COMPSCI 2C03 – Week 4 Exercises

Sample solutions and notes on sample solutions for some of this week's exercises.

Lecture 1: Merge Sort

1. Trace the Merge Sort algorithm for the array -5, 9, 10, 4, -8, 0, 12, 15, 1, 3.

   - When performing the simulation, remember that the larger half always goes to the left.

2. Suppose that the Merge Sort algorithm is modified to skip the call to the merge function whenever $a[mid] \leq a[mid+1]$. Explain why this might be a good idea. How does this change the best case running time of Merge Sort? (Hint: Use a recursion tree and consider the number of comparison operations required at each level in the best case.)

   - Best case = array already sorted. Each merge step now costs O(1) time instead of O(n) time.
   - So the amount of work is proportional to the number of nodes in the recursion tree.
   - For a tree of height h, there are approximately 2^h nodes. But h is approximately lg n, so there are approximately n nodes.
   - So the best case running time is in O(n).

3. [Tricky] Describe how the Merge Sort algorithm described in class could be implemented for a linked list. Are there any extra running time costs in this case? Would the algorithm still be $O(n \log n)$? Justify your answers.

   - It's an O(n) operation to find the midpoint of the list.
   - When merging sublists, you have two pointers i and j that advance through the list and you build a new sorted list with the original elements. So you don't need auxiliary storage.
   - Algorithm remains O(n log n) despite the extra running time cost of finding the list midpoint.
   - For details: https://www.interviewkickstart.com/learn/merge-sort-for-linked-list

4. [Tricky] You can eliminate some extra data movement if you swap the role of the `a` and `aux` arrays back and forth throughout the algorithm. This might require multiple versions and/or changes to the interfaces of the `msort` and `merge` methods. Can you put together pseudocode that will work? How big would you expect the improvement to be? Is the algorithm still $O(n \log n)$?

Lecture 2: Quicksort

5. Trace the Quicksort algorithm for the array -5, 9, 10, 4, -8, 0, 12, 15, 1, 3.
   - First pivot will be -5. Partition result will be:
     - -8, **-5**, 10, 4, 9, 0, 12, 15, 1, 3
     - First recursive call will be for singleton array -8
     - Second will be for 10, 4, 9, 0, 12, 15, 1, 3
     - Pivot is 10. Partition result will be:
       - 1, 4, 9, 0, 3, **10**, 15, 12
       - And so on…

6. About how many comparison operations will Quicksort have to make when sorting an array of n values that are all equal?
   - (See Quicksort slide 3)
     - If all values are equal, the i value will continue until i=hi. That's n-1 comparisons. The j value does the same. Then it breaks out of the infinite loop after one more comparison. So 2n-1 comparisons total
     - The partitions will be size 0 and n-1. On the larger partition there will be about 2(n-1)-1 = 2n-3 comparisons.
     - So it's a sum (2n-1)+(2n-3)+(2n-5)+…+5+3+1.

$$2 * \text{the sum} = (2n-1) + (2n-3) + (2n-5) + \ldots \quad \text{(there are n terms)}$$
$$+ \quad 1 \quad + \quad 3 \quad + \quad 5 \quad + \ldots = 2n * n = 2n^2$$

   - So it takes about $2n^2$ comparisons

7. Consider an algorithm that sorts by splitting into three equally sized partitions, examining each element in the partition, then recursively calling itself on each partition. You can assume a base case of n=1. Construct and solve a recurrence relation for the running time of this algorithm, then give an expression for its running time using Big O notation.
   - $T(1) = c$
   - $T(n) = 3T(n/3)+cn$
   - See the argument in Quicksort slide 7. $T(n) <= cn \log_3 n$ which is $O(n \log n)$

8. Rewrite the Partition algorithm in pseudocode so that it works for a doubly linked list. Swapping items within Nodes is ok, you don't have to swap the nodes themselves.

   Detecting i and j crossing over is a little more tricky than when i and j are indexes. I'm doing it with an extra check in each of the loops. Other than that, this is a direct translation of slide 3 of the Quicksort slides.

```
partition(list, lo, hi) : # lo/hi are pointers to head/tail of sublist
  pivot ← lo.item
  i ← lo.next
  j ← hi
  crossover ← false
  loop :
     while i.item <= pivot:
       i ← i.next
       if i == j: crossover = true
       if i == hi: break (while)
     while j.item >= pivot:
       j ← j.previous
       if i == j: crossover = true
       if j == lo: break (while)
     if crossover: break (infinite loop)
     swap i.item, j.item
  end loop
  swap lo.item, j.item
  return j
```

Lecture 3: Radix Sort

9. Simulate the Radix Sort algorithm for this data: 984, 234, 1, 945, 20, 345, 445, 211, 399, 288. Use $R = 10$.

Pass	0	1	2	3	4	5	6	7	8	9
1	20	1, 211			984, 234	945, 345, 445			288	299
2	1	211	20	234	945, 345, 445				984, 288	299
3	1, 20		211, 234, 284, 299	345	445					945, 984

The array after pass 1: 20, 1, 211, 984, 234, 945, 345, 445, 288, 299
The array after pass 2: 1, 211, 20, 234, 945, 345, 445, 984, 288, 299
The array after pass 3: 1, 20, 211, 234, 284, 299, 345, 445, 945, 984

10. Simulate the MSD-first recursive Radix Sort for the same data as the last question.

Here's the top level buckets…

0	1	2	3	4	5	6	7	8	9
1, 20		234, 211, 288	345, 399	445					984, 945

Then recursively sort each bucket on digit 2 and then digit 3. Here's the second pass for bucket 2:

0	1	2	3	4	5	6	7	8	9
	211		234					288	

No sorting left to be done on this bucket, rearranges them as 211, 234, 288.

11. Simulate the LSD Radix Sort algorithm for this data: 23, 22, 15, 5, 78, 99, 15. Use $R = 5$ (the original numbers are in base 10).

w = ceiling($\log_5$ 99)=ceiling(log 99/log 5) = 3
Use d ← item % 5^{p+1}// 5^p to extract each digit

12. [Tricky] Write pseudocode for the MSD Recursive Radix Sort algorithm described in class.

Here's a partial implementation. There may be other approaches that could be made to work.

```
Radix_sort(a, R, w) :
      q ← create_queue()
      for item in a :               # load up a queue
            enqueue(q, item)

      rsort(q, R, w, w-1)           # start the recursion

      for i = 0 to length(a)-1:     # dump queue back into array
            a[i] = dequeue(q)

rsort(q, R, w, current_digit) :
      if current_digit >= w: return # base case

      # create the queue buckets
      # dequeue each item and place in the correct bucket
      # call rsort on each bucket, incrementing the current digit
      # empty the buckets back into q
```