One of the exercises in this lab has to be done in a group of 4. In particular, you will be asked to do exercise 2 individually, and then provide feedback on the solutions other members of the group come up with.

Please include the names and student IDs of the students who reviewed your solution:

<table>
<thead>
<tr>
<th>Name</th>
<th>Student ID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Exercise 1  <<10 + 5 points>>

<<10 points>>
You will write a program that will allow the user to play a simple game: Guess a number. In this game, the program starts by picking a number between 1 and 100 and showing the following message:

I'm thinking of a number between 1 and 100. Can you guess it?

The program will ask for an integer number. If...

- ... the number is larger than the secret number:
  I'm thinking of a smaller number. Please try again.
  (the program must now ask the user for another number)

- ... the number is smaller than the secret number:
  I'm thinking of a larger number. Please try again.
  (the program must now ask the user for another number)

- ... the user guessed the secret number:
  Correct! You guessed the secret number!

<<5 points>>
Add the following features to the above program:

- The program must accept a command-line parameter to specify the upper bound for the range of possible numbers. For example, if the user runs the program like this...

  ./guess -n 150

  ... a number will be picked between 1 and 150.

- It is always possible to guess the number in at most \( \lceil \log_2(n) \rceil \) guesses, where \( n \) is the number of possible numbers (+3 bonus points if you can tell me why this is so). So, we can control the game's level of difficulty by limiting the number of guesses to some fraction of \( \lceil \log_2(n) \rceil \). You must add a command-line parameter to specify the difficulty of the game: easy ( \( \lceil 0.8 \cdot \log_2(n) \rceil \) guesses), medium ( \( \lceil 0.6 \cdot \log_2(n) \rceil \) guesses), hard ( \( \lceil 0.4 \cdot \log_2(n) \rceil \) guesses), or unlimited guesses:

  ./guess -d easy
  ./guess -d medium
  ./guess -d hard
  ./guess -d unlimited
Exercise 2 <<15 points>>

In this exercise, you will implement an exponentiation algorithm (i.e., given \( a \) and \( x \), compute \( a^x \)). However, the following will not be an acceptable solution:

```c
int pow(int a, int x)
{
    int p=1;
    for(int i=0; i<x; i++)
        p = p * a;
    return p;
}
```

The above function has a linear running time (with regards to \( x \); notice how the for loop will perform \( x \) iterations). This is not particularly efficient considering that there are exponentiation algorithms that will compute \( a^x \) in sub-linear time (i.e., requiring less than \( x \) operations). Assuming that \( x \) is a positive integer, you must implement such an algorithm.

However, you do not have to design this algorithm yourself. Choose an existing algorithm for fast exponentiation (on the web, from a previous math course, etc.) and implement it. Two important rules for this exercise:

- Make sure you cite your sources. Failure to do so will result in getting no credit for this exercise.
- Do not discuss your algorithm with your classmates before the peer review.
- Do not post questions about the exponentiation algorithm, or code from your implementation of the algorithm, on the mailing list.

Your implementation will be a single C program accepting two command-line parameters: \( a \) and \( x \).

```
$ pow 2 4
16

$ pow 42 7
230539333248
```
Extra Credit <<10 points>>

Implement a prefix notation evaluator. An expression written in prefix notation (also called polish notation), places the operator before the operands. For example:

```
+ 2 3
```

This expression evaluates to 5 (and is equivalent to the infix expression `2 + 3`). An attractive feature of prefix notation is the fact that it does not require parentheses to determine the order in which the operators must be evaluated. For example:

```
+ * 1 5 / 9 3
```

Evaluates to:

```
+ 5 3
```

And then to 8. We could rewrite the original expression with parentheses, although they would be redundant:

```
(+ (* 1 5) (/ 9 3))
```

Your program must take the prefix expression as a command-line parameter, and output the value it evaluates to. For example:

```
$ prefixeval + 5 7
12

$ prefixeval + \* 2 3 / 10 5  # Notice how we have to escape the '*' character
8
```

You can make the following assumptions:

- Only the following binary operators can be used: +, -, *, / (addition, subtraction, multiplication, division). For extra bragging rights, implement the unary negation operator.
- Operands are always positive integers.
- There is always a single space character separating the operators and operands.
- The expressions are always syntactically correct.
Documentation <<10 points>>

Besides making sure that your code has adequate comments, you must provide feedback on the solution to Exercise 2 from the other members of your group. In particular, for each member of your group, answer the following questions (your answers to these questions should be concise and to the point):

➢ What algorithm did your peer use? Is it the same algorithm you used?
  ➢ If so, how does your peer's implementation differ from yours?
  ➢ If not, is your algorithm better or worse? (cite specific reasons; e.g., your algorithm might be more efficient, or it might deal with more cases, etc.)
  ➢ Is the code easy to read and well-documented?

**Do not** change your algorithm based on the feedback you received from the group (if you chose an algorithm, stick to it). If you made other changes based on your peers' comments, describe them briefly (1-2 paragraphs).

You must write up this documentation in a text file called DOCUMENTATION.txt, which must be included in your lab submission.

**Using make**

When submitting your code, you must also include a Makefile to compile your code. At the very least, you should include a separate Makefile for each exercise, so that the code can be built like this:

```
$ make -f Makefile.ex1
$ make -f Makefile.ex2
$ make -f Makefile.extracredit
```

You can also write your Makefile using different styles (e.g. a single Makefile with separate targets for each exercise, and a default target that will build all exercises). However, in that case you must include a README file with your submission with instructions on how to compile your code.

*Failure to include a Makefile will result in a 30% point deduction.*