2024 CCSC Eastern Conference Programming Competition

October 19th, 2024

Mount St. Mary's University, Emmitsburg, MD

This page is intentionally left blank.

Problem 1 — SHOUTING

I ALWAYS YELL. FOR THE LONGEST TIME, WHEN I WROTE MESSAGES TO PEOPLE, THEY DIDN'T KNOW THAT I WAS YELLING AT THEM. THEN I LEARNED THAT CAPITAL LETTERS MEAN I'M SHOUTING, SO OF COURSE I WANT TO USE THEM ALL THE TIME. ALL. THE. TIME. NATURALLY, I WROTE A PROGRAM SO THAT PEOPLE WILL AUTOMATICALLY KNOW THAT I'M YELLING AT THEM. BECAUSE I AM! ALL THE TIME. YOU SHOULD WRITE YOUR OWN PROGRAM TO DO THE SAME SO WE CAN YELL AT EACH OTHER. DO IT. DO IT NOW!

Program input: There will be exactly 15 lines of text. Lines of text may include letters, digits, spaces, and any of these five punctuation symbols: {,.!?-}.

Program output: For each line of text, output that line of text with each lower-case letter replaced by the corresponding upper-case letter. Non-letter characters should remain unchanged.

Example input:

```
Hello there
Good to see you!
27
I will have 3 boxes, please.
Where is - the thing?
Hi there
Bad to see you!
28
I will not have 3 boxes.
What is - the thing?
Good-bye
I cannot see you!
29
There is no box.
Why is - the thing?
```

Example output (corresponding to the input shown above):

```
HELLO THERE
GOOD TO SEE YOU!
27
I WILL HAVE 3 BOXES, PLEASE.
WHERE IS - THE THING?
HI THERE
BAD TO SEE YOU!
28
I WILL NOT HAVE 3 BOXES.
WHAT IS - THE THING?
GOOD-BYE
I CANNOT SEE YOU!
29
THERE IS NO BOX.
WHY IS - THE THING?
```

Problem 2 — Class Photo

As a school photographer, you're keenly aware of the importance of composing a class photo for maximum dramatic effect. Lesser photographers would arrange students in rows, put the taller students in the back, and so forth. However, you know that there is only one proper way to arrange a class photo: put the students in a single line, sorted by either increasing height or decreasing height.

To make the most efficient use of your time, you decide to write a program to check whether students are arranged correctly for a class photo.

Program input: Each normal line of input is a sequence of one or more positive integers representing student heights in centimeters, with values separated by one or more spaces. As a special case, if a line contains only the value `-1`, it indicates the end of input, and the program should exit without any further output.

Note that there is no arbitrary upper limit on the number of values on one input line. For example, a very large class might have 1,000 students, and your program should be able to handle this situation.

Program output: For each normal line of input, the program should print a single line of text consisting of either "`yes`" or "`no`", depending on whether the values on the line were sorted in either increasing or decreasing order. Note that it is acceptable for two students to have the same height: an ordering of students is allowed as long as the sequence of students doesn't have cases of both increasing height and decreasing height.

Example input:

```
102 118 120 128 140
145 139 133 123 123 122 115
125 147 120 108 128 98 141 123 132
-1
```

Example output (corresponding to the input shown above):

```
yes
yes
no
```

[This page is intentionally left blank.]

Problem 3 — Combo Deal

It is several years in the future. During a season of economic troubles, there were a series of mergers among the major burger chains. Now there are only two major national fast food burger chains remaining: Checker McSonic's and WendCastle King.

Due to those same economic troubles, you and your friends insist on going to the restaurant with the lowest price for your total order. Fortunately, the comparison isn't too complicated, because you and your friends have simple food tastes. You only want burgers, fries, drinks, and dessert (and you don't care what the dessert is). Unfortunately, it is complicated by the fact that there are three types of burgers: small, large, and specialty. Fortunately, you and your friends aren't picky about which restaurant's version you get. Unfortunately, it's also complicated by the two restaurants' combo deals.

Each restaurant sells a full menu of items, but you and your friends only care about the prices of six things: small burger, large burger, specialty burger, fries, drink, and dessert. Checker McSonic's has two combo deals. The first grants a discount when a customer buys a specialty burger with a dessert. The second grants a discount when a customer buys a small burger and an order of fries. WendCastle King has a combo deal that grants a discount when a customer buys a large burger, fries, and a drink. Here are the prices at each restaurant:

Checker McSonic's

- Small Burger $3
- Large Burger $6
- Specialty Burger $10
- Fries $3
- Drink $2
- Dessert $4
- Combo #1 (Specialty Burger + Dessert) $11
- Combo #2 (Small Burger + Fries) $4

WendCastle King

- Small Burger $2
- Large Burger $7
- Specialty Burger $9
- Fries $3
- Drink $3
- Dessert $3
- Combo (Large Burger + Fries + Drink) $9

Program input: The first line of input has an integer n ($0 < n < 10000$), and that is followed by n additional lines. Each of the other lines indicates an order of food that you and your friends want to place. A line contains the following: the number of small burgers, large burgers, specialty burgers, orders of fries, drinks, and desserts, in that order.

Program output: For each of the orders, print one line containing either CMS (if Checker McSonic's provides a cost lower than WendCastle King) or WCK. Then, after a space, print the group's total cost in dollars.

Example input:

```
6
1 2 3 2 0 1
4 4 4 4 4 4
0 5 0 0 0 0
5 0 0 0 0 0
2 0 0 2 0 0
0 0 0 1 0 0
```

Example output (corresponding to the input shown above):

```
CMS 50
WCK 92
CMS 30
WCK 10
CMS 8
WCK 3
```

Problem 4 — Jumbled Up

You work for the Bitburg Post-Dispatch newspaper as a puzzle developer. Your readers have demanded to have a "Jumble" type puzzle in the paper, wherein readers must un-scramble a series of scrambled words. For instance if the scrambled word is "TWHISC", the savvy reader will realize the only word which can be made from these letters is "SWITCH".

Not every word can be used in creating a Jumble puzzle. For instance the scramble "GANEL" cannot be used since there are three possible solutions: "ANGEL", "ANGLE", and "GLEAN". A valid Jumble puzzle only has one possible solution, so the answer cannot have any anagrams which are also valid words.

In order to facilitate making Jumble puzzles, you need to write a program which will read in a set of words and find the ones which are "jumble-able" – that is the words for which there is no other word in the set of words containing the same set of letters.

Program input: Input consists of a number of test cases. Each test case begins with a line containing a number N giving the number of words in this test case. Following that is a line containing those N words, separated with spaces. A value of 0 for N indicates that there are no more test cases.

Program output: Output consists of one line containing the text "Results:", followed by a list of the words from the test case that are jumble-able, printed in alphabetical order.

Example input:

```
7
TASTE GLEAN ALERT ANGLE MOUNT ANGEL LATER
6
ZOO ALIGNED DEALING LEADING LEANING DEAL
0
```

Example output (corresponding to the input shown above):

```
Results:
MOUNT
TASTE
Results:
DEAL
LEANING
ZOO
```

[This page is intentionally left blank.]

Problem 5 — Top Ten

The music website Splotchify wants to be able to show each user their own personal top ten list based on what they listened to over the past 14 days. Splotchify decided to use a formula to determine the top songs. The formula is based on three values: the number of times the the song was played (n), the number of different days that the song was played (d), and the span from the earliest day the song was played to the most recent day the song was played (s). The span is defined as the difference between the first day the song was played and the last day the song was played.

The formula used to calculate the top ten is $nd + s$. The songs with the highest values for a given listener comprise that listener's top ten.

Program input: There will be 14 lines of input. Each line lists the id numbers of all the songs a person played on a given day, separated by spaces. Each id number is a positive integer from 1 to 1000000000. If no songs were played on a given day, then instead of a list of numbers, that line will have the number -1. An id number may be listed multiple times on the same line if a song is played multiple time on the same day.

Program output: Output the id numbers of the top ten songs (in order) based on the formula above. In the case of ties, lower-numbered songs win. Separate the ten id numbers with spaces.

Example input:

```
1 2 3 4 5 6 7 8 9 10 11 12 400000000
2 3
2
2
2 2
-1
-1
-1
11
12
13 13 13 13 13 13 13 13 13 13 13 13 13 13
9 10 9 9
1 11 10 10 10 10 10
400000000
```

Example output (corresponding to the input shown above):

```
2 10 11 9 400000000 1 13 12 3 4
```

[This page is intentionally left blank.]

Problem 6 — Werewolf Transformations

You, for better or for worse, are a werewolf. But unlike a werewolf who is a mindless beast, powerless to resist the call of the full moon, you can transform back and forth from wolf form at will. Naturally you use this power to become a super hero: Wolf Person™.

In your wolf form you are are able to run much faster than you can in your regular, human form. However you struggle with other tasks as a wolf, such as opening doors (with no opposable thumbs and all). In particular, you can run 10 meters per second as a wolf, while only 4 meters per second as a human. It takes 8 seconds to open a door in wolf form and only 1 second as a human. It also takes 5 seconds to transform from one form to another.

As Wolf Person™, you often find yourself chasing suspects down long hallways with doors along it. You want to know the fastest way for you to run along such a hallway. Should you do it as human to open the doors faster, as a wolf to run faster, or will you need to transform back and forth to maximize your overall speed?

You should write a program that, given the distance you need to travel, and the locations of all of the doors in that distance, calculates the minimum time needed to cover the distance, along with how many transformations will be made.

You always begin the chase in your human form. You can be in either human or wolf form when you reach the end of the hallway.

Program input: The first line of input is an integer, N, giving the number of test cases. Following that will be N test cases, each of which is comprised of three lines. The first line of each test case is the distance, in meters, that you need to travel. The second line of each test case will contain the number of doors. There will be between 1 and 100 doors. The third line contains the locations of the doors along that path, as floating-point numbers separated by spaces.

Program output: Your output for this program should consist of two numbers for each test case, separated by a space. The first of these is a floating-point number, with two decimal points of precision given, indicating the minimum number of seconds needed to make it through the hallway. The second is an integer giving the number of transformations made along the way.

Example input:

```
3
100.0
4
10.0 80.0 85.0 90.0
25.0
7
5.0 8.5 10.0 15.5 21.0 22.75 24.0
```

```
750.0
2
300.0 500.0
```

Example output (corresponding to the input shown above):

```
28.50 2
13.25 0
96.00 1
```

Problem 7 — Pokémon Island Survival

In the land of Pokémon, chaos has erupted on a mysterious island! The infamous Gyarados has been rampaging, causing a massive storm and making the water levels rise across the entire island. Why, you ask? Well, it turns out Gyarados got confused after a particularly puzzling conversation with Mr. Mime!

Every time Gyarados tries to calm down, the storm intensifies and causes a flood that lowers the land elevation. This happens three times! Your mission as a brave Pokémon Trainer is to figure out how many isolated islands of Pokémon will remain after the floods.

The island is represented as a grid where each cell has an elevation between 0 and 5. The value 0 represents water, and values 1 to 5 represent different land elevations, with 5 being the highest.

During the three flood steps, the elevation of each land cell decreases by 1. After the storm, isolated groups of Pokémon, or islands, remain. Islands are formed by connected land cells with elevation greater than 0, and Pokémon can only move between cells that share a side (north, south, east, or west).

Your job is to report to Dr. Oak the number of islands that are left after the storm.

Program input: The program will first receive an integer T representing the number of test cases. Then for each test case:

- The first line will contain two integers m and n, representing the dimensions of the grid.
- The following m lines will contain n integers each, representing the elevation of the island at each point in the grid (from 0 to 5).
  - 0 represents water
  - 1-5 represent the land's elevation

The program should simulate three flood steps, decreasing each land cell's elevation by 1 during each step. Then, it should count the number of isolated islands where the remaining land cells have values greater than 0.

Program output: The output is the number of islands left after the three flood steps.

Example input:

```
3
3 3
5 5 5
5 5 5
5 5 5

4 4
1 2 3 4
```

```
2 3 4 5
3 4 5 0
4 5 0 0

3 5
5 4 0 0 0
5 3 0 4 4
0 0 2 2 0
```

Example output (corresponding to the input shown above):

```
1
1
2
```

Problem 8 — Preprocessing

The C preprocessor transforms an input text file to produce an output text file. One feature of the C preprocessor is allowing the use of `#ifdef`, `#else`, and `#endif` directives for conditional compilation. You will implement a limited C preprocessor which works as follows.

If the source text has a sequence of lines starting with `#ifdef FOO` and ending with `#endif`, with no corresponding `#else` directive between them, then the text generated by the lines between the `#ifdef` and `#endif` directives is included in the output if and only if the `FOO` preprocessor symbol is defined.

If the source text has a sequence of lines starting with `#ifdef FOO` and ending with `#endif`, and there *is* a corresponding `#else` directive between them, then

1. If the preprocessor symbol `FOO` is defined, then the lines between the `#ifdef` and `#else` are included in the output
2. If the preprocessor symbol `FOO` is *not* defined, then the lines of text between the `#else` and `#endif` directives are included in the output

Note that `#ifdef`/`#endif` and `#ifdef`/`#else`/`#endif` constructs may be nested.

Program input: The input to the program starts with a sequence of one or more lines terminated by a line containing just `<EOF>`. All of the lines up to but not including the `<EOF>` line are the input text that the preprocessor will work with. Each subsequent line defines some number (0 or more) of preprocessor symbols, separated by one more spaces. You may assume that each preprocessor symbol will consist of 1 or more letters. You may also assume that preprocessor symbols will not be repeated on the same line. There is no explicit end-of-input indication. When there is no more input, the program should exit without printing any additional output.

You may assume that in the source text, each `#else` and `#endif` will be properly balanced with a previous `#ifdef`.

Program output: For each line containing 0 or more preprocessor symbols, the program should write a single comment line (explained below) followed by the output text resulting from preprocessing the input text.

When output text is printed, the comment line preceding it should have the form `/* sym1 sym2 ... */` where *sym1 sym2 ...* is the defined preprocessor symbols, sorted lexicographically, with adjacent symbols separated by a single space. One space should separate the `/*` from the first symbol, and one space should separate the last symbol from the `*/`. As a special case, if a line defining preprocessor symbols is empty (i.e., no preprocessor symbols are defined), then the comment line should be `/* */`, i.e., `/*`, followed by one space, followed by `*/`.

When lines of input text are generated as output, they should be generated exactly as they appear in the input.

Example input:

```
Hello, world
#ifdef Apple
Foobar
#ifdef Banana
  >>> A strong smell of petroleum prevails throughout <<<
#else
 Woo  hoo
#endif
#endif
Yeah
<EOF>
Apple Banana
Apple
Banana
```

Example output (corresponding to the input shown above):

```
/* Apple Banana */
Hello, world
Foobar
  >>> A strong smell of petroleum prevails throughout <<<
Yeah
/* Apple */
Hello, world
Foobar
 Woo  hoo
Yeah
/* Banana */
Hello, world
Yeah
```

Problem 9 — Logic Simulation

The following truth table defines the AND, OR, NAND, NOR, and XOR operations on boolean values (where *a* and *b* represent input values):

a	*b*	*a* AND *b*	*a* OR *b*	*a* NAND *b*	*a* NOR *b*	*a* XOR *b*
T	T	T	T	F	F	F
T	F	F	T	T	F	T
F	T	F	T	T	F	T
F	F	F	F	T	T	F

Each of the above operations can be a *logic gate* in a digital circuit. In addition, there are two special kinds of circuit elements, INPUT and OUTPUT, each of which has a single input and a single output. In general, each gate's input will be connected to exactly one gate's output. The only exceptions are INPUT elements (its input is not connected to any output) and OUTPUT elements (its output won't be connected to any gate's input.) As their names suggest, an INPUT gate provides an input to the circuit, and an OUTPUT gate provides an output from the circuit.

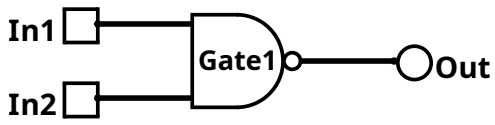
Program input: The program input consists of a series of simulations. Each simulation consists of a circuit description followed by simulation commands. There is no explicit end-of-input indication: the program should exit when there is no more input.

A circuit description consists of a series of node descriptions, one per line, terminated by a single line with the text "`simulate`". Each node description is a line in one of the following forms:

- *operation name pred1 pred2*
- `input` *name*
- `output` *name pred1*

In a node description line, *operation* is one of `and or nand nor xor`, and *name*, *pred1* and *pred2* are arbitrary node names. In each case, the node description defines a node whose name is given by *name*. For nodes specifying an operation, *pred1* and *pred2* represent nodes whose output is connected to one of the inputs of the node being defined. Here is an example circuit description (the corresponding circuit is shown on the right):

```
input In1
input In2
nand Gate1 In1 In2
output Out Gate1
simulate
```



Each simulation command is one of the following:

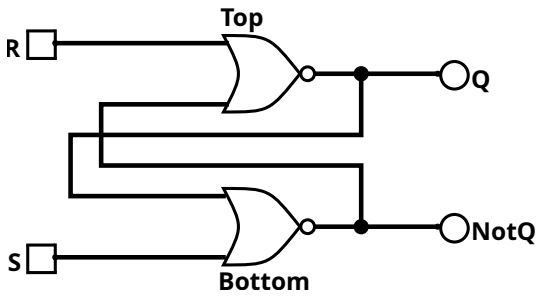
- `set` *name value*
- `run`
- `print` *name*
- `done`

The `set` command sets the input whose *name* is given to a specified value, either `T` or `F`. The `run` command runs the simulation by determining the final value of each input and output in the circuit when a steady state is reached. The `print` command prints a line of output of the form *name=value* indicating the output value of the named OUTPUT node. The `done` command finishes the overall simulation, allowing a new simulation (circuit description and simulation commands) to be started.

Names of gates, INPUT elements, and OUTPUT elements will consist of a letter followed by zero or more additional letters or digits. Names are case-sensitive, and may have upper and/or lower case letters.

You may assume that the input is well formed according to the specification above. You may also assume that in the circuit description, every input will be driven from the output of some gate or INPUT element. (The only exception is INPUT elements, whose inputs are only affected by `set` commands.) You may assume that for simulating a circuit, there will be at least one `run` command before any `print` command. Note that the circuit elements in the circuit description may be specified in any order, and it is *not* guaranteed that a node will be defined before it is referred to by another node. Also note that an output could be connected to more than one input, or could not be connected to any input.

Program output: For each `print` command, the program should print a line of output of the form *name=value* indicating the output value (`T` or `F`) of the named OUTPUT node, taking into account previous `set` and `run` commands. When a simulation starts (just after the `simulate` line that ends a circuit description), all circuit values (inputs and outputs) should start out as "unknown". When a `run` command is processed, the simulator should recompute all values that are affected (directly or indirectly) by previous `set` commands that may have changed the output values of INPUT elements. You can assume that a steady state will be reached in a finite number of steps (recomputations of output values.) Note that the circuit could have cycles, which means that when the simulator recomputes the output value at a circuit element, it might need to do so at a time when one of its input values is "unknown". When this happens, the simulator should treat the "unknown" value as false. For example, this will happen when simulating the circuit with NOR gates shown above (which is an "S-R latch").



An S-R latch

Example input:

```
input In1
input In2
nand Gate1 In1 In2
output Out Gate1
simulate
set In1 T
set In2 T
run
print Out
set In1 F
run
print Out
done
input R
input S
nor Top R Bottom
nor Bottom Top S
output Q Top
output NotQ Bottom
simulate
set S T
set R F
run
print Q
print NotQ
set S F
run
print Q
print NotQ
set R T
run
print Q
print NotQ
set R F
print Q
print NotQ
done
```

Example output (corresponding to the input shown on the left):

```
Out=F
Out=T
Q=T
NotQ=F
Q=T
NotQ=F
Q=F
NotQ=T
Q=F
NotQ=T
```

[This page is intentionally left blank.]