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 d-ary heaps in an array with 1-based indexing? Answer this question by:

- Giving an expression for the J-th Child(i,j): the index of the j-th child as a function of the index i of the given node, and the child index j within the given node.

$$Jth - Child(i, j) = d(i - 1) + j + 1$$

- Giving an expression for D-Ary-Parent(i): the index of the parent of a node as a function of its index i.

$$D - Ary - Parent(i) = Floor[\tfrac{i-2}{d} + 1]$$

- Checking 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).

This given example is expressed as i = 12, j = 2, and d = 4, where each number is chosen to represent the parameters of D-Ary-Parent.

J-th-Child (12,2)
    return 4(12-1)+2+1; //Value of 47
D-Ary-Parent(J-th-Child(i,j) = D-Ary-Parent(46))
D-Ary-Parent(46);
    return $\lfloor \frac{46-2}{4+1} \rfloor$
Since the value returned is 12, we know that the index i is equal to the D-Ary-Parent.

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?

From the Topic 9 Notes that indicate Heaps, we know that the "Height of an n-node nearly complete binary tree(6.1-2)" is as follows:

Given an n-node nearly complete binary tree of height h, from 6.1-1:
$$2^h \leqslant n \leqslant 2^{h+1} - 1 < 2^{h+1}$$
By taking the log of the first, second and last terms, this simplifies to:
$$h \leqslant lgn \leqslant h + 1$$
This means that:
$$h = \lfloor lgn \rfloor$$

By using the d for the d-ary heap as oppose to the 2 that is implemented in a binary heap, you get: $\theta(lg_d n)$
$\therefore$ The factor of the height difference from that of a binary heap otherwise differs by the factor of 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.)

```
1  D-Heap-Extract-Max(A,i,d)
2  if A.heapsize < 1
3    return the error of a heap underflow
4  maximum = A[1]
5  A[1] = A[A.heapsize]
6  heapsize = heapsize - 1
7  D-Heapify(A,1,d)
8  return maximum
```

This example of HEAP-EXTRACT-MAX is taken from CLRS, which exemplifies the d-ary max-heap. In terms of n and d, we notice that the runtime of each line excluding line 6 would run in O(1) time. Max-Heapify otherwise takes $O(lgn)$ time, meaning that the runtime of D-Heapify would otherwise taken $O(lg_d n)$ runtime. We take that this algorithm runs in $T(n) = O(1) + O(lg_d n)$ time, and when dropping all constants, this means that it runs in $O(lg_d n)$ time.

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

```
1  D-Heap-Insert(A, key)
2  A.heapsize = A.heapsize + 1
3  A[A.heapsize] = -infinity
4  D-Increase-Key(A, A.heapsize, key)
```

This example of HEAP-EXTRACT-MAX-INSERT is again taken from CLRS, which is modified to work for the D-Ary heap. Similar to HEAP-EXTRACT-MAX, we note that each line excluding the final, (line 3) runs in $O(1)$ time, which is constant. The Heap-Increase noted in CLRS is noted with being a runtime of $O(lgn)$. This means that the D-Increase-Key on line 3 would execute a runtime of $O(lg_d n)$ time. We take this algorithm that runs in $T(n) = O(1) + O(lg_d n)$ time, and by dropping all constant terms, this gives the final runtime of $O(lg_d n)$.

2. Quicksort Pathology (7 pts)

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

a (3) Trace the operation of a single call to Partition (A,1,9) (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.

By tracing the operation of the single call to Partition(A, 1, 9), and by using indexing as oppose to values:

A = [1, 6, 2, 8, 3, 9, 4, 7, 5] where i = 0, p,j = 1, and r = 9
A = [1, 6, 2, 8, 3, 9, 4, 7, 5] where i, p = 1, j = 2, and r = 9
A = [1, 6, 2, 8, 3, 9, 4, 7, 5] where i, p = 1, r = 9, and j = 3
A = [1, 2, 6, 8, 3, 9, 4, 7, 5] where p = 1, i = 2, r 9, and j = 3
A = [1, 2, 6, 8, 3, 9, 4, 7, 5] where p = 1, i = 3, r = 9, and j = 4
A = [1, 2, 3, 8, 6, 9, 4, 7, 5] where p = 1, i = 3, r = 9, and j = 5
A = [1, 2, 3, 8, 6, 9, 4, 7, 5] where p = 1, i = 4, r = 9, and j = 6
A = [1,2, 3, 4, 6, 9, 8, 7, 5] where p = 1, i = 4, r = 9, and j = 7
A = [1, 2, 3, 4, 5, 9, 8, 7, 6] where p = 1 and r = 9

By tracing the operations, we know that in the first partition, $1 > 5$, meaning the indices of i and j can be moved. The comparisons of these values, denoted as j and r would give the indices of $6 > 5$, which allows for the refractoring of j. The comparisons between 2 and would would allow for i to increment, and otherwise swap between the values of i and j. Once these values are swapped, there is a comparison between 5 and 3, and the array continues to increment and swap the values. At the end of the given array, the swaps are complete because the algorithm has completely traversed. The final swap is between the values of r and i.

b (2) On what subarray will Quicksort in line 3 be called? On what subarray will Quicksort in line 4 be called?

The subarray that Quicksort in line 3 will be called is [1,2,3,4]. This is due to the fact that it calls the Quicksort as Quicksort(A, 1, 5-1).

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 patters?
The keys that are organized in the two partitions as a result is:

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

This is due to the left side of the array being ordered, as the right side is the only section of the array that needs ordering.

The behavior will affect the runtime of Quicksort on the data with these patterns is O(n + x), where x is the number of comparisons that is computed in Partition over the n number

of elements in the array in Quicksort.

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 $T(n) = T(n - 10 + T(0) + O(n^2)$. 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 (q 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 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.

```
3WayPartition (A, p, r)
x = A[r]
less = p - 1
greater = r + 1
if r > p
  index = p
  while index < greater
    if A[index] > x
      greater = greater - 1
      exchange A[index] with A[greater]
    else if A[index] < x
      less = less + 1
      exchange A[index] with A[less]
      index = index + 1
    else
      index = index + 1
e = less + 1
return(e,greater)
```

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).

```
1  3WayQuicksort (A, p, r)
2  if p < r
3    (e, g) = 3WayPartition(A, p, r)
4    3WayQuicksort (A, p, e - 1)
5    3WayQuicksort (A, g, r)
```

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.

A Sequence of n Random Items

The runtime of *3WayQuicksort* with n random items is $O(nlgn)$. This is due to all of the items within the array being random, meaning that the number of items within the array will have an equal chance of being chosen to be the partitioned element of the Quicksort.

A Sequence of n Identical Items

The runtime of *3WayQuicksort* with n identical items is $O(n)$.