CS61B Fall 2015 Guerrilla Section 3 Worksheet SOLUTIONS

8 November 2015

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is.* Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 Which Sort to Use

For each of the following scenarios, choose the best sort to use and explain your reasoning.

(a) The list you have to sort was created by taking a sorted list and swapping N pairs of adjacent elements.

Insertion sort since a list created in such a manner will have at most N inversions. (Recall that insertion sort runs in $\Theta(N+K)$ time, where K is the number of inversions.) Bubble sort is also correct since that would also take $\Theta(N)$ time on such a list.

(b) The list you have to sort is the list of everyone who took the US Census and you want to sort based on last name.

MSD radix sort. LSD radix sort would work, but we would have to pad each last name with extra characters so that all names were the same length.

(c) You have to sort a list on a machine where swapping two elements is much more costly than comparing two elements (and you want to do the sort in place).

Selection sort since in its most common implementation, selection sort performs N swaps in the worst case, whereas all other common sorts perform $\Omega(N \log N)$ swaps in at least some cases.

(d) Your list is so large that not all of the data can fit into RAM at once. As in, at any given time most of the list must be stored in the external memory, where accessing it is much slower.

The answer to this problem was actually mentioned in lecture: merge sort since its divide-and-conquer strategy works well with the restriction on only being able to hold a portion of the list in RAM at any given time. Look up external sorting for more details.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

2 Balanced Search Tree Mechanics

Solutions for this problem: coming soon.

(a) Draw the result of inserting the following numbers into a 2-4 tree (in the order that they are listed): 1, 3, 5, 7, 2, 4, 8, 9, 10, 0, -1. For the purposes of this exercise, when a node becomes overfull, promote the second element from the left.

(b) Now draw the result of removing the following numbers from the last 2-4 tree from above: -1, 4, 0.

(c) Draw a red-black tree that corresponds to the last 2-4 tree you drew in part (a).

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

3 Identify the Sort

Match up the sorting algorithms I–VI to the sequences a–f, which represent an array being sorted at some intermediate steps in the computation (not necessarily consecutive or evenly spaced). In each case, the original array to be sorted consists of the integers

`5103 9914 0608 3715 6035 2261 9797 7188 1163 4411`

	Intermediate Steps	Algorithm
a.	2261 4411 5103 1163 9914 3715 6035 9797 0608 7188 6035 5103 1163 7188 2261 4411 0608 3715 9797 9914	V
b.	5103 9914 0608 3715 2261 6035 7188 9797 1163 4411 0608 2261 3715 5103 6035 7188 9797 9914 1163 4411	III
c.	0608 1163 2261 3715 4411 5103 6035 7188 9914 9797 0608 1163 2261 3715 4411 5103 6035 7188 9797 9914	VI
d.	0608 1163 5103 3715 6035 2261 9797 7188 9914 4411 0608 1163 2261 3715 6035 5103 9797 7188 9914 4411	IV
e.	9797 7188 5103 4411 6035 2261 0608 3715 1163 9914 4411 3715 2261 0608 1163 5103 6035 7188 9797 9914	I
f.	5103 0608 3715 2261 1163 4411 6035 9914 9797 7188 0608 2261 1163 3715 5103 4411 6035 9914 9797 7188	II

I. Heap sort

II. Quicksort

III. Merge sort

IV. Selection sort

V. LSD radix sort

VI. MSD radix sort

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

4 Trie Mechanics

In this exercise we will implement the insert method for a Trie. Specifically, we are assuming that all strings are over an alphabet of the four characters 'A', 'C', 'T', and 'G' and that the children of each node in the trie can thus be stored in an array of length 4. Fill in the insert methods below in accordance with the comments.

```java
public class TrieNode {

    /** Initializes the character of this TrieNode to VAL and the
     *  child pointer array to be an empty array of size 4. */
    public TrieNode(char val) {
        this.val = val;
        isWord = false;
        children = new TrieNode[4];
    }

    /** Inserts S into this Trie assuming that this node is the root.
     *  Returns true iff S was already in the Trie. This method assumes
     *  that s contains only the characters A, C, T, and G. */
    public boolean insert(String s){
        return insert(s, 0);
    }

    /** Inserts S into this Trie assuming that this node is an internal
     *  node of depth i + 1 in the Trie (where the root is at depth 0).
     *  Returns true iff S was already in the Trie. */
    public boolean insert(String s, int i) {
        if (i == s.length()) {
            boolean rtn = isWord;
            isWord = true;
            return rtn;
        }
        char cur = s.charAt(i);
        if (getChild(cur) == null) {
            children[getIndex(cur)] = new TrieNode(cur);
        }
        return getChild(cur).insert(s, i + 1);
    }

    /** Returns the child of this node corresponding to the character NEXT. */
    public TrieNode getChild(char next) {
        return children[getIndex(next)];
    }

    /** Returns the index in the child pointer array corresponding to the
     *  character A. */
    private static int getIndex(char a) {
        switch (a) {
        case 'A':
            return 0;
        case 'C':
            return 1;
        case 'T':
            return 2;
        case 'G':
            return 3;
        default:
            return -1;
        }
    }

    /** The children of this node. */
    private TrieNode[] children;
    /** The character on the edge leading to this node. */
    private char val;
    /** True if and only if this node represents a complete word in the Trie. */
    private boolean isWord;
}
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 Merging Many Sorted Lists (CS61B Fall 2013 Midterm 2)

Suppose that we have an array of `Iterator<String>` objects, where each `Iterator` is guaranteed to deliver its Strings in sorted order. We would like to form a single sorted list containing all items produced by these iterators. Describe how to do this efficiently. You do not need to give actual Java code, but be precise and clear in your description. Give the running time of your algorithm in terms of k, the number of iterators, and N, the total number of strings in the final list.
Call the iterators S_i, for $0 \leq i < k$. Create a priority queue containing pairs (E_i, S_i), where E_i is an item from S_i and S_i is positioned just after that element. The queue is ordered by the values E_i, with the smallest having highest priority. Repeatedly

- Remove an item (E_j, S_j) from the queue.
- Add E_j to the output list.
- If S_j has more elements in it, fetch a new value E'_j from S_j, and then add (E'_j, S_j) to the queue.

Running Time: At any time the heap has at most k elements, so removing the smallest value takes $O(\log k)$ time. Since we do this N total times, the running time is $O(N \log k)$.

STOP!
Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 Local Extrema

Given an array of integers, define a local maximum to be an element of the array that is larger than the adjacent integers in the array. A local minimum is defined similarly. Given an array of N integers, devise an $O(N \log(N))$ algorithm to rearrange the array so that every element is either a local maximum or a local minimum.
Simply sort the entire list using any of the $\Theta(N \log N)$ sorting algorithms we learned (i.e. merge sort, heap sort) and then break the list in half and interleave the two halves.

7 Partial Matches

Given a list of N input words all of length at most k and M query words, we would like to find, for each query word, the number of input words that match the first $k/2$ letters of the query word. Describe an algorithm that accomplishes this and give its running time as a function of N, M, and k.

Solution 1: Use a trie where every node of the trie tracks how many of its descendants are complete words. Insert all N input words into such a trie. This takes $O(Nk)$ time (assuming a constant alphabet size). Then for each query word, find the node in the trie corresponding to the word's first $k/2$ letters and output the number stored in that node (and output 0 if there is no such node). This takes $O(k)$ time for each query word. So the entire algorithm takes $O((N + M)k)$ time (note that in the best case, all of the query words begin with some letter that is not in the trie at all, in which case this solution will run in $O(Nk + M)$ time).

Solution 2: Use a HashMap where the keys are strings of length $k/2$ and the values are integers representing how many of the input words begin with those $k/2$ letters. For each input word x, look up the length $k/2$ prefix of x in the HashMap. If it is present, increment the corresponding value. If it is not present, add it as a key with corresponding value of 1. Then for each query word, check if its length $k/2$ prefix is present in the HashMap. If so, output the corresponding value. If not, output 0. In the worst case, inserting the input words into the HashMap takes $\Theta(N^2k)$ (because in the worst case all input words hash to the same bucket and comparing the inserted word to all previously inserted words takes up to $O(Nk)$ time because we have to compare the first $k/2$ letters of each word) and similarly, looking up all the query words takes $\Theta(MNk)$ time. So in the worst case this solution will take $\Theta((N + M)Nk)$ time. If we assume however that the keys are distributed uniformly among the buckets then this solution is also $\Theta((N + M)k)$ time.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

8 Analyze mergeAll (CS61BL Summer 2014 Midterm 2)

We would like to write a method called `mergeAll` that merges together N sorted lists with M elements each. To do so, we will use a method called `merge` that returns the result of merging two sorted linked lists. If one of the lists has length l and the other has length r then the merge method will run in $O(l + r)$ time. Below are two versions of the `mergeAll` method. For each one, give the running time in terms of N and M.

```java
public static LinkedList<Integer> mergeAll(ArrayList<LinkedList<Integer>> lists) {
    for (int i = 1; i < lists.size(); i++) {
        lists.set(0, merge(lists.get(0), lists.get(i)));
    }
    return lists.get(0);
}
```

Running Time:
This solution repeatedly merges the i^{th} list in `lists` with the first list in `lists`. So on the i^{th} iteration, the first list in `lists` has length Mi and is being merged with a list of length M. This takes $O(Mi + M)$ time. So the total running time is $O(\sum_{i=1}^{N} M(i+1)) = O(MN^2)$.

```java
public static LinkedList<Integer> mergeAll(ArrayList<LinkedList<Integer>> lists) {
    while(lists.size() > 1) {
        ArrayList<LinkedList<Integer>> newLists = new ArrayList<LinkedList<Integer
            >>();
        int numLists = list.size();
        for(int i = 0; i < numLists; i++) {
            if (numLists % 2 == 1 && i == numLists - 1) {
                newLists.add(lists.get(i));
            } else {
                newLists.add(merge(lists.get(i), lists.get(i + 1)));
                i++;
            }
        }
        lists = newLists;
    }
    return lists.get(0);
}
```

Running Time:
This solution is basically just merge sort. First we merge adjacent pairs of lists of length M to form a new list of about $N/2$ lists all of length at most $2M$. We then repeat this to get a list of about $N/4$ lists of length at most $4M$, and so on. So on the j^{th} iteration of the while loop, we have a list of $O(N/2^j)$ lists of length at most $2^j M$. Merging adjacent pairs of these takes $2^j M + 2^j M = 2^{j+1} M$ time per pair, so mergin all the adjacent pairs takes $O(\frac{N}{2^j} 2^{j+1} M) = O(NM)$ time. Since there are $O(\log N)$ total iterations, the total running time is $O(MN \log N)$.