TACC Invitational Hands-on Programming Set

I. General Notes

- All problems may be done in any order and are worth 60 points.
- There is no extraneous input. All input is exactly as specified in the problem. Unless otherwise specified, your program should read input to the end of the file.
- Your program should not print extraneous output. Follow the form exactly as given in the problem.
- A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of problems

1. Stampede
2. Jetstream
3. Chameleon
4. Corral
5. Hikari
6. Lonestar 5
7. Stallion
8. Maverick
9. Rodeo
10. Stampede2
11. Wrangler
12. Ranger

1. Stampede

Program name: stampede.java, stampede.py, stampede.cpp
Input file: None

Stampede was one of the fastest computing clusters in the world, when it was launched in 2012. It had `6,400` nodes. Each node had an onboard Xeon CPU with 16 processor cores, as well as an additional Xeon Phi Co-Processor add-on that added an additional 60 cores. Some nodes had GPU cores, as well. The entire system had `522,080` cores.

Computing performance is measured in terms of how many quadrillion floating-point operations it could perform per second, otherwise known as *PetaFLOPS*. The entire system performed at 12 PetaFLOPS. Each Xeon Phi Co-Processor operated at a speed of 1 TeraFLOPS, or one trillion floating-point operations per second. Each Xeon CPU operated at a speed of 500 GigaFLOPS, or 500 billion floating-point operations per second.

Input

There is no input for this problem.

Output

Your program must output the number of GigaFLOPS per GPU core, rounded to the nearest integer GigaFLOPS.

No Sample Input

No Sample Output¶

2. Jetstream

Program name: jetstream.java, jetstream.py, jetstream.cpp
Input file: jetstream.dat

The Jetstream system is an Elastic Cloud. This means that many virtual machines (VMs) run inside of the entire system for smaller workloads like running a web service. The system is Elastic. This means that if a VM gets overloaded, a Jetstream can automatically create additional VMs to split the workload.

Let's say that a VM can handle a certain number of users before 50% the memory on the server is consumed. Once it hits this limit, Jetstream will create an additional VM for any new users. New users will be distributed to the new VM. If the overall usage decreases, excess VMs are shut down.

Input

Your program must accept input from the text file `jetstream.dat`. The text file will contain an integer *N*, representing the number of data points in the text file. Each data point is a decimal number, representing an hourly measurement of how many fractions of VM memory are needed during that hour. For example, `2.25` means that at the current load would fill up 2 VMs completely and another VM 25% of the way. **Remember that this is not the number of VMs in actual use, since a new VM would be created when the memory usage exceeds 50%!** Therefore, a usage of `2.25` would require *5* VMs.

Output

Your program must print, to the screen, an integer representing the number of VM hours used across the specified period followed by `HOURS`. VM hours are counted as the sum of VMs across all hours in the test data.

Sample Input File `jetstream.dat`

```
10
1.0
2.0
1.5
2.25
2.5
2.25
1.0
1.5
1.25
1.0
```

Sample Output to Screen

```
34 HOURS
```

3. Chameleon

Program name: chameleon.java, chameleon.py, chameleon.cpp
Input file: chameleon.dat

Chameleon is a system designed for bare metal access to a variety of hardware types, such as GPU, FPGA, ARM, NVME, VM, Haswell and Infiniband. Individual users can reserve individual nodes on Chameleon for their use. However, to make sure that the system is available to all users, reservations cost Service Units. Each node type has a different cost of service units per hour.

- ARM, NVME, Haswell and Infiniband nodes all cost 1 Service Unit per node, per hour.
- GPU nodes are the most in demand, so they cost 8 Service Units per node, per hour.
- FPGA nodes only cost 1 Service Unit per node, per hour, but also require the use of a special Build Node. Any individual reservation for any number of FPGA nodes automatically incur a flat 4 Service Unit cost.
- VM nodes, or Virtual Machine nodes, are free.

Input

Your program must accept input from the file `chameleon.dat`. The first line of the text file is an integer N, specifying the number of reservations made by a Chameleon user. The next N lines of the text file each contain a reservation. Each reservation contains a reservation name, node type, number of nodes, and a number of hours for the reservation. If there is a reservation specified later in the text file that has the same name as a previously existing reservation, the earlier reservation is cancelled and the user should not be charged Service Units.

Output

Your program must output to the screen the number of Service Units used by the reservations contained in the input file, followed by `SUs`.

Sample Input File `chameleon.dat`

```
10
machine_learning GPU 5 10
low_power_experiment ARM 10 3
burst_writes NVME 1 4
load_balancing VM 10 10
open_cl FPGA 4 6
benchmarking Haswell 2 10
burst_writes NVME 1 6
low_power_experiment ARM 4 3
mpi Infiniband 4 4
parallel_circuit FPGA 2 2
```

Sample Output to Screen

```
486 SUs
```

4. Corral

Program name: corral.java, corral.py, corral.cpp
Input file: corral.dat

The TACC system Corral stores a lot of public data sets, including data for the DesignSafe platform. You can think of DesignSafe as the "Google Docs" for Civil Engineers that are trying to create structures that are resilient to natural disasters such as Earthquakes.

One common task in computational analysis of earthquakes is to figure out the wavelength of an earthquake. The wavelength of an earthquake can easily be determined by looking at the time in between peaks of accelerometer data from an earthquake.

Input

Your program must read input from a file named `corral.dat`. The first line of the file is an integer N that represents the number of data points in the file. The file will be followed by N floating point numbers, representing accelerometer samples taken every 100 milliseconds.

Output

Your program must output the wavelength of the earthquake in milliseconds. Peaks in the data can be identified as values whose neighbors are lower values. The interval between peaks is the wavelength. Peaks will never be the first nor last sample.

Sample Input File `corral.dat`

```
20
0.1
0.8
1.1
0.9
0.4
-0.2
-1.0
-0.5
-0.1
0.7
1.0
0.8
0.5
0.2
-0.4
-0.9
-0.5
0.1
0.5
0.7
```

Sample Output to Screen

```
800ms
```

5. Hikari

Program name: hikari.java, hikari.py, hikari.cpp
Input file: hikari.dat

The TACC system Hikari is unique, in that it is solar powered. The TACC parking lot is a large solar array, that can supply up to 220 kilowatts of power when it is very sunny outside. When the solar array is not providing enough power due to sunset or cloudy conditions, Hikari has a battery backup system with a 800 kilowatt capacity. Any power not used by Hikari is used to charge its backup system. If the battery is full, energy is returned to the city power grid. If the battery is empty and Hikari is consuming more power than is provided by solar, energy is consumed from the city power grid.

Input

Your program must read input from the text file `hikari.dat`. This file will contain 24 lines of input. Each line is a pair of integers separated by a space, representing Hikari power measurements for an hour of the day. The first number is the power output from the solar array for that hour. The second number is Hikari's power consumption for that hour. *Assume that the battery backup system has 400 kilowatts at the beginning of the 24 hour period.*

Output

Your program must output to the screen one line with the number of kilowatts of current supplied to the the city grid followed by `KW SUPPLIED`, and a second line with the number of kilowatts consumed from the city grid followed by `KW CONSUMED`.

Sample Input File `hikari.dat`

```
0 60
0 60
0 60
0 60
0 120
60 120
160 140
220 140
220 140
220 140
120 140
180 140
220 120
220 120
220 120
220 140
200 140
180 80
160 80
140 100
120 120
60 120
0 120
0 120
```

Sample Output to Screen

```
120 KW SUPPLIED
20 KW CONSUMED
```

6. Lonestar 5

Supercomputers like Lonestar 5 take large problems and divide them into smaller pieces. Each fragment of the task is distributed to different *nodes* on the cluster to be computed simultaneously, in parallel. The problem with parallel computation is that each node you add incurs a time penalty.

This loss of efficiency is called *overhead*. The more nodes you add, the more overhead you suffer. That means that distributing a task 20 ways doesn't result in a 20x speedup. In reality, it may result in a lower actual speedup due to this overhead. For example, you may have a 10 hour job that you want to divide across 2 nodes. Unfortunately, transferring data to each additional node past the first one may take 1 hour per node. Therefore, the real time for your task would be 6 hours. The *speedup*, in this case, was 1.7x (rounded to the tenths place.)

Input

Your program must read input from a file named lonestar5.dat. There is one line of input with three integers. The first integer represents the number of hours a task will take. The second integer represents the number of nodes assigned to the task. The third integer is the additional time in hours it will take, per node, to distribute the task.

Output

Your program must output the *speedup* for this task, rounded to the tenths place.

Sample Input File `lonestar5.dat`

```
4000 20 3
```

Sample Output to Screen

```
15.6x
```

7. Stallion

Program name: stallion.java, stallion.py, stallion.cpp
Input file: stallion.dat

In addition to the Data Center on the Pickle Research Campus, TACC also has a Visualization Laboratory on UT's Main Campus. One of the systems housed there is Stallion, a tiled display cluster. It has 80 monitors arranged in a 16x5 configuration. The screen resolution of the entire system is 40,960 x 8,000 pixels, for a total of 328 megapixels. This allows us to display extremely high resolution content, such as the full resolution composite of the Hubble Space Telescope view of the Andromeda Galaxy, which takes up the entire display area. One *node* of Stallion is responsible for operating a group of 4 monitors, in a horizontal line. Stallion is comprised of 20 nodes.

Input

Your program must read input from a text file named `stallion.dat`. The first line of input will be an integer N that represents the number of test cases in the file. Each test case will be a resolution of an image, represented as an integer width, an `x`, and an integer height.

Output

Your program must output the minimum number of nodes required to display the image in each test case.

Sample Input File `stallion.dat`

```
5
5000x3000
3000x5000
4096x2080
1024x768
2950x2800
```

Sample Output to Screen

```
2
4
2
1
2
```

8. Maverick

Program name: maverick.java, maverick.py, maverick.cpp
Input file: maverick.dat

The TACC system Maverick contains many General Purpose Graphics Processing Units, or GPGPUS. They are very good at performing machine learning tasks, such as computer vision. One such task is called a *Convolution*, which is a way of looking for shapes or features in an image. A square filter is applied on each group of pixels on the image, which may result in an image being "detected". For example, let's say you have an image with these pixel values.

```
    0   0   0   0   0
    0   0   255 255 255
    0   0   255 0   0
    0   0   255 0   0
```

A 3x3 filter that detects corner shapes in images may look like this:

```
    1   1   1
    1   -1  -1
    1   -1  -1
```

The convolution filter is applied by multiplying each value in the filter by each corresponding pixel in an image region with the same size and adding the products to get a "score". If you applied the filter to the upper left of the image (a region beginning with row 0, column 0) the result of the filter would be:

```
(1 * 0 + 1 * 0 + 1 * 0) + (1 * 0 + -1 * 0 + -1 * 255) + (1 * 0 + -1 * 0 + -1 * 255) =
-510
```

If you applied the filter to the lower right of the image (a region starting at row 1, column 2), the result of the filter would be:

```
(1 * 255 + 1 * 255 + 1 * 255) + (1 * 255 + 0 * 0 + 0 * 0) + (1 * 255 + 0 * 0 + 0 * 0)
= 1275
```

Therefore, it is much more likely that a corner is at row 1, column 2 due to its higher "score".

Input

Your program must accept input from the text file named `maverick.dat` that contains a 3x3 convolution filter with integer values, followed by image pixel data (in the shape of a rectangle of unspecified size) with only positive values between 0 and 255 for each pixel. Each value of each row will be separated by a space, and each row will be on a separate line. There is no extra blank line between the filter and the image.

Output

Your program must output the row and column of the region of the image with the highest "score", using the words `row` and `column` (in that order) to denote the row and column of the upper left corner of the region with the highest score. Assume the top row is 0 and the left column is 0.

Sample Input and Output on next page...

Sample Input File `maverick.dat`

```
-1 2 -1
2 3 2
-1 2 -1
28 38 21 40 80 35
78 23 22 56 93 54
76 25 36 81 100 91
23 47 23 45 87 87
12 14 79 22 23 25
80 81 82 43 21 24
```

Sample Output to Screen

```
row 1 column 3
```

9. Rodeo

Program name: rodeo.java, rodeo.py, rodeo.cpp
Input file: rodeo.dat

Rodeo is one of TACC's OpenStack Virtual Machine (VM) host systems. OpenStack has a firewall for each VM, that allows it to determine what traffic is allowed in and out of the system. Firewall rules consist of an action (`ALLOW` or `DENY`), a direction (`INGRESS` or `EGRESS`) and a port number (or `ALL` for all ports). For example, let's say a VM has these rules:

```
    DENY INGRESS ALL,ALLOW EGRESS ALL,ALLOW INGRESS 22
```

This means that by default, all incoming traffic will be denied. All outbound traffic will be allowed. Incoming traffic on port 22 will be allowed. There are a total of 65,535 ports that are all assumed to be closed in both directions to begin with. The ordering of the rules matters. For example, the same traffic would not be allowed if the rule was written:

```
    ALLOW INGRESS 22,ALLOW EGRESS ALL,DENY INGRESS ALL
```

This is because `DENY INGRESS ALL` turned off all ingress ports after `ALLOW INGRESS 22` opened port 22.

Input

Your program must read input from a file `rodeo.dat`. The first line of the input file will be an integer *N* specifying the number of test cases. The next *N* lines will each be individual test cases. A test case consists of a series of rules, separated by commas. This will be followed by a semicolon, and then one test traffic case that is either `INBOUND` or `OUTBOUND` followed by a port number.

Output

For each test case, print either `ALLOWED` or `DENIED` if the packet was allowed to pass through the Firewall given the test case's rules.

Sample Input and Output on next page...

Sample Input from File `rodeo.dat`

```
7
DENY INGRESS ALL,ALLOW EGRESS ALL,ALLOW INGRESS 22;INBOUND 22
DENY INGRESS ALL,ALLOW EGRESS ALL,ALLOW INGRESS 22;OUTBOUND 22
ALLOW INGRESS 22,ALLOW EGRESS ALL,DENY INGRESS ALL;INBOUND 22
DENY INGRESS ALL,DENY EGRESS ALL,ALLOW EGRESS 22;OUTBOUND 80
DENY INGRESS ALL,ALLOW EGRESS ALL,DENY EGRESS 22;OUTBOUND 80
ALLOW INGRESS ALL,DENY INGRESS 135,ALLOW EGRESS ALL;INBOUND 135
DENY INGRESS ALL,ALLOW INGRESS 22,ALLOW INGRESS 443;INBOUND 22
```

Sample Output to Screen

```
ALLOWED
ALLOWED
DENIED
DENIED
ALLOWED
DENIED
ALLOWED
```

10. Stampede2

Program name: stampede2.java, stampede2.py, stampede2.cpp
Input file: stampede2.dat

Machines like Stampede2 are designed to perform extremely high precision calculations, where every decimal place matters. This was extremely important in the LIGO Experiment, which needed to detect fluctuations in space-time at a scale of 1/20th the width of a proton. (That's a very small number.) These values are stored in scientific notation, with a mantissa and exponent. For example, the value `2.653E+10` has a mantissa of `2.653` and an exponent of `10`. Two problems in these types of calculations are Numeric *Overflow* and *Underflow*.

Numeric Overflow occurs when the mantissa becomes high enough to result in a loss of significant digits. This is due to the fact that the number of digits in the mantissa is limited by the size of the variable that stores it. **For our purposes, let's say that the maximum number of digits for the mantissa is 1 whole number and 10 decimal places.**

Here is an example of a situation where overflow occurs:

```
    2.653E+10 + 9.5912498745E+10
    = 26530000000 + 95912498745
    = 122442498745
    = 1.2244249874E+11      # Oops! We lost the 5 at the end
```

Numeric Underflow occurs when an operation, such as addition, between two numbers of extremely different scale result in no change to the stored value of the larger number. Here is an example:

```
    2.653E+10 + 9.5E-4
    = 26530000000 + 0.00095
    = 26530000000.00095
    = 2.6530000000E+10     # Oops! We didn't store the .00095!
```

Input

Your program must accept input from a text file `stampede2.dat`. This first line of the file contains an integer *N* which specifies the number of test cases. Each test case consists of two numbers written in scientific notation (mantissa, followed by `E` and then an exponent value). The numbers are separated by a space.

Output

For each test case, assume that the numbers are being added together. For each case, output `OVERFLOW` if Numeric Overflow occurs, `UNDERFLOW` if Numeric Underflow occurs, or `SAFE` if the addition operation is safe.

Sample Input and Output on next page...

Sample File Input `stampede2.dat`

```
8
2.653E+10 9.5912498745E+10
2.653E+10 9.5E-4
2.653E+10 9.59E+4
2.653E+4 9.2E+10
9.5E-4 2.653E+10
2.653E+10 7.27391E+5
2.36E-4 6.274E-5
2.36E-1 6.274992757E-5
```

Sample Output to Screen

```
OVERFLOW
UNDERFLOW
SAFE
SAFE
UNDERFLOW
SAFE
SAFE
OVERFLOW
```

11. Wrangler

Program name: wrangler.java, wrangler.py, wrangler.cpp
Input file: wrangler.dat

Wrangler is a data intensive compute system, with extremely fast Solid State storage technology used for performing tasks on large data sets, such as images. A common task related to image data is extracting Regions of Interest, often based on blobs that may appear in an image. These regions are rectangular sub-regions of a larger rectangular image. For example, let's consider this black and white image:

```
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 1 1 1 0
0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
```

There is a Region of Interest in this image:

```
0 1 0 0
1 1 1 1
0 1 1 0
```

The Region of Interest is the smallest rectangle that can fully contain the blob.

Input

Your program must accept input from the text file named `wrangler.dat`. The first line of the text file contains two integers, separated by a space. The first integer is the number of rows of image data, and the second integer is the number of columns in the image. The dimensions are followed by a matrix of integers, which contains a black and white image represented by `1` and `0` values separated by spaces. Each `1` and `0` value on a row is separated by a space, and each new row is on a new line of data.

Output

Your program must output the smallest rectangular Region of Interest that contains the blob.

Sample Input and Output on next page...

Sample Input File `wrangler.dat`

```
9 10
0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 0 0 0
0 0 1 1 0 0 1 0 0 0
0 0 1 0 1 0 1 0 0 0
0 0 0 1 1 1 1 0 0 0
0 0 1 0 1 1 0 0 0 0
0 1 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Sample Output to Screen

```
0 1 0 0 0 1 0
0 0 1 1 0 0 1
0 0 1 0 1 0 1
0 0 0 1 1 1 1
0 0 1 0 1 1 0
0 1 0 0 1 0 0
```

12. Ranger

Program name: ranger.java, ranger.py, ranger.cpp
Input file: ranger.dat

TACC's first cluster system, launched in 2007, was Ranger. Large scale systems like Ranger never truly go away. Parts of Ranger were distributed to other institutions to build new systems, and those systems are still operational. Just like Ranger, the language FORTRAN never goes away, either. It is still used in modern high performance and scientific computing!

In FORTRAN, loops look like this:

```
do i=0,20,1
    print i
end do
```

This loop works like a `for` loop in other languages. The loop control variable is `i`. The starting value of `i` is `0`. The loop terminates when `i` is equal to `20`, and each iteration of the loop will increment `i` by `1`.

Input

Your program must accept input from a file named `ranger.dat`. The first line of the input file will be an integer *`N`*, representing the number of test cases. Each test case will be the first line of a FORTRAN `do` loop, using the format seen above.

Output

For each test case, your program must print to the screen the name of the loop control variable, a single space, and then a list of values, separated by spaces. (There should be no space after the last value.) Each value must be the value of the loop control variable after each iteration of the `do` loop, excluding the value that causes the loop to exit. You may assume that each test case will reach the terminating value.

Sample Input File `ranger.dat`

```
3
do omg=0,20,1
do so=0,25,5
do old=10,2,-2
```

Sample Output to Screen

```
omg 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
so 0 5 10 15 20
old 10 8 6 4
```