Assignment 2

Instructions: For each problem except A, write a single source, and a script that functions as directed. The names of each solution should be X.c and X.sh where X is the letter associated with the problem. Tar and zip your solutions and submit them via [http://www.cs.uoregon.edu/Classes/12F/cis330/turnin.php](http://www.cs.uoregon.edu/Classes/12F/cis330/turnin.php) before midnight GMT on Saturday, Oct 13.

A) The following BNF grammar describes a prefix arithmetic problem on natural numbers.

E ::= unsignedInt space | operator E E

The production unsignedInt is given by the regular expression [0-9]+ and operator is given by the regular expression [-+*/^]. Note this this definition requires a trailing space character.

For example, the infix expression “(2 + 3) * 4”, would be “/+2 3 4 ” in prefix notation. Other legal prefix expressions would be “^2 *572 +34 75 ”, “***23 84 175 2 ” and “+*2 3 * 2 4 ”, representing $2^{572(34+75)}$, $23(84)(175)(2)$, and $2(3)+2(4)$, respectively.

Write a C program that reads a line from standard in and prints one of the following to standard out:

- Parameter is a well-formed prefix expression
- Parameter is missing
- Parameter contained illegal character(s)
- Parse encountered unexpected number
- Parse ended with insufficient numeric expressions
- Other parse problem

Hint: It may be easier to use getchar().

Your program should continue to read lines until it encounters a blank line. Your program should stop when it reads a blank line.

B) Write a C program that reads numbers from standard in until it reads a zero. For each number, your program should output one of the following messages on a line in standard out:

- The number is probably prime
- The number is divisible by two
- The number is divisible by three
- The number is divisible by five
- The number is not an Euler prime

Recall that $n$ is an Euler prime (or pseudoprime) if $p^{(n-1)/2} \equiv \pm 1 \pmod{n}$ for some prime $p$. For each parameter $n$, your program should test $p = 2, 3$ and $5$. $n$ is an Euler prime, if it is also prime, otherwise, it is an Euler pseudoprime. Most Euler numbers are prime.

Your program should not make use of the imprecise C pow() function. Rather, recall that we can use the following identity to quickly calculate powers:
\[a^b = \begin{cases} (a^2)^2 & \text{if } b \text{ even} \\ a(a^{b-1}) & \text{if } b \text{ odd} \end{cases}\]

C) The following structure represents a non-integral number:

```c
struct rational {
    bool is_exact;
    union {
        float flt;
        struct {
            int numerator;
            int denominator;
        };
    };
};
```

Using this structure, write a multiplication function that takes two rationals and computes their product as a rational. If either of the rationals uses the imprecise floating point format, then the answer must also be in floating point format; however, if both of the rationals use the precise numerator/denominator representation, the result should use the more precise representation. The product should always have the numerator and denominator in reduced form (all common factors cancelled.) Recall the Euclidean Algorithm for finding the greatest common divisor of two non-negative numbers uses the identity

\[\gcd(x, y) = \begin{cases} x & y = 0 \\ \gcd(y, x) & x < y \\ \gcd(y, x \mod y) & x \geq y \end{cases}\]

Write a program that reads a series of space-delimited numbers on a line in standard in. Each number has the format %d, %d/%d, or %f. For each line, print the product to standard out. The output should use a decimal point if the product uses the imprecise floating point format, numerator/denominator or simply numerator if the denominator is 1. Your program should stop when it encounters a blank line.

<table>
<thead>
<tr>
<th>Example Input</th>
<th>Example output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2 3.5 2</td>
<td>3.5</td>
</tr>
<tr>
<td>1/2 7/2 2</td>
<td>7/2</td>
</tr>
<tr>
<td>6.75 2.1</td>
<td>14.175</td>
</tr>
<tr>
<td>4/5 3/15 5/7</td>
<td>2/35</td>
</tr>
<tr>
<td>4/5 3/15 5/7</td>
<td>0</td>
</tr>
<tr>
<td>4/5 3/15 5/7</td>
<td>0.0</td>
</tr>
</tbody>
</table>

D) Write a BASH script that reads lines from standard in until it sees a blank line. On each line, it should find two space-separated strings of A’s, C’s, T’s and G’s. You should call your C program with these two parameters. If the result code of your C program is non-sero, then your script should print “Internal Error”. If there are more than two strings on a line, then your
program should print “Input Error” and not invoke your C program. In any event, you program should continue looping until it sees a blank line.

Create a C program for your script to invoke. The program should take two strings as arguments and output a single number to standard out which represents the minimum edit distance between the strings.

The edit distance between two strings A and B is the minimum number of character insert, character delete and character substitute operations that I have to perform on A to make it B. For example, if A begins with “T” and B begins with “G”, then it may be better to discard the “T” from A, add a “G” in the front of A, or change the “T” in A to a “G”. I have to consider doing all of these things to figure out which is part of the minimal solution. If A starts with “G” and B starts with “G”, then I can just consider the two G’s to be the same, and move on. Obviously, if A is the empty string, then the edit distance is |B|, and if B is the empty string, then the edit distance is |A|.

To solve this problem, use the following recursive function declaration

```c
int edit_dist(char* s1, char* s2)
```

Think about how to test your base cases. Think about each of the four choices, and how they would generate a recursive call, and affect the result.

E) Your recursive solution in problem D would take a long time even for relatively small problems. This is because it will compute the same thing many times. For example, to see the edit distance between ATTA and GATA, we would consider the following sequences:

- ATTA \rightarrow GTTA \rightarrow GATA \rightarrow GATA
- ATTA \rightarrow TTA \rightarrow GTTA \rightarrow GATTA \rightarrow GATA \rightarrow GATA
- ATTA \rightarrow GTTA \rightarrow GATA \rightarrow GATA \rightarrow GATA

Etc.

Notice that it would compare the final TA in ATTA and the final TA in GATA many times, but the shortest edit distance from TA to TA is 0. So, it makes sense to figure this out once and then use that figure to compute all of the strings that end with those two strings. Doing this for all substrings will take the running time from exponential to quadratic.

So, to your solution to problem D, add a two-dimensional array of ints (p 224 of your textbook). The first subscript represents the start position in A and the second subscript represents the start position in B. Each entry should contain either -1 if you haven’t figured it out yet, or the edit distance for the strings starting at those positions and going until the end of both strings. For example, if |A|=17 and |B|=29 then edit_dist_array[16][28] is zero or one, depending on whether the strings end in the same letter, or not.. edit_dist_array[0][0] is some number between 0 and 28, which is the answer to the problem. Rework your solution so that it either computes the answer using the function you wrote in problem D or finds it in the table you created here.

You may assume that no input string will be longer than 100 characters. Hint: start at the end of your strings and build toward the beginning.

Here is the answer to this problem in Python:
from sys import argv

def memo(f):
    d = {}
    def g(A, B):
        if (A, B) not in d:
            d[(A, B)] = f(A, B)
            return d[(A, B)]
    return g

def edit_dist(A, B):
    if len(A) == 0:
        return len(B)
    if len(B) == 0:
        return len(A)
    if A[0] == B[0]:
        return edit_dist(A[1:], B[1:])
    else:
        return min([1 + edit_dist(A[1:], B[1:]), 1 + edit_dist(A[1:], B), 1 + edit_dist(A, B[1:])])

edit_dist = memo(edit_dist)
print(edit_dist(sys.argv[1], sys.argv[2]))