Design Document

Description

- The purpose of this program is to explore and compare different sorting methods. We want to see how these different sorting algorithms operate by writing and understanding the code behind them. We want to compare their time complexities analyzing how many comparisons it takes for each algorithm to sort.

Structure

- **Insertion Sort:**
  - The idea of insertion sort is to sort the first number of the sequence, then the first two numbers, then the first three numbers, and so forth.
  - To sort the next number, keep moving it to the left until it reaches a number less than it or reaches the left end of the array.

```
func insertion_sort (int[] A):
      for each element A[i] in range(1, len(A)):
            while element not at left end of array and A[i] < A[i-1]:
                  move A[i] to the left
```

- **Shell Sort:**
  - The idea is similar to insertion sort, but by using gaps instead of comparing elements to their left neighbor.
  - For each gap, we run a variation of insertion sort. The final gap size will be 1.

```
func shell_sort (int[] A):
      int largest_gap = log(3 + 2 * len(A)) / log(3)
      for i in range(largest_gap, 0):
            gap = (pow(3, i) - 1) / 2
            for each element A[j] in range(gap, len(A)):
                  while (j - gap) is in the array and A[j] < A[j - gap]:
                        move A[j] to the left
```

- **Heap Sort:**
  - The idea is to build a heap, where each node points to two nodes less than it. The largest node will point to the 2nd and 3rd largest nodes (not necessarily ordered), which will point to the 4th, 5th, 6th, and 7th largest nodes (not necessarily ordered), and so forth.
  - Once we have this heap, we know the largest element is in the first index. Swap this element with the last element in the array. Now, we can set aside the largest element and build a heap with the remaining elements.

- Now, we have a random element in the first index. Fix the heap by moving this element down the heap, swapping with the largest child each time.
- Once again, we have the largest element of the heap in the first index. Repeating this *n* times we can sort the array. Also, fixing the heap only takes log(*n*) time, so the overall time complexity is O(*n* log(*n*)).

```
func fix_heap (int[] A, int index):
      int child1 = 2*index, child2 = 2*index + 1
      if A[index] < A[child1] or A[index] < A[child2]:
            swap A[index] with larger of A[child1] and A[child2]
            fix_heap(A, swapped index)

func heap_sort (int[] A):
      // building the heap- O(n log(n))
      for i in range(len(A) / 2, 0):
            fix_heap(A, i)

      // moving largest element and refixing heap- O(n log(n))
      for i in range(len(A) - 1, 0):
            swap A[0] with A[i]
            fix_heap(A, 0)
```

- **Quick Sort:**
  - The idea is to choose any element as a pivot. Then, put each element less than the pivot to the left of it, and elements larger than the pivot to the right of it.
  - Repeat this process on the “less-than-pivot” elements and the “greater-than-pivot” elements.
  - For this assignment I chose to use the last element as the pivot.

```
func quick_sort (int[] A, int first = 0, int last = len(A) - 1):
      int pivot = A[last - 1]
      int index = first
      for each element A[i] in range(first, last):
            if A[i] < pivot:
                  A[index] = A[i]
                  i+=1
      i -= 1 // puts i where the pivot element is
      quick_sort(A, first, i-1)
      quick_sort(A, i+1, last)
```

- **Stats:**
    - Used to calculate the number of moves and comparisons of each sorting algorithm.
    - To swap two elements of the array, 3 moves are added to the algorithm's stats.
    - To move an element of the array, 1 move is added to the algorithm's stats.
    - To compare two elements of the array, 1 compare is added to the algorithm's stats.

```
typedef struct {
      int moves;
      int compares;
} Stats;
```

Inputs and Outputs

This chart shows how the program will handle the selected options when run.