asgn 3: DESIGN - Sets and Sorting

Eric Wang

May 7, 2023

Purpose:

In this assignment, we are to implement five sorting algorithms. Which includes Insertion Sort, Shell Sort, Heap Sort, Quicksort, as well as Batcher Sort. We are also going to write two main files as the test files to the five sorting algorithms above. Which we have to define sets of Set functions as well as writing command line options just like the previous assignment. Set functions are basically ways to do bitwise operations in C, and command line options are the same things we did for the previous assignment, except the options this time are -a -i -s -h -q -b -r -n -p -H. Where -i enables the insertion sort, -s enables the shell sort, etc.

How to use the Program:

Download the make file and every required header and C file from the repo. Run “make” in the terminal. Once you get the binary file from making, type in “./sorting” with the appropriate command line options, which is aishqbrnpH.

Notice: Since we were given the opportunity to do 4 out of the 5 sorting algorithms, I decided to omit batcher sort. So ./sorting -a works for all sorting algorithms other than batcher.

All Command Line options:
-a : Employs all sorting algorithms that your implemented.
-i : Enables Insertion Sort.
-s : Enables Shell Sort.
-h : Enables Heap Sort.
-q : Enables Quicksort.
-b : Enables Batcher Sort.
-r seed : Set the random seed to seed. The default seed should be 13371453.
-n size : Set the array size to size. The default size should be 100.
-p elements : Print out elements number of elements from the array. The default number of elements to print out should be 100.

Pseudocodes:

sorting.c
Take the number of arguments in a command line and the respective characters in the command line
Define a integer opt
While the opt is not equal to -1

```
        Switch cases for opt
                Case a
                        Execute all the commands
                Case i
                        Get value from insertion sort
                Case s
                        Get value from shell sort
                Case h
                        Get value from heap sort
                Case q
                        Get value from quick sort
                Case H
                        Print help message

insert.c
def insertion_sort(A: list):
        for k in range(1, len(A)):
                j = k
                temp = A[k]
                while j > 0 and temp < A[j - 1]:
                        A[j] = A[j - 1]
                        j -= 1
                A[j] = temp

heap.c
def max_child(A: list, first: int, last: int):
        left = 2 * first
        right = left + 1
        if right <= last and A[right - 1] > A[left - 1]:
                return right
        return left
def fix_heap(A: list, first: int, last: int):
        found = False
        mother = first
        great = max_child(A, mother, last)
        while mother <= last // 2 and not found:
                if A[mother - 1] < A[great - 1]:
                        A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
                        mother = great
                        great = max_child(A, mother, last)
                else:
                        found = True
def build_heap(A: list, first: int, last: int):
        for father in range(last // 2, first - 1, -1):
```

```
                fix_heap(A, father, last)
def heap_sort(A: list):
        first = 1
        last = len(A)
        build_heap(A, first, last)
        for leaf in range(last, first, -1):
                A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
                fix_heap(A, first, leaf - 1)

shell.c
def shell_sort(arr):
        for gap in gaps:
                for i in range(gap, len(arr)):
                        j = i
                        temp = arr[i]
                        while j >= gap and temp < arr[j - gap]:
                                arr[j] = arr[j - gap]
                                j -= gap
                        arr[j] = temp

quick.c
def partition(A: list, lo: int, hi: int):
        i = lo - 1
        for j in range(lo, hi):
                if A[j - 1] < A[hi - 1]:
                        i += 1
                        A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
        A[i], A[hi - 1] = A[hi - 1], A[i]
        return i + 1

def quick_sorter(A: list, lo: int, hi: int):
        if lo < hi:
                p = partition(A, lo, hi)
                quick_sorter(A, lo, p - 1)
                quick_sorter(A, p + 1, hi)
def quick_sort(A: list):
        quick_sorter(A, 1, len(A))
```

Results:

```
galliumnitride@ewang:~/cse13s/asgn3$ ./sorting -a -p 0
Insertion Sort, 100 elements, 2741 moves, 2638 compares
Heap Sort, 100 elements, 1755 moves, 1029 compares
Shell Sort, 100 elements, 3025 moves, 1575 compares
Quick Sort, 100 elements, 1053 moves, 640 compares
Batcher Sort, 100 elements, 0 moves, 0 compares
galliumnitride@ewang:~/cse13s/asgn3$ ./sorting-x86 -a -p 0
Insertion Sort, 100 elements, 2741 moves, 2638 compares
Heap Sort, 100 elements, 1755 moves, 1029 compares
Shell Sort, 100 elements, 3025 moves, 1575 compares
Quick Sort, 100 elements, 1053 moves, 640 compares
Batcher Sort, 100 elements, 1209 moves, 1077 compares
```

	moves	compare
heap	1755	1029
insert	2741	2638
quick	1053	640
shell	3025	1575

This is the result where there are 0 elements printed compare to the binary results. As shown in the screenshot, every sorting algorithm is working as intended other than batcher sort since I omitted the algorithm.

I got the data for the table above from typing ./sorting -a -p 0, the default number of elements is 100.

Insertion sort works the best when there's a small number of inputs. Shell sort works well with a medium sized input. As we can see from the table, we can see that out of the 4 sorting algorithms, shell and insertion sort took the most number of moves and compares in order to have the arrays sorted. But if an array only has 10 elements, insertion sort will perform better than other sorting algorithms. Quick sort sorts quick (haha) when there's a large quantity of inputs. Same with heap sort.

Bugs:
There's a known formatting bug when you print a really small size and a small number of elements. The code doesn't break, though.

References:
asgn3.pdf from Dr. Kerry Veenstra