Road Map - Quarter

CS Concepts

- Client/Implementer
- Efficiency
- Recursion
- Regular Expressions
- Grammars
- Sorting
- Backtracking
- Hashing
- Huffman Compression

Java Language

- Exceptions
- Interfaces
- References
- Comparable
- Generics
- Inheritance/Polymorphism
- Abstract Classes

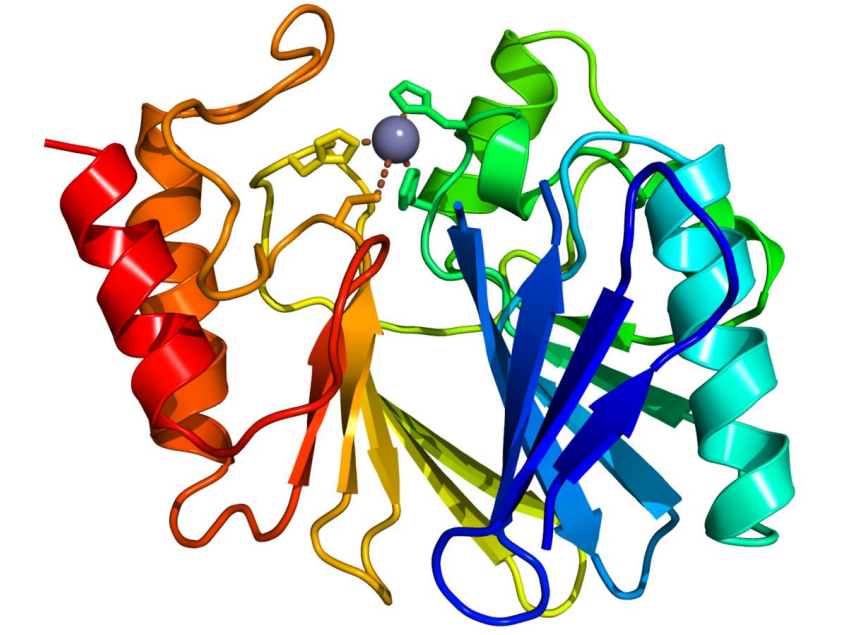
Data Structures

- Lists
- Stacks
- Queues
- Sets
- Maps
- Priority Queues

Java Collections

- Arrays
- ArrayList 🛠
- LinkedList 🛠
- Stack
- TreeSet / TreeMap
- HashSet / HashMap
- PriorityQueue

Two Not-so-Similar Problems

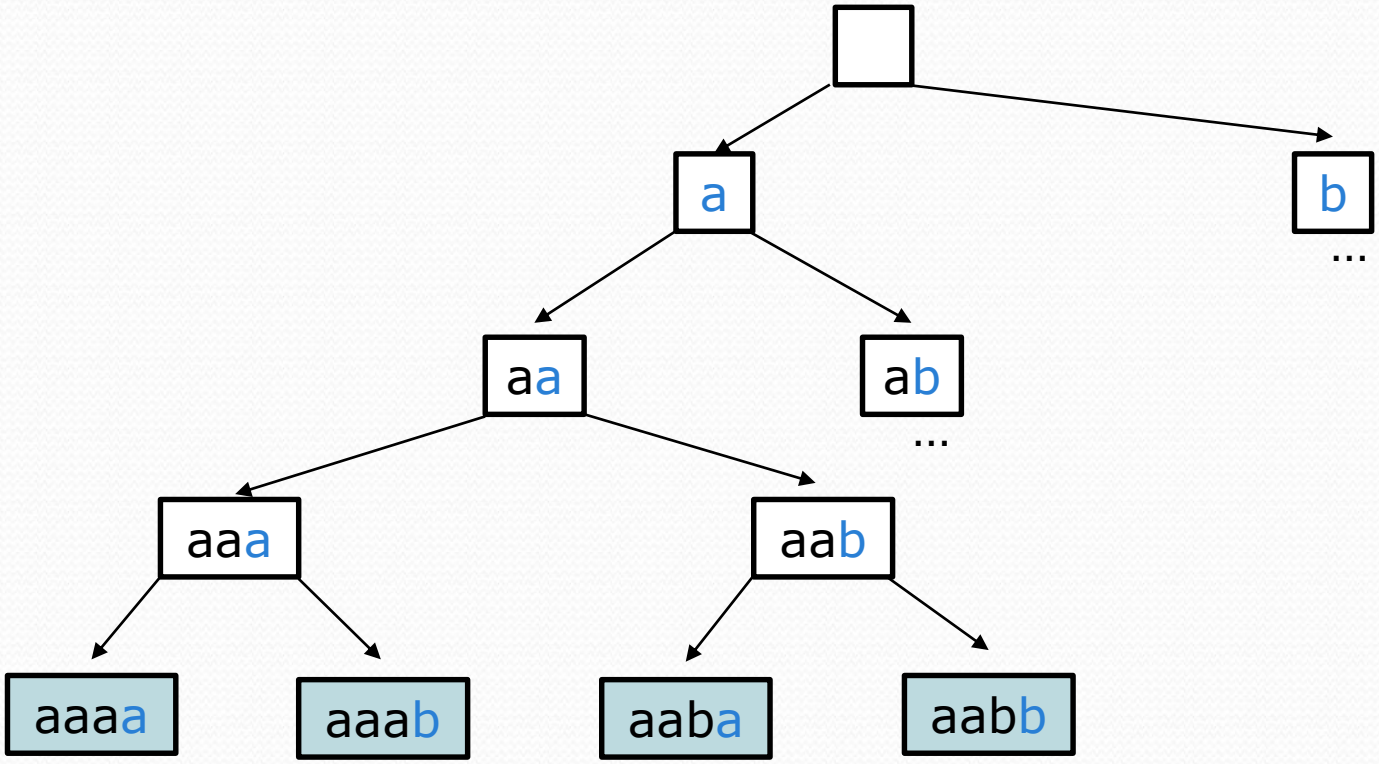


Exercise: fourAB

- 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



pollev.com/cse143

- Suppose we had the following method:

```
public static void mystery(String soFar) {
    if (soFar.length() == 3) {
        System.out.println(soFar);
    } else {
        mystery(soFar + "d");
        mystery(soFar + "a");
        mystery(soFar + "b");
    }
}
```

- What is the **fourth** line of output of the call mystery("");
  - This means you can stop once you've found 4 lines of output

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.

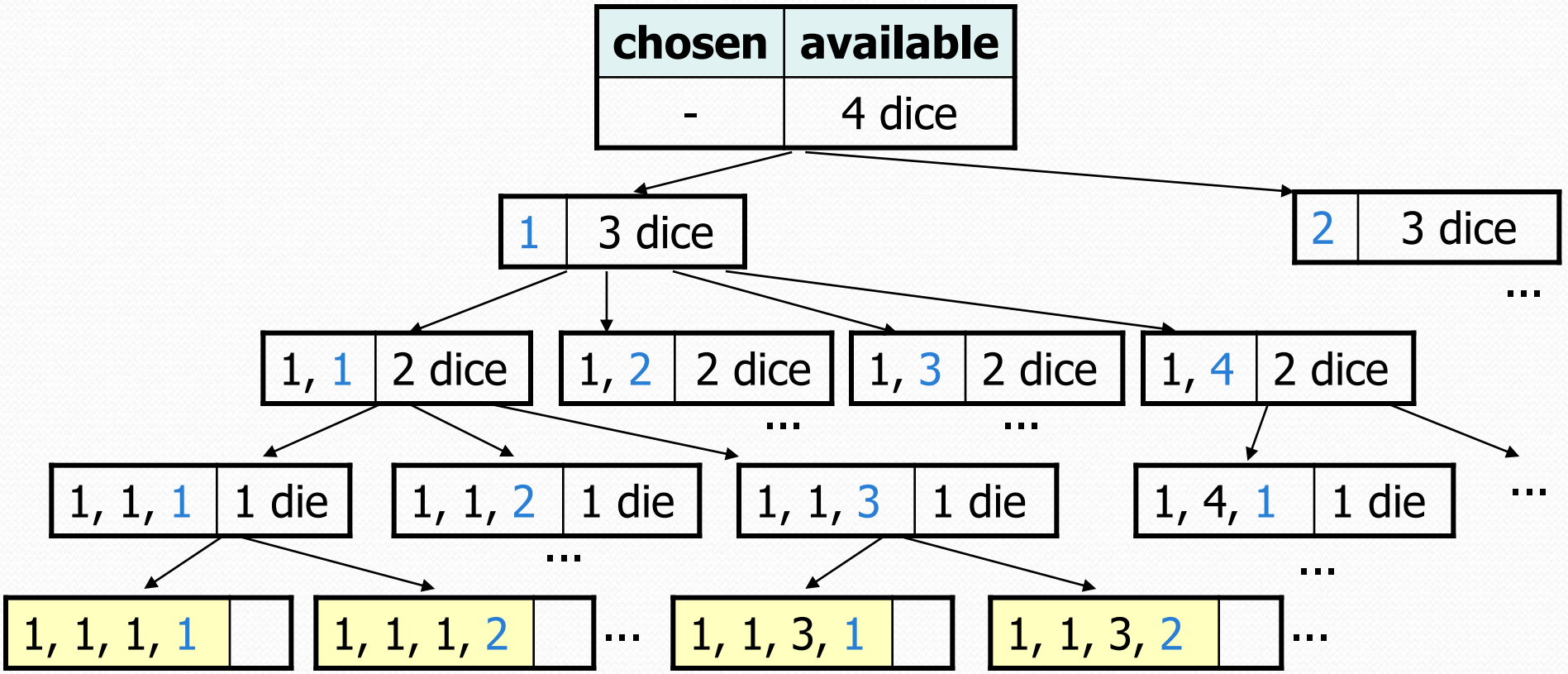
```
diceRoll(2);
```

```
[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



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?

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?