William Santosa
wsantosa@ucsc.edu
15 February 2021

CSE13S Winter 2021
Assignment 5 : Putting Your Affairs in Order
Design Document

PRELAB

Part 1

1. After completing the bubble sort by hand, I found out that it only took 5 rounds of swapping to sort the numbers in the array. However, I expected it to take 6 rounds of swapping as there are 7 elements in the array. Each iteration through the array would make the last item in the array sorted, and after 6 rounds, all 7 of them should be sorted since there would be one space left for the last item and the others are already sorted.
2. I found out on https://www.bigocheatsheet.com/ that the worst case time complexity of bubble sort is O(N^2), which didn't seem to match up with the amount of rounds it took to sort the array in question 1. That's when I remembered that it's an approximation, and the more accurate bubble sort time complexity is N^2 - N, where N is removed as it becomes irrelevant in larger arrays.
3. To order the array from largest to smallest, I would revise the algorithm by iterating from the last element to the first and swapping only if the element to the right is greater than the left.

Part 2

1. The worst time complexity for shell sort depends on the sequence of gaps because it directly influences the amount of comparisons that the sort has to make. Taking that into consideration, there are ways to improve the complexity by using better sequences. For example, Sedgewick's sequence of 1, 5, 19, 41, 109, and so on would improve it as it has a time complexity of O(N^(4/3)) as opposed to Pratt's sequence of Nlog^2(N).

Part 3

1. It is true that Quicksort's worst time complexity is O(N^2), which you would think makes it a terrible sort. However, in reality, this scenario rarely occurs. According to medium.com, Quicksort's average time complexity is actually closer to Nlog(N), and oftentimes lower than that. As such, Quicksort has been the go-to sort using comparisons for many programmers and not doomed as a sort.

Part 4

1. To keep track of all the moves and comparisons done for each sort, I plan to create two variables in sorting.c to keep track of the moves and comparisons. After each comparison is done and the values are printed, it'll set itself back to 0 for the next sort. Other than that, I can also implement two variables in each sort that will do the exact

same thing as the aforementioned method but instead increment itself within the sort file. It can then be accessed using accessor functions that I would have to implement.

SOURCES

1. https://www.bigocheatsheet.com/
2. https://en.wikipedia.org/wiki/Shellsort
3. https://www.youtube.com/watch?v=HDQd6_0TJIE&ab_channel=CSRocks
4. https://medium.com/better-programming/quicksort-explained-in-5-minutes-d32cf430a592

PAPER DRAFT/WORK

William Santosa
2/11/21
CSE 135

Prelab Part 1

1. {8, 22, 7, 9, 31, 5, 13} 0th Pass
{8, 7, 9, 22, 5, 13, 31} ~~1st~~ 1st Pass
{7, 8, 9, 5, 13, 22, 31} 2nd Pass
{7, 8, 5, 9, 13, 22, 31} 3rd Pass
{7, 5, 8, 9, 13, 22, 31} 4th Pass
{5, 7, 8, 9, 13, 22, 31} 5th Pass

I expected 6 rounds but it only requires 5 passes. I thought it was 6 because 7 elements are (7 - 1) = 6 rounds.

2. {4, 3, 2, 1} 0
{3, 4, 2, 1} 1
{3, 2, 4, 1} 2
{3, 2, 1, 4} 3 1 round
{2, 3, 1, 4} 4
{2, 1, 3, 4} 5 2 round
{1, 2, 3, 4} 6 3 round

{5, 4, 3, 2, 1} 0
{4, 5, 3, 2, 1} 1
{4, 3, 5, 2, 1} 2
{4, 3, 2, 5, 1} 3
{4, 3, 2, 1, 5} 4
{3, 4, 2, 1, 5} 5
{3, ~~4~~ 2, 4, 1, 5} 6
{2, 3, 1, 4, 5} 7
{2, 3, 1, 4, 5} 8
{2, ~~3~~ 1, 3, 4, 5} 9
{1, 2, 3, 4, 5} 10

From the lectures it says that the time complexity is $O(n^2)$, however, it doesn't seem to be completely accurate. ~~[illegible]~~ We can expect $N^2 - N$ comparisons in the worst case, where N = number of elements in the array.

3. I would revise the algorithm by iterating from the last element to the first and swapping if the right element is greater than the left.

William Santosa
2/11/21
CSE 13S

Prelab Part 2

1. The worst time complexity for shell sort depends on the sequence of gaps because it directly influences the amount of comparisons the sort has to make.

The time complexity of this sort could be changed by using Sedgewick's sequence of 1, 5, 19, 41, 109... gaps, which has a time complexity of $O(N^{\frac{4}{3}})$ rather than Pratt's $O(N\log^2 N)$ worst case time complexity.

Prelab Part 3

1. It may be true that Quicksort's worst time complexity is $O(N^2)$, however, this scenario rarely happens. When taking into account the average time complexity of $O(N\log(N))$, Quicksort becomes one of the fastest and best sorts to use for comparisons.

Prelab Part 4

1. To keep track of moves and comparisons for each sort, I plan to have two variables in sorting.c to keep track of the moves and comparisons. Then it resets to 0. That, or

I can have two variables in each sort file that increments whenever a move or comparison is made. Then, I can use two accessor functions to obtain then print those values.

DESCRIPTION

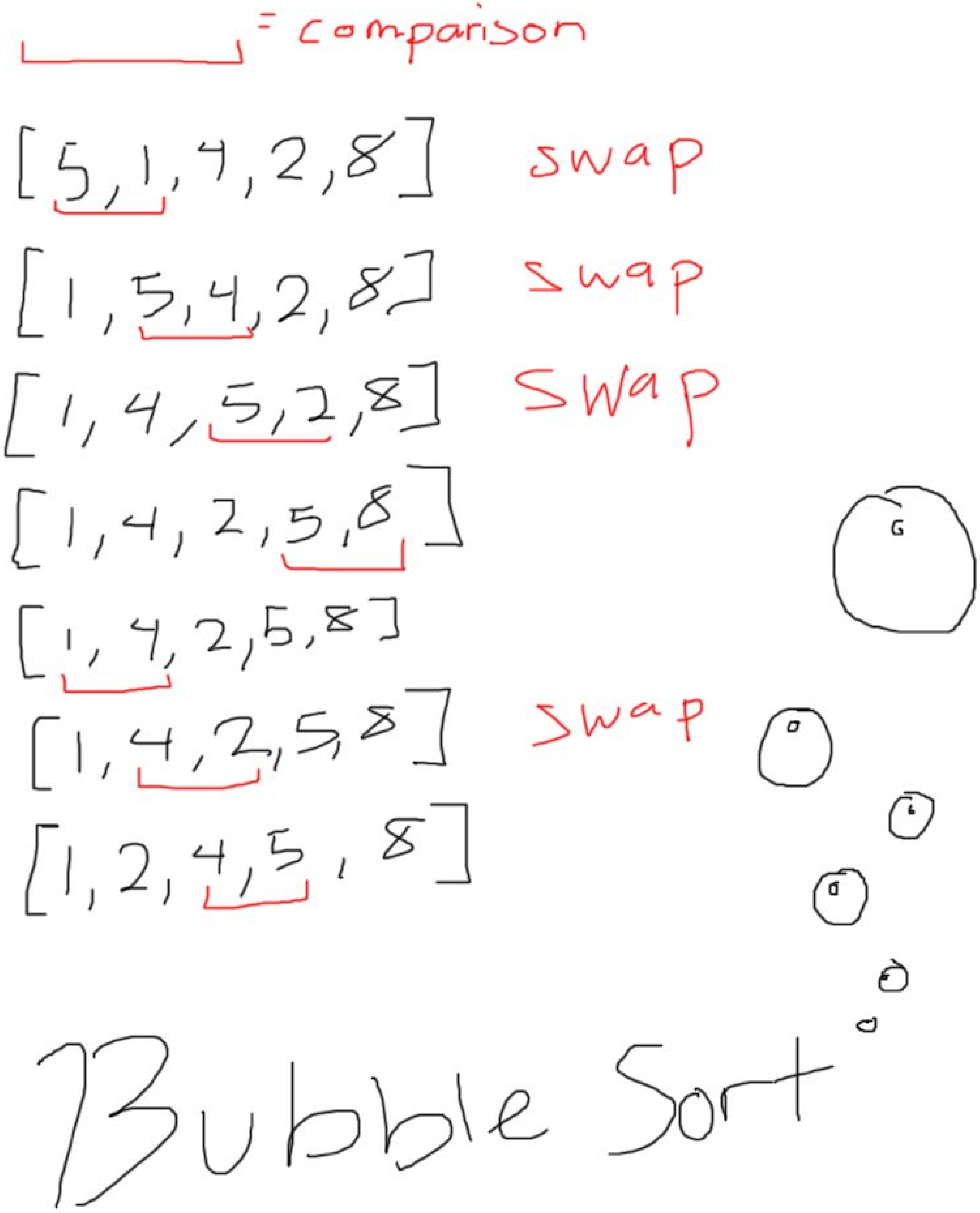
Computer scientists often have to put items into a sorted order when dealing with data. Consequently, there are a lot of different ways to do these sorts. Some of these methods include bubble sort, heap sort, quick sort, and shell sort. In this lab, we seek to implement these library functions/sorts in C and identify the strengths and weaknesses of each method.

Files:

- bubble.{c, h}
  - Specifies the interface and implementation of Bubble Sort
- gaps.h
  - Contains the Pratt gap sequence for Shell Sort
- shell.{c, h}
  - Specifies the interface and implementation of Shell Sort
- quick.{c, h}
  - Specifies the interface and implementation of Quick Sort
- stack.{c, h}
  - Specifies the interface and implementation of Stack ADT
- heap.{c, h}
  - Specifies the interface and implementation of Heap sort
- set.{c, h}
  - Specifies interface to set ADT/implements set ADT
- sorting.c
  - Contains main() and other things needed to complete the assignment
- Makefile
  - Runs program and creates an executable named sorting
- README.md
  - Information about building, running, and options of the program
- DESIGN.pdf
  - Describes purpose, covers the layout, clear description of program parts, pseudo code, and contains the pre lab questions.
- WRITEUP.pdf
  - Identifies respective time complexity, what I learned, how I experimented, graphs, and analysis of these graphs
- comparisons.h
  - Included compares and moves to return to sorting.c

EXAMPLE OF EACH SORT

Bubble Sort



Shell Sort

= comparison

[5, 1, 4, 2, 8] 5 elements

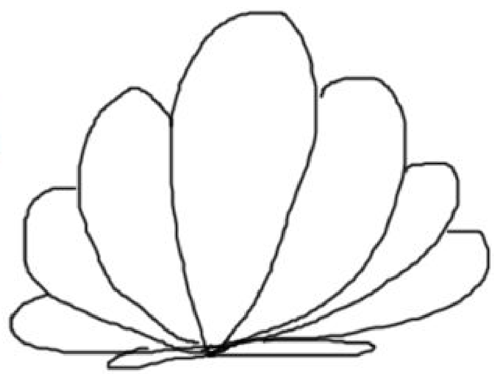
Pratt: $2^p 3^q$

[5, 1, 4, 2, 8] 4 {4, 3, 2, 1}

[5, 1, 4, 2, 8] 3

[2, 1, 4, 5, 8] 2 2

[2, 1, 4, 5, 8] 1



[1, 2, 4, 5, 8]

Shell Sort

Quick Sort

⊔ = left pointer ⊔ = right pointer

[5, 1, 4, 2, 8]

pivot (rand #)

[5, 1, 4, 2, 8]

[5, 1, 4, 2, 8]

[2, 1, 4, 5, 8]

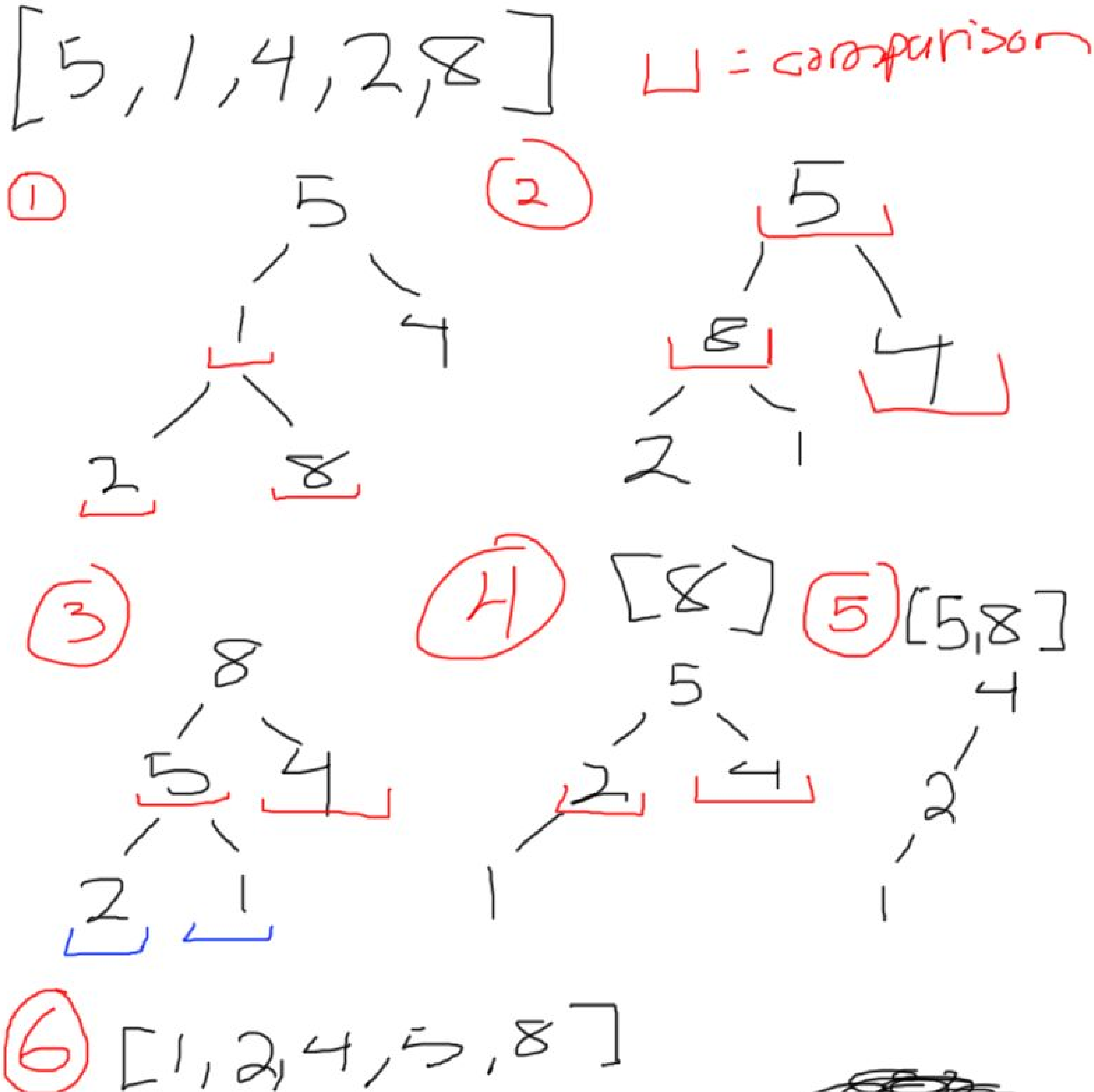
[2, 1] [4, 5, 8]

[1, 2] [4, 5, 8]

[1, 2, 4, 5, 8]

Quick Sort

<u>Heap Sort</u>



Heap Sort

TOP LEVEL DESIGN / PSEUDOCODE

Pseudocode is influenced by Eugene's lab section on 2/9/21

Stack

```
Struct Stack{
        U32 Top
        U32 Capacity
        Int *Items
}

Stack stack_create(){
        Create s pointer to (Stack *) calloc(1, sizeof(Stack))
        Top and capacity to 0
        Make a stack point toward array for items
        For loop to make all rows contain int
        Return s
}

Bool stack_empty(Stack){
        Return s -> top == 0 // Return if stack top is equivalent to 0
}

Bool stack_push(Stack, int){
        If s -> top == s -> capacity{
                Double s -> capacity
                S -> items = realloc(s -> items, capacity)
        }
        Stack -> items[s -> top] = x
        Stack -> top +=1
        Return true
}

Bool stack_push(Stack, * int){
        Stack -> top -=1
        *int = s -> items[s -> top]
        Int *arr = (int *) calloc (16, sizeof(int))
        Return true
}
```

Set

```
Typedef uint32_t Set{

}

Set set_empty(){
	Return 0
}

Set set_insert(Set s, uint8_t x){
	Return s | (1 << x %8)
}

Bool set_member(Set s, uint8_t x){
	Return s & (1 << x % 8)
}

Set set_remove(Set s, uint8_t x){
	Return s without x
}

Set set_intersect(Set s, Set t){
	Take one element of set s and find it in t
	If true, set it as element using set_insert in new set
	Repeat for all
	Return new set
}

Set set_union(Set s, Set t){
	Use OR to return a new set that merges elements in both
}

Set set_complement(Set s){
	Return set s after is XOR with all 1's
}

Set set_difference(Set s, set t){
	Iterate between all elements in set s and see if it's in t, else insert into new set
	Do the same for t and s
	Delete duplicates
	Return new set
}
```

NOTE : Will definitely be altered (Listed below)

DESIGN PROCESS / MODIFICATIONS

- Decided to store the comparisons and moves the second way I mentioned
    - Altered a little bit
        - Created another header file (comparisons.h)
        - Implemented the accessor functions and variables in each sort so I could call them back
- Had to change makefile to include format so it's easier
- Valgrind
- Added comments
- Had to change where the compares++ and moves++ made