1) Analysis of D-ary Heaps

(a) **How would you represent a d-ary heap in an array with 1-based indexing?**
Jth-Child(i,j) = d(i-1)+j+1
D-Ary-Parent(i) = $\lfloor \frac{i-2}{d} + 1 \rfloor$
Proof of Correctness

- D-Ary-Parent(Jth-Child(i,j)) =
- D-Ary-Parent(d(i-1)+j+1) =
- $= \lfloor \frac{di - d + j + 1 - 2}{d} + 1 \rfloor$
- $\lfloor i + \frac{j-1}{d} \rfloor$ since i is an integer then the floor will not affect its outcome $d \geq j$ then from mathematical ideas the greater the denominator it will then the whole will be closer to 0 then
- = i + 0 = i

Hence, D-Ary-Parent(Jth-Child)$(i, j) = i$.

(b) **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?**
Recall that the height of the binary heap of n elements is $\Theta(\lg(n))$ and that the d-ary heap of n elements is $\log_d n = \frac{lgn}{lgd}$ (from the change of base formula) hence the difference between the d-ary heaps is by a factor of $\frac{1}{\lg d}$.

(c) **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.**

Algorithm 1 EXTRACT-MAX

```
if A.heap-size < 1
    error "HEAP underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A,1)
return max
```

Algorithm 2 MAX-HEAPIFY(A,i)

```
j = d(i-1) + 2
imax = j + d - 1
if imax > A.length
    imax = A.length
largest = j
while j ≤ imax
    if A[j] > A[largest]
        largest = j
    j = j + 1
if A[largest] > A[i]
    swap A[largest] and A[i]
    MAX-HEAPIFY(A, largest)
```

Runtime: $\Theta(\frac{lgn}{lgd}d) = \Theta(d \log_d n)$

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

Algorithm 3 MAX-HEAP-INSERT(A.key)

```
A.heap-size = A.heap-size + 1
A[A.heap-size] = −∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Algorithm 4 HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
parent = D-Ary-Parent(i)
while i > 1 and A[i] > A[parent]
    swap A[i] and A[parent]
    parent = D-Ary-Parent(parent)
```

Run time of the algorithm in the worst case is $\Theta(\log_d n)$

2) Quicksort Pathology

(a) **Trace the operation of a single call to Partition (A, 1, 9) (not randomized) on this 1-based indexing array**

$$\begin{array}{c} \begin{array}{ccccccccc} i\ p,j & & & & & & & & r \end{array} \\ \begin{bmatrix} 1 & 6 & 2 & 8 & 3 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $1 < 5$ switch the i and j values

$$\begin{array}{c} \begin{array}{ccccccccc} i,p & j & & & & & & & r \end{array} \\ \begin{bmatrix} 1 & 6 & 2 & 8 & 3 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $6 > 5$ then

$$\begin{array}{c} \begin{array}{ccccccccc} i,p & & j & & & & & & r \end{array} \\ \begin{bmatrix} 1 & 6 & 2 & 8 & 3 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $2 < 5$, increment i and then swap between i and j

$$\begin{array}{c} \begin{array}{ccccccccc} p & i & j & & & & & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 6 & 8 & 3 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $8 > 5$ then

$$\begin{array}{c} \begin{array}{ccccccccc} p & i & & j & & & & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 6 & 8 & 3 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

But, $5 > 3$ therefore increase the i-th position and then swap the i and j

$$\begin{array}{c} \begin{array}{ccccccccc} p & & i & & j & & & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 8 & 6 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $9 > 5$ then

$$\begin{array}{c} \begin{array}{ccccccccc} p & & i & & & j & & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 8 & 6 & 9 & 4 & 7 & 5 \end{bmatrix} \end{array}$$

Since $5 > 4$ then swap 8 and 4 then

$$\begin{array}{c} \begin{array}{ccccccccc} p & & & i & & & j & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 4 & 6 & 9 & 8 & 7 & 5 \end{bmatrix} \end{array}$$

After that no more swap is necessary since j arrives to the last slot. After that swap 5 with the value located at i+1 which is 6 then the finalized partitioned array is

$$\begin{array}{c} \begin{array}{ccccccccc} p & & & & & & & & r \end{array} \\ \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 9 & 8 & 7 & 6 \end{bmatrix} \end{array}$$

Then this returns 5.

(b) **On what subarray will Quicksort in line 3 be called?**
The sub-array in line 3 of Quick-sort is QuickSort(A, 1, 5 -1) or

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

On what subarray will Quicksort in line 4 be called?
The sub-array of Quicksort in line 4 is (A, 5+1, 9) so

$$\begin{bmatrix} 9 & 8 & 7 & 6 \end{bmatrix}$$

(c) **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?**
The line 3 subarray does not need to be rearranged as it is already in rising order. On the other the sub-array in line 4 is not in arranged order. Therefore, the right array is partitioned, The finalized array after quicksort is

$$\begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9\end{bmatrix}$$

We know from the textbook that Partition has a best runtime of $\Theta(n \log n)$ and a worst case of $\Theta(n^2)$. From Lemma 7.1 of our textbook the running time of QuickSort is O(n+X) where X is the number of comparison performed in partition. Therefore, the run time of the quicksort is $O(n^2)$ as partition took n^2 and from a mathematical standpoint $n^2 > n$.

3) 3-way Quicksort

(a) **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] i s 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.**

Algorithm 5 3WayPartition(A, p, r)

```
pivot = A[r]
m = p - 1
n = r + 1
if p < r
    i = p
    while i < n
        if A[i] > pivot
            n = n - 1
            swap A[i] and A[n]
        else if (A[i] < pivot)
            m = m + 1
            swap A[i] and A[m]
            i = i + 1
        else
            i = i + 1
m = m + 1
return(m,n)
```

Loop Invariant

A[p...m] are all less than the pivot and A[n ... r] are all greater than the pivot.

The loop invariant is maintained since in either cases the distance between i and n decreases for every iteration.

(b) **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)**

Algorithm 6 3WayQuickSort(A, start, end)

```
if (start < end)
    (m,n) = 3WayPartition(A, start end)
    3WayQuickSort(A, start, m - 1)
    3WayQuickSort(A, n, end)
```

(c) **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.**

The expected runtime of 3WayQuickSort on a seqeunce of n random item is similar to randomized quicksort and is O($n \log n$).

However, for n identical items the expected run time for the QuickSort is O(n). On the first call on 3WayPartition it will return (0, n+1) for (m,n) because all the items will be equal to the pivot A[n] point.

Next, during the two recursive calls, of 3WayQuickSort it will return immediately because start > end both calls.

Therefore, the expected runtime is O(n).