Assignment 3 Writeup

Nhan Nguyen

January 2022

1 Introduction

- For this writeup, I'll be describing how I implemented the main() function using C code, as well as using the included statistics module in order to discuss some important lessons learned after reviewing the data from the statistics module

2 main() description and code

2.1 Idea

- main()'s purpose is to utilized 4 different sorting algorithms in order to sort a randomized array, additionally implements some commands the user can type in using getopt().

- To see which sorting algorithm is enabled, we will be using a set to determine that. A **set** is a bit-wise number that has each bit set to either 0 or 1 to determine if that bit is on or off. Each of the 4 bits in the **set** will be bound to a certain sorting algorithm

- To randomized an array, it's critical to use **srandom()** to set the seed before generating numbers using **random()**

2.2 Code

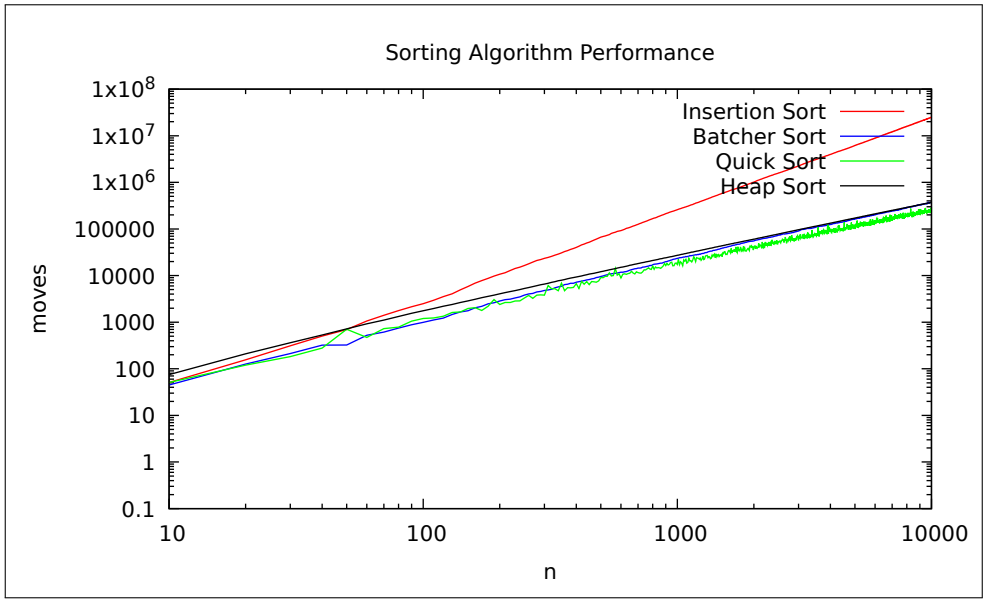
```
1  int main(int argc, char **argv) {
2      Stats test;
3      char *ptr;
4      int opt;
5      uint32_t seed = 13371453;
6      uint32_t size = 100;
7      uint32_t elements = 100;
8      Set coms = empty_set();
9      while ((opt = getopt(argc, argv, "ahbiqr:n:p:H")) != -1) {
10         switch (opt) {
11         case 'a':
12             coms = insert_set(0, coms);
```

```
            coms = insert_set(1, coms);
            coms = insert_set(2, coms);
            coms = insert_set(3, coms);
            break;
        case 'h': coms = insert_set(0, coms); break;
        case 'b': coms = insert_set(1, coms); break;
        case 'i': coms = insert_set(2, coms); break;
        case 'q': coms = insert_set(3, coms); break;
        case 'r': seed = (uint32_t) strtoul(optarg, &ptr, 10);
    break;
        case 'n': size = (uint32_t) strtoul(optarg, &ptr, 10);
    break;
        case 'p': elements = (uint32_t) strtoul(optarg, &ptr, 10);
    break;
        case 'H':
            usage();
            return 0;
            break;
        }
    }
    uint32_t *testarr = NULL;
    testarr = malloc(size * sizeof(uint32_t));
    uint32_t cnt = 0;
    if (testarr == NULL)
        return EXIT_FAILURE;
    while (cnt < 4) {
        if ((coms & 1) == 1) {
            srandom(seed);
            for (uint32_t i = 0; i < size; i++) {
                testarr[i] = (uint32_t)(random() & (uint32_t)((1 <<
     30) - 1));
            }
            if (cnt == 0) {
                heap_sort(&test, testarr, size);
                printf("Heap sort, %" PRIu32 " elements, %" PRIu64
    " moves, %" PRIu64 " compares\n",
                    size, test.moves, test.compares);
            } else if (cnt == 1) {
                batcher_sort(&test, testarr, size);
                printf("Batcher sort, %" PRIu32 " elements, %"
    PRIu64 " moves, %" PRIu64
                    " compares\n",
                    size, test.moves, test.compares);
            } else if (cnt == 2) {
                insertion_sort(&test, testarr, size);
                printf("Insertion sort, %" PRIu32 " elements, %"
    PRIu64 " moves, %" PRIu64
                    " compares\n",
                    size, test.moves, test.compares);
            } else {
                quick_sort(&test, testarr, size);
                printf("Quick sort, %" PRIu32 " elements, %" PRIu64
     " moves, %" PRIu64
                    " compares\n",
                    size, test.moves, test.compares);
            }
            uint32_t print = (size < elements) ? size : elements;
```

```
62              for (uint32_t i = 0; i < print; i++) {
63                  printf("%13" PRIu32 " ", testarr[i]);
64                  if ((i + 1) % 5 == 0 && i != print - 1)
65                      printf("\n");
66              }
67              printf("\n");
68          }
69          cnt++;
70          coms /= 2;
71      }
72      free(testarr);
73      return 0;
74  }
```

3 Graph analysis and interesting observations

- The graph generated below will be based on the provided statistics module. The statistics module calculates the number of moves and comparisons a sorting algorithm makes when executed, and based on this statistics we can start formulating some observations.



3.1 Observation 1

Right off the bat, the most apparent feature in this graph is that the 3 sorting algorithms: Quick sort, Batcher Sort, and Heap Sort do approximately the same number of moves as the length of the array **n** increases, while the insertion sort increases dramatically. From this feature, we can deduce that different sorting

algorithms have different performance as the size of the data (in this case the length of the array) increases

3.2 Observation 2

- From the 1st observation, you might be wondering "if Batcher Sort, Quick Sort, and Heap Sort exists, why do we even need Insertion Sort?" The answer to this question will be answered when we look at the smaller data size of the graph. As we can see, when it comes to smaller data sizes, insertion sort does fewer moves (in other words perform better) than the supposedly faster Batcher Sort. This is because Batcher Sort does a fixed number of comparisons and moves.

- Therefore while Batcher Sort is effective for large array sizes, performing a fixed number of moves and comparisons is not as effective against the smaller array size, in which the insertion sort excels at.

- Ultimately, we learned that while an algorithm may perform worst with large data sizes, it can perform better against smaller data sizes and should be used alongside with a faster algorithm to accommodate for their worse performance in the small data set

3.3 Observation 3

- Another feature we notice in the graph is that the lines are not very straight. Why is this the case? This question can be answered by looking at the performance of an algorithm like Quick Sort (The green line), Quick Sort is an algorithm that depends on the pivot position in order to swap elements around, which may result in more or fewer moves needed and ultimately lead to the algorithm being **unstable**. In other words, Quick Sort is an algorithm that may under-perform or over-perform slightly due to its instability. When we look at Batcher Sort, as explained in the 2nd Observation, since Batcher Sort does a fixed number of comparisons and moves, we can see a moderately straight line from it and hence its improved stability.

- From this observation, we learned that different sorting algorithms have different stability, it's important to know how stable each sorting algorithms are in order to use them in different situations, like sorting valued pairs with duplicate values in the first portion of the pair