DESIGN

Pre-lab Part 1

1. How many rounds of swapping do you think you will need to sort the numbers 8,22,7,9,31,5,13 in ascending order using Bubble Sort?

Rounds:

1) 8, 7, 9, 22, 5, 13, 31
2) 7, 8, 9, 5, 13, 22, 31
3) 7, 8, 5, 9, 13, 22, 31
4) 7, 5, 8, 9, 13, 22, 31
5) 5, 7, 8, 9, 13, 22, 31

5 rounds total.

2. How many comparisons can we expect to see in the worse case scenario for Bubble Sort?

It will perform (n - 1) comparisons in each round (and n will decrease with each round). The number of rounds is (n - 1). Total number of comparisons n (n - 1) / 2.

3. How would you revise the algorithm so the smallest element floats to the top instead?

I would start comparing elements not from the beginning, but from the end of the array.

Pre-lab Part 2

1. 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 gap size? Cite any sources you used.

“The running time of Shellsort is heavily dependent on the gap sequence it uses. For many practical variants, determining their time complexity remains an open problem.” Wikipedia

“The question of deciding which gap sequence to use is difficult. Every gap sequence that contains 1 yields a correct sort (as this makes the final pass an ordinary insertion sort); however, the properties of thus obtained versions of Shellsort may be very different. Too few gaps slows down the passes, and too many gaps produces an overhead.” Wikipedia

2. How would you improve the runtime of this sort without changing the gap sequence?

I can significantly reduce the number of moves if, instead of exchanging the values of **arr**[j] and **arr**[j - gap] I memorize the very first element in the subarray (**arr**[i]) and make a "hole" there and then I move the following elements there. As in the pseudocode shown on Wikipedia.

Pre-lab Part 3

1. Quicksort, with a worse case time complexity of $O(n^2)$, 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.

The worst case for this sorting is when either the first or the last element is selected as a pivot and the data is sorted. Since our sort selects the element in the middle, sorted data won't be a problem in our case.

Pre-lab Part 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 am planning to use global variables for counters in the "sorting.c" file. Like this:

```
uint32_t bubble_moves = 0;

uint32_t bubble_compares = 0;
```

I will use the "extern" keyword to access every variable in the corresponding file. For example, in file "bubble.c", I will do this:

```
extern uint32_t bubble_moves;

extern uint32_t bubble_compares;
```

Design

Pseudocode of the program.

```
Start:

bubble_moves = 0

bubble_compares = 0

shell_moves = 0

shell_compares = 0

quick_moves = 0

quick_compares = 0

heap_moves = 0

heap_compares = 0

seed = 7092016

size = 100
```

elements = 100

options = set_empty()

Parse command line and set corresponding members in ***options*** and set ***seed***, ***size*** and ***elements*** if corresponding options are present.

Initialize PRNG using srandom(seed)

Create a reference dynamic array ***X[]*** of appropriate ***size***

Populate ***X[]*** with random numbers

If ***options*** has option 'a' or options has option 'b' then:

    Create a copy of array ***X[]***, array ***A[]***

    Sort ***A[]*** using ***bubble_sort*** (it will update the values of the corresponding counters ***bubble_moves*** and ***bubble_compares***)

    If ***elements*** > 0 then print no more than ***elements*** of ***A[]*** in tabular format

    Report ***size***, ***bubble_moves*** and ***bubble_compares***

If ***options*** has option 'a' or options has option 's' then:

    Create a copy of array ***X[]***, array ***A[]***

    Sort ***A[]*** using ***shell_sort*** (it will update the values of the corresponding counters ***shell_moves*** and ***shell_compares***)

    If ***elements*** > 0 then print no more than ***elements*** of ***A[]*** in tabular format

    Report ***size***, ***shell_moves*** and ***shell_compares***

If ***options*** has option 'a' or options has option 'q' then:

    Create a copy of array ***X[]***, array ***A[]***

    Sort ***A[]*** using ***quick_sort*** (it will update the values of the corresponding counters ***quick_moves*** and ***quick_compares***)

    If ***elements*** > 0 then print no more than ***elements*** of ***A[]*** in tabular format

    Report ***size***, ***quick_moves*** and ***quick_compares***

If ***options*** has option 'a' or options has option 'h' then:

    Create a copy of array ***X[]***, array ***A[]***

    Sort ***A[]*** using ***heap_sort*** (it will update the values of the corresponding counters ***heap_moves*** and ***heap_compares***)

If ***elements*** > 0 then print no more than ***elements*** of ***A[]*** in tabular format

Report ***size***, ***heap_moves*** and ***heap_compares***

End