Section Handout

Problem 1: Keith Numbers

A Keith number is any k-digit number that appears in the Fibonacci-like sequence that starts off with the number's k digits and then continues such that each subsequent number is the sum of the preceding k.

All of the one-digit numbers are—trivially so—Keith numbers. Yawn.

The number 7385 is more interesting. It's a Keith number, because the following sequence says so:

7, 3, 8, 5, 23, 39, 75, 142, 279, 535, 1031, 1987, 3832, 7385

The sequence starts out 7, 3, 8, 5, because those are the digits making up 7385. Each number after the 5 is the sum of the four numbers that precede it (four, because 7385 has four digits). The fact that 7385—the number whose digits spawned it all—happens to be in the sequence is the happy accident that tells us it's a Keith number.

Write a function called **`isKeith`**, which returns **`true`** if and only if the supplied integer is a Keith number.

```
static bool isKeith(int n);
```

Problem 2: Publishing Stories

Social networking sites like Facebook, Twitter, and LinkedIn typically record and publish stories about actions taken by you and your friends and followers. Stories like:

Jessie Duan accepted your friend request.
Matt Anderson is listening to Green Day on Spotify.
Patrick Costello wrote a note called "Because Faiz told me to".
David Wang commented on Jeffrey Spehar's status.
Mike Vernal gave The French Laundry a 5-star review.

are created from story templates like

{name} accepted your friend request.
{name} is listening to {band} on {application}.
{name} wrote a note called "{title}".
{name} commented on {target}'s status.
{actor} gave {restaurant} a {rating}-star review.

The specific story is generated from the skeletal one by replacing the tokens—substrings like **"{name}"**, **"{title}"**, and **"{rating}"**—with event-specific values, like **"Jessie Duan"**, **"Because Faiz told me to"**, and **"5"**. The token-value pairs can be packaged in a **Map<string, string>**, and given a story template and a data map, it's possible to generate an actual story.

Write the **generateStory** function, which accepts a story template (like **"{actor} gave {restaurant} a {rating}-star review."**) and a **Map<string, string>** (which might map **"actor"** to **"Mike Vernal"**, **"restaurant"** to **"The French Laundry"**, and **"rating"** to **"5"**), and builds a string just like the story template, except the tokens have been replaced by the text they map to.

Assume the following is true:

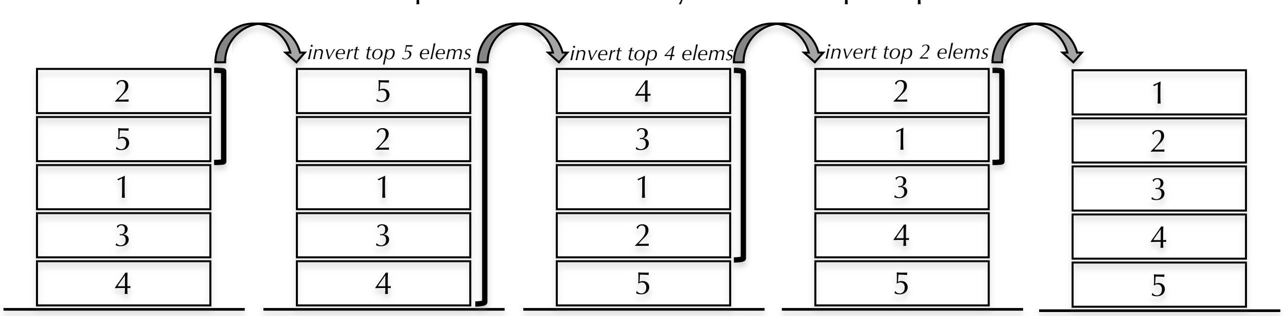
- **'{'** and **'}'** exist to delimit token names but won't appear anywhere else. In other words, if you encounter the **'{'** character, you can assume it marks the beginning of a token that ends with a **'}'**.
- We guarantee that all tokens are in the **Map<string, string>**. You don't need to do any error checking.

The prototype is:

```
static string generateStory(const string& storyTemplate,
                            const Map<string, string>& data);
```

Problem 3: Topswopping and Topswop Numbers

When a **Stack<int>** of depth n contains the numbers 1 through n in some order, its **Topswop number** is defined to be the number of times the top of the stack must be **swopped** before the top element becomes a 1. Each swop amounts to an examination of the topmost element of the stack—let's call it k—and then the inversion of the top k elements. (A stack whose top element is already 1 has a Topswop number of 0.)



Implement the **getTopswopNumber** function, which accepts a copy of a **Stack<int>** and returns the number of times the **Stack<int>** must be swopped before the top element becomes a 1. You may assume the **`Stack<int>`** contains the numbers 1 through n in some order, and you can trust the fact that the process always terminates.

```
static int getTopswopNumber(Stack<int> s);
```

Problem 4: Shunting-Yard Algorithm

The **shunting-yard** algorithm is an algorithm one can use to convert traditional, infix arithmetic expressions to postfix ones. For simplicity, we'll assume the operands are all single digit numbers and that the only two arithmetic operators are + and *. * is of higher precedence than +, but parentheses can be used to override that. Because the operands are constrained to be single digit, expressions can be codified as strings, as with **`"(3+1)*(8+2)"`**, no extraneous spaces. Like the RPN algorithm discussed in lecture, shunting-yard is stack-based.

Here's a short program and some sample input and output to illustrate what needs to happen:

```
int main() {
   while (true) {
      string infix = trim(getLine("Enter an infix expression: "));
      if (infix.empty()) break;
      cout << infix << " --> " << infixToPostfix(infix) << endl;
   }
   return 0;
}
```

```
Enter an infix expression: 3+6↵
3+6 --> 36+
Enter an infix expression: (1+3)*3↵
(1+3)*3 --> 13+3*
Enter an infix expression: 1+3+1+1*3↵
1+3+1+1*3 --> 13+1+13*+
Enter an infix expression: 4*(1+3)*(6+4+2)↵
4*(1+3)*(6+4+2) --> 413+*64+2+*
```

Here's the recipe your **`infixToPostfix`** implementation will follow:

```
for each ch in infix:
   if ch is a digit
      append ch to output string
   if ch is an open parenthesis
      push that open parenthesis onto stack
   if ch is a close parenthesis
      pop operators off stack, appending to the output string
      repeat until open parenthesis encountered, which should be discarded
   if ch is an operator
      while stack top is op of equal or higher precedence, pop and append to output
      push ch onto stack
once infix has been scanned, drain stack, appending popped items to output
```

Present an implementation of **`infixToPostfix`**, and assume the supplied expression is well formed so that no error checking is required.s

```
static string infixToPostfix(const string& infix);s
```