1 Potpourri

1.1 Each of the following sequences represent an array being sorted at some intermediate step. Match each sample with one of the sorting algorithms: **insertion sort, selection sort, heapsort, merge sort, quicksort**. The original array is below.

```
5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
```

(a)
```
5103 9914 0608 3715 2261 6035 7188 9797 1163 4411
0608 2261 3715 5103 6035 7188 9797 9914 1163 4411
```

Merge sort

(b)
```
0608 1163 5103 3715 6035 2261 9797 7188 9914 4411
0608 1163 2261 3715 6035 5103 9797 7188 9914 4411
```

Selection sort

(c)
```
9797 7188 5103 4411 6035 2261 0608 3715 1163 9914
4411 3715 2261 0608 1163 5103 6035 7188 9797 9914
```

Heapsort

(d)
```
5103 0608 3715 2261 1163 4411 6035 9914 9797 7188
0608 2261 1163 3715 5103 4411 6035 9914 9797 7188
```

Quicksort

(e)
```
0608 5103 9914 3715 6035 2261 9797 7188 1163 4411
0608 2261 3715 5103 6035 9914 9797 7188 1163 4411
```

Insertion sort

1.2 Give the *amortized runtime analysis* for `push` and `pop` for the priority queue below.

```
class TwinListPriorityQueue<E implements Comparable> {
    ArrayList<E> L1, L2;
    void push(E item) {
        L1.push(elem);
        if (L1.size() >= Math.log(L2.size())) {
            L2.addAll(L1);
            mergeSort(L2);
            L1.clear();
        }
    }
    E pop() {
        E min1 = getMin(L1);
        E min2 = L2.poll();
        if (min1.compareTo(min2) < 0) {
            L1.remove(min1);
            return min1;
        } else {
            L2.remove(min2);
            return min2;
        }
    }
}
```

Let N be the number of elements in the priority queue. Then the amortized runtime for `push` is in $O(N)$ as the cost for every $\log N$ insertions is in $O(\log N \cdot 1 + 1 \cdot N \log N)$ which simplifies to $O(N)$. Note that the size of `L1` is always constrained to be in $O(\log N)$.

The amortized runtime for `pop` is also in $O(N)$. `getMin` on the unsorted list, `L1`, is in $O(\log N)$, as with `L1.remove(min1)`. Polling from the front of `L2` is in $\Theta(1)$. The most expensive component is `L2.remove(min2)` which is in $O(N)$.

1.3 You have been hired by Alan to help design a priority queue implementation for *Kelp*, the new seafood review startup, ordered on the timestamp of each `Review`.

Describe a data structure that supports the following operations.

- `insert(Review r)` a `Review` in $O(\log N)$.
- `edit(int id, String body)` any one `Review` in $\Theta(1)$.
- `sixtyOne()`: return the sixty-first latest `Review` in $\Theta(1)$.
- `pollSixtyOne()`: remove and return the sixty-first latest `Review` in $O(\log N)$.

Maintain a max-heap called `firstSixtyOne` with 61 `Reviews`, a min-heap called `olderReviews` with all the rest, and a `HashMap` mapping any given integer `id` to its corresponding `Review`.

1.4 Find the Huffman encoding for the following alphabet and set of frequencies.

$\{(a, 0.12), (b, 0.38), (c, 0.1), (e, 0.25), (f, 0.06), (d, 0.05), (g, 0.01), (h, 0.03)\}$

When you build up your Huffman tree, you should place the branch of lower weight on the left. A left or right branch should respectively correspond to a 0 or 1 in the codeword.

$$
\begin{aligned}
a &= 1111 \\
b &= 0 \\
c &= 1110 \\
d &= 11011 \\
e &= 10 \\
f &= 1100 \\
g &= 110100 \\
h &= 110101
\end{aligned}
$$