CS 430 – Fall 2024
Introduction To Algorithms
Homework #3

1)
```
def find(A, p, r, i):
    if p == r:                      # Base case: if there is only one element
        return A[p]
    M = MEDIAN(A, p, r)

    # Partition the array into three parts
    L = [x for x in A[p:r+1] if x < M]      # Elements less than the median
    R = [x for x in A[p:r+1] if x > M]      # Elements greater than the median
    Mid = (r - p + 1) - len(L) - len(R)     # Elements equal to the median

    # Check if we found the ith smallest element
    if len(L) >= i:
        return find(L, 0, len(L) - 1, i) # Search in the left part
    elif len(L) + Mid >= i:
        return M                            # The median is the ith smallest element
    else:
        return find(R, 0, len(R) - 1, i - len(L) - Mid )  # Search in the right part
```

Since M is the median of A[p…r], each of the subarrays L, and R has at most half the number of elements of A[p…r],. The recurrence for the worst-case running time is

$T(n) = T(n/2) + O(n)$

$T(n) = O(n)$.

2a) The original array has n elements. The middle two-thirds of the array are indexed from $n/6$ to $5n/6-1$. The number of elements that are not in the middle two-thirds is the sum of the elements in the first one-sixth and the last one-sixth:

First one-sixth: $A[0], A[1], \dots, A[n/6 - 1] \rightarrow n/6$ elements.

Last one-sixth: $A[5n/6], A[5n/6 + 1], \dots, A[n - 1] \rightarrow n/6$ elements.

So, the number of values not in the middle two-thirds of the original array is $n/6 + n/6 = n/3$.

After sorting, the middle two-thirds of the array are indexed the same as in the original array. The number of elements not in the middle two-thirds of the sorted array is also $n/3$.

Since both the original array and the sorted array exclude $n/3$ elements from their respective middle two-thirds, the middle two-thirds of the original array and the middle two-thirds of the sorted array must overlap.

The overlap guarantees that there is at least one element that lies in both the middle two-thirds of the original array and the middle two-thirds of the sorted array.

2b) def find_middle_element(A, n):

```
    # Find the n/6-th smallest element (lower bound of middle two-thirds)
    lower_bound = select(A, 0, n-1, n//6)

    # Find the 5n/6-th smallest element (upper bound of middle two-thirds)
    upper_bound = select(A, 0, n-1, 5*n//6)

    # Choose an element between the lower and upper bounds
    # We are guaranteed that at least one element from the middle two-thirds of the original
    array will still be in the middle two-thirds after sorting.
    for element in A:
        if lower_bound <= element <= upper_bound:
            return element
```

select is the linear-time selection algorithm, which can find the i-th smallest element in O(n) time. We use it to find the n/6-th and 5n/6-th smallest element.

The select algorithm that we use only takes O(n) time and the final loop to find an element between the n/6-th and 5n/6-th smallest elements, also takes O(n).

$T(n) = O(n) + O(n) + O(n) = O(n)$

3) We can find the max and min by pairing up the numbers and comparing them within each pair.

For each pair (x,y), perform one comparison to determine which element is larger and smaller:

If $x > y$, then x is a candidate for the maximum, and y is a candidate for the minimum.
If $x < y$, then x is a candidate for the minimum, and y is a candidate for the maximum.

Now, there are two sets of numbers. A set of $n/2$ numbers that are candidates for the minimum and a set of $n/2$ numbers that are candidates for the maximum.

To find the global minimum, we need to perform $(n/2) - 1$ comparisons.
To find the global maximum, we need to perform $(n/2) - 1$ comparisons.

$$\begin{aligned}\text{Total Comparisons} &= n/2 + (n/2 - 1) + (n/2 - 1)\\ &= n/2 + n/2 - 1 + n/2 - 1\\ &= 3n/2 - 2\end{aligned}$$

4) Finding the minimum element using TREE-MINIMUM for a tree with height h takes O(h) time, which in the worst case is O(n).
The n−1 calls to TREE-SUCCESSOR collectively take O(n) time, since each node is visited exactly once.
Thus, the total time complexity is Θ(n)

5) The successor of a node x is the node with the smallest key greater than x and the predecessor of a node x is the node with the largest key smaller than x.

The successor of x is the minimum node in its right subtree.
To find the successor of x, you first move to the right child of x, then continue moving left until you reach a node that has no left child. This node is the minimum node in the right subtree and is the successor of x. If this node had a left child, then there would be a smaller element to its left, contradicting the fact that it is the minimum node.

The predecessor of x is the maximum node in its left subtree.
To find the predecessor of x, you first move to the left child of x, then continue moving right until you reach a node that has no right child. This node is the maximum node in the left subtree and is the predecessor of x. If this node had a right child, there would be a larger element to its right, contradicting the fact that it is the maximum node.

6) In a full binary tree, the number of leaf nodes is always n + 1. This is because each internal node can be thought of as distributing its edges between its two children, ultimately leading to one more leaf than the number of internal nodes.
Even for a binary tree where internal nodes may have only one child, the relationship between internal and leaf nodes holds because adding an internal node with one child increases the number of leaf nodes, while adding an internal node with two children replaces an existing leaf but creates two new leaves, thus maintaining the overall structure where the number of leaves is always n + 1.

7) Counter-example tree:

```
    5
   /
  2
 / \
1   3
```

Assume we are searching for 1.
A = {}, B = {5, 2, 1}, C = {3}
Since 3 from C is less than 5 from B, the professor’s claim does not hold.