Natalie Valett, nvalett
CSE13S
Prof. Darrell Long
April 19, 2021

Asgn3 Design:

PRELAB Q's:

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?

6

Round	Arr	Swapped (keep going?)
0	8,22,7,9,31,5,13	yes
1	8,7,9,22,5,13,31	yes
2	7,8,9,5,13,22,31	yes
3	7,8,5,9,13,22,31	yes
4	7,5,8,9,13,22,31	yes
5	5,7,8,9,13,22,31	yes
6	5,7,8,9,13,22,31	No (stop)

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.

The worst case time complexity is O(n^2). So if the size of the list is 5, the most comparisons executed could be 25.

3. 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 gapsize? Cite any sources you used.

The time complexity is dependent on the sequence of gaps because not only does the algorithm take more time the more gaps there are to compute, but the magnitude of each gap is also iterated through. This means that the quantity of gaps and the magnitude of each gap contributes to the time complexity, so depending on the size of gaps to be used and the magnitude of each gap, the time complexity could vary

widely. One could improve the time complexity by providing either less gaps or gaps of smaller magnitudes.

4. Quicksort, with a worse case time complexity of O(n2), 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.

Because of quicksort's methods of breaking the larger array into many smaller sub-arrays, it's likely that many of these random sub-arrays will happen to be sorted or have many sorted sections. The worst case scenario of time complexity O(n^2) would only occur if every random sub-array were independently sorted in the opposite manner from the function's goal (ascending vs descending), which is extremely unlikely. In most cases, the time complexity will be much less drastic because the maximum possible amount of moves and comparisons are rarely needed.

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

I expect the best way to do this is to make moves and comparisons EXTERN variables in order to access them from anywhere in the script. I would have to reset them at the start of each sort but that's be a small adjustment.

Bubble Sort:

Bubble sort iterates through the array and for each element, compares it to the one behind it. If the one behind it is bigger, the 2 elements get swapped. This loop happens iteratively, each time evaluating 1 less element than it did before as the highest values begin to accumulate at the end of the array.

Psuedocode (adopted from given python psuedocode):

```
bubble_sort(arr, n):
    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
```

Shell Sort:

Shell sort is given an array of gaps, and for each gap in that list it evaluates the elements in the array that are (gap) apart from each other. For each pair evaluated, if the lower index one is greater than the higher, they're swapped.

Psuedocode (adopted from given python psuedocode):

```
shell_sort(arr, n):
    For gap in gaps:
        For i in range(gap, n):
            j = i
            temp = arr[i]
            While (j >= gap) and (temp < arr[j - gap]):
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
```

<u>Quick Sort:</u>

Partitions the array by a pivot element, attempts to put all elements greater than the pivot value on the right side of the pivot index, and all less than the pivot on the left. Then the quicksort function is iteratively called on each sub-array in order to further sort the low values out from the high.

Psuedocode (adopted from given python psuedocode):

```
partition(arr, lo, hi):
    Pivot = arr[lo + ((hi - lo) // 2)]
    i = lo - 1
    j = hi + 1
    while i < j:
        i += 1
        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
quick_sort_stack(arr):
    Lo = 0
    Hi = len(arr) - 1
    Stack = []
    stack.append(lo)
    stack.append(hi)
```

```
        While len(stack) != 0:
                Hi = stack.pop()
                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)

quick_sort_queue(arr):
        Lo = 0
        Hi = len(arr) - 1
        queue = []
        stack.append(lo)
        stack.append(hi)
        While len(stack) != 0:
                Lo = stack.pop(0)
                Hi = stack.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)
```

Stack:

Psuedocode/ACTUAL code that was given:

```
Struct Stack {
        uint32_t top;
        uint32_t capacity;
        uint64_t *items;
}

Stack *stack_create(uint32 capacity) {
        Stack *s = (Stack *) malloc(sizeof(Stack));
        if (s) {
                s->top = 0;
```

```
                s->capacity = capacity;
                s->items = (int64_t *) calloc(capacity, sizeof(int64_t));
                If (!s->items) {
                        free(s)
                        s = NULL;
                }
        }
        return s;
}

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

// Example of stack creation
int main(void) {
        Stack *s = stack_create();
        stack_delete(&s);
        assert(s == NULL);
}

//PSUEDOCODE for other functions
Bool stack_empty(Stack *s) {
        If s.top == 0:
                Return true
        else:
                Return false
}

Bool stack_full(Stack *s) {
        If stack.top == stack.capacity:
                Return true
        Else:
                Return false
}
```

```
Bool stack_size(Stack *s) {

}

Bool stack_push(Stack *s, int64_t x) {
        //dynamically allocate space if stack is full
        If stack_full(s) {
                s.capacity = s.capacity * 2
                S.items = //use realloc to reallocate space in memory
        }
        S.items[s.top] = x
        S.top ++
}

Bool stack_pop(Stack *s, int64_t *x) {
        //if stack's empty, return false
        If stack_empty(s):
                Return false

        s.top -= 1
        //set *x to point to item popped
        *x = s.items[s.top]
        Return true
}

void stack_print(Stack *s) {
        For i in (0, s.size):
                print(s.items[i])
}
```

Queue:

Psuedocode/ ACTUAL given code:

```
struct Queue {
        uint32_t head;
        uint32_t tail;
        uint32_t size; //tracks num of elements in items
        uint32_t capacity;
        int64_t *items;
}

Queue *queue_create(uint32_t capacity) {
```

```
        Queue *q = (Queue  *) malloc(sizeof(Queue));
        if (q) {
                q->head = 0;
                q->tail = 0;
                s->capacity = capacity;
                s->items = (int64_t *) calloc(capacity, sizeof(int64_t));
                If (!s->items) {
                        free(s)
                        s = NULL;
                }
        }
        return s;
}

//actual psuedocode now:
void queue_delete(Queue **q) {
        if (q && items):
                free(items)
                free(q)
                *q = NULL
}


bool queue_empty(Queue *q) {
        Return (head == tail)
}

bool queue_full(Queue *q) {
        If head+1 == tail:
                Return true
        Else
                Return false
}

uint32_t queue_size(Queue *q) {
        Return q.size
}

bool enqueue(Queue *q, int64_t x) {
        if queue_full(q):
```

```
            Return false
        q.items[q.head] = x
        q.head++
        Return true
}

bool dequeue(Queue *q, int64_t *x) {
        If queue_empty(q):
            Return false
        *i = q.items[q.tail]
        q.tail--
}

void queue_print(Queue *q) {
        For i in (0, q.size):
            Print q.items[i]
}
```

Test Harness:

Interfaces w user input to perform correct sorts with specified preferences of seed, array size and elements to print

Should have 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.

Psuedocode:

```
ints bubble, shell, quick_stack, quick_queue = 0
int seed = 13371453
int size = 100
int elements = 100
while (user input still exists) { //uses getopt()
```

```
        // can use switch here or if statements
        If user input == ‘a’
                bubble = 1
                shell = 1
                quick_stack = 1
                quick_queue = 1
        If user input == ‘b’
                bubble = 1
        If user input == ‘s’
                shell = 1
        If user input == ‘q’
                quick_stack = 1
        If user input == ‘Q’
                quick_queue = 1
        If user_input == ‘r’
                seed = optarg // optarg = value of argument passed for an input
        If user_input == ‘n’
                size = optarg // optarg = value of argument passed for an input
        If user_input == ‘p’
                elements = optarg // optarg = value of argument passed for an input
        If (user_input is invalid)
                Print error message
                Print out program instructions
}
if bubble or shell or quick_stack or quick_queue:
        //generate array of random elements of length (size) to sort
        arr[size];
        for i in range(0, size- 1):
                arr[i] = random()
Else:
        Print “please select a sort to perform”
        Print program instructions
if bubble == 1
        Print “Bubble Sort”
        sorted = bubble_sort(arr, size)
        print(elements, moves, compares)
        For i in range(0, elements):
                print(sorted[i])
if shell == 1
        Print “Shell Sort”
```

```
        sorted = shell_sort(arr, size)
        print(elements, moves, compares)
        For i in range(0, elements):
                print(sorted[i])
if quick_stack == 1
        Print “Quick Sort (Stack)”
        sorted = quick_sort_stack(arr)
        print(elements, moves, compares)
        print(“Max stack size: “ (stack_size))
        For i in range(0, elements):
                print(sorted[i])
if quick_queue == 1
        Print “Quick Sort (Queue)”
        sorted = quick_sort_queue(arr)
        print(elements, moves, compares)
        print(“Max queue size: “ (queue_size))
        For i in range(0, elements):
                print(sorted[i])
```