12th Marathon of Parallel Programming

SBAC-PAD & WSCAD – 2017

October 18th, 2017.

Rules for Local 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.

All *Local Teams* have access to the target machine during the marathon. Your execution may have concurrent process from other teams. Only the judges have access to a non-concurrent environment.

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 5 problems; pages are numbered from 1 to 11.

General information

MPI

You must use *aprun -q* instead of *mpirun* inside your scripts:

```
aprun -q -n <number of process> [-N <process per node] [-d <threads per
process>] <program name>
```

You have 4 nodes with 2x18 cores. Examples:

```
# 144 processes automatically distributed between all 4 nodes
aprun -q -n 144 ./hello

# 72 process, 18 processes per node, 4 nodes
aprun -q -n 72 -N 18 ./hello

# 32 process, 8 processes per node, each process runs 2 OpenMP threads
OMP_NUM_THREADS=2 aprun -q -n 32 -N 8 -d 2 ./hello
```

Compilation

You must use CC or CXX inside your *Makefile*. Do not redefine them! Example:

```
FLAGS=-O3
EXEC=sum

all: $(EXEC)

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

Test machine (for *local teams*)

See the URL

https://wickie.hlrs.de/platforms/index.php/CRAY_XC40_Using_the_Batch_System to learn how to use the batch system at CRAY XC40.

Problem A

Transitive Closure

Let $G = (V, E)$ be an unweighted graph defined as a finite set V of nodes and a set E of edges, which are pairs of nodes. Given a directed graph G and two nodes s, $v \in V$, the reachability problem is related to find out whether there is a path from s to v. The generalization of the reachability problem is called Transitive Closure problem (TC). The solution of every reachability problem applied to a distinct vertex of a graph is the transitive closure of the own graph.

The transitive closure is based on finding if a vertex s is reachable from another vertex v for all vertex pairs (s, v). Thus, the transitive closure of a graph G is a graph that contains an edge (s, v) whenever there is a directed path from s to v in G. The transitive closure problem can be solved by different graph algorithms that use several techniques, such as: search algorithms, shortest paths algorithms, algorithms that find out strongly connected components of a graph and so on.

Create a parallel version of an algorithm that generates the transitive closure of a given graph.

Input

The input follows the GTgraph format. The lines starting with "c" are comment lines containing information about the graph. The problem line, starting with "p", is unique and must appear as the first non-comment line. This line has the format "p sp n m", where n and m are the number of nodes and the number of arcs (edges), respectively. Arc descriptors are of the form "a U V W", where U and V are the tail and the head node ids, respectively, and W is the arc weight.

The input must be read from the standard input.

Output

The output must have only the adjacency matrix. Columns separated by white space and each line ending with a newline ("\n").

The output must be written to the standard output.

Example

Input	Output for the input
c FILE : graph_5.gh c No. of vertices : 5 c No. of directed edges : 9 c Max. weight : 1 c Min. weight : 1 c A directed arc from u to v of weight w c is represented below as ' a u v w ' p sp 5 9 a 1 3 1 a 1 4 1 a 1 5 1 a 2 5 1 a 3 1 1 a 3 2 1 a 3 5 1 a 4 1 1 a 5 4 1	0 0 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 1 0

Problem B

Eternity II

The Eternity II puzzle[1] was released in 2007 with the promise to pay $2 million to the first person to present a complete solution. However, up until today no correct solution was presented and the prize remains unclaimed.

Eternity II is a classic edge-matching puzzle which involves placing 256 square tiles into a 16×16 grid. Each tile has its edges marked with different shape/color combinations (which we will simply call color here). The tiles must be placed in such a way that all the colors on their edges precisely match the colors of the adjacent tiles. The borders of the grid are a special case and match only tiles with gray edges. Tiles can be rotated; therefore, each tile has 4 possible placements for each grid position. There are 22 colors, not including the gray edges. On the original puzzle, the center tile is pre-determined and some tile positioning hints are given.

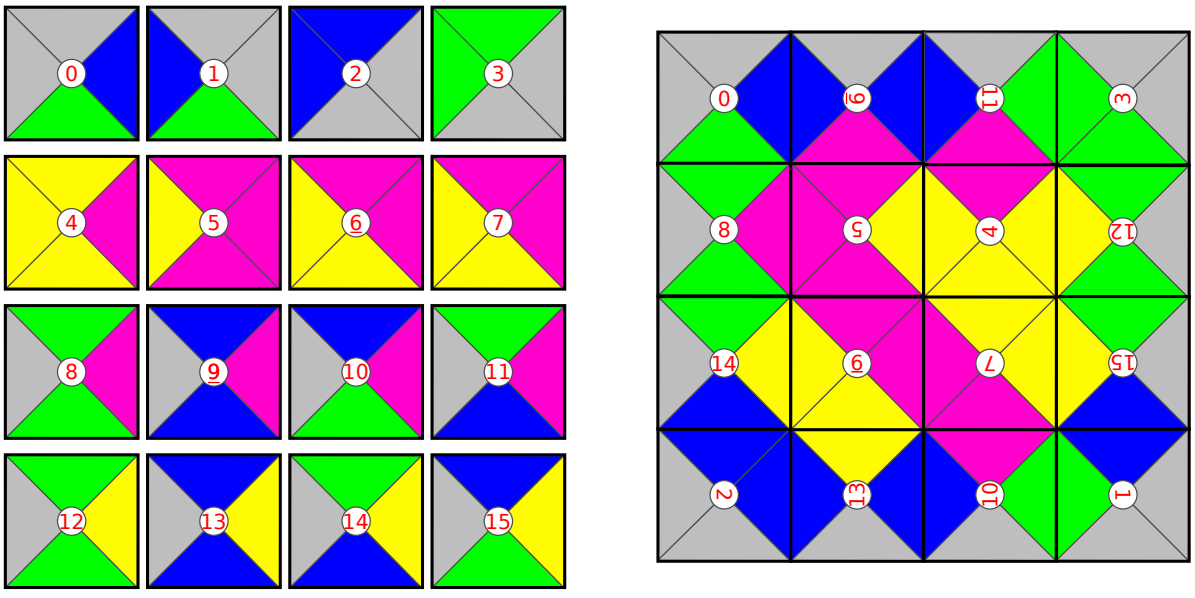


Figure B1. On the left, a set of tiles for a 4×4 puzzle. On the right the solved puzzle. Notice how this solution rotates some of the tiles.

This puzzle was designed to be difficult to solve by brute-force computer search, and remains intractable on its original configuration. Indeed, the number of possible configurations

[1] The game description was adapted from https://en.wikipedia.org/wiki/Eternity_II_puzzle

(assuming all the pieces are distinct, and ignoring the fixed pieces and tile positioning hints) is 256! × 4^{256} roughly \$1.15 × 10^{661}. A tighter upper bound can be obtained taking into account the fixed tile in the center and the positioning hints yielding a search space of 3.11×10^{545}.

Since this competition must end before the end of the universe, here we will deal which much smaller instances of the same problem. However, to keep things interesting we will not provide any hints or tiles with predefined positions. The puzzle grid size, number of colors and tiles will be given through the standard input and the solution should be presented using the standard output.

You were given a sequential version of a solver which uses a naïve brute-force backtracking method. Your task is to write a parallel version of this code. Feel free to use any heuristic or method to improve the performance of the sequential version. Notice, however, that it might be the case that a single input has multiple distinct correct solutions. This will not be a problem as long as the solution provided by your code is correct since the automated evaluation system already takes this into consideration.

Input

Each input contains one puzzle. It consists of a list of integers separated by spaces and new lines. The first line contains 2 integers: the grid size g and the number of colors c. The next g^2 lines list the tiles. The order of the tiles is important (it will be used for the output) and is counted from 0, thus tiles are numbered from 0 to $g^2 - 1$. Each tile is given by 4 integers between 0 and $c - 1$ describing the colors of its edges in clockwise order, starting from the top edge. The color 0 (gray) is considered to be a special case: the only acceptable color for the borders. The example input below represents the input tiles shown in Figure B1(a) You may assume $g \leq 16$ and $c \leq g$.

The input must be read from the standard input.

Output

The expected output must have g^2 lines, each one representing one of the cells of the grid. The order of the lines follows the grid from left to right, top to bottom. Each line is composed by 2 integers, the first indicates the tile number, and the second the number of clockwise

rotations needed for that tile. The expected output corresponding to the solution presented in Figure B1(b) is shown in the next section.

The output must be written to the standard output.

Example

Input	Output for the input
4 5 0 1 2 0 0 0 2 1 1 0 0 1 2 0 0 2 3 4 3 3 4 4 4 3 4 4 3 3 4 4 3 3 2 4 2 0 1 4 1 0 1 4 2 0 2 4 1 0 2 3 2 0 1 3 1 0 2 3 1 0 1 3 2 0	0 0 9 1 11 1 3 3 8 0 5 2 4 3 12 2 14 0 6 0 7 2 15 2 2 1 13 3 10 3 1 1

Problem C

Mandelbrot Set

The Mandelbrot set is the set of complex numbers *c* for which the function $f_c(Z) = Z^2 + c$ does not diverge when iterated from $Z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, *etc.*, remains bounded in absolute value. Its definition and name are due to Adrien Douady, in tribute to the mathematician Benoit Mandelbrot. The set is connected to a Julia set, and related Julia sets produce similarly complex fractal shapes. The Mandelbrot set is the set of values of *c* in the complex plane for which the orbit of *0* under iteration of the quadratic map

$$Z_{n+1} = Z_n^2 + c$$

remains bounded. That is, a complex number *c* is part of the Mandelbrot set if, when starting with $Z_0 = 0$ and applying the iteration repeatedly, the absolute value of Z_n remains bounded however large *n* gets.[2]

Create a parallel version of the given sequential algorithm[3] that generates a textual approach for the Mandelbrot set.

Input

The first line informs the maximum number of rows. The second line presents the maximum number of columns. Finally, the last one informs the number of iterations.

The input must be read from the standard input.

Output

Textual representation of the Mandelbrot set.

The output must be written to the standard output.

[2] Text from Wikipedia: https://en.wikipedia.org/wiki/Mandelbrot_set

[3] Code adapted from http://www.fractalforums.com/programming/mandelbrot-with-only-18-lines-of-cplusplus-code!/

Example

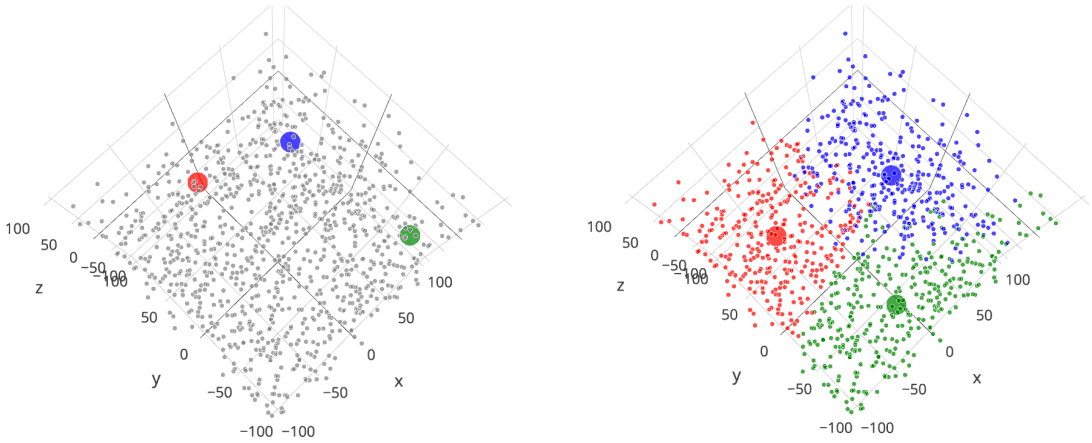
Input	Output for the input
23 79 24	

Problem D

K-Means Clustering

K-Means clustering is a method that allows the modeling of probability density functions by the distribution of prototype vectors and is popular for cluster analysis in data mining. The method will partition a set of *n* observations (points) into *k* clusters, where each point will be associated with the cluster with the nearest mean. The Euclidean distance is usually the adopted metric for proximity.

Clustering is a NP-hard problem, but there are efficient heuristic algorithms that can quickly to a local optimum. In this implementation, the application takes as input the coordinates (3D) of *k* initial centroids and a set of data points. K-Means performs an iterative process, where points are re-clustered according to the minimum Euclidean distance between them and the centroids. Next, the centroid of each partition is recalculated taking the mean of all points in the partition, and the whole procedure is repeated until no centroid is changed and no points are assigned to another cluster. Upon completion, the algorithm returns the coordinates of the final *k* centroids.



(a) initial centroids (b) re-clustered centroids

Figure D1. K-Means example with 3 centroids and 1,000 points on $\mathbb{R}^3$.

Figure D1 provides an example of K-Means in with 3 centroids and 1,000 points on $\mathbb{R}^3$. Figure D1(a) shows the initial position of the centroids (big points in red, green and blue), while Figure D1(b) shows the final position of the centroids and points associated with each cluster (with the same color of their respective centroids).

Write a parallel version of the given sequential program. We are not interested in other clustering methods; hence you should not change this aspect of the application.

Input

The first two lines with contain the number of centroids *k* and the number of points *n*. The next *k*+*n* lines will contain (*x*, *y*, *z*) coordinates for *k* centroids and *n* points, respectively.
The input must be read from the standard input.

Output

The program will print *k* lines containing (*x*, *y*, *z*) coordinates of the re-clustered *k* centroids.
The output must be written to the standard output.

Example

Input	Output for the input
3 9 -2.0 0.0 1.5 1.0 1.0 1.0 -1.0 -1.0 0.0 -1.2 0.6 0.8 1.5 0.5 1.8 0.5 -1.0 0.8 -0.5 1.2 0.7 1.0 1.5 -0.8 -1.2 0.2 0.5 0.2 1.0 0.2 -0.2 -1.0 1.8 0.5 -1.5 1.5	-0.97 0.67 0.67 0.90 1.00 0.40 0.27 -1.17 1.37

Problem E

Shortest Superstring

The shortest superstring problem belongs to the NP-Hard class and is defined as follows: given a set of strings *S* where no element is substring of another element, find the shortest string *s* that contains each string in *S* as substring. Formally, let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of strings such that

$$(\forall s_i, s_j \in S) s_i \nsubseteq s_j.$$

Thus, the shortest substring problem is to find a string

$$s \in S' = \{s' | (\forall s_i \in S) s_i \subseteq s'\} \text{ such that } (\forall s' \in S') |s| \leq |s'|.$$

For instance, consider the following set of strings:

S = {`CATGC`, `CTAAGT`, `GCTA`, `TTCA`, `ATGCATC`}.

The shortest string that contains all the above strings as a substring is

s = `GCTAAGTTCATGCATC`.

The proposed solution employs a greedy strategy over an operation called “overlap”, described next.

The *overlap* operation over strings *a* and *b* is their concatenation *ab* where the matching parts in the suffix of *a* and the prefix of *b* are merged. For instance, if *a* = {`ABC`} and b = {`BCDE`}, the left overlap of *a* and *b* is {`ABCDE`}. The overlap value of the operation is the size of the corresponding suffixes/prefixes of both strings. In the example, the overlap value is 2 (`BC`). Order matters; the overlap operation and its respective overlap value are not commutative.

With the overlap operation defined we can now show the greedy algorithm that solves the problem at hand. Its idea is simple; at each step, we replace the two strings whose resulting overlap value would be the highest by the result of their overlap. At the end, we have the shortest superstring:

1. Let *T* be a copy of *S*.
2. While |*T*| > 1 do:
    a. let *a* and *b* be the two strings that yield the highest overlap value;

b. pop *a* and *b* from T and insert the string obtained by overlapping *a* and *b*.

Now the only element of *T* is the shortest superstring containing as substrings all strings of *S*.
It is important to notice the overlap of strings *x* and *y* is probably not the same as the overlap of strings *y* and *x* because of the its aforementioned lack of commutativity. Thus, both configurations must be evaluated for each pair of strings taken into account and its order preserved.
You shall submit a parallel implementation of an algorithm that solves the shortest superstring problem.

Input

The first line contains the number *n* of strings to be read and processed. Each of the following *n* lines of the input contains a string at most 256 ASCII characters. All the characters are readable and there are no blanks.
The input must be read from the standard input.

Output

It is a single line containing the shortest superstring that contains all strings from the input.
The output must be written to the standard output.

Example

Input	Output for the input
5 CATG CTAAGT GCTA TTCA ATGCATC	GCTAAGTTCATGCATC