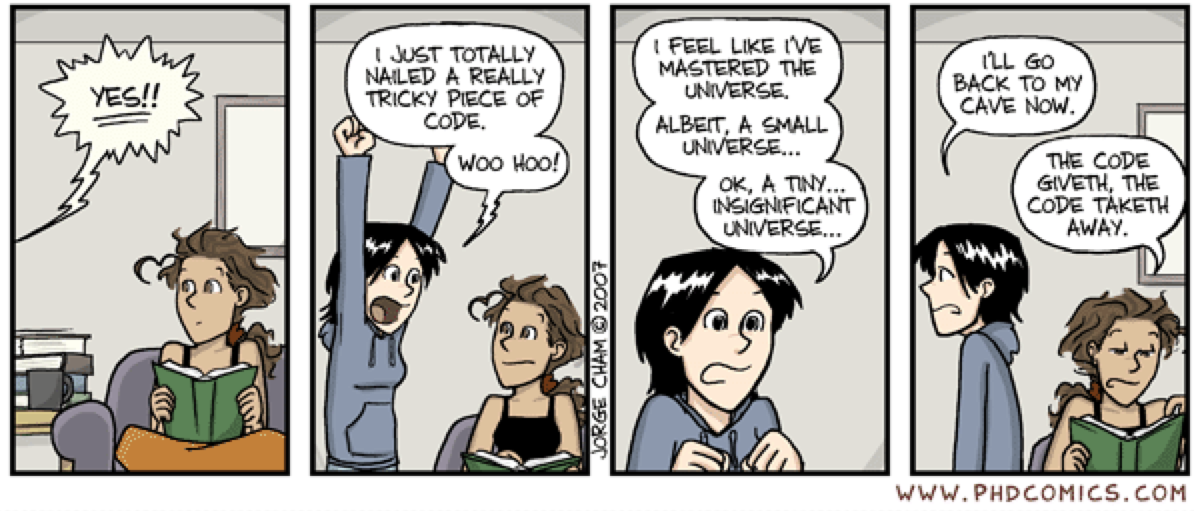
CSE 143

Recursive backtracking


YES!!
I JUST TOTALLY NAILED A REALLY TRICKY PIECE OF CODE.
WOO HOO!
JORGE CHAM © 2007
I FEEL LIKE I'VE MASTERED THE UNIVERSE.
ALBEIT, A SMALL UNIVERSE...
OK, A TINY... INSIGNIFICANT UNIVERSE...
I'LL GO BACK TO MY CAVE NOW.
THE CODE GIVETH, THE CODE TAKETH AWAY.
WWW.PHDCOMICS.COM

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]
```

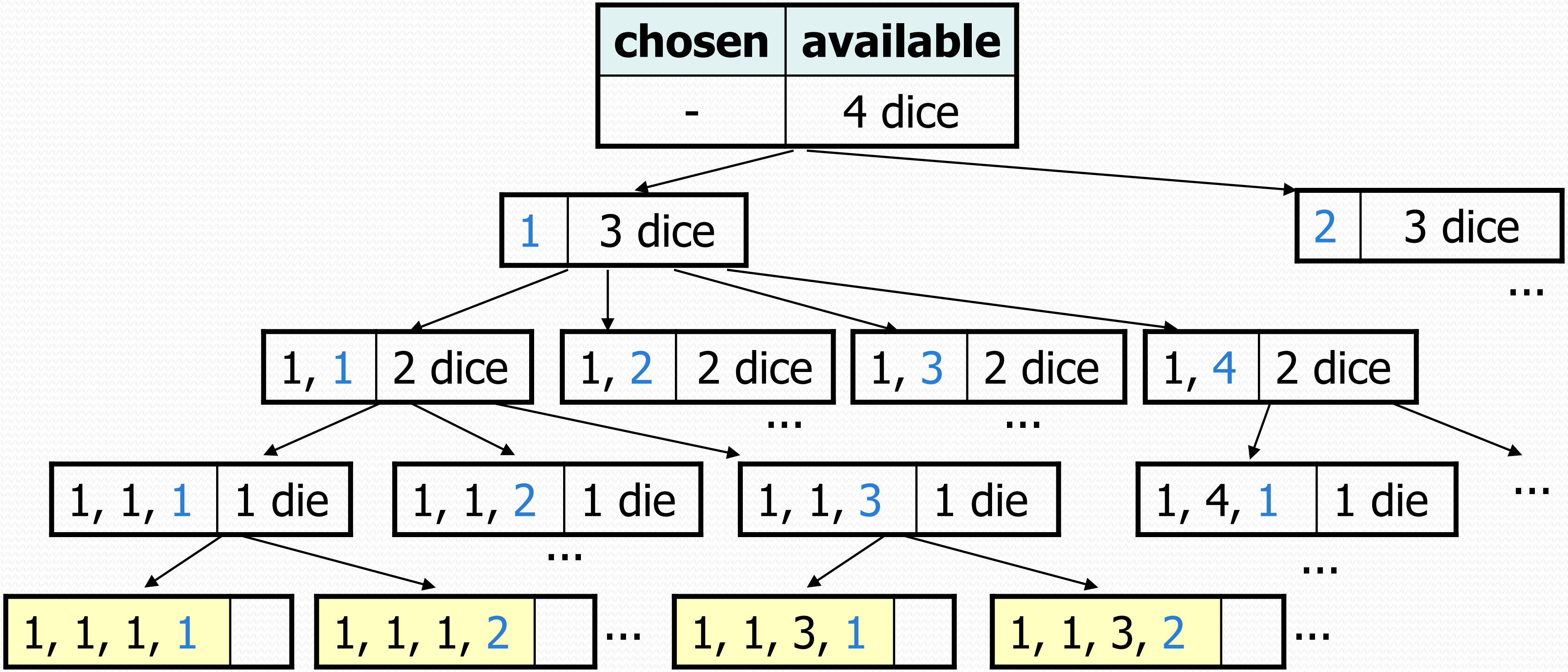
Examining the problem

- 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?

A decision tree



chosen | available

- | 4 dice

1 | 3 dice

2 | 3 dice

...

1, 1 | 2 dice

1, 2 | 2 dice

...

1, 3 | 2 dice

...

1, 4 | 2 dice

1, 1, 1 | 1 die

1, 1, 2 | 1 die

...

1, 1, 3 | 1 die

1, 4, 1 | 1 die

...

...

1, 1, 1, 1

1, 1, 1, 2

...

1, 1, 3, 1

1, 1, 3, 2

...

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);
```

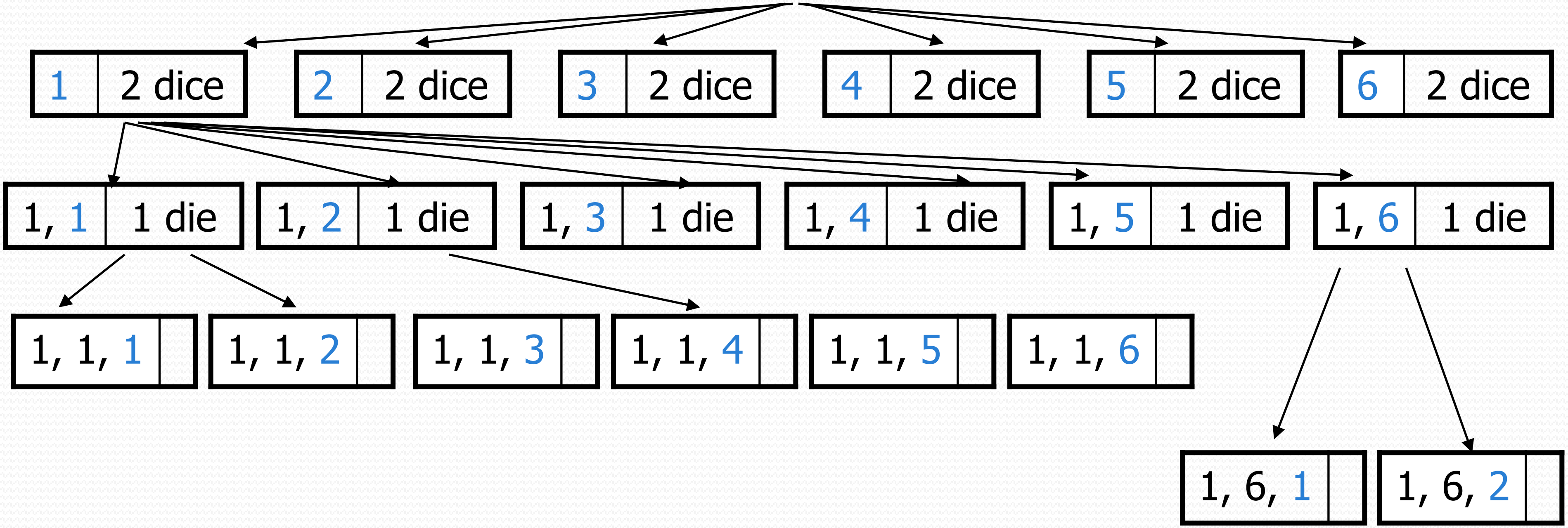
```
[1, 6]
[2, 5]
[3, 4]
[4, 3]
[5, 2]
[6, 1]
```

```
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



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

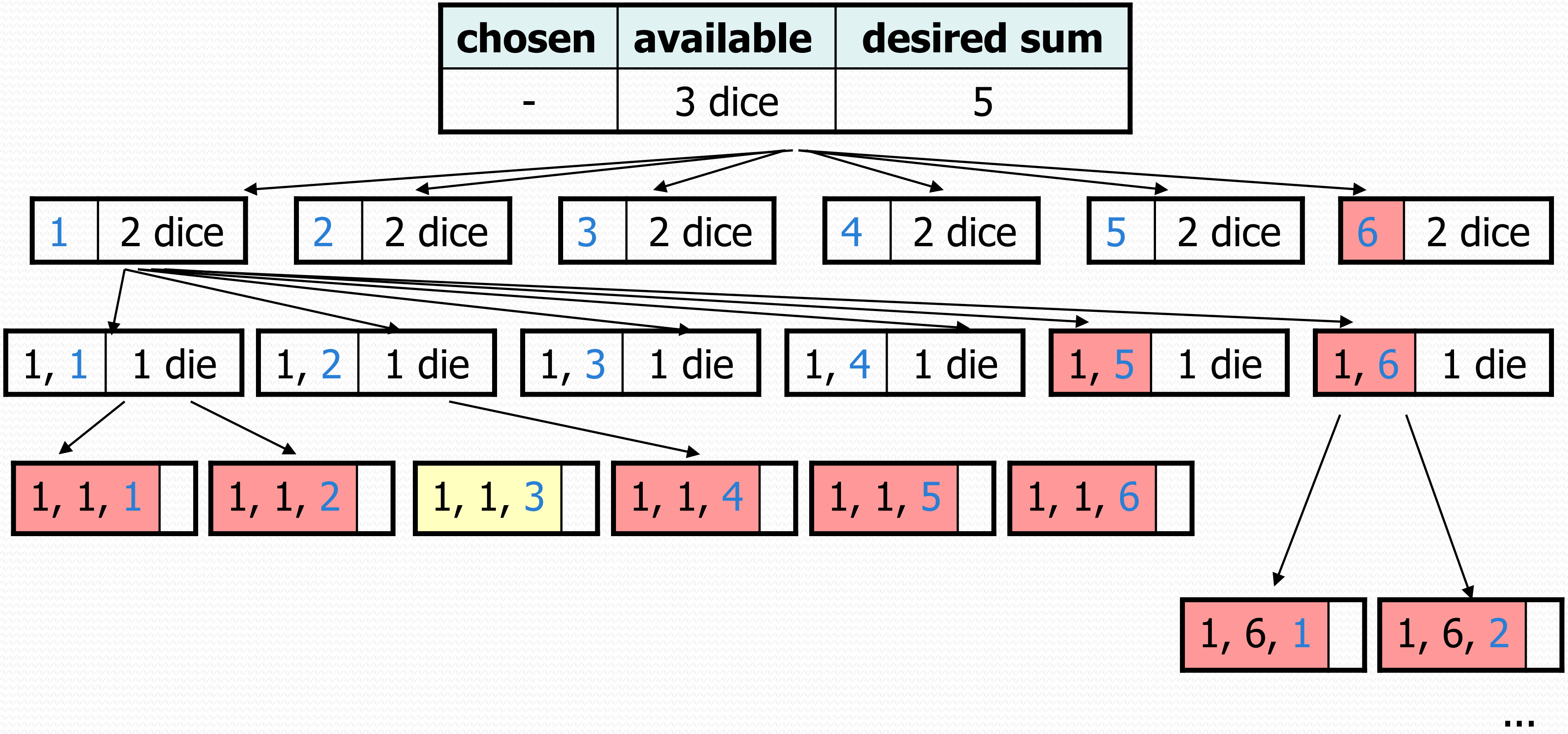
1, 1, 6

1, 6, 1

1, 6, 2

...

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

...

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 algorithms

A general pseudo-code algorithm for backtracking problems:

Explore(**choices**):

- if there are no more **choices** to make: stop.
- else:
  - Make a single choice **C**.
  - Explore the remaining **choices**.
  - Un-make choice **C**, if necessary. (backtrack!)

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: Combinations

- Write a method `combinations` that accepts a string *s* and an integer *k* as parameters and outputs all possible *k* -letter words that can be formed from unique letters in that string. The arrangements may be output in any order.
  - Example:
    `combinations("GOOGLE", 3)`
    outputs the sequence of lines at right.
  - To simplify the problem, you may assume that the string *s* contains at least *k* unique characters.

```
EGL   LEG
EGO   LEO
ELG   LGE
ELO   LGO
EOG   LOE
EOL   LOG
GEL   OEG
GEO   OEL
GLE   OGE
GLO   OGL
GOE   OLE
GOL   OLG
```

Initial attempt

```
public static void combinations(String s, int length) {
    combinations(s, "", length);
}

private static void combinations(String s, String chosen, int length) {
    if (length == 0) {
        System.out.println(chosen);     // base case: no choices left
    } else {
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1);
            if (!chosen.contains(ch)) {
                String rest = s.substring(0, i) + s.substring(i + 1);
                combinations(rest, chosen + ch, length - 1);
            }
        }
    }
}
```

- Problem: Prints same string multiple times.

Exercise solution

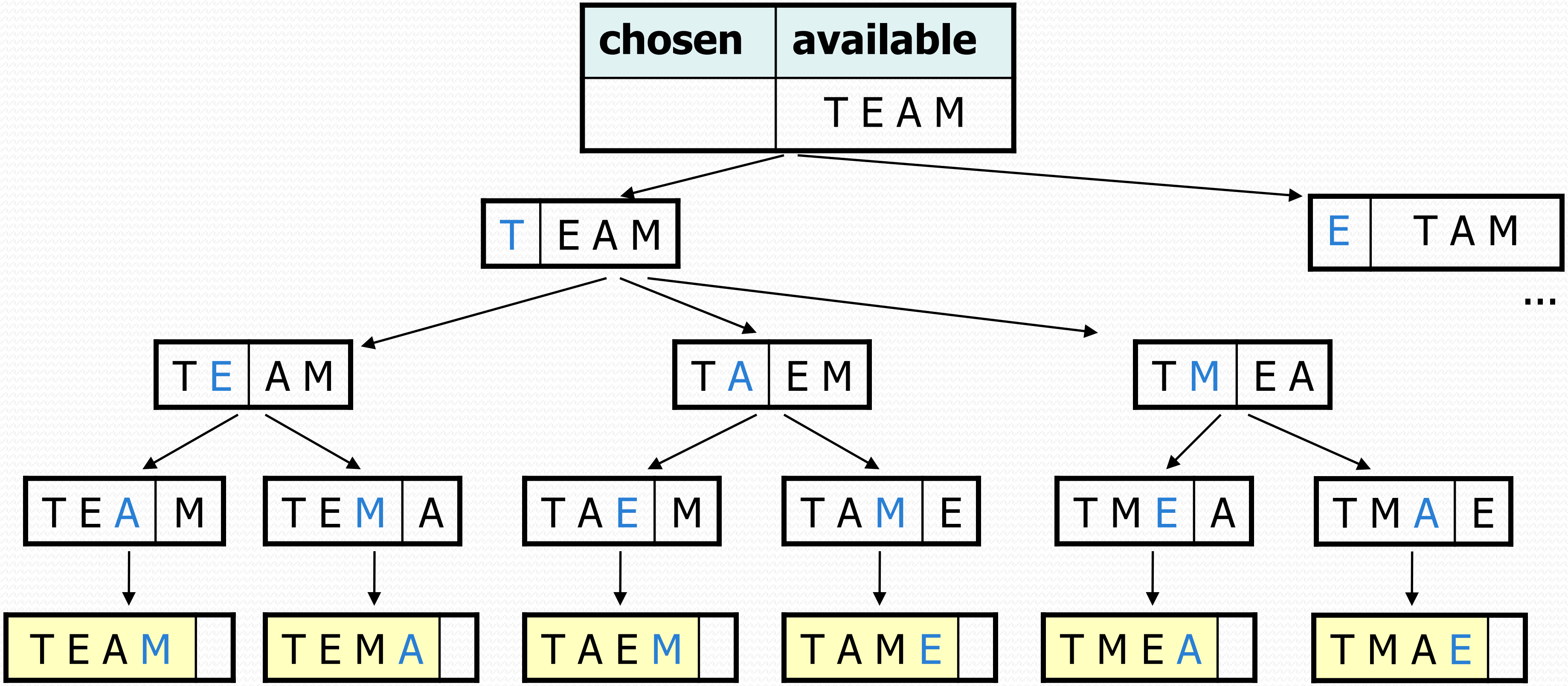
```
public static void combinations(String s, int length) {
    Set<String> all = new TreeSet<String>();
    combinations(s, "", all, length);
    for (String comb : all) {
        System.out.println(comb);
    }
}

private static void combinations(String s, String chosen,
                                 Set<String> all, int length) {
    if (length == 0) {
        all.add(chosen);         // base case: no choices left
    } else {
        for (int i = 0; i < s.length(); i++) {
            String ch = s.substring(i, i + 1);
            if (!chosen.contains(ch)) {
                String rest = s.substring(0, i) + s.substring(i + 1);
                combinations(rest, chosen + ch, all, length - 1);
            }
        }
    }
}
```

Exercise: Permutations

- Write a method `permute` that accepts a string as a parameter and outputs all possible rearrangements of the letters in that string. The arrangements may be output in any order.
  - Example:
    `permute("TEAM")`
    outputs the following sequence of lines:

```
TEAM  |  ATEM
TEMA  |  ATME
TAEM  |  AETM
TAME  |  AEMT
TMEA  |  AMTE
TMAE  |  AMET
ETAM  |  MTEA
ETMA  |  MTAE
EATM  |  META
EAMT  |  MEAT
EMTA  |  MATE
EMAT  |  MAET
```

Decision tree



chosen	available
	TEAM

T EAM

E TAM

...

TE AM

TA EM

TM EA

TEA M

TEM A

TAE M

TAM E

TME A

TMA E

TEAM

TEMA

TAEM

TAME

TMEA

TMAE

Exercise solution

```
// Outputs all permutations of the given string.
public static void permute(String s) {
    permute(s, "");
}

private static void permute(String s, String chosen) {
    if (s.length() == 0) {
        // base case: no choices left to be made
        System.out.println(chosen);
    } else {
        // recursive case: choose each possible next letter
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);               // choose
            s = s.substring(0, i) + s.substring(i + 1);
            chosen += c;

            permute(s, chosen);                 // explore

            s = s.substring(0, i) + c + s.substring(i);
            chosen = chosen.substring(0, chosen.length() - 1);
        }                                       // un-choose
    }
}
```