Assignment 3 Design

Arnav Nepal

February 6, 2023

1 Introduction

In this assignment we will be implementing several sorting algorithms. We will be implementing the Batcher, Heap, Shell, and Quick sorts. Each sorting algorithm and method uses very different strategies to compare elements and sort them. Furthermore, we are also tasked with implementing a set library for keeping track of command line options. The files we will be writing are:

- batcher.c
- shell.c
- heap.c
- quick.c
- stats.c
- sorting.c
- set.c
- Makefile

Important Note to graders: I took this class last quarter with professor Miller. Since he was using Dr. Long's material, I have working code for some of the functions, which I might utilize. This is most pertinent to shell sort and heap sort, both of which I got mostly working (I did not get quick sort working at all last quarter, and merge sort is new). However, since there seem to be slight differences to the implementation of heap sort and shell sort this quarter, I will at least do this design without looking at last quarters code in order to refresh my memory and see if I can come up with a better design. However, I might use last quarters code in my actual C files. I will cite this fact in the file if I do.

2 Design and Psuedocode

In this assignment, we have been provided a considerable amount of python code to act as psuedocode for our C program. Thus, most of my design will be based on the psuedocode given. Furthermore, my design will incorporate the following characteristics:

- all references to "move", "swap", and "cmp" are references to the stats.c file, which includes functions for moving, swapping, and comparing elements and also gathers statistics on the program

- the number of elements in each array is provided by n_elements
- since we were provided extensive python code as a basis for this assignment, the structure of my design might mirror it. However, Python can do more than C (albeit in a far far longer time frame) so I will adapt my design so it's suitable for C instead of python
- In sections where python code alone suffices with slight addition like adding semicolons, I will write down "follow python"
- for heapsort and quicksort, arrays will utilize 1 based indexing (that is, the index 1 will refer to arr[0]). This is due to how indexes are determined for heapsort, where the children of the parents index k have a value of 2k and 2k+1. This would not make sense if index 0 existed.

2.1 Shell Sort

Shell sort will the first sort I will implement because it seems relatively simple compared to the other sorts. Shell sort sorts pairs of elements that are seperated by a gap, which is defined by a gap sequence. We will be using a given gap sequence (the Pratt sequence) in gap.h.py to define our gaps (which will be stored after creation in gap.h).

```
#this code will very closely resemble the python psuedocode because
#most of what was included in the python code is almost directly
#translateable to C code

include "gap.h"
func shell_sort(*arr, n_elements):
index = 0 #index for gap
    while (true) loop:
        for loop where (i = gap[index], i < n_elements, i++)
            int j = i
            int temp = move(arr[i]) #temp var needs to access element
            while (j > gap AND cmp(temp, arr[j - gap]) < 0):
                arr[j] = move(arr[j - gap])
                j = j - gap
            arr[j] = move(temp)
    if gap[index] is == 1, break #gap gets prog. smaller, 1 is final val
    increment index
```

Changes from draft: Shell only required minor changes from the draft, most of which was due to my mistake in translating from the python psuedocode.

2.2 Heapsort

Heapsort is a sorting mechanism that utilizes heaps, which are a type of binary tree. There are 2 types of heaps, min and max heaps. We will be implementing a heapsort using max heap, which means that the largest element of the heap is the first node, with each of it's children being smaller. In C, heaps can be represented as an array, where the children of arr[k] is equal to arr[2k] for the left child and arr[2k+1] for the right child.

```
#include relevant files
#include stdbool
#include stats

#this func returns the value of the greater child
func max_child(arr*, int first, int last):
    int left, right = follow python code
    if right <= last AND arr[right - 1] > arr[left - 1]:
        return right
    return left

#the provided swap() function in stat.c will do the work
#of emulating parallel assignments and gathering statistics
#for number of moves
func fix_heap(arr*, int first, int last):
    bool found = 0
    int mother = first
    int great = max_child(arr, mother, last)

    while mother <= last // 2 AND found == 0:
        int temp
        if arr[mother - 1] < arr[great - 1]:
            swap arr[mother - 1] and arr[great - 1]
            mother = great
            great = max_child(arr, mother, last)
        else:
            found = 1

func build_heap(arr*, int first, int last):
    for (int i = last/2, i > first - 1, decrement i):
        fix_heap(arr, father, last)

func heap_sort(stats, arr*, n_elements):
    int first = 1
    last = n_elements
    build_heap(arr, first, last)
```

```
    int temp
    for (int leaf = last, leaf > first, decrement leaf):
        swap arr[first - 1] and arr[leaf - 1]
        fix_heap(arr, first, leaf-1)
```

2.3 Quicksort

Quicksort is a recursive sorting algorithm. It partitions arrays into 2 sub arrays based on a selected pivot, placing elements greater than or equal to the pivot on the right side and elements less than the pivot on left. This process is recursively applied into each partition, so each partition in turn partitions into smaller partitions until the array is sorted.

```
#include relevant files

#The logic of the quicksort algorithm
func partition(arr*, int low, int high):
    int i = low - 1
    for (int j = low, j < high, j++):
        if arr[j - 1] < arr[high - 1]:
            increment i
            swap arr[i - 1] and arr[j - 1]
    swap arr[i] and arr[high - 1]
    return i + 1

#the main, recursive, part of quick sort. No need for a "base case"
#since if the first bool is false than array is sorted
#This function gets the partition index, then calls itself again
#based on conditions set by partition.
func quick_sorter(arr*, int low, int high
    if low < high:
        int p = partition(arr, low, high)
        quick_sorter(arr, low, p - 1)
        quick_sorter(arr, p + 1, high)

#function needed to comply with interface in quicksort header
#file.
fun quick_sorter(arr):
    quick_sorter(arr, 1, n_lements)
```

2.4 Batcher's Odd even Merge sort

Batcher's odd even Merge sort is a variation of merge sort that k-sorts the even and odd sequences within the array. While Batcher sort is capable of sorting in parallel, our implementa-

tion will be sequential.

```
#include relevant files

#this func compares x and y, swapping them if there are out of order
func comparator(arr*, int x, int y):
    if arr[x] > arr[y]:
        swap arr[x] and arr[y]

#main batcher sort code
#most of this can be directly translated from python
func batcher_sort(arr*):
    if len(arr) == 0: #establishes case for empty array
        return

    int n = n_elements
    int t = sizeof(n_elements)  #returns size in bytes of int type
                                #used by n_elements
    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, increment i):
                if (i AND p) == r:
                    comparator(arr, i, i + d)
            d = q - p
            q = q >> 1
            r = p

        p = p >> 1
```

2.5 Set Library

In this assignment, we are also tasked with implementing a slate of set related functions to keep track of command line options being passed in.

```
All the following (except set_member()) return the "Set" ADT

#returns empty set
Set set_empty():
```

```
    Set x = 0;
    return x;

#returns a set of 1's. This means max value for uint32_t, which is
#equivalent to 8 F's in hexadecimal.
Set set_universal():
    Set x = 0xFFFFFFFF

#For this func, first shift s to the right x bits so that bit X can have
#bitwise OR applied to it
Set set_insert(Set s, uint8_t x):
    Set temp = (s shift left x) OR 1
    return temp

#This function is similar to set_insert
Set set_remove(Set s, uint8_t temp):
    Set temp = (s shift left x) AND 0
    return temp

bool set_member(Set s, uint8_t x):
    if (s AND (1 shift left x) > 0)
        return true
    return false

Set set_union(Set s, Set t):
    return s OR t

Set set_interset(Set s, Set t):
    return s AND t

Set set_difference(Set s, Set t):
    Set temp = s AND NOT t
    return temp

Set set_complement(Set s):
    Set temp = set_universal AND NOT s
    return temp
```