UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B
Fall 2013

P. N. Hilfinger

Test #2 Solutions

Problems 1 and 2 refer to the following rather inept search structure, `AutoTree`, and two functions, `makeTree` and `repeated`, which use it.

```
public class AutoTree {
    private int _val;
    private AutoTree _left;
    private AutoTree _right;

    private AutoTree(int val,
                     AutoTree left,
                     AutoTree right) {
        _val = val;
        _left = left; _right = right;
    }

    public static singleton(int val) {
        return new AutoTree(val, null, null);
    }

    /** Insert VAL into me destructively. */
    public void add(int val) {
        if (_val <= val) {
            _left =
               new AutoTree(_val, _left,
                            null);
            _val = val;
        } else if (_left != null) {
            _left.add(val);
        } else {
            _left =
                new AutoTree(val, _left,
                             null);
        }
    }

    /** Return true iff I contain a value
     *  equal to ITEM */
    public boolean contains(int item) {
        if (this._val == item) {
            return true;
        } else {
            return (_left != null
                    && _left.contains(item))
                || (_right != null
                    && _right.contains(item));
        }
    }
}
```

```
/** Return an AutoTree containing
 *  the items in A, assuming A.length>0. */
AutoTree makeTree(int[] A) {
    if (A.length == 0) {
        return null;
    }
    AutoTree result =
        AutoTree.singleton(A[0]);
    for (int i = 1; i < A.length; i += 1) {
        result.add(x);
    }
    return result;
}

/** Creates an AutoTree containing the items
 *  in the first half of A.  Then returns true iff
 *  any items in the second half of A are contained
 *  in the AutoTree. */
boolean repeated(int[] A) {
    if (A.length == 0) {
        return;
    }
    AutoTree result = AutoTree.singleton(A[0]);

    for (int i = 1; i < A.length / 2; i += 1) {
        result.add(A[i]);
    }

    for (int i = A.length / 2; i < A.length; i += 1) {
        if (result.contains(A[i])) {
            return true;
        }
        return false;
    }
}
```

1. [2 points] Consider the `AutoTree` class on page 2. As a function of $N = A$.`length`, what is the tightest upper bound you can find for the worst-case execution time of `makeTree`? What are inputs that would cause this worst-case behavior?

I $O(N)$;

II $O(N \log N)$;

III $O(N^2)$;

IV $O(2^N)$

Ans. $O(N^2)$

Worst case inputs:

Ans. *A sequence of inputs in descending order. The tree is left-leaning (since all right children are null). In this worst case, each input goes to the bottom of the tree, whose height grows linearly.*

2. [2 points] Let $N = A$.`length`. What is the tightest upper bound you can find for the worst-case running time of `repeated`? (**Important:** Because the instance variables and constructor of `AutoTree` are private, you should consider *only* `AutoTree`s constructed using its public methods.

I $O(N)$;

II $O(N \log N)$;

III $O(N^2)$;

IV $O(2^N)$;

Ans. *Again, $O(N^2)$. The algorithm given takes linear time to search.*

3. [3 points] The following code is intended to perform an in-place merge sort of an array, but it doesn't.

```
private static void swap(int[] A, int i, int j) {
    int t = A[i]; A[i] = A[j]; A[j] = t;
}

/** Sort the elements ARR[START .. END]. */
public static void mergeSort(int[] arr, int start, int end) {
    if (start >= end) {
        return;
    }
    int mid = (start + end)/2;
    mergeSort(arr, start, mid);
    mergeSort(arr, mid + 1, end);
    int index1 = start;
    int index2 = mid + 1;

    while (index1 < index2) {
        if (arr[index2] < arr[index1]) {
            swap(arr, index1, index2);
            if (index2 < end) {
                index2 += 1;
            }
        }
        index1 += 1;
    }
}
```

a. Instead, this code produces an array that is almost entirely unsorted. Why doesn't this code work?

Ans. *The merging code (the* **while** *loop) is wrong. When its swaps an* `arr[index1]` *from the left subarray with* `arr[index2]` *from the right, it causes the original* `arr[index1]` *to be out of order with respect to* `arr[index1+1]`*.*

b. On the other hand, *some* items in the array are guaranteed to be correctly placed in the result. Which?

Ans. *The largest and smallest elements will be placed correctly. Clearly, the smallest (first) element is correctly placed after the first step. If there is an element in the right half greater than all those on the left,* `index2` *will stop there and the greatest element will necessarily be at the end of the array. Otherwise, the largest element will be the one on the end of the left half and it will repeatedly be swapped until it reaches the end of the array. Therefore, the largest element will be end up on the right.*

c. How can the property described in (b) be used to properly sort the array using only calls to mergeSort to perform modifications of the array, if one doesn't care about execution time?

Ans. *We simply apply* `mergeSort` *repeatedly, incrementing* `left` *and decrementing* `right` *each time until* `left` *and* `right` *meet.*

4. [3 points] The *lowest common ancestor* of two nodes in a tree is the deepest node in the tree (furthest from the root) that has both of the nodes as descendants. It can be one of the two nodes (if one is an ancestor of the other). Write a function that, given the root of a binary search tree containing integers, and two integer keys that are in that tree, finds the key at the lowest common ancestor of the nodes containing the given keys

```
class IntTree {
    public int label;
    public IntTree left, right;
}

public class Utils {
    /** Assuming that T is a binary search tree and ITEM1 and ITEM2
     *  are labels in it, return the label in the lowest common
     *  ancestor of the nodes containing ITEM1 and ITEM2. */
    public static int findLCA(IntTree t, int item1, int item2) {
    // SOLUTION:
        if (item1 > item2)
            return findLCA(t, item2, item1);
        if (t.label > item2)
            return findLCA(t.left, item1, item2);
        if (t.label < item1)
            return findLCA(t.right, item1, item2);
        return t.label;
    // END SOLUTION
    }
```

5. [1 point] What is the size of the smallest field containing at least 114 elements?

Ans. *121 (that is, 11^2).*

6. [4 point] Suppose that we are building a specialized hash table that stores strings having some maximum length, L, and that we can allocate arrays of any finite length, no matter how large (yes, the Staff is aware that this problem has nothing to do with reality).

a. Show how to make a perfect (collision-free) hash table, in which each bucket in the table is guaranteed to have no more than one entry.

   Ans. *Treat the strings as numbers base K, where K is one larger than the size of the alphabet. Allocate an array of size K^L. The hash function then simply converts its string argument as for a base-K numeral.*

b. Considering the answer to part (a), Harry Hacker has an idea. Most of the hash table will consist of empty buckets; the number of non-empty buckets (non-null elements) is equal to the number of strings in the table. So why not use a sparse array structure (which actually *is* physically possible, in contrast to an infinite array)? A sparse array stores just the non-null entries and the indices at which they occur, so that the space it requires depends only on N, the number of strings stored. What's wrong with this strategy as a way to implement collision-free hash tables?

   Ans. *It just changes one search problem into another. In effect, this sparse array is a dictionary mapping integer indices into elements stored in the table.*

7. [3 point] One can perform a merge operation (such as used in merge sorting) on any number of sorted lists. Suppose that we have an array of `Iterator<String>` in which each `Iterator` delivers `Strings` in sorted order. We want to create a sorted list containing all the items produced by these iterators.

a. Describe how to do this quickly. You may use any of the data structures we've talked about in your implementation. You can use pseudo-code in your description, and don't need to provide actual Java code, but you must be precise in your description. If you use a particular search structure, for example, say exactly what you are storing in that structure and how it get used. Vagueness will lose points.

   Ans. *Call the iterators S_i, for $0 \leq i < N$, where N is the number of sources. 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.*

b. As a function of K, the number of `Iterators` and N, the total number of `Strings` delivered by all the iterators, give a worst-case bound on the number of string comparisons required by your solution.

   Ans. $\Theta(N \lg K)$.