Exercise: Dice roll sum

Write a method `diceSum` similar to `diceRoll`, but it also accepts a desired sum and prints only arrangements that add up to exactly that sum.

```
diceSum(2, 7);
[1, 6]
[2, 5]
[3, 4]
[4, 3]
[5, 2]
[6, 1]

diceSum(3, 7);
[1, 1, 5]
[1, 2, 4]
[1, 3, 3]
[1, 4, 2]
[1, 5, 1]
[2, 1, 4]
[2, 2, 3]
[2, 3, 2]
[2, 4, 1]
[3, 1, 3]
[3, 2, 2]
[3, 3, 1]
[4, 1, 2]
[4, 2, 1]
[5, 1, 1]
```
Consider all paths?

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
<th>desired sum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 dice</td>
<td>5</td>
</tr>
</tbody>
</table>

- 1, 1, 1
  - 1, 1, 2
  - 1, 1, 3
  - 1, 1, 4
  - 1, 1, 5
  - 1, 1, 6
- 1, 2
  - 1, 2, 1
  - 1, 2, 2
  - 1, 2, 3
  - 1, 2, 4
  - 1, 2, 5
  - 1, 2, 6
- 1, 3
  - 1, 3, 1
  - 1, 3, 2
  - 1, 3, 3
  - 1, 3, 4
  - 1, 3, 5
  - 1, 3, 6
- 1, 4
  - 1, 4, 1
  - 1, 4, 2
  - 1, 4, 3
  - 1, 4, 4
  - 1, 4, 5
  - 1, 4, 6
- 1, 5
  - 1, 5, 1
  - 1, 5, 2
  - 1, 5, 3
  - 1, 5, 4
  - 1, 5, 5
  - 1, 5, 6
- 1, 6
  - 1, 6, 1
  - 1, 6, 2
  - 1, 6, 3
  - 1, 6, 4
  - 1, 6, 5
  - 1, 6, 6
Optimizations

• We need not visit every branch of the decision tree.
  • Some branches are clearly not going to lead to success.
  • We can preemptively stop, or prune, these branches.

• Inefficiencies in our dice sum algorithm:
  • Sometimes the current sum is already too high.
    • (Even rolling 1 for all remaining dice would exceed the sum.)
  
  • Sometimes the current sum is already too low.
    • (Even rolling 6 for all remaining dice would not reach the sum.)
  
  • When finished, the code must compute the sum every time.
    • (1+1+1 = ..., 1+1+2 = ..., 1+1+3 = ..., 1+1+4 = ..., ...
New decision tree

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
<th>desired sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>3 dice</td>
<td>5</td>
</tr>
</tbody>
</table>

```
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
```

```
1, 1, 1
1, 1, 2
1, 1, 3
1, 1, 4
1, 1, 5
1, 1, 6
```

```
1, 6, 1
1, 6, 2
```

...
The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.

- What are the "choices"?

- How do we "make" or "un-make" a choice?

- How do we know when to stop?
Naive algorithm

- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.

- How large is the solution space for this algorithm?
  - $64 \times 63 \times 62 \times \ldots$
Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.

  - Redefine a "choice" to be valid placement of a queen in a particular column.

  - How large is the solution space now?
    - $8 \times 8 \times 8 \times \ldots$
Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore($\text{choices}$):
  • if there are no more $\text{choices}$ to make: stop.
  • else, for each available choice $C$:
    • Choose $C$.
    • Explore the remaining $\text{choices}$.
    • Un-choose $C$, if necessary. (backtrack!)
Exercise

- Suppose we have a `Board` class with these methods:

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public <code>Board</code> (int size)</td>
<td>construct empty board</td>
</tr>
<tr>
<td>public boolean <code>isSafe</code> (int row, int column)</td>
<td>true if queen can be safely placed here</td>
</tr>
<tr>
<td>public void <code>place</code> (int row, int column)</td>
<td>place queen here</td>
</tr>
<tr>
<td>public void <code>remove</code> (int row, int column)</td>
<td>remove queen from here</td>
</tr>
<tr>
<td>public String <code>toString</code> ()</td>
<td>text display of board</td>
</tr>
</tbody>
</table>

- Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.