Assignment 4
The Perambulations of Denver Long

Prof. Darrell Long
CSE 13S – Fall 2021

Due: October 24th at 11:59 pm

1 Introduction

I wonder why it is that when I plan a route too carefully, it goes to pieces, whereas if I blunder along in blissful ignorance aimed in a fancied direction I get through with no trouble.

—John Steinbeck, *Travels with Charley: In Search of America*

Denver Long decided to augment his income during his retirement years by selling the prestigious products produced by the Shinola Corporation. He enjoys driving his Cadillac, so it's the life of a traveling salesman for him. He loads up his little dog Satan—whom he calls *Baby*—and heads out on his new career.

But his first trip does not go so well. Heading to his son's house after visiting his new grandson, he accidentally takes a wrong turn near Chula Vista and winds up in Mexico. Despite many pleasant visits to Tijuana in years past, visiting his old friend Señor Vasquez on his ranchero, shooting their rifles at an old El Dorado that he has traded to Señor Vasquez decades before, it's no longer the familiar Mexico of 1974. Having no passport, and speaking very little Spanish, he turns around in frustration. The Border Patrol won't let him back into the United States for many hours, until he finally wears them down through his power of persuasion.

Since the profit of his new enterprise depends on the cost and duration of travel, and losing a day in Tijuana cost him a potential sale in Barstow, he asks his eldest son to have his class create a computer program that will provide an optimal route to all the cities along his way and then return him to his home in scenic Clearlake.

2 Directed Graphs

I mean, if 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself "Dijkstra would not have liked this," well, that would be enough immortality for me.

—Edsger W. Dijkstra

A graph is a data structure $G = \langle V, E \rangle$ where $V = \{v_0, \ldots, v_n\}$ is the set of vertices (or nodes) and $E = \left\{\langle v_i, v_j \rangle, \ldots\right\}$ is the set of edges that connect the vertices. For example, you might have a set of cities $V = \{\text{El Cajon}, \text{La Mesa}, \text{San Diego}, \ldots, \text{La Jolla}\}$ as the vertices and write "El Cajon" → "Lakeside" to indicate that there is a path (Los Coches Road) from El Cajon to Lakeside. If there is a path from El Cajon to Lakeside, as well as a path from Lakeside to El Cajon, then the edge connecting El Cajon and Lakeside is *undirected*.

Such a simple graph representation simply tells you how vertices are connected and provides the idea of one-way roads. But it really does not help Denver, since it does not provide any notion of distance. We solve this problem by associating a weight with each edge and so we might write "Santee → El Cajon, 2" to indicate that there is a path of two miles long from Santee to El Cajon. Given a set of edges and weights, then we can then find the shortest path from any vertex to any other vertex (the answer may be that there is no such path). There are elegant (and quick) algorithms for computing the shortest path, but that is not exactly what we want to do. We want to find a path through all of the vertices, visiting each *exactly once*, such that there is a direct (single step) connection from the last vertex to the first. This is called a *Hamiltonian path*. This will address Denver's need to get home, but won't necessarily be the shortest such path. So, we need to go through every possible Hamiltonian path given the list of cities Denver is traveling through and pick the shortest path. Note: if you find yourself on a path that is longer than the current best found Hamiltonian path, then you can preemptively reject your current path.

3 Representing Graphs

We live on a placid island of ignorance in the midst of black seas of infinity, and it was not meant that we should voyage far.

—H. P. Lovecraft, *The Call of Cthulhu*

Perhaps the simplest way to represent a graph is with an *adjacency matrix*. Consider an $n \times n$ adjacency matrix M, where n is the number of vertices in the graph. If $M_{i,j} = k$, where $1 \le i \le j \le n$, then we say that there exists a *directed edge* from vertex i to vertex j with weight k. Traveling through wormholes is considered hazardous, so any valid edge weight k must be non-zero and positive.

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & \cdots & 25 \\ \hline 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 5 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 5 \\ 3 & 0 & 0 & 0 & 0 & 21 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Each edge will be represented as a triple $\langle i, j, k\rangle$. The set of edges in the adjacency matrix above is

$$E = \{\langle 0,1,10\rangle, \langle 1,2,2\rangle, \langle 1,3,5\rangle, \langle 2,5,3\rangle, \langle 2,25,5\rangle, \langle 3,4,21\rangle\}.$$

If the above adjacency matrix were made to be *undirected*, it would be reflected along the diagonal.

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & \cdots & 25 \\ \hline 0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 10 & 0 & 2 & 5 & 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 & 0 & 3 & 0 & 5 \\ 3 & 0 & 5 & 0 & 0 & 21 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 21 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ \vdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 25 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The first of the ADTs you will need to implement for this assignment is for the graph. An **ADT** is an *abstract data type*. With any ADT comes an interface comprised of *constructor*, *destructor*, *accessor*, and *manipulator* functions.

```c
struct Graph {
    uint32_t vertices;                      // Number of vertices.
    bool undirected;                        // Undirected graph?
    bool visited[VERTICES];                 // Where have we gone?
    uint32_t matrix[VERTICES][VERTICES];    // Adjacency matrix.
};
```

We elect to use an adjacency matrix with set maximum dimensions. This is both to simplify the abstraction and also due to the computational complexity of solving the Traveling Salesman Problem (TSP) with depth-first search (DFS), which is discussed in §5. The `VERTICES` macro will be defined and supplied to you in `vertices.h`. In this header file, there is another macro `START_VERTEX` which defines the origin vertex of the shortest Hamiltonian path we will be searching for. You may not modify this file. The `struct` definition of a graph *must* go in `graph.c`.

vertices.h

```c
#pragma once

#define START_VERTEX 0   // Starting (origin) vertex.
#define VERTICES     26  // Maximum vertices in graph.
```

The interface for the graph ADT is defined as follows:

`Graph *graph_create(uint32_t vertices, bool undirected)`

The constructor for a graph. A constructor function is responsible for initializing and allocating any memory required for the type it is constructing. It is through this constructor in which a graph can be specified to be undirected. Make sure each cell of the adjacency matrix, `matrix`, is set to zero. Also make sure that each index of the `visited` array is initialized as `false` to reflect that no vertex has been visited yet. The `vertices` field reflects the number of vertices in the graph. A working constructor for a graph is provided below. Note that it uses `calloc()` for *dynamic memory allocation*. This function is included in `<stdlib.h>`.

```c
Graph *graph_create(uint32_t vertices, bool undirected) {
    Graph *G = (Graph *)calloc(1, sizeof(Graph));
    G->vertices = vertices;
    G->undirected = undirected;
    return G;
}
```

`void graph_delete(Graph **G)`

The destructor for a graph. A working destructor for a `Graph` is provided below. Any memory that is allocated using one of `malloc()`, `realloc()`, or `calloc()` *must* be freed using the `free()` function. The job of the destructor is to free all the memory allocated by the constructor. Your programs are expected to be free of memory leaks. A pointer to a pointer is used as the parameter because we want to avoid *use-after-free* errors. A use-after-free error occurs when a program uses a pointer that points to freed memory. To avoid this, we pass the *address of a pointer* to the destructor function. By *dereferencing* this double pointer, we can make sure that the pointer that pointed to allocated memory is updated to be `NULL`.

```c
void graph_delete(Graph **G) {
    free(*G);
    *G = NULL;
    return;
}
```

`uint32_t graph_vertices(Graph *G)`

Since we will be using `typedef` to create *opaque* data types, we need functions to access fields of a data type. These functions are called *accessor* functions. An opaque data type means that users do not need to know its implementation outside of the implementation itself. This also means that it is incorrect to write `G->vertices` outside of `graph.c` since it violates opacity. This accessor function returns the number of vertices in the graph.

`bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)`

The need for *manipulator* functions follows the rationale behind the need for accessor functions: there needs to be some way to alter fields of a data type.

This function adds an edge of weight `k` from vertex `i` to vertex `j`. If the graph is undirected, add an edge, also with weight `k` from `j` to `i`. Return `true` if both vertices are within bounds and the edge(s) are successfully added and `false` otherwise.

`bool graph_has_edge(Graph *G, uint32_t i, uint32_t j)`

Return `true` if vertices `i` and `j` are within bounds and there exists an edge from `i` to `j`. Remember: an edge exists if it has a non-zero, positive weight. Return `false` otherwise.

`uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j)`

Return the weight of the edge from vertex `i` to vertex `j`. If either `i` or `j` aren't within bounds, or if an edge doesn't exist, return 0.

`bool graph_visited(Graph *G, uint32_t v)`

Return `true` if vertex `v` has been visited and `false` otherwise.

`void graph_mark_visited(Graph *G, uint32_t v)`

If vertex `v` is within bounds, mark `v` as visited.

`void graph_mark_unvisited(Graph *G, uint32_t v)`

If vertex `v` is within bounds, mark `v` as unvisited.

`void graph_print(Graph *G)`

A debug function you will want to write to make sure your graph ADT works as expected.

4 Depth-first Search

Again it might have been the American tendency in travel. One goes, not so much to see but to tell afterward.

—John Steinbeck, *Travels with Charley: In Search of Amerca*

We need a methodical procedure for searching through the graph. Once we have examined a vertex, we do not want to do so again—we don't want Denver going through cities where he has already been (he has been known to wear out his welcome: charming women and fighting men).

Depth-first search (DFS) first marks the vertex v as having been visited, then it iterates through all of the edges $\langle v, w\rangle$, recursively calling itself starting at w if w has not already been visited.

```
procedure DFS(G,v):
    label v as visited
    for all edges from v to w in G.adjacentEdges(v) do
       if vertex w is not labeled as visited then
          recursively call DFS(G,w)
    label v as unvisited
```

Finding a Hamiltonian path then reduces to:

1. Using DFS to find paths that pass through all vertices, and
2. There is an edge from the last vertex to the first. The solutions to the Traveling Salesman Problem are then the shortest found Hamiltonian paths.

5 Computational Complexity

Many a trip continues long after movement in time and space have ceased. I remember a man in Salinas who in his middle years traveled to Honolulu and back, and that journey continued for the rest of his life. We could watch him in his rocking chair on his front porch, his eyes squinted, half-closed, endlessly traveling to Honolulu.

—John Steinbeck, *Travels with Charley: In Search of Amerca*

How long will this take? The answer is, it will take a very long time if there are a large number of vertices. The running time of the simplest algorithm is $O(n!)$ and there is no known algorithm that runs in less than $O(2^n)$ time. In fact, the TSP has been shown to be NP-hard, which means that it is as difficult as any problem in the class NP (you will learn more about this in CSE 104: Computability and Computational Complexity). Basically, it means that it can be solved in polynomial time if you have a magical computer that at each if-statement is takes both branches every time (creating a copy of the computer for each such branch).

6 Stacks

You will need to implement the *stack* ADT for tracking the path Denver has traveled. This section defines the interface for a stack. A stack is an ADT that implements a *last-in, first-out*, or **LIFO**, policy. Consider a stack of pancakes. A pancake can only be placed (*pushed*) on top of the stack and can only be removed (*popped*) from the top of the stack. We do not eat pancakes from the middle or bottom of a stack. The header file containing the interface will be given to you as `stack.h`. You may not modify this file. If you borrow code from any place — including Prof. Long — you must cite it. It is *far better* if you write it yourself.

The stack datatype will be abstracted as a `struct` called `Stack`. We will use a `typedef` to construct a new type, which you should treat as opaque — which means that you cannot manipulate it directly, like the graph ADT. We will *declare* the `stack` type in `stack.h` and you will define its concrete implementation in `stack.c`. This means the following `struct` definition *must* be in `stack.c`.

```c
struct Stack {
    uint32_t top;        // Index of the next empty slot.
    uint32_t capacity;   // Number of items that can be pushed.
    uint32_t *items;     // Array of items, each with type uint32_t.
};
```

`Stack *stack_create(uint32_t capacity)`

The constructor function for a `Stack`. The `top` of a stack should be initialized to 0. The capacity of a stack is set to the specified capacity. The specified capacity also indicates the number of items to allocate memory for, the items in which are held in the dynamically allocated array `items`. A working constructor for a stack is provided below. Note that it uses `malloc()`, as well as `calloc()`, for *dynamic memory allocation*. Both these functions are included from `<stdlib.h>`.

```c
Stack *stack_create(uint32_t capacity) {
    Stack *s = (Stack *) malloc(sizeof(Stack));
    if (s) {
        s->top = 0;
        s->capacity = capacity;
        s->items = (uint32_t *) calloc(capacity, sizeof(uint32_t));
        if (!s->items) {
            free(s);
            s = NULL;
        }
    }
    return s;
}
```

`void stack_delete(Stack **s)`

The destructor function for a stack. A working destructor is provided below. Pay attention to the things that are freed and what is set to `NULL`.

```c
void stack_delete(Stack **s) {
    if (*s && (*s)->items) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}
```

`bool stack_empty(Stack *s)`

Returns `true` if the stack is empty and `false` otherwise.

`bool stack_full(Stack *s)`

Returns `true` if the stack is full and `false` otherwise.

`uint32_t stack_size(Stack *s)`

Returns the number of items in the stack.

`bool stack_push(Stack *s, uint32_t x)`

The need for *manipulator* functions follows the rationale behind the need for accessor functions: there needs to be some way to alter fields of a data type. `stack_push()` is a manipulator function that pushes an item `x` to the top of a stack.

This function returns a `bool` in order to signify either success or failure when pushing onto a stack. When can pushing onto a stack result in failure? *When the stack is full.* If the stack is full prior to pushing the item `x`, return `false` to indicate failure. Otherwise, push the item and return `true` to indicate success.

`bool stack_pop(Stack *s, uint32_t *x)`

This function pops an item off the specified stack, passing the value of the popped item back through the pointer `x`. Like with `stack_push()`, this function returns a `bool` to indicate either success or failure. When can popping a stack result in failure? *When the stack is empty.* If the stack is empty prior to popping it, return `false` to indicate failure. Otherwise, pop the item, set the value in the memory `x` is pointing to as the popped item, and return `true` to indicate success.

```c
// Dereferencing x to change the value it points to.
*x = s->items[s->top];
```

`bool stack_peek(Stack *s, uint32_t *x)`

Peeking into a stack is synonymous with querying a stack about the element at the top of the stack. If the stack is empty prior to peeking into it, return `false` to indicate failure. This functions like `stack_pop()`, but doesn't modify the contents of the stack.

`void stack_copy(Stack *dst, Stack *src)`

Assuming that the destination stack `dst` is properly initialized, make `dst` a copy of the source stack `src`. This means making the *contents* of `dst->items` the same as `src->items`. The top of `dst` should also match the top of `src`.

`void stack_print(Stack *s, FILE *outfile, char *cities[])`

Prints out the contents of the stack to `outfile` using `fprintf()`. Working through each vertex in the stack starting from the *bottom*, print out the name of the city each vertex corresponds to. This function will be given to aid you.

```c
void stack_print(Stack *s, FILE *outfile, char *cities[]) {
    for (uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, "%s", cities[s->items[i]]);
        if (i + 1 != s->top) {
            fprintf(outfile, " -> ");
        }
    }
    fprintf(outfile, "\n");
}
```

7 Paths

We find after years of struggle that we do not take a trip; a trip takes us.

—John Steinbeck, *Travels with Charley: In Search of Amerca*

Given that vertices are added to and removed from the traveled path in a stack-like manner, we decide to abstract a path as follows:

```c
struct Path {
    Stack *vertices; // The vertices comprising the path.
    uint32_t length; // The total length of the path.
};
```

The following functions define the interface for the path ADT.

`Path *path_create(void)`

The constructor for a path. Set `vertices` as a freshly created stack that can hold up to `VERTICES` number of vertices. Initialize `length` to be 0. The `length` field will track the length of the path. In other words, it holds the sum of the edge weights between consecutive vertices in the `vertices` stack.

`void path_delete(Path **p)`

The destructor for a path. Remember to set the pointer `p` to `NULL`.

`bool path_push_vertex(Path *p, uint32_t v, Graph *G)`

Pushes vertex `v` onto path `p`. The `length` of the path is *increased* by the edge weight connecting the vertex at the top of the stack and `v`. Return `true` if the vertex was successfully pushed and `false` otherwise.

`bool path_pop_vertex(Path *p, uint32_t *v, Graph *G)`

Pops the `vertices` stack, passing the popped vertex back through the pointer `v`. The `length` of the path is *decreased* by the edge weight connecting the vertex at the top of the stack and the popped vertex. Return `true` if the vertex was successfully popped and `false` otherwise.

`uint32_t path_vertices(Path *p)`

Returns the number of vertices in the path.

`uint32_t path_length(Path *p)`

Returns the length of the path.

`void path_copy(Path *dst, Path *src)`

Assuming that the destination path `dst` is properly initialized, makes `dst` a copy of the source path `src`. This will require making a copy of the `vertices` stack as well as the `length` of the source path.

`void path_print(Path *p, FILE *outfile, char *cities[])`

Prints out a path to `outfile` using `fprintf()`. Requires a call to `stack_print()`, as defined in §7.10, in order to print out the contents of the `vertices` stack.

8 Command-line Options

Attitude is a choice. Happiness is a choice. Optimism is a choice. Kindness is a choice. Giving is a choice. Respect is a choice. Whatever choice you make makes you. Choose wisely.

—Roy T. Bennett, *The Light in the Heart*

Your program must support any combination of the following command-line options.

- `-h`: Prints out a help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- `-v`: Enables verbose printing. If enabled, the program prints out *all* Hamiltonian paths found as well as the total number of recursive calls to `dfs()`.
- `-u`: Specifies the graph to be undirected.
- `-i infile`: Specify the input file path containing the cities and edges of a graph. If not specified, the default input should be set as `stdin`.
- `-o outfile`: Specify the output file path to print to. If not specified, the default output should be set as `stdout`.

9 Reading An Input Graph

It behooves a man who wants to see wonders sometimes to go out of his way.

—John Mandeville, *The Travels of Sir John Mandeville*

We will be storing graphs in specially formatted files. Here is an example:

```
$ cat mythical.graph
4
Asgard
Elysium
Olympus
Shangri-La
0 3 5
3 2 4
2 1 10
1 0 2
```

The first line of a graph file is the number of vertices, or cities, in the graph. Assuming n is the number of vertices, the next n lines of the file are the names of the cities. Each line after that is an edge. It is to be scanned in as a triple $\langle i, j, k \rangle$ and interpreted as an edge from vertex i to vertex j with weight k.

10 Specifics

The gladdest moment in human life, methinks, is a departure into unknown lands.

—Sir Richard Burton

Here are the specifics for your program implementation.

1. Parse command-line options with looped calls to `getopt()`. This should be familiar from assignments 2 and 3.

2. Scan in the first line from the input. This will be the number of vertices, or cities, that will be in the graph. Print an error if the number specified is greater than `VERTICES`, the macro defined in `vertices.h`.

3. Assuming the number of specified vertices is n, read the next n lines from the input using `fgets()`. Each line is the name of a city. Save the name of each city to an array. You will want to either make use of `strdup()` from `<string.h>` or implement your own `strdup()` function. If the line is malformed, print an error and exit the program. Note: using `fgets()` will leave in the newline character at the end, so you will manually have to change it to the null character to remove it.

4. Create a new graph G, making it undirected if specified.

5. Scan the input line by line using `fscanf()` until the end-of-file is reached. Add each edge to G. If the line is malformed, print an error and exit the program.

6. Create two paths. One will be for tracking the current traveled path and the other for tracking the shortest found path.

7. Starting from the origin vertex, defined by the macro `START_VERTEX` in `vertices.h`, perform a depth-first search on *G* to find the shortest Hamiltonian path. Here is an example function prototype that you may use as the recursive depth-first function:

```c
void dfs(Graph *G, uint32_t v, Path *curr, Path *shortest, char *
    cities[], FILE *outfile);
```

   The parameter `v` is the vertex that you are currently on. The currently traversed path is maintained with `curr`. The shortest found path is tracked with `shortest`. The array of city names is `cities`. Finally, `outfile` is the output to print to.

8. After the search, print out the length of the shortest path found, the path itself (remember to return back to the origin), and the number of calls to `dfs()`.

```
$ ./tsp < mythical.graph
Path length: 21
Path: Asgard -> Shangri-La -> Olympus -> Elysium -> Asgard
Total recursive calls: 4
```

   If the verbose command-line option was enabled, print out *all* the Hamiltonian paths that were found as well. It is recommended that you print out the paths as you find them.

```
$ ./tsp -v < ucsc.graph
Path length: 7
Path: Cowell -> Stevenson -> Merrill -> Cowell
Path length: 6
Path: Cowell -> Merrill -> Stevenson -> Cowell
Path length: 6
Path: Cowell -> Merrill -> Stevenson -> Cowell
Total recursive calls: 5
```

The output of your program must match the output of the reference program when given a properly formatted graph exactly to receive full credit.

11 Deliverables

Travel isn't always pretty. It isn't always comfortable. Sometimes it hurts, it even breaks your heart. But that's okay. The journey changes you; it should change you. It leaves marks on your memory, on your consciousness, on your heart, and on your body. You take something with you. Hopefully, you leave something good behind.

—Anthony Bourdain

You will need to turn in the following source code and header files:

1. Your program, called `tsp`, *must* have the following source and header files:

   - `graph.h` specifies the interface to the graph ADT.
   - `graph.c` implements the graph ADT.
   - `path.h` specifies the interface to the path ADT.
   - `path.c` implements the path ADT.
   - `stack.h` specifies the interface to the stack ADT.
   - `stack.c` implements the stack ADT.
   - `tsp.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.
   - `vertices.h` defines macros regarding vertices.

You can have other source and header files, but *do not try to be overly clever*. You may not modify any of the supplied header files. You will also need to turn in the following:

1. `Makefile`:

   - `CC = clang` must be specified.
   - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
   - `make` must build the `tsp` executable, as should `make all` and `make tsp`.
   - `make clean` must remove all files that are compiler generated.
   - `make format` should format all your source code, including the header files.

2. `README.md`: This must use proper Markdown syntax. It must describe how to use your program and `Makefile`. It should also list and explain any command-line options that your program accepts. Any false positives reported by `scan-build` should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.

3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

4. Your code must pass `scan-build` *cleanly*.

12 Submission

Refer back assignment 0 for the instructions on how to properly submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

13 Supplemental Readings

Books are not made to be believed, but to be subjected to inquiry. When we consider a book, we mustn't ask ourselves what it says but what it means…

—Umberto Eco

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 4 §4.10
  - Chapter 7 §7.4–7.8
- *Introduction to Algorithms* by T. Cormen, C. Leiserson, R. Rivest, & C. Stein
  - Chapter 10 §10.1
  - Chapter 35 §35.2

Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn.