Assignment 3 – Sets and sorting

Kevin Gallegos

CSE 13S – Spring 2023

Purpose

The purpose of this algorithm is to create sorting algorithms that will connect and run into a file called sorting.c, a test file that will run any sorting method created of the user's choice and test different inputs to the sorting algorithms from official libraries.

How to Use the Program

To use the program make sure to run sorting.c by compiling it first and using commands. This program uses command line prompts and while be used through the method of entering this in your command line "./sorting -(command)" Below is the list of every command that can be used in this program. "-i" enables insertion sort. "-s" enables shell sort. "-h" enables heap sort. "-q" enables Quicksort. "-b" enables batcher sort. "-r" set the random seed "-n (size)" set the array size "-p (element)" print out elements number of elements from the array.

Program Design

The program will use sorting.c as its main file running every other file through that file. The program will go trough every program regardless of which of ones were choosing to be worked on and output all of them and the statistics through them. Using this Method it will run multiple sorting algorithms that use a random integers as input and sort them accordingly. Insertion sort is a method that will be used by the algorithm as it compares element k with each proceeding element in descending order until its position is found. Shell sort will sort pairs of integers far from each other and close gap repetitively until they are arranged in the desired order. Heapsort will use a style similar to binary tree in which it will sort everything down having the parent node of the true be the highest value while it branches to lower variables and so on. Quick sort will create two arrays and exchange from one another until the sort is created. Batcher sort will sort everything using a sorting network. Each sort will include stats pointer as a way to record each sorts iterations of different loops or if switching between variables occurs.

Data Structures

Using random() and a mask to keep each randomly generated integer as within the confines of a bit length of 30 they will be saved into a pointer (p). The amount of integers generated will be determined by the set size by '-n' but will be defaulted to 100 in case of not setting one through command line. Insertion sort goes through one element at a time from *A (the array of numbers created by random()) placing them in correct positions comparing the k-th element with each proceeding element in descending order. array A is passed through this sorting and does not return anything, rather sorts the array and continues to the next line after the function is executed.

Algorithms

```
insert sort algorithm
    for k(1) in range of array size
        j = k
        temp = a[k]
        while j > 0 and temp array[j-1]
            array[j] = array{j-1}
                array[i] = temp
```

```
shell sort algorithm
    for x in gaps
        for i(x) in range of array size
        j = k
        temp = array[k]
        while j > x and temp< array[j-gap]
            array[j] = array{j-gap}
                array[j] = temp
```

```
quick sorter algorithm
    if low < high
        p = partition
        quick_sorter( (p-1)=hi)
        quicksorter(p+1) = low
quick sort
    quick sorter(hi = array length)
partition(array lo hi)
    i = low -1
    for lo in range to hi:
        if array(j-1) < array(hi - 1)
            i += 1
            swap a{i-j} value with a{j-1} value
    swap a{i} value with a{hi-1} value
    return i +1
```

```
batcher sorter algorithm
comparator
    if array[x] > a[y]
        swap a{x} value with a{y} value vice versa
batcher:
        if length of array is 0:
            return
    n is array length
    t is array length but as bits
    p is 1 left shit bit by (t-1)
    r is 0
    d =p
    while d>0
            for i in range(0, n-d)
            comparator
    d = q-p
    q >>= 1
    r is p
```

```
p is p right shift 1
```

```
sorting.c
    initializer all vars
    allocate memory for p and copyp
    bit track is 0
    get opt:
    case 1 add 0001 to bit track with bit or
    case 2 add 0010 to track with bit or
    case 3 add 0100 to track with bit or
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- insert takes array list, stats, array length, outputs sorted array, check if it is in the correct order by comparing it with the element k 1
- shell takes an array, stats, and array length, uses gaps from gaps.h returns sorted array The gap sequence is represented by the array gaps. You will be given a gap sequence, and close the 'gap'
- quick takes array list, stats, array length, outputs sorted array moves that into a helper function called quick sorter which takes array low and high. is a divide-and-conquer algorithm. It partitions arrays into two sub-arrays by selecting an element from the array and designating it as a pivot. Elements in the array that are less than the pivot go to the left sub-array, and elements in the array that are greater than or equal to the pivot go to the right sub-array. This also done with the help of partition, taking in an array and 2 integers hi and low
- batcher takes array list, stats, array length, outputs sorted arrays. it k-sorts the even and odd subsequences of the array to be used with its company function, Comparator which takes in an array, and 2 integers x and y swaping their values if x equals y.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge.

Results

The code runs Insertion, shell, quick sort and Batcher successfully run with them fully being able to have all getopt commands run well with all sorts mentioned above. The only issue I had was getting the pipeline to pass the shell sort stats as regardless if it was spot on or over the amount of comparisons it would fail that part of the pipeline. Other than that everything runs as it should and is clang formatted. I learned how to use pointers and the different methods that could be used to create sorting functions and such. I also learned a clever way to create command line prompts using getop and bitwise functions.

Numeric results

The screenshot below shows '-a' used to show the difference of each sorting and compare how many moves and comparison they do to reach the target.

Error Handling

Passing the pipeline for shell.(see Figure 2)

```
kevin@cse13:~/cse13s/asgn3$ ./sorting -a -n 5
Insertion Sort, 5 elements, 14 moves, 13 compares
     42067670    134750049    182960600    966879077    989854347
Heap Sort, 5 elements, 0 moves, 0 compares
    966879077    182960600     42067670    989854347    134750049
Shell Sort, 5 elements, 24 moves, 152 compares
     42067670    134750049    182960600    966879077    989854347
Quick Sort, 5 elements, 12 moves, 24 compares
     42067670    134750049    182960600    966879077    989854347
Batcher Sort, 5 elements, 12 moves, 9 compares
     42067670    134750049    182960600    966879077    989854347
```

Figure 1: Screenshot of sorting.c running.

Sources used

Jessie Tutoring sections Monday 12pm-2pm, Thursday 10pm-12pm

https://youtu.be/2ybLD6_2gKM

https://youtu.be/Vch7_YeGKH4

https://youtu.be/3eJkWkHcPc8

https://www.youtube.com/@nesoacademy

```
  • Quick Sort: Stats: failed
      - [why] Number of moves and/or compares is not close enough to the reference
  • Batcher Sort: passed
  • Batcher Sort: Stats: passed
[Functionality: Sets]
  • set_empty: passed
  • set_member: passed
  • set_insert: passed
  • set_remove: passed
  • set_union: passed
  • set_intersect: passed
  • set_difference: passed
  • set_complement: passed
  • set_universal: passed
[Functionality: Test Harness]
  • All basic options implemented: passed
  • -p sets how many elements to print: passed
  • -n and -r set the array size and seed: passed
[Summary ()]
  5.0/5.0   | Files                      | passed
  2.0/2.0   | clang-format               | passed
  3.0/3.0   | Makefile                   | passed
  41.0/58.0 | Functionality: Sorting     | errored
  5.0/5.0   | Functionality: Sets        | passed
  10.0/10.0 | Functionality: Test Harness | passed
  Estimated total: ~66.0/83
```

Figure 2: Screenshot of the pipeline running.