Assignment 3
Sorting: Putting your affairs in order

Prof. Darrell Long
CSE 13S – Spring 2021

Due: April 25th 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 lecture and in sections, 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, and two Quicksorts. The purpose of this assignment is to get you fully familiarized with each sorting algorithm. They are well-known sorts (save for the latter Quicksort). 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, call them i and j. If item j is smaller than item i, swap them. As a result, the largest element *bubbles* up to the top 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 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.

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 discrete mathematics class, CSE 16.

Bubble Sort in 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 earned 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 entirely 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 ($2^p 3^q$ 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

```
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]
                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) 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 applied 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 two *iterative* Quicksorts: one that utilizes a *stack* and one that utilizes a *queue*. In the implementation using a stack, two indices are pushed into the stack at a time. The first and second indices respectively represent the leftmost and rightmost indices of the array partition to sort. Similarly, in the implementation using a queue, two indices are enqueued at a time, where the first and second indices respectively 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 with a stack

```
def quick_sort_stack(arr):
    lo = 0
    hi = len(arr) - 1
    stack = []
    stack.append(lo) # Pushes to the top.
    stack.append(hi)
    while len(stack) != 0:
        hi = stack.pop() # Pops off the top.
        lo = stack.pop()
        p = partition(arr, lo, hi)
        if lo < p:
            stack.append(lo)
            stack.append(p)
        if hi > p + 1:
            stack.append(p + 1)
            stack.append(hi)
```

Quicksort in Python with a queue

```
def quick_sort_queue(arr):
    lo = 0
    hi = len(arr) - 1
    queue = []
    queue.append(lo) # Enqueues at the head.
    queue.append(hi)
    while len(queue) != 0:
        lo = queue.pop(0) # Dequeues from the tail.
        hi = queue.pop(0)
        p = partition(arr, lo, hi)
        if lo < p:
            queue.append(lo)
            queue.append(p)
        if hi > p + 1:
            queue.append(p + 1)
            queue.append(hi)
```

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 the first of your two iterative Quicksorts. An **ADT** is an *abstract data type*. With any ADT comes an interface comprised of *constructor*, *destructor*, *accessor*, and *manipulator* functions. The following subsections define 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. 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 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. 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`.

```
1 struct Stack {
2   uint32_t top;       // Index of the next empty slot.
3   uint32_t capacity;  // Number of items that can be pushed.
4   int64_t *items;     // Array of items, each with type int64_t.
5 };
```

4.1.2 `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()` and `calloc()` for *dynamic memory allocation*. Both these functions are included from `<stdlib.h>`.

```
1  Stack *stack_create(uint32_t capacity) {
2    Stack *s = (Stack *) malloc(sizeof(Stack));
3    if (s) {
4      s->top = 0;
5      s->capacity = capacity;
6      s->items = (int64_t *) calloc(capacity, sizeof(int64_t));
7      if (!s->items) {
8        free(s);
9        s = NULL;
10     }
11   }
12   return s;
13 }
```

4.1.3 `void stack_delete(Stack **s)`

The destructor function for a stack. A working destructor for a stack 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`.

```
1  void stack_delete(Stack **s) {
2    if (*s && (*s)->items) {
3      free((*s)->items);
4      free(*s);
5      *s = NULL;
6    }
7    return;
8  }
9
10 int main(void) {
11   Stack *s = stack_create();
12   stack_delete(&s);
13   assert(s == NULL);
14 }
```

4.1.4 `bool stack_empty(Stack *s)`

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 `s->top` outside of `stack.c` since it violates opacity. This accessor function returns `true` if the stack is empty and `false` otherwise.

4.1.5 `bool stack_full(Stack *s)`

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

4.1.6 `uint32_t stack_size(Stack *s)`

Returns the number of items in the stack.

4.1.7 `bool stack_push(Stack *s, int64_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.

4.1.8 `bool stack_pop(Stack *s, int64_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.

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

4.1.9 `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.

4.2 Queues

You will need to implement the *queue* ADT for the second of your two iterative Quicksorts. The following subsections define the interface for a queue. The header file containing the interface will be given to you as `queue.h`. You may not modify this file. A queue is an ADT that implements a *first-in, first-out*, or **FIFO**, policy. Consider a checkout line. Customers are *dequeued* from the front (the *head*) of the line and *enqueued* at the end (the *tail*) of the line.

4.2.1 `Queue`

The queue `struct` must be defined as follows in `queue.c`:

```
1 struct Queue {
2   uint32_t head;      // Index of the head of the queue.
3   uint32_t tail;      // Index of the tail of the queue.
4   uint32_t size;      // The number of elements in the queue.
5   uint32_t capacity;  // Capacity of the queue.
6   int64_t *items;     // Holds the items.
7 }
```

4.2.2 `Queue *queue_create(uint32_t capacity)`

The constructor for a `Queue`. The structure of this function is very similar to the stack constructor function given in §4.1.2. The `head` and `tail` of a queue should be initialized to 0. The capacity of a queue is set to the specified capacity. The capacity also indicates the number of items to allocate memory for, the items

in which are contained in the dynamically allocated array `items`. The `size` of a queue tracks the number of enqueued items. Return a pointer to the dynamically allocated `Queue`. If `malloc()` or `calloc()` fail, return a `NULL` pointer to indicate failure.

4.2.3 `void queue_delete(Queue **q)`

The destructor for a `Queue`.

4.2.4 `bool queue_empty(Queue *q)`

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

4.2.5 `bool queue_full(Queue *q)`

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

4.2.6 `uint32_t queue_size(Queue *q)`

Returns the number of items in the queue.

4.2.7 `bool enqueue(Queue *q, int64_t x)`

Enqueues an item `x` at the tail of a queue. If the queue is full prior to enqueuing `x`, return `false` to indicate failure. Otherwise, enqueue `x` and return `true` to indicate success.

4.2.8 `bool dequeue(Queue *q, int64_t *x)`

Dequeues an item from the head of a queue, passing the value of the dequeued item back through the pointer `x`. If the queue is empty prior to dequeuing the item, return `false` to indicate failure. Otherwise, dequeue the item, set the value in the memory `x` is pointing to as the dequeued item, and return `true` to indicate success.

4.2.9 `void queue_print(Queue *q)`

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

5 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, and both iterative Quicksorts based on the provided Python pseudocode in **C**. The interface for these sorts will be given as the header files `bubble.h`, `shell.h`, and `quick.h`. You are not allowed to modify these files for any reason other than gathering statistics.

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. For the iterative Quicksorts, you will additionally need to gather statistics on the maximum stack and queue sizes. You will want to create some graphs.

6 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.

6.1 Command-line Options

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

- `-a` : Enables *all* sorting algorithms.
- `-b` : Enables Bubble Sort.
- `-s` : Enables Shell Sort.
- `-q` : Enables the Quicksort that utilizes a stack.
- `-Q` : Enables the Quicksort that utilizes a queue.
- `-r seed` : Set the random seed to `seed`. The *default* seed should be 13371453.
- `-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*.

6.2 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, 758313 moves, 498465 compares
Quick Sort (Stack)
1000 elements, 7257 moves, 13643 compares
Max stack size: 22
$ ./sorting -bqQ -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 (Stack)
15 elements, 60 moves, 98 compares
Max stack size: 6
     45003895     46620555    199644728    203770850    218081181
    230022357    273593510    314322227    377988577    458735007
    553822818    570456718    653251166    802708864    890975627
Quick Sort (Queue)
15 elements, 60 moves, 98 compares
Max queue size: 8
     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 with a width of 13. You should make use of the following `printf()` statement:

```
1 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.

7 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 *both* iterative Quicksorts.
        - `stack.h` specifies the interface to the stack ADT.
        - `stack.c` implements the stack ADT.
        - `queue.h` specifies the interface to the queue ADT.
        - `queue.c` implements the queue 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`.
    - `make format` should format all your source code, including the header files.
    - 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.

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.
- How big do the stack and queue get? Does the size relate to the input? How?
- 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.

8 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 `asgn0` 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*.

9 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

用 C 編程就像給猴子電鋸一樣。