1. Memory.

\[ \sim 48n \text{ bytes} \]

Each \texttt{Node} object requires 48 bytes: 16 (object overhead) + 16 (two references) + 8 (double) + 4 (int) + 4 (padding). In total the \( n \) \texttt{Node} objects consume 48\( n \) bytes.

2. Five sorting algorithms.

0  original array  
5  quicksort (after first partition)  
1  selection sort (after 12 iterations)  
3  mergesort (just before left half of the array is sorted)  
2  insertion sort (after 16 iterations)  
4  heapsort (after heap construction phase and putting 6 largest keys into place)  
6  sorted array  

3. Analysis of algorithms.

(a) \( \sim 2n^2 \)

Selection sort makes \( \sim \frac{1}{2}m^2 \) compares to sort any array of length \( m \). Here, \( m = 2n \).

(b) \( \sim n^2 \)

Each integer \( i \) in the right half is inverted with with \( n-i \) integers in the left half and the same \( n-i \) integers in the right half. So, the number of inversions is

\[ 0 + 2 + 4 + \ldots + 2(n - 1) \sim n^2. \]

The number of compares in insertion sort is always within \( n \) of the number of inversions.

(c) \( \sim n \log_2 n \)

Recall that the best case for a merge happens when all of the keys in one subarray are larger than all of the keys in the other subarray. Sorted arrays always result in best-case merges, as do reverse-sorted arrays. As a result, sorting the left half (a sorted array of length \( n \)) takes \( \frac{1}{2}n \log_2 n \) compares and sorting the right half (a reverse sorted array of length \( n \)) takes \( \frac{1}{2}n \log_2 n \) compares. Merging them together takes an extra \( 2n - 1 \) compares.

With tilde notation, be sure to include the leading coefficient and the base of the logarithm and to discard lower-order terms.
   (a) 3 6 14 16
   (b) 4 5 6 7 9 13 14

5. Red–black BSTs.
   22 color flip → 18 rotate left → 24 rotate right → 22 color flip → 14 rotate left

6. Data structure and algorithm properties.
   (a) \( n^4 \)
       Each computational experiment involves opening about \( 0.593n^2 \) sites, where 0.593 is the percolation threshold. Opening a site takes a constant number of union and find operations. Since there are \( n^2 \) sites, the quick-find data structure has \( n^2 \) elements. So, each find operation makes 1 array access and each union operation makes about \( n^2 \) array accesses.
       Even in the worst case, if \( 0.593n^2 \) sites are opened, a constant fraction of them will have one (or more) open neighbors, each of which triggers a union operation.
   (b) \( n \)
       The amortized number of array accesses per operation is bounded by a some constant \( c > 0 \). So, starting from an initially empty data randomized queue, any sequence of \( n \) operations makes at most \( cn \) array accesses.
   (c) \( \log n \)
       The range count requires two (deluxe) binary searches in a sorted array of length \( n \).
   (d) exponential
       The A* algorithm with the Manhattan priority function is incapable of solving even some 5-by-5 puzzles in a reasonable amount of time.
   (e) \( n \log n \)
       Inserting a sequence of \( n \) keys in ascending order into a binary heap takes \( \sim n \log_2 n \) compares. (Each insert and delete-the-max operation makes at most \( 2 \log_2 n \) compares, so no sequence of \( n \) operations makes more than \( 2n \log_2 n \) compares.)
   (f) \( n \log n \)
       The height of any binary tree on \( n \) nodes is at least \( \log_2 n \).
   (g) \( n^2 \)
       Consider a sequence of \( n \) insert operations in which each of the \( n \) keys has the same hash code.
7. System sort.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort</th>
<th>dual-pivot quicksort</th>
<th>Timsort</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stable.</strong></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td><strong>In-place.</strong></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td><strong>At most $\sim n \log_2 n$ compares.</strong></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td><strong>Linear number of compares on arrays with only 3 distinct keys.</strong></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td><strong>Linear number of compares on arrays in ascending order.</strong></td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

8. Duplicate in two arrays.

The key idea is to sort the smaller array and use binary search to check for duplicates.

1. Heapsort $a[]$.
2. For each $j$, binary search for $b[j]$ in $a[]$. If a search hit, then return $b[j]$ since it appears in both arrays.

Heapsorting $a[]$ takes $m \log m$ time and uses constant extra space. Binary searching for $b[j]$ in $a[]$ takes $\log m$ time (for a total of $n \log m$ time). Standard binary search (nonrecursive) uses only constant extra space.

Some alternative approaches that don’t meet the performance requirements:

- Using mergesort instead of heapsort (linear extra space).
- Using quicksort instead of heapsort (logarithmic extra space for the recursion and does not achieve a linearithmic running time in the worst case).
- Using a red–black BST or a hash table (linear extra memory).
- Heapsorting both $a[]$ and $b[]$ and then checking for duplicates with a merge operation ($n \log n$ time instead of $n \log m$).
9. Data structure design.

The main idea is to maintain a hash table (such as java.util.HashMap) for each list, with key = integer and value = number of times the integer appears in the list. Also maintain a duplicate counter that counts the number of integers that appears in both lists.

- Increment the duplicate counter whenever
  - an integer is added to a list for the first time and
  - it also appears in the other list
- Decrement the duplicate counter whenever
  - an integer is deleted from a list and
  - it is the last such integer in the list and
  - it appears in the other list

```java
public class Duo {
    private HashMap<Integer, Integer> list1 = new HashMap<>();
    private HashMap<Integer, Integer> list2 = new HashMap<>();
    private int duplicates = 0;

    public void addToList1(int x) {
        if (!list1.containsKey(x)) {
            list1.put(x, 1);
            if (list2.containsKey(x)) duplicates++;
        } else list1.put(x, list1.get(x) + 1);
    }

    public void deleteFromList1(int x) {
        if (list1.get(x) == 1) {
            list1.remove(x);
            if (list2.containsKey(x)) duplicates--;
        } else list1.put(x, list1.get(x) - 1);
    }

    public boolean hasDuplicate() {
        return duplicates > 0;
    }
}
```

Without deletion, it suffices to maintain a hash set (such as java.util.HashSet) for each list containing the set of integers in that list. Also, maintain a boolean variable that indicates whether there exists an integer that appears in both lists.

- Set the boolean variable to true whenever
  - an integer is added to a list for the first time and
  - that integer also appears in the other list