Exercise: fourAB

• Write a method `fourAB` that prints out all strings of length 4 composed only of a’s and b’s
• Example Output

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
<td>baaa</td>
</tr>
<tr>
<td>aaab</td>
<td>baab</td>
</tr>
<tr>
<td>aaba</td>
<td>baba</td>
</tr>
<tr>
<td>aabb</td>
<td>babb</td>
</tr>
<tr>
<td>abaa</td>
<td>bbba</td>
</tr>
<tr>
<td>abab</td>
<td>bbab</td>
</tr>
<tr>
<td>abba</td>
<td>bbba</td>
</tr>
<tr>
<td>abbb</td>
<td>bbbb</td>
</tr>
</tbody>
</table>
Decision Tree

- a
  - aa
    - aaa
      - aaaa
    - aaab
  - ab
    - aaba
    - aabb
Exercise: Dice rolls

- Write a method `diceRoll` that accepts an integer parameter representing a number of 6-sided dice to roll, and output all possible arrangements of values that could appear on the dice.

```
diceRoll(2);
```

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

```
diceRoll(3);
```

```
[1, 1, 1]
[1, 1, 2]
[1, 1, 3]
[1, 1, 4]
[1, 1, 5]
[1, 1, 6]
[1, 2, 1]
[1, 2, 2]
...  
[6, 6, 4]
[6, 6, 5]
[6, 6, 6]
```
Examining the problem

- We want to generate all possible sequences of values.
  
  for (each possible first die value):
  for (each possible second die value):
  for (each possible third die value):
    ...
    print!

- This is called a **depth-first search**

- How can we completely explore such a large search space?
A decision tree

<table>
<thead>
<tr>
<th>chosen</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>4 dice</td>
</tr>
</tbody>
</table>

1 dice
1, 1, 1 1 die
1, 1, 1, 1
1, 1, 1, 2
1, 1, 1, 3
1, 1, 1, 3
1, 1, 1, 4
1, 1, 1, 4

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

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

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

...
Backtracking

- **backtracking**: Finding solution(s) by trying partial solutions and then abandoning them if they are not suitable.
  - a "brute force" algorithmic technique (tries all paths)
  - often implemented recursively

Applications:
- producing all permutations of a set of values
- parsing languages
- games: anagrams, crosswords, word jumbles, 8 queens
- combinatorics and logic programming
Backtracking strategies

- When solving a backtracking problem, ask these questions:
  - What are the "choices" in this problem?
    - What is the "base case"? (How do I know when I'm out of choices?)
  - How do I "make" a choice?
    - Do I need to create additional variables to remember my choices?
    - Do I need to modify the values of existing variables?
  - How do I explore the rest of the choices?
    - Do I need to remove the made choice from the list of choices?
  - Once I'm done exploring, what should I do?
  - How do I "un-make" a choice?
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.

```java
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, 1, 1   1, 1, 2   1, 1, 3   1, 1, 4   1, 1, 5   1, 1, 6
1, 2, 1 die 1, 2, 1 die 1, 2, 1 die 1, 2, 1 die 1, 2, 1 die 1, 2, 1 die
1, 3, 1 die 1, 3, 1 die 1, 3, 1 die 1, 3, 1 die 1, 3, 1 die 1, 3, 1 die
1, 4, 1 die 1, 4, 1 die 1, 4, 1 die 1, 4, 1 die 1, 4, 1 die 1, 4, 1 die
1, 5, 1 die 1, 5, 1 die 1, 5, 1 die 1, 5, 1 die 1, 5, 1 die 1, 5, 1 die
1, 6, 1 die 1, 6, 1 die 1, 6, 1 die 1, 6, 1 die 1, 6, 1 die 1, 6, 1 die
...
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
...```