Case digit > 2

Now, let's look at the case where dth digit of `x` is greater than 2 (`x[d] > 2`). We can apply almost the exact same logic to see that there are the same number of 2s in the 3rd digit in the range `0 - 63525` as there as in the range `0 - 70000`. So, rather than rounding down, we round up.

```
if x[d] > 2: count2sInRangeAtDigit(x, d) =
    let y = round up to nearest $10^{d+1}$
    return y / 10
```

Case digit = 2

The final case may be the trickiest, but it follows from the earlier logic. Consider `x = 62523` and `d = 3`. We know that there are the same ranges of 2s from before (that is, the ranges `2000 - 2999`, `12000 - 12999`, ..., `52000 - 52999`). How many appear in the 3rd digit in the final, partial range from `62000 - 62523`? Well, that should be pretty easy. It's just 524 (`62000, 62001, ..., 62523`).

```
if x[d] = 2: count2sInRangeAtDigit(x, d) =
    let y = round down to nearest $10^{d+1}$
    let z = right side of x (i.e., x % $10^d$)
    return y / 10 + z + 1
```

Now, all you need is to iterate through each digit in the number. Implementing this code is reasonably straightforward.

```
int count2sInRangeAtDigit(int number, int d) {
  int powerOf10 = (int) Math.pow(10, d);
  int nextPowerOf10 = powerOf10 * 10;
  int right = number % powerOf10;

  int roundDown = number - number % nextPowerOf10;
  int roundUp = roundDown + nextPowerOf10;

  int digit = (number / powerOf10) % 10;
  if (digit < 2) { // if the digit in spot digit is
    return roundDown / 10;
  } else if (digit == 2) {
    return roundDown / 10 + right + 1;
  } else {
    return roundUp / 10;
  }
}

int count2sInRange(int number) {
  int count = 0;
  int len = String.valueOf(number).length();
  for (int digit = 0; digit < len; digit++) {
    count += count2sInRangeAtDigit(number, digit);
  }
  return count;
}
```

This question requires very careful testing. Make sure to generate a list of test cases, and to work through each of them.

17.7 **Baby Names:** Each year, the government releases a list of the 10,000 most common baby names and their frequencies (the number of babies with that name). The only problem with this is that some names have multiple spellings. For example, "John" and "Jon" are essentially the same name but would be listed separately in the list. Given two lists, one of names/frequencies and the other of pairs of *equivalent* names, write an algorithm to print a new list of the true frequency of each name. Note that if John and Jon are synonyms, and Jon and Johnny are synonyms, then John and Johnny are synonyms. (It is both transitive and symmetric.) In the final list, any name can be used as the "real" name.

EXAMPLE

Input:

Names: John (15), Jon (12), Chris (13), Kris (4), Christopher (19)

Synonyms: (Jon, John), (John, Johnny), (Chris, Kris), (Chris, Christopher)

Output: John (27), Kris (36)

pg 187

SOLUTION

Let's start off with a good example. We want an example with some names with multiple synonyms and some with none. Additionally, we want the synonym list to be diverse in which name is on the left side and which is on the right. For example, we wouldn't want Johnny to always be the name on the left side as we're creating the group of (John, Jonathan, Jon, and Johnny).

This list should work fairly well.

Name	Count
John	10
Jon	3
Davis	2
Kari	3
Johnny	11
Carlton	8
Carleton	2
Jonathan	9
Carrie	5

Name	Alternate
Jonathan	John
Jon	Johnny
Johnny	John
Kari	Carrie
Carleton	Carlton

The final list should be something like: John (33), Kari (8), Davis(2), Carleton (10).

Solution #1

Let's assume our baby names list is given to us as a hash table. (If not, it's easy enough to build one.)

We can start reading pairs in from the synonyms list. As we read the pair (Jonathan, John), we can merge the counts for Jonathan and John together. We'll need to remember, though, that we saw this pair, because, in the future, we could discover that Jonathan is equivalent to something else.

We can use a hash table (`L1`) that maps from a name to its "true" name. We'll also need to know, given a "true" name, all the names equivalent to it. This will be stored in a hash table `L2`. Note that `L2` acts as a reverse lookup of `L1`.

```
READ (Jonathan, John)
```

```
        L1.ADD Jonathan -> John
        L2.ADD John -> Jonathan
    READ (Jon, Johnny)
        L1.ADD Jon -> Johnny
        L2.ADD Johnny -> Jon
    READ (Johnny, John)
        L1.ADD Johnny -> John
        L1.UPDATE Jon -> John
        L2.UPDATE John -> Jonathan, Johnny, Jon
```

If we later find that John is equivalent to, say, Jonny, we'll need to look up the names in L1 and L2 and merge together all the names that are equivalent to them.

This will work, but it's unnecessarily complicated to keep track of these two lists.

Instead, we can think of these names as "equivalence classes." When we find a pair (Jonathan, John), we put these in the same set (or equivalence classes). Each name maps to its equivalence class. All items in the set map to the same instance of the set.

If we need to merge two sets, then we copy one set into the other and update the hash table to point to the new set.

```
    READ (Jonathan, John)
        CREATE Set1 = Jonathan, John
        L1.ADD Jonathan -> Set1
        L1.ADD John -> Set1
    READ (Jon, Johnny)
        CREATE Set2 = Jon, Johnny
        L1.ADD Jon -> Set2
        L1.ADD Johnny -> Set2
    READ (Johnny, John)
        COPY Set2 into Set1.
            Set1 = Jonathan, John, Jon, Johnny
        L1.UPDATE Jon -> Set1
        L1.UPDATE Johnny -> Set1
```

In the last step above, we iterated through all items in Set2 and updated the reference to point to Set1. As we do this, we keep track of the total frequency of names.

```
HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
                                          String[][] synonyms) {
  /* Parse list and initialize equivalence classes.*/
  HashMap<String, NameSet> groups = constructGroups(names);

  /* Merge equivalence classes together. */
  mergeClasses(groups, synonyms);

  /* Convert back to hash map. */
  return convertToMap(groups);
}

/* This is the core of the algorithm. Read through each pair. Merge their
 * equivalence classes and update the mapping of the secondary class to point to
 * the first set.*/
void mergeClasses(HashMap<String, NameSet> groups, String[][] synonyms) {
  for (String[] entry : synonyms) {
    String name1 = entry[0];
    String name2 = entry[1];
    NameSet set1 = groups.get(name1);
```

```
      NameSet set2 = groups.get(name2);
      if (set1 != set2) {
        /* Always merge the smaller set into the bigger one. */
        NameSet smaller = set2.size() < set1.size() ? set2 : set1;
        NameSet bigger = set2.size() < set1.size() ? set1 : set2;

        /* Merge lists */
        Set<String> otherNames = smaller.getNames();
        int frequency = smaller.getFrequency();
        bigger.copyNamesWithFrequency(otherNames, frequency);

        /* Update mapping */
        for (String name : otherNames) {
          groups.put(name,  bigger);
        }
      }
    }
  }

  /* Read through (name, frequency) pairs and initialize a mapping of names to
   * NameSets (equivalence classes).*/
  HashMap<String, NameSet> constructGroups(HashMap<String, Integer> names) {
    HashMap<String, NameSet> groups = new HashMap<String, NameSet>();
    for (Entry<String, Integer> entry : names.entrySet()) {
      String name = entry.getKey();
      int frequency = entry.getValue();
      NameSet group = new NameSet(name, frequency);
      groups.put(name,  group);
    }
    return groups;
  }

  HashMap<String, Integer> convertToMap(HashMap<String, NameSet> groups) {
    HashMap<String, Integer> list = new HashMap<String, Integer>();
    for (NameSet group : groups.values()) {
      list.put(group.getRootName(), group.getFrequency());
    }
    return list;
  }

  public class NameSet {
    private Set<String> names = new HashSet<String>();
    private int frequency = 0;
    private String rootName;

    public NameSet(String name, int freq) {
      names.add(name);
      frequency = freq;
      rootName = name;
    }

    public void copyNamesWithFrequency(Set<String> more, int freq) {
      names.addAll(more);
      frequency += freq;
    }

    public Set<String> getNames() { return names; }
    public String getRootName() { return rootName; }
    public int getFrequency() { return frequency; }
    public int size() { return names.size(); }
}
```

The runtime of the algorithm is a bit tricky to figure out. One way to think about it is to think about what the worst case is.

For this algorithm, the worst case is where all names are equivalent—and we have to constantly merge sets together. Also, for the worst case, the merging should come in the worst possible way: repeated pairwise merging of sets. Each merging requires copying the set's elements into an existing set and updating the pointers from those items. It's slowest when the sets are larger.

If you notice the parallel with merge sort (where you have to merge single-element arrays into two-element arrays, and then two-element arrays into four-element arrays, until finally having a full array), you might guess it's O(N log N). That is correct.

If you don't notice that parallel, here's another way to think about it.

Imagine we had the names (a, b, c, d, . . ., z). In our worst case, we'd first pair up the items into equivalence classes: (a, b), (c, d), (e, f), . . ., (y, z). Then, we'd merge pairs of those: (a, b, c, d), (e, f, g, h), . . ., (w, x, y, z). We'd continue doing this until we wind up with just one class.

At each "sweep" through the list where we merge sets together, half of the items get moved into a new set. This takes O(N) work per sweep. (There are fewer sets to merge, but each set has grown larger.)

How many sweeps do we do? At each sweep, we have half as many sets as we did before. Therefore, we do O(log N) sweeps.

Since we're doing O(log N) sweeps and O(N) work per sweep, the total runtime is O(N log N).

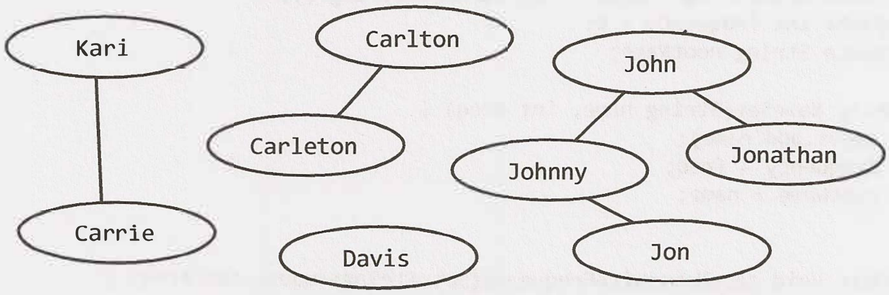
This is pretty good, but let's see if we can make it even faster.

Optimized Solution

To optimize the old solution, we should think about what exactly makes it slow. Essentially, it's the merging and updating of pointers.

So what if we just didn't do that? What if we marked that there was an equivalence relationship between two names, but didn't actually do anything with the information yet?

In this case, we'd be building essentially a graph.



Now what? Visually, it seems easy enough. Each component is an equivalent set of names. We just need to group the names by their component, sum up their frequencies, and return a list with one arbitrarily chosen name from each group.

In practice, how does this work? We could pick a name and do a depth-first (or breadth-first) search to sum the frequencies of all the names in one component. We would have to make sure that we hit each component exactly once. That's easy enough to achieve: mark a node as `visited` after it's discovered in the graph search, and only start the search for nodes where `visited` is false.

```java
HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
                                          String[][] synonyms) {
  /* Create data. */
  Graph graph = constructGraph(names);
  connectEdges(graph, synonyms);

  /* Find components. */
  HashMap<String, Integer> rootNames = getTrueFrequencies(graph);
  return rootNames;
}

/* Add all names to graph as nodes. */
Graph constructGraph(HashMap<String, Integer> names) {
  Graph graph = new Graph();
  for (Entry<String, Integer> entry : names.entrySet()) {
      String name = entry.getKey();
      int frequency = entry.getValue();
      graph.createNode(name, frequency);
  }
  return graph;
}

/* Connect synonymous spellings. */
void connectEdges(Graph graph, String[][] synonyms) {
  for (String[] entry : synonyms) {
      String name1 = entry[0];
      String name2 = entry[1];
      graph.addEdge(name1,  name2);
  }
}

/* Do DFS of each component. If a node has been visited before, then its component
 * has already been computed. */
HashMap<String, Integer> getTrueFrequencies(Graph graph) {
  HashMap<String, Integer> rootNames = new HashMap<String, Integer>();
  for (GraphNode node : graph.getNodes()) {
    if (!node.isVisited()) { // Already visited this component
      int frequency = getComponentFrequency(node);
      String name = node.getName();
      rootNames.put(name, frequency);
    }
  }
  return rootNames;
}

/* Do depth-first search to find the total frequency of this component, and mark
 * each node as visited.*/
int getComponentFrequency(GraphNode node) {
  if (node.isVisited()) return 0; // Already visited

  node.setIsVisited(true);
  int sum = node.getFrequency();
  for (GraphNode child : node.getNeighbors()) {
    sum += getComponentFrequency(child);
  }
  return sum;
}

/* Code for GraphNode and Graph is fairly self-explanatory, but can be found in
 * the downloadable code solutions.*/
```

To analyze the efficiency, we can think about the efficiency of each part of the algorithm.

- Reading in the data is linear with respect to the size of the data, so it takes `O(B + P)` time, where B is the number of baby names and P is the number of pairs of synonyms. This is because we only do a constant amount of work per piece of input data.
- To compute the frequencies, each edge gets "touched" exactly once across all of the graph searches and each node gets touched exactly once to check if it's been visited. The time of this part is `O(B + P)`.

Therefore, the total time of the algorithm is `O(B + P)`. We know we cannot do better than this since we must at least read in the `B + P` pieces of data.

17.8 **Circus Tower:** A circus is designing a tower routine consisting of people standing atop one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

pg 187

SOLUTION

When we cut out all the "fluff" to this problem, we can understand that the problem is really the following.

We have a list of pairs of items. Find the longest sequence such that both the first and second items are in non-decreasing order.

One thing we might first try is sorting the items on an attribute. This is useful actually, but it won't get us all the way there.

By sorting the items by height, we have a relative order the items must appear in. We still need to find the longest increasing subsequence of weight though.

Solution 1: Recursive

One approach is to essentially try all possibilities. After sorting by height, we iterate through the array. At each element, we branch into two choices: add this element to the subsequence (if it's valid) or do not.

```
ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> items) {
  Collections.sort(items);
  return bestSeqAtIndex(items, new ArrayList<HtWt>(), 0);
}

ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array, ArrayList<HtWt> sequence,
                               int index) {
  if (index >= array.size()) return sequence;

  HtWt value = array.get(index);

  ArrayList<HtWt> bestWith = null;
  if (canAppend(sequence, value)) {
    ArrayList<HtWt> sequenceWith = (ArrayList<HtWt>) sequence.clone();
    sequenceWith.add(value);
    bestWith = bestSeqAtIndex(array, sequenceWith, index + 1);
  }

  ArrayList<HtWt> bestWithout = bestSeqAtIndex(array, sequence, index + 1);

  if (bestWith == null || bestWithout.size() > bestWith.size()) {
    return bestWithout;
  } else {
    return bestWith;
  }
}

boolean canAppend(ArrayList<HtWt> solution, HtWt value) {
  if (solution == null) return false;
  if (solution.size() == 0) return true;

  HtWt last = solution.get(solution.size() - 1);
  return last.isBefore(value);
}

ArrayList<HtWt> max(ArrayList<HtWt> seq1, ArrayList<HtWt> seq2) {
  if (seq1 == null) {
    return seq2;
  } else if (seq2 == null) {
    return seq1;
  }
  return seq1.size() > seq2.size() ? seq1 : seq2;
}

public class HtWt implements Comparable<HtWt> {
  private int height;
  private int weight;
  public HtWt(int h, int w) { height = h; weight = w; }

  public int compareTo(HtWt second) {
    if (this.height != second.height) {
      return ((Integer)this.height).compareTo(second.height);
    } else {
      return ((Integer)this.weight).compareTo(second.weight);
    }
  }

  /* Returns true if "this" should be lined up before "other". Note that it's
   * possible that this.isBefore(other) and other.isBefore(this) are both false.
   * This is different from the compareTo method, where if a < b then b > a. */
  public boolean isBefore(HtWt other) {
    if (height < other.height && weight < other.weight) {
      return true;
    } else {
      return false;
    }
  }
```

```
}
```

This algorithm will take O(2^n) time. We can optimize it using memoization (that is, caching the best *sequences).*

There's a cleaner way to do this though.

Solution #2: Iterative

Imagine we had the longest subsequence that terminates with each element, A[0] through A[3]. Could we use this to find the longest subsequence that terminates with A[4]?

```
Array: 13, 14, 10, 11, 12
Longest(ending with A[0]): 13
Longest(ending with A[1]): 13, 14
Longest(ending with A[2]): 10
Longest(ending with A[3]): 10, 11
Longest(ending with A[4]): 10, 11, 12
```

Sure. We just append A[4] on to the longest subsequence that it can be appended to.

This is now fairly straightforward to implement.

```
ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> array) {
  Collections.sort(array);

  ArrayList<ArrayList<HtWt>> solutions =  new ArrayList<ArrayList<HtWt>>();
  ArrayList<HtWt> bestSequence = null;

  /* Find the longest subsequence that terminates with each element. Track the
   * longest overall subsequence as we go. */
  for (int i = 0; i < array.size(); i++) {
    ArrayList<HtWt> longestAtIndex = bestSeqAtIndex(array, solutions, i);
    solutions.add(i, longestAtIndex);
    bestSequence = max(bestSequence, longestAtIndex);
  }

  return bestSequence;
}

/* Find the longest subsequence which terminates with this element. */
ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array,
    ArrayList<ArrayList<HtWt>> solutions, int index) {
  HtWt value = array.get(index);

  ArrayList<HtWt> bestSequence = new ArrayList<HtWt>();

  /* Find the longest subsequence that we can append this element to. */
  for (int i = 0; i < index; i++) {
    ArrayList<HtWt> solution = solutions.get(i);
    if (canAppend(solution, value)) {
      bestSequence = max(solution, bestSequence);
    }
  }

  /* Append element. */
  ArrayList<HtWt> best = (ArrayList<HtWt>) bestSequence.clone();
  best.add(value);

```

```
    return best;
}
```

This algorithm operates in O(n²) time. An O(n log(n)) algorithm does exist, but it is considerably more complicated and it is highly unlikely that you would derive this in an interview—even with some help. However, if you are interested in exploring this solution, a quick internet search will turn up a number of explanations of this solution.

17.9 Kth Multiple: Design an algorithm to find the kth number such that the only prime factors are 3, 5, and 7. Note that 3, 5, and 7 do not have to be factors, but it should not have any other prime factors. For example, the first several multiples would be (in order) 1, 3, 5, 7, 9, 15, 21.

pg 187

SOLUTION

Let's first understand what this problem is asking for. It's asking for the kth smallest number that is in the form $3^a * 5^b * 7^c$. Let's start with a brute force way of finding this.

Brute Force

We know that biggest this kth number could be is $3^k * 5^k * 7^k$. So, the "stupid" way of doing this is to compute $3^a * 5^b * 7^c$ for all values of a, b, and c between 0 and k. We can throw them all into a list, sort the list, and then pick the kth smallest value.

```
int getKthMagicNumber(int k) {
  ArrayList<Integer> possibilities = allPossibleKFactors(k);
  Collections.sort(possibilities);
  return possibilities.get(k);
}

ArrayList<Integer> allPossibleKFactors(int k) {
  ArrayList<Integer> values = new ArrayList<Integer>();
  for (int a = 0; a <= k; a++) { // loop 3
    int powA = (int) Math.pow(3, a);
    for (int b = 0; b <= k; b++) { // loop 5
      int powB = (int) Math.pow(5, b);
      for (int c = 0; c <= k; c++) { // loop 7
        int powC = (int) Math.pow(7, c);
        int value = powA * powB * powC;

        /* Check for overflow. */
        if (value < 0 || powA == Integer.MAX_VALUE ||
            powB == Integer.MAX_VALUE ||
            powC == Integer.MAX_VALUE) {
          value = Integer.MAX_VALUE;
        }
        values.add(value);
      }
    }
  }
  return values;
}
```