3.1 Recursion: Introduction

An ***algorithm*** is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

Figure 3.1.1: Algorithms are like recipes.

Make lemonade:

- Add sugar to pitcher
- Add lemon juice
- Add water
- Stir

Some problems can be solved using a recursive algorithm. A ***recursive algorithm*** solves a problem by breaking that problem into smaller subproblems, solving these subproblems, and combining the solutions.

Figure 3.1.2: Mowing the lawn can be broken down into a recursive process.

- Mow the lawn
  - Mow the frontyard
    - Mow the left front
    - Mow the right front
  - Mow the backyard
    - Mow the left back
    - Mow the right back

An algorithm that is defined by repeated applications of the same algorithm on smaller problems is a ***recursive*** algorithm. The mowing algorithm consists of applying the mowing algorithm on smaller pieces of the yard.

At some point, a recursive algorithm must describe how to actually do something, known as the ***base case***. The mowing algorithm could thus be written as:

- *Mow* the lawn
  - *If lawn is less than 100 square meters*
    - Push the lawnmower left-to-right in adjacent rows
  - *Else*
    - Mow one half of the lawn
    - Mow the other half of the lawn

PARTICIPATION ACTIVITY 3.1.1: Recursion.

Which are recursive definitions/algorithms?

1) Helping N people:

   If N is 1, help that person.
   Else, help the first N/2 people, then help the second N/2 people.

   - True
   - False

2) Driving to the store:

   Go 1 mile.
   Turn left on Main Street.
   Go 1/2 mile.

   - True
   - False

3) Sorting envelopes by zipcode:

   If N is 1, done.
   Else, find the middle zipcode. Put all zipcodes less than the middle zipcode on the left, all greater ones on the right. Then sort the left, then sort the right.

   - True
   - False

3.2 Recursive methods

A method may call other methods, including calling itself. A method that calls itself is a ***recursive method***.

PARTICIPATION ACTIVITY 3.2.1: A recursive method example.

Animation captions:

1. The first call to countDown( ) method comes from main. Each call to countDown( ) effectively creates a new "copy" of the executing method, as shown on the right.
2. Then, the countDown( ) function calls itself. countDown(1) similarly creates a new "copy" of the executing method.
3. countDown( ) method calls itself once more.
4. That last instance does not call countDown( ) again, but instead returns. As each instance returns, that copy is deleted.

Each call to countDown() effectively creates a new "copy" of the executing method, as shown on the right. Returning deletes that copy.

The example is for demonstrating recursion; counting down is otherwise better implemented with a loop.

Recursion may be direct, such as f() itself calling f(), or indirect, such as f() calling g() and g() calling f().

PARTICIPATION ACTIVITY 3.2.2: Thinking about recursion.

Refer to the above countDown example for the following.

1) How many times is countDown() called if main() calls CountDown(5)?

Check Show answer

2) How many times is countDown() called if main() calls CountDown(0)?

Check Show answer

3) Is there a difference in how we define the parameters of a recursive versus non-recursive method? Answer yes or no.

Check Show answer

CHALLENGE ACTIVITY 3.2.1: Calling a recursive method.

Write a statement that calls the recursive method backwardsAlphabet() with parameter startingLetter.

```
1  import java.util.Scanner;
2
3  public class RecursiveCalls {
4     public static void backwardsAlphabet(char currLetter) {
5        if (currLetter == 'a') {
6           System.out.println(currLetter);
7        }
8        else {
9           System.out.print(currLetter + " ");
10          backwardsAlphabet((char)(currLetter - 1));
11       }
12    }
13
14    public static void main (String [] args) {
15       Scanner scnr = new Scanner(System.in);
16       char startingLetter;
17
18       startingLetter = scnr.next().charAt(0);
```

Run

3.3 Recursive definitions

Recursive algorithms

An ***algorithm*** is a sequence of steps, including at least 1 terminating step, for solving a problem. A ***recursive algorithm*** is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.

Because a problem cannot be endlessly divided into smaller subproblems, a recursive algorithm must have a ***base case***: A case where a recursive algorithm completes without applying itself to a smaller subproblem.

PARTICIPATION ACTIVITY 3.3.1: Recursive factorial algorithm for positive numbers.

Animation captions:

1. A recursive algorithm to compute N factorial has 2 parts: the base case and the non-base case.
2. N is assumed to be a positive integer. N = 1 is the base case, wherein a result of 1 is returned.
3. The non-base case computes the result by multiplying N by (N - 1) factorial.
4. The algorithm applying itself to a smaller subproblem is what makes the algorithm recursive.

PARTICIPATION ACTIVITY 3.3.2: Recursive algorithms.

1) A recursive algorithm applies itself to a smaller subproblem in all cases.
   - ○ True
   - ○ False

2) The base case is what ensures that a recursive algorithm eventually terminates.
   - ○ True
   - ○ False

3) The presence of a base case is what identifies an algorithm as being recursive.
   - ○ True
   - ○ False

Recursive functions

A ***recursive function*** is a function that calls itself. Recursive functions are commonly used to implement recursive algorithms.

Table 3.3.1: Sample recursive functions: Factorial, CumulativeSum, and ReverseList.


```
Factorial(N) {
   if (N == 1)
      return 1
   else
      return N * Factorial(N - 1)
}
```

```
CumulativeSum(N) {
   if (N == 0)
      return 0
   else
      return N + CumulativeSum(N - 1)
}
```

```
ReverseList(list, startIndex, endIndex) {
   if (startIndex >= endIndex)
      return
   else {
      Swap elements at startIndex and endIndex
      ReverseList(list, startIndex + 1, endIndex - 1)
   }
}
```

PARTICIPATION ACTIVITY 3.3.3: CumulativeSum recursive function.

1) What is the condition for the base case in the CumulativeSum function?
   - N equals 0
   - N does not equal 0

2) If Factorial(6) is called, how many additional calls are made to Factorial to compute the result of 720?
   - 7
   - 5
   - 3

3) Suppose ReverseList is called on a list of size 3, a start index of 0, and an out-of-bounds end index of 3. The

base case ensures that the function still properly sorts the list.

- ○ True
- ○ False

3.4 Creating a recursive method

Creating a recursive method can be accomplished in two steps.

- **Write the base case** -- Every recursive method must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may write that part of the method first, and then test. There may be multiple base cases.
- **Write the recursive case** -- The programmer then adds the recursive case to the method.

The following illustrates for a simple method that computes the factorial of N (i.e. N!). The base case is N = 1 or 1! which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for N > 1, and written as `else { fact = N * NFact( N – 1 ); }`.

PARTICIPATION ACTIVITY 3.4.1: Writing a recursive method for factorial: First write the base case, then add the recursive case.

Animation captions:

1. The base case, which returns a value without performing a recursive call, is written and tested first. If N is less than or equal to 1, then the NFact() method returns 1.
2. Next the recursive case, which calls itself, is written and tested. If N is greater than 1, then the NFact() method returns N * NFact(N - 1).

A common error is to not cover all possible base cases in a recursive method. Another common error is to write a recursive method that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Typically, programmers will use two methods for recursion. An "outer" method is intended to be called from other parts of the program, like the method `int calcFactorial(int inVal)`. An "inner" method is intended only to be called from that outer method, for example a method `int calcFactorialHelper(int inVal)`. The outer method may check for a valid input value, e.g., ensuring inVal is not negative, and then calling the inner method. Commonly, the inner

method has parameters that are mainly of use as part of the recursion, and need not be part of the outer method, thus keeping the outer method more intuitive.

PARTICIPATION ACTIVITY 3.4.2: Creating recursion.

1) Recursive methods can be accomplished in one step, namely repeated calls to itself.
   - True
   - False

2) A recursive method with parameter N counts up from any negative number to 0. An appropriate base case would be `N == 0`.
   - True
   - False

3) A recursive method can have two base cases, such as `N == 0` returning 0, and `N == 1` returning 1.
   - True
   - False

Before writing a recursive method, a programmer should determine:

1. Does the problem naturally have a recursive solution?
2. Is a recursive solution better than a non-recursive solution?

For example, computing N! (N factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution. The figure below illustrates how the factorial computation can be implemented as a loop. Conversely, binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

Figure 3.4.1: Non-recursive solution to compute N!

```
for (i = inputNum; i > 1; --i) {
   facResult = facResult * i;
}
```

PARTICIPATION ACTIVITY 3.4.3: When recursion is appropriate.

1) N factorial (N!) is commonly implemented as a recursive method due to being easier to understand and executing faster than a loop implementation.

- ○ True
- ○ False

zyDE 3.4.1: Output statements in a recursive function.

Implement a recursive method to determine if a number is prime. Skeletal code is the isPrime method.

Load default template... **Run**

```
1
2  public class PrimeChecker {
3  // Returns 0 if value is not prime, 1 if
4     public static int isPrime(int testVal,
5        // Base case 1: 0 and 1 are not pri
6
7        // Base case 2: testVal only divisi
8
9        // Recursive Case
10          // Check if testVal can be evenl
11          // Hint: use the % operator
12
13          // If not, recursive call to isP
14
15    }
16
17    public static void main(String[] args)
18       int primeCheckVal;     // Value che
```

CHALLENGE ACTIVITY 3.4.1: Recursive method: Writing the base case.

Write code to complete doublePennies()'s base case. Sample output for below program with inputs 1 and 10:

```
Number of pennies after 10 days: 1024
```

Note: If the submitted code has an infinite loop, the system will stop running the code

after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

```
1  import java.util.Scanner;
2
3  public class CalculatePennies {
4  // Returns number of pennies if pennies are doubled numDays times
5     public static long doublePennies(long numPennies, int numDays) {
6        long totalPennies;
7
8        /* Your solution goes here  */
9
10       else {
11          totalPennies = doublePennies((numPennies * 2), numDays - 1);
12       }
13
14       return totalPennies;
15    }
16
17 // Program computes pennies if you have 1 penny today,
18 // 2 pennies after one day, 4 after two days, and so on
```


Run

CHALLENGE ACTIVITY 3.4.2: Recursive method: Writing the recursive case.

Write code to complete printFactorial()'s recursive case. Sample output if input is 5:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

```
1  import java.util.Scanner;
2
3  public class RecursivelyPrintFactorial {
4     public static void printFactorial(int factCounter, int factValue) {
5        int nextCounter;
6        int nextValue;
7
8        if (factCounter == 0) { // Base case: 0! = 1
9           System.out.println("1");
10       }
11       else if (factCounter == 1) { // Base case: Print 1 and result
12          System.out.println(factCounter + " = " + factValue);
13       }
14       else { // Recursive case
15          System.out.print(factCounter + " * ");
16          nextCounter = factCounter - 1;
17          nextValue = nextCounter * factValue;
18
```

Run

3.5 Recursive algorithms

Fibonacci numbers

The ***Fibonacci sequence*** is a numerical sequence where each term is the sum of the previous 2 terms in the sequence, except the first 2 terms, which are 0 and 1. A recursive function can be used to calculate a ***Fibonacci number***: A term in the Fibonacci sequence.

Figure 3.5.1: FibonacciNumber recursive function.

```
FibonacciNumber(termIndex) {
   if (termIndex == 0)
      return 0
   else if (termIndex == 1)
      return 1
   else
      return FibonacciNumber(termIndex - 1) + FibonacciNumber(termIndex - 2)
}
```

PARTICIPATION ACTIVITY 3.5.1: FibonacciNumber recursive function.

1) What does FibonacciNumber(2) return?

Check Show answer

2) What does FibonacciNumber(4) return?

Check Show answer

3) What does FibonacciNumber(8)

return?

Check Show answer

Recursive binary search

Binary search is an algorithm that searches a sorted list for a key by first comparing the key to the middle element in the list and recursively searching half of the remaining list so long as the key is not found.

Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the index. If the search key is not found, the algorithm recursively searches the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

Figure 3.5.2: BinarySearch recursive algorithm.

```
BinarySearch(numbers, low, high, key) {
   if (low > high)
      return -1

   mid = (low + high) / 2
   if (numbers[mid] < key) {
      return BinarySearch(numbers, mid + 1, high, key)
   }
   else if (numbers[mid] > key) {
      return BinarySearch(numbers, low, mid - 1, key)
   }
   return mid
}
```

PARTICIPATION ACTIVITY 3.5.2: Recursive binary search.

Suppose BinarySearch(numbers, 0, 6, 42) is used to search the list (14, 26, 42, 59, 71, 88, 92) for key 42.

1) What is the first middle element that is compared against 42?

- 42
- 59
- 71

2) What will the low and high argument values be for the first recursive call?

- low = 0
  high = 2
- low = 0
  high = 3
- low = 4
  high = 6

3) How many calls to BinarySearch will be made by the time 42 is found?

- 2
- 3
- 4

PARTICIPATION ACTIVITY 3.5.3: Recursive binary search base case.

1) Which does not describe a base case for BinarySearch?

- The low argument is greater than the high argument.
- The list element at index `mid` equals the key.
- The list element at index `mid` is less than the key.

3.6 Analyzing the time complexity of recursive algorithms

Recurrence relations

The runtime complexity T(N) of a recursive function will have function T on both sides of the equation. Ex: Binary search performs constant time operations, then a recursive call that

operates on half of the input, making the runtime complexity T(N) = O(1) + T(N / 2). Such a function is known as a ***recurrence relation***: A function f(N) that is defined in terms of the same function operating on a value < N.

Using O-notation to express runtime complexity of a recursive function requires solving the recurrence relation. For simpler recursive functions such as binary search, runtime complexity can be determined by expressing the number of function calls as a function of N.

PARTICIPATION ACTIVITY 3.6.1: Worst case binary search runtime complexity.

Animation captions:

1. In the non-base case, BinarySearch does some O(1) operations plus a recursive call on half the input list.
2. The maximum number of recursive calls can be computed for any known input size. For size 1, 1 recursive call is made.
3. Additional entries in the table can be filled. A list of size 32 is split in half 6 times before encountering the base case.
4. By analyzing the pattern, the total number of function calls can be expressed as a function of N.
5. The number of function calls corresponds to the runtime complexity.

PARTICIPATION ACTIVITY 3.6.2: Binary search and recurrence relations.

1) When the low and high arguments are equal, BinarySearch does not make a recursive call.
   - ○ True
   - ○ False

2) Suppose BinarySearch is used to search for a key within a list with 64 numbers. If the key is not found, how many recursive calls to BinarySearch are made?
   - ○ 1
   - ○ 6
   - ○ 64

3) Which function is a recurrence relation?

- $T(N) = N^2 + 6N + 2$
- $T(N) = 6N + T(N/4)$
- $T(N) = log_2 N$

Recursion trees

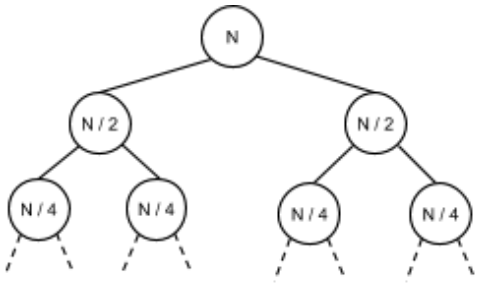
The runtime complexity of any recursive function can be split into 2 parts: operations done directly by the function and operations done by recursive calls made by the function. Ex: For binary search's T(N) = O(1) + T(N / 2), O(1) represents operations directly done by the function and T(N / 2) represents operation done by a recursive call. A useful tool for solving recurrences is a ***recursion tree***: A visual diagram of an operation done by a recursive function, that separates operations done directly by the function and operations done by recursive calls.

PARTICIPATION ACTIVITY 3.6.3: Recursion trees.

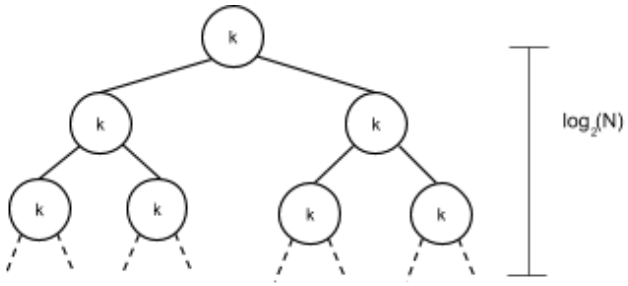
Animation captions:

1. An algorithm like binary search does a constant number of operations, k, followed by a recursive call on half the list.
2. The root node in the recursion tree represents k operations inside the first function call.
3. Recursive operations are represented below the node. The first recursive call also does k operations.
4. The tree's height corresponds to the number of recursive calls. Splitting the input in half each time results in $log_2 N$ recursive calls. O($log_2 N$) = O($log N$).
5. Another algorithm may perform N operations then 2 recursive calls, each on N / 2 items. The root node represents N operations.
6. The initial call makes 2 recursive calls, each of which has a local N value of the initial N value / 2.
7. N operations are done per level.
8. The tree has O($log_2 N$) levels. O($N * log_2 N$) = O($N log N$) operations are done in total.

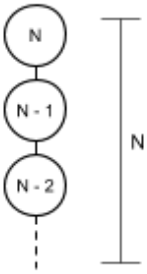
PARTICIPATION ACTIVITY 3.6.4: Matching recursion trees with runtime complexities.



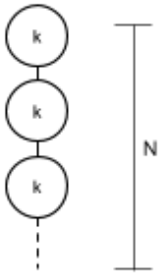
Tree 1



Tree 2



Tree 3



Tree 4

Tree 4 | Tree 1 | Tree 2 | Tree 3

	T(N) = k + T(N / 2) + T(N / 2)
	T(N) = k + T(N - 1)
	T(N) = N + T(N - 1)
	T(N) = N + T(N / 2) + T(N / 2)

Reset

PARTICIPATION ACTIVITY 3.6.5: Recursion trees.

Suppose a recursive function's runtime is $T(N) = 7 + T(N - 1)$.

1) How many levels will the recursion tree have?

- 7
- $log_2 N$
- N

2) What is the runtime complexity of the function using O notation?

- O(1)
- O($logN$)
- O(N)

PARTICIPATION ACTIVITY 3.6.6: Recursion trees.

Suppose a recursive function's runtime is $T(N) = N + T(N - 1)$.

1) How many levels will the recursion tree have?

- $log_2 N$
- N
- N^2

2) The runtime can be expressed by the series N + (N - 1) + (N - 2) + ... + 3 + 2 + 1. Which expression is mathematically equivalent?

- $N * log_2 N$
- $(N/2) * (N + 1)$

3) What is the runtime complexity of the function using O notation?

- O(N)
- O(N^2)

3.7 Recursive algorithm: Search

Recursive search (general)

Consider a guessing game program where a friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.

What algorithm would you use to minimize the number of guesses?

A first try might implement an algorithm that simply guesses in increments of 1:

- Is it 0? Higher
- Is it 1? Higher
- Is it 2? Higher

This algorithm requires too many guesses (50 on average). A second try might implement an algorithm that guesses by 10s and then by 1s:

- Is it 10? Higher
- Is it 20? Higher
- Is it 30? Lower
- Is it 21? Higher
- Is it 22? Higher
- Is it 23? Higher

This algorithm does better but still requires about 10 guesses on average: 5 to find the correct tens digit and 5 to guess the correct ones digit. An even better algorithm uses a binary search. A ***binary search*** algorithm begins at the midpoint of the range and halves the range after each guess. For example:

- Is it 50 (the middle of 0-100)? Lower
- Is it 25 (the middle of 0-50)? Higher
- Is it 37 (the middle of 25-50)? Lower
- Is it 31 (the middle of 25-37).

After each guess, the binary search algorithm is applied again, but on a smaller range, i.e., the algorithm is recursive.

PARTICIPATION ACTIVITY 3.7.1: Binary search: A well-known recursive algorithm.

Animation captions:

1. A friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.
2. Using a binary search algorithm, you begin at the midpoint of the lower range. (highVal + lowVal) / 2 = (100 + 0) / 2, or 50.
3. The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
4. After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.
5. A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high

sides of the guessing range.

Recursive search method

A recursive method is a natural match for the recursive binary search algorithm. A method guessNumber(lowVal, highVal, scnr) has parameters that indicate the low and high sides of the guessing range and a Scanner object for getting user input. The method guesses at the midpoint of the range. If the user says lower, the method calls guessNumber(lowVal, midVal, scnr). If the user says higher, the method calls guessNumber(midVal + 1, highVal, scnr)

The recursive method has an if-else statement. The if branch ends the recursion, known as the ***base case***. The else branch has recursive calls. Such an if-else pattern is common in recursive methods.

Figure 3.7.1: A recursive method carrying out a binary search algorithm.

```
import java.util.Scanner;

public class NumberGuessGame {
   public static void guessNumber(int lowVal, int highVal, Scanner scnr) {
      int midVal;            // Midpoint of low..high
      char userAnswer;       // User response

      midVal = (highVal + lowVal) / 2;

      // Prompt user for input
      System.out.print("Is it " + midVal + "? (l/h/y): ");
      userAnswer = scnr.next().charAt(0);

      if ((userAnswer != 'l') && (userAnswer != 'h')) { // Base case: found number
         System.out.println("Thank you!");
      }
      else {                                           // Recursive case: split
into lower OR upper half
         if (userAnswer == 'l') {                      // Guess in lower half
            guessNumber(lowVal, midVal, scnr);               // Recursive call
         }
         else {                                        // Guess in upper half
            guessNumber(midVal + 1, highVal, scnr);          // Recursive call
         }
      }
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);

      // Print game objective, user input commands
      System.out.println("Choose a number from 0 to 100.");
      System.out.println("Answer with:");
      System.out.println("   l (your num is lower)");
      System.out.println("   h (your num is higher)");
      System.out.println("   any other key (guess is right).");

      // Call recursive function to guess number
      guessNumber(0, 100, scnr);
   }
}
```

```
}
```

```
Choose a number from 0 to 100.
Answer with:
   l (your num is lower)
   h (your num is higher)
   any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): l
Is it 32? (l/h/y): l
Is it 29? (l/h/y): h
Is it 31? (l/h/y): y
Thank you!
```


Calculating the middle value

Because midVal has already been checked, it need not be part of the new window, so midVal + 1 rather than midVal is used for the window's new low side, or midVal - 1 for the window's new high side. But the midVal - 1 can have the drawback of a non-intuitive base case (i.e., midVal < lowVal, because if the current window is say 4..5, midVal is 4, so the new window would be 4..4-1, or 4..3). rangeSize == 1 is likely more intuitive, and thus the algorithm uses midVal rather than midVal - 1. However, the algorithm uses midVal + 1 when searching higher, due to integer rounding. In particular, for window 99..100, midVal is 99 ((99 + 100) / 2 = 99.5, rounded to 99 due to truncation of the fraction in integer division). So the next window would again be 99..100, and the algorithm would repeat with this window forever. midVal + 1 prevents the problem, and doesn't miss any numbers because midVal was checked and thus need not be part of the window.

PARTICIPATION ACTIVITY 3.7.2: Binary search tree tool.

The following program guesses the hidden number known by the user. Assume the hidden number is 63.

```
import java.util.Scanner;

public class NumberGuessGame {
   public static void guessNumber(int lowVal, int highVal, Scanner scnr) {
      int midVal;            // Midpoint of low..high
      char userAnswer;       // User response

      midVal = (highVal + lowVal) / 2;

      System.out.print("Is it " + midVal + "? (l/h/y): ");
      userAnswer = scnr.next().charAt(0);

      if ((userAnswer != 'l') && (userAnswer != 'h')) { // Base case:
         System.out.println("Thank you!");              // Found number
      }
      else { // Recursive case: split into lower OR upper half
         if (userAnswer == 'l') {                    // Guess in lower half
            guessNumber(lowVal, midVal, scnr);       // Recursive call
         }
         else {                                      // Guess in upper half
            guessNumber(midVal + 1, highVal, scnr); // Recursive call
         }
      }
      return;
   }

   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      System.out.println("Choose a number from 0 to 100.");
      System.out.println("Answer with:");
      System.out.println("   l (your num is lower)");
      System.out.println("   h (your num is higher)");
      System.out.println("   any other key (guess is right).");

      guessNumber(0, 100, scnr);

      return;
   }
}
```


→ main()

```
public static void main (String[] args) {
   Scanner scnr = new Scanner(System.in);
   System.out.println("Choose a number from 0 to 100.");
   System.out.println("Answer with:");
   System.out.println("   l (your num is lower)");
   System.out.println("   h (your num is higher)");
   System.out.println("   any other key (guess is right).");

   guessNumber(0, 100, scnr);

   return;
}
```

Recursively searching a sorted list

Search is commonly performed to quickly find an item in a sorted list stored in an array or ArrayList. Consider a list of attendees at a conference, whose names have been stored in alphabetical order in an array or ArrayList. The following quickly determines whether a particular person is in attendance.

findMatch() restricts its search to elements within the range lowVal to highVal. main() initially passes a range of the entire list: 0 to (list size - 1). findMatch() compares to the middle element, returning that element's position if matching. If not matching, findMatch() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found. If neither of those two base cases are satisfied, then findMatch() recursively searches either the lower or upper half of the range as appropriate.

Figure 3.7.2: Recursively searching a sorted list.

```
import java.util.Scanner;
import java.util.ArrayList;

public class NameFinder {
   /* Finds index of string in vector of strings, else -1.
      Searches only with index range low to high
      Note: Upper/lower case characters matter
   */
   public static int findMatch(ArrayList<String> stringList, String itemMatch,
                               int lowVal, int highVal) {
      int midVal;         // Midpoint of low and high values
      int itemPos;        // Position where item found, -1 if not found
      int rangeSize;      // Remaining range of values to search for match

      rangeSize = (highVal - lowVal) + 1;
      midVal = (highVal + lowVal) / 2;

      if (itemMatch.equals(stringList.get(midVal))) {          // Base case 1:
item found at midVal position
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                               // Base case 2:
match not found
         itemPos = -1;
      }
      else {                                                   // Recursive case:
search lower or upper half
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower
half, recursive call
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal);
         }
         else {                                                // Search upper
```

```
half, recursive call
            itemPos = findMatch(stringList, itemMatch, midVal + 1, highVal);
         }
      }

      return itemPos;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<String> attendeesList = new ArrayList<String>(); // List of
attendees
      String attendeeName;                                       // Name of
attendee to match
      int matchPos;                                              // Matched
position in attendee list

      // Omitting part of program that adds attendees
      // Instead, we insert some sample attendees in sorted order
      attendeesList.add("Adams, Mary");
      attendeesList.add("Carver, Michael");
      attendeesList.add("Domer, Hugo");
      attendeesList.add("Fredericks, Carlos");
      attendeesList.add("Li, Jie");

      // Prompt user to enter a name to find
      System.out.print("Enter person's name: Last, First: ");
      attendeeName = scnr.nextLine(); // Use nextLine() to allow space in name

      // Call function to match name, output results
      matchPos = findMatch(attendeesList, attendeeName, 0, attendeesList.size() -
1);
      if (matchPos >= 0) {
         System.out.println("Found at position " + matchPos + ".");
      }
      else {
         System.out.println("Not found.");
      }
   }
}
```

```
Enter person's name: Last, First: Meeks, Stan
Not found.

...

Enter person's name: Last, First: Adams, Mary
Found at position 0.

...

Enter person's name: Last, First: Li, Jie
Found at position 4.
```


PARTICIPATION ACTIVITY 3.7.3: Recursive search algorithm.

Consider the above findMatch() method for finding an item in a sorted list.

1) If a sorted list has elements 0 to 50 and the item being searched for is at

element 6, how many times will findMatch() be called?

Check Show answer

2) If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many calls to findMatch() will be made during the failed search for "Apples"?

Check Show answer

PARTICIPATION ACTIVITY 3.7.4: Recursive calls.

A list has 5 elements numbered 0 to 4, with these letter values: 0: A, 1: B, 2: D, 3: E, 4: F.

1) To search for item C, the first call is findMatch(0, 4). What is the second call to findMatch()?
   - findMatch(0, 0)
   - findMatch(0, 2)
   - findMatch(3, 4)

2) In searching for item C, findMatch(0, 2) is called. What happens next?
   - Base case 1: item found at midVal.
   - Base case 2: rangeSize == 1, so no match.
   - Recursive call: findMatch(2, 2)

Exploring further:

- Binary search from GeeksforGeeks.org

3.8 Adding output statements for debugging

Recursive methods can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a findMatch() method that searches a sorted list for an item. All of findMatch()'s print statements start with `System.out.print(indentAmt + ...);`. Indent is typically some number of spaces. main() sets indent to three spaces. Each recursive call *adds* three more spaces. Note how the output now clearly shows the recursion depth.

Figure 3.8.1: Output statements can help debug recursive methods, especially if indented based on recursion depth.

```
import java.util.Scanner;
import java.util.ArrayList;

public class NameFinder {
   /* Finds index of string in vector of strings, else -1.
      Searches only with index range low to high
      Note: Upper/lower case characters matter
   */
   public static int findMatch(ArrayList<String> stringList, String itemMatch,
                               int lowVal, int highVal, String indentAmt) { //
indentAmt used for print debug
      int midVal;        // Midpoint of low and high values
      int itemPos;       // Position where item found, -1 if not found
      int rangeSize;     // Remaining range of values to search for match

      System.out.println(indentAmt + "Find() range " + lowVal + " " + highVal);
      rangeSize = (highVal - lowVal) + 1;
      midVal = (highVal + lowVal) / 2;

      if (itemMatch.equals(stringList.get(midVal))) {           // Base case 1:
item found at midVal position
         System.out.println(indentAmt + "Found person.");
         itemPos = midVal;
      }
      else if (rangeSize == 1) {                                // Base case 2:
match not found
         System.out.println(indentAmt + "Person not found.");
         itemPos = -1;
      }
      else {                                                    // Recursive case:
search lower or upper half
         if (itemMatch.compareTo(stringList.get(midVal)) < 0) { // Search lower
half, recursive call
            System.out.println(indentAmt + "Searching lower half.");
            itemPos = findMatch(stringList, itemMatch, lowVal, midVal, indentAmt +
"   ");
         }
         else {                                                 // Search upper
half, recursive call
            System.out.println(indentAmt + "Searching upper half.");
```

```
            itemPos = findMatch(stringList, itemMatch, midVal + 1, highVal,
indentAmt + "    ");
         }
      }

      System.out.println(indentAmt + "Returning pos = " + itemPos + ".");
      return itemPos;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      ArrayList<String> attendeesList = new ArrayList<String>(); // List of
attendees
      String attendeeName;                                       // Name of
attendee to match
      int matchPos;                                              // Matched
position in attendee list

      // Omitting part of program that adds attendees
      // Instead, we insert some sample attendees in sorted order
      attendeesList.add("Adams, Mary");
      attendeesList.add("Carver, Michael");
      attendeesList.add("Domer, Hugo");
      attendeesList.add("Fredericks, Carlos");
      attendeesList.add("Li, Jie");

      // Prompt user to enter a name to find
      System.out.print("Enter person's name: Last, First: ");
      attendeeName = scnr.nextLine(); // Use nextLine() to allow space in name

      // Call function to match name, output results
      matchPos = findMatch(attendeesList, attendeeName, 0, attendeesList.size() -
1, "    ");
      if (matchPos >= 0) {
         System.out.println("Found at position " + matchPos + ".");
      }
      else {
         System.out.println("Not found.");
      }
   }
}
```

```
Enter person's name: Last, First: Meeks, Stan
   Find() range 0 4
   Searching upper half.
      Find() range 3 4
      Searching upper half.
         Find() range 4 4
         Person not found.
         Returning pos = -1.
      Returning pos = -1.
   Returning pos = -1.
Not found.

...

Enter person's name: Last, First: Adams, Mary
   Find() range 0 4
   Searching lower half.
      Find() range 0 2
      Searching lower half.
         Find() range 0 1
         Found person.
         Returning pos = 0.
      Returning pos = 0.
   Returning pos = 0.
Found at position 0.
```

Some programmers like to leave the output statements in the code, commenting them out with "//" when not in use. The statements actually serve as a form of comment as well.

PARTICIPATION ACTIVITY 3.8.1: Recursive debug statements.

Refer to the above code using indented output statements.

1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.
   - True
   - False

2) Each recursive call should add a few spaces to the indent parameter.
   - True
   - False

3) The method should remove a few spaces from the indent parameter before returning.
   - True
   - False

zyDE 3.8.1: Output statements in a recursive function.

- Run the recursive program, and observe the output statements for debuggin the person is correctly not found.
- Introduce an error by changing `itemPos = -1` to `itemPos = 0` in the ran base case.
- Run the program, notice how the indented print statements help isolate the e person incorrectly being found.

Load default template... Run

```
1
2 import java.util.Scanner;
3 import java.util.ArrayList;
4
5 public class NameFinder {
6    /* Finds index of string in vector of
7       Searches only with index range low
```

```
8       Note: Upper/lower case characters m
9     */
10    public static int findMatch(ArrayList<
11                                    int lowVal
12      int midVal;         // Midpoint of l
13      int itemPos;        // Position wher
14      int rangeSize;      // Remaining ran
15
16      System.out.println(indentAmt + "Fin
17      rangeSize = (highVal - lowVal) + 1;
18      midVal = (highVal + lowVal) / 2;
```


3.9 Recursive exploration of all possibilities

Recursion is a powerful technique for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples.

Word scramble

Consider printing all possible combinations (or "scramblings") of a word's letters. The letters of abc can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be listed by making three choices: Choose the first letter (a, b, or c), then choose the second letter, then choose the third letter. The choices can be depicted as a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right.

PARTICIPATION ACTIVITY 3.9.1: Exploring all possibilities viewed as a tree of choices.

Animation captions:

1. Consider printing all possible combinations of a word's letters. Those possibilities can be listed by choosing the first letter, then the second letter, then the third letter.
2. The choices can be depicted as a tree. Each level represents a choice.
3. A recursive exploration function is a natural match to print all possible combinations of a string's letters. Each call to the function chooses from the set of unchosen letters, continuing until no unchosen letters remain.

The tree guides creation of a recursive exploration method to print all possible combinations of a string's letters. The method takes two parameters: unchosen letters, and already chosen letters. The base case is no unchosen letters, causing printing of the chosen letters. The recursive case calls the method once for each letter in the unchosen letters. The above animation depicts how

the recursive algorithm traverses the tree. The tree's leaves (the bottom nodes) are the base cases.

The following program prints all possible ordering of the letters of a user-entered word.

Figure 3.9.1: Scramble a word's letters in every possible way.


```
import java.util.Scanner;

public class WordScrambler {
   /* Output every possible combination of a word.
      Each recursive call moves a letter from
      remainLetters" to scramLetters".
   */
   public static void scrambleLetters(String remainLetters,  // Remaining letters
                                      String scramLetters) { // Scrambled letters
      String tmpString;      // Temp word combinations
      int i;                 // Loop index

      if (remainLetters.length() == 0) { // Base case: All letters used
         System.out.println(scramLetters);
      }
      else {                             // Recursive case: move a letter from
                                         // remaining to scrambled letters
         for (i = 0; i < remainLetters.length(); ++i) {
            // Move letter to scrambled letters
            tmpString = remainLetters.substring(i, i + 1);
            remainLetters = RemoveFromIndex(remainLetters, i);
            scramLetters = scramLetters + tmpString;

            scrambleLetters(remainLetters, scramLetters);

            // Put letter back in remaining letters
            remainLetters = InsertAtIndex(remainLetters, tmpString, i);
            scramLetters = RemoveFromIndex(scramLetters, scramLetters.length() -
1);
         }
      }
   }

   // Returns a new String without the character at location remLoc
   public static String RemoveFromIndex(String origStr, int remLoc) {
      String finalStr;      // Temp string to extract char

      finalStr = origStr.substring(0, remLoc);                    // Copy before
location remLoc
      finalStr += origStr.substring(remLoc + 1, origStr.length()); // Copy after
location remLoc

      return finalStr;
   }

   // Returns a new String with the character specified by insertStr
   // inserted at location addLoc
   public static String InsertAtIndex(String origStr, String insertStr, int addLoc)
{
      String finalStr;      // Temp string to extract char

      finalStr = origStr.substring(0, addLoc);                  // Copy before
location addLoc
      finalStr += insertStr;                                    // Copy character to
location addLoc
      finalStr += origStr.substring(addLoc, origStr.length()); // Copy after
location addLoc
```

```
      return finalStr;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      String wordScramble;      // User defined word to scramble

      // Prompt user for input
      System.out.print("Enter a word to be scrambled: ");
      wordScramble = scnr.next();

      // Call recursive method
      scrambleLetters(wordScramble, "");
   }
}
```



```
Enter a word to be scrambled: cat
cat
cta
act
atc
tca
tac
```

PARTICIPATION ACTIVITY 3.9.2: Letter scramble.

1) What is the output of scrambleLetters("xy", "")? Determine your answer by manually tracing the code, not by running the program.

- yx xy
- xx yy xy yx
- xy yx

Shopping spree

Recursion can find all possible subsets of a set of items. Consider a shopping spree in which a person can select any 3-item subset from a larger set of items. The following program prints all possible 3-item subsets of a given larger set. The program also prints the total price of each subset.

shoppingBagCombinations() has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items, which prints the items. The recursive case moves one of the remaining items to the bag, recursively calling the method, then moving the item back from the bag to the remaining items.

Figure 3.9.2: Shopping spree in which a user can fit 3 items in a shopping bag.

GroceryItem.java:

```
public class GroceryItem {
   public String itemName;  // Name of item
   public int priceDollars; // Price of item
}
```

```
Milk  Belt  Toys  = $45
Milk  Belt  Cups  = $38
Milk  Toys  Belt  = $45
Milk  Toys  Cups  = $33
Milk  Cups  Belt  = $38
Milk  Cups  Toys  = $33
Belt  Milk  Toys  = $45
Belt  Milk  Cups  = $38
Belt  Toys  Milk  = $45
Belt  Toys  Cups  = $55
Belt  Cups  Milk  = $38
Belt  Cups  Toys  = $55
Toys  Milk  Belt  = $45
Toys  Milk  Cups  = $33
Toys  Belt  Milk  = $45
Toys  Belt  Cups  = $55
Toys  Cups  Milk  = $33
Toys  Cups  Belt  = $55
Cups  Milk  Belt  = $38
Cups  Milk  Toys  = $33
Cups  Belt  Milk  = $38
Cups  Belt  Toys  = $55
Cups  Toys  Milk  = $33
Cups  Toys  Belt  = $55
```


ShoppingSpreeCombinations.java:

```
import java.util.ArrayList;

public class ShoppingSpreeCombinations {
   public static final int MAX_SHOPPING_BAG_SIZE = 3; // Max number of items in
shopping bag

   /* Output every combination of items that fit
      in a shopping bag. Each recursive call moves
      one item into the shopping bag.
   */
   public static void shoppingBagCombinations(ArrayList<GroceryItem> currBag,
// Bag contents
                                              ArrayList<GroceryItem>
remainingItems) { // Available items
      int bagValue;               // Cost of items in shopping bag
      GroceryItem tmpGroceryItem; // Grocery item to add to bag
      int i;                      // Loop index

      if (currBag.size() == MAX_SHOPPING_BAG_SIZE) {   // Base case: Shopping bag
full
         bagValue = 0;
         for (i = 0; i < currBag.size(); ++i) {
            bagValue += currBag.get(i).priceDollars;
            System.out.print(currBag.get(i).itemName + "  ");
         }
         System.out.println("= $" + bagValue);
      }
      else {                                           // Recursive case: move
one
         for (i = 0; i < remainingItems.size(); ++i) { // item to bag
            // Move item into bag
            tmpGroceryItem = remainingItems.get(i);
            remainingItems.remove(i);
            currBag.add(tmpGroceryItem);
```

```
            shoppingBagCombinations(currBag, remainingItems);

            // Take item out of bag
            remainingItems.add(i, tmpGroceryItem);
            currBag.remove(currBag.size() - 1);
         }
      }
   }

   public static void main(String[] args) {
      ArrayList<GroceryItem> possibleItems = new ArrayList<GroceryItem>(); //
Possible shopping items
      ArrayList<GroceryItem> shoppingBag = new ArrayList<GroceryItem>(); //
Current shopping bag
      GroceryItem tmpGroceryItem;                                         //
Temp item

      // Populate grocery with different items
      tmpGroceryItem = new GroceryItem();
      tmpGroceryItem.itemName = "Milk";
      tmpGroceryItem.priceDollars = 2;
      possibleItems.add(tmpGroceryItem);

      tmpGroceryItem = new GroceryItem();
      tmpGroceryItem.itemName = "Belt";
      tmpGroceryItem.priceDollars = 24;
      possibleItems.add(tmpGroceryItem);

      tmpGroceryItem = new GroceryItem();
      tmpGroceryItem.itemName = "Toys";
      tmpGroceryItem.priceDollars = 19;
      possibleItems.add(tmpGroceryItem);

      tmpGroceryItem = new GroceryItem();
      tmpGroceryItem.itemName = "Cups";
      tmpGroceryItem.priceDollars = 12;
      possibleItems.add(tmpGroceryItem);

      // Try different combinations of three items
      shoppingBagCombinations(shoppingBag, possibleItems);
   }
}
```


PARTICIPATION ACTIVITY 3.9.3: All letter combinations.

1) When main() calls shoppingBagCombinations(), how many items are in the remainingItems list?

- None
- 3
- 4

2) When main() calls shoppingBagCombinations(), how

many items are in currBag list?

- None
- 1
- 4

3) After main() calls shoppingBagCombinations(), what happens first?

- The base case prints Milk, Belt, Toys.
- The method bags one item, makes recursive call.
- The method bags 3 items, makes recursive call.

4) Just before shoppingBagCombinations() returns back to main(), how many items are in the remainingItems list?

- None
- 4

5) How many recursive calls occur before the first combination is printed?

- None
- 1
- 3

6) What happens if main() only put 2, rather than 4, items in the possibleItems list?

- Base case never executes; nothing printed.
- Infinite recursion occurs.

Traveling salesman

Recursion is useful for finding all possible paths. Suppose a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those

three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 3.9.3: Find distance of traveling to 3 cities.


```
import java.util.ArrayList;

public class TravelingSalesmanPaths {
   public static final int NUM_CITIES = 3;                              // Number
of cities
   public static int[][] cityDistances = new int[NUM_CITIES][NUM_CITIES]; //
Distance between cities
   public static String[] cityNames = new String[NUM_CITIES];           // City
names

   /* Output every possible travel path.
      Each recursive call moves to a new city.
   */
   public static void travelPaths(ArrayList<Integer> currPath,
                                  ArrayList<Integer> needToVisit) {
      int totalDist;     // Total distance given current path
      int tmpCity;       // Next city distance
      int i;             // Loop index

      if ( currPath.size() == NUM_CITIES ) { // Base case: Visited all cities
         totalDist = 0;                      // Return total path distance
         for (i = 0; i < currPath.size(); ++i) {
            System.out.print(cityNames[currPath.get(i)] + "   ");

            if (i > 0) {
               totalDist += cityDistances[currPath.get(i - 1)][currPath.get(i)];
            }
         }

         System.out.println("= " + totalDist);
      }
      else {                                 // Recursive case: pick next city
         for (i = 0; i < needToVisit.size(); ++i) {
            // add city to travel path
            tmpCity = needToVisit.get(i);
            needToVisit.remove(i);
            currPath.add(tmpCity);

            travelPaths(currPath, needToVisit);

            // remove city from travel path
            needToVisit.add(i, tmpCity);
            currPath.remove(currPath.size() - 1);
         }
      }
   }

   public static void main (String[] args) {
      ArrayList<Integer> needToVisit = new ArrayList<Integer>(); // Cities left to
visit
      ArrayList<Integer> currPath = new ArrayList<Integer>();    // Current path
traveled

      // Initialize distances array
      cityDistances[0][0] = 0;
      cityDistances[0][1] = 960;  // Boston-Chicago
      cityDistances[0][2] = 2960; // Boston-Los Angeles
      cityDistances[1][0] = 960;  // Chicago-Boston
```

```
        cityDistances[1][1] = 0;
        cityDistances[1][2] = 2011; // Chicago-Los Angeles
        cityDistances[2][0] = 2960; // Los Angeles-Boston
        cityDistances[2][1] = 2011; // Los Angeles-Chicago
        cityDistances[2][2] = 0;

        cityNames[0] = "Boston";
        cityNames[1] = "Chicago";
        cityNames[2] = "Los Angeles";

        needToVisit.add(new Integer(0)); // Boston
        needToVisit.add(new Integer(1)); // Chicago
        needToVisit.add(new Integer(2)); // Los Angeles

        // Explore different paths
        travelPaths(currPath, needToVisit);
    }
}
```



```
Boston   Chicago   Los Angeles   = 2971
Boston   Los Angeles   Chicago   = 4971
Chicago   Boston   Los Angeles   = 3920
Chicago   Los Angeles   Boston   = 4971
Los Angeles   Boston   Chicago   = 3920
Los Angeles   Chicago   Boston   = 2971
```

PARTICIPATION ACTIVITY 3.9.4: Recursive exploration.

1) You wish to generate all possible 3-letter subsets from the letters in an N-letter word (N>3). Which of the above recursive methods is the closest?
   - shoppingBagCombinations
   - scrambleLetters
   - main()

Exploring further:

- Recursive Algorithms from khanacademy.org

3.10 Stack overflow

Recursion enables an elegant solution to some problems. But, for large problems, deep recursion can cause memory problems. Part of a program's memory is reserved to support function calls. Each method call places a new ***stack frame*** on the stack, for local parameters, local variables, and more method items. Upon return, the frame is deleted.

Deep recursion could fill the stack region and cause a ***stack overflow***, meaning a stack frame extends beyond the memory region allocated for stack, Stack overflow usually causes the program to crash and report an error like: stack overflow error or stack overflow exception.

PARTICIPATION ACTIVITY 3.10.1: Recursion causing stack overflow.

Animation captions:

1. Deep recursion may cause stack overflow, causing a program to crash.

The animation showed a tiny stack region for easy illustration of stack overflow.

The number (and size) of parameters and local variables results in a larger stack frame. Large ArrayLists, arrays, or Strings declared as local variables can lead to faster stack overflow.

A programmer can estimate recursion depth and stack size to determine whether stack overflow might occur. Sometime a non-recursive algorithm must be developed to avoid stack overflow.

PARTICIPATION ACTIVITY 3.10.2: Stack overflow.

1) A memory's stack region can store at most one stack frame.
   - True
   - False

2) The size of the stack is unlimited.
   - True
   - False

3) A stack overflow occurs when the stack frame for a method call extends past the end of the stack's memory.
   - True
   - False

4) The following recursive method will

result in a stack overflow.

```
int recAdder(int inValue) {
   return recAdder(inValue + 1);
}
```

- True
- False

3.11 Java example: Recursively output permutations

zyDE 3.11.1: Recursively output permutations.

The below program prints all permutations of an input string of letters, one permut
line. Ex: The six permutations of "cab" are:

```
cab
cba
acb
abc
bca
bac
```

Below, the permuteString method works recursively by starting with the first chara
permuting the remainder of the string. The method then moves to the second char
permutes the string consisting of the first character and the third through the end
and so on.

1. Run the program and input the string "cab" (without quotes) to see that the a
is produced.
2. Modify the program to print the permutations in the opposite order, and also
permutation count on each line.
3. Run the program again and input the string cab. Check that the output is reve
4. Run the program again with an input string of abcdef. Why did the program t
to produce the results?

Load default

```
1  import java.util.Scanner;
2
3  public class Permutations {
4     // FIXME: Use a static variable to count permutations. Why must it be s
5
6     public static void permuteString(String head, String tail) {
7        char current;
8        String newPermute;
9        int len;
10       int i;
11
12       current = '?';
13       len = tail.length();
14
15       if (len <= 1) {
16          // FIXME: Output the permutation count on each line too
17          System.out.println(head + tail);
18       }
```


cab

Run

zyDE 3.11.2: Recursively output permutations (solution).

Below is the solution to the above problem.

Load default

```
1  import java.util.Scanner;
2
3  public class PermutationsSolution {
4     static int permutationCount = 0;
5
6     public static void permuteString(String head, String tail) {
7        char current;
8        String newPermute;
9        int len;
10       int i;
11
12       current = '?';
13       len = tail.length();
14
15       if (len <= 1) {
16          ++permutationCount;
17          System.out.println(permutationCount + ") " + head + tail);
18       }
```

```
cab
abcdef
```

Run

3.12 Greedy algorithms

This section has been set as optional by your instructor.

Greedy algorithm

A ***greedy algorithm*** is an algorithm that, when presented with a list of options, chooses the option that is optimal at that point in time. The choice of option does not consider additional subsequent options, and may or may not lead to an optimal solution.

PARTICIPATION ACTIVITY 3.12.1: MakeChange greedy algorithm.

Animation content:

undefined

Animation captions:

1. The change making algorithm uses quarters, dimes, nickels, and pennies to make change equaling the specified amount.
2. The algorithm chooses quarters as the optimal coins, as long as the remaining amount is >= 25.
3. Dimes offer the next largest amount per coin, and are chosen while the amount is >= 10.
4. Nickels are chosen next. The algorithm is greedy because the largest coin <= the amount is always chosen.
5. Adding one penny makes 91 cents.
6. This greedy algorithm is optimal and minimizes the total number of coins, although not all greedy algorithms are optimal.

PARTICIPATION ACTIVITY 3.12.2: Greedy algorithms.

1) If the MakeChange function were to make change for 101, what would be the result?
   - 101 pennies
   - 4 quarters and 1 penny
   - 3 quarters, 2 dimes, 1 nickel, and 1 penny

2) A greedy algorithm is attempting to minimize costs and has a choice between two items with equivalent functionality: the first costing $5 and the second costing $7. Which will be chosen?
   - The $5 item
   - The $7 item
   - The algorithm needs more information to choose

3) A greedy algorithm always finds an optimal solution.
   - True
   - False

Fractional knapsack problem

The ***fractional knapsack problem*** is the knapsack problem with the potential to take each item a fractional number of times, provided the fraction is in the range [0.0, 1.0]. Ex: A 4 pound, $10 item could be taken 0.5 times to fill a knapsack with a 2 pound weight limit. The resulting knapsack would be worth $5.

While a greedy solution to the 0-1 knapsack problem is not necessarily optimal, a greedy solution to the fractional knapsack problem is optimal. First, items are sorted in descending order based on the value-to-weight ratio. Next, one of each item is taken from the item list, in order, until taking 1 of the next item would exceed the weight limit. Then a fraction of the next item in the list is taken to fill the remaining weight.

Figure 3.12.1: FractionalKnapsack algorithm.

```
FractionalKnapsack(knapsack, itemList, itemListSize) {
   Sort itemList descending by item's (value / weight) ratio
   remaining = knapsack→maximumWeight
   for each item in itemList {
      if (item→weight <= remaining) {
         Put item in knapsack
         remaining = remaining - item→weight
      }
      else {
         fraction = remaining / item→weight
         Put (fraction * item) in knapsack
         break
      }
   }
}
```


PARTICIPATION ACTIVITY 3.12.3: Fractional knapsack problem.

Suppose the following items are available: 40 pounds worth $80, 12 pounds worth $18, and 8 pounds worth $8.

1) Which item has the highest value-to-weight ratio?
   - 40 pounds worth $80
   - 12 pounds worth $18
   - 8 pounds worth $8

2) What would FractionalKnapsack put in a 20-pound knapsack?
   - One 12-pound item and one 8-pound item
   - One 40-pound item
   - Half of a 40-pound item

3) What would FractionalKnapsack put in a 48-pound knapsack?
   - One 40-pound item and one 8-pound item
   - One 40-pound item and 2/3 of a 12-pound item

Activity selection problem

The ***activity selection problem*** is a problem where 1 or more activities are available, each with a start and finish time, and the goal is to build the largest possible set of activities without time conflicts. Ex: When on vacation, various activities such as museum tours or mountain hikes may be available. Since vacation time is limited, the desire is often to engage in the maximum possible number of activities per day.

A greedy algorithm provides the optimal solution to the activity selection problem. First, an empty set of chosen activities is allocated. Activities are then sorted in ascending order by finish time. The first activity in the sorted list is marked as the current activity and added to the set of chosen activities. The algorithm then iterates through all activities after the first, looking for a next activity that starts after the current activity ends. When such a next activity is found, the next activity is added to the set of chosen activities, and the next activity is reassigned as the current. After iterating through all activities, the chosen set of activities contains the maximum possible number of non-conflicting activities from the activities list.

PARTICIPATION ACTIVITY 3.12.4: Activity selection problem algorithm.

Animation content:

`undefined`

Animation captions:

1. Activities are first sorted in ascending order by finish time. The set of chosen activities initially has the activity that finishes first.
2. The morning mountain hike does not start after the history museum tour finishes and is not added to the chosen set of activities.
3. The boat tour is the first activity to start after the history museum tour finishes, and is the "greedy" choice.
4. Hang gliding and the fireworks show are chosen as 2 additional activities.
5. The maximum possible number of non-conflicting activities is 4, and 4 have been chosen.

PARTICIPATION ACTIVITY 3.12.5: ActivitySelection algorithm.

1) The fireworks show and the night movie both finish at 9 PM, so the sorting algorithm could have swapped the order of the 2. If the 2 were swapped, the number of

chosen activities would not be affected.

- ○ True
- ○ False

2) Changing snorkeling's _____ would cause snorkeling to be added to the chosen activities.

- ○ start time from 3 PM to 4 PM
- ○ finish time from 5 PM to 4 PM

3) Regardless of any changes to the activity list, the activity with the _____ will always be in the result.

- ○ earliest start time
- ○ earliest finish time
- ○ longest length

3.13 Dynamic programming

This section has been set as optional by your instructor.

Dynamic programming overview

Dynamic programming is a problem solving technique that splits a problem into smaller subproblems, computes and stores solutions to subproblems in memory, and then uses the stored solutions to solve the larger problem. Ex: Fibonacci numbers can be computed with an iterative approach that stores the 2 previous terms, instead of making recursive calls that recompute the same term many times over.

PARTICIPATION ACTIVITY 3.13.1: FibonacciNumber algorithm: Recursion vs. dynamic programming.

Animation content:

```
undefined
```

Animation captions:

1. The recursive call hierarchy of FibonacciNumber(4) shows each call made to FibonacciNumber.
2. Several terms are computed more than once.
3. The iterative implementation uses dynamic programming and stores the previous 2 terms at a time.
4. For each iteration, the next term is computed by adding the previous 2 terms. The previous and current terms are also updated for the next iteration.
5. 3 loop iterations are needed to compute FibonacciNumber(4). Because the previous 2 terms are stored, no term is computed more than once.

PARTICIPATION ACTIVITY 3.13.2: FibonacciNumber implementation.

1) If the recursive version of FibonacciNumber(3) is called, how many times will be FibonacciNumber(2) be called?
   - 1
   - 2
   - 3

2) Which version of FibonacciNumber is faster for large term indices?
   - Recursive version
   - Iterative version
   - Neither

3) Which version of FibonacciNumber is more accurate for large term indices?
   - Recursive version
   - Iterative version
   - Neither

4) The recursive version of FibonacciNumber has a runtime complexity of $O(1.62^N)$. What is the

runtime complexity of the iterative version?

- O(N)
- O(N^2)
- O(1.62^N)

PARTICIPATION ACTIVITY 3.13.3: Dynamic programming.

1) Dynamic programming avoids recomputing previously computed results by storing and reusing such results.
   - True
   - False

2) Any algorithm that splits a problem into smaller subproblems is using dynamic programming.
   - True
   - False

Longest common substring

The ***longest common substring*** algorithm takes 2 strings as input and determines the longest substring that exists in both strings. The algorithm uses dynamic programming. An N x M integer matrix keeps track of matching substrings, where N is the length of the first string and M the length of the second. Each row represents a character from the first string, and each column represents a character from the second string.

An entry at i, j in the matrix indicates the length of the longest common substring that ends at character i in the first string and character j in the second. An entry will be 0 only if the 2 characters the entry corresponds to are not the same.

The matrix is built one row at a time, from the top row to the bottom row. Each row's entries are filled from left to right. An entry is set to 0 if the two characters do not match. Otherwise, the entry at i, j is set to 1 plus the entry in i - 1, j - 1.

PARTICIPATION ACTIVITY 3.13.4: Longest common substring algorithm.

Animation content:

undefined

Animation captions:

1. Comparing "Look" and "zyBooks" requires a 7x4 matrix.
2. In the first row, 0 is entered for each pair of mismatching characters.
3. In the next row, 'o' matches in 2 entries. In both cases the upper-left value is 0, and 1 is entered into the matrix.
4. Two matches for 'o' exist in the next row as well, with the second having a 1 in the upper-left entry.
5. The character 'k' matches once in the last row and an entry of 2 + 1 = 3 is entered.
6. The maximum entry in the matrix is the longest common substring's length. The maximum entry's row index is the substring ending index in the first string.

PARTICIPATION ACTIVITY 3.13.5: Longest common substring matrix.

Consider the matrix below for the two strings "Programming" and "Problem".

		P	r	o	g	r	a	m	m	i	n	g
		1	0	0	0	0	0	0	0	0	0	0
r		0	2	0	0	?	0	0	0	0	0	0
o		0	0	?	0	0	0	0	0	0	0	0
b		0	0	0	0	0	0	0	0	0	0	0
l		0	0	0	0	0	0	0	0	0	0	0
e		0	0	0	0	0	0	0	0	0	0	0
m		0	0	0	0	0	0	1	?	0	0	0

1) What should be the value in the green cell?

- 0
- 1
- 2

2) What should be the value in the yellow cell?

- 1
- 2
- 3

3) What should be the value in the blue cell?

- 0
- 1
- 2

4) What is the longest common substring?

- Pr
- Pro
- mm

Longest common substring algorithm complexity

The longest common substring algorithm operates on two strings of length N and M. For each of the N characters in the first string, M matrix entries are computed, making the runtime complexity O($N \cdot M$). Since an N x M integer matrix is built, the space complexity is also O($N \cdot M$).

Common substrings in DNA

A real-world application of the longest common substring algorithm is to find common substrings in DNA strings. Ex: Common substrings between 2 different DNA sequences may represent shared traits. DNA strings consist of characters C, T, A, and G.

PARTICIPATION ACTIVITY 3.13.6: Finding longest common substrings in DNA.

Animation content:


```
undefined
```

Animation captions:

1. Finding common substrings in DNA strings can be used for detecting things such as genetic disorders or for tracing evolutionary lineages.

2. DNA strings are very long, often billions of characters. Dynamic programming is crucial to obtaining a reasonable runtime.
3. Optimizations can lower memory usage by keeping only the following in memory: previous row data and largest matrix entry information.

PARTICIPATION ACTIVITY 3.13.7: Common substrings in DNA.

1) Which cannot be a character in a DNA string?
   - A
   - B
   - C

2) If an animal's DNA string is available, a genetic disorder might be found by finding the longest common substring from the DNA of another animal _____ the disorder.
   - with
   - without

3) When computing row X in the matrix, what information is needed, besides the 2 strings?
   - Row X - 1
   - Row X + 1
   - All rows before X

PARTICIPATION ACTIVITY 3.13.8: Longest common substrings - critical thinking.

1) If the largest entry in the matrix were the only known value, what could be determined?
   - The starting index of the longest common substring within either string
   - The character contents of the common substring

- The length of the longest common substring

2) Suppose only the row and column indices for the largest entry in the matrix were known, and not the value of the largest or any other matrix entry. What can be determined in O(1)?
   - Only the longest common substring's ending index within either string
   - The longest common substring's starting and ending indices within either string

Optimized longest common substring algorithm complexity

The longest common substring algorithm can be implemented such that only the previously computed row and the largest matrix entry's location and value are stored in memory. With this optimization, the space complexity is reduced to O(N). The runtime complexity remains O($N \cdot M$).

©zyBooks 02/06/20 14:39 652770
JEFFREY WAN
JHUEN605202Spring2020