**Lecture 14: Finish up Maps, recursive backtracking**

- We'll finish up the Friends program from Monday
- run Jessica/Melissa
- Another problem: Ashley appears at distance 1, 3, 4...
  - Why is this?
    - She's friends with a friend of a friend
  - But we don't want this - we just want the first time we see the friend
  - How can we do this?
    - Naive: ignore the previous group's names in the next group

```
nextGroup.removeAll(currentGroup);
```

    - But this doesn't actually work because names like Ashley might not appear for a level, but then reappear b/c friend-of-a-friend
    - Keep track of another set of people who have been seen before

```
Set<String> alreadySeen = new TreeSet<String>();
...
while (!currentGroup.contains(name2)) {
    distance++;
    alreadySeen.addAll(currentGroup);
    Set<String> nextGroup = new TreeSet<String>();
    for (String friend : currentGroup) {
        nextGroup.addAll(friends.get(friend));
    }
    nextGroup.removeAll(alreadySeen);
    ...
```

- Another problem: run Melissa/Bart
  - We never stop!
  - Solution:

```
while (!currentGroup.contains(name2) && !currentGroup.isEmpty()) {
        ...
}
if (currentGroup.contains(name2)) {
    System.out.println("found at a distance of "+distance);
} else {
    System.out.println("not found");
}
```

- The goal of this program
  - Review of sets/maps
  - Demonstration of mapping with complicated values

- Now we're switching back to a new topic: a particular application of recursion called *recursive backtracking*
    - An approach to solving some types of problems
    - An example: write a method to print out all possible ways that you could roll *n* dice.
    - So, for example, diceRoll(2) would give:

      [1, 1] [2, 1] [3, 1]
      [1, 2] [2, 2] [3, 2]
      [1, 3]... [2, 3]... [3, 3]...
    - When you were writing out the possibilities, how did you figure out what to do?
        - Set the first dice to 1, then considered all possible rolls of dice 2
        - Set the first dice to 2, then considered all possible rolls of dice 2
        - Set the first dice to 3, then considered all possible rolls of dice 2
        - ...
    - You can make a table to describe our process

```
     1st   2nd   3rd...
1     y     -     y
2     -     y     -
3     -     -     -...
```

    - The idea here is that we make one choice (e.g. the first die's value), then we EXPLORE all possibilities that include that choice, then UNCHOOSE that choice and make a different one
        - This process - choose, explore, unchoose - is common to recursive backtracking problems
    - We could do this with nested loops
        - for (int i = 1; i < 7; i++)
            - for (int j = 1; j < 7; j++) ...
    - But we don't know how many loops we'll need (don't know how many dice)
    - So, not surprisingly, we'll use recursion
    - (show the solution, explain)
    - You can also visualize the process as a decision tree

| | | 3 dice | | | |
|---|---|---|---|---|---|
| 1, 2 dice | 2, 2 dice | 3, 2 dice | 4, 2 dice | 5, 2 dice | 6, 2 dice |
| 1,1, 1 die | 1,2, 1 die | 1, 3, 1 die | 1, 4, 1 die | 1,5, 1 die | 1,6, 1 die |

...
- Backtracking is a BRUTE-FORCE search algorithm that explores all possible options in some SEARCH SPACE
    - Questions to ask
        - What is a "choice"? How do I know when I'm out of choices?
        - How do I "make" a choice?
        - How do I explore the remaining choices?
        - Once I'm done exploring what do I do? Print, return?
        - How do I "unmake" a choice?
- I want to modify my dice-roll solution to instead print out only rolls that have a sum equal to some

given value

- For example diceSum(2, 11) will print out [5,6] and [6,5]
- So what do we want to say (in English)
  - In the base case, print only if the sum is equal to the target sum
  - So our private helper needs more information - the desired sum and the current sum (current sum isn't strictly necessary) as parameters
  - when recursing, pass desired sum and sum so far
  - Can also prune sums that are too big, or that cannot possibly make it all the way
- Solution:

```
public static void diceSum(int dice, int desiredSum) {
    List<Integer> chosen = new ArrayList<Integer>();
    diceSum2(dice, desiredSum, chosen, 0);
}
private static void diceSum(int dice, int desiredSum,
    List<Integer> chosen, int sumSoFar) {
    if (dice == 0) {
        if (sumSoFar == desiredSum) {
            System.out.println(chosen);
        }
    } else if (sumSoFar <= desiredSum &&
               sumSoFar + 6 * dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceSum(dice - 1, desiredSum, chosen, sumSoFar + i);
            chosen.remove(chosen.size() - 1);
        }
    }
}
```