PROBLEM 4

a. Using an oracle, we can be guaranteed that the quicksort algorithm would partition its elements with a 3-1 split or better. Meaning that the pivot would be an element where at most ¾ of the elements are to the right or the left of the pivot. Say we generalise this to it being ¼ on the left and ¾ on the right, this would mean that the left elements (left children) would be a subproblem a quarter of the size of the problem for the pivot (parent). This also means that the right child would have a subproblem ¾ times the size of the parent's problem. This means that to get from a left-most child up to the root node (n), we would need to keep multiplying it by 4 till we reach n while we would have to keep multiplying the right-most child by 4/3 to get up to n.

   From that we can conclude that the height of the left-most child would be at most $\log_4 n$ while the height of the right-most child would have at most $\log_{4/3} n$.

   The partitioning process for the levels above and including the height of the left most child, we would have to run through all the nodes, n. This would mean that total partitioning time would be cn, where c is a constant. For the rest of the levels down to the right-most child, the because the number of nodes we have to partition is less than the total number of nodes, the cost would be less than cn.

   Since there are $\log_{4/3} n$ levels in total, the total partitioning time could be represented as $O(n \log_{4/3} n)$ which is the same as O(n log n).

b. 50% chance.

c. Because the oracle algorithm includes the middle element, which would give us the best case of O(n), we can count it as part of the above situation and therefore we would only need to consider the worst case scenario, where the pivot is not in the middle half of the elements in the array or pivot chosen is the smallest/largest element of the array, which would lead to all the elements being on one side of the partition. This leads to a 50% chance of an array being split 3-1 or better and 50% chance of the array being split in the worst case. In other words, it would take us an estimated 20 trials to achieve 10 successes (success being the better case of the oracle pivot).

   If we had the oracle algorithm, it would take us 10 trials to have 10 successes. This means that we can conclude that if we alternated getting the worst-case scenario half the time and the 3-1 split or better the other half, we would have a running time that is twice the running time of getting the 3-1 split or better all the time. Since this is a constant, we can therefore say that the average case time complexity remains at O(n log n), which is the same as the worst case time complexity of using the oracle algorithm

PROBLEM 5

```
function minMaxWeightPath(⟨V,E⟩,v0)

    // Initialise all the nodes
    for each v ∈ V do

        dist[v] ← ∞
        max_weight[v] ← ∞
        prev[v] ← nil
```

```
dist[v0] ← 0
Q ← InitPriorityQueue(V )

while Q is non-empty do

    u ← EjectMin(Q)

    for each(u,w)∈E do

        // Check if the maximum weight of a path using the edge is lower than or equal to
        // the current maximum weight. If it is larger than the current max weight, then we
        // do not use it
        if max_weight_to($v_0$, w) <= max_weight[u] then
            // Follow Dijkstra's algorithm by choosing the path with the lowest cost and
            // updating the priority queue
            if dist[u] + weight(u,w) < dist[w] then

                dist[w] ← dist[u] + weight(u, w)
                prev[w] ← u
                max_weight[w] ← max_weight_to($v_0$, w)
                Update(Q,w,dist[w], max_weight[w])
```

We can use a modified version of Dijkstra's algorithm, which finds the shortest path from a node to any other node, to help us find the min max weight path. This is because Dijkstra can help us find the shortest path from a single source to connect to any other node. However, this is not guaranteed to have the minimum maximum cost. This can be modified whereby we change the rule we follow to update the priority queue to update it using the vertex with the minimum maximum weight. This is done by checking if the maximum weight of a path using the edge is lower than or equal to the current maximum weight. If it is larger than the current max weight, then we do not use it. This then helps us satisfy the first two requirements which is to minimise the maximum weight and that among all such paths, this is the path with the lowest cost.

Because this algorithm relies heavily on Dijkstra's algorithm, which has a time complexity of O(|V|*|E|), we can assume that this algorithm would have a similar time complexity, the only difference being our additional rule of getting the maximum weight of the path from the source to a new node w, given by max_weight_to(v_0, w). This function would essentially run through a smaller version of Dijkstra using the nodes we have already connected to and the new node w, where we would greedily bring in edges to connect to the source node until we reach w. Because we do this for every edge, and because the length of the path from the source to the destination gets longer the more edges we bring in, this would cost us O(|E| log |V|). Multiply both together, we get a overall time complexity of O((|E|^2 * |V|) log |V|).

In my algorithm, max_weight_to(v0, w) calculates the maximum edge weight of a path from the source node v0 to w. My algorithm is a modified version of Dijkstra's algorithm which can be found in the lecture slides.

PROBLEM 6

```
function weightedASPS(G)

    // Store the pair, {u, v} and the minimum cost of the unweighted graph in an array
    min_cost[ ]  ←  unweightedASPS(G)
```

```
// For every pair given, we find the cost of the shortest path between {u, v} (given by unweightedASPS)
for each (<u, v>, cost) ∈ min_cost do

        // Find the cost of the shortest DAG path using k = number of edges in shortest path
        // (unweightedASPS)
        min_cost[ ] → max_k   ←   shortestDAGkpath( <u, v>, cost )

// Using the max_k calculated, find the lowest cost for {u, v} using at most max_k steps
for each (<u, v>, max_k) ∈ min_cost do

        // Replace old cost with new minimum cost for each pair {u, v} given by shortestDAGpath
        // using max_k steps
        min_cost[ ] → cost   ←   shortestDAGkpath( <u, v>, max_k )

return min_cost
```

This algorithm assumes that all edges would have a cost corresponding to a positive integer. This is so that we can use the max_k for the second iteration of shortest DAG k-paths. This is because the minimum cost of a path in the weighted graph would have at most max_k edges due to the minimum cost of an edge being 1. Also, we are able to use the shortest DAG k-paths function on this undirected graph as we could represent an undirected graph as a directed graph by representing each edge between (u, v) as two edges, one edge from u to v and another from v to u, both of which would have the same cost.

PROBLEM 7a

```
function siftDown(heap[], n, parent)

    // Get the largest among the node and its children
    left   ←   heap[ 2 * parent + 1 ]
    right  ←   heap[ 2 * parent + 2 ]

    largest   ←   max(parent, left, right)

    // One of the children is larger than the parent
    if not largest == parent then

        // Swap the parent and said largest element
        swap(parent, largest)

        // Sift down to check the parent (now the child of the largest element and is at the
        // position given by the largest index) and its new children
        siftDown(heap, largest)
```

In the main function, we run this function starting from the middle of the array as we do not need to check the lowest layer (the second half of the array) since they do not have children and would definitely satisfy the max heap requirement. The function essentially just checks to find the largest amongst a node and its two children and swap the child and the parent if the child is larger than the parent (both of which can be done in constant time).

The sifting down process is done recursively on the node that had been swapped down to a lower level, and the total cost of this varies depending on which level the node is on. This is because the height of a node would determine the maximum number of times that it would have to be sifted down one of its subtrees. The total cost of the sift down process could be calculated as the summation of the number of elements in a particular level (n/4 on the second lowest level, n/8 for the third lowest, and so on) multiplied by the maximum number of times a term might be sifted down.

This summation can be written as:

$$\left(1*\frac{n}{4}\right)+\left(2*\frac{n}{8}\right)+\cdots+(h*1)=\sum_{k=1}^{h}\frac{\mathrm{kn}}{2^{\mathrm{k+1}}}=\frac{n}{2}\sum_{k=1}^{h}\frac{\mathrm{k}}{2^{\mathrm{k}}}=\frac{n}{2}(2)=n$$

This is because the summation term grows smaller as k increases and this in turns allows the summation to converge to a constant number, 1. In other words, the summation of the term converges as it passes the divergence test:

$$\lim_{\mathrm{k}\to\infty}\frac{k}{2^k}=0$$

Therefore, it is proven that the function is bounded by a time complexity of O(n).

PROBLEM 7b

```
function right_max_heapify(heap[], n, parent)

    // Check if the left child is larger than the right child
    left  ←  heap[ 2 * parent + 1 ]
    right  ←  heap[ 2 * parent + 2 ]

    if left > right  then
        // Swap the left and right child, then check the new left child if it satisfies the max
        // heap requirement
        swap(left, right)
        siftDown(heap, left)

    // Recursively check the left and right children and their grandchildren
    right_max_heapify(heap, left)
    right_max_heapify(heap, right)
```

The above function takes in a heap and sifts down from the root node down its two children, and if the left child is smaller than the right child, we swap them and sift down the left child to ensure it is still a heap. We do not need to sift down the new right child after the swap as it is guaranteed to be larger than the bottom two. As we can tell from the above, sifting down is takes O(n) time.

PROBLEM 8

Professor Dubious's argument is wrong because the children of a node actually depend on the order that the array is parsed into the max-heap function. This is because elements can only be swapped if they are parent and child and only if they are in the same subtree. In other words, a grandchild being swapped with its parent to satisfy the max heap requirement would not affect the grandchild in another subtree if they do not have the same parent.

An example is the if the array [25, 15, 8, 14, 20, 2, 10] is parsed into the function, the output would be [null, 25, 20, 10, 14, 15, 2, 8] and the elements in the first two layers would be 25, 20, and 10, but 10 is not one of the top 3 largest elements in the heap.