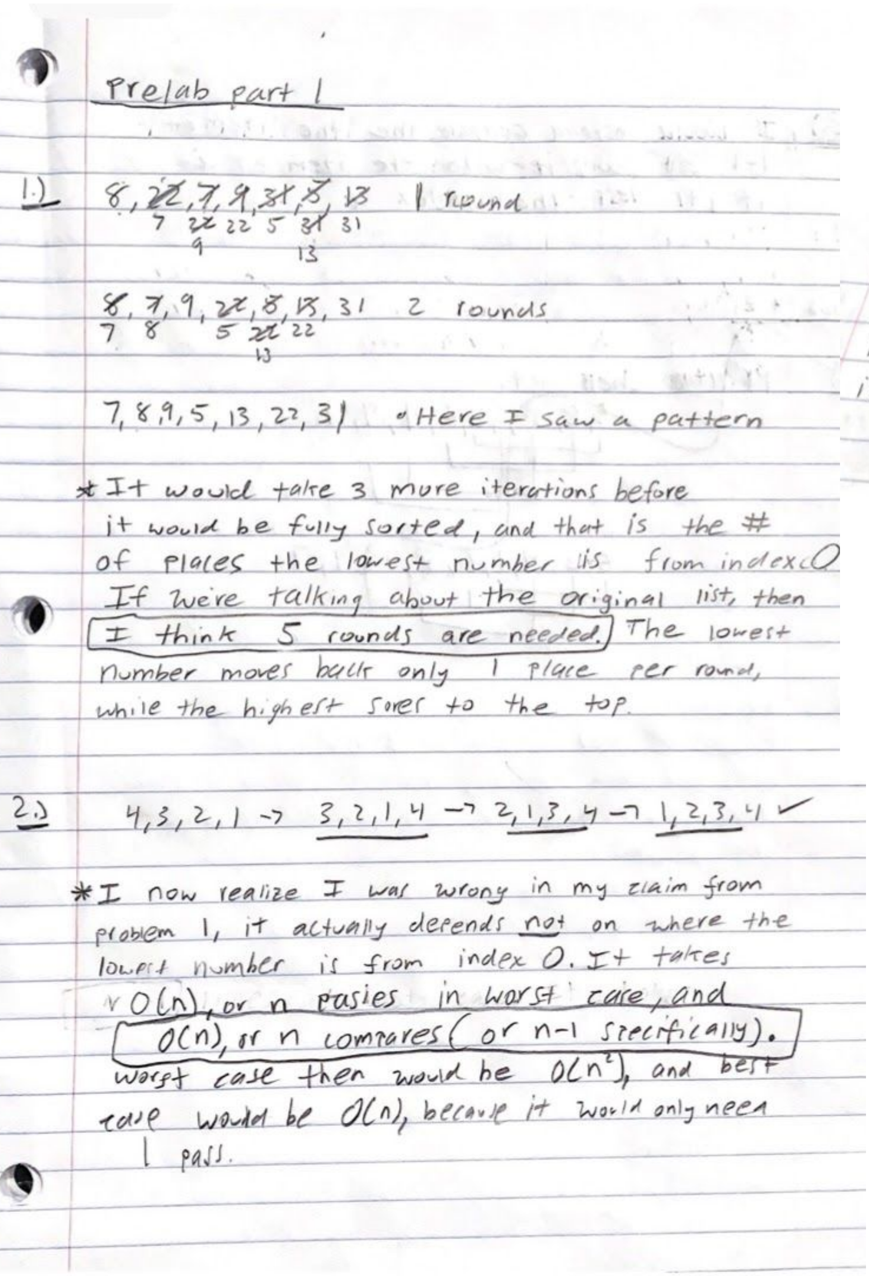
Version 1.0

Section 0: Pre-Lab

- Citations
  - https://www.youtube.com/watch?v=NYWEM7H3iYc For better understanding gap lists at high array lengths.
  - Afzal, Tanveer & Shahzad, Basit & Aslam, Salman. (2007). ENHANCED SHELL SORT ALGORITHM. https://www.researchgate.net/publication/234791427_ENHANCED_SHELL_SORT_ALGORITHM



Prelab part 1

1.) 8, ~~22~~, ~~7~~, ~~9~~, ~~31~~, ~~5~~, ~~13~~ 1 round
7 ~~22~~ 22 5 ~~31~~ 31
9 13

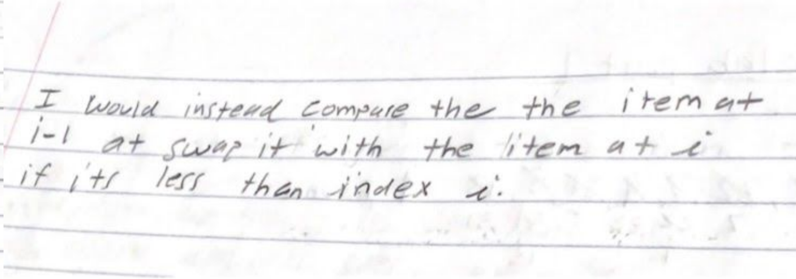
~~8~~, ~~7~~, 9, ~~22~~, ~~5~~, ~~13~~, 31 2 rounds
7 8 5 ~~22~~ 22
13

7, 8, 9, 5, 13, 22, 31 • Here I saw a pattern

* It would take 3 more iterations before it would be fully sorted, and that is the # of places the lowest number is from index 0. If we're talking about the original list, then $\boxed{\text{I think 5 rounds are needed.}}$ The lowest number moves back only 1 place per round, while the highest soner to the top.

2.) 4, 3, 2, 1 → 3, 2, 1, 4 → 2, 1, 3, 4 → 1, 2, 3, 4 ✓

* I now realize I was wrong in my claim from problem 1, it actually depends not on where the lowest number is from index 0. It takes O(n), or n passes in worst case, and $\boxed{\text{O(n), or n compares (or n-1 specifically).}}$ worst case then would be $O(n^2)$, and best case would be O(n), because it would only need 1 pass.



I would instead compare the the item at i-1 at swap it with the item at i if it's less than index i.

Pre-lab pt 2

1.) It would seem most gap sequences achieve very similar max comparisons at high array lengths, however, shell's original sequence $\frac{N}{2^2}$ is very outdated. In recent simulations, Marcin Ciura's sequence proves much faster than most predecessors, almost half the time of shell's at worst case. The gap sequence is yet to be understood strictly at an [illegible] average case scenario, but edge cases are clear. more recent sequences, like from Robert Sedgewick, undergo $O(n^{\frac{4}{3}})$ at worse case, significantly faster than Shell's $O(n^2)$.

You can use while loops as an alternative to the [illegible] 3rd for loop. For the 3rd loop, your [illegible] executing the if statement each time, regardless. A conditional while loop would be better

Part 3

3.) So if an element picked for the pivot is either the smallest or largest value, then $O(n^2)$ will occur

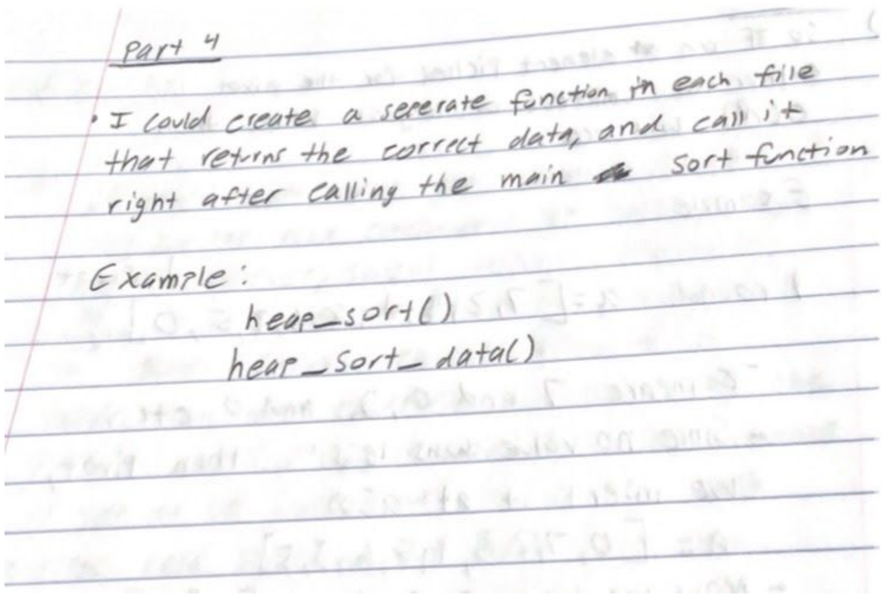
Example:

↓ pivot

1 round: $a=[7,2,3,1,8,6,3,5,0]$

- compare 7 and 0, 2 and 0 etc...
- Since no value was less than pivot, we insert it at a[0]
  $a = [0,7,2,3,1,8,6,3,5]$
- Now we have 2 partions [0], [7,2,3,1,8,6,3,

* In an extremley rare case, the non-0 partion will also have its lowest value last.



Part 4

- I could create a seperate function in each file that returns the correct data, and call it right after calling the main sort function

Example:

```
heap_sort()
heap_sort_data()
```

Version 2.0

Section 1: Overview

- This program is meant to take in 4 fundamental sorts, Bubble, Shell, Quick, and Heap sort and sort an array of elements according to their properties as well as keep track of the number of compares and moves. It can be modified to take any number of elements so long as it doesn’t extend past the max size of an unsigned integer of 32 bit length. The array elements are generated by a random seed that will either be defaulted or imputed by the user. The program displays all this in an organized fashion.

Section 2: Sorts

- **Bubble Sort**
  - Bubble sort starts at index 1, not 0. It iterates through an array comparing the value at index with index - 1 until index reaches the length of the array + 1.
  - For this sort I will be adding in a function to help track comparisons inside the if statement for checking values.
    - Ex: [9,6,1,7,5,2,3,8,0]
    - Bubble sort is simple yet tedious.
    - We start at index 1, so we compare 9,6
    - 6 < 9? Yes. Swap ->[6,9,1,7,5,2,3,8,0]
    - This process then repeats all the way until 9 will eventually “bubble” to the top.
    - First iteration will look like [6,1,7,5,2,3,8,0,9]
    - 0 will take a while to reach index 0.
- **Shell Sort**
  - Shell sort must include the files containing the predetermined gap lengths, which follow the 3-smooth pattern.
  - Shell sort begins at the first gap length value index of the array and checks each value preceding it by gap length value and makes a comparison until it reaches or extends out of array bounds.
  - There will come a point when gap length becomes 1, in which case it will default to an insertion sort.
    - Ex: [9,6,1,7,5,2,3,8,0]
    - Lets say our first gap is 3. So we start at 7 and check if 7 < 9. Yes? Swap.
    - Now we have [7,6,1,9,5,2,3,8,0]
    - Next is 5 < 6? Yes? -> [7,5,1,9,6,2,3,8,0]
    - And so on. Remember that for when we reach the value of 3 in the array, or index 6, we check if 3 < 9 and it works. However if it wasn’t then we go back another 3 and check if it's less than that number.

        - Repeat this for each gap in gaps until array sorted.

- **Quick Sort**
    - In this particular sort, we will be using a stack to store the min and max values of the array or array partition. Since we pop off the last values to be added, it can be a proper substitution for a recursive function which would return the last values to be called up until the first values to be called.
    - We start at the beginning of the array and push the min and max index values to the stack and partition it such that one side is less and one side is greater than the pivot.
    - The partition function returns p, or partition index of the array to split the original array in 2 with the new arrays having p as either the max or min value.
    - This will repeat until there are 2 arrays of length 1, which is by default sorted and after sorting all the subarrays the original array will be put together in order.
        - Ex: [6,3,7,5,8,2,9]
        - We pick our pivot, pivot = 5.
        - For this particular quicksort we set i = -1 and j = 7.
        - Decrement j by 1. If 9 < 6 we do a swap. No? Then decrement j by 1
        - Now j points to 2. Is 2 < 6. Yes? We swap 6 and 2.
        - New array = [2,3,7,5,8,6,9] and so on until we get [2,3,5,7,8,6,9]
        - Now we create 2 sub arrays, or in the code partition such that p will = 2, the index where we partition.
        - Now we have : [2,3] and [7,8,6,8,9] and repeat the above process until all subarrays are sorted.

- **Heap Sort**
    - This particular heap sort will utilize a max heap. We first build a max heap and swap the value at root with the last item in the array.
    - We need to create a variable for storing the current array size because after each swap we need to decrement that variable by 1 since the next swap will be with n - 1.
    - A function max_child will be implemented to return the highest value of the 2 children at indices 2k and 2k+1.
        - Ex: [5,2,1,3,7,0]
        - First build the max heap, which will be [7,5,3,2,1,0]
        - We then swap 7 and 0, so -> [0,5,3,2,1,7].
        - We can then look at the array in terms of [0,5,3,2,1,**7**] (ignoring 7).
        - Next max heap will be [5,3,2,1,0,**7**].
        - Swap 5 and 0, getting [0,3,2,1,**5**,**7**].
        - Repeat.

Section 3: sets and stacks

- **Set.c**
    - This file will be very small and brief. Almost all functions will return a set that will be the outcome of an insertion,removal,intersection, etc… of 1 or 2 sets. The only exception will be set_member which returns whether a value is a part of a set.
    - A set in this case is an unsigned integer of 32 bits with each bit representing a storage location.
    - I plan on using an enumeration for my sets and setting bubble to 0, shell to 1 etc… and storing them at the set index of their enumeration.
- **Stack.c**
    - A stack will have 3 tags: top, capacity and *items.
    - Top represents the top of the stack, so index 0 initially. When an item is added it increases by 1.
    - Capacity represents the max amount of items a stack can hold.

- The items array holds all the stack values.
- Stack_create will allocate enough memory for a stack of min_capacity num of items each item being an int of 64 bits.
- Stack_empty will return whether the top of the stack points to the index value of 0.
- Stack_delete will free each spot in the array then the array itself, then set the array to NULL.
- Stack_push will first check if the capacity is full, in which case it needs to allocate more memory, doubling the stack capacity. Realloc will be helpful. Once it is known that there is more space in the stack, we can then set top to the value being pushed and increment top by 1.
- Stack_pop will first check if the stack is empty in which case it can't pop anything, but if it isnt then decrement top by 1, set the variable to the item being popped and set the value at top to 0.
- Stack_print will be helpful for testing my stack functions.

<u>Section 4: sorting.c</u>

- First Initialize and set all default and necessary items specified in the asgn document.
- **I will be using an external file labeled "utilities.h" that declares two helper variables for keeping track of the moves and compares in each sorting file.**
- I will also be creating 3 helper functions
    - **Fill_array** will take in an array and its size and according to the seed value fill the array with random numbers
    - **Print_array** will take an array, its size, and the print item value or "p". It will print the amount of items specified in an organized fashion, 5 columns per row.
    - **Print_error** will be used for the multiple occasions in which command line input is faulty
- First off I will iterate through the command line and depending on argument values, add the right sorts to the sort_list set and change the 3 non sort arguments accordingly

- There will be error checking scattered throughout to check for an empty set, no arguments etc…
- Create the main array of correct length.
- Next, I will iterate through the sort_list set and run each sort in the set and print the statistics. Each case will fill the array at the beginning and set moves and compares to 0 and the end.
- Finally free the array and set to NULL.

Section 5: Layout

- Each ".c" file will include its ".h" except sorting.c which will include "utilities.h"
- The makefile will compile the sorting binary like so:
    - $(CC) -o sorting sorting.c set.c bubble.c quick.c stack.c shell.c heap.c