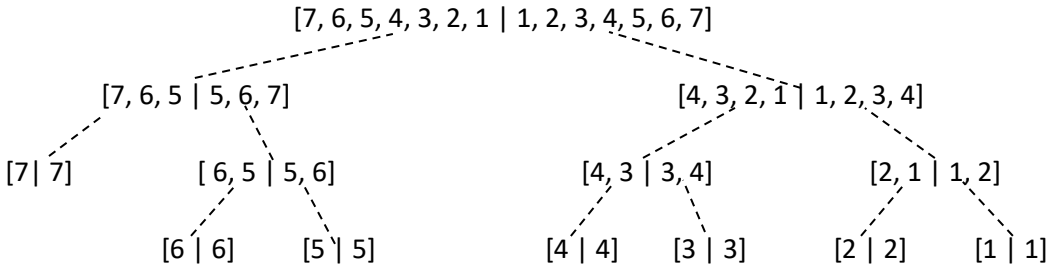
Ochirgarid Chinzorig, Assignment 3B

1. Merge sort recursion tree:



2. The Binary search algorithm calls itself at most one time in every recursion. For every recursion step the gap between lower and upper get halved until it they become equal. In other word, if we say the algorithm did self-call x times then the gap between lower and upper limit can be at most 2^x. If we put it in an equation form

   "the difference between upper and lower bound"=$2^{\text{number of self-call}}$

   from this

   "number of self-call"=log(difference between upper and lower boud)

   Which gives us our algorithm runtime O(log(N)).
3. The run time of the algorithm will be an O(N) since we are accessing each element only once and the self-call is bound with the number of elements.

```
Algorithm ReverseList(arr, idx)
Input: a list with n elements arr, index of current element idx
Output: input array in reversed order
        right_idx <- arr.length – idx – 1
        if right_idx <= idx then                    // recursion end
                return arr

        temp <- arr[idx]
        arr[idx] <- arr[right_idx]
        arr[right_idx] <- temp

        return ReverseList(arr, idx + 1)
```

4. The iterative algorithm that runs O(N)

```
Algorithm FindFibIterate(n)
Input: an integer n
Output: n-th Fibonacci number
        If n < 2 then
                return n
        prev <- 0
        current <- 1
        for i<-2 to n do
                temp <- prev + current
                prev <- current
                current <- temp
        return temp
```

The recursive algorithm with memorization O(N)

```
Algorithm FindFibRecursive(n, fib)
Input: an integer n, HashMap of computed values fib
Output: n-th Fibonacci number
        If fib.containsKey(n) then
                return fib.get(n)
        If n < 2 then
                fib.put(n, n)
                return n
        f1 <- FindFibRecursive(n – 1, fib)
        f2 <- FindFibRecursive(n – 2, fib)
        fib.put(n, f1 + f2)
        return fib.get(n)
```

5. We can use same approach for the ThirdSmallest problem with O(N) run time. We need the first 3 numbers in a sorted order and for the incoming numbers we need to put them in a corresponding position if it is smaller than one of the 3 numbers.

   If we use this technique for the k-th smallest number problem we will get a O(K*N) runtime because the putting in a corresponding position operation will run a O(K) time. For a K<log(N) this approach is faster than sorting, otherwise the sorting is faster.

   For find the k-th smallest number problem we can use QuickSelect algorithm with a O(log N) runtime. We can use pivot partitioning like we used in the QuickSort algorithm and instead of we need only 1 part that contains our answer and we can skip the merge part. With this the recursion will become T(n) = T(n/2) + c which is O(log N).