CSE13s Winter 2022
Assignment 3: Sorting - Putting your affairs in order
Theodore Ikehara, tikehara@ucsc.edu
CSE 13S - Winter 2022
Due: January 26th at 11:59 pm
Design Document

1. Introduction
   I. About:

   This assignment is going to be focusing on one of the most important and common tasks in computer science, sorting. Sorting is important as data that is sorted is easy to read and deal with. Sorted data is way more useful to humans than unsorted data. However, there is not only one way to sort items. There exists numerous ways to sort data and many algorithms written with the sole intention of sorting data. Each sorting algorithm has its own merits and demerits, whether it be easier to implement or sort really quickly or be very memory efficient or combination of all factors is dependent on the sorting algorithm. In this assignment we are going to explore four different sorting algorithms and collect data we can later conduct analysis on and determine and observe the merits and demerits of these sorting algorithms. Of course there exists more than just four algorithms but we will be exploring: insertion sort, heapsort, quicksort, and batcher sort. All of these sorts work very differently and we will see in this assignment why certain sorts do what they do and their approach to the idea of sorting.

   II. Time Complexity:

   In this assignment one of the most important factors we will be considering is the time complexity of the sorting algorithm. What this means is the speed or efficiency of the algorithm. We will be observing the time complexities as shown below:

   Insertion Sort: $O(n^2)$
   Heapsort: $O(n\,log(n))$
   Quicksort: $O(n\,log(n))$

Batcher's Odd-Even Merge Sort: $O(n\ log(n)^2)$

These are all the average time complexities that we will observe with these sorts with big O notation. Looking at these time complexities we can infer or hypothesise that insertion sort no doubtedly will be the slowest sort at all intervals. Heapsort and quicksort have the same average time complexity, but heap sort will perform the same given any array of items to sort where quicksort will perform worst given the worst case scenario. Batcher's odd-even merge sort will perform better on smaller data and will start decreasing in performance where heapsort and quicksort will perform better the larger the data array. In computer science we mostly only care about the average case in time complexity as we just need to know how it will perform on the average circumstance.

2. The plan what I will be writing
   I. batcher.c

      In this file we will be implementing batcher sort. This sort differs from the other sorts as this is a sorting network. What this means is that it sorts multiple things at the same time or in parallel. Usually sorting networks like this one could only accept input with the powers of 2. This is why in order to remedy this issue we will be applying Knuth's modification to batcher sort to make it able to take any input. We will be implementing 2 functions one to compare and one to actually do the sorting. In order to implement this sort in the main sorting routine we will need to have a nested while loop with a for loop. We will compare and fix loop increments in the for loop. Direct implementation will be shown in the pseudo code.

   II. insert.c

      Insertion sort is the least efficient but easiest to conceptualize compared to the other sorting algorithms. Insertion sort increments through the elements that are being considered and individually places them in the correct location. This is amongst the oldest sorting algorithms. This sort will require a nested loop as one to increment through the elements and one to place the elements in the correct location.

III. heap.c

Heapsort will require us to build a heap and create a max child heap. This will be implemented with 4 functions as we will need to build the heap and assign the max childs. And the other function to actually perform the sorting routine. This is one of the best sorting algorithms that we will be implementing in this assignment.

IV. quick.c

Quicksort will require us to create a partition and sort individual parts of the array with recursion, meaning that it tries to split up the task amongst itself and sort each part individually. When implemented well this is the fastest known algorithm using comparisons. This will be the only algorithm that uses recursion that we will be implementing in this assignment.

V. sorting.c

In this section we will be implementing the tests for the assignment. We will be including all the test cases with a random seed so that the same numbers can be generated multiple times. Here is where we will also be implementing getopt.

3. Pseudo code (python)

Insertion Sort:

```
# Insertion sort

def insertion_sort(A: list): # This is passing in the list

    # elements = len(A)
    # moves = 0
    # compares = 0

    # This loop iterates the list
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        # comparisons are done here
        compares += 2
        while j > 0 and temp < A[j - 1]:
            compares += 2

            A[j] = A[j - 1]
            j -= 1
        #switching done here
        A[j] = temp
        moves += 1

    print('elements: ', elements, 'moves: ', moves, 'compares: ', compares)
    return A

##############################################################################
```

Quick Sort:

```
# Quicksort

# creating the partitions
def partition(A: list, lo: int, hi: int):
    i = lo - 1

    # all the switching in the array positions are done here
    for j in range(lo, hi):

        compares = compares + 1

        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]

            moves = moves + 1

    A[i], A[hi - 1] = A[hi - 1], A[i]

    return i + 1

# This is incharge of running the partitions and recursion
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        # spliting the work here
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)


# This is the actual sorting meathod

def quick_sort(A: list):
    elements = len(A)
    compares = 0
    moves = 0
    quick_sorter(A, 1, len(A))

    return A
```

Heapsort:

```python
# Heapsort

# Heap maintenance

# This is where the max child is created
# This sort uses the child and parent relation
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1

    # Compares
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

# This function fixes the heap so that the parent and child relation is
# maintained
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]:
            # swaps done here
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found True

# heapsort main part
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
    lsat = 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)


```

Batcher Sort:

```python

# Batcher sort
def comparator(A: list, x: int, y: int):
    # compares here
    if A[x] > A[y]:
        # swaps
        A[x], A[y] = A[y], A[x]

# This is the main function of the sorting routine
# splitting the sorting tasks
def batcher_sort(A: list):
    if len(A) == 0:
        return

    n = len(A)
    t = n.bit_length()
    p = 1 << (t - 1)

    while p > 0:
        q = 1 << (t - 1)
        r = 0
        d = p

        while d > 0:
            for i in range(0, n - d):
                if (i & p) == r:
                    comparator(A, i, i + d)

            d = q - p
            q >>= 1
            r = p

        p >>= 1

```