CSE 13S: Assignment 3 Design Document

Johnny Li

February 5, 2022

1 Introduction

Sorting data is an essential and common aspect of programming so it's surprise that there are numerous libraries and routines available for this task. These resources make it easier to order data but just because they exist and are easily accessible, it does not mean that we can ignore the importance of learning the various sorting algorithms. In fact, it is even more critical for programmers to understand the algorithms and their underlying principles.

Having a solid understanding of different sorting algorithms is crucial since it allows us to make informed decisions when choosing the best one for a particular task. The efficiency of the algorithm used can greatly impact the overall performance of the program, and a sub-optimal choice can result in slow and inefficient execution. Each algorithm has their own strengths and weakness. For example, QuickSort is the most common and fastest sorting algorithm using comparisons, usually being two to three times faster than its counterparts. However, it also has a worst case performance of $O(n^2)$ while its competitors are strictly $O(nlogn)$ in their worst case; so if used or implemented incorrectly, QuickSort can actually be significantly slower. Therefore, it is important to have a deep understanding of the algorithms and their implementation so that we can choose the most appropriate one for each task.

That is the aim of this lab, for us to understand how each sorting method is implemented. And by building them ourselves, we can see how and when each should be used to their greatest effect. Through hands-on experience building these methods, we will be able to gain valuable insights into when and how each method should be used effectively. By constructing each sorting method from scratch, we will be able to see the inner workings of each algorithm and understand the strengths and weaknesses of each method. By the end of the lab, we should be well-equipped to choose the right sorting method for any given task and be able to effectively implement and optimize it for maximum performance.

2 Tasks

The four sorting methods we'll be implementing are Shell Sort, Batcher Sort, HeapSort, and QuickSort. The implementation will be based on the provided Python pseudocode in header files shell.h, batcher.h, heap.h, and quick.h respectively. The header files will serve as the interface for these sorts and should not be modified. A test harness will be created in sorting.c to test the sorting algorithms by creating an array of pseudo-random elements. Statistics about each sort's performance, such as the size of the array, the number of moves required, and the number of comparisons required, will also be gathered. The test harness will support a number of command-line options including, but not limited to, specifying which sorting algorithm to use, setting the random seed, setting the array size, and printing out elements from the array. The program will track which command-line options are specified using a set data structure, and the necessary set functions will be implemented in set.c.

2.1 Sorts

The implementation of the sorting methods will focus solely on Shell Sort, Batcher Sort, HeapSort, and QuickSort. The provided Python pseudocode in header files shell.h, batcher.h, heap.h, and quick.h

will serve as the basis for the implementation and should not be altered. The sorting algorithms will be tested using a test harness in sorting.c, which will generate an array of pseudo-random elements and will also gather statistics on each sort's performance, including the size of the array, the number of moves required, and the number of comparisons required.

The test harness will allow for various command-line options, such as selecting a specific sorting algorithm, setting the random seed, defining the array size, and printing elements from the array. The program will keep track of these options using a set data structure and the necessary set functions will be implemented in set.c. The program will provide a comprehensive and thorough evaluation of the performance of the sorting algorithms.

Shell Sort

The Shell Sort algorithm was created and published in the July 1959 issue of the Communications of the ACM journal by Donald L. Shell after having just graduated from the University of Cincinnati with a Ph.D in mathematics. The algorithm that Shell published was a variation of insertion sort although it sorts based on "gaps" rather than nearest neighbor. So depending upon the data and how large the gaps are, Shell Sort can be exponentially more efficient than insertion sort.

Shell Sort first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted. Said interval between the elements is reduced based on the sequence used. So its expected time complexity is entirely dependent upon the gap sequence. Below is the Python pseudocode provided in the Assignment 3 PDF.

```
void shell_sort(int arr[], int n, int gaps[], int m) {
    for (int k = 0; k < m; k++) {
        int gap = gaps[k];
        for (int i = gap; i < n; i++) {
            int j = i;
            int temp = arr[i];
            while (j >= gap && temp < arr[j - gap]) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
        }
    }
}
```

Batcher Sort (Merge Exchange Sort)

Batcher's odd-even mergesort (or Batcher's method) unlike the other methods we'll be implementing, is a sorting network. Sorting networks, or comparator networks, are circuits built for the express purpose of sorting a set number of inputs, that number usually being a power of 2 (as was the case in Batcher's original method) although Knuth's modification to Batcher's work (Merge Exchange Sort) removes this restriction and allows an input of any infinite size. This is the algorithm we'll be implementing with its provided, respective Python pseudocode below:

```
void comparator(int *A, int x, int y) {
    if (A[x] > A[y]) {
        int temp = A[x];
        A[x] = A[y];
        A[y] = temp;
    }
}

void batcher_sort(int *A, int n) {
```

```
    if (n == 0)
        return;

    int t = 0;
    for (int i = n; i; i >>= 1) t++;

    int p = 1 << (t - 1);
    while (p > 0) {
        int q = 1 << (t - 1);
        int r = 0;
        int d = p;

        while (d > 0) {
            for (int i = 0; i < n - d; i++) {
                if ((i & p) == r) {
                    comparator(A, i, i + d);
                }
            }
            d = q - p;
            q >>= 1;
            r = p;
        }
        p >>= 1;
    }
}
```

Quite a bit more complicated than the other methods, Batcher's method basically acts as a parallelized Shell Sort. While the latter sorts elements based on gaps until the gap reached one, with the most distant elements being sorted first. Batcher's method is similar but instead of relying on gaps, it k-sorts the even and odd sub-sequences of the array – where k is some power of 2. By doing this, all elements are – at maximum – only k-indices away from their sorted position.

Consider an array – A – whose number of elements is equal to some power of 2. Batcher's method would first $|A|/2$ sort the even and odd sub-sequences. This would repeat, sorting the remaining elements with $|A|/2^x$ sorts each cycle until it finally 1-sorted them, merging the even and off sequences. So for a 32 element array, it would first conduct a 16-sort, then 8-sort, 4-sort, 2-sort, and a final 1-sort. Each level of k-sorting is called a round and each the sort occurs in parallel, meaning it's usually more efficient than Shell Sort. Due to the nature of the algorithm, any indices involved in a pairwise comparison when sorting a sub-sequence using k-sort will not appear in any other comparison during the same round. The clever use of the bitwise AND operator ensures this characteristic.

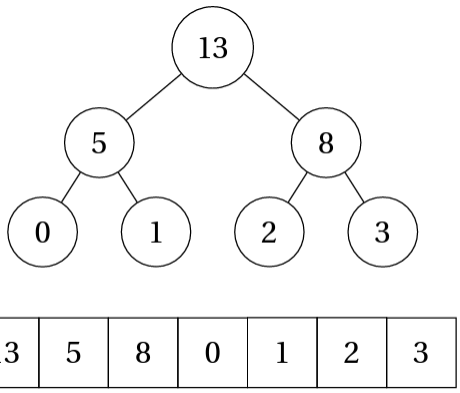
Although Batcher's Sort can be written to run parallel, our implementation will run sequentially, sorting the input over several rounds. Thus, for an array size of n, the initial value to k-sort with is $k = log2(n)$. As stated previous paragraph, the even and odd sub-sequences are first k-sorted, then $k/2$-sorted, $k/4$-sorted, and so on until they are 1-sorted.

HeapSort

HeapSort, along with the heap data structure, was invented in 1964 by J. W. J. Williams, just five years after Shell published his Shell Sort method. The heap data structure is typically implemented as a specialized binary tree with two kinds of heaps: max heaps and min heaps. In a max heap, any parent node must have a value that is greater than or equal to the values of its children while for a min heap, any parent node must have a value that is less than or equal to the values of its children. The heap is typically represented as an array, in which for any index k, the index of its left child is $2k$ and the index of its right child is $2k + 1$; thus, the parent index of any index k should be $k/2$.

Building a Heap The process of constructing a heap involves transforming an array into a data structure that follows the rules of either a max or min heap. In this scenario, we will be building a

max heap, where the largest element in the array is positioned as the root of the heap. This element will occupy the first position of the array from which the heap is built. To ensure that the elements in the array comply with the constraints of a max heap, they must be rearranged in a specific manner.



Fixing a Heap In the next step, we will be implementing the HeapSort algorithm. The main idea behind HeapSort is to extract the largest elements in the array and place them at the end of the sorted array in ascending order. This is achieved by first transforming the unsorted array into a binary heap data structure, where the largest element is always at the top.

Once the heap is constructed, we repeatedly remove the largest element from the top and place it at the end of the sorted array. After removing the element, the heap needs to be fixed to maintain its properties as a binary heap. This involves restructuring the remaining elements in the heap so that the largest element is once again placed at the top.

This process continues until all elements have been removed from the heap and placed in the sorted array. At the end of the process, we will have a sorted array in increasing order. It is important to note that the process of removing elements from the heap and fixing the heap takes $O(logn)$ time, making the overall time complexity of HeapSort $O(nlogn)$.

Heap Maintenance in Python

```
int max_child(int *A, int first, int last) {
    int left = 2 * first;
    int right = left + 1;
    return (right <= last && A[right - 1] > A[left - 1]) ? right : left;
}

void fix_heap(int *A, int first, int last) {
    int found = 0;
    int mother = first;
    int great = max_child(A, mother, last);

    while (mother <= last / 2 && !found) {
        if (A[mother - 1] < A[great - 1]) {
            int temp = A[mother - 1];
            A[mother - 1] = A[great - 1];
            A[great - 1] = temp;
            mother = great;
            great = max_child(A, mother, last);
        } else {
            found = 1;
```

```
        }
    }
}
```

HeapSort in Python

```
void build_heap(int *A, int first, int last) {
    int i;
    for (i = last / 2; i >= first; i--) {
        fix_heap(A, i, last);
    }
}

void heap_sort(int *A, int n) {
    int first = 1, last = n, leaf;
    build_heap(A, first, last);
    for (leaf = last; leaf >= first; leaf--) {
        int temp = A[first - 1];
        A[first - 1] = A[leaf - 1];
        A[leaf - 1] = temp;
        fix_heap(A, first, leaf - 1);
    }
}
```

QuickSort

QuickSort (sometimes called partition-exchange sort) was developed by British computer scientist C.A.R. "Tony" Hoare in 1959 and published in 1961. It's one of the most commonly used sorting algorithms used by programmers; and the fastest comparison-based method when properly used and implemented. It is usually two or three times faster than its two main competitors, Merge Sort and HeapSort. Although it has a worst case performance of $O(n^2)$ while its counterparts are strictly $O(nlogn)$ in their worst case.

QuickSort operates based on the divide-and-conquer approach, which means that it partitions an array into two sub-arrays by selecting an element in the array to act as a pivot. Elements that are less than the pivot are placed in the left sub-array, while elements that are greater than or equal to the pivot are placed in the right sub-array. QuickSort is considered to be an in-place algorithm, meaning that it does not require additional memory to allocate for sub-arrays. Instead, QuickSort utilizes a subroutine called "partition" that places the elements less than the pivot to the left side of the array and elements greater than or equal to the pivot to the right side of the array, returning the index that separates the partitioned parts of the array. QuickSort is then recursively applied to the partitioned parts of the array, leading to the sorting of each array partition that contains at least one element.

Like the HeapSort algorithm, the provided QuickSort pseudocode operates based on 1-based indexing, meaning it subtracts 1 to account for the 0-based indexing that the C programming language (along with most other programming languages) use when accessing elements in the array.

Quicksort operates in-place, without the need for additional memory allocation for sub-arrays to hold the partitioned elements. Instead, it uses a subroutine called "partition()" which sorts the elements in the array based on the pivot selected. The pivot element is used to divide the array into two sub-arrays, with elements less than the pivot placed on the left side of the array and elements greater than or equal to the pivot placed on the right side. The "partition()" subroutine returns the index that separates the two partitions. The Quicksort algorithm is then applied recursively to each partition until the entire array is sorted. It's worth mentioning that, similar to the Heapsort algorithm, the Quicksort pseudocode is based on 1-based indexing and adjusts for 0-based indexing by subtracting one when accessing elements in the array.

Partition in Python

```
int partition(int A[], int lo, int hi) {
    int i = lo - 1;
    for (int j = lo; j < hi; j++) {
        if (A[j] < A[hi]) {
            i++;
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
    int temp = A[i + 1];
    A[i + 1] = A[hi];
    A[hi] = temp;
    return i + 1;
}
```

Recursive Quicksort in Python

```
void quick_sorter(int A[], int lo, int hi) {
    if (lo < hi) {
        int p = partition(A, lo, hi);
        quick_sorter(A, lo, p - 1);
        quick_sorter(A, p + 1, hi);
    }
}

void quick_sort(int A[], int size) {
    quick_sorter(A, 0, size - 1);
}
```

2.2 Test Harness

The test harness – sorting.c – will contain the main function from which all our other files are called and run from. Unlike the previous assignment, Assignment 2, it can also contain other functions we deem necessary. In the test harness, we'll be creating ab array of pseudo-random elements and testing each of the sorts.

Like the previous assignment, our test harness must support several, non-exclusive command-line options:

- -a: Utilizes all sorting algorithms
- -h: Activates HeapSort
- -b: Activates BatcherSort
- -s: Activates Shell Sort
- -q: Activates Quicksort
- -r seed: Sets the random seed to the specified value of "seed". The default seed is 13371453.
- -n size: Sets the array size to "size". The default array size is 100.
- -p elements: Outputs "elements" number of elements from the array. The default number of elements to print is 100. If the size of the array is smaller than the specified number of elements to print, the entire array will be printed.

- -H: Outputs program usage information. A reference program is provided to demonstrate the format of the output.

2.3 Gathering Statistics

To facilitate the gathering of statistics, we are provided, and must use, a small statistics module. The module itself revolves around the following structure:

```
typedef struct {
    uint64_t moves;
    uint64_t comparisons;
} Stats;
```

The module also includes functions to compare, swap, and move elements which are listed below:

```
int cmp(Stats *stats, uint32\_t x, uint32\_t y)
```

Compares x and y and increments the comparisons field in stats. Returns -1 if x is less than y, 0 if x is equal to y, and 1 if x is greater than y.

```
uint32_t move(Stats *stats, uint32_t x)
```

"Moves" x by incrementing the moves field in stats and returning x. This is intended for use in Insertion Sort and Shell Sort, where array elements aren't swapped, but instead moved and stored in a temporary variable.

```
void swap(Stats *stats, uint32_t *x, uint32_t *y)
```

Swaps the elements pointed to by pointers x and y, incrementing the moves field in stats by 3 to reflect a swap using a temporary variable.

```
void reset(Stats *stats)
```

Resets stats, setting the moves field and comparisons field to 0. It is possible that you don't end up using this specific function, depending on your usage of the Stats struct.

3 Testing

To test our sorting algorithms, we sill sort an array of pseudo-random numbers generated using the "random()" function. The program must bit-mask the pseudo-random numbers to fit within 30 bits using bit-wise AND and the test harness must be capable of testing the sorts with array sizes up to the computer's memory limit. Additionally, the program must have no memory leaks and must pass "valgrind" cleanly with any combination of command-line options.