1 How low can you go?

- 4!
- \(1/2\)
- \(\lceil \log_2 4! \rceil = 5\)
- \(\lceil \log_2 n! \rceil = O(n \log n)\)

2 Simply the best

```java
public static void sortFour(int[] a) {
    assert a.length == 4;
    // compare (0 and 1) and (2 and 3), swap as needed
    if (a[0] > a[1]) {
        swap(a, 0, 1);
    }
    if (a[2] > a[3]) {
        swap(a, 2, 3);
    }
    // check and possibly swap min’s and max’s
    if (a[0] > a[2]) {
        swap(a, 0, 2); // swap min’s
    }
    if (a[1] > a[3]) {
        swap(a, 1, 3); // swap max’s
    }
    // compare max of min’s w/ min of max’s
    if (a[1] > a[2]) {
        swap(a, 1, 2);
    }
}
```
3 Sorting donkeys and elephants

See partition function of quick sort. Once pivot value is selected, the array contains elements of two types: those less than or equal to the pivot, those greater than the pivot.

4 Stimulating simulating of sorting algorithms

4.1 Selection sort

- 28
- \( f(n) = n - 1 + n - 2 + n - 3 + \ldots + 1 = (n + 1)n/2 \) for even \( n \); \( f(n) = O(n^2) \).

4.2 Insertion sort

- 12
- While both sort by “growing” a sorted list from left to right, they differ in how they find the next element. Selection sort finds the minimum of the remaining. Insertion sort simply inserts the next element into the sorted order. When the list is partially sorted, this latter approach requires fewer comparisons.
- Sorted list is best-case, \( O(n) \) time. Worst-case is reverse sorted list, \( O(n^2) \) time.
- Bad things happen if you replace the line that says \( a[k] = \text{temp} \) with \( \text{swap}(a, i, k) \)! The value at \( a[i] \) may change during the execution of the inner loop – this is why \( \text{temp} \) is assigned its value before the inner loop.

4.3 Quicksort

- \( O(n^2) \) because the choice of pivot ensures that in each recursive call, the array is split into one subarray of size 1 and another subarray of size \( n - 1 \). This is not an effective way to divide and conquer!
- Even when we select the actual median, this implementation performs poorly if many elements are exactly equal to the median. A better approach is to write the partition algorithm differently so that even the pivot ends up in the middle, even when many elements are equal to the pivot. See partition2 below.
• The pivot swaps position only once. Observe that once the pivot is placed into position, this will be its final position – even though the left and right subarrays still need to be sorted, the pivot does not move.

4.4 Code

```java
// invariants:
// pivot value is a[left]
// every thing left of fromLeft is <= x
// every thing right of fromRight is >= x
// a[fromLeft...fromRight] is unknown
public static int partition2(int[] a, int left, int right) {
    int pivot = a[left];
    int fromLeft = left+1;
    int fromRight = right;
    while (true) {
        while ( a[fromRight] > pivot ) {
            fromRight--;
        }
        while ( fromLeft < right && a[fromLeft] < pivot ) {
            fromLeft++;
        }
        if ( fromLeft >= fromRight ) {
            break;
        } else {
            swap(a, fromLeft, fromRight);
            fromLeft++;
            fromRight--;
        }
    }
    swap(a, left, fromRight);
    return fromRight;
}
```