Lydia Feng
CS323
Written #1: UF and PQ

1. QuickFindUF2
    a. QuickFindUF2 takes $3N + O(1)$ space for a structure of size N, because it requires three arrays of size N each, and the $O(1)$ term includes overhead to keep track of objects.
    b. Consider an arbitrary item i. Suppose that the union operation modifies id[i]. This means that for an arbitrary item q, the size of the component of i is less than or equal to the size of the component of q, since the union operation modifies the id of the component that is smaller. Suppose the size of the component of i is an arbitrary non-negative integer x. Then the size of the component of q is $x + y$ where y is any non-negative integer. When the union operation is called on item i and item q, it will modify id[i], because $x <= x + y$ and the resulting component is $2x + y$. Therefore, the size of i's component at least doubles. Since each union operation at least doubles the size of i's component, we can express the size of the structure as $N = 2^k$. This can be rewritten to find the number of times the component of i is doubled: $k = log_2 N$, where k is the number of times id[i] is modified. So id[i] is modified $O(\log N)$ times.
    c. If we construct a QuickFindUF2 structure of size N and perform M operations of unions or finds, the total running time is O(M + NlgN). This is because id[i] is modified O(lgN) times and there are at most N number of ids which must be modified. This gives us our O(NlogN) term. The find operation takes constant time each for a total of M operations. The connected operation also takes constant time for a total of M operations so all operations would take O(M) time. In total, a structure of size N with M operations takes O(M + NlgN) time.

2. *d*-ary heap
    a. Given an item at array index k, where k is a non-negative integer, its leftmost child is at array index $kd + 1$.
    b. Given an item at array index k, where k is a non-negative integer, its parent is at array index $\lfloor (k - 1) / d \rfloor$.
    c. Given a d-ary heap of n items, the largest index k such that k has at least one child is the index $\lfloor (n - 1) / d \rfloor$.
    d. Give a d-ary heap of n items, the height of the tree is $\lceil log_d(nd - n + 1) - 1 \rceil$.

3. Binary heap
    a. Insert all N numbers into a binary max heap implementation with an array of size N. Then sort the heap by sinking the parents starting from the bottom of the heap or the right of the array. So if the parent is smaller than the child, swap the two so that each mini-tree is sorted from the bottom up. Do this until the entire heap is sorted. And remove the top M numbers in the heap or the left-most numbers in the array as they are the greatest M numbers.

        M*lg*N = O(N) because removing the maximum requires 2*log*N comparisons and M numbers must be removed so it requires 2M*log*N comparisons.

        This takes O(N) time, because it takes O(N) time to construct heap order because there are at most N exchanges and at most 2N comparisons.

        This takes O(N) space, because the binary heap needs to hold all N numbers.

    b. Create a binary min heap implementation with an array of size M from the first M numbers from the stream, and sort so that each parent is less than or equal to both of its children. For each number in the stream, check it against the minimum number in the heap (at the root or the left-most element in the array). If the number is less than the minimum in the heap, skip that number and move on to the next number in the stream. Otherwise, replace the minimum in the heap with the new number and sink the new number up until the heap is sorted.

        This takes O(NlogM) time. Checking to see if the number is greater than the minimum takes constant time. Inserting the number into the heap takes O(logM) time since the heap is restricted to size M. At most, N numbers must be inserted, so it takes O(NlogM) time.

        This takes O(M) space, because the binary heap only ever holds M items.