1. Analysis of algorithms.
   (a) $\frac{3}{800,000,000} N^3$
   (b) $N^3$

2. Data structure and algorithm properties.
   (a) $E$ Min height of a binary heap with $N$ keys.
       $E$ Max height of a binary heap with $N$ keys.
       $C$ Min height of a 2-3 tree with $N$ keys.
       $E$ Max height of a 2-3 tree with $N$ keys.
       $E$ Min height of left-leaning red-black BST with $N$ keys.
       $F$ Max height of left-leaning red-black BST with $N$ keys.
       $A$ Min height of a weighted quick union tree with $N$ items.
       $E$ Max height of a weighted quick union tree with $N$ items.
       A. $\sim 1$
       B. $\sim \frac{1}{2} \lg N$
       C. $\sim \log_3 N$
       D. $\sim \ln N$
       E. $\sim \lg N$
       F. $\sim 2\lg N$
       G. $\sim 2\ln N$
       H. $\sim N$

   (b) insertion sort and top-down mergesort are parsimonious
       Selection sort counterexample: C B A. The keys B and C get compared twice, once in first iteration and once in second iteration.
       Heapsort counterexample: C B A. The keys A and B get compared twice, once in the heap construction phase (when sinking C) and once in the sortdown phase (when sinking A after C and A are exchanged).

3. Data structures.
   (a) • Best case: $\sim 2N$
       When the array is full.
       • Worst case: $\sim 8N$
       When the array is one-quarter full.

<table>
<thead>
<tr>
<th>operation</th>
<th>description</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt(int i)</td>
<td>return the ith character in sequence</td>
<td>1</td>
</tr>
<tr>
<td>deleteCharAt(int i)</td>
<td>delete the ith character in the sequence</td>
<td>$N$</td>
</tr>
<tr>
<td>append(char c)</td>
<td>append c to the end of the sequence</td>
<td>1</td>
</tr>
<tr>
<td>set(int i, char c)</td>
<td>replace the ith character with c</td>
<td>1</td>
</tr>
</tbody>
</table>
4. 8 sorting and shuffling algorithms.

7 9 3 5 4 2 6 8

5. Red-black BSTs.

(a) U V W X

(b) P S Y

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>N</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotLeft()</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rotRight()</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>flipColors()</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

6. Hashing.

(a)  

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>G</td>
<td>D</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

(b) I. Possible.

Consider the order F D B G E C A.

II. Impossible.

No key is in the correct position.

III. Impossible.

We can assume B and G were inserted first since they are in correct position. But then third key inserted is guaranteed to be in correct position.

7. Comparing two arrays of points.

(a) • Sort a[] using heapsort (using the point’s natural order).

• For each point b[j], use binary search to search for it in the sorted array a[], incrementing a counter if found.

(b) $N \log M$.

The running time is $M \log M$ for the sort and $N \log M$ for the $N$ binary searches. Since $N \geq M$, the latter term is the bottleneck.

(c) 1.

Both heapsort and binary search use at most a constant amount of extra space.
8. Randomized priority queue.

- **sample()**: Pick a random array index \( r \) (between 1 and \( N \)) and return the key \( a[r] \).

- **delRandom()**:
  - *Select*: pick a random array index \( r \) (between 1 and \( N \)) and save away the key \( a[r] \), to be returned.
  - *Delete*: exchange \( a[r] \) and \( a[N] \) and decrement \( N \).
  - *Restore heap order invariants*: call \( \text{sink}(r) \) and \( \text{swim}(r) \) to fix up any heap order violation at \( r \). Note that \( a[N] \) in the original heap need not be the largest key, so the call to \( \text{swim}(r) \) is necessary.

```java
public Key sample() {
    int r = 1 + StdRandom.uniform(N);  // between 1 and N
    return a[r];
}

public Key delRandom() {
    int r = 1 + StdRandom.uniform(N);  // between 1 and N
    Key key = a[r];  // save away
    exch(r, N--);    // to make deleting easy
    sink(r);        // if a[N] was too big
    swim(r);        // if a[N] was too small
    a[N+1] = null;   // avoid loitering
    return key;
}
```