

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]
```

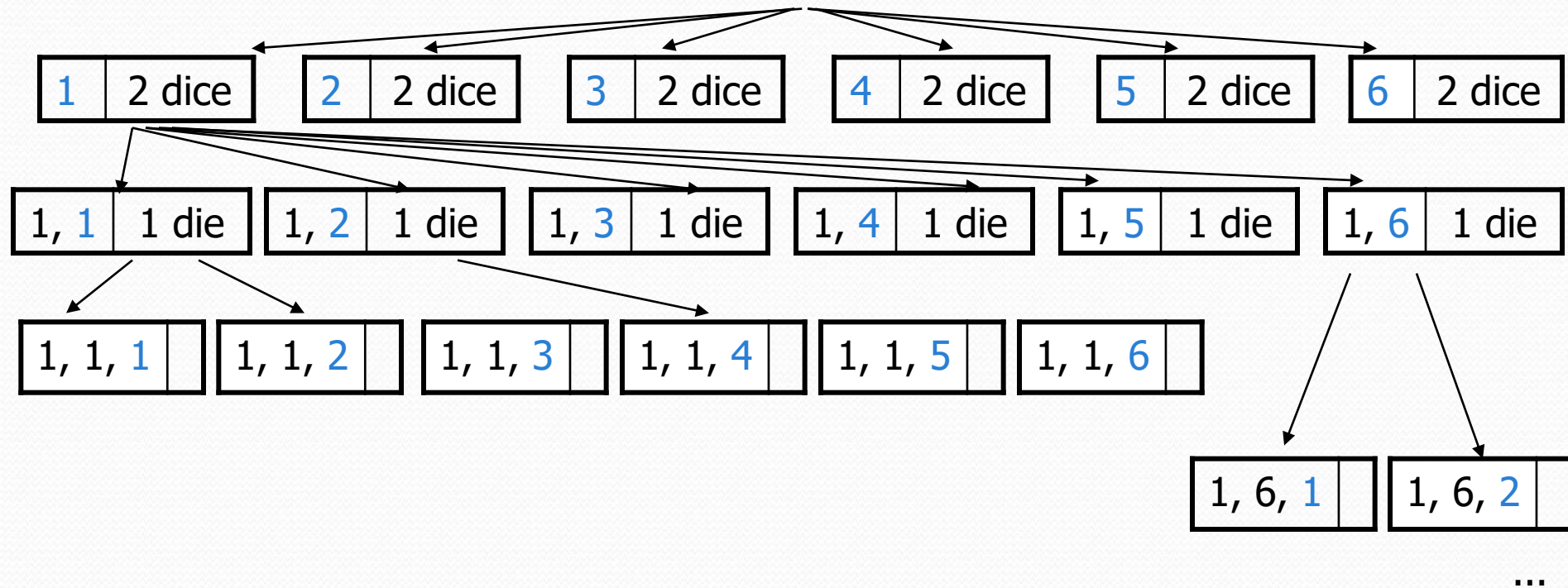


```
diceSum(3, 7);
```

```
[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?

chosen	available	desired sum
-	3 dice	5

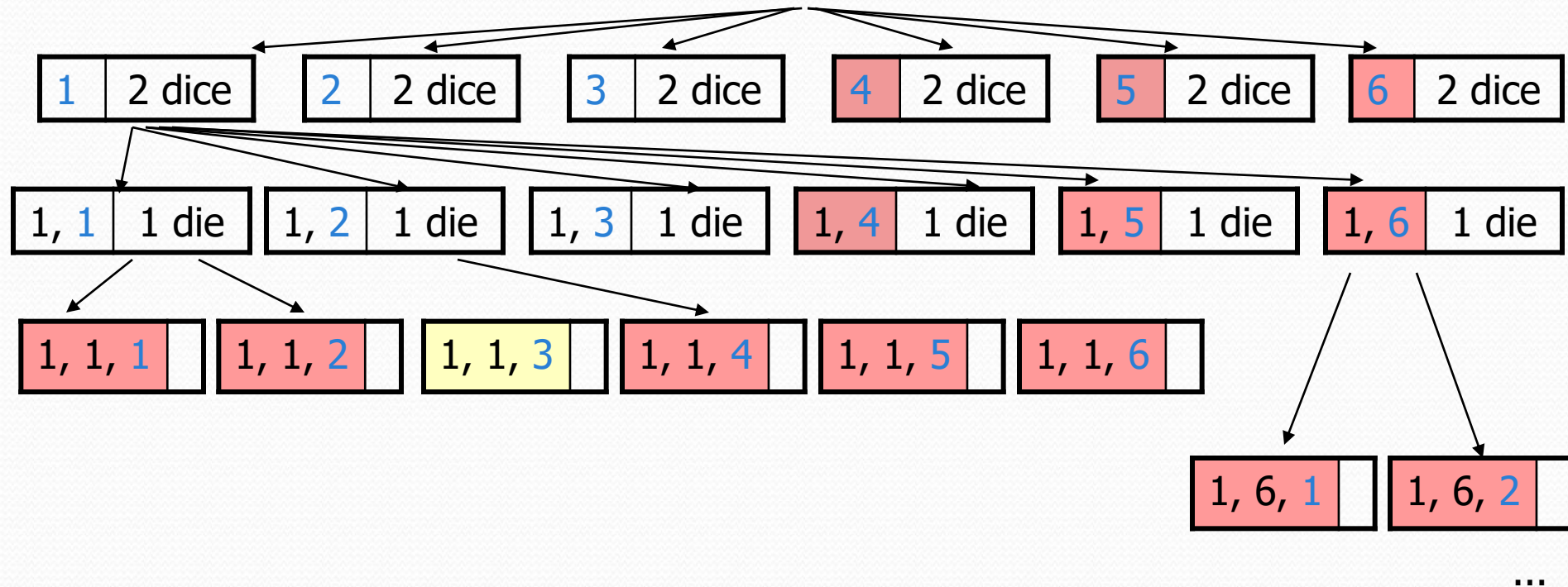


Optimizations

- 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 goal is too low.
 - (Even rolling 1 for all remaining dice would exceed the sum.)
 - Sometimes the current goal is too high.
 - (Even rolling 6 for all remaining dice would not reach the sum.)

New decision tree

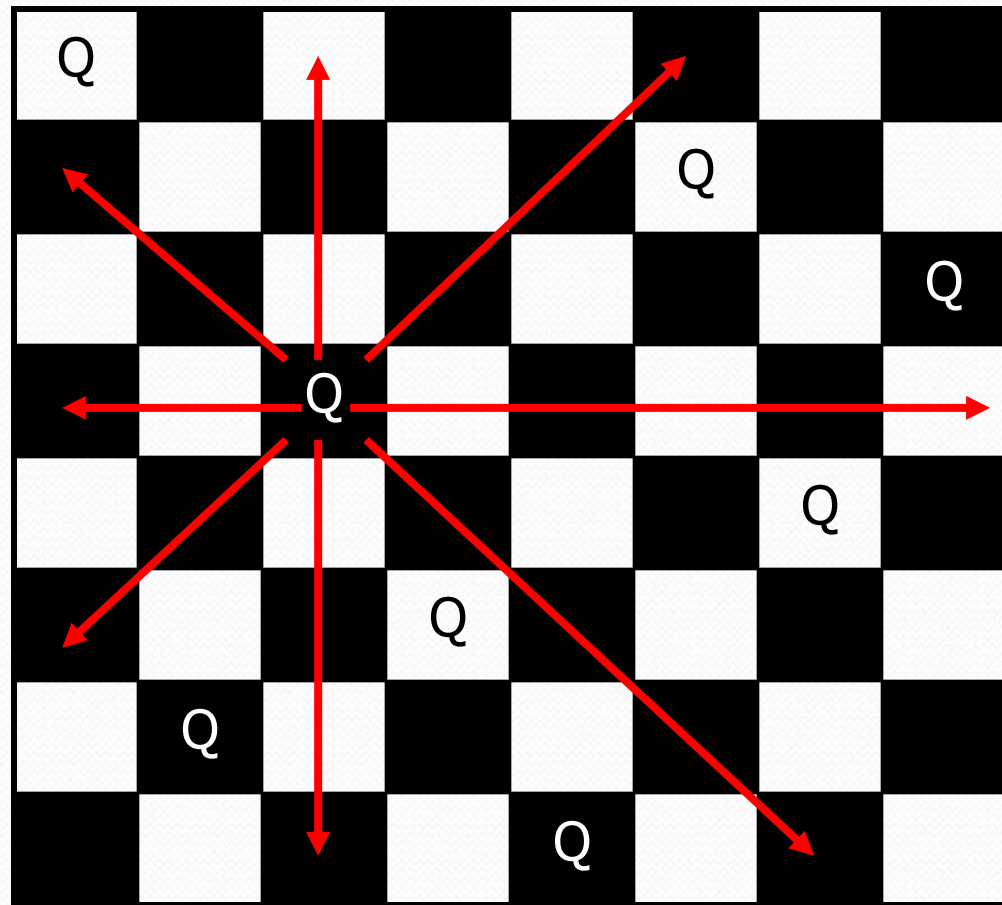
chosen	available	desired sum
-	3 dice	5



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?

- $64 * 63 * 62 * 61 * 60 * 59 * 58 * 57$

	1	2	3	4	5	6	7	8
1	Q
2
3	...							
4								
5								
6								
7								
8								

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?

- $8 * 8 * 8 * 8 * 8 * 8 * 8 * 8$

	1	2	3	4	5	6	7	8
1	Q					
2						
3		Q	...					
4			...					
5			Q					
6								
7								
8								

Exercise

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

Method/Constructor	Description
<code>public Board(int size)</code>	construct empty board
<code>public int size()</code>	returns the length/width of the board
<code>public boolean safe(int row, int column)</code>	true if queen can be safely placed here
<code>public void place(int row, int column)</code>	place queen here
<code>public void remove(int row, int column)</code>	remove queen from here
<code>public void print()</code>	displays the board

- Write a method `solve` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.
 - Your method should find all solutions.

Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(**choices**):

- if there are no more **choices** to make: stop.
- else, for each available choice **C**:
 - Choose **C**.
 - Explore the remaining **choices**.
 - Un-choose **C**, if necessary. (backtrack!)