17^{th} Marathon of Parallel Programming
SBAC-PAD & WSCAD – 2022

Calebe Bianchini[1] **and Maurício Aronne Pillon**[2]

[1]Mackenzie Presbyterian University

[2]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 4 problems; pages are numbered from 1 to 5.

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	command
host	GCC 12.2.0 (*default*)	C = gcc C++ = g++
	Intel C/C++ Compiler 19.1.3.304	C = icc C++ = icpc
MPI	Intel MPI Compiler 2019u12 (*default*)	C = mpiicc C++ = mpiicpc
	Intel C/C++ Compiler 19.1.3.304	C = icc C++ = icpc
	GCC 12.2.0	C = gcc C++ = g++
gpu	NVidia CUDA 11.6 (*default*)	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 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 *mpiicc*/*mpiicpc*. 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
k-Nearest Neighbors Classifier

Given a set of two-dimensional points S divided intro n groups, an integer k and a two-dimensional point P to be classified, the k-Nearest Neighbors (KNN) algorithm can be used to solve the classification problem of the point P, i.e., classifying the point P into one of the n existing groups based on some criteria.

The KNN algorithm works by comparing the distances from each point $s \in S$ to the point P, then selecting the k points with the smallest distance values and counting the frequency of each group in these k points, the point P is then classified to belong to the group with the highest frequency among the k nearest points.

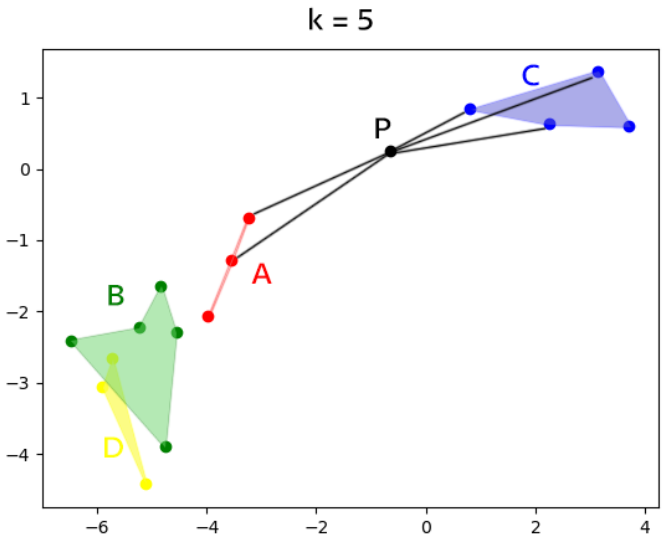

Figure 1. Simple example of the KNN classification.

The Figure 1 illustrates an example of the KNN classification algorithm being applied. The points have been segregated between 4 different groups (*i.e.*, A, B, C and D) and a point P with no group is provided to be classified, the KNN algorithm then searches for the k points closest to the point P and, after that, the groups have their frequencies counted, with the highest frequency being chosen as the classification of P, in the case of this example, the point P is classified to belong to the group C.

Input

An input represents the classification of a single point. The first line contains the pattern "n_groups=n", where n is the total number of groups in this input. Then, for each of the n groups the following lines will be provided, the first line has the pattern "label=c" where c is a single character that represents the label (*i.e.*, identification) of the group, then the second line follows the pattern "length=L", where L is the number of two-dimensional points in this group and then, the next L lines contain the pattern "(x,y)" that represents

each point that belongs to the aforementioned group. After all groups have been represented there's a single line with the pattern "k=k", where k is the parameter previously discussed. Lastly, the final line also contains the pattern "(x,y)", representing the coordinates to the two-dimensional point to be classified.

The input must be read form the standard input.

Output

The output contains a single character, it being the label of the group chosen as the classification of the provided point.

The output must be written to the standard output.

Example

Input example	Output example
n_groups=4 label=A length=3 (-3.55,-1.28) (-3.99,-2.06) (-3.23,-0.70) label=B length=5 (-4.85,-1.65) (-5.23,-2.22) (-4.75,-3.89) (-6.48,-2.41) (-4.56,-2.29) label=C length=4 (2.25,0.64) (0.80,0.85) (3.13,1.37) (3.71,0.59) label=D length=3 (-5.73,-2.65) (-5.11,-4.41) (-5.92,-3.06) k=5 (-0.65,0.25)	C

Problem B
Submatrix of Maximum Sum

Given a matrix M (with dimensions NxN) and an integer S, the objective is to find the SxS submatrix of M with maximum sum. A trivial algorithm to solve this problem consists in generating all $(N - S + 1)^2$ such submatrices and, then, computing their sum (the total time complexity would be, in the worst case, $O((N - S)^2 S^2 = O(N^2 S^2)$)

The proposed baseline algorithm (solution.cpp) employs a sliding window technique to reduce this complexity to $O(SN^2)$.

Input

The input has only one test case. The first line contains two integers N and S ($0 < S < N < 8000$). Then, there are N rows representing the matrix M. Each row contains N integers (with a whitespace after each one).

The input must be read from the standard input.

Output

The output contains a single line. Print the sum of the SxS submatrix of M with maximum sum. It is guaranteed the result will fit a 32-bit signed integer.

The output must be written to the standard output.

Example

Input Example	Output example
5 3 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2	51

Problem C
N-body

An n-body simulation models a dynamic system of particles, usually under the influence of physical forces, predicting their individual motions. A well known application of n-body simulation is in astrophysics, where particles refer to celestial objects interacting with each other gravitationally. The provided serial code contains a simple, yet working, n-body simulator for bodies moving through a 3 dimensional space. Your job is to create a parallel version of this n-body simulator, retaining the correctness of the solution.

Input

The input consists of a binary file containing a structure with six float typed data items for each particle in the system. These six data items refer to the position of the particle in the x, y, and z axes, and the velocity of the particle in the x, y, and z axes.

The input must be read form the standard input.

Output

The ouput is also a binary file, with the same format of the input file, containing the positions and velocities of each particle in the x, y, and z axes.

The output must be written to the standard output.

Example – *in string form, just for readability purposes*

Input example	Output example

`0.680375 -0.211234 0.566198`
`0.596880 0.823295 -0.604897`
`-0.329554 0.536459 -0.444451`
`0.107940 -0.045206 0.257742`
`-0.270431 0.026802 0.904459`
`0.832390 0.271423 0.434594`
`...`
	`-5.184253 11.632033`
	`-11.737427 -63.642937`
	`146.243469 -147.690735`
	`10.654608 -15.666613`
	`15.572982 133.827255`
	`-195.720352 194.728912`
	`22.627201 -1.267884 -5.422720`
	`283.039612 -15.879736`
	`-67.786301`

Problem D
Bucketsort

Bucket sort or Binsort is a sorting algorithm. Its sorting strategy is to divide the numbers into 'buckets' with a finite number of elements. The bucket's advantage is its complexity (O(n)), where 'n' is the size of the array. However, linear complexity is only achieved when the number is evenly distributed.

Input

An input has 2 parts: the parameters and the data. The first four numbers are string length, array size, offset, and bucket number. The remaining parameters are integer numbers to order.

The input must be read form the standard input.

Output

The output contains only the sorted integers.

The output must be written to the standard output.

Example

Input example	Output example
5 100000 48 9 99997 99996 99995 99994 99993 99992 99991 99990 99989 99988 99987 99986 ...	00000 00001 00002 00003 00004 00005 00006 00007 00008 00009 00010 00011 ...