Assignment 3 - DESIGN.pdf

Description of the program:

For the DESIGN I will be focusing on the implementation of sorting functions, for an array of integers. In this assignment, I will be producing 4 different sorting functions, for which the **compute time**, **moves**, and **compares** will be recorded. The program will work by having an array as an input with arguments of: n(number of elements), r(seed). The following are the sorting functions I will be implementing:

- Insertion Sort
- Heap Sort
- Quick Sort
- Batcher's Odd-Even Merge Sort

Files to be included:

Source and header files:

1. **batcher.c** implements Batcher Sort
2. **batcher.h** specifies the interface to batcher.c
3. **insert.c i**mplements Insertion Sort
4. **insert.h** specifies the interface to insert.c
5. **heap.c** implements Heap Sort
6. **heap.h** specifies the interface to heap.c
7. **quick.c** implements recursive Quicksort
8. **quick.h** specifies the interface to quick.c
9. **set.h** implements and specifies the interface for the set ADT
10. **stats.c** implements the statistics module
11. **stats.h** specifies the interface to the statistics module
12. **sorting.c** contains main() and may contain any other functions necessary to complete the assignment.

Additional Files:

1. **Makefile:** formats all source code, including the header files.

2. **README.md:** Description of how to use my program and Makefile. It also includes any command-line option that my program accepts. Any false positives reported by scan-build should be documented and explained here as well. Note down any known bugs or errors in this file as well for the graders.
3. **DESIGN.pdf (This file):** Design document describes the preliminary design and design process for my program with sufficient detail for potential replication
   WRITEUP.pdf: Analysis and description of the produced program, as well as graphs generated and/or information, gathered from outputs

Pseudocode/Structure:

Insertion Sort:

O- upper bound - TIME: n^2

The insertion sort is a very simple sorting algorithm. I understand it in a way as if humans were to sort a specific order of numbers. Firstly one puts the numbers in a random sequence, and takes the numbers one by one, sorting them by comparison to each other sequentially.

Below is the pseudocode -

```
def insertion_sort(A: list):
        For i in range(i, len(A)):
                j = i
                Temp = A[i]
                While j > 0 and temp < A[j-i]:
                        A[j] = A[j-i]
                        J -= 1
                A[j] = temp
```

Heap Sort:

O- upper bound - TIME: $log_2(n)$

The heap sort is a bit more complicated. It works in a way of tree-like structure. In this structure, the largest element is in place on the top, and its branches are the elements with smaller values. This also applies to the branches of the branches. Therefore it sorts the array one by one, however in batches of values.

Below is the pseudocode(excluding the build and heapsort infrastructure) -

```
Def heap_sort(A: list):
    First = 1
    Last = len(A)
    build_heap(A, first, least) - this builds the tree like heap sort structure
    For leaf in range(last, first, - 1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A,first, leaf - 1)  - this reload the heap with the new value on top
```

Quick Sort:

O- upper bound - TIME: $n * log(n)$

NOTE: I used this source to help me understand the quick sort-

https://www.programiz.com/dsa/quick-sort

Quicksort is very efficient and very simple. It takes a particular pointer element from an array and arranges itself so bigger values are on its side from that pointer, and smaller on the other. Then it takes the number from one of its sides and repeats the process until all elements are arranged.

Below is the pseudocode(excluding partition)-

```
Def quick_sorter(A: list, lo: int, hi: int):
    If lo < hi:
        P = partition(A, lo, hi)
        quick_sorter(A, lo, p-i)
        quick_sorter(A, p+1,hi)
Def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

Batcher’s Odd-Even Merge Sort:

NOTE: I used this source to help me understand the merge sort-

https://www.programiz.com/dsa/merge-sort

O- upper bound - TIME: $log(n)^2$

In Merge sort, the problem is divided into multiple sub-problem, which are solved individually. When the sub-problems are small enough, they will be sorted sequentially, and afterward, the whole array is put together, to return the formed sequence.

Below is the pseudocode-

```
def comparator(A: list , x: int , y: int):
    If A[x] > A[y]:
        A[x], A[y] = A[y], A[x]


Def batcher_sort(A: list):
    If len(A) == 0:
        Return


    n=len(A)
    t = n.bit_length()
    p = i << (t - 1)

    While p > 0:
        q = 1 << (t - 1)
        r = 0
        d = p


        While d > 0:
            For i in range(0, n-d):
                if  (i & p) == r :
                    comparator(A,i, i + d)
        d = q - p
        q >>= 1
        r = p
    p >>= 1
```

<u>Batcher's Odd-Even Merge Sort:</u>
My last task was to put it all together and collect the data from the specific algorithms. The following is the data I will be gathering:

- the size of the array
- the number of moves required
- the number of comparisons required

This will all be put together in *sorting.c* function. The command options and how does the program function are explained in README.md

Credit:

- I took the majority of the pseudocode from the assignment 3 paper written by prof. Darrel Long
- I have watched the recording of Eugenes Lab section on 1/21/22, uploaded on Yuja. Unfortunately, I could not have attended the section live.
- I have used external sources cited in the paper, to gain a deeper understanding of how to do the individual sorting algorithms function