2023 CCSC Eastern Conference Programming Competition

October 21st, 2023

Bay Atlantic University, Washington, DC

This page is intentionally left blank.

Problem 1 — Nice Twit!

In your position as senior social media officer at Yummy TreatCo™, your main responsibility is to post to Tweeter.com™, a social media site known for its welcoming atmosphere and intelligent discourse. A post on Tweeter.com, known as a "Twit™", is a short line of text characters. As a representative of the delicious snack brands Yummy TreatCo produces, your mission is to inform the public of the superior qualities of these products. In order to perform your duties more efficiently, you decide to write a program to generate Twits from a template, allowing you to substitute various words into the template so that a variety of similar Twits can be produced.

Program input: The program input starts with a line containing an integer value indicating the number of Twit templates to be expanded. Each Twit template consists of three input lines. The first is the actual template text. The second is a single token indicating the placeholder into which a word should be substituted. The third consists of a sequence of one or more words separated by whitespace characters; these are the words to be substituted into the template for each occurrence of the placeholder. Note that there could be space(s) at the beginning and/or end of the line, and also note that the words could be separated by more than one space.

Program output: For each Twit template, one line should be printed for each substitution word. Each line should be constructed by substituting the substitution word for each occurrence of the placeholder in the template text. You may assume that the input is properly formed.

Example input:

```
4
You will enjoy our delicious {} snack cakes!
{}
BingBong Twynkie
Goritos: taste the XYZZY
XYZZY
 deliciousness zing   triangle
No one does na-flavored chips as well as Yummy TreatCo
na
banana fish cactus
Gimme a @! Gimme a @!
@
JellyBlast MegaMint
```

Example output (corresponding to the input shown above):

```
You will enjoy our delicious BingBong snack cakes!
You will enjoy our delicious Twynkie snack cakes!
Goritos: taste the deliciousness
Goritos: taste the zing
Goritos: taste the triangle
No one does banana-flavored chips as well as Yummy TreatCo
No one does fish-flavored chips as well as Yummy TreatCo
No one does cactus-flavored chips as well as Yummy TreatCo
Gimme a JellyBlast! Gimme a JellyBlast!
Gimme a MegaMint! Gimme a MegaMint!
```

Problem 2 — Pikachu's Journey to Victory Road!

Pikachu, the electrifying Pokémon, has embarked on an epic adventure to reach Victory Road, the ultimate challenge before facing the formidable Elite Four. To get there, Pikachu must ascend a staircase of n steps. However, there's a twist—there are multiple "Elite Four Chambers" steps on the staircase that Pikachu must absolutely not step into, or else it will find itself in an unexpected battle with a member of the Elite Four!

Pikachu starts at Step 0 and needs your help to find the number of distinct ways to reach Victory Road while ensuring that it never steps into the Elite Four Chamber steps. Pikachu can move from its current step to any unobstructed step that is one or two steps ahead, without ever stepping into any chambers. For example, if Pikachu is at Step 0 and there are no chambers at Steps 1 and 2, Pikachu can move directly from Step 0 to Step 1 or from Step 0 to Step 2 in a single move. However, if there is a chamber in step 1, then the only valid move Pikachu can make is jumping to step 2. If there were to be chambers in both steps 1 and 2, then Pikachu would never be able to reach Victory Road!

Program input: The first line of input is an integer N giving the number of staircases to traverse. Each staircase is represented with three lines:

1. The first line of each problem instance contains an integer n $(1 \leq n \leq 30)$, representing the number of steps on the staircase to Victory Road.
2. The second line contains an integer k $(0 \leq k \leq n)$, representing the number of Elite Four Chambers on the staircase.
3. The third line contains k space-separated integers representing the positions of the chambers (e_i). If k == 0, this line will contain a single placeholder value of "-1" to indicate the absence of chambers.

Program output: For each problem instance, print (on a single line) the integer value corresponding to the number of distinct ways Pikachu can reach Victory Road while ensuring that it never steps into the Elite Four Chamber steps.

Example input:

```
3
3
0
-1
10
2
4 7
10
3
4 5 6
```

Example output (corresponding to the input shown above):

```
3
6
0
```

In the example shown above, there are three problem instances. The first two instances have valid combinations, but the last instance has zero valid combinations due to the presence of chambers in consecutive steps.

Help Pikachu reach Victory Road safely by avoiding the Elite Four Chamber steps at all times!

Problem 3 — 500

There is a popular children's game called "500" which is played with one ball and at least three players. One of the players is designated as the thrower. This player throws the ball up in the air towards the other players. While throwing the ball, the thrower also shouts out a point value. Whoever catches the ball then receives that many points. If the ball hits the ground without being caught, then nobody receives those points. This continues until one player reaches 500 points, which makes them the winner.

Instead of calling out a numeric value for a throw, the thrower can opt to shout "Jackpot!" which means that anyone who catches that throw immediately wins. They can also instead shout "Bankrupt!" which means that whoever catches that throw loses all of the points they've accrued, going back to 0.

Sometimes the players lose track of how many points they have and so your task is to write a program which will decide the winner of games of 500, given the sequence of throws that take place. Keep in mind that it's possible that a game will continue a bit after someone has won. Any throws that take place after a player has won should not have any effect on the outcome.

Program input: The input consists of multiple games. The first line of input for each game is an integer N giving the number of throws in that game. Following that will be N lines, with each corresponding to one throw of the ball. Each of these lines consists of two parts, separated by a space.

The first is the value called out for the throw. This will either be a positive integer less than 500, "BANKRUPT" or "JACKPOT". The second part of each throw line contains either the name of the player who caught the throw, or "GROUND" if the ball hits the ground before anyone could catch it.

If N is 0, that indicates that there are no more games to process.

Program output: The output should consist of one one line for each game of 500 in the input. Each of these output lines should contain either "The winner is " followed by the name of the winner, or the text "There is no winner." if none of the players reached five hundred points.

Example input:

```
10
150 Mark
200 GROUND
75 Stephanie
100 GROUND
200 Stephanie
BANKRUPT Mark
325 Luis
400 Mark
JACKPOT GROUND
250 Stephanie
3
150 Mark
JACKPOT GROUND
50 Stephanie
0
```

Example output (corresponding to the input shown above):

```
The winner is Stephanie.
There is no winner.
```

Problem 4 — Lawn Mower Control

Susan has just purchased a new XQ7-1000 automatic lawn mower to mow her yard. These mowers work with the help of a control program which tells the mower how to move through the yard. Once the control program is activated, the mower follows the instructions therein, leaving a trail of beautifully cut grass and its owner free to enjoy a leisure-time activity of their choice. At the end of the program, the mower halts and turns itself off.

These mowers operate as stack-based machines where each instruction in the control program causes the mower to take some action. The stack of the mower is initially empty and can hold up to 100 integer values. The XQ7-1000 model supports the following instructions in its control programs:

Instruction	**Meaning**
push [value]	Pushes the given integer value onto the stack.
add	Pops the top two values off the stack and adds them, putting the result on the stack.
move	Pops the top value off the stack and moves the mower by that number of meters. If the value is positive, the mower should move forward; if negative, it should move backward.
turn	Pops the top value off the stack and turns the mower by that angle. The value will always be a multiple of 90. Positive values turn the mower to its left and negative values turn it to its right.
branch [line]	If the top value on the stack is non-zero, control transfers to the given line of the program. If the top value is zero, the instruction does nothing. This instruction does not modify the stack. Lines are numbered starting from 0.
halt	Stops the mower and ends the control program.

When the mower begins operation, it starts by executing the first line in the control program, and continues on executing them one by one until it halts. Along the way, the move and turn instructions will cause the mower to move throughout its environment.

Susan has written several of these control programs for the new mower, and wants to know where each one will leave the mower when they eventually end. Your job is to write a program that will read in a mower control program and report the final position of the mower once it has finished, relative to where it starts.

When the mower begins execution, it is at the (0, 0) origin point of a coordinate system, facing in the positive direction along the Y-axis. You may assume that the control program consists of fewer than 100 lines, and will always halt eventually. You may also assume that each branch instruction will specify a valid line number.

Program input: The input consists of multiple control programs. The first line of input for each program will consist of an integer N, giving the number of lines in the program. Following will be N lines, each giving one line of the control program in the format given above. If N is 0, that indicates there are no more control programs to test.

Program output: The output should consist of one line for each control program being tested. Each of these output lines should report the final position of the mower relative to where it started in (X, Y) coordinates.

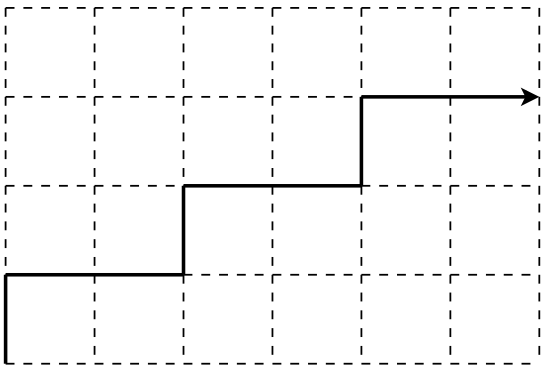
Example input:

```
13
push 3
push 1
move
push 270
turn
push 2
move
push 90
turn
push -1
add
branch 1
halt
0
```

Example output (corresponding to the input shown above):

```
Mower halts at (6, 3).
```

Visualization of Example input:



Problem 5 — Mystic: the Happening

Mystic: The Happening is a new card game, and the publishers have decided to use a novel approach to distributing its rare promo cards. In order to encourage players to keep buying their game, anyone who buys a pack of cards for the first time gets a copy of promo card number 1. Anyone who buys a pack of cards for a second consecutive day instead gets 2 copies of promo card number 2. Anyone who buys a pack of cards for a third consecutive day instead gets 3 copies of promo card number 3. In general, anyone who buys a pack of cards on the nth consecutive day gets n copies of promo card number n. Whenever a buyer does not buy a pack of cards on a particular day, they can no longer get any promo cards. The game designer is curious which rare promo card has the most copies out in circulation.

Program input: The first line of input indicates the number of data sets. Each data set is on its own line and consists of a series of integers between 1 and 200 (inclusive) separated by spaces. Each integer represents one buyer and indicates the number of consecutive days that the game-buyer received promo cards. There will be no more than 25,000 integers per line.

Program output: For each data set, output the card number that has the most copies in circulation. In the case of a tie, output the highest number involved in the tie.

Example input:

```
3
1 1 1 1 1 1 1 1 1 3
1 1 1 1 1 1 1 1 1 15
1 2 3 4 5 6 5 4 3 2 1 9
```

Example output (corresponding to the input shown above):

```
1
15
4
```

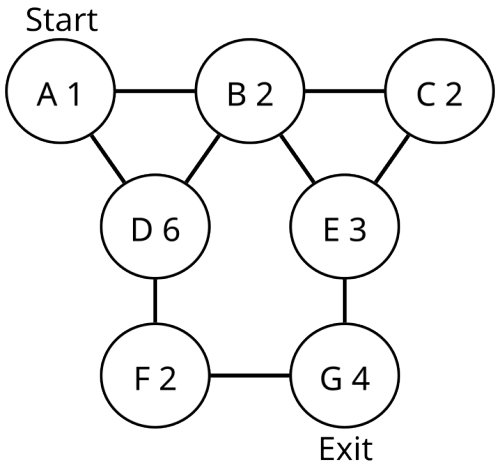
This page is intentionally left blank.

Problem 6 — Cave Escape!

Archaeologist Montana Smith is exploring a cave system, in search of a lost artifact. The cave system consists of a number of open rooms connected together by passageways. Unfortunately at some point Montana steps on a hidden pressure plate trap, which triggers a collapse in the cave system! Now he must escape before he is crushed in the ensuing cave in.

The rooms of the cave system collapse on themselves at different times. In his escape, Montana needs to reach the room containing the exit without going through a room which has already collapsed, or being in a room while it collapses. You need to write a program which will read in the structure of the cave system, including when each room collapses, and determine if and how he can reach the cave's exit.

Below is an example cave structure. Montana begins in room A and must get to the exit, room G. The numbers refer to the step after which the room collapses. So room A will collapse after Montana's first step, and room B after his second. Montana always takes 1 step to travel from one room to another.



Montana can reach rooms B and D immediately. If he goes to room D, he will not be able to reach the exit because room F will collapse after his second move (when he would reach room F), so he must start by going to room B. Likewise Montana cannot continue to room C, but must escape via rooms E, then finally G. This case appears as the sample input below.

It is possible that there will be no escape from the cave system. For example if room E were to collapse after step 2 instead of 3, then Montana would not be able to reach it in time and there would be no possible escape for him.

Program input: The first line of input is a number N giving the number of cave systems in the input set. Each of the N cave systems begins with an integer R giving the

number of rooms in the cave. Following are R lines, one for each room. Each of these lines contains an upper-case character which gives the name of the room, and an integer giving the step after which the room collapses. There will be no more than 26 rooms, and no room will collapse before time step 1. The first room listed is the one in which Montana begins, and the last room is the one he must reach in order to escape.

Next is a line containing an integer P giving the number of passageways in the cave system. Following that are P lines which give the passageways. Each of these lines consists of two characters, indicating which two rooms the passageway connects. The passageways are bi-directional and only listed for one direction -– so BC indicates that there is a passageway that can be used to travel from room B to C or from C to B.

Program output: You should output one line for each of the N cave systems in the input. If there is a way for Montana to escape the cave system, your program should print "Montana can escape: " followed by the sequence of moves by which Montana can escape the cave. The sequence should be given as a string of upper-case characters which indicate which rooms Montana travelled through (including the starting room and the exit). If there are multiple solutions, you may print any of them. Your program should output "There is no escape!" if there is no escape from a cave system.

Example input:

```
1
7
A 1
B 2
C 2
D 6
E 3
F 2
G 4
9
AB
AD
BC
BD
BE
CE
DF
EG
FG
```

Example output (corresponding to the input shown above):

```
Montana can escape: ABEG
```

Problem 7 — Constant expressions

Prefix expressions have three forms:

- A *literal* is a sequence of one or more digits, optionally preceeded by a minus sign ("−")
- A *variable reference* is a sequence of one or more lower case letters
- An *application* is a left parenthesis, followed by an *operator*, followed by two prefix expressions, followed by a right parenthesis

The operators which may be used in an application are `+`, `-`, `*`, and `/`. Applications represent arithmetic. For example, the prefix expression

```
(* 6 a)
```

represents the product $6 \times a$ (where a is a reference to a variable with an unknown value), and the prefix expression

```
(- (+ 4 x) y)
```

represents the computation $(4 + x) - y$ (where x and y are the values of variables.)

The tokens in a prefix expressions may be separated by space characters, although spaces are not necessary to separate two tokens if the second one begins with a character that the first can't contain. For example, in

```
(+ a b)
```

no space is required to separate `(` from `+`. However, at least one space is necessary to separate `a` from `b`, since `ab` is a single variable name.

Your task is to write a program which reads prefix expressions and transforms them so that all operations on literal values are replaced with the result of that operation. Note that this transformation should be recursive, so that any arbitrarily complex subexpression involving only literal values should be replaced with a single literal. Note that your program should not take algebraic equalities into account. For example, `(* x 0)` should not be replaced by `0`, even though $x \times 0 = 0$ for all values of x. All operations should be performed using integer arithmetic, and divisions should discard the fractional part of the result. For example, `(/ 7 2)` should be replaced by `3`. You do not need to check for attempts to divide by 0.

Program input: The input to the program is a sequence of (possible) prefix expressions, one per line. Note that some lines of input might not contain exactly one valid prefix expression.

Program output: For each input line, the program should print either (1) the transformed expression (with computations on literal values replaced with the computed literal value), or (2) the text `Invalid expression` if the line didn't contain exactly one valid prefix expression (as specified by the syntax description above.)

When printing an application, your program should print a single space between the operator and first subexpression, and also between the first subexpression and the second subexpression.

Your program should be able to correctly handle constant expressions with values in the range $-2^{63}..2^{63}-1$, inclusive.

Example input:

```
a
(* a (+ 2 3))
(* (/ vn (/ (- (/ 6 6) (- -5 -19)) (+ (* 17 1) 4))) 15)
(/ 4 (- (- (/ (/ 15 11) 4) fx) (/ -8 -2)))
(/ (- (+ -4 2) (* (* (* nq -9) (- 18 -16)) -6)) 6)
(+ -16 (- 19 -14))
(+ a 4
(* 2 3 4)
(* 2 3) a
ALL YOUR BASE ARE BELONG TO US
```

Example output (corresponding to the input shown above):

```
a
(* a 5)
(* (/ vn 0) 15)
(/ 4 (- (- 0 fx) 4))
(/ (- -2 (* (* (* nq -9) 34) -6)) 6)
17
Invalid expression
Invalid expression
Invalid expression
Invalid expression
```

Problem 8 — Zombie Cypher

Zed's dead. This has given him a lot of free time. His contemporaries are mostly into typical post-life activities like resting, rotting, and occasionally rising from their graves to wreak havoc upon the local populace, but Zed's no B-movie stereotype! Zed's a hacker, baby! Let's all be like Zed. Well, not the dead part.

Zed's cypher is simple, yet effective in fooling other post-life security teams. He takes a string as input and transforms it according to the following rules:

- If the character is a vowel, he replaces it with the next vowel in the sequence
- ('a' $\rightarrow$ 'e', 'e' $\rightarrow$ 'i', 'i' $\rightarrow$ 'o', 'o' $\rightarrow$ 'u', 'u' $\rightarrow$ 'a').
- If the character is a consonant, he replaces it with the next consonant in alphabetical order ('z' wraps around to 'b').
- If the character is not a letter, he leaves it unchanged.

Program input: The input is one line with N, the number of test cases. The next N lines include one spooky word, each. Word length can be between 1 and 100, and can include any characters except white space.

Program output: Each spooky word transformed into less-spooky cipher, one line each.

Example input:

```
4
CCSCE
SpookySkeletons
TrickOrTreat!
Haunted_House
```

Example output (corresponding to the input shown above):

```
DDTDI
TquulzTlimiuuot
UsodlUsUsieu!
Ieaouie_Iuati
```

This page is intentionally left blank.

Problem 9 — Chant GPT

In the eerie town of Bitsburg, the moon was always full, the owls hooted ominously, and the code... well, the code was downright spooky. In a secluded castle on a hill, lived Dr. Frankencode, a spellhacker of terrifying talent. He had a pet project that sent shivers down the spine of every motherboard in town – a downright evil AI spellhacking language known as Chant GPT (trademark pending).

Dr. Frankencode had designed Chant GPT's language to be as simple to understand as the doctor himself. It was a language of matching letters, each represented by letter (A-Z). Every letter comes in a spine-chilling pair. A capital has a matching triplet of lower case letters (A is spine-chillingly paired with aaa, B with bbb, and so on). X is an exception, as it can only be used by itself, nested inside one of these spine-chilling pairs, but it's not as spine-chilling itself. Spine-chilling pairs can also nest inside other spine-chilling pairs if they are nested alphabetically, or if they do not follow a similarly nested X.

For example, Aaaa is a valid Chant GPT as A is matched with aaa (spine chilling!). Same for ABbbbaaa or AaaaBbbb (using A and B). ABbbbXaaa and ABXbbbaaa are beautiful too, especially when put to the melody of Don't Stop Believin' in Ghosts, by Evil Journey.

Your task is to step into Dr. Frankencode's terrifying shoes and validate proposed Chant GPT's. Print "Treat" when chants are disturbingly valid and "Trick" when they aren't. Be sure to tread carefully. In Bitsburg, even the bugs have bugs.

Program input: The input is one line with N, the number of chants to test. The next N lines include one terrifying chant each. Chant length can be between 1 and 100, and can include only letters, no other characters.

Program output: Treat for valid chants, trick for invalid chants.

Example input:

```
4
ABbbbXaaa
CcccBbbbAaaa
ABXCcccbbbaaa
CBAaaabbbccc
```

Example output (corresponding to the input shown above):

```
Treat!
Treat!
Trick!
Trick!
```

This page is intentionally left blank.