Question 1: Median of n values using groups

1.1 K groups

Lemma 1. *If we have groups of k, in each iteration we can remove at least* $\frac{n}{k}$ *elements.*

Proof.

$$\text{K-rows}\left\{\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & X & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & 0 \end{array}\right.$$

Following the steps in the algorithm covered in class:

i. Form $\frac{n}{k}$ groups of k elements. (O(n))

ii. Find the median of each group. (O (n))

iii. Recursively find the median among all group medians (T($\lceil \frac{n}{k} \rceil$))

iv. Partition around x. (O(n))

v. If rank(x) is not the target rank, recurse on the subset of elements that contains the target rank.

To figure out the time complexity of step v, we look at the subsets included in the recursion in the worst case. (Assuming n is divisible by k for simplicity). We start with n elements, and since we are sorting in groups of k we know that there are $\frac{n}{k}$ columns. Since these rows and columns are sorted already in a way that the elements to the lower left of our median of medians (X in the diagram) all must be less than X, and the elements to the upper right all must be greater than X. Assuming $k \geq 6$, we know that at worst case our recursion will be:

$$n - (\tfrac{n}{k}) = \tfrac{nk}{k} - \tfrac{n}{k} = \tfrac{nk-n}{k} = \tfrac{n(k-1)}{k}$$

Therefore the worst case time complexity for step v. is T($\frac{n(k-1)}{k}$). Which means that:

$$T(n) = \Theta(n) + T(\tfrac{n}{k}) + T(\tfrac{n(k-1)}{k})$$

□

1.2 Modifying K

If we suppose one really large K, 1,000,000, and one relatively small, 100, modifying the constant K with a smaller value, as noted in the recitation notes, will increase the runtime of the algorithm. Thus it follows that when implementing the algorithm, the larger the K, the more efficient the algorithm. I suggest that a K value of 1,000,000 should be used.

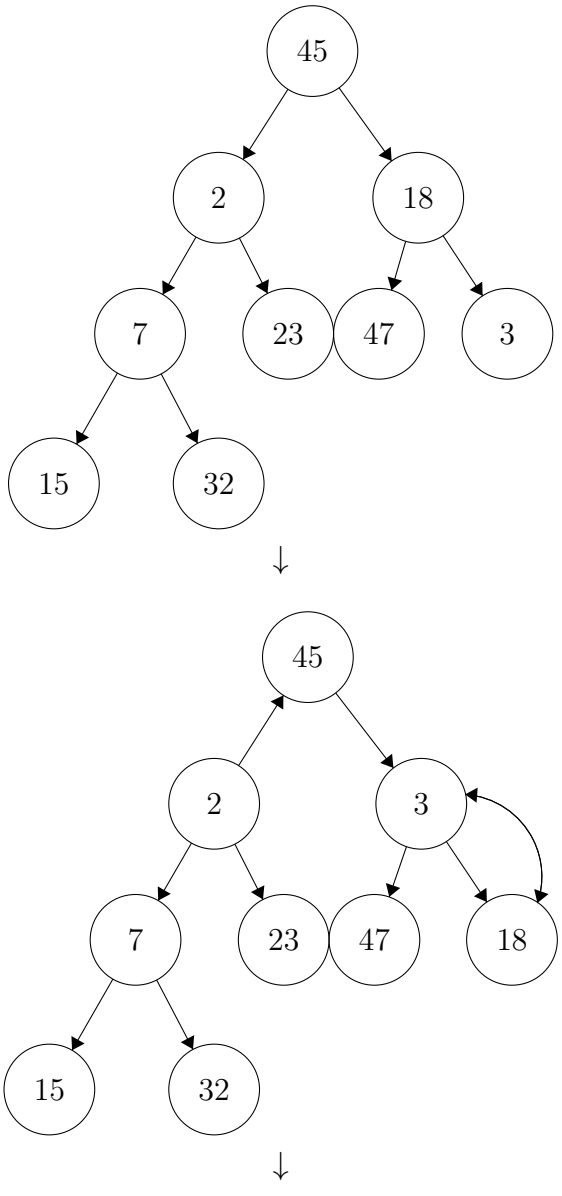
Question 2: Algorithm

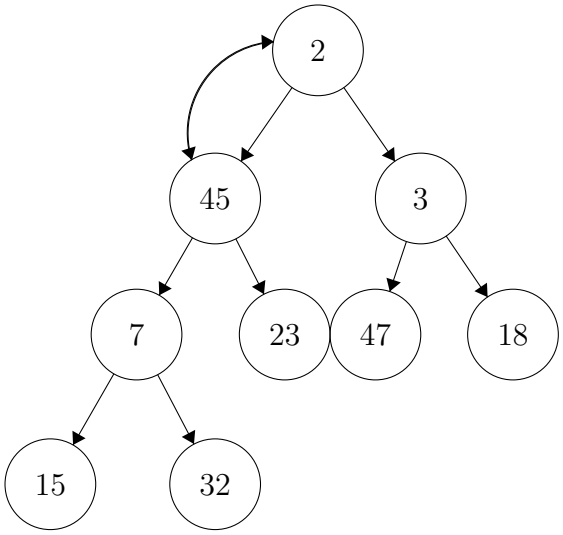
```
bool has_K_equal_values(int Array, int k) {
    // first we must sort the array, any sorting
    // method will work, i.e. insertion
    sortedArray = sort(Array);
    // once we have the sorted array we need to define
    // two variables, one that counts the number of duplicated
    // and one that is the total number of duplicates we are
    // looking for
    int count = 1;
    int total_duplicates = Array.length / k;
    // now we loop through the entire array
    for (int i = 0; i < sortedArray.length; i++) {
        // first we must make sure we dont go out of bounds
        if (i == sortedArray.length - 2) {
            break;
        }
        // now we can safely check if the current value at our
        // position in the array is the same as the next
        // position
        if (sortedArray[i] == sortedArray[i+1]) {
            //add one to our duplicate counter
            count++;
        }
        // if it is not the case that the next value matches
        // current one check for the end condition, the number
        // of counted dupicates matches the total desired one.
        else if (count == total_duplicates) {
            return true;
        }
        // if all else fails reset the counter to one
        // and begin again
        else {
            count = 1;
        }
    }
    // lastly, we need to test for the condition that our set
    // of duplicates occured at the end of our Array, so we
    // need another comparison of counted duplicates to total
    // desired ones
    if (count == total_duplicates) {
        return true;
    }

    // if we are at this point the desired amount of duplicates
    // has not been found
    return false;
}
```

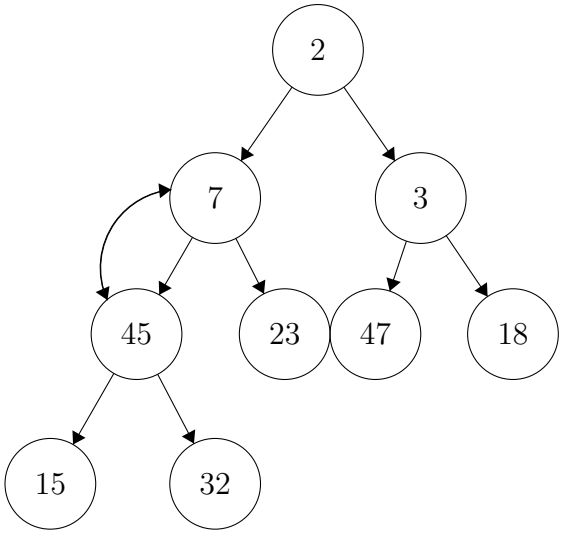
Question 3: Heaps

3.1 Min-Heap Bottom Up

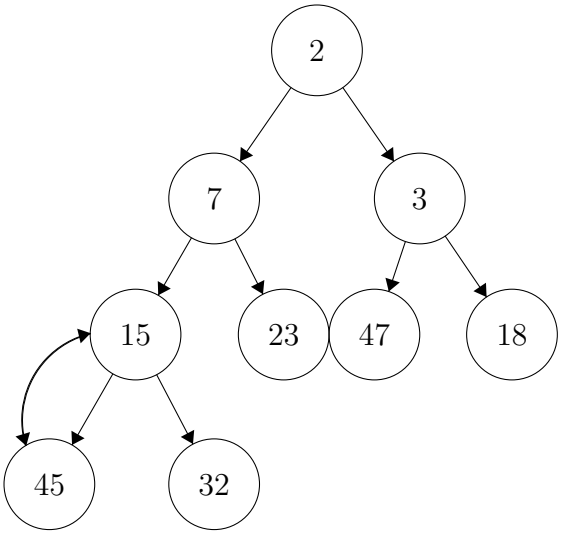



2
45
3
7
23
47
18
15
32

↓

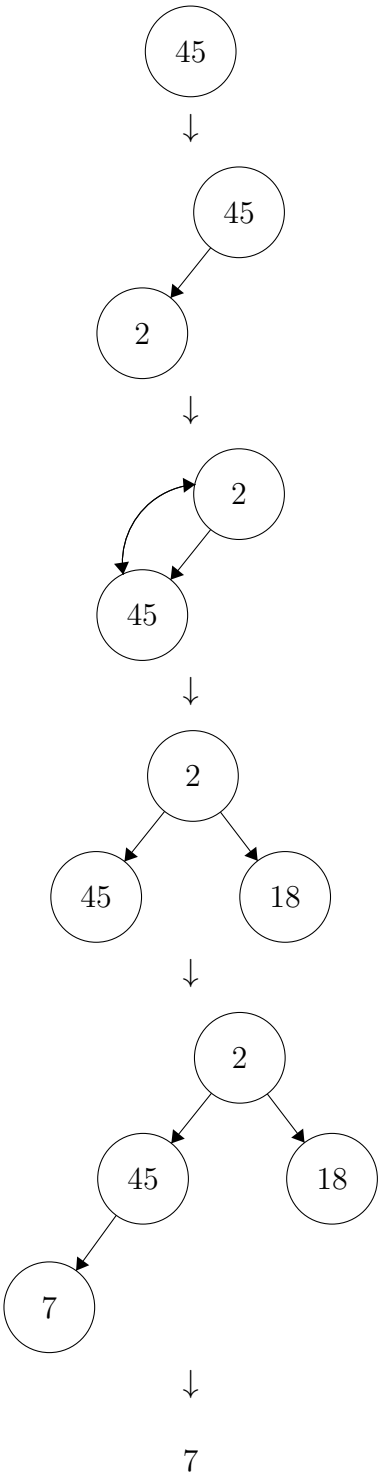

2
7
3
45
23
47
18
15
32

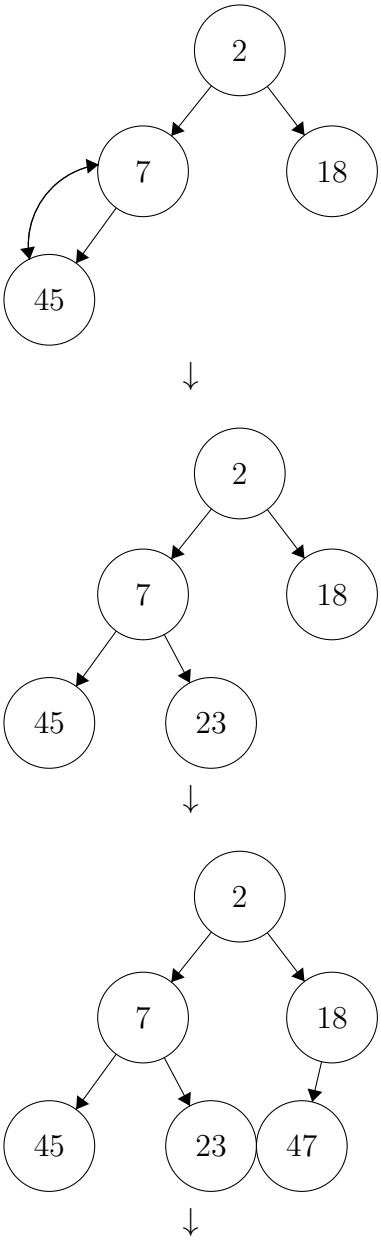
↓

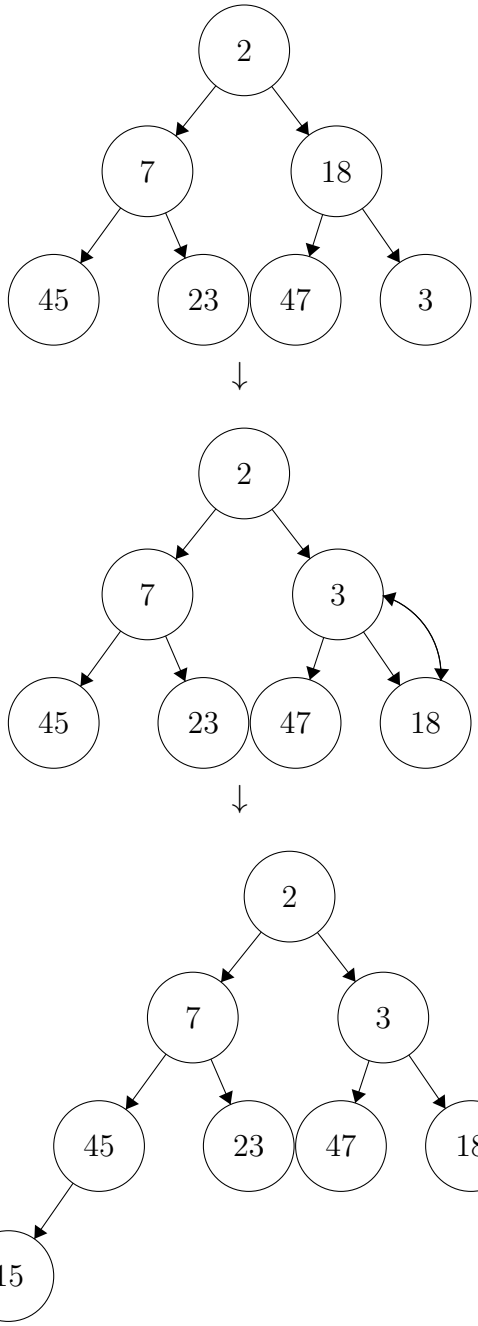


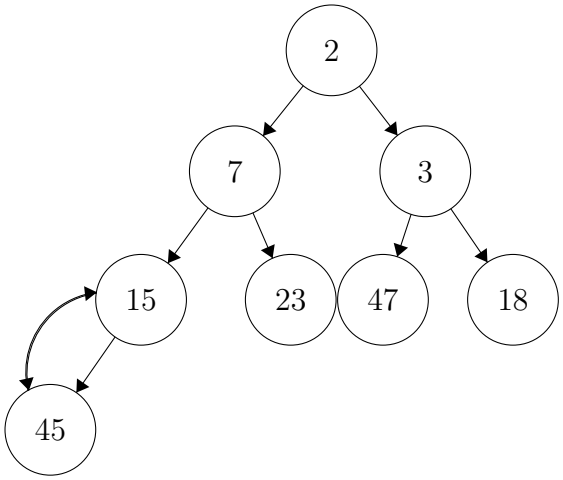
This is the final min-heap with the bottom up method. Each node must be compared to its two daughter nodes so total comparisons made = 8

3.2 Min-Heap Top Down

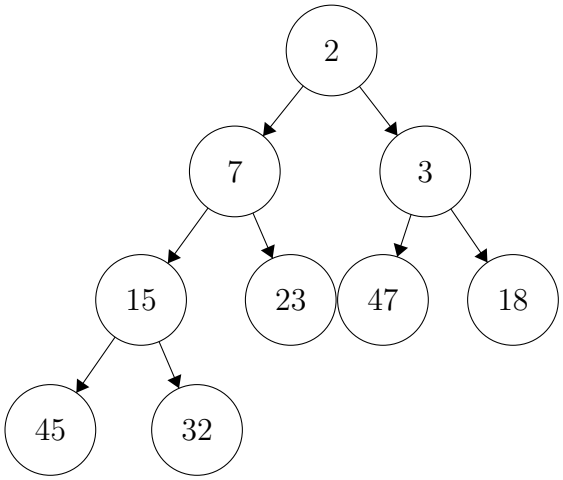



2
7
18
45
↓
2
7
18
45
23
↓
2
7
18
45
23
47
↓


2
7
18
45
23
47
3
↓
2
7
3
45
23
47
18
↓
2
7
3
45
23
47
18
15
↓



↓



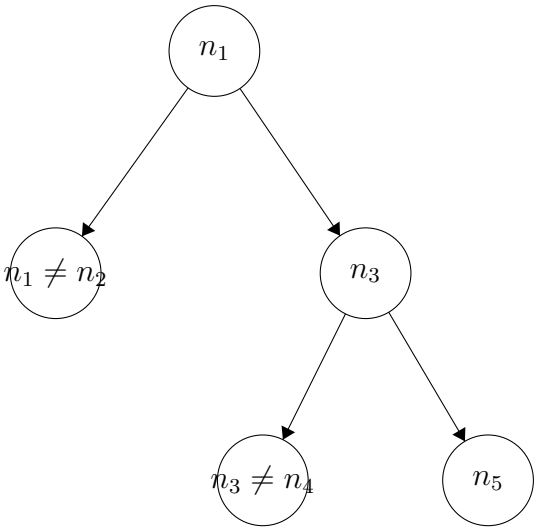
This is the final min-heap with the Top Down method. Each node must be compared to its two daughter nodes so total comparisons made = 8

3.3 Analysis

Yes, the two heaps are identical in the end. I don't believe this is just a peculiar case where both heaps end up exactly the same. Since we are starting with the data in the same order in each case: 45, 2, 18, 7, 23, 47, 3, 15, 32, the resulting construction will be the same no matter which method is used because the comparisons are carried out on the same values just with different core methods. If however the order of the incoming data was swapped, e.g. 2, 3, 45, 7, 32, 15, 47, 18, 23. The resultant min-heap would be different than with the original incoming data set. However both Bottom up and Top down would again produce the same heaps on the new set.

Question 4: Fake Coin

In order to find the asymptotic lower bound for which of the coins is fake, we need to look at an $f(n)$, that regardless of the algorithm used to produce the correct answer, will take a minimum of $f(n)$ weighings to do. Hypothetically, if you select just two coins from the collection and weight them against each other and one is lighter than the other. Then you have found the fake coin in the minimum number of steps possible, but you still needed 1 weighing to do so. So no matter which method you use to uncover the fake coin, it will require 1 weighing and thus a lower bound is $\Omega(1)$.



When you select two coins, and weigh them, you only have two options, they weigh the same or not, if they weigh the same, select another 2 coins and repeat. (Note: this method is far inferior to the grouping method, it would take at worst case $\frac{n}{2}$ weighings to find the fake coin as opposed to $log(n)$, it was used just to illustrate a point). So the best case for any method would also require at minimum 1 weighing.