CS 106B, Lecture 11
Exhaustive Search and Backtracking

This document is copyright (C) Stanford Computer Science and Ashley Taylor, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Marty Stepp, Chris Gregg, Keith Schwarz, Julie Zelenski, Jerry Cain, Eric Roberts, Mehran Sahami, Stuart Reges, Cynthia Lee, and others

Plan for Today

- New recursive problem-solving techniques
  - Exhaustive Search
  - Backtracking

Plan for Today

- New recursive problem-solving techniques
  - Exhaustive Search
  - Backtracking

Exhaustive search

- **exhaustive search**: Exploring every possible combination from a set of choices.
  - often implemented recursively
  - Sometimes called *recursive enumeration*

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

- 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* by modifying parameters.
- **Explore** the remaining decisions that could follow *C*.
- **Un-choose** *C* by returning parameters to original state (if necessary).

Exercise: printAllBinary

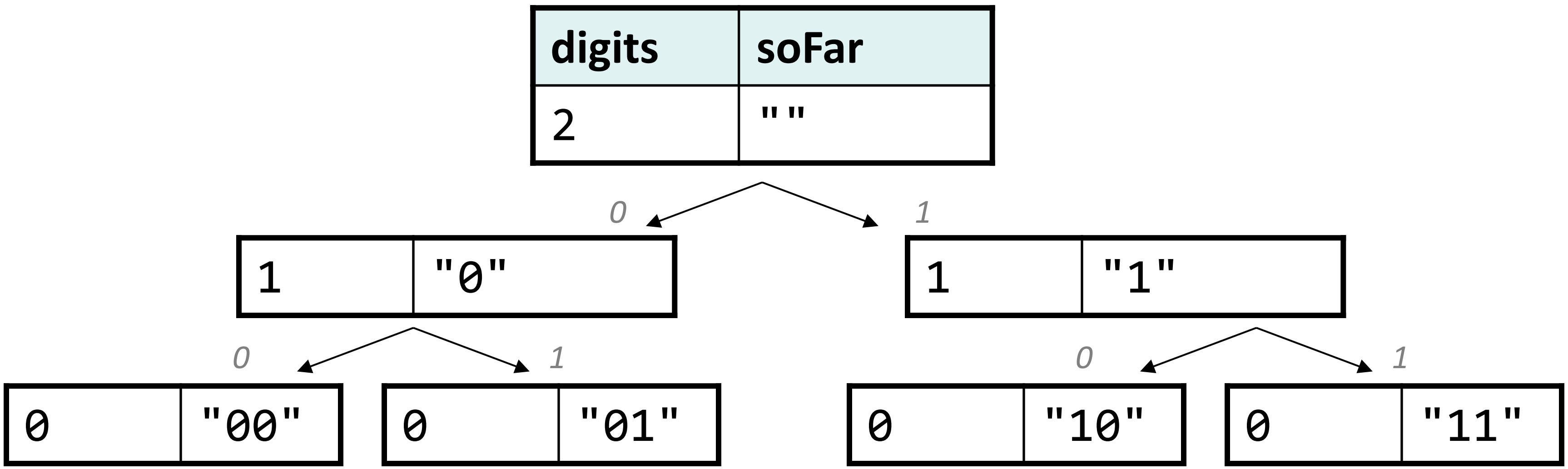
- Write a recursive function **`printAllBinary`** that accepts an integer number of digits and prints all binary numbers that have exactly that many digits, in ascending order, one per line.
  - `printAllBinary(2);`
    ```
    00
    01
    10
    11
    ```
    `printAllBinary(3);`
    ```
    000
    001
    010
    011
    100
    101
    110
    111
    ```

printAllBinary solution

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

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

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 or 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
    ```

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));
        }
    }
}
```

Announcements

- Homework 3 due on Wednesday at **5PM**

- **Midterm next Wednesday,** 7/24 7-9PM

Plan for Today

- New recursive problem-solving techniques
  - Exhaustive Search
  - Backtracking

Backtracking

- **Backtracking**: Finding solution(s) by trying all possible paths and then abandoning them if they are not suitable.
  - a "brute force" algorithmic technique
  - often implemented recursively
  - Could involve looking for **one** solution
    - If any of the paths found a solution, a solution exists! If none find a solution, no solution exists
  - Could involve finding **all solutions**
  - Idea: it's exhaustive search **with conditions**

Applications:

- games: anagrams, crosswords, word jumbles, 8 queens, sudoku
- combinatorics and logic programming
- escaping from a maze

Backtracking: One Solution

A general pseudo-code algorithm for backtracking problems searching for one solution

Backtrack(*decisions*):

- if there are no more decisions to make:
  - if our current solution is valid, return **true**
  - else, return **false**
- else, let's handle one decision ourselves, and the rest by recursion. for each available **valid** choice *C* for this decision:
  - **Choose** *C* by modifying parameters.
  - **Explore** the remaining decisions that could follow *C*. If any of them find a solution, return **true**
  - **Un-choose** *C* by returning parameters to original state (if necessary).
- If no solutions were found, **return false**

Backtracking: All Solutions

A general pseudo-code algorithm for backtracking problems searching for all solutions

Backtrack(*decisions*):

- if there are no more decisions to make:
  - if our current solution is valid, add it to our list of found solutions
  - else, do nothing or return
- else, let's handle one decision ourselves, and the rest by recursion. for each available **valid** choice *C* for this decision:
  - **Choose** *C* by modifying parameters.
  - **Explore** the remaining decisions that could follow *C*. Keep track of which solutions the recursive calls find.
  - **Un-choose** *C* by returning parameters to original state (if necessary).
- Return the list of solutions found by all the helper recursive calls.

Exercise: Dice roll sum

- 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

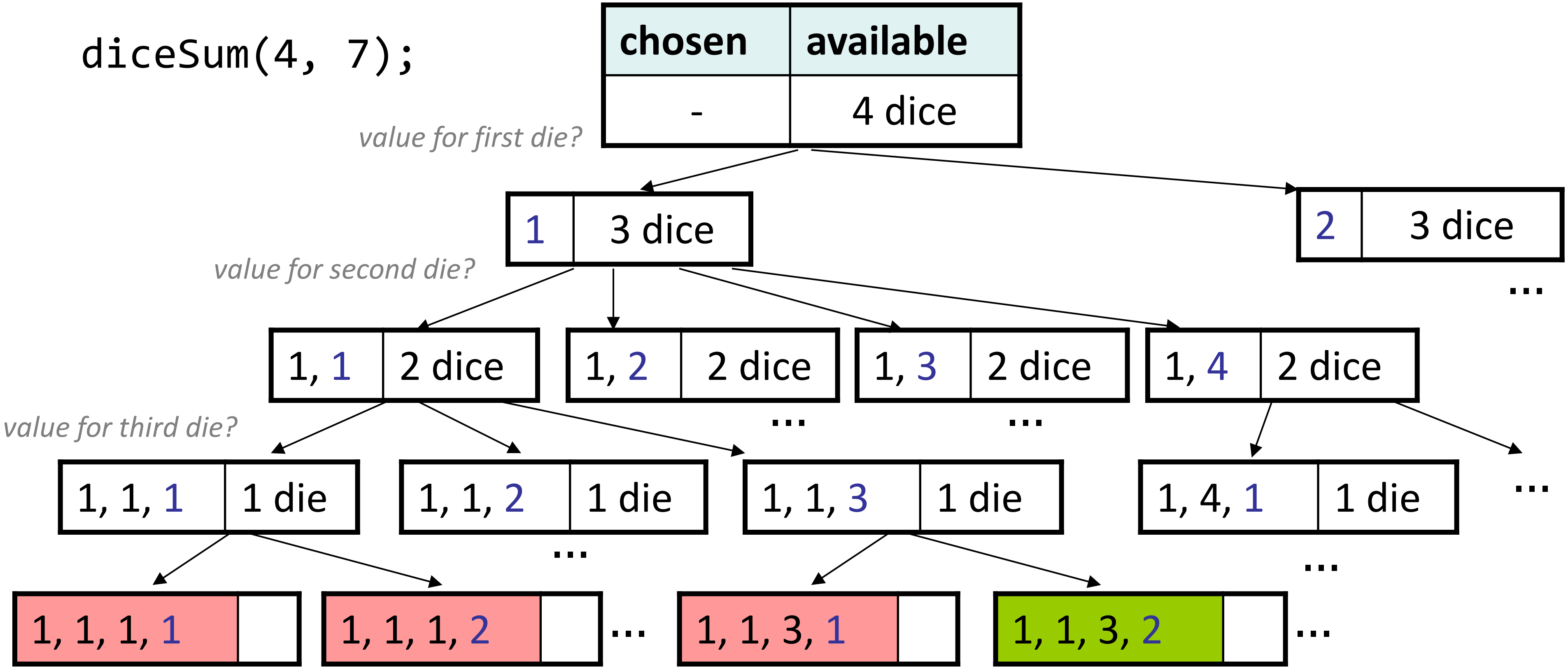
- **Suggestion:** First just output all possible combinations of values that could appear on the dice.
- This is just exhaustive search!
- In general, starting with exhaustive search and then adding conditions is not a bad idea

```
diceSum(2, 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}
```

```
diceSum(3, 7);

{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

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

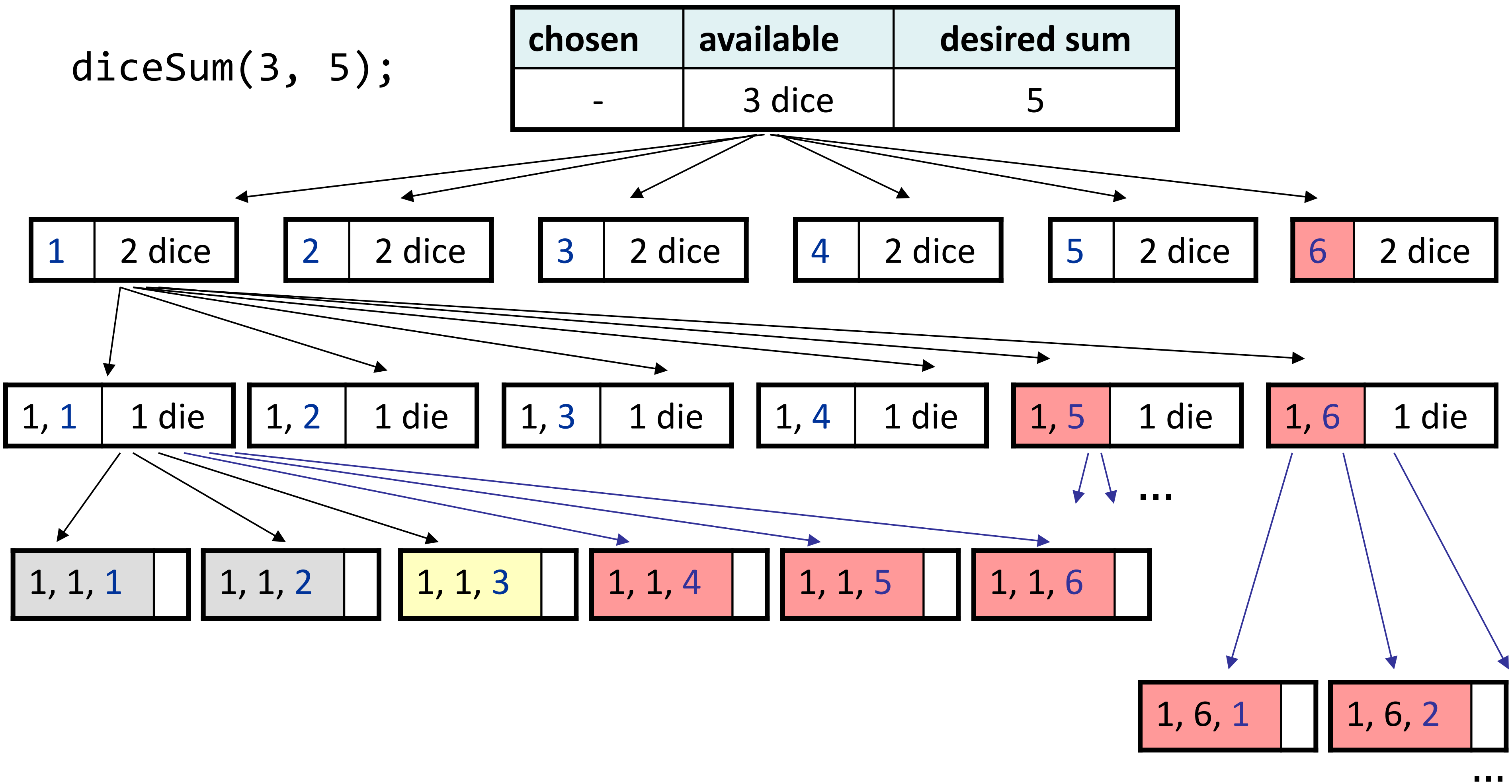


Initial solution

```
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, desiredSum, chosen);
}

void diceSumHelper(int dice, int desiredSum, Vector<int>& chosen) {
    if (dice == 0 && desiredSum == 0) {
        cout << chosen << endl;
    } else if (dice > 0 && (dice <= desiredSum && desiredSum <= dice*6)) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceSum(dice - 1, desiredSum - i, chosen);
            chosen.removeBack();
        }
    }
}
```