COS 226 **Algorithms and Data Structures** **Fall 2021**

Midterm Solutions

1. **Initialization.** Don't forget to do this.

2. **Running time.**

   (2.1) $81{,}920 = 10 \times 2^{13}$

   Going down one row makes the running time go up by a factor of 4*; going right one column makes the running time go up by a factor of* 2*. So, the running time for* $m = 320$ *and* $n = 160$ *is* $2560 \times 4^2 \times 2 = 10 \times 2^{13}$.

   (2.2) $\Theta(m^2 n)$

   Doubling m *makes the running time go up by a factor of 4; doubling* n *makes the running time go up by a factor of 2.*

3. **Memory usage.**

   $32n$

   Each `Node` *object uses* 32 *bytes of memory:*

   - 16 bytes of object overhead
   - 4 bytes for the integer `key`
   - 8 bytes for the double `value`
   - 4 bytes of padding (to make memory usage a multiple of 8)

4. **Five sorting algorithms.**

   (4.1) *mergesort, just before merging the first two subarrays of length 6.*

   (4.2) *selection sort, after 12 iterations*

   (4.3) *quicksort, after the first partitioning step*

   (4.4) *insertion sort, after 16 (or 17) iterations*

   (4.5) *heapsort, immediately after the heap construction phase and putting 6 keys in place*

5. **Analysis of algorithms.**

(5.1) $\sim \frac{1}{2}n^2$

Selection sort makes $\sim \frac{1}{2}n^2$ *compares to sort any array of length* n*.*

(5.2) $\sim \frac{5}{16}n^2$

Let's consider first $n/2$ *and last* $n/2$ *iterations of insertion sort separately.*

- *In the first* $n/2$ *iterations (when inserting a random permutation of the integer* $1, 2, \ldots, n/2$*), insertion sort makes* $(n/2)^2/4$ *compares (on average) because we expect each key to be exchanged with half of the keys to its left (on average).*
- *In the second* $n/2$ *iterations (when inserting the 0s), insertion sort makes* $(n/2) \times (n/2)$ *exchanges because the 0 must be exchanged with all of the integers to its left.*

So, the expected number of exchanges is $\frac{1}{16}n^2 + \frac{1}{4}n^2 = \frac{5}{16}n^2$.
The number of compares in insertion sort is always within an additive factor of n *of the number of exchanges.*

(5.3) $\sim \frac{3}{4}n \log_2 n$

The left subarray contains a random permutation of $n/2$ *integers, so it takes* $\sim \frac{1}{2}n \log_2 n$ *compares to sort it. The right subarray contains all 0s, so it takes* $\sim \frac{1}{4}n \log_2 n$ *compares to sort it. Merging two subarrays of this form involves* $\frac{1}{2}n$ *compares, which is a lower-order term.*

(5.4) $\Theta(n)$, $O(n)$, $O(n^2)$, $O(2^n)$

$f(n) = 2^0 + 2^1 + 2^2 + 2^3 \ldots + n \ = 2n - 1.$

Big O and big Theta notations discard both lower-order terms and the leading coefficient. The main difference is that big O notation includes functions that grow more slowly. So, $O(n^2)$ *includes not only functions like* $2n^2$ *and* $\frac{1}{2}n^2$*, but also* $2n - 1$ *and* $\frac{1}{8}\sqrt{n}$*.*

6. **Rank in a BST.**

A J D K C K or A J D K C G or A J D K A J

This is a non-recursive version of the recursive algorithm discussed in precept.

7. **Data structure operations.**

(7.1) X O O X O X O O

The sequence of exchanges in the binary heap is:

- `exch(1, 19)`
- `exch(1, 3)`
- `exch(3, 6)`

(7.2) O O X X O X O

The sequence of elementary operations in the red–black BST is:

- color flip 8
- left rotate 4
- right rotate 10
- color flip 8

(7.3) O O O X O X X

The sequence of elementary operations in the linear-probing hash table is:

- `hash("F")`
- `keys[5].equals("F")`
- `keys[6].equals("F")`

8. **Data structure invariants**

(8.1) X X X O X

- *Symmetric order is always maintained during an insertion.*
- *Perfect black balance is always maintained during an insertion.*
- *The left rotate operation is applied only when the right link is red. So, after the rotation, the left link is red.*
- *If the left child is red and its right child is red, then the left child will be rotated left, leading to two left-leaning red links in a row.*
- *There is at most one right-leaning red link during an insertion. So, immediately after a left rotation, there cannot be any right-leaning red links.*

(8.2) X O X O X

(8.3) O O X X O

9. **Why did Java do that?**

(9.1) O X O X X

Like 3-way quicksort, dual-pivot quicksort makes $\Theta(n)$ *compares on inputs with* $\Theta(1)$ *distinct keys.*

(9.2) X O X O X

With a resizing array, insertions at the end take $\Theta(1)$ *amortized time but insertions at the front take* $\Theta(n)$ *time.*

(9.3) X X O X

We must rehash all of the keys during a resizing operation.

10. **Frequent word.**

We divide the problem into the following four subtasks:

(1) Count the number of occurrences of each word.
(2) Find a most frequent word of length k (for each word length k).
(3) Find a most frequent word of length $\geq k$ (for each word length k).
(4) Process the queries.

Here's one approach to efficiently implement the four subtasks.

(1) Use a `RedBlackBST<String, Integer> st`, where each key is a word in the array and the value is its frequency. Initialize `st` using the same algorithm from Lecture 10.

key	value
A	5
B	2
C	1
DD	4
EEEEE	3
FFFFFF	1
GGG	1

(2) Initialize a `String[] a` of length n so that `a[k]` is a word of length k that occurs most frequently. For each word w in `st`:

- let k denote the length of w
- update `a[k]` to w if w occurs more frequently than the existing `a[k]` (or if `a[k]` is `null`)

0	1	2	3	4	5	6
–	A	DD	GGG	–	EEEEE	FFFFFF

(3) Update `a[]` so that `a[k]` is a word of length $\geq k$ that occurs most frequently.

For $k = n - 1$ to 1:

- if `a[k+1]` occurs more frequently than `a[k]`, set `a[k] = a[k+1]`

0	1	2	3	4	5	6
A	A	DD	EEEEE	EEEEE	EEEEE	FFFFFF

(4) Return `a[k]`.

If the words could be longer than n, then the array `a[]` *may need more than $\Theta(n)$ space, violating one of the performance requirements. In this case, we could replace the array* `a[]` *with a* `RedBlackBST<Integer, String>`*, where each key is length k of some word in the array and the value is a word of length $\geq k$ that occurs most frequently. Then, the queries can be processed by using the ceiling operation.*

Alternative solution. A slicker solution is to sort `words[]` in reverse order by length, breaking ties lexicographically. The sorting brings duplicate words together, with longer words appearing in the array before shorter words. Now, we do a linear scan of the words, counting their frequencies, and keeping track of the champion words that we encounter. By scanning the words in decreasing order of length, each new champion is the most frequent word of length $\geq k$ that we have processed so far.

Here's the corresponding Java code.

```
public class FrequentWord {
    // key k = length of word
    // value = word of length >= k that occurs most frequently
    private ST<Integer, String> st = new ST<>();

    public FrequentWordSort(String[] words) {

        // sort words by length, breaking ties lexicographically
        Arrays.sort(words);
        Arrays.sort(words, new ByStringLengthComparator());

        // champion word and its frequency
        String champWord = "";
        int champCount = 0;

        // current word and its frequency
        String currentWord = "";
        int currentCount = 0;


        for (int i = 0; i < words.length; i++) {

            // update current word and its frequency
            if (!words[i].equals(currentWord)) {
                currentWord = words[i];
                currentCount = 0;
            }
            currentCount++;

            // if current word appears more frequently than old champion,
            // add (or update) symbol table entry
            if (currentCount > champCount) {
                champCount = currentCount;
                champWord = currentWord;
                st.put(champWord.length(), champWord);
            }
        }
    }

    public String query(int k) {
        if (k > st.max()) return null;
        return st.get(st.ceiling(k));
    }
}
```