CS 61B Disc 10: MT2 Review Fall 2014

1 Sometimes Sort of Sorted (from 61BL SU2014 Final)

Let n be some large integer, $k < \log_2 n$. For each of the input arrays described below, explain which sorting algorithm you would use to sort the input array the fastest and why you chose this sorting algorithm.

Make sure to state any assumptions you make about the implementation of your chosen sorting algorithm. Also specify the big-Oh running time for each of your algorithms in terms of k and n. Your big-Oh bounds should be simplified and as tight as possible.

(a) Input: An array of n `Comparable` objects in completely random order

Sorts: Quicksort, merge sort, heap sort, tree sort

Runtime: $O(n \log n)$

Explanation: Merge sort and heap sort are always $O(n \log n)$. For quicksort, we can easily choose a good pivot for randomly ordered inputs. For tree sort, the resulting tree will be fairly balanced in the average case.

We needed to use a comparison-based sort because input contains `Comparable` objects. Bubble, insertion, and insertion sort are inefficient compared to the four listed above. Runtime should not include k.

(b) Input: An array of n `Comparable` objects that is sorted except for k randomly located elements that are out of place (that is, the list without these k elements would be completely sorted)

Sort: Insertion sort

Runtime: $O(nk)$

Explanation: For the $n-k$ sorted elements, insertion sort only needs 1 comparison to check that it is in the correct location (larger than the last element in the sorted section). The remaining k out-of-place elements could be located anywhere in the sorted section. In the worst case, they would be inserted at the beginning of the sorted section, which means there are $O(n)$ comparisons in the worst-case for these k elements. This leads to an overall runtime of $O(nk+n)$, which simplifies to $O(nk)$.

It is a tempting mistake to say the runtime was $O(n)$ or $O(n^2)$. $O(n)$ is incorrect because it underestimates the number of comparisons required for the out-of-place elements. $O(n^2)$ is incorrect because it ignores the fact that only 1 comparison is needed for already sorted elements.

Also, it is incorrect to equate k with $\log_2 n$. It was only given that $k < \log_2 n$.

(c) Input: An array of n `Comparable` objects that is sorted except for k randomly located pairs of adjacent elements that have been swapped (each element is part of at most one pair).

Sort Option 1: Bubble sort with optimization

Runtime 1: $O(n)$ or $O(n+k)$

Explanation 1: The optimization for bubble sort has it stop after a complete iteration of no swaps occurring. It stops once the list is sorted, not after it has run through n iterations for a runtime of $O(n^2)$.

It takes one iteration of bubble sort to swap the k randomly reversed pairs of adjacent elements then another iteration before optimized bubble sort stops due to no swaps. If you count each swap as taking $O(1)$ time, the total runtime is $O(2n+k)$, which simplifies to $O(n)$.

Sort Option 2: Insertion sort

Runtime 2: $O(n)$ or $O(n+k)$

Explanation 2: Insertion sort requires 1 comparison for the $n-k$ sorted elements then requires 2 comparisons for the second element in each of the k pairs. This leads to a runtime of $O(n+k)$, which simplifies to $O(n)$.

(d) Input: An array of n elements where all of the elements are random `ints` between 0 and k

Sort Option 1: Counting sort

Runtime 1: $O(n)$ or $O(n+k)$

Explanation 1: Counting sort involves initializing an array of size k, then going through n elements while incrementing numbers in the array. Recovering the sorted list requires going through k buckets and outputting n numbers. This is a total runtime of $O(2n+2k)$, which simplifies to $O(n+k)$ or $O(n)$.

Sort Option 2: Radix sort

Runtime 2: $O(n)$

Explanation 2: Radix sort is $O((n+b)d)$, where b is the number of buckets used and d is the number of digits representing the largest element. (In this case, the largest element was k.)

Because the input array is composed of Java `ints`, we can say that b is equal to 2 and d is equal to 32 because Java `ints` are 32-bits long, and each bit can be a 0 or 1. Thus, because b and d are constants, the runtime for radix sort on Java `ints` is $O(n)$.

2 Up (from 61BL SU14 MT2)

```
import java.util.*;
public class Tree {
    private TreeNode root;
    public Tree(Object rootItem) {
        this.root = new TreeNode(rootItem);
    }
    private class TreeNode {
        private Object item;
```

```
        private ArrayList<TreeNode> children;
        public TreeNode(Object inputObj) {
            this.item = inputObj;
            children = new ArrayList<TreeNode>();
        }
        public void addChild(Object childObj) {
            children.add(new TreeNode(childObj));
        }
    }
}
```

(a) Write a method in the provided `Tree` class that returns the items of the tree in reverse BFS order. That is, it returns an `ArrayList` of items such that for all positive `i`, all items of nodes at depth `i + 1` come before all items of nodes at depth `i` in the returned `ArrayList`. Items of nodes at the same depth level can be in any order in the list. While not necessary, you are allowed to write up to one helper method.

```
public ArrayList<Object> reverseBFS() {
    // Using LinkedList as queue
    LinkedList<TreeNode> fringe = new LinkedList<TreeNode>();
    Stack<Object> nodeItems = new Stack<Object>();

    // Check if root is null
    if (root == null) {
        return new ArrayList<Object>();
    } else {
        fringe.add(root);
    }

    // Add items to a stack in BFS order
    while (fringe.peek() != null) {
        TreeNode currentNode = fringe.poll();
        nodeItems.push(currentNode.item);
        ArrayList<TreeNode> nodeChildren = currentNode.children;
        for (TreeNode nodeChild : nodeChildren) {
            fringe.add(nodeChild);
        }
    }

    // Remove items from stack in reverse BFS order and
    // add them to the ArrayList that is returned
    ArrayList<Object> returnList = new ArrayList<Object>();
    while (!nodeItems.empty()) {
        returnList.add(nodeItems.pop());
    }

    return returnList;
}
```

(b) What is the big-O running time of your algorithm from part (a)? Your answer should be as tight of a bound as possible.

$O(n)$ where n is the number of nodes in the tree.

3 HashMap Modification (from 61BL SU2010 MT2)

(a) When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain?

☐ Always ☒ Sometimes ☐ Never

It is possible that the new Key will end up colliding with the old Key. Only in this rare situation will we be able to retrieve the value. It is very bad to modify the Key in a Map because we cannot guarantee that the data structure will be able to find the object for us if we change the Key.

(b) When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain?

☒ Always ☐ Sometimes ☐ Never

You can safely modify the value without any trouble. If you reference the value that you put in the tree, the changes will be reflected.

4 isMaxHeap? (based on 61B SP1996 MT3)

Given an array, the `isMaxHeap` function determines whether the array represents a valid max heap.

```
public static boolean isMaxHeap(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        int left = i * 2 + 1;
        int right = i * 2 + 2;
        if (arr.length > left && arr[i] < arr[left]) {
            return false;
        }
        if (arr.length > right && arr[i] < arr[right]) {
            return false;
        }
        if (arr.length <= right) {
            return true;
        }
    }
    return true;
}
```

```
assert isMaxHeap(new int[]{264, 38, 79, 2, 37, 77, -1, -2});
assert !isMaxHeap(new int[]{264, 266, 79, 2, 37, 77, -1, -2});
assert !isMaxHeap(new int[]{264, 38, 79, 2, 37, 77, -1, 3});
```