2022 CCSC Eastern Conference Programming Competition

October 22nd, 2022

DeSales University, Center Valley, Pennsylvania

This page is intentionally left blank.

Problem 1 — Letter Substitution

Phrases can often be improved by substituting occurrences of one letter for another. For example, it seems pretty clear that rather than starting with "Four score and seven years ago", the Gettysburg Address would have sounded better starting with "Youv scova and savan yaavs ago". As every English major knows, "y" is a much better letter than "f", and any sentence can be spiced up considerably by replacing "e" with "a". And of course "v" is a more exciting sound than "r". To make it easier to make improvements of this nature, you decide write a program to substitute occurrences of one letter with another.

Program input: The first line of input contains a number of phrases that the program will transform. In each pair of subsequent lines, the first is a phrase, and the second is a substitution string. You may assume the substitution string will consist of only letters (a–z or A–Z).

Program output: For each phrase accompanied by a substitution string with an even number of letters, the program should print out the transformed phrase. The transformed phrase is produced as follows: for each letter pair in the substitution string, every occurrence of the first letter should be replaced with an occurrence of the second. Case should be ignored in the substitution string, but it should be preserved when transforming the phrase. I.e., lower case letters in the original phrase should remain lower case in the transformed phrase, and likewise for upper case letters. If there are multiple pairs specifying the same original letter, the last pair in the substitution string should be the one that is used. As a special case, if the substitution string had an odd number of letters, the program should print a line with the output `Wut`.

Example input:

```
4
Four score and seven years ago
fyrvea
Hello world
HmdT
Every day, you must say, "So how do I feel about my life?"
yiio
A strong smell of petroleum prevails throughout
aiouq
```

Example output (corresponding to the input shown above):

```
Youv scova and savan yaavs ago
Mello worlt
Everi dai, iou must sai, "So how do O feel about mi lofe?"
Wut
```

This page is intentionally left blank.

Problem 2 — Word Tower

A *word tower* is a sequence of words such that each word (other than the first) is formed by rearranging the letters in its predecessor and adding one or more additional letters anywhere in the word. For example:

```
        moo
       mono
       moons
      someones
     moroseness
    enormousness
overnumerousness
```

Your task is to write a program to check a list of words to see whether they could form a word tower.

Program input: The first line of input specifies the number of lists of words the program should check. Each subsequent line contains a list of words to check, with each word separated by at least one space character. Note that the words will not be in any particular order. You may assume that words will consist of only lower-case letters.

Program output: If a word list contains words that could form a valid word tower, the program should print a line with the text `yes`. If not, it should print a line with a reason message in one of the following forms:

P `and` *Q* `are the same length`
Q `is missing` *C* `(needed for` *P*`)`

In these messages, *P* and *Q* are a predecessor word and a successor word, and *C* is the earliest character in the alphabet such that *P* contains the letter but *Q* does not. The reason message should identify the words *P* and *Q* such that if the entire sequence of words is sorted by length from shortest to longest, using lexicographical order to break ties, *P* and *Q* would be the first pair violating the requirements. If *P* and *Q* are the same length, the first message should be printed (even if *P* contains a character that *Q* does not.)

Example input:

```
4
mono moons moroseness someones overnumerousness moo enormousness
a cat tack trace stacktrace
orates rotates arose soar toaster
roseate tear eaters star
```

Example output (corresponding to the input shown above):

```
yes
trace is missing k (needed for tack)
rotates and toaster are the same length
star and tear are the same length
```

Problem 3 — Football Scores

In the game of American football, there are different ways to score that are worth different numbers of points. Specifically[1]:

- A safety is worth 2 points
- A field goal is worth 3 points
- A touchdown without an extra point is worth 6 points
- A touchdown with an extra point is worth 7 points
- A touchdown with a two-point conversion is worth 8 points

This means that to score a certain number of points, there are generally many ways to score that number of points. For example, to score a total of 16 points, possibilities include

- a touchdown with and extra point, and three field goals
- two touchdowns, each with a two-point conversion
- eight safeties

Overall, there are 14 different ways to score 16 points.

Program input: Each line of input specifies an integer point total. You may assume that the value will not be greater than 100. As a special case, if a line has a negative value, the program should exit without printing further output.

Program output: For each input line, the program prints a line consisting of

1. the number of different ways that point total could be accomplished, and
2. if there is at least one way to achieve the point total, a space followed by the scoring combination that has the smallest number of scoring plays, as a comma-separated list of point values, in curly braces, in order from least to greatest

Note that the *order* in which the scoring plays happen is not significant. For example, a field goal followed by a safety is the same as a safety followed by a field goal, so there is only one possible way to score 5 points. If there are multiple candidates for smallest number of scoring plays, the one that is first in lexicographical order should be printed.

Example input:

```
0
1
5
17
-1
```

[1]Please ignore the possibility of the defensive team blocking an extra point attempt and running it back to the opposite end zone, which technically would allow a single point to be scored.

Example output (corresponding to the input shown above):

```
1 {}
0
1 {2,3}
14 {2,7,8}
```

Problem 4 — Constants

The FROBOZZ 2000™ computer architecture supports four instructions which operate on 32-bit signed integer values: `add`, `sub`, `mul`, and `div`. Each of these instructions stores a value in a destination register (the first operand of the instruction) which is computed from two source operands (the second and third operands of the instruction.) Source operands can be registers (`r1`, `r2`, etc.) or integer constants prefixed by "`$`" (e.g., "`$1`" representing the constant integer value 1.) Here is a very simple code example:

```
add r1, $0, $3
add r2, $0, $5
add r3, r1, r2
```

After this code is executed, the `r1` register will contain the value `3`, the `r2` register will contain the value `5`, and the `r3` register will contain the value `8`.

Your task is to write a program which will determine which registers have known constant values in them when a sequence of instructions finishes executing. Your program should consider a register to hold a constant value in the following three situations:

1. It was computed from literal constant values, e.g. `add r1, $0, $3`
2. It was computed from a literal constant value and a register with a known constant value, e.g. `sub r2, $42, r1` assuming that `r1` holds a known constant value
3. It was computed from two registers with known constant values, e.g., `mul r3, r1, r2` assuming that both `r1` and `r2` have known constant values

Your program should *not* attempt to take algebraic identities into account. For example, the instruction `sub r2, r1, r1` would be guaranteed to place the value `0` in `r2`. However, `r2` should not be considered constant in this case.

All arithmetic should be done on 32-bit signed integer values. Integer division should truncate, i.e., $7/3 = 2$. The order of the source operands is significant for the `sub` and `div` instructions. For example, the instruction `sub r5, $10, $3` would result in the `r5` register containing the value 7. Your program should allow up to 64 registers (`r1` through `r64`) to be used.

Program input: The input is a sequence of instruction sequences. Each instruction sequence begins with a single line specifying an integer N. This line is then followed by N instructions as described above. As a special case, if N is less than 0, then the program should exit immediately without producing further output.

Program output: For each input instruction sequence, your program should output a sequence of registers (in sorted order) known to have constant values, along with the value of each register, in the format `r`*X*=*V*, where *X* is the register number, and *V* is the constant value in that register. Each know register value should be separated by a single space character. As a special case, if no registers are known to contain constant values at

the end of an input instruction sequence, your program should output a line with the text "`No known constants`".

Example input:

```
3
add r1, $0, $3
add r2, $0, $5
add r3, r1, r2
3
mul r1, r2, $5
sub r6, r1, $1
add r7, r6, $3
4
add r6, $11, $6
mul r7, r5, r6
div r8, r6, $2
mul r9, r8, r8
2
add r1, $0, $-1
mul r2, $4, r1
-1
```

Example output (corresponding to the input shown above):

```
r1=3 r2=5 r3=8
No known constants
r6=17 r8=8 r9=64
r1=-1 r2=-4
```

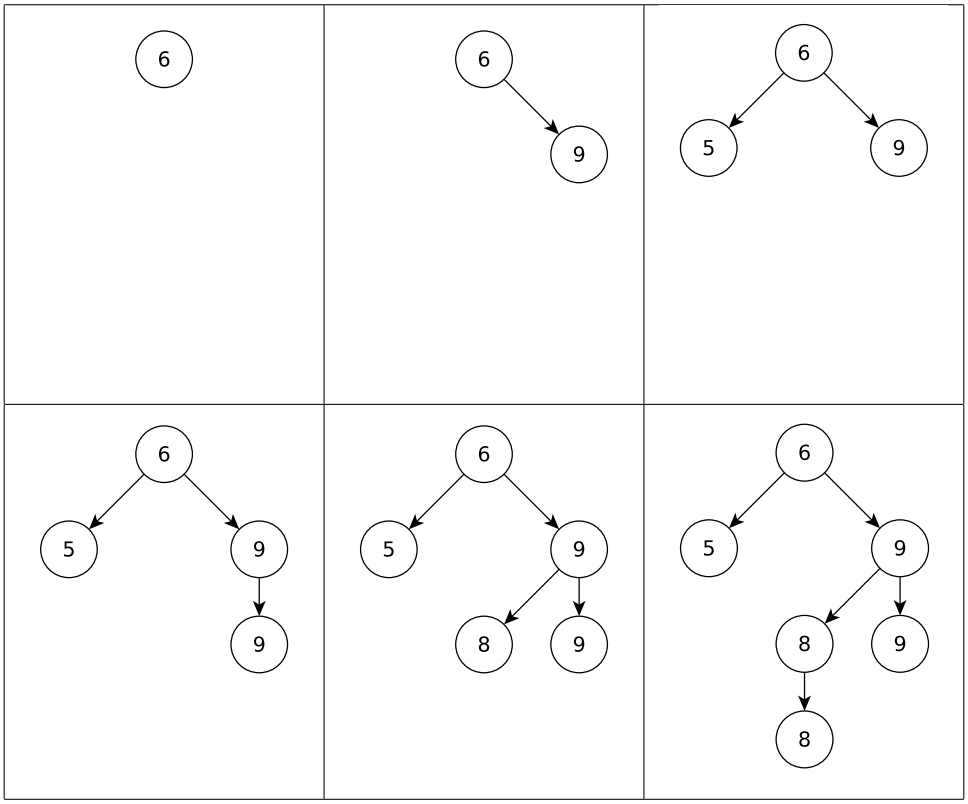
Problem 5 — Tie-nary Trees

As a CS major on Melmac, Alf learned all about Binary Search Trees and even a little about Ternary Search Trees. Being the enterprising little alien he is, Alf decided to invent his own type of Search Tree. The Tie-nary Search Tree.

A Tie-nary Search Tree (TST) is a tree data structure in which each node has at most three children, which are referred to as the left child, the middle child, and the right child. With a TST, the value stored in the left child of any node is strictly less than the value stored in the node, and the value stored in the right child of any node, is strictly greater than the value stored in the node. So, what happens when two equal values need to be stored in a TST?

Good question! That is where the middle child comes into use. The value stored in the middle child of any node is equal to the value stored in the node.

For example, if we insert the sequence of values 6 9 5 9 8 8 into an empty TST the following sequence (left to right, top to bottom) of trees would be created:



Note that the height (length of longest path from root to leaf) of the final tree in this case is 3. Your job is to write a program that calculates the height of the final TST, given a sequence of nodes to be inserted into an empty TST.

Program input: The first line of input contains an integer greater than 0 and less than 100 that indicates how many node sequences are represented in the rest of the input. This is followed by information about each node sequence in turn.

Program output: For each node sequence output a line indicating the height of the resultant TST.

Example input:

```
4
6 9 5 9 8 8
17
3 3 3 3 3 3 3 3 3 3 3
10 5 15 3 1 2 13 17 8 8 8
```

Example output (corresponding to the input shown above):

```
3
0
10
4
```

Problem 6 — Combos™ Combos

Edgar's favorite snack is Combos™, especially the peanut-butter filled pretzels. He likes to mix up Combos™ from different bags. As a Combos™ superfan and president of the North American Combos™ Appreciation Society (NACAS), Edgar has access to little-known flavors such as Anchovy, Marmite, and WhiteTruffle. He wonders how many different ways there are to eat a mouthful of different types of Combos™. Then he wonders if he could make a sign describing that combination. He may not have enough letters for his sign.

Program input: The first line of input contains the number of data sets. Each other line contains a data set consisting of words, where each word describes a flavor of Combos™. You may assume that a line will not have more than 20 words.

Program output: For each data set, output the number of possible combinations of half of those items (rounded up), if order does not matter. After that, on the same line and after a colon, output the combination that contains the fewest total number of characters over all words, with the items in lexicographic order and separated by a space. In the case of ties for word length, choose the word that comes first lexicographically. Lexicographical comparisons of words should be case-sensitive using ASCII/Unicode order, so that "A" would come before "a".

Example input:

```
2
CheeseCracker PeanutButterPretzel PizzeriaPretzel
Marmite Herring MeatLovers Avocado PeanutButterAndJelly
```

Example output (corresponding to the input shown above):

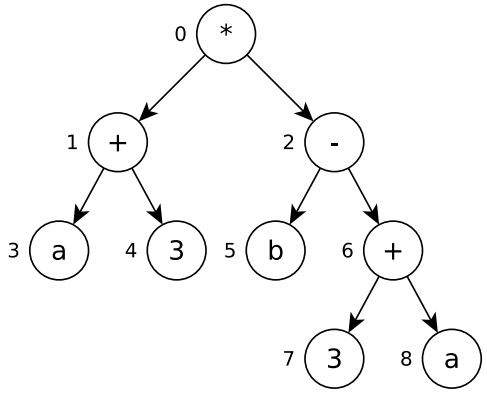
```
3:CheeseCracker PizzeriaPretzel
10:Avocado Herring Marmite
```

This page is intentionally left blank.

Problem 7 — Common Subexpressions

Trees are commonly used to represent *expressions* which perform a computation on values. For example, below are shown an expression performing a computation on the values `a`, `b`, and `3`, and the corresponding expression tree (in which each node is labeled with a unique integer identifier):

```
(a + 3) * (b - (3 + a))
```



Because addition is commutative, the nodes with identifiers 1 and 6, representing the expressions `a + 3` and `3 + a`, are guaranteed to compute the same result.

Your task is to write a program to analyze a tree representing an expression, and determine which pairs of nodes are guaranteed to compute the same value. The operators allowed are `+`, `-`, `*`, and `/`. Your program should assume that `+` and `*` are commutative, but that `-` and `/` are not. Note that your program should *not* take algebraic identities into account. For example, the expression `a - a` should not be considered equivalent to 0, the expression `a + 0` should not be considered equivalent to `a`, etc. In addition, your program should not treat operations on constant values as being constant. For example, it should not treat `1 + 2` as being equivalent to `3`. Expressions are equivalent if and only if they either

- directly represent the same variable or literal value, or
- represent the same operation on two subtrees that are equivalent (taking commutativity into account for `+` and `*`)

Program input: The first line of input specifies the number of test cases. Each test case is a single line which starts with the integer identifier of the root node, followed by a description of each node in the expression tree. A node description has one of the following forms:

- *id*`:`*literal*
- *id*`:`*varname*
- *id*`:`*operator*`,`*id*`,`*id*

An *id* is the non-negative unique integer identifier of a node. *literal* will be a literal integer value. *varname* will be a single lower-case character representing a variable name. Note that the node descriptions should not be assumed to be in any particular order.

Program output: For each test case, the program should print an output line indicating each set of nodes which are equivalent, separating each set with a comma. Each set should be printed as a comma-separated sequence of node *id* values, in order from least to greatest, enclosed by curly braces. The program should not print sets containing only a single node. The sets should be ordered by the *id* of their first member, from least to greatest. As a special case, if no node is considered equivalent to any other node, the program should print a line with the text `No equivalent nodes`.

Example input:

```
3
0 1:+,3,4 2:-,5,6 3:a 4:3 5:b 6:+,7,8 7:3 8:a 0:*,1,2
7 1:a 4:4 6:b 5:-,6,3 7:*,2,5 2:+,1,4 3:5
0 10:b 1:3 2:3 3:b 4:a 5:+,10,1 6:+,2,3 7:a 8:*,4,5 9:*,6,7 0:-,8,9
```

Example output (corresponding to the input shown above):

```
{1,6},{3,8},{4,7}
No equivalent nodes
{1,2},{3,10},{4,7},{5,6},{8,9}
```

Problem 8 — Beesanese

Bees! Did you know that they communicate? No no, not by dancing, where did you hear that? My cousin Tancred told me that he heard that bees communicate via an unnecessarily complicated translation algorithm, and he also told me that he's a bee scientist. He calls this language Beesanese.

According to Tancred, bees understand English words, and we just need to learn how to buzz them the same way! Bees pronounce most consonants as with a "Buzz!" and vowels with two of these. Pretty simple, but they get especially excited when they see a "z". When they first see one, they give a quieter "bzz", but they can't hold in their excitement and follow up with a full "Buzz!" later.

Translate all the words!

Program input: A list of words in lower case, one word at a time. Words aren't longer than 10 letters and are at least 1.

The first line of the input file is an integer, T, the number of test cases. (1 <= T <= 1000) Each case is one word.

Program output: Translate each word into Beesanese.

Example input:

```
4
cow
buzz
zigzag
zzz
```

Example output (corresponding to the input shown above):

```
Buzz!Buzz!Buzz!Buzz!
Buzz!Buzz!Buzz!bzzbzzBuzz!Buzz!
bzzBuzz!Buzz!Buzz!bzzBuzz!Buzz!Buzz!Buzz!Buzz!
bzzbzzbzzBuzz!Buzz!Buzz!
```

This page is intentionally left blank.

Problem 9 — Zombiemon Go!

Zombie facts:

- Zombies are mammals.
- Zombies smell.
- Zombie infestations should be contained.

Wastelander Bette has a problem. She's been surviving the post-apocalyptic badlands for a number of years now, and having watched far too much Pokeymans, has taken to collecting zombies as she comes across them. She's proud of her zombie collection and want to show them off, but as we know from our zombie facts, we have to be careful of how we store them. When zombies get too close together, they become dangerous, and more importantly, smell like somebody dropped their dirty socks into a mud pit full of old cheese at the gym.

Bette has a sensitive nose, so she would like to spread her zombies out. She's planning her zombie show and plans to store them in rows of stalls in her zombie barn. She'd like help figuring out how many different ways she can arrange her zombies to minimize the smells. We don't want zombies in adjacent stalls because the smell is just too overwhelming, but otherwise any arrangement works. For example, if there are 2 stalls, there are 3 valid arrangements: `|-|X|`, `|X|-|`, and `|-|-|`. A single zombie can sit in either stall, but the stalls are too close for two zombies.

Program input: The first line of input is T, the number of test cases. (1 <= T <= 100). Each other line represents a single case. Each test case is just an integer N (1 <= N <= 25). N represents a set of stalls in a row. 3 means there are three stalls side by side.

Program output: The answer to each input should be a single integer, representing the total number of valid zombie arrangements in a set of stalls.

Example input:

```
4
1
2
3
7
```

Example output (corresponding to the input shown above):

```
2
3
5
34
```

This page is intentionally left blank.