COS 226 **Algorithms and Data Structures** **Spring 2022**

Midterm Solutions

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

2. **Running time.**

   (2.1) $31{,}250 = 6250 \times 5^1$

   Increasing the problem size by a multiplicative factor of 3 increases the running time by a factor of 5.

   (2.2) $\Theta(n^{\log_3 5})$

   Increasing the problem size by a multiplicative factor of 3 increases the running time by a factor of 5.

3. **Memory usage.**

   The memory usage is $\sim 4m$*, where* m *is the length/capacity of the array.*

   (3.1) $\sim 4n$

   In the best case, $n = m$*.*

   (3.2) $\sim 16n$

   In the worst case, $n = \frac{1}{4} \times m$*.*

4. **Five sorting algorithms.**

   (4.1) *selection sort, after 12 iterations*

   (4.2) *insertion sort, after 16 iterations*

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

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

   (4.5) *heapsort, after putting the 12 largest keys in place during the sortdown phase*

5. **Quicksort analysis.**

This was intended as a difficult question.

(5.1) X

This implementation is slow and wastes memory, but it is correct. The integers smaller than the pivot end up in `queue1`*. The integers larger than (or equal to) the pivot end up in* `queue2`*, with the pivot being the first integer in* `queue2`*.*

(5.2) O X

The `RandomizedQueue` *version fails to sort arrays, even when the* n *integers are distinct. In the partitioning step, the pivot ends up in* `queue2`*. But it end up in its correct position only if it is the first integer iterated over when processing* `queue2`*.*

The `MinPQ` *version is correct because the pivot ends up as the first integer in* `queue2`*.*

(5.3) $\sim 2n \ln n$

Just as in standard quicksort, the expected number of compares to sort a random permutation is $\sim 2n \ln n$*.*

(5.4) X X X X

This version of quicksort does not do the shuffle. As a result, quicksort makes $\Theta(n^2)$ *compares on inputs in in ascending (or descending) order.*

This version of quicksort puts all keys equal to the pivot on one side of the partition. As a result, quicksort makes $\Theta(n^2)$ *compares on inputs with all equal keys (or two distinct keys).*

(5.5) $\Theta(n^2)$

This version of quicksort uses $\Theta(n^2)$ *extra memory on arrays with* n *equal keys because (1) every partition is degenerate and (2) each recursive call allocates memory proportional to the length of the subarray to be sorted.*

(5.6) *stable*

This version of quicksort is stable. Any two equal keys end up in the same queue, in the same relative order. Then, they are assigned back to the original array in the same relative order.

This problem demonstrates that it is easy to make quicksort stable if you use an extra array of length n*.*

6. **Ceiling in a BST.**

F D C E F C or F D C E B C

We discussed a recursive version of the floor method in lecture. The ceiling method is very similar.

7. **Data structure operations.**

(7.1) O O O X X X O

The sequence of exchanges in the binary heap is:

- `exch(9, 19)`
- `exch(4, 9)`
- `exch(2, 4)`

(7.2) X X X O O X X O

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

- left rotate 26
- right rotate 28
- color flip 27
- color flip 24
- left rotate 14

(7.3) O O O O O X

Only one parent link is updated per call to `union()`*. In this case, the union causes the root of the tree containing element 4 (size = 8) to point to the root of the tree containing element 1 (size = 10).*

8. **Why did the system architect do that?**

(8.1) X X X O

(8.2) X X X O

(8.3) X X X O

9. **Triangle inequality.**

   This is similar to the $\Theta(n^2 \log n)$ algorithm for the 3-SUM problem, with one twist.

   1. Sort `c[]` using heapsort.
   2. For each i and j: count how many entries in `c[]` are strictly less than `a[i] + b[j]`. Each such entry `c[k]` satisfies the triangle inequality `a[i] + b[j] > c[k]`.

      Step 2 can be computed efficiently using a version of *binary search*. In particular, we need to compute the *rank* of `a[i] + b[j]` in the sorted array `c[]`. In lecture, we described a binary search algorithm for computing the rank of a key in a sorted array. Actually, if duplicate integers are allowed, we need a slightly refined version of this rank algorithm, ala `firstIndexOf()` from the *Autocomplete* assignment. Alternatively, we could slightly modify our rank implementation to support real-valued arguments and compute the rank of `a[i] + b[j] - 0.5`.
   3. Return the sum of all of the counts from Step 2.

   Running time. This solution takes $\Theta(n^2 \log n)$ time in the worst case.

   - Sorting takes $\Theta(n \log n)$ time in the worst case with *heapsort*.

     Either *insertion sort* or *selection sort* would also be fine because sorting is not the bottleneck operation.
   - Performing $\Theta(n^2)$ *binary searches* in an array of length n takes $\Theta(n^2 \log n)$ time in the worst case.

   Note that there can be $\Theta(n^3)$ triples. So, any solution that processes each triple individually (e.g., by incrementing a counter by 1) will take at least $\Theta(n^3)$ time in the worst case.

   Extra memory. This solution uses only $\Theta(1)$ extra memory.

   - *Heapsort* uses only $\Theta(1)$ extra memory.

     Either *insertion sort* or *selection sort* would also be fine because they also use only $\Theta(1)$ extra memory.

     Either *mergesort* and *quicksort* would be a deduction because they use more than $\Theta(1)$ extra memory (for the auxiliary array or function-call stack).
   - Our binary search algorithm uses only $\Theta(1)$ extra memory, provided we do it non-recursively.

     An explicitly *recursive* version of binary search would be a deduction because it uses $\Theta(\log n)$ extra memory for the function-call stack.