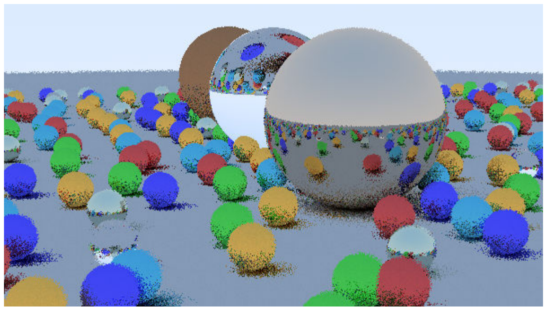
Sample B Input

```
1200 10
```

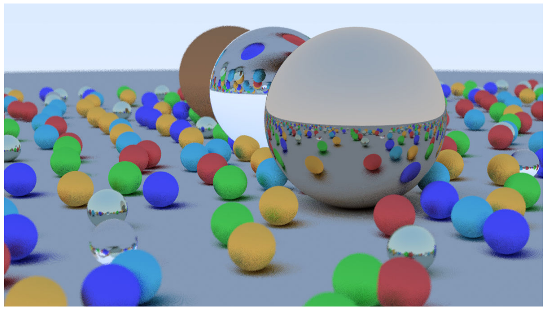
Sample B Output

```
P3
1200 675
255
220 235 255
220 235 255
220 235 255
...
132 153 181
132 153 181
132 153 181
```

Visualization Sample A



Visualization Sample B



E Galaxy Simulator

16 nodes — Timelimit: 5 minutes

From Wikipedia:

> Newton's law of universal gravitation states that every particle attracts every other particle in the universe with a force which is directly proportional to the product of their masses and inversely proportional to the square of the distance between their centers.

Newton's law leads up to an $O(n^2)$ time complexity algorithm since the force of every particle must act upon all other particles when one tries to simulate a particle system. Such an algorithm is too much slow to simulate a large number of particles, such as galaxies that are commonly composed of billions of stars. For example, the Milky Way galaxy alone is supposed to have between 100 and 400 billion stars.

Barnes-Hut[4] came up with an $O(nlogn)$ time complexity algorithm that enables one to approximate particle interaction by assuming that nearby bodies, a group, work as a single large fictitious body in the center of mass of the group. For a 2D galaxy simulation (where all stars share the same plane), the Barnes-Hut algorithm divides the space into four quadrants. Depending on the number of stars, each quadrant can be divided again into four quadrants, and so on. A tree data structure is used to keep all information: each node has four children, each one representing a quadrant of the two-dimensional space associated with the node. The particle interaction uses that tree data structure to compute approximations by considering the center of mass of each quadrant.

Input

An input represents only one test case. The first line contains one integer value N representing the number of stars. Each one of the following N lines contains five floating-point values that describe the mass, the plane position (x, y), and the plane velocity (u, v) of each star. The last line contains the number of time steps that is used in this galaxy simulation.

The input must be read from the standard input.

Output

The output contains two lines: the first line contains the total vector velocity (u, v) considering all stars, in both dimensions; and the second line contains the center (x, y) of mass considering all stars.

The output must be written to the standard output.

Sample Input	Sample Output

3
1.000000 0.334531 0.532345 -5.192624 -6.640964
1.000000 0.558670 0.453306 3.576856 1.123553
1.000000 0.573755 0.553912 -4.322105 1.478229
1000
	-13.109914 -6.611076
	0.509590 0.580083

[4] Josh Barnes and Piet Hut. A Hierarchical O(n log n) Force-calculation Algorithm. Nature, 324(6096):446, 1986.