CS 106B, Lecture 9
Exhaustive Search and Backtracking

reading:

Programming Abstractions in C++, Chapters 8.2 - 8.3; 9

Exhaustive search

- **exhaustive search**: Exploring every possible combination from a set of choices or values.
  - often implemented recursively

  Applications:
  - producing all permutations of a set of values
  - enumerating all possible names, passwords, etc.
  - combinatorics and logic programming

- Often the search space consists of many ***decisions***, each of which has several available ***choices***.
  - Example: When enumerating all 5-letter strings, each of the 5 letters is a *decision*, and each of those decisions has 26 possible *choices*.

Exhaustive search

A general pseudo-code algorithm for exhaustive search:

Explore(*decisions*):

– if there are no more decisions to make: Stop.

– else, let's handle one decision ourselves, and the rest by recursion.
for each available choice *C* for this decision:

- **Choose** *C*.
- **Explore** the remaining decisions that could follow *C*.

Exercise: printAllBinary

printBinary

- Write a recursive function **`printAllBinary`** that accepts an integer number of digits and prints all binary numbers that have exactly that many digits, in ascending order, one per line.

  - `printAllBinary(2);` `printAllBinary(3);`

```
00          000
01          001
10          010
11          011
            100
            101
            110
            111
```

  - Use recursion; do not use any loops.
  - How is this problem self-similar (recursive)?

Helper functions

- If the required function doesn't accept the parameters you need:
  - Write a **helper function** that accepts more parameters.
  - Extra params can represent current state, choices made, etc.

```
returnType functionName(params) {
    ...
    return helper(params, moreParams);
}

returnType helper(params, moreParams) {
    ...
}
```

(use moreParams to help solve the problem)

printAllBinary solution

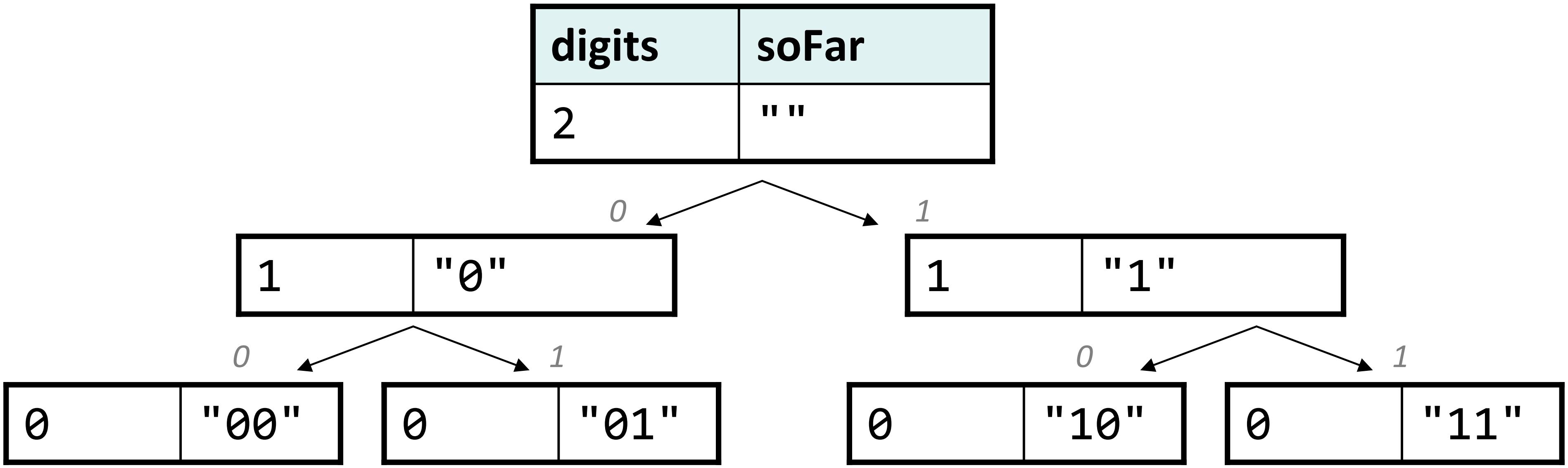
```
void printAllBinary(int digits) {
    printAllBinaryHelper(digits, "");
}

void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

– *Observation:* We write a 'helper' function to pass along the digits that were chosen so far by previous calls.

A tree of calls

- `printAllBinary(2);`



  - This kind of diagram is called a *call tree* or *decision tree*.
  - Think of each call as a choice or decision made by the algorithm:
    - Should I choose 0 as the next digit?
    - Should I choose 1 as the next digit?

The base case

```
void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

- The **base case** is where the code stops after doing its work.
  - `pAB(3) -> pAB(2) -> pAB(1) -> pAB(0)`
- Each call should keep track of the work it has done.
- Base case should print the result of the work done by prior calls.
  - Work is kept track of in some variable(s) - in this case, `string soFar`.

Exercise: printDecimal

- Write a recursive function **`printDecimal`** that accepts an integer number of digits and prints all base-10 numbers that have exactly that many digits, in ascending order, one per line.
  - `printDecimal(2);`

    ```
    00
    01
    02
    ..
    98
    99
    ```

    `printDecimal(3);`

    ```
    000
    001
    002
    ...
    997
    998
    999
    ```

  - Use recursion. *

printDecimal solution

```
void printDecimal(int digits) {
    printDecimalHelper(digits, "");
}

void printDecimalHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        for (int i = 0; i < 10; i++) {
            printDecimalHelper(digits - 1, soFar +
                               integerToString(i));
        }
    }
}
```

– *Observation:* When the set of digit choices available is large, using a loop to enumerate them avoids redundancy. *(This is okay!)*

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
  - escaping from a maze

Backtracking algorithms

A general pseudo-code algorithm for backtracking problems:

Explore(*decisions*):

- if there are no more decisions to make: stop.
- else, let's handle one decision ourselves, and the rest by recursion.
  for each available choice *C* for this decision:
  - **Choose** *C*.
  - **Explore** the remaining decisions that could follow *C*.
  - **Un-choose** *C*. (backtrack!)

Exercise: Dice roll sum

diceSum

- Write a function **`diceSum`** that accepts two integer parameters: a number of dice to roll, and a desired sum of all die values. Output all combinations of die values 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}
```

Easier: Dice rolls

diceRoll

- **Suggestion:** First just output <u>all</u> possible combinations of values that could appear on the dice.

```
diceSum(2, 7);                                   diceSum(3, 7);
```

```
{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}
```

```
{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}
```

  – How is this problem recursive (self-similar)?

Examining the problem

- Generate all possible sequences of values:

  for (each possible first die value):

    for (each possible second die value):

      for (each possible third die value):

        ...

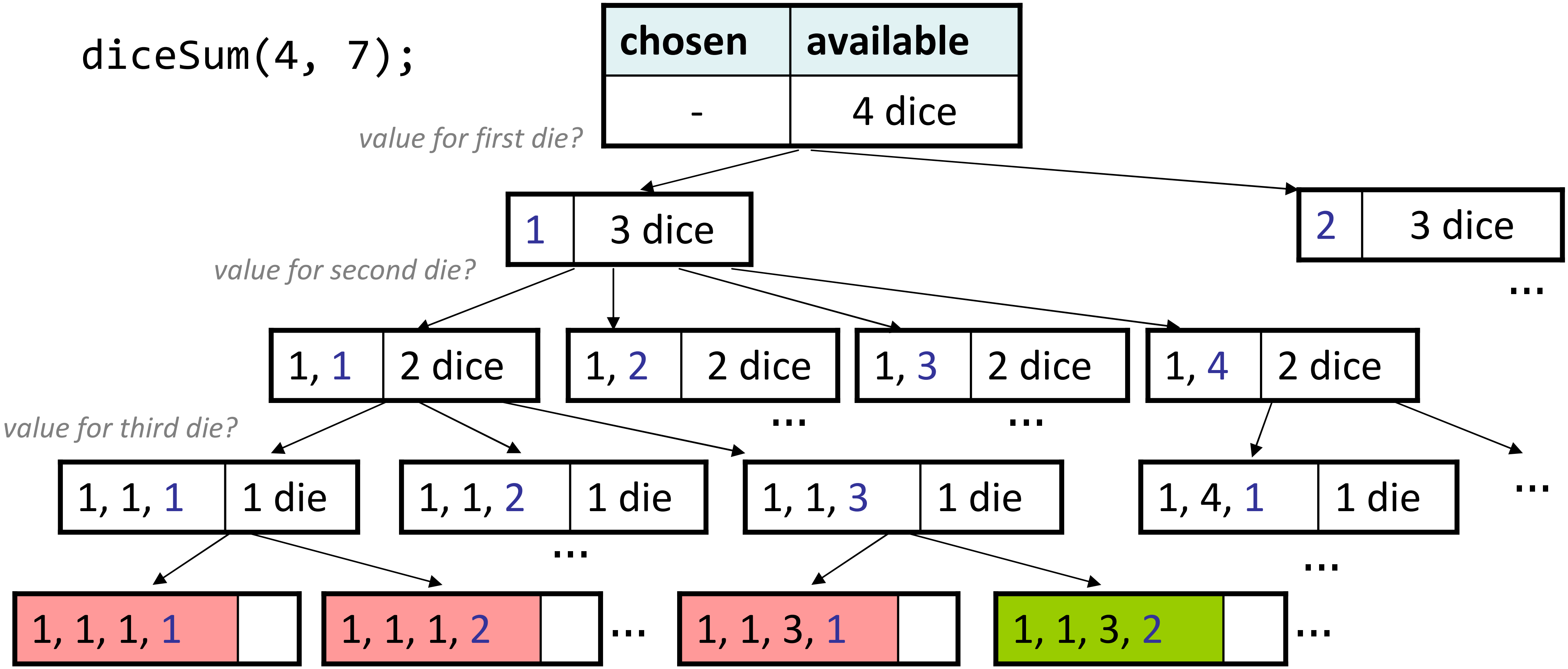
          print!

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

    - How many loops are needed?
    - How can we completely explore such a large search space?

A decision tree

```
diceSum(4, 7);
```



Initial solution

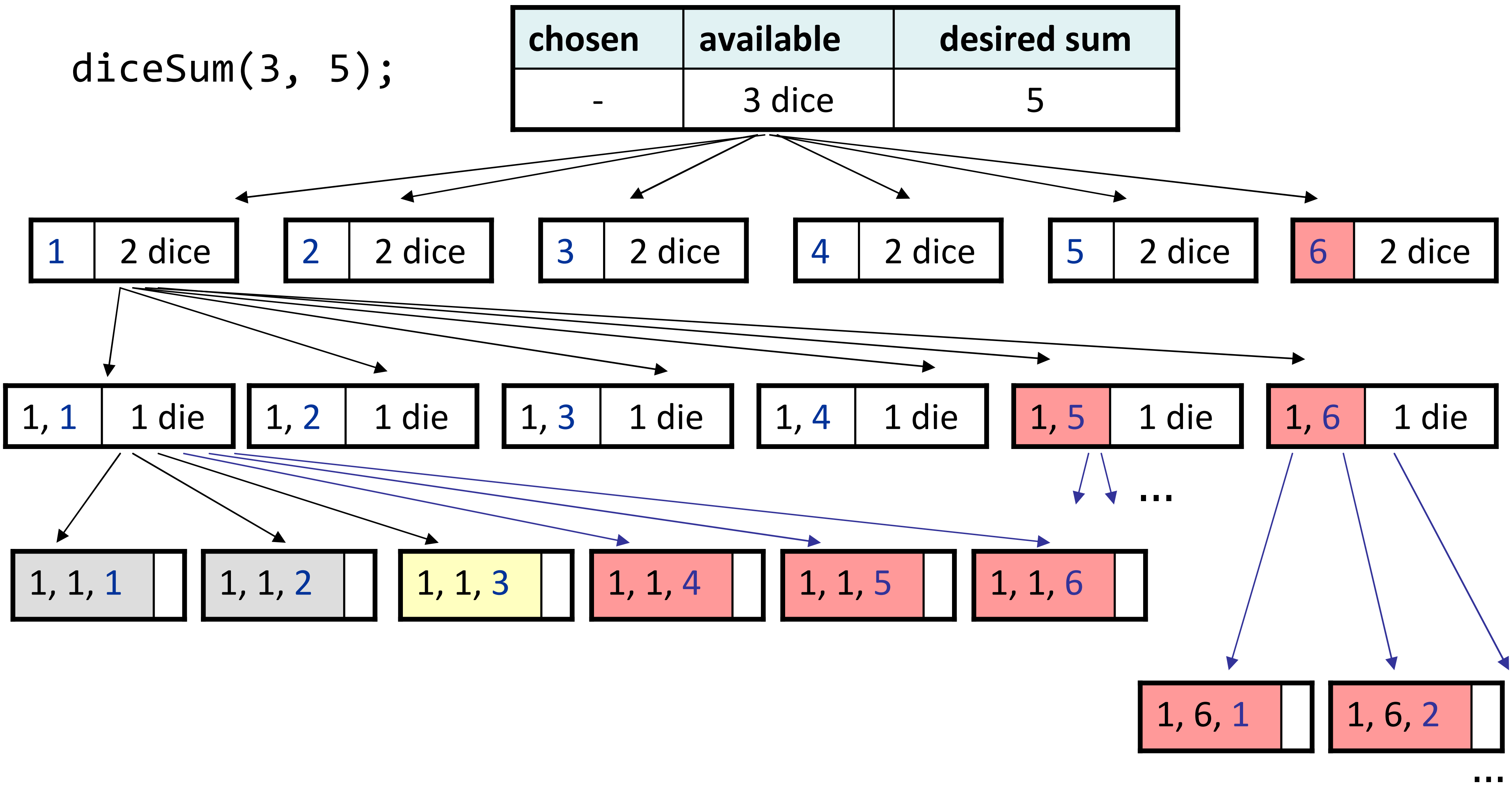
```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, desiredSum, chosen);
}

void diceSumHelper(int dice, int desiredSum, Vector<int>& chosen) {
    if (dice == 0) {
        if (sumAll(chosen) == desiredSum) {
            cout << chosen << endl;                         // base case
        }
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                                  // choose
            diceSumHelper(dice - 1, desiredSum, chosen);    // explore
            chosen.remove(chosen.size() - 1);               // un-choose
        }
    }
}

int sumAll(const Vector<int>& v) {   // adds the values in given vector
    int sum = 0;
    for (int k : v) { sum += k; }
    return sum;
}
```

Wasteful decision tree

```
diceSum(3, 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 sum is already **too high**.
    - (Even rolling 1 for all remaining dice would exceed the desired sum.)
  - Sometimes the current sum is already **too low**.
    - (Even rolling 6 for all remaining dice would exceed the desired sum.)
  - The code must **re-compute** the sum many times.
    - (1+1+1 = ..., 1+1+2 = ..., 1+1+3 = ..., 1+1+4 = ..., ...)

diceSum solution

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, 0, desiredSum, chosen);
}

void diceSumHelper(int dice, int sum, int desiredSum, Vector<int>& chosen) {
    if (dice == 0) {
        if (sum == desiredSum) {
            cout << chosen << endl;                              // base case
        }
    } else if (sum + 1*dice <= desiredSum
            && sum + 6*dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);                                         // choose
            diceSumHelper(dice - 1, sum + i, desiredSum, chosen); // explore
            chosen.remove(chosen.size() - 1);                    // un-choose
        }
    }
}
```

For you to think about...

- How would you modify **`diceSum`** so that it prints only unique combinations of dice, ignoring order?
  - (e.g. don't print both `{1, 1, 5}` and `{1, 5, 1}`)

```
diceSum2(2, 7);
```

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

```
diceSum2(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}
```