

# 16<sup>th</sup> Marathon of Parallel Programming SBAC-PAD & WSCAD – 2021

Calebe Bianchini<sup>1</sup> and Maurício Aronne Pillon<sup>2</sup>

<sup>1</sup>Mackenzie Presbyterian University

<sup>2</sup>Santa Catarina State University

## 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 5 problems; pages are numbered from 1 to 9.**

# General Information

## Compilation

You should use **CC** or **CXX** inside your *Makefile*. Be careful when redefining them! There is a simple *Makefile* inside your 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	command
host	GCC 10.2.11 ( <i>default</i> )	C = gcc C++ = g++
	Intel C/C++ Compiler 19.1.3.304	C = icc C++ = icpc
MPI	Intel MPI Compiler 2019u12 ( <i>default</i> )	C = mpiicc C++ = mpiicpc
	Intel C/C++ Compiler 19.1.3.304	C = icc C++ = icpc
	GCC 10.2.11	C = gcc C++ = g++
gpu	NVidia CUDA 11.2 ( <i>default</i> )	C = nvcc C++ = nvcc
	GCC 8.3.1	C = gcc C++ = g++

## 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 your 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/mpicpc*. 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

## Graph Isomorphism Detection

### Problem definition

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 isomorphic 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 a single line. If  $G$  and  $H$  are not isomorphic graphs, the message *The graphs are not isomorphic* is printed to the output. Otherwise, a stream of integers  $v \in V_h$  presents 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.

*The output must be written to the standard output.*

---

## Example

Input example	Output example
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

---

## Problem B

# Descriptive Statistics Metrics

A researcher has a set of samples grouped in matrices, where each column represents a sample. The researcher needs to calculate the following values for each sample (column): arithmetic mean, harmonic mean, median, mode, sample variance, standard deviation and coefficient of variation. Each value can be set as follows:

- **Arithmetic mean:** Sum of all elements in the sample, divided by the sample size (sum of the rows in the matrix column divided by the number of rows).
- **Harmonic mean:** Ratio between the sample size and the sum of the inverse of the samples.
- **Median:** Average element of the sample (average element of the ordered column). For an even number of elements, the median is the mean between the elements in the middle ( $\frac{\frac{n}{2} + \frac{n}{2+1}}{2}$ ).
- **Mode:** Most frequent number of the sample (the element that most appears in the column; if there is more than one, only the first is considered. If not, it returns -1).
- **Sample Variance:** Sum of squares of data deviations from the mean and dividing by the number of items minus one.
- **Standard deviation:** Square root of the variance.
- **Coefficient of variation:** Ratio between standard deviation and arithmetic mean.

### Problem input

The input has only one test case. The first line has the number of rows and columns of the matrix, separated by a single space. In the following lines are the elements of the matrix of type double, where the lines are separated by a single line break and the columns by a single space.

*The input must be read from the standard input.*

### Problem output:

The output contains the calculated metrics, respectively. A single space will separate the calculated values for each column, and there will also be a single space at the end of the line. Each metric starts on a new line, i.e., is separated by a single line break. Printed values consider a single decimal place.

*The output must be written to the standard output.*

---

## Example

Input example	Output example
6 4 9 8 4 5 4 12 20 40 8 8 4 4 8 12 4 21 33 44 20 1 10 18 17 10	12.0 17.0 11.5 13.5 8.1 12.1 6.6 3.7 8.5 12.0 10.5 7.5 8.0 8.0 4.0 -1.0 110.0 188.4 68.7 217.9 10.5 13.7 8.3 14.8 0.9 0.8 0.7 1.1

---

# Problem C

## Sokoban

Sokoban is a videogame created in 1981 by Hiroyuki Imabayashi.

The board, whose dimension varies according to each stage, is divided into squares which can be one of the following elements:

- Empty square (that may be marked as a target)
- Wall
- Box
- Player

To organize the boxes, the player can move in 4 directions (up, down, left, and right) and also push the boxes. The player cannot, however, pull boxes. Therefore, if a box is pushed to a corner it can no longer be moved. Moreover, only a single box can be moved at the same time.

The goal of the game is to solve the puzzle by placing a box over each one of the target squares. The secondary objective is to solve the puzzle in the least number of movements possible.

### Input

The input contains a single puzzle stage to be solved. The input follows the format below in which each character represents a square of the board:

- Empty squares are represented by spaces
- # represents a wall
- \$ indicates the presence of a box
- . is a target square
- \* is a box over a target square
- @ is the player
- + is a target square

*The input must be read from the standard input.*

### Output

The output of your program is the sequence of movements made by the player to solve the puzzle. The output format is the following:

- u - up
- d - down
- l - left
- r - right
- U - up, pushing a box
- D - down, pushing a box
- L - left, pushing a box



- R - right, pushing a box

The solution length must be the shortest possible. Since there might be more than one possible solution of the same length for the same input, only the first solution ordered by the movements of the player (considering the order up, down, left, and then right) must be shown. For instance, if the result of the sequences d11 and 11d is the same, then d11 is preferred.

*The output must be written to the standard output.*

### Example

Input	Output
<pre>##### #       # #       # # . #   # # . \$\$  # # . \$\$  # # . #   @# #####</pre>	<pre>ulULLulDDurrrddlULrruLLrrUruLLlulD</pre>

---

## Problem D

# Johnson's Algorithm for SSSP

Let  $G = (V; E; w)$  a weighted directed graph with non-negative weights, where  $V = v_1, v_2, \dots, v_n$  denote the vertices,  $E = e_1, e_2, \dots, e_m$  denotes the edges and  $w$  denotes the weight function  $w : E \rightarrow R$  which connects the different vertices in the graph with only positive integer. The problem of single-source shortest paths (SSSP) consists of finding the shortest paths from a source vertex  $v \in V$  to all other vertices in  $V$ .

Dijkstra's algorithm can solve the SSSP problem, however when the number of edges is much less than the possible number of edges or mathematically  $|E| \ll V^2$ , Johnson's algorithm obtains better results. Johnson's Algorithm updates a heap structure while the number of visited vertices is increased, this technique is known as "reweighting". With non-negative weights the Johnson algorithm is solve in three steps:

1. Add a new node  $x$  to the graph and connect this node with zero-weight edges to all other nodes.
2. The edges of the original graph are reweighted, *i.e.* the weight function is updated. For each edge  $(u, v)$ , assign the new weight as "original weight +  $h[u] - h[v]$ ".
3. Finally,  $x$  is removed, and Dijkstra's algorithm is used to find the shortest paths from the node  $v = 0$  to other vertices in the reweighted graph.

### Input

The input has only one test case. The input has three lines: first line is the number of vertices, second is the number of edges and the third is the mean Single-Source Shortest Paths using the vertex 0 as source for all the executions. The edges between the vertices are created using random functions with range  $1 \leq E \leq 10$  or  $[1, 10]$ . The seed of the random number generator is set to generate reproducible cases for the same case size.

*The input must be read from the standard input.*

### Output

The output a message with a single float number. The float number is the mean of the Single-Source Shortest Paths using the vertex 0 as source.

*The output must be written to the standard output.*

### Example

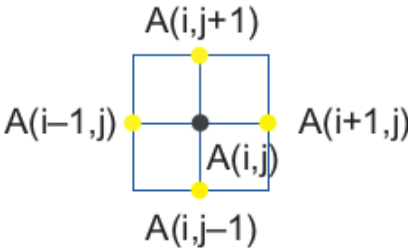
Input	Output
20 100 4.9000	Mean SSSP = 4.9000

---

# Problem E

## Jacobi Iterative Method that Solves the Laplace Equation

The code implements a Jacobi iterative method that solves the Laplace equation for heat transfer. The Jacobi iterative method is a means for iteratively calculating the solution to a differential equation by continuously refining the solution until the answer has converged upon a stable solution or some fixed number of steps have completed and the answer is either deemed good enough or unconverged. The example code represents a 2D plane of material that has been divided into a grid of equally sized cells. As heat is applied to the center of this plane, the Laplace equation dictates how the heat will transfer from grid point to grid point over time. To calculate the temperature of a given grid point for the next time iteration, one simply calculates the average of the temperatures of the neighboring grid points from the current iteration. Once the next value for each grid point is calculated, those values become the current temperature and the calculation continues. At each step the maximum temperature change across all grid points will determine if the problem has converged upon a steady state.


$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

The Figure E.1 below shows the result of applying the algorithm on a surface represented by a 512 x 512 matrix with 10000 steps.

### Input

The input consist of two values: the size of the square matrix that will be generated; and the maximum number of steps.

*The input must be read from the standard input.*

### Output

The output is a matrix that represents heat transferred from one point on the grid to another over time.

*The output must be written to the standard output.*

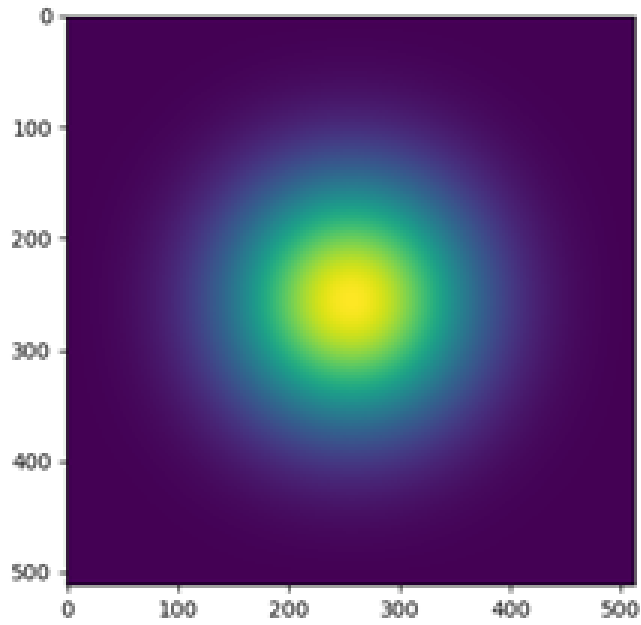


Figure E.1. 512 x 512 heated surface.

### Example

Input	Output
7 100	0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000003 0.000000 0.000006 0.000000 0.000003 0.000000 0.000000 0.000000 0.000009 0.000000 0.000009 0.000000 0.000000 0.000000 0.000006 0.000000 0.000013 0.000000 0.000006 0.000000 0.000000 0.000000 0.000009 0.000000 0.000009 0.000000 0.000000 0.000000 0.000003 0.000000 0.000006 0.000000 0.000003 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000