

Building Java Programs Chapter 12

Recursion

Recursion

- **recursion:** The definition of an operation in terms of itself.

– Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- **recursive programming:** Writing methods that call themselves to solve problems recursively.

– An equally powerful substitute for *iteration* (loops)
– Particularly well-suited to solving certain types of problems

2

Why learn recursion?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- A key component of the rest of our assignments in CSE 143

3

Exercise

- (To a student in the front row)
How many students total are directly behind you in your "column" of the classroom?

– You have poor vision, so you can see only the people right next to you.
So you can't just look back and count.
– But you are allowed to ask questions of the person next to you.

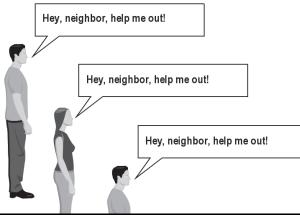


– How can we solve this problem?
(recursively)

4

The idea

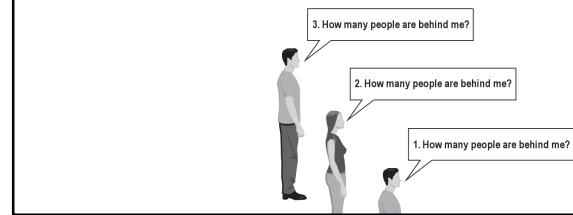
- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
 - Each person can solve a small part of the problem.
 - What is a small version of the problem that would be easy to answer?
 - What information from a neighbor might help me?



5

Recursive algorithm

- Number of people behind me:
 - If there is someone behind me, ask him/her how many people are behind him/her.
 - When they respond with a value **N**, then I will answer **N + 1**.
 - If there is nobody behind me, I will answer **0**.



6

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
 - **base case:** A simple occurrence that can be answered directly.
 - **recursive case:** A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
 - Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
 - A crucial part of recursive programming is identifying these cases.

7

Recursion in Java

- Consider the following method to print a line of * characters:

```
// Prints a line containing the given number of stars.
// Precondition: n >= 0
public static void printStars(int n) {
    for (int i = 0; i < n; i++) {
        System.out.print("*");
    }
    System.out.println(); // end the line of output
}
```

- Write a recursive version of this method (that calls itself).

- Solve the problem without using any loops.
- Hint: Your solution should print just one star at a time.

8

A basic case

- What are the cases to consider?
 - What is a very easy number of stars to print without a loop?

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        ...
    }
}
```

9

Handling more cases

- Handling additional cases, with no loops (in a bad way):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        System.out.println("*");
    } else if (n == 3) {
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else if (n == 4) {
        System.out.print("*");
        System.out.print("*");
        System.out.print("*");
        System.out.println("*");
    } else ...
}
```

10

Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else if (n == 2) {
        System.out.print("*");
        printStars(1); // prints **
    } else if (n == 3) {
        System.out.print("*");
        printStars(2); // prints ***
    } else if (n == 4) {
        System.out.print("*");
        printStars(3); // prints ****
    } else ...
}
```

11

Using recursion properly

- Condensing the recursive cases into a single case:

```
public static void printStars(int n) {
    if (n == 1) {
        // base case; just print one star
        System.out.println("*");
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

12

"Recursion Zen"

- The real, even simpler, base case is an n of 0, not 1:

```
public static void printStars(int n) {
    if (n == 0) {
        // base case; just end the line of output
        System.out.println();
    } else {
        // recursive case; print one more star
        System.out.print("*");
        printStars(n - 1);
    }
}
```

- **Recursion Zen:** The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.
(A CSE 143 informal term)

13

Recursive tracing

- Consider the following recursive method:

```
public static int mystery(int n) {  
    if (n < 10) {  
        return n;  
    } else {  
        int a = n / 10;  
        int b = n % 10;  
        return mystery(a + b);  
    }  
}
```

- What is the result of the following call?
`mystery(648)`

14

A recursive trace

```
mystery(648):  
    □ int a = 648 / 10;           // 64  
    □ int b = 648 % 10;          // 8  
    □ return mystery(a + b);     // mystery(72)  
  
mystery(72):  
    □ int a = 72 / 10;           // 7  
    □ int b = 72 % 10;          // 2  
    □ return mystery(a + b);     // mystery(9)  
  
mystery(9):  
    □ return 9;
```

15

Recursive tracing 2

- Consider the following recursive method:

```
public static int mystery(int n) {  
    if (n < 10) {  
        return (10 * n) + n;  
    } else {  
        int a = mystery(n / 10);  
        int b = mystery(n % 10);  
        return (100 * a) + b;  
    }  
}
```

- What is the result of the following call?
mystery(348)

16

A recursive trace 2

```

mystery(348)
  ▪ int a = mystery(34);
    ▪ int a = mystery(3);
      return (10 * 3) + 3; // 33
    ▪ int b = mystery(4);
      return (10 * 4) + 4; // 44
  • return (100 * 33) + 44; // 3344

  ▪ int b = mystery(8);
    return (10 * 8) + 8; // 88

- return (100 * 3344) + 88; // 334488

```

- What is this method really doing?

17

Exercise

- Write a recursive method `pow` accepts an integer base and exponent and returns the base raised to that exponent.

- Example: `pow(3, 4)` returns 81

- Solve the problem recursively and without using loops.

18

pow solution

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else {
        // recursive case: x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

19

An optimization

- Notice the following mathematical property:

$$\begin{aligned} 3^{12} &= 531441 &= 9^6 \\ &&= (3^2)^6 \\ 531441 &= (9^2)^3 \\ &&= ((3^2)^2)^3 \end{aligned}$$

- When does this "trick" work?
- How can we incorporate this optimization into our `pow` method?
- What is the benefit of this trick if the method already works?

20

pow solution 2

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is 1
        return 1;
    } else if (exponent % 2 == 0) {
        // recursive case 1: x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {
        // recursive case 2: x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

21

Exercise

- Write a recursive method `printBinary` that accepts an integer and prints that number's representation in binary (base 2).

- Example: `printBinary(7)` prints 111
- Example: `printBinary(12)` prints 1100
- Example: `printBinary(42)` prints 101010

place	10	1	32	16	8	4	2	1
value	4	2	1	0	1	0	1	0

- Write the method recursively and without using any loops.

22

Case analysis

- Recursion is about solving a small piece of a large problem.
 - What is 69743 in binary?
 - Do we know *anything* about its representation in binary?
 - Case analysis:
 - What is/are easy numbers to print in binary?
 - Can we express a larger number in terms of a smaller number(s)?
 - Suppose we are examining some arbitrary integer N .
 - If N 's binary representation is 10010101011
 - $(N / 2)$'s binary representation is 1001010101
 - $(N \% 2)$'s binary representation is 1

23

printBinary Solution

```
// Prints the given integer's binary representation.
// Precondition: n >= 0
public static void printBinary(int n) {
    if (n < 2) {
        // base case; same as base 10
        System.out.print(n);
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}
```

- Can we eliminate the precondition and deal with negatives?

24

printBinary solution 2

```
// Prints the given integer's binary representation.
public static void printBinary(int n) {
    if (n < 0) {
        // recursive case for negative numbers
        System.out.print("-");
        printBinary(-n);
    } else if (n < 2) {
        // base case; same as base 10
        System.out.print(n);
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}
```

25

Exercise

- Write a recursive method `isPalindrome` accepts a String and returns true if it reads the same forwards as backwards.

- <code>isPalindrome("madam")</code>	→ true
- <code>isPalindrome("racecar")</code>	→ true
- <code>isPalindrome("step on no pets")</code>	→ true
- <code>isPalindrome("able was I ere I saw elba")</code>	→ true
- <code>isPalindrome("Java")</code>	→ false
- <code>isPalindrome("rotater")</code>	→ false
- <code>isPalindrome("byebye")</code>	→ false
- <code>isPalindrome("notion")</code>	→ false

26

Exercise solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true; // base case
    } else {
        char first = s.charAt(0);
        char last = s.charAt(s.length() - 1);
        if (first != last) {
            return false;
        } // recursive case
        String middle = s.substring(1, s.length() - 1);
        return isPalindrome(middle);
    }
}
```

27

Exercise solution 2

```
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
public static boolean isPalindrome(String s) {
    if (s.length() < 2) {
        return true; // base case
    } else {
        return s.charAt(0) == s.charAt(s.length() - 1)
            && isPalindrome(s.substring(1, s.length() - 1));
    }
}
```

28

Exercise

- Write a recursive method `reverseLines` that accepts a file `Scanner` and prints the lines of the file in reverse order.

- Example input file:

```
Roses are red,
Violets are blue.
All my base
Are belong to you.
```

Expected console output:

```
Are belong to you.
All my base
Violets are blue.
Roses are red,
```



- What are the cases to consider?

- How can we solve a small part of the problem at a time?
- What is a file that is very easy to reverse?

29

Reversal pseudocode

- Reversing the lines of a file:
 - Read a line L from the file.
 - Print the rest of the lines in reverse order.
 - Print the line L.
- If only we had a way to reverse the rest of the lines of the file....

30

Reversal solution

```
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        // recursive case
        String line = input.nextLine();
        reverseLines(input);
        System.out.println(line);
    }
}
```

- Where is the base case?

31

Tracing our algorithm

- **call stack:** The method invocations running at any one time.

```
reverseLines(new Scanner("poem.txt"));
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine(); // "Roses are red."
    }
}
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine(); // "Violets are blue."
    }
}
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine(); // "All my base"
    }
}
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) {
        String line = input.nextLine(); // "Are belong to you."
    }
}
public static void reverseLines(Scanner input) {
    if (input.hasNextLine()) { // false
        ...
    }
}

Roses are red,
Violets are blue.
All my base
Are belong to you.

All my base
Violets are blue.
Roses are red,
```

32

Exercise

- Write a method `crawl` accepts a `File` parameter and prints information about that file.
 - If the `File` object represents a normal file, just print its name.
 - If the `File` object represents a directory, print its name and information about every file/directory inside it, indented.

```
cse143
    handouts
        syllabus.doc
        lecture_schedule.xls
    homework
        l-sortedintlist
            ArrayIntList.java
            SortedIntList.java
            index.html
            style.css
```

- **recursive data:** A directory can contain other directories.

33

File objects

- A `File` object (from the `java.io` package) represents a file or directory on the disk.

Constructor/method	Description
<code>File(String)</code>	creates <code>File</code> object representing file with given name
<code>canRead()</code>	returns whether file is able to be read
<code>delete()</code>	removes file from disk
<code>exists()</code>	whether this file exists on disk
<code>getName()</code>	returns file's name
<code>isDirectory()</code>	returns whether this object represents a directory
<code>length()</code>	returns number of bytes in file
<code>listFiles()</code>	returns a <code>File[]</code> representing files in this directory
<code>renameTo(File)</code>	changes name of file

34

Public/private pairs

- We cannot vary the indentation without an extra parameter:


```
public static void crawl(File f, String indent) {
```
- Often the parameters we need for our recursion do not match those the client will want to pass.

In these cases, we instead write a pair of methods:

 - 1) a public, non-recursive one with the parameters the client wants
 - 2) a private, recursive one with the parameters we really need

35

Exercise solution 2

```
// Prints information about this file,
// and (if it is a directory) any files inside it.
public static void crawl(File f) {
    crawl(f, ""); // call private recursive helper
}

// Recursive helper to implement crawl/indent behavior.
private static void crawl(File f, String indent) {
    System.out.println(indent + f.getName());
    if (f.isDirectory()) {
        // recursive case; print contained files/dirs
        for (File subFile : f.listFiles()) {
            crawl(subFile, indent + "    ");
        }
    }
}
```

36

Recursive Backtracking

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 ("MARTY")`
outputs the following sequence of lines:

```
MARTY MRYPTA ATYMR RTMAY TARMY YMTAR
MARYT MRYPTA ATYRM RTMYA TARYM YMTRA
MARTY MRYPTA ATYRM RTMYA TARYM YMTRA
MAYTR MYPTRA AYMTR RTAYM TAYMR YAMTR
MAYRT AMRTY AYRMT RTYMA TRMAY YARMR
MAYTR AMRTY AYRMT RTYAM TRMYA YARTM
MRAYT AMTRY AYTRM RYMTA TRAYM YATRM
MRTAY AMYRT RMTAY RYAMT TRYMA YRMAT
MRTYA AMYTR RMTAY RYATM TRYAM YRMTA
MRYAT ARMTY RMTAY RYTMA TYMAR YRMT
MRYTA ARMTY RMTAY RYTMA TYMAR YRMT
MRTYA ARTYU RMTAU RYTUA TYMAR YRMT
MTAYR ARTYM RMTYA TMAYR TYARM YMTRAM
MTRAY ARYTM RMTAY TMRAY TYRMA YTMAR
MTRYA ARYTM RMTAY TMRYA TYRAM YTMR
MTYAR ATMRY RMTYM TMYAR YMART YTARM
MTYRA ATMRY RMTYM TMYRA YMTR YMTR
MYART ATRYM RAYMT TAMRY YMTR YMTR
MYATR ATRYM RAYTM TAMYR YMTR YMTR
```

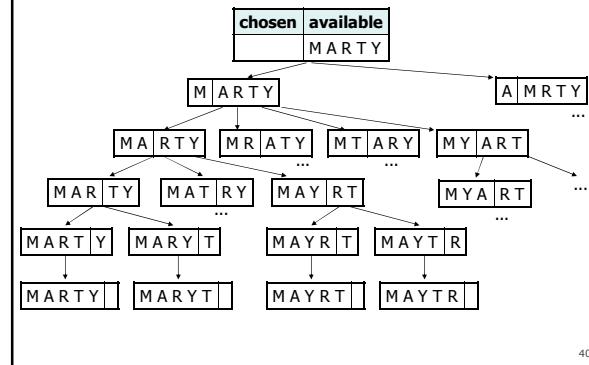
38

Examining the problem

- Think of each permutation as a set of choices or **decisions**:
 - Which character do I want to place first?
 - Which character do I want to place second?
 - ...
- **solution space**: set of all possible sets of decisions to explore
- We want to generate all possible sequences of decisions.
for (each possible first letter):
for (each possible second letter):
for (each possible third letter):
...
print!
- This is called a **depth-first search**

39

Decision trees



40

Backtracking

- **backtracking**: A general algorithm for finding solution(s) to a computational problem by trying partial solutions and then abandoning them ("backtracking") if they are not suitable.
 - a "brute force" algorithmic technique (tries all paths; not clever)
 - often (but not always) implemented recursively

Applications:

- producing all permutations of a set of values
- parsing languages
- games: anagrams, crosswords, word jumbles, 8 queens
- combinatorics and logic programming

41

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** from the set of choices.
– Remove **C** from the set of **choices**.
 - explore the remaining **choices**.
 - Un-make choice **C**.
 - Backtrack!

42

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 the rest, what should I do?
 - How do I "un-make" a choice?

43

Permutations revisited

- 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("MARTY")`
 outputs the following sequence of lines:

MARTY	MYRAT	ATYMR	RTMAY	TARMY	YMTAR
MARYT	MYART	ATYRM	RTMYA	TARYM	YATMR
MATRY	MYTRA	ATYMR	RTMAY	TARMY	YAMTR
MATYR	MYART	AYMTR	RTYMA	TRAMY	YARTM
MAYTR	AMTRY	AYRTM	RTYAM	TRMYA	YARTM
MAYRT	AMTRY	AYTRM	RTYAM	TRMAY	YATRM
MRAYT	AMTYR	AYTRM	RYMTA	TRAYM	YATRM
MRTAY	AMYTR	FRMAYT	RYATM	TRYAM	YRMTA
MRTYA	AMYTR	FRMAYT	RYATM	TRYAM	YRMTA
MRYAT	ARMYT	FRMTAY	RYTM	TYRAM	YRMTA
MRYTA	ARMYT	FRMTA	RYTM	TYRAM	YRMTA
MRTYA	ARTYM	FRTAY	RYTA	TYRAM	YRMTA
MTRAY	ARYTM	FRMTA	TMAYR	TYARM	YTMAR
MTRYA	ARYTM	FRMTY	TMRAY	TYRAM	YTMAR
MTYAR	ATMRY	FRATM	TMYAR	YMAR	YTAMR
MTYRA	ATMRY	FRATM	TMYRA	YMATR	YTARM
MYART	ATRMY	FRAYM	TMARY	YMRAT	YTARM
MYATR	ATRMY	FRAYM	TMAYR	YMTAR	YTARM

44

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

45

Exercise solution 2

```
// 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++) {
            String ch = s.substring(i, i + 1); // choose
            String rest = s.substring(0, i) + s.substring(i + 1); // remove
            rest += ch; // explore
            permute(rest, chosen + ch);
        }
    }
    // (don't need to "un-choose" because
    // we used temp variables)
}
```

46

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

47

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.

48

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);
            }
        }
    }
}

```

49

Exercise: Dominoes

- The game of dominoes is played with small black tiles, each having 2 numbers of dots from 0-6. Players line up tiles to match dots.



- Given a class `Domino` with the following methods:

```

public int first()          // first dots value
public int second()         // second dots value
public String toString()    // e.g. "(3|5)"

```

- Write a method `hasChain` that takes a `List` of dominoes and a starting/ending dot value, and returns whether the dominoes can be made into a chain that starts/ends with those values.

- If the chain's start/end are the same, the answer is always `true`.

50

Domino chains

- Suppose we have the following dominoes:



- We can link them into a chain from 1 to 3 as follows:
– Notice that the 3|5 domino had to be flipped.



- We can "link" one domino into a "chain" from 6 to 2 as follows:



51

Exercise client code

```

import java.util.*; // for ArrayList
public class SolveDominoes {
    public static void main(String[] args) {
        // [(1|4), (2|6), (4|5), (1|5), (3|5)]
        List<Domino> dominoes = new ArrayList<Domino>();
        dominoes.add(new Domino(1, 4));
        dominoes.add(new Domino(2, 6));
        dominoes.add(new Domino(4, 5));
        dominoes.add(new Domino(1, 5));
        dominoes.add(new Domino(3, 5));
        System.out.println(hasChain(dominoes, 5, 5)); // true
        System.out.println(hasChain(dominoes, 1, 5)); // true
        System.out.println(hasChain(dominoes, 1, 3)); // true
        System.out.println(hasChain(dominoes, 1, 6)); // false
        System.out.println(hasChain(dominoes, 1, 2)); // false
    }
    public static boolean hasChain(List<Domino> dominoes,
                                  int start, int end) {
        ...
    }
}

```

52

Exercise solution

```

public static boolean hasChain(List<Domino> dominoes,
                               int start, int end) {
    if (start == end) {
        return true;                      // base case
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i); // choose
            if (d.first() == start) {      // explore
                if (hasChain(dominoes, d.second(), end)) {
                    return true;
                }
            } else if (d.second() == start) {
                if (hasChain(dominoes, d.first(), end)) {
                    return true;
                }
            }
            dominoes.add(i, d);           // un-choose
        }
        return false;
    }
}

```

53

Exercise: Print chain

- Write a variation of your `hasChain` method that also prints the chain of dominoes that it finds, if any.

```

hasChain(dominoes, 1, 3);

[(1|4), (4|5), (5|3)]

```

54

Exercise solution

```

public static boolean hasChain(List<Domino> dominoes, int start, int end) {
    Stack<Domino> chosen = new Stack<Domino>();
    return hasChain(dominoes, chosen, start, end);
}

private static boolean hasChain(List<Domino> dominoes,
                               Stack<Domino> chosen, int start, int end) {
    if (start == end) {
        System.out.println(chosen);
        return true;
    } else {
        for (int i = 0; i < dominoes.size(); i++) {
            Domino d = dominoes.remove(i);
            if (d.first() == start) {
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    return true;
                }
                chosen.pop();
            } else if (d.second() == start) {
                d.flip();
                chosen.push(d);
                if (hasChain(dominoes, chosen, d.second(), end)) {
                    return true;
                }
                chosen.pop();
            }
            dominoes.add(i, d);
        }
    }
    return false;
}

```

55

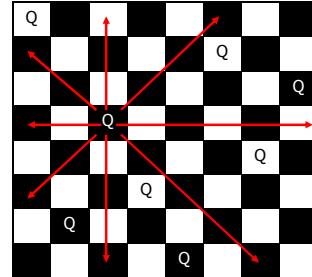
The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.

– What are the "choices"?

– How do we "make" or "un-make" a choice?

– How do we know when to stop?



56

Naive algorithm

- for (each square on board):

- Place a queen there.

	1	2	3	4	5	6	7	8
1	Q
2
3	...							
4								
5								
6								
7								
8								

57

Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.

	1	2	3	4	5	6	7	8
1	Q					
2						
3		Q	...					
4			...					
5				Q	...			
6					...			
7						Q	...	
8							...	

58

Exercise

- Suppose we have a Board class with the following methods:

Method/Constructor	Description
public Board(int size)	construct empty board
public boolean isSafe(int row, int column)	true if queen can be safely placed here
public void place(int row, int column)	place queen here
public void remove(int row, int column)	remove queen from here
public String toString()	text display of board

59

Exercise solution

```

// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (!explore(board, 1)) {
        System.out.println("No solution found.");
    } else {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    }
}
...

```

60

Exercise solution, cont'd.

```
// Recursively searches for a solution to 8 queens on this
// board, starting with the given column, returning true if a
// solution is found and storing that solution in the board.
// PRE: queens have been safely placed in columns 1 to (col-1)
public static boolean explore(Board board, int col) {
    if (col > board.size()) {
        return true; // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 1; row <= board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col); // choose
                if (explore(board, col + 1)) { // explore
                    return true; // solution found
                }
                b.remove(row, col); // un-choose
            }
        }
        return false; // no solution found
    }
}
```

61