CS101 Lab-5

Today's Linux Commands

- Work in a directory named `~/lab5g1`
- We've learned how to redirect input from a command (`|`), a file (`<`) and how to redirect output to a file (`>`).
- Today, we will time our program to see how long it takes to run. We can then optimize our code to make it run quicker.
- The time command runs another command or program and, once it's finished, reports how long it took. It's an essential tool for seeing how efficient your code is.
- Syntax: `time ./my_program.out`
- Understanding its output:
    a. `real`: This is the total "wall clock" time. It's the amount of time that passed from the moment you hit Enter until the program finished. This includes time the CPU spent on other tasks or waiting for resources.
    b. `user`: This is the total time the CPU spent executing your program's code in "user mode." This is the most direct measure of how much CPU work your program did.
    c. `sys`: This is the total time the CPU spent executing system calls on behalf of your program (e.g., reading input or printing output).
- If you want to avoid taking time taken by input/output into consideration, you can also write `time ./my_program.out < in.txt > out.txt`
- Tired of retyping the same gcc command? The `history` command shows you a numbered list of the commands you've recently used. You can re-run a command from your history by typing `!` followed by its number (e.g. `!123`). You can also use keyboard up/down arrows to cycle through past commands.

Instructions for the lab assignment:

Solve the **first three problems** in the lab. The other problems are for homework.

1. Write a program to take as input a long int N. Write a loop that checks if any of the numbers from 2 … N-1 divides N. If so, print that N is composite. Else, print that N is prime. Time the runtime for large numbers, say, N=111191111. How about N=99999199999? If you are not patient enough, can you extrapolate the running time? Convince yourself that the "maximum work" happens when N is prime, and in that case, the computer has to do roughly N operations. We call it O(N) **time-complexity**.

2. The above program is very slow for large numbers. As discussed in the class, it suffices to check factors from 2 until ⌈√N⌉. Time the runtime for N=111191111 and N=99999199999. Convince yourself that the "maximum work" happens when N is prime, and in that case, the computer has to do roughly √N operations. We call it O(√N) time-complexity. This is a great improvement! People have also devised **algorithms** for this problem that only do roughly log N work! If you like this sort of questions, look at https://cp-algorithms.com/algebra/primality_tests.html and implement Miller-Rabin algorithm for checking primes that can be stored in 64 bits (i.e., as unsigned long).

3. Initialize an array of int with 5 elements. Write a program to compute its min, and max values. Modify the program to read the array size, say n, as input, then define an array of size n, and then scan n integers. For extra challenge, compute its median and mode. (There is a solution that does not use array sorting.)

4. Write programs for Mario-bricks (1) and Mario-bricks(2).

5. Build a countdown launch timer. Read the timer duration in seconds from the user and save it as int, say, `time_to_launch`. For input `time_to_launch = 10`, the program should print "`T-minus 10... 9... 8... 7... 6... 5... 4... 3... 2... 1... BLASTOFF!`" In between the print calls, to simulate countdown speed, use the `sleep()` function to pause your program. You'll need `#include<unistd.h>` header file. `sleep(1);` will pause the program for one second. C often "buffers" output, meaning it waits to print a chunk of text at once. To make the countdown appear in real-time, you must force the program to print immediately using `fflush(stdout);`. For a more dramatic effect, use the carriage return character `\r` in your `printf` statement.

This special character moves the cursor to the beginning of the current line without moving down, allowing you to overwrite the text. This creates a single, updating countdown timer instead of a new line for each second.

Solution for Lab-4 Problem 4 and Lab-5 Problem 5:

Use proper indentations, comments, and descriptive naming of variables.

```
#include<stdio.h>
#include<unistd.h>
int main() {
    int time_to_launch;
    scanf("%d", &time_to_launch);
    if(time_to_launch < 0) {
        printf("Launch aborted!\n");
        return 1;
    }
    while(time_to_launch > 0) {
        int hours = time_to_launch / 3600;
        int minutes = (time_to_launch % 3600) / 60;
        int seconds = time_to_launch % 60;
        printf("\rT-minus %02d:%02d:%02d...", hours, minutes,
seconds);
        fflush(stdout);
        sleep(1); // Sleep for 1 second to simulate countdown
speed
        time_to_launch--;
    }
    printf("\nBLASTOFF!\n");
    return 0;
}
```

You can also make the printing more fun:

```
for(int t=0; t<2; t++) {
      printf("\rT-minus %02d %02d %02d...", hours, minutes,
seconds); //without colons
      fflush(stdout);
      usleep(250000); // Sleep for 0.25 seconds
```

```
        printf("\rT-minus   %02d:%02d:%02d...",   hours,   minutes,
seconds); //with colons
        fflush(stdout);
        usleep(250000); // Sleep for 0.25 seconds
}
```