Bill Zhang
jzhan411@ucsc.edu
4/21/2021

CSE13S Spring 2021
Assignment 3: Sorting: Putting your affairs in order
Design Document

This lab is about making a small library that will contain all the sorting algorithms.

Pre Lab Questions:

1. 10 rounds of swapping is required to sort this list in ascending order using bubble sort.
2. 21 comparisons
3. algorithm - Time complexity for Shell sort? - Stack Overflow if all the even positioned elements are greater than the median, then the odd and even elements will not be compared until the gap is 1, leading to a waste of time sorting when the gap is not 1.
    a. sorting - Worst scenario for shell sort: Θ(N^3/2) or O((NlogN)^2)? - Stack Overflow Time case depends on gap. Different gaps give different times.
    b. python - Why the time complexity for shell sort is nlogn in my data? - Stack Overflow gap sequence 1,4,13,40,121 give a time sequence of O(n^1.5). 1,2,3,4 gives O(nlog^2(n)).
    c. Shellsort - Wikipedia Different gap sizes can give different worst case time complexities.
    d. algorithm - Fastest gap sequence for shell sort? - Stack Overflow best gap sequences are found experimentally. Best one currently seems to be Marcin Ciura's sequence: 1,4,9,24,57
4. Without consulting outside sources, I have 2 guess as to why Quicksort is faster that other sorting algorithms.
    a. Quicksort does not allocate additional memory for sorting
        i. Could save time by not writing and reading from memory
    b. O(n^2) time is only for the worst case. Chances are most lists are not the worst case and quicksort will be faster than other sorting algorithms for most lists.
    c. (39) Why is quick sort named 'quick' even when it has O(n2) complexity in the worst case? - Quora says that quicksort is faster than other sorting algorithms because it is very easy to avoid quicksort's worst-case run time by picking pivot points at random.
    d. algorithm - Why is quicksort better than mergesort? - Stack Overflow says that the average runtime is closer to n log n. Also requires not alot of additional space and exhibits good cache locality.
5. Each sort will have 2 variable listing the moves and comparisons and will have methods returning those.

Top Level Diagram

Bubble.c

```
Void bubble_sort(Array 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
    Return
```

Shell.c

```
Void shell_sort(Array 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
    Return
```

Quick.c

```
Int64_t partition(Array arr, int low, int hi)
    pivot=arr[lo+ ((hi-lo) // 2)];
    i=lo- 1
    j=hi+ 1
    whilei<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
```

```
Void quick_sort_stack(Array arr)
        Lo = 0
        Hi = len(arr) -1
        Stack = []
        stack.append(lo)
        stack.append(hi)
        While (len(stack) != 0)
                Hi = stack.pop
                Lo = stack.pop
                If lo < p
                        stack.append(lo)
                        Stack.append{p}
                If hi > p + 1
                        stack.append(p+1)
                        stack.append(hi)
        Return

Void quick_sort_queue(Array arr)
        Lo = 0
        Hi = len(arr)-1
        Queue = []
        queue.append(lo)
        queue.append(hi)
        whilelen(queue) != 0
                lo=queue.pop(0)
                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)
```

Stack.c

```
Stack *stack_create(capacity)
        Stack *s = (stack *) malloc(sizeof(Stack))
        If (S){
                s->head = 0
                s->tail = 0
                s->capacity = capacity
                s->items = (int *) calloc(capacity, sizeof(int)))
```

```
		If (!s>items)
			free(s)
			S = null
	Return s

Void stack_delete(Stack **s)
	If (*s && (*s->items)
		free((*s)->items)
		free(*s)
	return
Int main(void)
	Stack *s = stack_create()
	stack_delete(&s)
	assert(s == NULL)

Bool stack_emtpy(Stack *s)
	If *items length == 0
		Return true
	Return false

Bool stack_full(Stack *s)
	If (*items length == capacity)
		Return true
	Return false

Uint32_t stack_size(Stack *s)
	Return length of *items

Bool stack_push(Stack *s, int 64_t x)
	If stack_full
		Return false
	add x to *items
	Top ++
	Return true

Bool stack_pop(Stack *s, int64_t *x)
	If stack_empty
		Return false
	Top --
	*x=s->items[s->top]
	Return True

Void stack_print(Stack *s)
	For int x, x < *items lengh, x++
```

```
        Print item at x position in *items
```

Queue.c

```
Queue *queue_create(capacity)
    Queue *s = (Queue *) malloc(sizeof(Queue))
    If (S){
        s->top = 0
        s->capacity = capacity
        s->items = (int *) calloc(capacity, sizeof(int)))
        If (!s>items)
            free(s)
            S = null
    Return s

Void queue_delete(Queue **q)
    If (*q && (*q->items)
        free((*q)->items)
        free(*q)
    return
Bool queue_empty(Queue **q)
    If length *items = 0
        Return true
    Return false

Bool queue_full(Queue **q)
    If (*items length == capacity)
        Return true
    Return false
Uint32_t queue_size(Queue **q)
    Return len *items
Bool enqueue(Queue *q, int64_t x)
    If queue_full
        Return false
    *items[tail] = x
    Tail++
    If tail > capacity - 1
        Tail = 0
    Return true
Bool dequeue(Queue *q. Int64_t *x)
    If queue_empty
        Return false
    *x=s->items[s->top];
    *items[head] = null
```

```
        Head++
        If head > capacity - 1
                Head  = 0
        Return true

Void queue_print(Queue *q)
        For int x = 0 ; x < length *items; x++
                Printf item at x position of *items
```

Sorting.c

```
Main(arguments)
        Variable opt
        Variable boolean bubble
        Variable boolean shell
        Variable boolean quick
        Variable boolean quickq
        Int Mainseed = 13371453
        Int Mainsize = 100
        Int mainelements
        Numtoprint = 100
        While (opt = getopt(argc, argv, option) != 0)

                switch (opt)
                Case a
                        Bubble = true
                        Shell = true
                        Quick = true
                        Quickq = true
                Case b
                        Bubble = true
                Case s
                        Shell = true
                Case q
                        Quick = true
                Case Q
                        Quickq = true
                Case r
                        Mainseed = seed
                Case n
                        Mainsize = size
                Case p
```

```
                Mainelements = true
                Numtoprint =  = elements


    If Bubble
        Do bubblesort tests
    If Shell
        Do shellsort tests
    If Quick
        Do quicksort tests with stacks
    If quickq
        Do quicksort tests with queue
    If Main elements
        If numtoprint > size of array
            Print out all elements of array
        Print out first numtoprint elements of array
```

Design Progress:

- First design was made on 4/21