1. **Comparing playing cards** We would like to sort playing cards from a deck. Associated with each card is a denomination (1 to 13) and a suit (CLUBS | DIAMONDS | HEARTS | SPADES). A card c1 is considered less than a card c2 if either of the following is true:

- the suit of c1 is less than the suit of c2, or

- c1 and c2 are of the same suit, but the denomination of c1 is less than the denomination of c2. The Card class is implemented in Java as follows. Complete the compareTo() function, implementing the ordering described on the previous page, and assuming that the argument is not null.

```java
public class Card implements Comparable {
    // Comparators by suit and by denomination
    public static final Comparator SUIT_ORDER = new SuitOrder();
    public static final Comparator DENOM_ORDER = new DenomOrder();
    // Suit of the card (CLUBS = 1, DIAMONDS = 2,
    // HEARTS = 3, SPADES = 4)
    private final int suit;

    // Denomination of the card
    private final int denom;

    public Card(int suit, int denom) {
        if (suit < 1 || suit > 4)
            throw new IllegalArgumentException("Invalid suit");
        if (denom < 1 || denom > 13)
            throw new IllegalArgumentException("Invalid denomination");
        this.suit = suit;
        this.denom = denom;
    }

    public int compareTo(Card that) {
        ...
Solution: One solution is:

```java
public int compareTo(Card that) {
    if (this.suit < that.suit)
        return -1;
    if (this.suit > that.suit)
        return 1;
    if (this.denom < that.denom)
        return -1;
    if (this.denom > that.denom)
        return 1;
    return 0;
}
```

Recommended Rubric:

- Total value: 10 points
- Full points:
  - algorithm correctly sorts by suit first, then denom
    - returns -1 if this < that, 1 if this > that, and 0 otherwise
- -5 points if sorts by denom first, then by suit
- -5 points if return values are incorrect

2. Sorting playing cards

We would like to sort playing cards from a deck. Associated with each card is a denomination (1 to 13) and a suit (CLUBS \(\parallel\) DIAMONDS \(\parallel\) HEARTS \(\parallel\) SPADES). A card \(c_1\) is considered less than a card \(c_2\) if either of the following is true:

- the suit of \(c_1\) is less than the suit of \(c_2\), or
- \(c_1\) and \(c_2\) are of the same suit, but the denomination of \(c_1\) is less than the denomination of \(c_2\).

Suppose that the variable cards is an array of cards. We could sort it, using your compareTo function, with a call to MergeX.sort(cards). Which of the following code fragments would produce an equivalent final result? Circle all equivalent code fragments.

- Option 1:
  ```java
  MergeX.sort(cards, Card.SUIT_ORDER);
  MergeX.sort(cards, Card.DENOM_ORDER);
  ```

- Option 2:
• Option 3:

```c
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards, Card.SUIT_ORDER);
```

• Option 4:

```c
MergeX.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```

• Option 5:

```c
Quick.sort(cards, Card.SUIT_ORDER);
Quick.sort(cards, Card.DENOM_ORDER);
```

• Option 6:

```c
Quick.sort(cards, Card.DENOM_ORDER);
Quick.sort(cards, Card.SUIT_ORDER);
```

• Option 7:

```c
MergeX.sort(cards);
Quick.sort(cards, Card.SUIT_ORDER);
```

• Option 8:

```c
Quick.sort(cards, Card.DENOM_ORDER);
MergeX.sort(cards);
```

**Solution:** Options 2, 3, 4, 8

**Recommended Rubric:**

- Total value: 16 points
- -2 points for each incorrectly classified option
3. **Backing off to insertion sort**  One strategy to speeding up quicksort is to stop quicksort when it reaches subarrays of size 10 or smaller. After quicksort completes, we run insertion sort on the entire array to make sure that everything is in order. Prove that applying insertion sort to the entire array is a linear time step, despite insertion sorts worst case $N^2$ performance.

**Solution.** We can prove that insertion sort is a linear time step by bounding the number of inversions that can appear in the array once quicksort has reached subarrays of size 10 or smaller.

Let $a$ denote the input array of length $N$. Let $p_i$ denote a subarray of $a$ once the algorithm’s stopping point has been reached, and let $l_i$ denote the length of subarray $p_i$. Based on the stopping criteria, $l_i \leq 10$. Because the subarrays are not overlapping, $\sum_i l_i = N$. Thus, it follows that $i \geq N/10$.

Quicksort guarantees that items to the left of each partition are no greater than the partitioning element, and items to the right of each partition are no less than the partitioning element. Therefore there can be no inversions between partitions, and it is enough to bound the number of possible inversions within each partition. The maximum number of inversions that can appear in any subarray $p_i$ is $\frac{l_i(l_i-1)}{2}$ (which occurs when the elements are in reverse order). The maximum total number of inversions in $a$ when the stopping criteria has been reached is then $\#Inv \leq \sum_i \frac{l_i(l_i-1)}{2}$.

The maximum number of inversions in $a$ occurs when the partitions are as long as possible (10). When this is the case, $i$ is at its minimum ($N/10$). The number of inversions in this case is:

$$\#Inv \leq \sum_i \frac{l_i(l_i-1)}{2} \leq \frac{1}{2} \sum_{N/10} 10(9) \leq \frac{1}{2} \cdot \frac{N}{10} \cdot 90 \leq 4.5N$$

Thus we have bounded the number of inversions possible in $a$ once the stopping criteria is reached to be linear in $N$. We already know from chapter 2.1 that the run time of insertion sort is directly proportional to the number of inversions that appear in the input array. Since the number of inversions is linear in $N$, the run time of insertion sort once this quicksort stopping criteria is reached is also linear in $N$.

**Recommended Rubric.**

- Total value: 25 points
  - -10 for not bounding number of inversions to be linear in $N$
  - -10 for not explaining that quicksort guarantees each subarray is bounded by the partitioning elements to either side
4. Detecting a duplicate  Given k sorted arrays containing N keys in total, design an algorithm that determines whether there is any key that appears more than once. Your algorithm should use extra space at most proportional to k. For full credit, it should run in time at most proportional to N log k in the worst case; for partial credit, time proportional to Nk.

Give a crisp and concise English description of your algorithm. Your answer will be graded on correctness, efficiency, and clarity.

What is the order of growth of the worst case running time of your algorithm as a function of N and k? Briefly justify your answer.

Solution. The main idea is to consider the N keys in ascending order by constructing an index priority queue (such as the IndexMinPQ on pages 320-322) of size k, to hold the minimum unseen item from each of the k arrays.

- Scan through adjacent entries in each of the k sorted arrays to check for any duplicate key within one of the original sorted arrays. If a duplicate is detected, stop.

- Construct an IndexMinPQ of size k. Initialize the PQ by inserting the minimum item from each of the k arrays, and associating it with an index i that refers to its original array (insert(i, item)).

- Create a variable lastmin that will hold the value of the last minimum item removed from the PQ. Initialize it to be an arbitrary number (not equal to any of the items inserted into the PQ in step 2).

- While the PQ is not empty:
  - Remove the minimum item (let’s call it min) from the PQ, and its associated array index, i = delMin(). Check whether min == lastmin. If this is true, then min is a duplicate, so stop.
  - Otherwise, if min is not a duplicate, update lastmin = min.
  - If array i is not empty, retrieve the next item from array i and insert it into the PQ (insert(i, item)).

- Once the PQ is empty and there are no items left in any array, we know there are no duplicates.

The amount of space is proportional to k because the PQ contains at most k items and we need to maintain one index into each of the k sorted arrays, an an additional variable holding the value of the lastmin item.

The total running time is proportional to N log k. The bottleneck operations are insert and delMin. The cost per each of those operations is log k because the PQ has at most k items. We run at most N iterations of the algorithm (until all items in all arrays are exhausted), so the total running time is proportional to N log k.
5. Picking the right sorting algorithm for the job  Choose appropriate sorting algorithm for each type of input described below, and give a one sentence explanation why it is the best choice. For the sorting algorithms, you can pick from among Radix Sort, Mergesort, insertion sort, quick sort, selection sort, and 3-way quicksort. Please pick all that apply. An algorithm can be chosen multiple times for different questions.

A. I want to sort the list of CIS 121 students so that they grouped first by recitation (in ascending order), and their names appear alphabetically within each recitation. Which algorithm(s) should I use?

B. NASA is processing noisy satellite data. Occasionally some packets arrive in a different sequence than when they were sent by the satellite. They want to sort the packets by their timestamp instead of their arrival time.

C. You are performing sorting on an embedded device with limited memory.

D. The Social Security Administration wants to sort all Americans by their birth year.

Solution.

A. The list will need to be sorted twice. The first pass should sort by names, the second pass should sort by recitation number. Anything can be used for the first sort. The second sort needs to be stable. Stable sorts are: insertion sort, mergesort, radix sort, 3-way quicksort. Incorrect answers are quicksort and selection sort, since they are not stable.

B. Insertion sort is best for processing NASA’s satellite data, since it will be largely ordered to begin with and insertion sort can run in near linear time on partially ordered data.

C. Quicksort is the best choice on a low memory device, because we should select an in-place sorting algorithm (and it is faster in expectation than the other in-place sorting algorithms above)

D. 3-way quicksort is the best choice, since it improves quicksort when there are lots of duplicate keys. Radix sort is also an acceptable choice for this.