CS201 Lab-1

Data Structures and Algorithms

Shreya Agarwal & Rachit Nimavat

Recall `clock()` function from `time.h` we did in Lab-8 of CS101: It returns the number of clock "ticks" that have passed since the program started. This is a measure of processor time used, not calendar time. Notice that its return type is `clock_t` which can be safely typecasted to long int in our labs. As before, divide the difference between `clock()` return values by `CLOCKS_PER_SEC` to get runtime in seconds.

Today's Linux Command:

- Create your personal directory at `~/<your-name>` and a sub-directory named `lab1` inside it for today's lab
- Don't Google standard functions; use the system's built-in manual! Linux has a `man` command that provides a user manual for the argument.
- Before starting the second problem, type `man malloc` in your terminal.

Lab Assignment:

1. The Fibonacci sequence is defined as `F(n) = F(n-1) + F(n-2)` with base cases `F(0)=0` and `F(1)=1`. Write two functions:
   i. `long fib_recursive(int n);` The standard recursive function (calls itself twice)
   ii. `long fib_iterative(int n);` Uses a simple loop to calculate the sum.
   In `main()`, run and time both functions for `n=10`, `n=30`, and `n=45` using `clock()`. What do you find? Draw the recursion tree for `n=4` to explain why the recursive version is repeating work.

2. In CS101 we used Variable Length Arrays to create arrays of 'fixed' (but unknown at compile time) size. Real-world data structures like Python lists grow dynamically. Create a struct `IntVector` that acts as a growable array.

```c
typedef struct {
    int *data;    // Pointer to the array on the heap
    int size;     // Number of elements currently stored
    int capacity; // Total slots available in memory
} IntVector;
```

Implement the following functions and test them with some sample calls:

i. `IntVector create_vector(int initial_cap);` Creates the struct `IntVector` with `malloc`ed `data` array with `capacity=initial_cap`.

ii. `void append(IntVector *v, int val);` Adds `val` to the end of the `data` array, if possible. If not, `realloc` the `data` array to double its current capacity and print a message "Resizing from X to Y...", where X and Y are old and new sizes respectively. Then add `val` to the end of the `data` array.

iii. `void free_data(IntVector *v);` Free memory of the `data` array.

Questions to ponder over this week:

i. Consider two `IntVectors`. `IntVector v1 = create_vector(100);` and `IntVector v2 = create_vector(200);` What are `sizeof(v1)` and `sizeof(v2)`? Are they identical? Why or why not?

ii. Is it possible to free the memory of the `data` array by implementing `void free_data(IntVector v);`? Why or why not?

iii. How would you implement `void concat(IntVector *v1, IntVector *v2);` to concatenate elements of `v2` after `v1`?