Solution 1: Shunting-Yard Algorithm [10 points]

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)". 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:

```cpp
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  -->  133*
Enter an infix expression: 1+(3*3)
1+(3*3)  -->  133*
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 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

Using the next page to implement `infixToPostfix`, and assume the supplied expression is well formed so that no error checking is required. Feel free to rip this page out so you can easily refer to it.
Solution 1: Shunting-Yard Algorithm [continued]

```c++
static bool isOperator(char ch) {
    return ch == '+' || ch == '*';
}

static int precedence(char op) {
    return op == '+' ? 1 : 2;
}

static bool shouldReduce(const Stack<char>& stack, char op) {
    return !stack.isEmpty() && precedence(stack.peek()) >= precedence(op);
}
```

// to reduce is to pop something from the stack and append it to the output

```c++
static string infixToPostfix(const string& infix) {
    string postfix;
    Stack<char> stack;
    for (size_t i = 0; i < infix.size(); i++) {
        if (isdigit(infix[i])) {
            postfix += infix[i];
        } else if (isOperator(infix[i])) {
            while (shouldReduce(stack, infix[i]))
                postfix += stack.pop();
            stack.push(infix[i]);
        } else if (infix[i] == '(') {
            stack.push(infix[i]);
        } else {
            while (stack.peek() != '(')
                postfix += stack.pop();
            stack.pop();
        }
    }
    while (!stack.isEmpty())
        postfix += stack.pop();
    return postfix;
}
```

Criteria for Problem 1: 10 points

- Correctly declares the output `string`, the operator stack, and returns some string at the end: 1 point
- Correctly marches over all characters of the input `string`: 1 point
- Immediately appends a digit characters to the end of the running output `string`: 1 point
- Correctly identifies that a character is one of the two operators: 1 point
- Correctly determines whether the stack is empty as part of what I include in `shouldReduce`: 1 point
- Correctly reduces while the stack top is an operator of equal or higher precedence: 2 points
- Correctly pushes a ')' onto the stack to mark the beginning of a sub-stack: 1 point
- Correctly drains the sub-stack when encountering a ')', appending popped operators in LIFO order until sub-stack is drained: 1 point
- After input has been full processed, drain stack of all remaining operators: 1 point
Solution 2: Autocorrect [10 points]

We all know that when our big thumbs type out big words on our smart phones, we mistype and spell some words incorrectly. We also know the phone itself presents one or more words it thinks we meant to type. If, for instance, we’re texting and type out "tounf", the phone might suggest "young", because it knows that "tounf" isn’t a word but that 't' is right next to 'y' and 'f' is right next to 'g' on the keypad. This particular suggestion required two changes, but there aren’t any words in the English language that are one character away from "tounf", so "young" is a reasonably good suggestion (as are "round" and "found").

Implement the recursive ls function (ls is short for listSuggestions), which given a string, lists all of the words in the English language that require no more than a threshold number of substitutions. Your implementation should code to the following prototype:

    static void ls(const string& str, const Lexicon& english, const Map<char, string>& alternatives, int maxChanges);

str may or may not be a word in the English language, but if it is, it should be printed. Other words in the language should be printed if they require at most maxChanges letters to be replaced by their neighbors. alternatives has 26 keys—one for each lowercase letter—and each maps to a string of all of the keyboard letters immediately adjacent to it—that is, what we consider reasonable alternatives. For example, 'g' maps to "tyfhcvb", because those seven letters represent what a big thumb might have intended to hit when it tapped the 'g'.

Use the next page to present your implementation, and feel free to tear this page out so you can easily refer to it.
Solution 2: Autocorrect [continued]

```cpp
static void ls(const string& prefix, const string& suffix,
               const Lexicon& english, const Map<char, string>& map, int max) {
    if (max < 0 || !english.containsPrefix(prefix)) return;
    if (suffix.empty()) {
        if (english.contains(prefix)) cout << prefix << endl;
        return;
    }

    string rest = suffix.substr(1);
    ls(prefix + suffix[0], rest, english, map, max);
    for (const char& ch: map[suffix[0]]) {
        ls(prefix + ch, rest, english, map, max - 1);
    }
}

static void ls(const string& str, const Lexicon& english,
               const Map<char, string>& alternatives, int maxChanges) {
    ls("", str, english, alternatives, maxChanges);
}
```

Criteria for Problem 2: 10 points

- Implements `ls` as a wrapper that introduce the empty `string` as running prefix we’ve committed to starting out: 1 point
- Properly rejects anything that’ll require more than the allowed corrections: 2 points
- Immediately or eventually rejects a running prefix unless it’s a prefix of some word: 1 point (my base case prune isn’t strictly necessary—it’s just more efficient)
- When `suffix` is empty, returns regardless: 1 point
- When `suffix` is empty, conditionally prints the running prefix if it happens to be a word: 1 point
- Properly recurs on the `suffix.substr(1)` without consuming one of the allowed changes by sticking to the leading character of `suffix`: 2 points
- Properly iterates over all of the alternatives, extending the running prefix with each alternative for a recursive call, but imposing a change penalty: 2 points (if they interpret the problem that allows an alternative to itself be replaced by one of its own alternatives, then that’s fine, but still impose a penalty and require they recur on `suffix`, not `rest`). Don’t worry about whether or not some words are printed multiple times.
Solution 3: Regular Expressions and String Matching [20 points]

A regular expression—or regex, for short—is a string used to pattern match words in the English language. The simplest regular expressions consist of just lowercase letters, but they’re also allowed to contain one or more character sets like \[a-z\], and the presence of \[a-z\] in a regular expression matches any lowercase letter. Here’re a few examples of regular expressions and the English words that match them:

\[
\begin{align*}
\text{regex} & \quad \text{matches} \\
\text{and} & \quad \text{and} \\
[a-z]\text{lur} & \quad \text{blur, slur} \\
wil[a-z] & \quad \text{wild, wile, will, wily, wily} \\
m[a-z][a-z]\text{m} & \quad \text{maim, malm, marm, mumm} \\
x[a-z][a-z][a-z]\text{x} & \quad \text{xerox} \\
[a-z]\text{x[a-z]} & \quad \text{axe, exo, o xo, oxy}
\end{align*}
\]

The notion of a character set can be generalized to specify one or more smaller ranges to represent sets of lowercase letters, as with:

\[
\begin{align*}
\text{character set} & \quad \text{possible characters} \\
[a-g] & \quad \text{abcdefg} \\
[c-gm-w-z] & \quad \text{cdefgmxwxyz} \\
[a-e-i-o-u] & \quad \text{aeiou} \\
[x-z-a-b-p] & \quad \text{abpxyz}
\end{align*}
\]

Note that isolated characters can sit among zero or more ranges to compactly express a small set of characters, as I do with the three of the four sample character sets above. This notation allows us to match a more constrained set of English words:

\[
\begin{align*}
\text{regex} & \quad \text{matches} \\
m[a-e-i-o-u][x-z] & \quad \text{max, may, mix, miz, moy, moz, mux} \\
z[a-c-o-r-z][a-g-k-n-p] & \quad \text{zag, zap, zoa, zoo} \\
[a-c][d-g][h-m][n-q][r-z] & \quad \text{adios, agios, aglow, below}
\end{align*}
\]

Finally, an asterisk (i.e., one of these things: ‘*’) can follow any character or character set as an instruction that the single character or character set preceding it can be skipped and go unmatched, be matched exactly once, or be matched an arbitrarily large number of times.

What can regexes look like now, and what strings do they match? Here are some examples:

* \text{aa}[a-z]\text{*} matches all those words that begin with aa, including aa, aah, aahed, aardvark, aardvarks, aarti, and aasvogel. The \text{[a-z]*} portion of \text{aa[a-z]*} can match the empty string, a single letter, or an arbitrary string of length 2 or more.
• \([a-z]*zz[a-z]*\) matches all of those words that contain a zz somewhere, including buzz, jacuzzi, pizzelle, sizzle, spazzing, zyzyvas, and zzzs.

• \([a-z]*zz[a-z]*zz[a-z]*\) matches all of those words containing two independent double z’s. This list is pretty small, but it’s nonempty! It matches exactly 11 words, and bezzazz, pizzazz, and razzamatazz are among them.

• \([a-g]\)* matches all those words that can be formed using just the first seven letters of the alphabet, including begged, cabbage, deface, defaced, feedbag, and gaffed. Musicians love these words, because they can be formed using just the notes of a C major scale.

• \([aeiou][aeiou][aeiou][aeiou]*\) matches all of the English words of length 3 or more that contain only the five principal vowels.

• \([a-z]*a[a-z]*e[a-z]*i[a-z]*o[a-z]*u[a-z]*y[a-z]*\) matches the six words that contain all six vowels (this time counting y) where a, e, i, o, u, and y appear in that order. Congratulations to abstemiously, adventitiously, autoeciously, facetiously, halfseriously, and sacrilegiously for being part of this distinguished set.

For this problem, you’ll be led through the decomposition of a recursive function called \texttt{matches} that decides whether a regex matches a string of lowercase letters (presumably a word in the English language). Over the course of the problem, we’ll confirm you’re fluent with C++ strings and pass-by-reference.
a) [5 points] Your implementation of matches should benefit from a helper function called `expand`, which takes a single character set and returns a sorted string of all of the lowercase letters it expands to, as with:

<table>
<thead>
<tr>
<th>set</th>
<th>expand(set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a-g]</td>
<td>abcdefg</td>
</tr>
<tr>
<td>[x-ya-g]</td>
<td>abcdefgxy</td>
</tr>
<tr>
<td>[a-empw-z]</td>
<td>abcdempwxyz</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>aeiou</td>
</tr>
<tr>
<td>[a-ea-ed-fa-eeeee]</td>
<td>abcddef</td>
</tr>
</tbody>
</table>

Your implementation should be able to handle redundancies like those you see in the last example above, and the string of lowercase letters returned should be sorted in lexicographic order. You should assume that the first character is always ' [ ', the last character is always ' ] ', there's at least one character between the ' [ ' and the ' ] ', and that the character set identifies only lowercase letters and is otherwise well formed. This part doesn’t involve recursion, but it will test your ability to manipulate strings.

Use the rest of this page for your implementation:

```cpp
class Set {
public:
    Set() { set<char>().clear(); }

private:
    set<char> charset;
};

static string expand(const string& set) {
    Set<char> charset;
    size_t i = 1;
    while (i < set.size() - 1) {
        char low = set[i];
        char high = set[i + 1] == '-' ? set[i + 2] : low;
        i += set[i + 1] == '-' ? 3 : 1;
        for (char ch = low; ch <= high; ch++) charset += ch;
    }

    string characters;
    for (char ch: charset) characters += ch;
    return characters;
}
```

Criteria for Problem 3a: 5 points
- Correctly ingests an isolated character (e.g. the m in [a-emq-z]): 1 point
- Correctly ingests a range of characters (e.g. the a through e in [a-emq-z]): 1 point
- Correctly advances either 1 or 3 characters at a time, depending on what was ingested: 1 point
- Returns a sorted version of everything that was ingested: 2 points
b) [5 points] Your implementation of `matches` will also benefit from a second helper function called `split`, which takes a nonempty regular expression and pulls off the portion that might be matched by a word’s first character. Here’s the interface you’re coding to:

```cpp
static void split(const string& regex, string& first,
                   bool& starred, string& rest);
```

Assuming that `first` and `rest` are `strings` and `starred` is a `bool`, the following illustrates how `first` and `rest` would be populated for the provided regexes:

- `regex` `split(regex, first, starred, rest)`
  - `awxyz` `first` gets "a", `starred` gets `false`, `rest` gets "wxzy"
  - `[ae]*w*` `first` gets "[ae]", `starred` gets `true`, `rest` gets "w*"
  - `z` `first` gets "z", `starred` gets `false`, `rest` gets ""
  - `z*` `first` gets "z", `starred` gets `true`, `rest` gets ""

To be clear, `starred` is populated with `true` if any only if the leading portion placed in `first` is optional and repeatable, and `rest` is populated with everything beyond `first` and an, if present, the companion `*`.

Use the rest of this page to present your implementation, and don’t expand the character sets (you’ll do that part for part c). There’s no recursion here either, but it exercises your fluency with C++ strings and your understanding of pass by reference.

```cpp
static void split(const string& regex, string& first,
                   bool& starred, string& rest) {
    size_t pos = !isalpha(regex[0]) ? regex.find(']') + 1 : 1;
    first = regex.substr(0, pos);
    starred = pos < regex.size() && regex[pos] == '*';
    if (starred) pos++;
    rest = regex.substr(pos);
}
```

**Criteria for Problem 3b: 5 points**

- Correctly identifies where either the end of a character or a character set is: 2 points
- Properly substrings into the space referenced by `first`: 1 point
- Properly tests for the presence of an asterisk right after the leading component and updates `starred` as appropriate: 1 point
- Properly updates `rest` with everything after what is capable of matching a string’s first character: 1 point
c) [10 points] Using the `expand` and `split` functions you’ve already implemented, present your implementation of `matches`, which uses recursive backtracking to decide whether the supplied regex matches the supplied word. Because backtracking is required, you should only make as many recursive calls as needed in order to produce a true or false. This is your chance you convey your understanding of recursive backtracking, pass by reference, and string manipulation, all in the same problem.

```cpp
static bool matches(const string& regex, const string& word) {
    if (regex.empty()) return word.empty();
    bool starred;
    string first, rest;
    split(regex, first, starred, rest);
    string set = first.size() == 1 ? first : expand(first);
    if (word.empty() || set.find(word[0]) == string::npos) {
        return starred && matches(rest, word);
    }
    // consider first character being consumed by first
    if (matches(rest, word.substr(1))) return true;
    if (!starred) return false;
    return matches(rest, word) || matches(regex, word.substr(1));
}
```

Criteria for Problem 3c: 10 points

- Base cases all involve `regex.empty()`, and do the right thing (confirm that, for an empty regex, that "" matches and everything else fails): 2 points
- Advances on to split the nonempty regex into `first`, `starred`, and `rest`: 1 point
- Expands character sets into the range of characters that might match the leading character: 1 point
- For the empty string, returns true if and only if `first` can be skipped and `rest` can be (recursively) skipped: 1 point
- For a nonempty string whose leading doesn’t match anything that `first` expands to, return true iff first can be skipped and `rest` can match the entire string: 1 point

Continuing, the leading character is capable of matching `first`

- If `rest` matches everything after the leading character, then return true: 1 point
- If not, then match is only possible if `first` is starred. If it’s not starred, then return false: 1 point
- If it is starred, then `rest` could match entire word (i.e. `first` is skipped): 1 point
- If it is starred, then original regex could match `rest` of word (i.e. `first` matched leading character but might match next character as well): 1 point