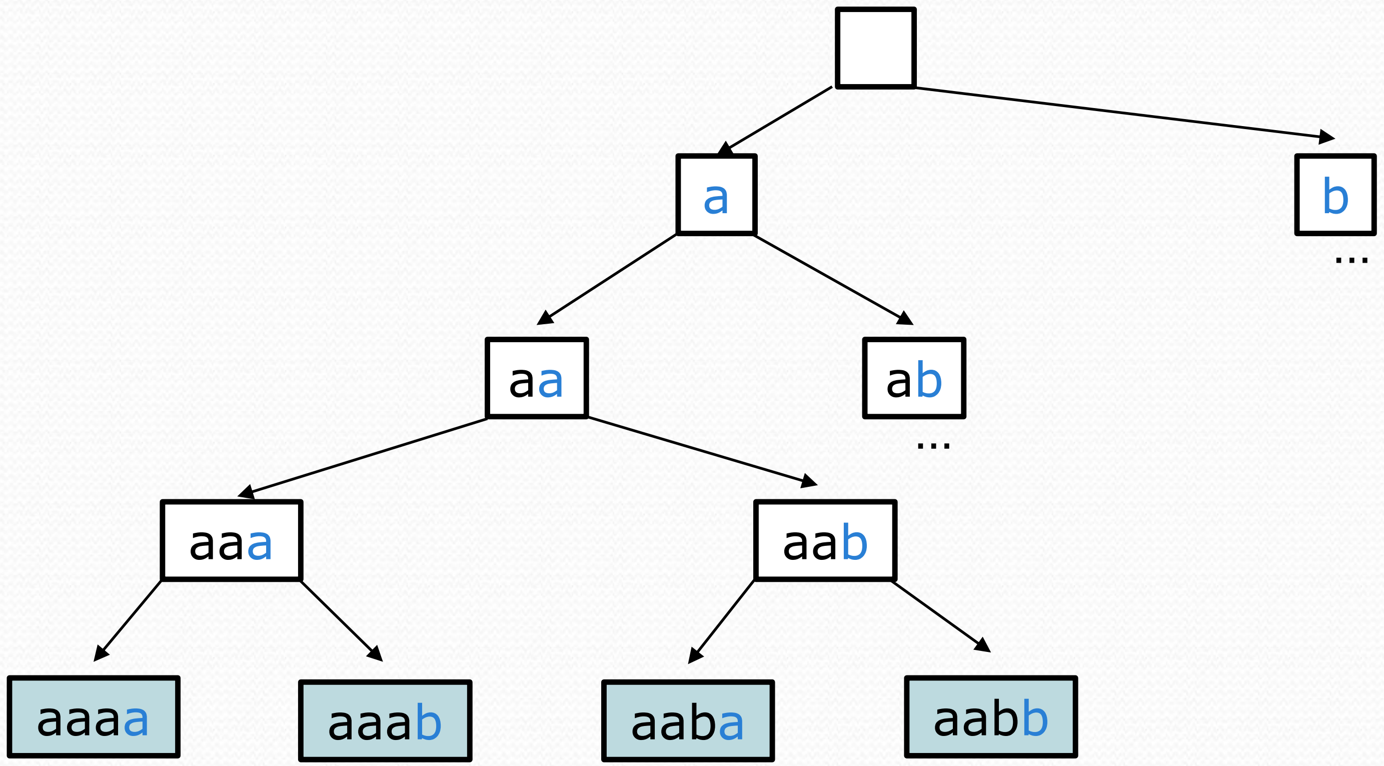
Exercise: fourAB

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

```
aaaa        baaa
aaab        baab
aaba        baba
aabb        babb
abaa        bbaa
abab        bbab
abba        bbba
abbb        bbbb
```

Decision Tree



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.

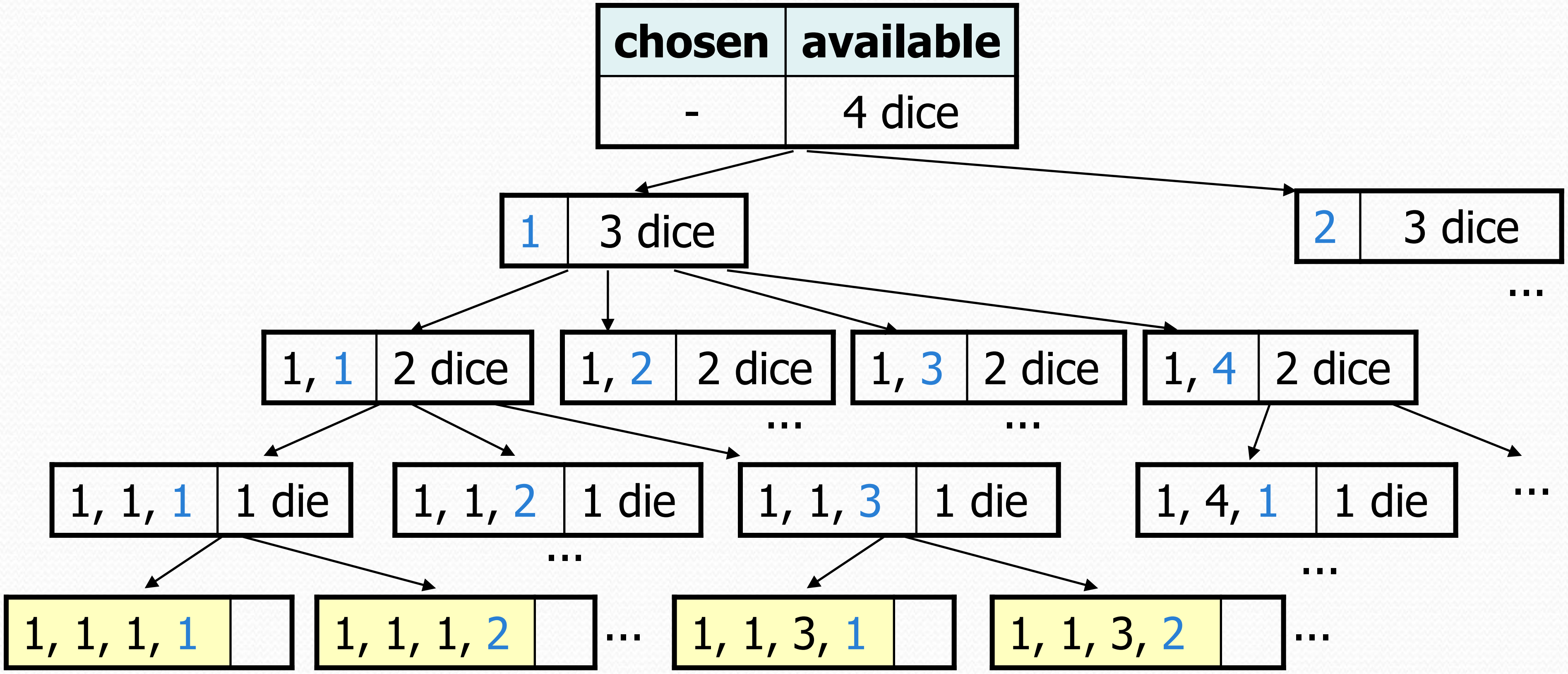
```
diceRoll(2);

[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]
```

A decision tree



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.

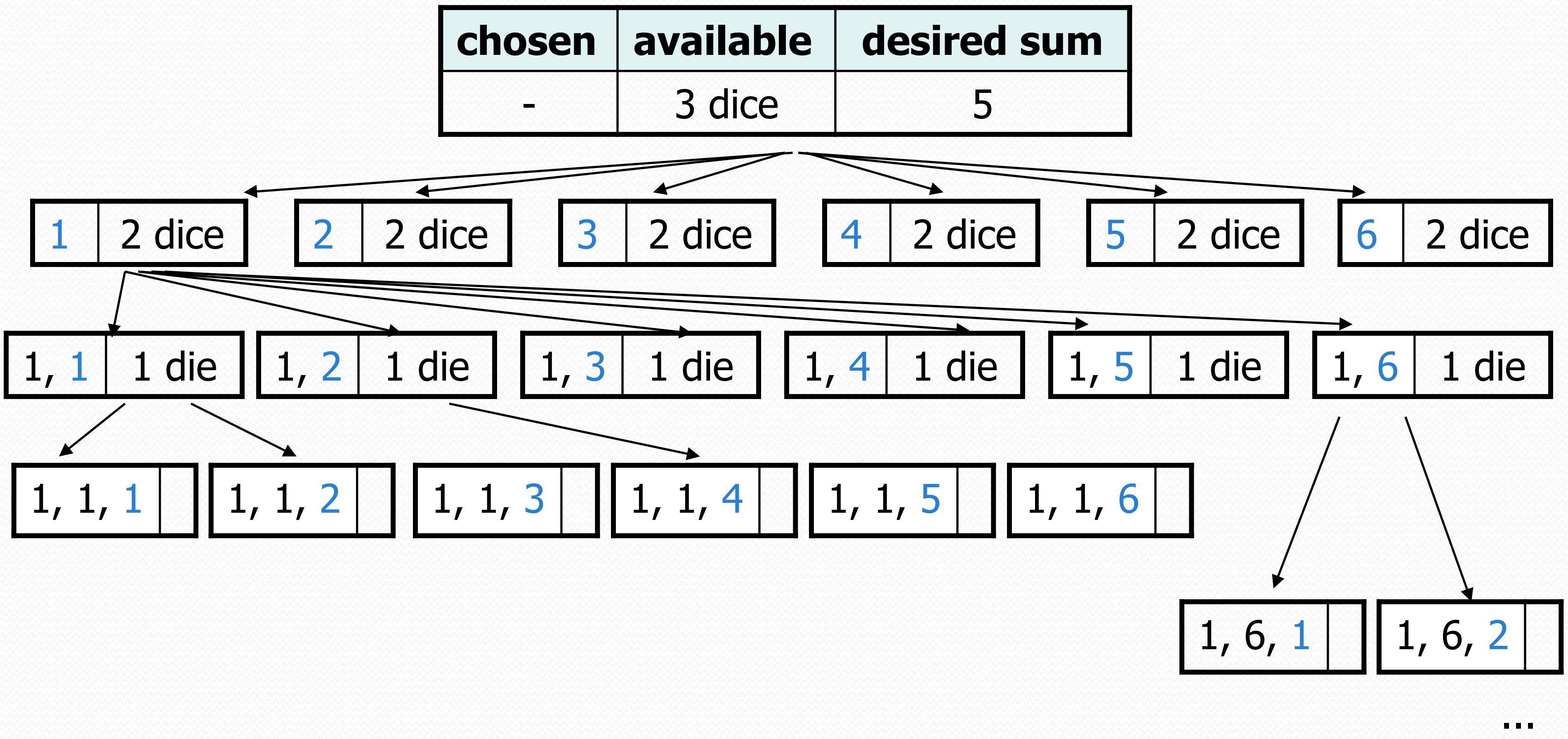
```
diceSum(2, 7);

    [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?

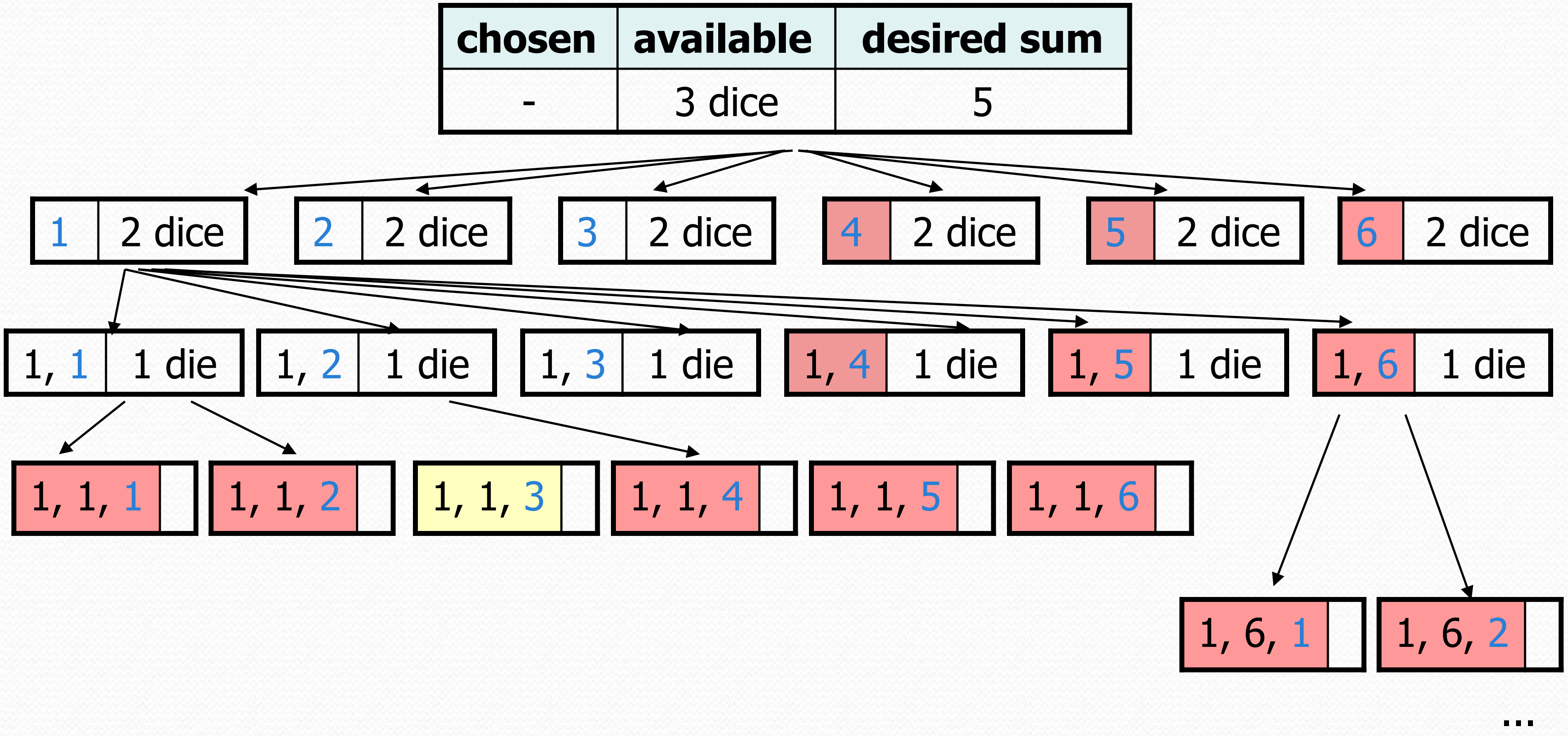


Optimizations

- 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



chosen | available | desired sum
- | 3 dice | 5
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
...