Assignment 3 Design Document

Nicholas Chu

January 22, 2022

1 Purpose

The purpose of this assignment is to learn and implement four different sorting algorithms using C. These four sorting algorithms are: Insertion Sort, Batcher Sort, Heapsort, and a recursive Quicksort which will be respectively implemented into the header files insert.h, batcher.h, heap.h, and quick.h. These header files will be used in sorting.c, and this file will contain the testing for the sorting algorithms. An included file to take note of when working on the assignment is the stats.h file, which is a provided file that will be used to edit input arrays inside the sorting functions and keep track of statistics relating to the sorting algorithms. The functions in stats.h are:

- cmp(Stats *stats, uint32_t x ,uint32_t y): Compares x and y. Returns -1 when x is less than y, 1 when x is greater than y, and 0 when anything else. It also counts the number of compares that took place during the sort.
- move(Stats *stats, uint32_t x): Returns what is being moved, x, and tracks number of moves during the sort.
- swap(Stats *stats, uint32_t *x, uint32_t *y): Swaps x and y values and counts the number of swaps made during the sort.
- reset(Stats *stats): Resets all the stats tracked by stats.h, used when starting to track a new sort.

2 Sorting

Below is the pseudo-code for the implementation of the sorting algorithms, along with a brief explanation of what each one does. The pseudo-code is mostly based on what is given in the assignment document. Another necessary file to explain is the provided stats.h file, which contains the functions to edit the input arrays in the sorting functions and keeps track of statistics regarding the changes, tracking the number of swaps, compares, and moves.

The function will take an array of data and then sort it, and does not need to return anything. This means when declaring the function it should be void, and for the parameters it should take the pointer to the functions in stats.h, the pointer to the array being used as input, and the length of the array. The reason why the parameter needs to be a pointer to the array is because we want to call-by-reference, meaning we want to directly edit the list and not just a copy. Due to the fact that C can not normally take an array as a parameter function.

2.1 Insertion Sort

This definition of this function will be located in insert.c and implemented with insert.h. An Inserting Sort algorithm works by going through every element in a set of data and taking each element, comparing it to all previous data in the set, and inserting it into the appropriate location. In this case it goes through comparing each previous element and moving the element back a position until that element is is larger that the one before it.

```
void insertion_sort(Stats *stats, uint32_t *A, uint32_t n) {

    for (k=0; k<n+1; k++){

        # Gets the current element and copies it to a temporary variable
        tmpIndex = k
        tmpElement = A[k]

        # cmp updates compare counter and returns -1 if the.
        # temporary element is less than the temporary element.
        while tmpIndex > 0 and cmp(tmpElement, A[tmpIndex-1]) == -1{
            # This loops through the previous elements
            # and inserting current element into the right place.
            A[tmpIndex] = move(A[tmpIndex - 1])
            tmpIndex--
        }

        #Swaps current value with the temporary one
        # If it is in the correct position the value won't change.
        # This is to keep it from swapping when it doesn't need to,
        # which could cause an inaccurate count for number of swaps.
        if tmpElement != A[k]{
            swap(A[tmpIndex],tmpElement)
        }
    }
    return
}
```

2.2 Heapsort

Heapsort uses the heap data structure, which uses a specialized binary tree to sort the data. It first creates the heap, takes the first element of the heap and appends it to the back for sorting, fixes the heap, and repeats until there are no more values in the heap and all the values have been sorted. There are multiple steps that need to be implemented to sort this data structure, including building the heap and fixing it.

First its important to organize the data into the heap data structure. To organize this you would need to swap the parent with the child values until all parent values are larger than their child values. The function below is used to first find the maximum child value of the parent value. Some miscellaneous things to note is that these functions are using base indexing of 1 to avoid

issues with using the parent index, k, to find the child values by using 2k to find the left child and 2k+1 for the right child when k is the first index.

```
int max_child (uint32_t *A, int firstIndex, int lastIndex){
    /* Finding the indexes for the child values. */
    leftChildIndex = 2 * firstIndex
    rightChildIndex = 2 * firstIndex + 1

    /* The if right <= last is to check to see if there is supposed */
    /* to be since heaps can have either 1 or 2 child values*/
    /* Cmp checks to see if the right child is larger */
    if rightChildIndex <= lastIndex and
    cmp(A[rightChildIndex-1], A[leftChildIndex-1]) == 1{
        return rightChildIndex
    }
    return leftChildIndex
}
```

Once we are able to find the maximum child, it is now possible to begin iterating over the array and fixing it, organizing it into the heap data structure, by swapping the child and parent values until all parent values are larger than their children and becomes a heap. This is for when you actually begin sorting and taking the top elements from the heap to add them to the end of the array to sort, because you will need to reorganize it after each time. This is taking place in the same array, the heap is in a certain range and the sorted in the

```
void fix_heap (uint32_t *A, int firstIndex, int lastIndex) {
    fixed = false
    fixParentIndex = firstIndex
    greatestChildIndex = max_child(A, fixParentIndex, last)
    /* fixParentIndex<=lastIndex//2 makes sure that the parent index is a valid one */
    while fixParentIndex <= lastIndex // 2 and fixed == false{
        if A[fixParentIndex - 1] < A[greatestChildIndex]{
            swap(A[fixParentIndex], A[great-1])
            fixParentIndex = greatestChildIndex
            greatestChildIndex = max_child(A, fixParentIndex, last)
        }
        else {
            fixed = True
        }
    }
}
```

The next two functions build and start sorting the heap. The build_heap function builds the starting heap, and then the next will sort the heap by taking the top value from the heap, adding it to a sorted array, then fixing the heap using fix_heap, and repeating.

```
void build_heap(uint32_t *A, int firstIndex, int lastIndex){
    for (parentIndex=lastIndex // 2, parentIndex > firstIndex - 1, parentIndex-=1){
```

```
        fix_heap(A, parentIndex, lastIndex)
    }
}

void heap_sort(Stats *stats, uint32_t *A, uint32_t n){
    firstIndex = 1
    lastIndex = n
    build_heap(A, fistIndex, lastIndex)
    for (leaf = lastIndex,  leaf > firstIndex, leaf-=1)
        swap(A[firstIndex-1], A[leaf-1])
        fix_heap(A, firstIndex, leaf-1)
    }
}
```

2.3 Batcher's Odd-Even Merge Sort

Batcher's method utilizes a sorting network. It works very similarly to a shell sort and first sorts a pair of elements that are far apart from one another, with the distance between the pair being called a gap. This particular version of Batcher Sort will sort the sequence by sorting each of the sub-sequences of the even indexed values and the odd indexed values. They sort using a 2^k gaps, meaning that, for example, the gaps between the indexes would be 1,2,4, and etc. To find bit length, it is necessary to include the math library and use log in order to find bit length. The two functions that will be used for this sort are comparator and batcher_sort. The comparator function swaps the values in the indexes x and y when the value at index x is larger than the one in y.

```
void comparator(uint32_t *A, int x, int y){
    if cmp(A[x],A[y]) == 1 {
        swap(A[x], A[y])
    }
}

void batcher_sort(Stats *stats, uint32_t *A, uint32_t n){
    /* Special case when length 0 */
    if n == 0{
        return
    }

    /* This is to find the indicies that are being compared */
    length = n
    bitLength = log2(n)+1
    gap1 = 1 << (bitLength -1)

    while cmp(gap,0) == 1{
        gap2 = 1 << (bitLength -1)
        r = 0
        gapDist = gap1
```

```
        while cmp(bitLenght2,0) == 1{
            for (i=0; n-d; i+=1){

                /* Partitioning the array and k sorting*/
                if (i & bitlength) == r{
                    comparator(A, i, i+gap_dist)
                }
            gapDist = gap2 - gap1
            gap2 >>= 1
            r = gap
            }
        }
        gap >>= 1
    }
}
```

2.4 Quicksort

Quicksort works by dividing the current array that is being sorted into smaller subarrays. It uses pivots, the pivot would move left from the right value and would move left, in the process, moving all of the values less than it behind the it and keeping the ones greater than it in front of it. The pivot will always be the rightmost element of the array, and at the end, once all the values higher than the pivot are moved behind the pivot, it will act as a partition and everything behind it should now be sorted. This sort uses the functions recursively for its purposes. Partition returns the index of where the partition should be, which would separate the array into its sub arrays.

```
int partition(uint32_t *A, int low, int high){
    i = low - 1
    for j = low; j < high; j+=1{
        if cmp(A[j-1], A[high-1]) == -1{
            i += 1
            swap(A[i-1], A[j-1])
        }
    }
    swap(A[i],A[high-1])
    return i+1
}

void quick_sorter(*A, int low, int high){
    if cmp(low, high) == -1{
        pivot = partition(*A, low, high)
        quick_sorter(*A, low, pivot-1)
        quick_sorter(*A, pivot + 1, high)
    }
```

```
}

void quick_sort(Stats *stats, uint32_t *A, uint32_t n){
    quick_sorter(A,1,n)
}
```

2.5 sorting.c

The file sorting.c should take exclusive command line arguments, meaning it should be able to take any number of command line options, which includes 0. For sorting.c we want the user to be able to specify the sorting algorithms they want to use, not just one or all. To do this it is best to use a set to store all the options that the user wants into a set and have the program do specific things based off the elements in the set. It would probably be best to organize these command line options into different sets just for organization and better access. Take for example the options for the sorting algorithm would be organized into their own set and would include:

- -a: Uses all sorting algorithms. Inserts h, b, i, and q into the set that tracks the command line options.
- -h: Uses Heap Sort. Inserts only itself into the set as h.
- -b: Uses Batcher Sort. Inserts only itself into the set as b.
- -i: Uses Insertion Sort. Inserts only itself into the set a i.
- -q: Uses Quicksort. Inserts only itself into the set as q.

The other options could be put into another set and would include:

- -r seed: Sets the random seed to seed by saving it to some variable, the default value that variable should be 13371453. Would add r to the set.
- -n size: Sets the array size to size by saving it to some variable, the default value of that variable should be 100. Would add n to the set.
- -p elements: Print out 'elements' number of elements from the array, the default number should print 100. Would add p to the set.
- -H: Prints out the program's usage.

After taking these inputs in, there would be switch cases that detected if the option is in the sets and set specified values and call the appropriate functions.