**CS120L – C Programming Language Labs**
Fall 2014
**Week 9: Functions, Arrays & Simple Problem Solving**

In this lab, you are not allowed to use any libraries other than `stdio.h`. You are also not allowed to use `goto` statement. 20 marks will be deducted if you violate these rules.

## <u>Task 1 (30 marks)</u>: Frog in a Well

At the bottom of a well, there is a frog that starts to travel upwards to the ground. During his journey, he ascends x feet every day, but falls back y feet every night. The distance he ascends and falls back each day and night is shown in the following table.

| | Day | Night |
|---|---|---|
| #1 | Ascend 10 feet | Fall back 2 feet |
| #2 | Ascend 8 feet | Fall back 2.5 feet |
| #3 | Ascend 7 feet | Fall back 2.7 feet |
| #4 | Ascend 5 feet | Fall back 3 feet |

Your task is to compute the number of days the frog needs to reach the ground. Note that once the frog has reached the ground, he will not fall back into the well. You can assume the well is shallow enough that the frog will get out in no more than 4 days.

For example, if the well is 4-feet deep, the frog will need to travel for 1 day. If the well is 15-feet deep, the frog will need to travel for 2 days (he can go up by 10 feet after the 1st day, and falls down by 2 feet after the 1st night; hence he needs to travel 7 feet more on the 2nd day to reach the ground).

In this program, you need to write a function `daysTravel` whose parameter is the depth of the well. It should return the number of days the frog needs to travel to the ground.

The function prototype should look as follows:

```
int daysTravel(double depth);
```

**Input:** The non-negative depth of the well. It is guaranteed to be small enough for the frog to reach the ground within 4 days.

**Output:** the number of days the frog needs to travel.

Note that printf and scanf should only be used in the main function.

**Sample Test Cases**
The ↵ denotes an invisible new line character at the end of the output. For all the labs, you should not print out any extra characters or extra blanks before and after the output.

Test case 1:
```
4.2
1↵
```

Test case 2:
```
15.3
2↵
```

You will get 5 extra marks if you are able to come out with a solution with the least number of lines of codes (not counting the preprocessing directives, variable declaration & initialization).

## Task 2 (70 marks): Sum of Prime Numbers

There is a conjecture in mathematics that every **even** number larger than 2 can be written as the sum of two prime numbers. It has never been proven true mathematically, but a counter-example has never been found either.

4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
10 = 3 + 7
12 = 5 + 7
.
.
122 = 13 + 109
124 = 11 + 113
126 = 13 + 113
128 = 19 + 109
.
.
194 = 3 + 191
196 = 3 + 193

198 = 5 + 193
200 = 3 + 197 …

In this task, you need to write two functions.

First, you need to write a function named `conjecture` that will takes a range (inclusive) of numbers and show that each even number within the range is the sum of two prime numbers. The prototype looks like this:

```
void conjecture(int low, int high);
```

The output is as above. The format string for the function ouput is: "`%3i = %3i + %3i\n`"

Then you need to write a helper function to determine whether an integer is a prime number or not. The prototype of the helper function could look like this:

```
int is_prime(int number);
```

It should return 1 if the number of prime and 0 otherwise.

All of the prime numbers between 2 and 100:
```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97
```

**Hint for prime checking:** You will need to use the modulus operator `%`, since a prime number is only divisible by itself and one. If you find another value that divides your number, then it can't be a prime number. By definition, 1 is not prime and 2 is prime, so you can return false or true immediately for those. Also, another quick check to do is to see if the number is even. Even numbers (except 2) are not prime.

One possible approach can be: For each number, use a loop to test every odd value from 3 to the number divided by 2. If any of those divides the number evenly, it's not prime. If none of them divides it evenly, then the number is prime.
Here is the list of the first 1000 primes: output-primes1000.txt.

Now in this task, we only want to find the first pair of prime numbers that add up to the input number. Note that most even numbers have several pairs of primes that add up. **We want the first pair, which means the first number in the pair is always lower than the second one.**

The output shown below is from the call: `conjecture(2, 20)`

Correct:
```
4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
10 = 3 + 7
12 = 5 + 7
14 = 3 + 11
16 = 3 + 13
18 = 5 + 13
20 = 3 + 17
```

Incorrect (some numbers are represented by more than one pairs, as highlighted):

```
4 = 2 + 2
6 = 3 + 3
8 = 3 + 5
10 = 3 + 7
10 = 5 + 5
12 = 5 + 7
14 = 3 + 11
14 = 7 + 7
16 = 3 + 13
16 = 5 + 11
18 = 5 + 13
18 = 7 + 11
20 = 3 + 17
20 = 7 + 13↲
```

Here is an output for the call `conjecture(2, 2000)`: output-conjecture2000.txt

**Implementation**

- `conjecture` and `is_prime` should be implemented as two separate functions in a given file named `q2-template.c`. You do not have to perform any validity check on the input numbers, since they will be read in using the given `main.c`.
- The `main.c` file should not be updated. Update the code in `q2-template.c` file only. It is the only file to be submitted and it should **not** contain any main function. You need to rename the file to `q2.c` for submission.

## Command for compiling

```
/usr/bin/gcc -Wall -Wextra -ansi -pedantic q1.c -o q1.exe
/usr/bin/gcc -Wall -Wextra -ansi -pedantic q2.c main.c -o q2.exe
```

## Grading scheme

- 100 marks in total (30 marks for task 1 and 70 marks for task 2).
- Relatively for each task:
  - 20% for successful build.
  - 20% for successful execution and correct output (4% * 5 hidden test cases).
    Note that your program should only print out the required output, ending immediately with an end-of-line character `\n`, and not any redundant character.
  - 40% for correct implementation.
  - 20% for good programming style. For the complete guide, please see the Assignment Guideline on CS120 page. In general, you should have clear variable names, reasonable comments to explain your code, and consistent indentation. Please use **2-blank spaces** instead of tab spaces.

## Output formatting

You should use the given sample input and output files to check your output format before submission. After compiling your program into `a.exe`, you can generate your output using:

```
./a.exe < q1.in > result.out
```

Then compare it with the sample output by typing:

```
diff result.out q2.out
```

You need to make sure that diff does not print out anything. Otherwise, your output is considered wrong and you will lose all the 10 marks for output correctness.

## Submission

Please name your source code for task 1 as **q1.c**, task 2 as **q2.c**, put them in a folder named **cs120<session>_<your Digipen login id>_<labnumber>**, in which <**labnumber>** is **9** this week and **<session>** is either **a** or **b**, and zip them in **cs120<session>_<your Digipen login id>_<labnumber>.zip** for submission.

Wrong submission file/folder name will cause 10 marks deducted.

Note that from this lab onwards, the file and folder names must be **lowercase**.

The deadline of submission is 31st October 23:59 and late submission will receive zero mark.