CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

HW1

- Due: Fri. Sept. 4, 2015, 11:59pm (PST)
- Directory name in your github repository for this homework (case sensitive): `hw1`
  - Once you have cloned your `hw_usc-username` repo, create this `hw1` folder underneath it (i.e. `hw_usc-username/hw1`)
  - If your `hw_usc-username` repo has not been created yet, please do your work in a separate folder and you can copy over relevant files before submitting

A Few Notes on Repositories

1. Never clone one repo into another. If you have a folder `cs104` on your VM and you clone your personal repo `hw_usc-username` under it (i.e. `cs104/hw_usc-username`) then whenever you want to clone some other repo, you need to do it back up in the `cs104` folder or other location, NOT in the `hw_usc-username` folder.
2. Your repos may not be ready immediately but be sure to create your GitHub account and fill out the GitHub information form linked to at the end of Lab 01.

Skeleton Code

On many occasions we will want to distribute skeleton code, tests, and other pertinent files. To do this we have made a separate repository, `homework-resources`, under our class GitHub site. You should clone this repository to your laptop and do a `git pull` regularly to check for updates.

```
$ git clone git@github.com:usc-csci104-fall2015/homework-resources.git
```

Again, be sure you don't clone this repo into your `hw_usc-username` repo but at some higher up point like in a `cs104` folder on your laptop.

Problem 1 (Course Policies, 10%)

Carefully study the information on the course web site, then answer the following questions about course policies:

Place your answers to this question in a file name `hw1.txt`

Part (a):

Which of the following are acceptable behaviors in solving homeworks/projects?

1. Looking up information relevant to the course online.
2. Looking up or asking for sample solutions online.
3. Talking to my classmates about the problems.
4. Copying code from my classmates, and then editing it significantly.
5. Asking the course staff for help.
6. Sitting next to my classmate and coding together as a team or with significant conversation about approach.
7. Sharing my code with a classmate, if he/she just wants to read over it and learn from it

Part (b):

Which of the following are recommended ways of writing code?

1. gedit
2. emacs
3. Eclipse
4. sublime
5. Microsoft Visual Studio
6. notepad

Part(c):

What is the late submission policy?

1. Each assignment can be submitted up to two days late for 50% credit.
2. Each student has 4 late days of which only 1 can be used per HW
3. Students need to get an approval before submitting an assignment late.

Part(d):

After making a late submission by pushing your code to Gihub you should...

1. Do nothing! Sit back and enjoy.
2. Complete the online late submission form
3. Start the next HW sooner

Part (e):

Is there a grace period to submit assignments?

1. No
2. There is an hour grace period per assignment.

3. Yes, but only if there is a technical difficulty with submission.

Problem 2 (Git, 10%)

Carefully review and implement the steps discussed in Lab1. Then, answer the following questions:

Continue your answers to this question in the file name `hw1.txt`

Part (a):

Which of the following git user interfaces are accepted and supported in this course?

1. Git Bash (Windows)
2. GitHub Desktop Client
3. Terminal (Mac or Linux)
4. Eclipse eGit
5. Tower Git Client

Part (b):

Provide the appropriate git command to perform the following operations:

1. Stage an untracked file to be committed. The file is called 'hw1q2b.cpp'.
2. Display the details of the last three commits in the repository.

Part (c)

Let's say you staged three files to be committed. Then, you ran the following command:

```
git commit
```

What will git do?

Problem 3 (Review Material, and Programming Advice)

Carefully review recursion and dynamic memory management from your CSCI 103 notes and textbook. You may also find Chapters 2 and 5 from the textbook, Chapters 2 and 3 from the lecture notes, and the C++ Interlude 2, helpful.

You will lose points if you have memory leaks, so be sure to run `valgrind` once you think your code is working.

```
$ valgrind --tool=memcheck --leak-check=yes ./sum_pairs input.txt output.txt
```

Hint: In order to read parameters as command line arguments in C++, you need to use a slightly different syntax for your `main` function: `int main (int argc, char * argv[])`. Here, `argc` is the total number of arguments that the program was given, and `argv` is an array of strings, the parameters the program was passed. `argv[0]` is always the name of your program, and `argv[1]` is the first argument. The operating system will assign the values of `argc` and `argv`, and you can just access them inside your program.

Problem 4 (Recursive Definitions, 15%)

In class, we saw how to recursively define things like palindromes. Remember that a string being a palindrome can be characterized as follows:

- The empty string is a palindrome.
- Any single character is a palindrome.
- If *c* is a character, and *p* is a palindrome, then the string *cpc* is also a palindrome.

Here, you are to give us a similar definition for correctly parenthesized expressions. Our expressions will consist only of the following characters: opening and closing parentheses and brackets, and lowercase letters. No other characters need to be accounted for, not even spaces. You are to characterize using recursive formulas what a correctly parenthesized string is. In such a string, all opening parentheses must match the corresponding closing parentheses. Letters are allowed to occur in arbitrary places. Here are some examples to illustrate this:

- `abc`: correct (no parentheses at all)
- `(a[cc()]b((n)m)d)zz[]`: correct (everything matches)
- `[ab)`: incorrect (square doesn't match round)
- `[c(d)bba`: incorrect (square isn't matched)
- `)ab[](`: incorrect (first round parentheses doesn't match anything)

Your solution to this problem should be a text file (or MarkDown, or PDF; no Word, Apple Writer, or other formats please), in which you formally write down the recursive definition for correctly parenthesized expressions. Your solution should not contain any actual code for anything.

Problem 5 (Dynamic Memory, 20%)

Write a program to solve the following problem: you will be given a "grid" of floating point numbers (`double`), and are to look for the longest strictly increasing sequence along a row or column, forward or backward (but no diagonals). Notice that the grid need not be regular. The input will look as follows (explained below):

```
6
3 2 6 2 4 5
-1.0 3.14  42
0.2 2.7172
0.5 1.4142 -0.5 0 -0.5 0
0 0.577
10000 9999.99 -3 10
1 1 1 1 1
```

The first row tells you the number `n` of data rows you will have. The second row gives you the number of floating point numbers for each of the following `n` rows - here, since we have 6 rows of data, there are six numbers. Then, you get that many numbers in the next `n` rows.

We promise you that the total number of floating point numbers will always comfortably fit into main memory (say, no more than 10,000,000 total). But they could be in one long row, or in very many short rows, or anything in between. You will need to take care of this.

As output, you should just write the length of the longest increasing sequence in the input data. In the example above, that would be `4`; the sequence is (0.577, 1.4142, 2.7172, 3.14). (Pat yourself on the shoulder if you know what all those numbers are.) There are several sequences of length 3, such as (-1.0, 3.14, 42), or (-3, 9999.99, 10000). (We're mentioning this just to illustrate that you can go in a bunch of different directions.) Notice that (1,1,1,1,1) is not an increasing sequence of length 5, as the numbers need to **strictly** increase (not be equal).

Your program should be called `sequencesearch`, read its input from a file whose name is given at the command line, and write its output to a file whose name is also given at the command line. So for instance, if the file above were named `input.txt`, then running

```
./sequencesearch input.txt output.txt
```

should result in `output.txt` looking as follows:

```
4
```

Problem 6 (Parsing 25%)

Markdown is the language used inside `github` for documentation, and the language your course staff use to maintain your course web page. (It is then used translated to HTML.) We will also use Markdown as a format for "web pages" when you write your own version of Google this semester. While we will later give you code for parsing web pages, you should at least have an idea of what's going on inside that code, so here, you'll write a simple parser.

The input will be a plain text file, in which only the characters `[`, `]`, `(`, and `)` have special meaning. Words will be separated by anything that is not a letter, including numbers, white space, special characters, etc. Links to other pages are denoted by one of the following:

- `(link location)` is a link to `link location` that is just displayed as is. For instance, `(www.usc.edu)` would display as `www.usc.edu`.
- `[anchor text](link location)` is a link to `link location` that is displayed as `anchor text`. So `[USC](www.usc.edu)` would display as `USC`, and when you click on it, you are taken to `www.usc.edu`.
- `[anchor text]` is not a valid link. In this case, the square brackets are just punctuation characters that should be ignored.

Your task is to write a program named parsemd, which reads the name of a Markdown text file at the command line, and outputs into a file, one per line, each **word** in the file in the order in which they appear. You should not output any special characters, numbers, white space, etc. For each link that you encounter, you should output `LINK (destination, anchor text)`, where `destination` is where the link points, and `anchor text` is the anchor text that is displayed. (If none was specified, that's the same as the destination.)

Here are a few answers to special cases you may have about this:

- To simplify your life, we promise that each link `[` or `(` is preceded by a whitespace, tab, or newline character.
- The text inside the `[]` or `()` could be anything, except it will not contain any `[`, `]`, `(`, or `)`. It may contain spaces or numbers or punctuation marks, and the "link location" may not be a well-formed web address. You should just output them as they are, not breaking them into words.

- The text or link location (or both) may also be empty, in which case you should output the empty string.
- There may be text immediately after the closing `)`; you should just treat it as a new word.

For example, suppose that the input was the following file `input.md`:

```
Writing my own (www.google.com)in   my 3rd
semester at [USC](www.usc.edu).
[Parsing] #!@@!  text   is part of [Assignment 1].
```

You would run

```
$ ./parsemd input.txt output.txt
```

The file output.txt should afterwards be

```
Writing
my
own
LINK (www.google.com, www.google.com)
in
my
rd
semester
at
LINK (www.usc.edu, USC)
Parsing
text
is
part
of
Assignment
```

Problem 7 (Using Recursion to Generate Combinations, 20%)

A palindrome is a string that is the same when read forward and backward (e.g. `racecar`, `mom`, `otto`). It can be recursively defined as `P = {empty string}` or `P = xPx` where x is some character.

Your job is to write a program that will read in a string of characters and an integer size and generate **all possible** palindromes of the **given size or less** using the characters provided as your options. Each palindrome should be output to a file.

All the input information will come from the command line:

```
$ ./palindrome out.txt abc 4
```

The contents of `out.txt` should have the following palindromes:

```
aa
aaaa
baab
caac
```

```
bb
abba
bbbb
cbbc
cc
acca
bccb
cccc
a
aaa
bab
cac
b
aba
bbb
cbc
c
aca
bcb
ccc
```

Note: In the above file the first line is the empty palindrome (length 0) which is a valid palindrome.

To do this you must write a function that directly or indirectly (via a helper function) uses recursion to generate these palindromes. The function signature should take an ostream (by reference), the array of character options to use, and the size of the desired palindromes. Thus, your function's signature should be:

```
// client uses this interface
void makePalindromes(ostream& ofile, char* options, int size);
```

You may feel free to define a helper function such as:

```
// recursive helper function
void makePalindromeHelper(ostream& ofile, char* options, int len, int size, string
pal);
```

Again, the helper function is optional or you can change its interface if you desire.

A few things to note/remember.

1. The empty string is a valid palindrome
2. Palindromes can be odd or even in length
3. The characters in `option` will be unique (no duplicates)
4. Recursion usually replaces 1 loop (in this case it will be easiest to make a recursive call to add more letter(s) to the palindrome.
5. Recursive functions can contain loops (in this case it will be easiest to use a normal loop to iterate over each character in the set of options.
6. Strings support the "+" operator to append words (e.g. `"cat" + " " + "dog"` yields `"cat dog"`)

7. Your palindromes can be output in any order

We have provided a skeleton for you in the `homework_resources/hw1` folder/repo. Copy the skeleton file `palindrome.cpp` to your `hw_usc-username/hw1` folder and write the code.