16.35 - Real-Time Systems and Software
Asst. #1

Handed Out: February 20th, 2020
Due: February 27th, 2020 at 5:00 PM

Submission Instructions

Please submit your assignments electronically using `git`, and upload the files to Gradescope. Some notes about the submission:

1. Please submit your code for each question (named appropriately) and without object or executable files.
2. Your submission should consist of 4 files, names q1.c, q2.c, sorters.c, and sorters.h.
3. Write how long each question in a comment in your q1.c and q2.c files.

Please ensure that your code compiles and runs from within your source folder. You will not receive grades for code that does not compile.

Assignment Overview

This assignment will give 2 simple programs to write in `C`. Each of the programs is designed to test specific learning objectives from the class material. Partial credit will be given for incorrect implementations, provided that they **compile** (code that does not compile will receive no credit). The total number of available points is 50.

1 String Sorting using a Binary Tree [30]

In this program we will be creating a **binary tree**. This is a very (very) simple data structure that can be useful for sorting incoming data. The rules are very simple

- Each node contains an `item` (in this case a string), and a pointer to two other nodes.
- The upper node (if it exists) contains a node who's item precedes the item stored in the node while the lower node (if it exists) contains an item that follows the item stored in the node.
- To insert an item, we need only follow the tree, checking whether the item we wish to insert should precede or follow the item stored in each node. When we eventually reach an empty node, we insert it.
- If we wish to get an ordered list, we simply collapse the tree in the recursive order for each node (`nodes(upper), item, nodes(lower)`).

We are going to use this data structure to sort some incoming strings.

Input: `./q1 alligator porcupine giraffe zebra eland lion`

Output: `alligator eland giraffe lion porcupine zebra`

We will use the following structure definition for our nodes:

```
typedef struct Node {
    struct Node * upper;
    struct Node * lower;
    char * item;
} node;
```

We can use this in our functions very easily. To help you along, and to get you started for this problem, we've written some of the program for you. **Your task in this problem is to write the `insert` and `print_tree` commands for this program**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// Node structure
typedef struct Node {
    struct Node * upper;
    struct Node * lower;
    char * item;
} node;

void insert(node * root, char * str) {
  /** Y O U R   C O D E   H E R E  */
}

void print_tree(node * root) {
  /** Y O U R   C O D E   H E R E  */
}

void free_tree(node * root) {
    if (root != NULL) {
```

```
        free_tree(root->upper);
        free_tree(root->lower);
        free(root);
    }
}

int main (int argc, char ** argv) {
    node * root = malloc(sizeof(node));
    root->upper = NULL;
    root->lower = NULL;
    root->item = "";
    for (int i = 1; i < argc; i++){
        insert(root,argv[i]);
    }
    print_tree(root);
    free_tree(root);
}
```

Some advice:

1. Remember to initialize your fields that are pointers to some value when creating a structure (i.e. node) - one would typically initialize a pointer to `NULL`.
2. In this case, because we are only sorting strings and then printing them out, we don't need to copy them. You can simply assign the `item` pointer to the `str` pointer. As a result it is okay to initialize strings in this data structure to `""`.
3. Read the online help and description of the function `strcmp` - you will see that this will give you ordering information for this problem.

You may assume that the input to the program is only lower case strings.

Learning Objectives: Basic Memory Management, Structures, Pointers

2 Comparing Different Sorting Methods [20]

There are an unbelievable number of different sorting methods out there. You, as an intelligent and pragmatic developer, obviously wish to see how well your method compares to others out there. In this question we're going to help you do that.

We've provided a secondary source file called `sorters.c` with corresponding header file `sorters.h`. This file contains several sorting methods, including the extraordinarily inefficient bogosort - a sorting algorithm based on random chance. It will be your job to add a better sorting method to this file, along with parts of the framework for comparing and testing these methods.

The program is designed for two cases - in the first case, you simply pass the names of the sorting methods you wish to benchmark. These methods will then be benchmarked using an internal string array. We have written this part for you.

Input: `./q2 bogo bubble`

Output:
```
-------------------------------------------------
|    name      |    duration (ms)    |    correct    |
-------------------------------------------------
```

```
|    bubble    |          0.003 ms  |        yes   |
|     bogo     |        985.316 ms  |        yes   |
-------------------------------------------------
```

The second use case will allow you to pass your own string array to the function, on which all the methods (including one that you have written) will be run. This is your job.

Input: `./q2 elephant bacon gerbil avocado jesse jacob martha jimmy alleviate`

Output:
```
-------------------------------------------------
|    name      |    duration (ms)   |    correct   |
-------------------------------------------------
|    bubble    |          0.003 ms  |        yes   |
|     bogo     |        685.316 ms  |        yes   |
|    merge     |          0.002 ms  |        yes   |
|    quick     |          0.001 ms  |        yes   |
-------------------------------------------------
```

Your statement of work can be summarized as follows:

1. Implement a sorting method (eg. quicksort, mergesort) in the file `sorters.c`. Implementing it in the `q2.c` file will receive no credit.
2. Ensure your sorting method gets called in the benchmark (i.e. edit the code appropriately to ensure your sorting method is called).
3. Modify the existing code to test all the available methods on the arguments passed to the function. This code must only execute if the passed values do not match to named sorting methods (otherwise the original code should execute!).

Partial code for this problem is appended to this document (code will also be provided as a resource on stellar). It's up to you to figure out how to compile your code now that you have multiple files: feel free to look around on google or ask classmates.

Learning Objectives: Header Files, Multiple Source Files, Memory Allocation, Function Pointers

sorters.h

```c
#ifndef SORTERS_H
#define SORTERS_H
// Various sort methods
void bogo_sort(int argc, char ** args);
void bubble_sort(int argc, char ** args);
#endif // SORTERS_H
```

sorters.c

```c
// Various implementations of sorting algorithms.
#include "sorters.h"
#include <string.h>
#include <stdbool.h>
#include <stdio.h>
```

```c
#include <stdlib.h>

void swap(char ** elem1, char ** elem2) {
    char * temp = *elem1;
    *elem1 = *elem2;
    *elem2 = temp;
}

void bubble_sort(int argc, char ** args) {
    for (int i = 0; i < argc; i++) {
        for (int j = 0; j < argc - i - 1; j++) {
            if (strcmp(args[j], args[j + 1]) > 0) {
                swap(&args[j], &args[j + 1]);
            }
        }
    }
}

void bogo_sort(int argc, char ** args) {
    int num_swaps = 10;
    int seed = 2;
    srand(seed);
    // Idea: randomly shuffle. If the list is sorted, return. If not, try again.
    bool is_sorted = false;
    while (!is_sorted) {
        // Shuffle the list.
        for (int i = 0; i < num_swaps; i++) {
            int index1 = rand() % argc;
            int index2 = rand() % argc;
            swap(&args[index1], &args[index2]);
        }
        // Update is_sorted depending on list.
        is_sorted = true;
        for (int i = 0; i < argc - 1; i++) {
            if (strcmp(args[i], args[i + 1]) > 0) {
                is_sorted = false;  break;
            }
        }
    }
}
```

q2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "sorters.h"
// Test data for the algorithms
// Add more elements, and see how the algorithms scale!
#define N_ELEMENTS 10
char * data[N_ELEMENTS] = {"frog","gerbil","aardvark","horse","zebra",
```

```
                            // "monkey","penguin","blesbok","gazelle","nyala",
                            "elephant","seal","shark","hamster","bear"};
// Print function
void print_line(const char * name, double time, bool correct);
// Compares array (returns true if the array is sorted correctly).
bool sorted_correctly(int size, char ** arr);
// Copies the array
char ** get_copied_array(int size, char ** arr);
// Frees the copied array
void free_copied_array(int size,char ** copied_array);
// Structure for different sorting methods
typedef struct {
    char * name;
    void (*fnc) (int argc, char ** args);
} sorting_methods;
// Vector of methods - Add your method here!
#define N_SORTERS 2
sorting_methods sorters[N_SORTERS] = {{"bubble",bubble_sort},{"bogo",bogo_sort}};
// main function
int main(int argc, char ** argv) {
    // MODIFY THIS MAIN LOOP
    // print a table of the results:
    printf("-----------------------------------------------------\n");
    printf("|    name      |    duration (ms)   |    correct    |\n");
    printf("-----------------------------------------------------\n");
    for (int requested_sorter = 1; requested_sorter < argc; requested_sorter++ ) {
      for (int sorter_idx = 0; sorter_idx < N_SORTERS; sorter_idx++) {
          if (strcmp(argv[requested_sorter],sorters[sorter_idx].name) == 0) {
            char ** copied_array = get_copied_array(N_ELEMENTS,data);
            clock_t start_time = clock();
            sorters[sorter_idx].fnc(N_ELEMENTS, copied_array);
            clock_t end_time = clock();
            double time = ((double)(end_time - start_time))/CLOCKS_PER_SEC*1000;
            bool correct = sorted_correctly(N_ELEMENTS, copied_array);
            print_line(sorters[sorter_idx].name,time,correct);
            free_copied_array(N_ELEMENTS,copied_array);
            break;
        }
      }
    }
    printf("-----------------------------------------------------\n");
    return 0;
}

/********************** Do not edit these functions ************************/

// prints a table line
void print_line(const char * name, double time, bool correct){
  int space = 14 - strlen(name);
  printf("|"); if (space > 0) for (int i = 0; i < space/2; i++) printf(" ");
  printf("%s",name);
  if (space > 0) for (int i = 0; i < space/2; i++) printf(" ");
  if (space %2 !=0) printf(" ");
  printf("|"); char tmp [20]; sprintf(tmp,"%.3f ms", time);
```

```
  space = 20 - strlen(tmp);
  if (space > 2) for (int i = 0; i < space-2; i++) printf(" ");
  printf("%s",tmp); printf("  |");
  if (correct) printf("        yes   |\n");
  else         printf("         no   |\n");
}

bool sorted_correctly(int size, char ** arr){
    bool v = true; for (int i =1; i < size; i++) v &= strcmp(arr[i-1],arr[i]) <= 0;
    return v;
}

char ** get_copied_array(int size, char ** arr){
  char ** copied_array = malloc(sizeof(char*)*size);
  for (int i = 0; i < size; i++) {
      copied_array[i] = malloc(strlen(arr[i])+1);
      strcpy(copied_array[i], data[i]);
  }
  return copied_array;
}

void free_copied_array(int size,char ** copied_array) {
  for (int i = 0; i < size; i++) free(copied_array[i]);
  free(copied_array);
}
```