Design Document - Sorting: Putting Your Affairs in Order

Lev Teytelman

January 24, 2022

1 Program Description

This program generates pseudorandom numbers via a seed, sorts them using various sorting algorithm, and prints them.

2 Included Files

- batcher.c - The source file with the implementation of batcher sort and helper functions.
- batcher.h - The header file with the function declaration of `batcher_sort` for use in other files.
- insert.c - The source file with the implementation of the insertion sort function.
- insert.h - The header file with the declaration of the `insertion_sort` function for use in other files.
- heap.c - The source file with the implementation of heap sort and helper functions.
- heap.h - The header file with the declaration of the `heap_sort` function for use in other files.
- quick.c - The source file with the implementation of quick sort and helper functions.
- quick.h - The header file with the declaration of the `quick_sort` function for use in other files.
- set.h - Provides the set ADT and implements the appropriate functions for it.
- stats.c - The source file with the implementation for the Stats struct functions.
- stats.h - The header file with the function declaration and the Stats struct.

- sorting.c - The file containing the `main` function and logic for calling the appropriate sorts.
- Makefile - The logic to preprocess, compile, and link the program files with a single command.
- README.md - A short rundown of the program's functionality and usage.
- DESIGN.pdf - This file, describing preliminary ideas and pseudocode for the assignment.
- WRITEUP.pdf - The document analyzing findings and the code used to make the program work.

3 Structure

This program will take flags to specify which sorts to run, the size of the array, the amount of sorted numbers to print, and the seed for the pseudorandom number generator. It will then check which sorts are enabled, and call them accordingly. The sorts have varying efficiencies, as will be shown with the differing compare and move amounts for each sort. The more sorts/compares it does, the worse its performance is.

This assignment uses a set data structure to store information about which sorts are toggled. During argument processing, bits will be set to 1 using the `insert_set` function for the corresponding sort index. Afterwards, the saved options are processed in a `for` loop and the corresponding function is called if the bit is set (checked using `member_set`).

An important but easily overlooked aspect of the program is that the array's size should be able to go up to the memory limit of the computer. This aspect of scaling can be covered using C's `malloc` function when creating the array along with `free` after using it to ensure no memory leaks. The array will also be reused by each sort, as there's no reason to use separate arrays for each one as long as the values are set back in the same configuration each time.

4 Pseudocode

A big part of this assignment is understanding and effectively utilizing the set data structure, as well as pseudorandom number generation. To generate pseudorandom numbers for the array, I will create a function which will operate similarly to this:

```
def generate_random(array, seed):
    set_seed(seed)
    for i in range(len(array)):
        array[i] = random() & ~(3 << 30)
```

This will populate the array with pseudorandom numbers generated from the given seed. The `& ~(3 << 30)` is a trick in which I take only the 30 least significant bits from the generated number (by inverting 11000... and then bitwise ANDing the result).

The set is used for storing which sorts to do depending on the options, and the usage in the `main` will look similar to the following:

```
set = empty_set()
while(get_opt() != -1):
    if (option == 'a'):
        add(quick, set)
        add(heap, set)
        add(insertion, set)
        add(batcher, set)
    else if (option == 'q'):
        add(quick, set)
    else if (option == 'h'):
        add(heap, set)
    else if (option == 'i'):
        add(insertion, set)
    else if (option == 'b'):
        add(batcher, set)
```

This will ensure that all specified sorts are added to the set, which can later be checked for the appropriate sorts:

```
if (has(quick, set)):
    quick_sort()
if (has(heap, set)):
    heap_sort()
if (has(insertion, set)):
    insertion_sort()
if (has(batcher, set)):
    batcher_sort()
```

We can use the set we populated with our option processing to call the proper functions and do the right sorts.