Merilyn Kuo
CSE 13S
Spring 2021

Assignment 2: A Small Numerical Library
Design Document

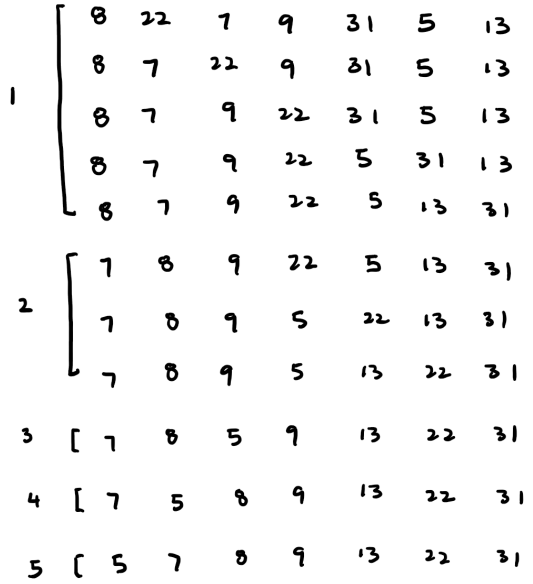
Purpose:

The purpose of this assignment is to understand different sorting algorithms, specifically Bubble Sort, Shell Sort, and Quick Sort. We also learn about big O and how to implement stacks and queues.

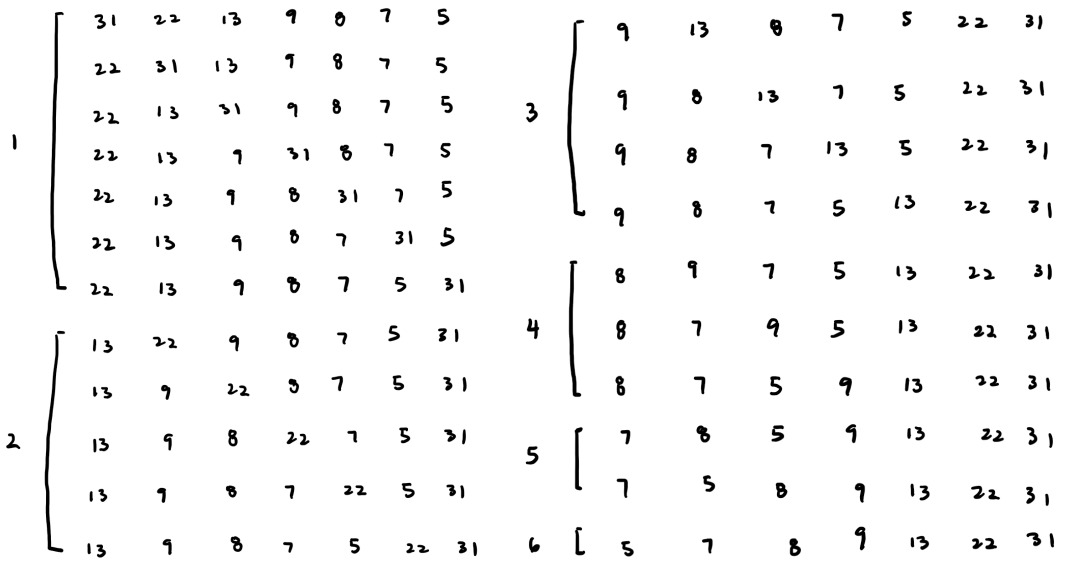
Pre-Lab Questions:

Part 1:

1. For my answer, I interpreted a "round of swapping" as a pass. To sort the numbers in ascending order using bubble sort, 5 rounds of swapping are needed as shown in the work below.



2. If the worst case scenario means the largest number of swaps needed to sort a list of numbers, from intuition I would think that the worst case scenario would be having to sort a list that is in descending order to ascending order since for each number you need to do swaps for every number. Testing this out on paper I get:



We can see that it requires the sort to do the maximum number of passes since every number will take one pass to be sorted to its correct position except for the smallest number. The smallest number gets sorted into its correct position simultaneously with the number prior.

Part 2:

1. Since the shell sorting process is repeated until the gap size becomes zero, the time complexity of the shell sort depends on how large the gap size in the beginning is and how they are decreased. According to Wikipedia, “Too few gaps slows down the passes and too many gaps produces an overhead”. Since every list of numbers is different it is hard to choose the best sequence of gaps but there are sequences that generally work well on a majority of lists. For example, Kunth’s sequence uses the formula ((3^k) - 1) / 2 and has time complexity of O(n^(3/2)).
   https://en.wikipedia.org/wiki/Shellsort#Gap_sequences
   https://www.codesdope.com/blog/article/shell-sort/

Part 3:

1. I suppose the worst case time complexity of Quicksort does not doom the Quicksort algorithm because there are ways to easily avoid the worst case scenarios by using a random position for the pivot, choosing the middle of a partition for the pivot and choosing the median of the first, middle, and last element of the partition for the pivot.
   https://en.wikipedia.org/wiki/Quicksort

Part 4:

1. To keep track of moves and comparisons, I thought of creating a header file containing global variables for moves and comparisons. Then, in each sorting algorithm's file, I can access those variables for use.

Program Implementation:

Supporting functions:

```
function swap ( dereference to x , dereference to y ):
    temp = dereference to x;
    dereference to x = dereference to y
    dereference to y = temp

Function less than (x, y):
    If x < y
       Return true
    Else
       Return false
```

Stack

Code for defining the stack structure, constructor function, and destructor function were given in the Assignment 3 PDF. The following pseudocode scripts are for the accessor and manipulator functions of stack. The functions headers are also from the Assignment 3 PDF and the pseudocode is referenced from Sahiti's section.

```
bool stack_empty(Stack *s):
    return top == 0

bool stack_full(Stack *s):
    return top == capacity

uint32_t stack_size(Stack *s):
    return top + 1

bool stack_push(Stack *s, int64_t x):
    if top == capacity
        return false
    items[top] = x
    top += 1
    return true

bool stack_pop(Stack *s, int64_t *x)
    if top == 0:
        return false
    top -= 1
```

```
    *x = items[top]
    return true

void stack_print(Stack *s):
    print(s)
```

Queue

Code for the definition of the queue structure and function headers are given in the Assignment 3 PDF. The following two constructor and destructor functions are based on the constructor and destructor functions for stack from the assignment PDF.

```
Queue *queue_create(uint32_t capacity):
    Queue *q = (Queue *) malloc(sizeof (Queue))
    if q:
        head = 0
        tail = 0
        capacity = capacity
        items = (int *) calloc(capacity, sizeof(int))
        if (items not exist):
            free(q)
            q = null
    return q;

void queue_delete(Queue **q):
    if  q && items exists:
        free(items)
        free(q)
        s = null
    return

bool queue_empty(Queue *q):
    return tail == 0

bool queue_full(Queue *q):
    return tail == capacity

uint32_t queue_size(Queue *q):
    return size
```

The pseudocode for enqueue and dequeue is referenced from Sahiti's section.

```
bool enqueue(Queue *q, int64_t x):
    if size == capacity:
        return false
    item[tail] = x
    tail += 1
    size += 1
```

```
    return true

bool dequeue(Queue *q, int64_t *x):
    if tail == 0:
        return false
    size - = 1
    *x = items[head]
    head += 1
    return true

void queue_print(Queue *q):
    print q
```

Test Harness Pseudocode:
This pseudocode will use the functions from the Set ADT provided to us. This pseudocode is based on the test harness example Eugene demonstrated in his section.

```
typedef enum sort_types { BUBBLE, SHELL, STACK, QUEUE } sort_types
seed = 13341453
size = 100
elements = 100

main(argc, **argv):
    opt = 0
    sorts = set_empty()
    void(*sorting_funcs[4])(void) = { bubble, shell, stack quicksort,
                                      queue quicksort }
    while((opt = getopt(argc, argv, “absqQr:n:p:”)) != -1):
        switch (opt):
            case ‘a’:
          sorts =
            case ‘b’:
                sorts = set_insert(sorts, BUBBLE)
            case ‘s’:
                sorts = set_insert(sorts, SHELL)
            case ‘q’:
                sorts = set_insert(sorts, STACK)
            case ‘Q’:
                sorts = set_insert(sorts, QUEUE)
            case ‘r’:
               seed = optarg
          case ‘n’:
              size = optarg
```

```
          case ‘p’:
              elements = optarg
      default:
                print error message
for each in sort_types:
    if (set_member(sorts, i):
        run sorting_functions[i]()
        print statistics
        print elements?
return 0
```

Process and Changes to Design:

I added a function to my helper file called less_than. It checks if one integer is less than another and returns true or false.

Credits and Resources Used:

Python Pseudocode for sorts was from the Assignment PDF
https://www.gnu.org/software/libc/manual/html_node/Using-Getopt.html