ICS 311, Fall 2020, Problem Set 05, Topics 9 & 10A SOLUTIONS

Firstname Lastname <uhumail@hawaii.edu>
Due by midnight Tuesday October 13. 40 points total.

#1. Analysis of d-ary heaps (15 pts)

In class you did preliminary analysis of ternary heaps. Here we generalize to *d*-ary heaps: heaps in which non-leaf nodes (except possibly one) have *d* children.

a. (5) How would you represent a *d*-ary heap in an array with 1-based indexing? Answer this question by

- (1) Giving an expression for **Jth-Child(i,j)**: the index of the *j*th child as a function of *j* and the index *i* of a given node, and
- (1) Giving an expression for **D-Ary-Parent(i)**: the index of the parent of a node as a function of its index *i*.
- (3) Checking that your solution works by showing that D-Ary-Parent(Jth-Child(i,j)) = i (Show that if you start at node i, apply your formula to go to a child, and then your other formula to go back to the parent, you end up back at i).

  Solution:
  Jth-Child(i, j) = d(i - 1) + j + 1
  D-Ary-Parent(i) = floor((i - 2)/d + 1)

  Proof of invertibility
  D-Ary-Parent(Jth-Child(i,j))
  = D-Ary-Parent(d(i - 1) + j + 1)
  = floor((d(i - 1) + j + 1 - 2)/d + 1)
  = floor(d(i - 1)/d + j/d - 1/d + 1)
  = floor(i - 1 + j/d - 1/d + 1)
  = floor(i + j/d - 1/d)
  = floor(i + (j-1)/d)
  = floor(i) + floor((j-1)/d) *// since i is an integer*
  = floor(i) *// j ≤ d so (j-1) < d so (j-1)/d < 1 so floor((j-1)/d) = 0*
  = i

b. (2) What is the height of a *d*-ary heap of *n* elements as a function of *n* and *d*? By what factor does this height differ from that of a binary heap of n elements?

> **Solution :** $\Theta(\log_d n) = \Theta(\lg n / \lg d)$, differing from binary heaps by a constant factor of 1/lg d.

c. (4) Give an efficient implementation of EXTRACT-MAX in a *d*-ary max-heap. (Hint: consider how you would modify existing code.) Analyze its running time in terms of *n* and *d*. (*Note that d must be part of your Θ expression.*)

> **Solution:** Use Heap-Extract-Max, but change Max-Heapify to compare to all d children. (Students may write code here.)
>
> Time is proportional to height of heap times number of children:
> $\Theta(\lg n / \lg d) * d = \Theta(d \lg n / \lg d)$ or $\mathbf{\Theta(d \log_d n)}$

d. (4) Give an efficient implementation of INSERT in a *d*-ary max-heap. Analyze its running time in terms of *n* and *d*.

> **Solution:** Use existing Insert but change Heap-Increase-Key to call the D-Ary-Parent. (Students may write code here.)
> Since Heap-Increase-Key climbs the tree, in the worst case it must climb $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

#2. Quicksort Pathology (7 pts)

The point of this question is to show that data patterns other than strictly sorted data can be problematic in non-randomized Quicksort.

a. (3) Trace the operation of a single call to <u>Partition (A, 1, 9)</u> (not randomized) on this 1-based indexing array:

> A = [1, 6, 2, 8, 3, 9, 4, 7, 5], p=1, r=9

Show the state of A after the call and the value Partition returns.

> **Solution:** A = [1, 2, 3, 4, 5, 9, 8, 7, 6] with returned value 5.
> (see appendix for full trace)

b. (2) On what subarray will Quicksort in line 3 be called? A[**1, 4**] = [1, 2, 3, 4]
On what subarray will Quicksort in line 4 be called? A[**6, 9**] = [9, 8, 7, 6]

c. (2) How are the keys organized in the two partitions that result? How do you expect that this behavior will affect the runtime of Quicksort on data with these patterns?

Solution:
The keys are sorted in increasing order in the first partition and decreasing order in the second. We know that (nonrandomized) quicksort on sorted data is O(n^2). The above example shows that it can also be O(n^2) when sorted sequences are interleaved or embedded in the data. We need randomization for more than just the sorted case.

#3. 3-way Quicksort (18 pts)

In class we saw that the runtime of Quicksort on a sequence of *n* identical items (i.e. all entries of the input array being the same) is O(n^2). All items will be equal to the pivot, so n-1 items will be placed to the left. Therefore, the runtime of QuickSort will be determined by the recurrence T(n) = T(n-1) + T(0) + O(n) = O(n^2) To avoid this case, and to handle duplicate keys in general, we are going to design a new partition algorithm that partitions the array into three partitions, those that are strictly less than the pivot, those equal to the pivot, and those strictly greater than the pivot.

a. (10) Develop a new algorithm *3WayPartition(A, p, r)* that takes as input array *A* and two indices *p* and *r* and returns a pair of indices (*e, g*). *3WayPartition* should partition the array *A* around the pivot *q* = *A*[*r*] such that every element of *A[p..(e-1)]* is strictly smaller than *q*, every element of *A[e..g-1]* is equal to *q* (e indicates the start of “equal” keys), and every element of *A[g..r]* is strictly greater than *q* (g indicates the start of “greater” keys). Explain why your code is correct.

Hint: modify Partition(A,p,r) presented in the lecture notes/book, such that it adds the items that are greater than q from the right end of the array and all items that are equal to q to the right of all items that are smaller than q. You will need to keep additional indices that will track the locations in A where the next item should be written.

Solution (student code may differ):
3WayPartition(A, p, r)

```
1  q = A[r]                    // q is the pivot
2  l = p-1                     // invariant: A[p…l] are all less than the pivot
3  g = r+1                     // invariant: A[g...r] are all greater than the pivot
4  if (r > p)
5      i = p
6      while (i < g)
7          if (A[i] > q)
8          g = g - 1
9              exchange A[i] and A[g]
10         else if (A[i] < q)
11             l = l + 1
12             exchange A[i] and A[l]
13             i = i + 1
14         else
15             i = i + 1
16  e = l+1    // A[l] is the last one that is smaller than pivot q, so A[l+1] = q
18  return (e, g)
```

The correctness of the algorithm follows from the loop invariants that A[p...l] are all less than the pivot q, A[g...r] are greater than the pivot q. It is easy to verify that the properties of the loop invariants are maintained for these two loop invariants. Then it follows that elements in A[l+1...g-1], which are not less than the pivot and not greater than the pivot, are equal to the pivot (else they would have been exchanged out in lines 9 or 12).

The procedure terminates because in each iteration either *i* increases or *g* decreases, i.e. the distance between *i* and *g* always decreases.

b. (4) Develop a new algorithm *3WayQuicksort* that uses *3WayPartition* to sort a sequence of *n* items, keeping in mind that *3WayPartition* returns a pair of indices (e, g).

Solution:

```
3WayQuicksort(A, p, r)
1  if (p < r)
2      (e, g) = 3WayPartition(A, p, r)
3      3WayQuicksort(A, p, e-1)
4      3WayQuicksort(A, g, r)
```

Notice that we *do not recurse* on the “equal” keys, so the larger this partition is the more time we save, as shown below.

c. (4) What is the runtime of *3WayQuicksort* on a sequence of *n* random items? What is the runtime of *3WayQuicksort* on a sequence of *n* identical items? Justify your answers.

Solution:

The expected runtime of 3WayQuicksort on a sequence of *n* random items is the same as expected runtime of randomizedQuicksort, i.e. O(n log n). The analysis is nearly identical because we expect the middle partition to consist of 1 or a small number k of identical keys, so the recurrence is bounded above by $T(n) = 2T(n/2) + \Theta(n)$.

More precisely, if k is the expected number of identical keys the recurrence is $T(n) = 2T((n-k)/2) + \Theta(n)$. The expected value of k depends on information we don't have, such as the number of possible key values, but will be very small for any nontrivial key sets. Furthermore, as k increases, the algorithm gets faster, as we see below.

The runtime of 3WayQuicksort(A, 1, n) on a sequence of *n* identical keys is O(*n*) because the first call on 3WayPartition will return (e,g) = (0, n+1), because all items will be equal to the pivot A[n]. Thus, the recursive calls will be 3WayQuicksort(A, 1, 0) and 3WayQuicksort(A, n+1, n), which will return immediately because both times (p > r). Thus, the runtime is just a single call to 3WayPartition, which takes O(n) time.

More simply, substitute k=n into $T(n) = 2T((n-k)/2) + \Theta(n)$ and we get $T(n) = 2T(0/2) + \Theta(n) = \Theta(n)$.

Appendix: Full trace of call to Partition (A, 1, 9)

Initially:
A = [1, 6, 2, 8, 3, 9, 4, 7, 5], i=0, j=1, pivot = A[r] = A[9] = 5

Trace at the conclusion of each pass through the loop lines 3-6
A = [1, 2, 6, 8, 3, 9, 4, 7, 5], i=2, j=4, no exchange
A = [1, 2, 3, 8, 6, 9, 4, 7, 5], i=3, j=5, exchanged A[3] with A[5]
A = [1, 2, 3, 8, 6, 9, 4, 7, 5], i=3, j=6, no exchange
A = [1, 2, 3, 4, 6, 9, 8, 7, 5], i=4, j=7, exchanged A[4] with A[7]

A = [1, 2, 3, 4, 6, 9, 8, 7, 5], i=4, j=8, no exchange

After the swap in line 7:
A = [1, 2, 3, 4, 5, 9, 8, 7, 6], i=4, j=9, exchange A[5] with A[9]

What does Partition(A, 1, 9) return? **5**

Continuing execution of the top level call to Quicksort, identify the two partitions that will be handled by the recursive calls to Quicksort at this level:
On what subarray will Quicksort in line 3 be called? A[**1, 4**]
On what subarray will Quicksort in line 4 be called? A[**6, 9**]

Now in parts b and c we trace these two calls in a manner similar to above.