University of California San Diego

ECE15: Lab #3

This lab relates specifically to the material covered in Lecture Units 6 and 7 in class, although it assumes knowledge of the previous Lecture Units as well. Here are several instruction specific to this lab:

- In Problems 1 and 3, you will be reading input from a file; use the **`fscanf()`** function to do so. When reading input from a file, you ***do not need*** to verify its validity. You ***can assume*** that it has the expected type *and* the expected value(s). In Problem 2, you will be reading input from `stdin`; input verification requirements for Problem 2 are described therein.
- In each of the three problems in this lab, you will be required to declare and define certain functions. Doing so is essential: code that compiles and runs correctly, but does not implement all the required functions will *not* receive full credit. You are also welcome to implement additional functions. While this is not required, it can make your code easier to write.
- Throughout this lab, you ***should not use*** any functions from the C standard library, ***except*** for those functions that are declared in the file `<stdio.h>`, such as **`printf`** (or **`fprintf`**), **`scanf`** (or **`fscanf`**), **`fopen`**, **`fclose`**, etc. In Problem 2, you can also use the functions **`rand`** and **`srand`** declared in `<stdlib.h>` and the function **`time`** declared in `<time.h>` Using other library functions in Problems 2 and 3 *may* result in partial credit. However, if you use *any* function from `<math.h>` in Problem 1, you will receive 0 points for this problem.
- For each of the three programs below, you should use the file `templateF.c` provided on the course website at `http://ece15.ucsd.edu/Main/Homeworks.html`. Note that this file differs from the file `template.c` used in all the previous labs.
- Try to make your programs easy to follow and understand: choose meaningful names for identifiers, use proper indentation, and provide comments wherever they are useful. While this is not required, it could help you receive partial credit: programs that are not correct and are also difficult to understand will receive zero credit, even if they are partially correct.

This lab will be graded out of 100 points. Problems 1 and 3 are worth 30 points each, while Problem 2 is worth 40 points. You should submit the lab by following the instructions posted on the course website at `http://ece15.ucsd.edu/Labs/TurnIn3.html`.

Problem 1.

Write a C program, called `cos_approx.c`, that computes the approximate value of $\cos(x)$ according to its Taylor series expansion:

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \cdots \tag{1}$$

This series produces the *exact* value of $\cos(x)$ for any real number x, but contains an infinite number of terms. Obviously, a computer program can compute only a finite number of terms. Thus, you will have to truncate the infinite series in (1). Your program should be able to do so in two different ways.

Fixed number of terms: Implement the function **`cos_N(double x, int N)`** that accepts as parameters a real number x and an integer N (you can assume that N will be always positive). The function **`cos_N`** should return the sum of the first N terms in (1), as a **`double`**.

Fixed precision: Implement the function **`cos_delta(double x, double delta)`** that accepts as parameters a real number x and another real number δ (you can assume that δ will be always positive). The function **`cos_delta`** should return, as a **`double`**, the sum of the first N terms in (1), where N is the smallest positive integer such that

$$\left| \sum_{n=0}^{N-1} \frac{(-1)^n x^{2n}}{(2n)!} - \sum_{n=0}^{N-2} \frac{(-1)^n x^{2n}}{(2n)!} \right| < \delta \tag{2}$$

Notice that the first sum in (2) contains N terms, while the second sum contains $N-1$ terms. It is possible for the second sum in (2) to be empty — this happens when $N = 1$. You should assume that an empty sum evaluates to zero.

Your program should read its input from a file called `cos_input.dat`. The first line in this file is a positive integer m (you can assume that $m \leqslant 64$). The first line is followed by m other lines; each such line constitutes a *test case*. Every test-case line contains three numbers separated by whitespace. The first number is either 1 or 2, indicating whether you should use **`cos_N`** or **`cos_delta`**. The second number is the value of x for which you should compute $\cos(x)$. The third number y is *either* the required precision δ (if the first number is 2) or the required number of terms N (if the first number is 1). In the former case, y will be a floating-point number while in the latter case, it will be an integer. In both cases, you can assume that y is positive. Here is a sample `cos_input.dat` file:

```
5
1 -1.00 6
2 1 0.00001
1 1.5     2
2 2 0.09
2 2 1.1
```

The program should write its output to the file `cos_output.dat`. The output file should consist of m lines, one per test case. For each test case, the program should print the test-case number followed by $\cos(x.xxx) = y.yyyyyyyyyyyy$. In the above, the argument of $\cos(\cdot)$ should be printed with 3 digits of precision, while its (approximate) value should be printed with 12 digits of precision. For example, here is the file `cos_output.dat` that results upon processing the file `cos_input.dat` above:

```
Case 1:  cos(-1.000) = 0.540302303792
Case 2:  cos(1.000) = 0.540302303792
Case 3:  cos(1.500) = -0.125000000000
Case 4:  cos(2.000) = -0.422222222222
Case 5:  cos(2.000) = 1.000000000000
```

Notes:

- As part of the solution, you are required to declare, define, and call the following two functions. The function **`factorial(int n)`** that accepts as a parameter an integer $n \geqslant 0$ and then returns $n!$ as an **`int`**. Recall that $n! = 1{\cdot}2{\cdot}3{\cdots}(n-1){\cdot}n$ for $n \geqslant 1$, while $0! = 1$. The function **`power(double x, int n)`** that accepts as parameter a real number x and an integer $n \geqslant 0$ and returns x^n as a **`double`**. Recall that $x^0 = 1$ for any real number x.

- ▶ You *do not need to worry* about numerical overflows. You can assume that the input is such that they do not occur. For example, in the function **`factorial(int n)`**, you can assume that the parameter n is small enough so that $n!$ fits in a variable of type **`int`**. For most machines, this means that $n \leqslant 12$. $\big[$**Optional, not for credit:** How would you modify this function so that it can compute $n!$ for much larger values of n?$\big]$

- ▶ You are ***not allowed*** to use ***any*** functions declared in `<math.h>` in the file `cos_approx.c` that you submit. However, it might be advisable to use standard-library functions such as **`pow(x,y)`** and **`cos(x)`** during the *development and debugging* of the program, in order to compare their output with the output of the functions that you implement. If you choose to do so, do not forget to delete the relevant parts of the code in the final version of `cos_approx.c` that you submit.

Problem 2.

The famous ***Match & Hit*** game is played by a computer and a human player as follows. First, the computer selects a random 4-digit number $N = a\cdot 10^3 + b\cdot 10^2 + c\cdot 10 + d$, where a, b, c, d are *distinct nonzero digits* — that is, a, b, c, d are distinct elements of the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Let us call numbers like this ***valid***. The human player then tries to deduce N from a sequence of queries to the computer. Each query consists of a valid 4-digit number $M = x\cdot 10^3 + y\cdot 10^2 + z\cdot 10 + w$. The computer responds to each query with the *number of matches* and the *number of hits*. A ***match*** is a digit of N that appears in M at the same position (thus each of $x = a$, $y = b$, $z = c$, or $w = d$ counts as one match). A ***hit*** is a digit of N that appears in M, but <u>not</u> at the same position. For example, if $N = 5167$, then the queries 2934, 1687, 7165, 5167 will result in the following numbers of matches and hits:

$$\begin{array}{c l}
\underline{5167} & \\
2934 & \longrightarrow \text{ no matches and no hits} \\
\underset{\circ\circ\ \bullet}{1687} & \longrightarrow \text{ one match and two hits} \\
\underset{\circ\bullet\bullet\circ}{7165} & \longrightarrow \text{ two matches and two hits} \\
\underset{\bullet\bullet\bullet\bullet}{5167} & \longrightarrow \text{ four matches and no hits}
\end{array}$$

where matches are denoted by $\bullet$ and hits are denoted by $\circ$. The play continues until the human player either wins or loses the game, as follows:

Human player wins if he submits a query with 4 matches (that is, $M = N$).

Human player loses if he submits 12 queries, none of them with 4 matches.

Write a C program, called `match_and_hit.c`, that implements (the computer part of) the Match & Hit game. In this program, you are required to declare, define, and call the following functions:

- ▶ The function **`isvalid(int n)`** that accepts as a parameter an arbitrary integer. The function should return **`1`** if the integer is valid, and **`0`** otherwise. Recall that an integer is valid if it is positive, consists of exactly 4 decimal digits, and all these digits are nonzero and distinct.

- ▶ The function **`choose_N(void)`** that returns, as an **`int`**, a uniformly random choice of a valid integer N. The function should call both the **`rand`** library function and the function **`isvalid`**. It should keep calling **`rand`** until the number generated thereby is valid. Recall that before the function **`rand`** is invoked, the random seed should be initialized using **`srand(time(0))`**.

- The function **`matches(int N, int M)`** that accepts as parameters two integers N and M and returns, as an **`int`**, the number of matches. You can assume than both N and M are valid.
- The function **`hits(int N, int M)`** that accepts as parameters two integers N and M, then returns, as an **`int`**, the number of hits. You can assume than both N and M are valid.

Here is a sample run of this program, assuming that the executable file is called `match_and_hit`. This run illustrates a situation where the human player wins the game. User input is underlined.

```
/home/userXYZ/ECE15/Lab3> match_and_hit
 ***Welcome to the MATCH and HIT game***
The computer has selected a 4-digit number.
Try to deduce it in 12 rounds of queries.

Round #1
Please enter your query (4 digits): 5341
-> 2 matches and 1 hit

Round #2
Please enter your query (4 digits): 1235
-> 1 match and 2 hits

Round #3
Please enter your query (4 digits): 2345
-> 4 matches and 0 hits

**********************************
CONGRATULATIONS! You won the game!
**********************************
```

Notice that a singular form (**`match`**, **`hit`**) is used with the number **`1`**. You can easily achieve this functionality with the **`?:`** conditional expression, as shown in class. The next sample run illustrates a situation where the human player loses the game. Again, user input is underlined.

```
/home/userXYZ/ECE15/Lab3> match_and_hit
 ***Welcome to the MATCH and HIT game***
The computer has selected a 4-digit number.
Try to deduce it in 12 rounds of queries.

Round #1
Please enter your query (4 digits): 1234
-> 0 matches and 3 hits
 ⋮    ⋮

Round #12
Please enter your query (4 digits): 2435
-> 2 matches and 2 hits

*********************************
Sorry, out of queries. Game over!
*********************************
```

Notes:

- If the user enters a non-integer, print the message "**Invalid query. Please try again!**" Notice that if the user enters an integer *followed* by noninteger characters, these characters should be simply ignored (check the value returned by the **scanf** function). If the user enters an integer that is not valid, print the message "**Invalid number. Please try again!**" In both of these cases, you should then prompt the user to enter another query, and keep doing so until the user provides a valid input. The following sample run illustrates all this.

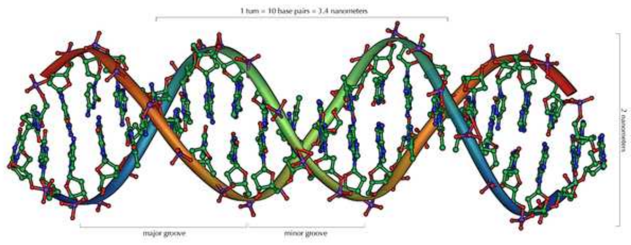
```
/home/userXYZ/ECE15/Lab3> match_and_hit
 ***Welcome to the MATCH and HIT game***
The computer has selected a 4-digit number.
Try to deduce it in 12 rounds of queries.

Round #1
Please enter your query (4 digits): ##
Invalid query. Please try again!
Please enter your query (4 digits): 0##
Invalid number. Please try again!
Please enter your query (4 digits): -5341
Invalid number. Please try again!
Please enter your query (4 digits): 123456789
Invalid number. Please try again!
Please enter your query (4 digits): 5340
Invalid number. Please try again!
Please enter your query (4 digits): 5344
Invalid number. Please try again!
Please enter your query (4 digits): 5341#OK?
-> 2 matches and 1 hit

Round #2
Please enter your query (4 digits): 1235
-> 1 match and 2 hits

Round #3
Please enter your query (4 digits): 2345
-> 4 matches and 0 hits

**********************************
CONGRATULATIONS! You won the game!
**********************************
```

- The maximum number of queries (which we have, so far, assumed to be 12) should be introduced as a symbolic constant **MAX_QUERIES** in the beginning of the program, using the **#define** compiler directive. The number of digits in a valid integer (which we have, so far, assumed to be 4) should be also introduced as a symbolic constant called **N_DIGITS**. You can assume that **N_DIGITS** is one of $1,2,3,4$ and **MAX_QUERIES** is an integer in the range $1,2,\ldots,24$. The program should keep working correctly for all such values of **N_DIGITS** and **MAX_QUERIES**.

- You can always win the game with at most 12 queries, if you query cleverly. Try it, it's fun! Can you guarantee a win with less than 12 queries? If so, what is the minimum number of queries you need? You do not need to answer these questions, but you are welcome to consider them.

Problem 3.

The deoxyribonucleic acid (DNA) is a molecule that contains the genetic instructions required for the development and functioning of all known living organisms. The basic double-helix structure of the DNA was co-discovered by Prof. Francis Crick, a long-time faculty member at UCSD.



The DNA molecule consists of a long sequence of four nucleotide bases: adenine (A), cytosine (C), guanine (G) and thymine (T). Since this molecule contains all the genetic information of a living organism, geneticists are interested in understanding the roles of the variuos DNA sequence patterns that are continuously being discovered worldwide. One of the most common methods to identify the role of a DNA sequence is to compare it with other DNA sequences, whose functionality is already known. The more similar such DNA sequences are, the more likely it is that they will function similarly.

Your task is to write a C program, called `dna.c`, that reads three DNA sequences from a file called `dna_input.dat` and prints the results of a comparison between each pair of sequences to the file `dna_output.dat`. The input file `dna_input.dat` consists of three lines. Each line is a single sequence of characters from the set {**A**, **C**, **G**, **T**}, that appear without spaces in some order, terminated by the end of line character **\n**. You *can assume* that the three lines contain the same number of characters, and that this number is at most 241 (including the character **\n**). Here is a sample input file:

```
ACGTTTTAAGGGCTGAGCTAGTCAGTTCATCGCGCGTATATCCTCGATCGATCATTCT
CTCTAGACGTTTTAAGGGCTGAGCTAGTCAGTTC
ACGTTTTAAGGGCTTAGAGCTTATGCTAATCGCGCGTATATCCTCGATCGATCATTCT
CTCTAGACGTTTTAAGGGCTAAGGCGCGTAATTA
TCGTTTGAAGGGCTTAGTTAGTTAGTTCATCGGCGGCGTATATCCTCGATCGATCATTCT
CTCTAGACGTTTTAAGGGCTGAGCCGGTCAGTTA
```

Each of the three lines (shown with wrap-around above) consists of 95 characters: the 94 letters from {**A**, **C**, **G**, **T**} and the character **\n** (not shown). The output file `dna_output.dat` must be structured as follows. For each pair of sequences #*i* and #*j*, with $i, j \in \{1, 2, 3\}$ and $i > j$, you should print:

- A single line, saying "**`Comparison between sequence #i and sequence #j:`**"
- The entire sequence #*i* in the *first row*, and the entire sequence #*j* in the *third row*.
- The comparison between the two sequences in the *second (middle) row*. This should be printed as follows. For each position, if the two bases are the *same* in both sequences then the corresponding base letter (one of **A**, **C**, **G**, **T**) should be printed; otherwise a blank **" "** should be printed.
- A single line, saying "**`The overlap percentage is x%`**" where *x* is a floating-point number which measures the percentage of letters that match in the two sequences. This number should be printed with a single digit of precision after the decimal point.

Each line in the output file `dna_output.dat` should contain at most 61 characters, including the end of line character **\n**. If the DNA sequences are longer than that, then each of the three rows mentioned above should be split across several lines, with the first few lines containing *exactly* 60 letters, and the last containing the rest of the letters. Here is a sample file `dna_output.dat` which results upon processing the file `dna_input.dat` above:

```
Comparison between sequence #1 and sequence #2:
ACGTTTTAAGGGCTGAGCTAGTCAGTTCATCGCGCGCGTATATCCTCGATCGATCATTCT
ACGTTTTAAGGGCT AG    T  G T ATCGCGCGCGTATATCCTCGATCGATCATTCT
ACGTTTTAAGGGCTTAGAGCTTATGCTAATCGCGCGCGTATATCCTCGATCGATCATTCT

CTCTAGACGTTTTAAGGGCTGAGCTAGTCAGTTC
CTCTAGACGTTTTAAGGGCT AG      A TT
CTCTAGACGTTTTAAGGGCTAAGGCGCGTAATTA

The overlap percentage is 80.9%


Comparison between sequence #1 and sequence #3:
ACGTTTTAAGGGCTGAGCTAGTCAGTTCATCGCGCGCGTATATCCTCGATCGATCATTCT
 CGTTT AAGGGCT AG TAGT AGTTCATCG   GCGTATATCCTCGATCGATCATTCT
TCGTTTGAAGGGCTTAGTTAGTTAGTTCATCGGCGGCGTATATCCTCGATCGATCATTCT

CTCTAGACGTTTTAAGGGCTGAGCTAGTCAGTTC
CTCTAGACGTTTTAAGGGCTGAGC  GTCAGTT
CTCTAGACGTTTTAAGGGCTGAGCCGGTCAGTTA

The overlap percentage is 88.3%


Comparison between sequence #2 and sequence #3:
ACGTTTTAAGGGCTTAGAGCTTATGCTAATCGCGCGCGTATATCCTCGATCGATCATTCT
 CGTTT AAGGGCTTAG    T  G T ATCG   GCGTATATCCTCGATCGATCATTCT
TCGTTTGAAGGGCTTAGTTAGTTAGTTCATCGGCGGCGTATATCCTCGATCGATCATTCT

CTCTAGACGTTTTAAGGGCTAAGGCGCGTAATTA
CTCTAGACGTTTTAAGGGCT AG CG   A TTA
CTCTAGACGTTTTAAGGGCTGAGCCGGTCAGTTA

The overlap percentage is 79.8%
```

Notes:

- As part of the solution, you are required to declare, define, and call the following functions. In these functions, you can assume that **`input`** and **`output`** are *global variables* of type **`FILE *`**.

  - The function **`read_DNA(char sequence[])`** that reads a DNA sequence from **`input`**, stores it in the array **`sequence[]`**, and returns the number of letters read, as an **`int`**.
  - The function **`compare_DNA(char seq1[], char seq2[], char seq3[], int n)`** that stores in the array **`seq3[]`** the comparison sequence of the two DNA sequences stored in **`seq1[]`** and **`seq2[]`**. The length of these DNA sequences is assumed to be **`n`**. The function returns, as a **`double`**, the percentage of overlap between the two DNA sequences.
  - The function **`print_DNA(char seq1[], char seq2[], char seq3[], int n)`** that prints to **`output`** the DNA sequences stored in **`seq1[]`** and **`seq2[]`**, as well as their comparison sequence stored in **`seq3[]`**, according to the rules explained above. The length of all these sequences is assumed to be **`n`**. The function does not return a value.

- The numbers 241 and 60, used above, should be defined as symbolic constants **`MAX_IN_LENGTH`** and **`OUT_LENGTH`**, using the **`#define`** compiler directive. The program should keep working correctly if the values of these symbolic constants are changed (within a reasonable range).