The homework assignment is available at:
`http://www.jennylam.cc/courses/146-s17/homework03.html`

1. (a) Here is a working Java solution which runs in linear time in the total length of the input lists.

```
public static List<Integer> minDistance(List<Integer> a, List<Integer> b,
    List<Integer> c) {
     if (a.isEmpty() || b.isEmpty() || c.isEmpty())
         return null;
     int ia = 0;
     int ib = 0;
     int ic = 0;
     int bestA = a.get(ia);
     int bestB = b.get(ib);
     int bestC = c.get(ic);
     int minDist = max(max(a), max(b), max(c)) - min(min(a), min(b), min(
        c));

     while (ia < a.size() && ib < b.size() && ic < c.size()) {
         int newMin = min(a.get(ia), b.get(ib), c.get(ic));
         int newMinDist = max(a.get(ia), b.get(ib), c.get(ic)) - newMin;
         if (minDist > newMinDist) {
             minDist = newMinDist;
             bestA = a.get(ia);
             bestB = b.get(ib);
             bestC = c.get(ic);
         }
         if (a.get(ia) == newMin)
             ia++;
         else if (b.get(ib) == newMin)
             ib++;
         else
             ic++;
     }
     return new ArrayList<>(Arrays.asList(bestA, bestB, bestC));
 }
```

(b) In the following solution, we use a SortedMap rather than a SortedSet in order to include auxiliary information, namely the index of the list whose pointer is going to be incremented next. The same idea can be implemented directly with a SortedSet and together with a custom class.

For the sake of clarity, this code does not handle the case when the same integer appears in more than one list, because the values are used as keys, and keys must be unique to be stored in the SortedMap. This problem can be remedied in one of several ways (although these are implementation details that do not affect the time complexity of the algorithm), all of which involve using or implementing a sorted multi-map data structure, to map a key to more than one value. For example, one is available in the Guava library.

The functions minMin and maxMax return the minimum and maximum element in the list of lists of integers.

```
 1 public static List<Integer> minDistance(List<List<Integer>> lists) {
 2     List<Integer> pointers = new ArrayList<>();
 3     List<Integer> bestTuple = new ArrayList<>();
 4     SortedMap<Integer, Integer> currentTuple = new TreeMap<>();
 5     for (List<Integer> list : lists) {
 6         if (list.isEmpty()) return null;
 7         pointers.add(0);
 8         bestTuple.add(list.get(0));
 9         currentTuple.put(list.get(0), lists.indexOf(list));
10     }
11     int minDist = maxMax(lists) - minMin(lists);
12     while (true) {
13         int newMinDist = currentTuple.lastKey() - currentTuple.firstKey()
               ;
14         if (minDist > newMinDist) {
15             minDist = newMinDist;
16             bestTuple = new ArrayList<>(currentTuple.keySet());
17         }
18         int listIndex = currentTuple.get(currentTuple.firstKey());
19         int pointer = pointers.get(listIndex) + 1;
20         if (pointer >= lists.get(listIndex).size())
21             break;
22         int value = lists.get(listIndex).get(pointer);
23         pointers.set(listIndex, pointer);
24         currentTuple.remove(currentTuple.firstKey());
25         currentTuple.put(value, listIndex);
26     }
27     return bestTuple;
28 }
```

(c) Let m be the number of input lists and n be the sum of the lengths of the input lists.

The initalization step consists of a for-loop which loops m times. In each iteration of the for-loop, we have constant-time list-adding operations and one SortedMap insertion.

maxMax and minMin are simply scans and run in $\Theta(n)$ time.

The while-loop runs in at most n times, because each iteration results in one pointer being incremented. Within each iteration, there is at most one tuple copy operation (line 16), a constant number of calls to the SortedMap's firstKey and lastKey operations (the Java names for min and max keys, lines 13, 18, 24), and at most one call to the SortedMap's remove and put operations. All other operations within the loop are constant time. In total, we have a running time of

$$\text{O(m Time(put)} + \text{n} \times \text{(Time(tuple copy)} + \text{Time(firstKey)} + \text{Time(lastKey)} + \text{Time(put)} + \text{Time(remove)))}$$

(d) In the following analysis, we use the fact that the SortedMap stores m items (not n) at all times, hence has methods with running time in terms of m. The tuple copying will take time $\Theta(m)$ since each element of the tuple is copied over.

- If the SortedMap is implemented with a sorted linked-list, put and remove are linear time operations and firstKey and lastKey are constant-time operations. Therefore, we have a running time of $O(m^2 + nm)$ in the worst case.

- If the SortedMap is implemented with a non-balanced binary search tree, all SortedMap operations are linear time in the worst case. Therefore, we have a running time of $O(m^2 + nm)$ in the worst case.
- If the SortedMap is implemented with a balanced binary search tree, all operations are logarithmic time in the worst case. Therefore, we have a running time of $O(m \log m + n(m + \log m)) = O(m \log m + nm)$ in the worst case.
- If the SortedMap is implemented with a skiplist, put and remove are logarithmic-time operations on expectation, and firstKey and lastKey are constant-time operations. Therefore, we have a running time of $O(m \log m + n(m + \log m)) = O(m \log m + nm)$ on expectation.

2. (a)
```
Node min() { return left == null ? this : left.min(); }
```

(b)
```
// return the node which contains e, if it exists, or the node whose
    child would contain e otherwise.
Node find(E e) {
        if (this.value.equals(e))
                return this;
        if (this.value.compareTo(e) > 0)
                return left == null ? this : left.find(e);
        return right == null ? this : right.find(e);
}
```

(c)
```
Node successor() {
    if (right != null)
        return right.min();
    Node ancestor = this;
    while (ancestor.isRightChild())
        ancestor = ancestor.parent;
    return ancestor.parent;
}
```

(d)
```
void remove() {
    if (left == null || right == null) {
        linkParentAndChild();
    } else {
        Node successor = successor();
        value = successor.value;
        successor.remove();
    }
}

void linkParentAndChild() {
    Node child = left == null ? right : left;
    if (isLeftChild()) {
        parent.left = child;
        if (child != null)
            child.parent = parent;
    } else if (isRightChild()) {
        parent.right = child;
        if (child != null)
            child.parent = parent;
    } else {
        root = child;
        if (child != null)
            child.parent = null;
    }
}
```

3.
```
    // Return a node such that node.xvalue < value <= node.right.xvalue &&
       node.down == null
    private Node find(E value) {
        Node node = head;
        while (node.right.xvalue.compareTo(value) < 0)
            node = node.right;
        while (node.down != null) {
            node = node.down;
            while (node.right.xvalue.compareTo(value) < 0)
                node = node.right;
        }
        return node;
    }
```

4. (a) There are $\log_2 n + 1$ levels on expectation: there are n nodes at level 0, and for every other level on average half as many as the level below it.

   (b) Consider the sequence: 5, 8, 1, 7, 10. We will insert in this particular order into an empty skiplist 10 times. Here are the outcomes:

```
|-oo|----------->|7|------------>|+oo|
|-oo|----------->|7|------------>|+oo|
|-oo|->|1|->|5|->|7|------------>|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|---------------->|8|------->|+oo|
|-oo|---------------->|8|->|10|->|+oo|
|-oo|---------------->|8|->|10|->|+oo|
|-oo|->|1|->|5|------>|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|---------------->|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|---------------->|8|------->|+oo|
|-oo|---------------->|8|------->|+oo|
|-oo|----------->|7|->|8|------->|+oo|
|-oo|----------->|7|->|8|->|10|->|+oo|
|-oo|------>|5|->|7|->|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|-------------------->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|------>|5|---------------->|+oo|
|-oo|->|1|->|5|->|7|----------->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|->|1|--------------------->|+oo|
|-oo|->|1|->|5|---------------->|+oo|
|-oo|->|1|->|5|---------------->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|-------------------->|10|->|+oo|
|-oo|-------------------->|10|->|+oo|
|-oo|---------->|7|->|8|->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|

|-oo|------>|5|---------------->|+oo|
|-oo|->|1|->|5|---------------->|+oo|
|-oo|->|1|->|5|---------->|10|->|+oo|
|-oo|->|1|->|5|->|7|->|8|->|10|->|+oo|
```

The number of levels is: 4, 6, 2, 6, 1, 2, 3, 4, 4, 4, which averages to 3.6, which is about one level off from the expected $\log_2(5) + 1 \simeq 3.5$, which is pretty close

(c) Discussed in the next part.

(d) In this experiment, we generate four lists of random numbers of lengths 100, 1000, 10,000 and 100,000 numbers respectively. For each list, the experiment consists of inserting the list into an initially empty skiplist and measuring the number of levels generated afterwards. Each experiment is reapeated 100 times. This is the generated output:

```
number of items inserted: 100
number of times experiment is repeated: 100
predicted: 7.643856189774725
average: 8.25
standard dev: 2.062743396226107
%experiments where #levels was within 5 levels of predicted: 96.0%
%experiments where #levels was within 3 levels of predicted: 90.0%
%experiments where #levels was within 2 levels of predicted: 78.0%
%experiments where #levels was within 1 level of predicted: 44.0%

number of items inserted: 1000
number of times experiment is repeated: 100
predicted: 10.965784284662087
average: 11.29
standard dev: 1.9674897280728239
%experiments where #levels was within 5 levels of predicted: 96.0%
%experiments where #levels was within 3 levels of predicted: 89.0%
%experiments where #levels was within 2 levels of predicted: 75.0%
%experiments where #levels was within 1 level of predicted: 49.0%

number of items inserted: 10000
number of times experiment is repeated: 100
predicted: 14.28771237954945
average: 14.54
standard dev: 1.8899865193785386
%experiments where #levels was within 5 levels of predicted: 98.0%
%experiments where #levels was within 3 levels of predicted: 92.0%
%experiments where #levels was within 2 levels of predicted: 84.0%
%experiments where #levels was within 1 level of predicted: 44.0%

number of items inserted: 100000
number of times experiment is repeated: 100
predicted: 17.609640474436812
average: 18.11
standard dev: 1.9204842240491908
%experiments where #levels was within 5 levels of predicted: 97.0%
%experiments where #levels was within 3 levels of predicted: 88.0%
%experiments where #levels was within 2 levels of predicted: 82.0%
%experiments where #levels was within 1 level of predicted: 48.0%
```

We can see that, as the size of the skiplist grows, the average number of levels converges towards the predicted value, but also, the frequency with which the actual number of levels is within a certain bound of the predicted number of levels increases. In other words, the number of levels in a skiplist becomes increasingly closer to the predicted value with increasing probability, with increasing skiplist size.