Chad Hirsch
CS4310

CS4310 Assignment 4

Question 1: A cable company needs to decide where to put a new service point in a small town where all houses are along one street. Let us view this street as a line and the houses on it have real numbered coordinates $\{x_1, x_2, \dots, x_n\}$. The service point, p, should be chosen to minimize the total length of cables to all houses, that is, $\sum_{i=1}^{n} |p - x_i|$. Describe an efficient algorithm for finding the optimal location of the service point.

```
minimalDistance(coordinates):
        QuickSort(coordinates)
        Median = coordinates.length /2
        return coordinates[median]
```

Running Time: O(nlogn)

Question 2: A sorting algorithm is said to be ***stable*** if it preserves the initial ordering of elements with the same key. Stability of the underlying algorithm is essential for the radix sort algorithm. For the following sorting algorithms, check whether the algorithm is stable or not. If not, can you make it stable without affecting the asymptotic time complexity of the algorithm?

Bubble sort, insertion sort, selection sort, quick sort, merge sort, heap sort.

For the following sorting algorithms, we will sort a lists of tuples by the first value in the tuple.

Bubble sort is stable.

(3,a), (2,b), (3,c), (1,d)
(2,b), (3,a), (3, c), (1,d)
(2,b), (3,a), (1,d), (3,c)
(2,b), (1,d), (3,a), (3,c)
(1,d), (2,b), (3,a), (3,c)

Insertion sort is stable.

(3,a), (2,b), (1,d), (3,c)
(2,b), (3,a), (1,d), (3,c)
(1,d), (2,b), (3,a), (3,c)

Merge sort is stable.

(3,a), (1,b), (5,c), (4,d), (7,e), (2,f), (3,g), (6,h)
(1,b), (3,a), (4,d), (5,c), (2,f), (7,e), (3,g), (6,h)
(1,b), (3,a), (4,d), (5,c), (2,f), (3,g), (6,h), (7,e)
(1,b), (2,f), (3,a), (3,g), (4,d), (5,c), (6,h), (7,e)

Selection sort is not stable.

(3,a), (2,b), (3,c), (1,d)
(1,d), (2,b), (3,c), (3,a)

Chad Hirsch
CS4310

Selection sort can be made stable by instead of swapping the minimum value with the current index, it is inserted there instead, shifting the rest down. This insertion can be done in O(n) time. So the time complexity would still be O(n^2).

(3,a), (2,b), (3,c), (1,d)
(1,d), (3,a), (2,b), (3,c)
(1,d), (2,b), (3,a), (3,c)

Quicksort is not stable.

(3,a), (2,b), (3,c), (1,d), pivot = (3,c)
(2,b), (1,d), (3,c), (3,a), pivot = (1,d)
(1,d), (2,b), (3,c), (3,a)

Quicksort can be made stable by using O(n) space. We create 2 arrays of size n/2. For a given pivot, one array stores all the elements less than the pivot and the other the elements greater than the pivot. If two elements have the same value, we compare their index. If the index is less than the pivot, then it goes into the array with smaller values. Otherwise, it goes in the array with larger values. We then recursively call quicksort on each array and combine in the end.

(3,a), (2,b), (3,c), (1,d), pivot = (3,c)
(3,a), (2,b), (1,d), (3,c), pivot = (2,b)
(1,d), (2,b), (3,a), (3,c)
(1,d), (2,b), (3,a), (3,c)

Heap sort is not stable.

(1,a) After removing (1,a) (2,c)
(2,b) (2,c) → (2,b)

array representation:
(1,a) (2,b), (2,c)

final array representation:
(1,a), (2,c), (2,b)

Heap sort can be turned into a stable algorithm by keeping track of the index of each element. This index can be a 3rd value in our tuple. For example, if two elements have the same value, then the one with the smaller index would be "smaller" than the other.

(1,a,0) After removing (1,a) (2,b,1)
(2,b,1) (2,c,2) → (2,c,2)

array representation:
(1,a,0) (2,b,1), (2,c,2)

final array representation:
(1,a,0), (2,b,1), (2,c,2)

Question 3: Let $S = \{a, b, c, d, e, f, g\}$ be a collection of objects with the following benefit-weight value s: a: (12,4). b: (10,6), c: (8,5), d: (11,7), e: (14,3), f: (7,1), g: (9,6). What is an optimal solution to the fractional knapsack problem for S, assuming we have a sack that can hold objects with total weight of 18? Show your work.

1. Calculate ratio (benefit/weight) for all values
    a. a: 3, b: 1.67, c: 1.60, d: 1.57, e: 4.66, f: 7, g: 1.5

2. Sort by ratio
    a. f: 7, e: 4.66, a: 3, b: 1.67, c: 1.60, d: 1.57, g: 1.5
3. Add entire object to knapsack from highest ratio to lowest until the entire object cannot be added to the knapsack. From there, take the largest fraction of that object that can fit into the knapsack and add it. Once we get here we are done.
    a. Add f to knapsack
        i. Knapsack value: 7, remaining weight: 17
    b. Add e to knapsack
        i. Knapsack value: 21, remaining weight: 14
    c. Add a to knapsack
        i. Knapsack value: 33, remaining weight: 10
    d. Add b to knapsack
        i. Knapsack value: 43, remaining weight: 4
    e. Add 4/5 of c to knapsack
        i. Knapsack value: 49.4, remaining weight 0
4. The optimal solution was to add f, e, a, b, and 4/5 of c to the knapsack to get a value of 49.4

Question 4: Given an unordered sequence of n integers, describe an efficient algorithm for finding k items in the middle of an ordered version of the sequence. Give pseudo code of your algorithm.

```
middleKelements(input):
    for i = 0 to (n-k)/2  - 1:
        minHeap.insert(input[i])     // in the end min heap stores (n-k)/2 largest elements
        maxHeap.insert(input[i])    // in the end max heap stores (n-k)/2 smallest elements
    for i = (n-k)/2 to n -1:
        if maxHeap.max > input[i]
            maxHeap.insert(input[i])
            maxHeap.removeMax()
        if minHeap.min < input[i]
            minHeap.insert(input[i])
            minHeap.removeMin()
    remove all elements from the min and max heaps and store them in Hash Set, S
    output = new list of size k
    for i = 0 to n-1
        if input[i] is not in S
            output.insert(input[i])
    return output
```

O(n-k) to build both heaps, O(k log(n-k)) for finding all the (n-k)/2 largest/smallest elements. O(n) for finding the k middle elements. The overall runtime would be O(k log(n-k) + n + (n-k)) which simplifies to O(k log(n-k) + n)