Assignment 5
Sorting: Putting your affairs in order

Prof. Darrell Long
CSE 13S – Winter 2021

Due: February 15th at 11:59 pm

Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.

—Donald Knuth, Vol. III, *Sorting and Searching*

1 Introduction

Putting items into a sorted order is one of the most common tasks in Computer Science. As a result, there are a myriad of library routines that will do this task for you, but that does not absolve you of the obligation of understanding how it is done. In fact it behooves you to understand the various algorithms in order to make wise choices.

The best execution time that can be accomplished, also referred to as the *lower bound*, for sorting using *comparisons* is $\Omega(n \log n)$, where n is the number is elements to be sorted. If the universe of elements to be sorted is small, then we can do better using a *Count Sort* or a *Radix Sort* both of which have a time complexity of $O(n)$. The idea of *Count Sort* is to count the number of occurrences of each element in an array. For *Radix Sort*, a digit by digit sort is done by starting from the least significant digit to the most significant digit. It may also use *Count Sort* as a subroutine.

What is this O and Ω stuff? It's how we talk about the execution time (or space used) by a program. We will discuss it in class, and you will see it again in your Data Structures and Algorithms class, now named CSE 101.

The sorting algorithms that you are expected to implement are Bubble Sort, Shell Sort, Quicksort and Heapsort. The purpose of this assignment is to get you fully familiarized with each sorting algorithm. They are well-known sorts. You can use the Python pseudocode provided to you as guidelines. Do not get the code for the sorts from the Internet or you will be referred to for cheating. We will be running plagiarism checkers.

2 Bubble Sort

C is peculiar in a lot of ways, but it, like many other successful things, has a certain unity of approach that stems from development in a small group.

—Dennis Ritchie

Bubble Sort works by examining adjacent pairs of items. If the second item is smaller than the first, swap them. As a result, the largest element falls to the bottom of the array in a single pass. Since it is in fact the largest, we do not need to consider it again. So in the next pass, we only need to consider $n-1$ pairs of items. The first pass requires n pairs to be examined; the second pass, $n-1$ pairs; the third pass $n-2$ pairs, and so forth. If you can pass over the entire array and no pairs are out of order, then the array is sorted.

Pre-lab Part 1

1. How many rounds of swapping do you think you will need to sort the numbers $8, 22, 7, 9, 31, 5, 13$ in ascending order using Bubble Sort?
2. How many comparisons can we expect to see in the worse case scenario for Bubble Sort? Hint: make a list of numbers and attempt to sort them using Bubble Sort.
3. How would you revise the algorithm so the smallest element floats to the top instead?

In 1784, when Carl Friedrich Gauß was only 7 years old, he was reported to have amazed his elementary school teacher by how quickly he summed up the integers from 1 to 100. The precocious little Gauß produced the correct answer immediately after he quickly observed that the sum was actually 50 pairs of numbers, with each pair summing to 101 totaling to 5,050. We can then see that:

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$$

so the *worst case* time complexity is $O(n^2)$. However, it could be much better if the list is already sorted. If you haven't seen the inductive proof for this yet, you will in the applied discrete mathematics class.

Bubble Sort in Python

```python
def bubble_sort(arr):
    n = len(arr)
    swapped = True
    while swapped:
        swapped = False
        for i in range(1, n):
            if arr[i] < arr[i - 1]:
                arr[i], arr[i - 1] = arr[i - 1], arr[i]
                swapped = True
        n -= 1
```

3 Shell Sort

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—C.A.R. Hoare

Donald L. Shell (March 1, 1924–November 2, 2015) was an American computer scientist who designed the Shell sort sorting algorithm. He acquired his Ph.D. in Mathematics from the University of Cincinnati in 1959, and published the shell sort algorithm in the Communications of the ACM in July that same year.

Shell Sort is a variation of insertion sort, which sorts pairs of elements which are far apart from each other. The *gap* between the compared items being sorted is continuously reduced. Shell Sort starts with distant elements and moves out-of-place elements into position faster than a simple nearest neighbor exchange. What is the expected time complexity of Shell Sort? It depends upon the gap sequence.

The following is the pseudocode for Shell Sort. The gap sequence is represented by the array `gaps`. You will be given a gap sequence, the Pratt sequence (2^p3^q also called 3-smooth), in the header file `gaps.h`. For each gap in the gap sequence, the function compares all the pairs in `arr` that are gap indices away from each other. Pairs are swapped if they are in the wrong order.

Shell Sort in Python

```python
def shell_sort(arr):
    for gap in gaps:
        for i in range(gap, len(arr)):
            j = i
            temp = arr[i]
            while j >= gap and temp < arr[j - gap]:
                arr[j], arr[j - gap] = arr[j - gap], arr[j]
                j -= gap
            arr[j] = temp
```

Pre-lab Part 2

1. The worst time complexity for Shell Sort depends on the sequence of gaps. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap size? Cite any sources you used.

4 Quicksort

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Edsger Dijkstra

Quicksort (sometimes called partition-exchange sort) is, on average, the most efficient sorting algorithm. It was Developed by British computer scientist C.A.R. "Tony" Hoare in 1959 and published in 1961. It is perhaps the most commonly used algorithm for sorting (by competent programmers). When implemented well, it is the fastest known algorithm that sorts using *comparisons*. It is usually two or three times faster than its main competitors, Merge Sort and Heapsort. It does, though, have a worst case performance of $O(n^2)$ while its competitors are strictly $O(n \log n)$ in their worst case.

Quicksort is a divide-and-conquer algorithm. It partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array.

Note that Quicksort is an *in-place* algorithm, meaning it doesn't allocate additional memory for sub-arrays to hold partitioned elements. Instead, Quicksort utilizes a subroutine called `partition()` that places elements less than the pivot into the left side of the array and elements greater than or equal to the pivot into the right side and returns the index that indicates the division between the partitioned parts of the array. In a recursive implementation, Quicksort is then run recursively on the partitioned parts of the array, thereby sorting each array partition containing at least one element. Instead of a recursive Quicksort, you will be writing an *iterative* Quicksort that utilizes a *stack*. Indices are pushed into the stack two at a time. The first and second indices represent the leftmost and rightmost indices of the array partition to sort. Hint: the return type of partition should be `int64_t`.

Partition in Python

```python
def partition(arr, lo, hi):
    pivot = arr[lo + ((hi - lo) // 2)]; # Prevent overflow.
    i = lo - 1
    j = hi + 1

    while i < j:
        i += 1 # You may want to use a do-while loop.
        while arr[i] < pivot:
            i += 1

        j -= 1
        while arr[j] > pivot:
            j -= 1

        if i < j:
            arr[i], arr[j] = arr[j], arr[i]

    return j
```

Quicksort in Python

```python
def quick_sort(arr):
    left = 0
    right = len(arr) - 1

    stack = []
    stack.append(left) # Pushes to the end of the stack.
    stack.append(right)

    while len(stack) != 0:
        hi = stack.pop() # Pops off the end of the stack.
        lo = stack.pop()
        p = partition(arr, lo, hi)
        if p + 1 < hi:
            stack.append(p + 1)
            stack.append(hi)
        if lo < p:
            stack.append(lo)
            stack.append(p)
```

Pre-lab Part 3

1. Quicksort, with a worse case time complexity of O((n^2)), doesn't seem to live up to its name. Investigate and explain why Quicksort isn't doomed by its worst case scenario. Make sure to cite any sources you use.

4.1 Stacks

You will need to implement the *stack* ADT for your iterative Quicksort. The following subsections define the interface for 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.

4.1.1 `Stack`

The stack `struct` must be defined as follows:

```c
struct Stack {
  uint32_t top;        // Points to the next empty slot.
  uint32_t capacity;   // Number of items that can be pushed.
  int64_t *items;      // Holds the items.
};
```

This `struct` *must* be placed in `stack.c`. The field `top` indicates the index that the next pushed item should go. The field `capacity` indicates the number of items that the stack can hold. Finally, the field `items` is the underlying array in which items are stored in. The type of each item will be a `int64_t`.

4.1.2 `Stack *stack_create(void)`

The constructor function for a stack. The `top` of a stack should be initialized to 0. The default starting capacity of a stack should be `MIN_CAPACITY`, a macro that is defined in `stack.h`. The starting capacity also indicates the amount of memory to allocate for `items`.

4.1.3 `void stack_delete(Stack **s)`

The destructor function for a stack. Remember: your program *must* be free of memory leaks.

4.1.4 `bool stack_empty(Stack *s)`

This functions checks if the specified stack is empty or not. If the stack is empty, return `true`. Otherwise, return `false`.

4.1.5 `bool stack_push(Stack *s, int64_t x)`

You may notice that the interface does not include a function to check if a specified stack is full or not. This is because you will be implementing a *dynamically* growing stack, as demonstrated in lecture. If the `top` of the stack matches its `capacity`, double the capacity and use `realloc()` to reallocate the memory that `items` points to. The amount of memory to reallocate is given by the new capacity. In the event that reallocating this memory fails, return `false`. Otherwise, push item `x` into the stack and return `true` to indicate success.

4.1.6 `bool stack_pop(Stack *s, int64_t *x)`

Pops an item off the specified stack, passing the value of the item back through the pointer to `x`. Hint: you will need to dereference `x` in order to change the value in the memory `x` is pointing to. If the specified stack is empty prior to popping, return `false`; you cannot pop an empty stack. Otherwise, pop an item off the stack, pass it back using `x`, and return `true` to indicate success.

4.1.7 `void stack_print(Stack *s)`

This is a debug function that you should write early on. It will help greatly in determining whether or not your stack implementation is working correctly.

5 Heapsort

Increasingly, people seem to misinterpret complexity as sophistication, which is baffling – the incomprehensible should cause suspicion rather than admiration.

—Niklaus Wirth

Heapsort, along with the heap data structure, was invented in 1964 by J. W. J. Williams. The heap data structure is typically implemented as a specialized binary tree. There are two kinds of heaps: *max* heaps and *min* heaps. In a max heap, any parent node *must* have a value that is greater than or equal to the values of its children. For a min heap, any parent node *must* have a value that is less than or equal to the values of its children. The heap is typically represented as an *array*, in which for any index k, the index of its left child is $2k$ and the index of its right child is $2k+1$. It's easy to see then that the parent index of any index k should be $\lfloor \frac{k}{2} \rfloor$.

Heapsort, as you may imagine, utilizes a heap to sort elements. Heapsort sorts its elements in two steps. The first step is involves creating the heap. This means taking an array, and *fixing* it such that it obeys the constraints of a max or min heap. For our purposes, the heap will be a max heap. The second step entails repeatedly "removing" the largest element in the heap and "placing" it into the end of the sorted array. This means that the root (the first element of the array) is always the smallest element. Since Heapsort is an in-place algorithm, we cannot simply remove an element from the heap and place it into the sorted array; the heap and the sorted array are the same container. What we can do is indicate where the end of the heap is, which also indicates where each successive largest element should go. Each time the largest element is selected, the heap has to be fixed. In the following Python code for Heapsort, you will notice a lot of indices are shifted down by 1. Why is this? Recall how indices of children are computed. The formula of the left child of k being $2k$ only works assuming 1-based indexing. We, in

Computer Science, use 0-based indexing. So, we will run the algorithm assuming 1-based indexing for the Heapsort algorithm itself, subtracting 1 on each array index access to account for 0-based indexing.

Heap maintenance in Python

```python
def max_child(arr, first, last):
    left = 2 * first
    right = left + 1
    if right <= last and arr[right - 1] > arr[left - 1]:
        return right
    return left

def fix_heap(arr, first, last):
    found = False
    parent = first
    great = max_child(arr, parent, last)

    while parent <= last // 2 and not found:
        if arr[parent - 1] < arr[great - 1]:
            arr[parent - 1], arr[great - 1] = arr[great - 1], arr[parent - 1]
            parent = great
            great = max_child(arr, parent, last)
        else:
            found = True
```

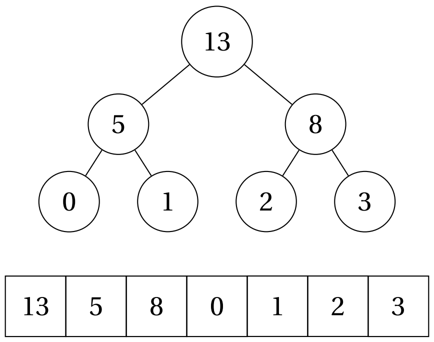

Figure 1: A max heap and its array representation.

Heapsort in Python

```python
def build_heap(arr, first, last):
    for parent in range(last // 2, first - 1, -1):
        fix_heap(arr, parent, last)

def heap_sort(arr):
    first = 1
    last = len(arr)
    build_heap(arr, first, last)
    for leaf in range(last, first, -1):
        arr[first - 1], arr[leaf - 1] = arr[leaf - 1], arr[
    first - 1]
        fix_heap(arr, first, leaf - 1)
```

6 Your Task

Die Narrheit hat ein großes Zelt; Es lagert bei ihr alle Welt, Zumal wer Macht hat und viel Geld.

—Sebastian Brant, *Das Narrenschiff*

For this assignment you have 3 tasks:

1. Implement Bubble Sort, Shell Sort, Quicksort, and Heapsort using the provided Python pseudocode in **C**. The interface for these sorts will be given as the header files `bubble.h`, `shell.h`, `quick.h`, and `heap.h`. You are not allowed to modify these files.
2. Implement a test harness for your implemented sorting algorithms. In your test harness, you will creating an array of random elements and testing each of the sorts. Your test harness *must* be in the file `sorting.c`.
3. Gather statistics about each sort and its performance. The statistics you will gather are the *size* of the array, the number of *moves* required, and the number of *comparisons* required. Note: a comparison is counted only when two array elements are compared.

7 Specifics

Vielleicht sind alle Drachen unseres Lebens Prinzessinnen, die nur darauf warten uns einmal schön und mutig zu sehen. Vielleicht ist alles Schreckliche im Grunde das Hilflose, das von uns Hilfe will.

—Rainer Maria Rilke

The following subsections cover the requirements of your test harness. It is crucial that you follow the instructions.

7.1 Command-line Options

Your test harness must support any combination of the following command-line options:

- `-a` : Employs *all* sorting algorithms.
- `-b` : Enables Bubble Sort.
- `-s` : Enables Shell Sort.
- `-q` : Enables Quicksort.
- `-h` : Enables Heapsort.
- `-r seed` : Set the random seed to `seed`. The *default* seed should be 7092016.
- `-n size` : Set the array size to `size`. The *default* size should be 100.
- `-p elements` : Print out `elements` number of elements from the array. The *default* number of elements to print out should be 100. If the size of the array is less than the specified number of elements to print, print out the entire array and nothing more.

It is important to read this *carefully*. None of these options are *exclusive* of any other (you may specify any number of them, including *zero*). The most natural data structure for this problem is a *set*.

7.2 Sets

For this assignment, you are required to implement simple *sets* using bits to track which command-line options are specified when your program is run. The number of bits will be small, but in subsequent assignments you will be implementing sets for an *arbitrary* number of bits. Your set will be initialized using an unsigned int of a size equivalent to the number of bits as shown below.

```c
typedef uint32_t Set;
```

For manipulating the bits in a set, we use bit-wise operators. These operators as the name suggests will perform an operation on every bit in a number. The following are the six bit-wise operators specified in **C**:

&	bit-wise AND	Performs the AND operation on every bit of two numbers.
|	bit-wise OR	Performs the OR operation on every bit of two numbers.
~	bit-wise NOT	Inverts all bits in the given number.
^	bit-wise XOR	Performs the exclusive-OR operation on every bit of two numbers.
<<	left shift	Shifts bits in a number to the left by a specified number of bits.
>>	right shift	Shifts bits in a number to the right by a specified number of bits.

Recall that the basic set operations are: *membership*, *union*, *intersection* and *negation*. For this assignment you will be implementing functions for these operations and a few more helper functions. Using these functions, you will set (make the bit 1) or clear (make the bit 0) bits in the `Set` depending on the command-line options read by `getopt()`. You can then check the states of all the bits (the members) of the `Set` using a single `for` loop and execute the corresponding sort. Note: you will not use all of the functions, but we will check their presence in `set.c`. Again, you *must* use sets to track which command-line options are specified when running your program.

7.2.1 `Set set_empty(void)`

This function is used to return an empty set. In this context, an empty set would be a set in which all bits are equal to 0.

7.2.2 `bool set_member(Set s, uint8_t x)`

$$x \in s \iff x \text{ is a member of set } s$$

This function returns a `bool` indicating the presence of the given value `x` in the set `s`. You will use the bit-wise AND operator to determine set membership. The first operand for the AND operation is the set `s`. The second operand is the value obtained by left shifting 1 `x` number of times. If the result of the AND operation is a non-zero value, then `x` is a member of `s` and `true` is returned to indicate this. Return `false` if the result of the AND operation is 0.

7.2.3 `Set set_insert(Set s, uint8_t x)`

This function returns a set with the bit corresponding to `x` equal to 1. Here, the bit is set using the bit-wise OR operator. The first operand for the OR operation is the set `s`. The second operand is value obtained by left shifting 1 by `x` number of bits.

7.2.4 `Set set_remove(Set s, uint8_t x)`

Similar to insertion, removing a member from the set means to clear the bit corresponding to `x`. Here, the bit is cleared using the bit-wise AND operator. The first operand for the AND operation is the set `s`. The second operand is a negation of the number 1 left shifted to the same position that `x` would occupy in the set. This means that the second operand would be all 1s except for the bit at `x`'s position. The function returns set `s` after removing `x`.

7.2.5 `Set set_intersect(Set s, Set t)`

$$s \cap t = \{x | x \in s \land x \in t\}$$

An intersection of two sets is a collection of elements that are common to both sets. Here, to calculate the intersection of the two sets, `s` and `t` we need to use the AND operator. Only the bits corresponding to members in both `s` and `t` need to be equal to 1 and the new set is returned by the function.

7.2.6 `Set set_union(Set s, Set t)`

$$s \cup t = \{x | x \in s \lor x \in t\}$$

A union of two sets is a collection of all elements in both sets. Here, to calculate the union of the two sets, `s` and `t` we need to use the OR operator. The bits corresponding to members in `s` or `t` equal to 1 therefore are in the new set is returned by the function.

7.2.7 `Set set_complement(Set s)`

This function is used to return the complement of a given set. By complement we mean that all bits in the set are flipped using the NOT operator.

7.2.8 `Set set_difference(Set s, Set t)`

The difference of two sets refers to the elements of set `s` which are not in set `t`. In other words, it refers to the members of set `s` that are unique to set `s`. The difference is calculated using the AND operator where the two operands are set `s` and the negation of set `t`.

This function can be used to find the complement of a given set as well, in which case the first operand would be the universal set $\mathbb{U}$ and the second operand would be the set you want to complement as shown below.

$$\overline{s} = \{x | x \notin s\} = \mathbb{U} - s$$

7.3 Testing

- You will test each of the sorts specified by command-line option by sorting an array of pseudorandom numbers generated with `random()`. Each of your sorts should sort the *same* pseudorandom array. Hint: make use of `srandom()`.
- The pseudorandom numbers generated by `random()` should be *bit-masked* to fit in *30* bits. Hint: use bit-wise AND.
- Your test harness *must* be able to test your sorts with array sizes *up to the memory limit of the computer*. That means that you will need to dynamically allocate the array.
- Your program should have no *memory leaks*. Make sure you `free()` before exiting. `valgrind` should pass cleanly with any combination of the specified command-line options.
- Your algorithms *must* correctly sort. Any algorithm that does not sort correctly will receive a *zero*.

A large part of this assignment is understanding and comparing the performance of various sorting algorithms. You essentially conducting an experiment. As stated in §6, you *must* collect the following statistics on each algorithm:

- The *size* of the array,
- The number of *moves* required (each time you transfer an element in the array, that counts), and
- The number of *comparisons* required (comparisons *only* count for *elements*, not for logic).

7.4 Output

The output your test harness produces *must* be formatted like in the following examples:

```
$ ./sorting -bq -n 1000 -p 0
Bubble Sort
1000 elements, 733833 moves, 498972 compares
Quick Sort
1000 elements, 6975 moves, 14190 compares
$ ./sorting -bq -n 15 -r 2021
Bubble Sort
15 elements, 228 moves, 105 compares
     45003895     46620555    199644728    203770850    218081181
    230022357    273593510    314322227    377988577    458735007
    553822818    570456718    653251166    802708864    890975627
Quick Sort
15 elements, 60 moves, 98 compares
     45003895     46620555    199644728    203770850    218081181
    230022357    273593510    314322227    377988577    458735007
    553822818    570456718    653251166    802708864    890975627
```

For each sort that was specified, print its name, the statistics for the run, then the specified number of array elements to print. The array elements should be printed out in a table with 5 columns. Each array element should be printed using the following `printf()` statement:

```c
printf("%13" PRIu32); // Include <inttypes.h> for PRIu32.
```

Pre-lab Part 4

1. Explain how you plan on keeping track of the number of moves and comparisons since each sort will reside within its own file.

8 Deliverables

Dr. Long, put down the Rilke and step away from the computer.

—Michael Olson

You will need to turn in:

1. Your program *must* have the following source and header files:
    - Each sorting method will have its own pair of header file and source file.
        - `bubble.h` specifies the interface to `bubble.c`.
        - `bubble.c` implements Bubble Sort.

    - `gaps.h` contains the Pratt gap sequence for Shell Sort.
    - `shell.h` specifies the interface to `shell.c`.
    - `shell.c` implements Shell Sort.
    - `quick.h` specifies the interface to `quick.c`.
    - `quick.c` implements Quicksort.
    - `stack.h` specifies the interface to the stack ADT.
    - `stack.c` implements the stack ADT.
    - `heap.h` specifies the interface to `heap.c`.
    - `heap.c` implements Heap Sort.
    - `set.h` specifies the interface to the set ADT.
    - `set.c` implements the set ADT.
  - `sorting.c` contains `main()` and *may* contain any other functions necessary to complete the assignment.

   You can have other source and header files, but *do not try to be overly clever*. The header files for each of the sorts are provided to you. Each sort function takes the array of `uint32_t`s to sort as the first parameter and the length of the array as the second parameter.

2. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. Typing `make` must build your program and `./sorting` alone as well as flags must run your program.

   - `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
   - `CC=clang` must be specified.
   - `make clean` must remove all files that are compiler generated.
   - `make` should build your program, as should `make all`.
   - Your program executable *must* be named `sorting`.

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

4. `README.md`: This *must* be in *markdown*. This must describe how to use your program and `Makefile`.

5. `DESIGN.pdf`: This *must* be a PDF. The design document should contain answers to the pre-lab questions at the beginning and describe your design for your program with enough detail 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 pseudo-code. You *must* push the `DESIGN.pdf` before you push *any* code.

6. `WRITEUP.pdf`: This document *must* be a PDF. The writeup must include the following:

   - Identify the respective time complexity for each sort and include what you have to say about the constant.
   - What you learned from the different sorting algorithms.
   - How you experimented with the sorts.
   - Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in reverse order, arrays with a small number of elements, and arrays with a large number of elements.
   - Analysis of the graphs you produce.

9 Submission

Daß Gott ohn Arbeit Lohn verspricht, Verlaß dich darauf und backe nicht Und wart, bis dir 'ne Taube gebraten Vom Himmel könnt in den Mund geraten!

—Sebastian Brant, *Das Narrenschiff*

To submit your assignment, refer back to `assignment0` for the steps on how to 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 on Canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

10 Supplemental Readings

The more you read, the more things you will know. The more that you learn, the more places you'll go.

—Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 1 §1.10
  - Chapter 3 §3.5
  - Chapter 4 §4.10–4.11
  - Chapter 5 §5.1–5.3 & 5.10
- *C in a Nutshell* by T. Crawford & P. Prinz.
  - Chapter 6 – Example 6.5
  - Chapter 7 – Recursive Functions
  - Chapter 8 – Arrays as Arguments of Functions
  - Chapter 9 – Pointers to Arrays