|  |
| --- |
| **Exercise 11.1**  **A reasonable answer would be that it will go on printing 'Hello!' forever. However, the memory limitations of the JVM, and the lack of elimination of tail recursion mean that the program will eventually stop through lack of memory.** |
| **Exercise 11.3**  **A possible solution.**  public class RecursionTester  {  private int helloCount;  /\*\*  \* Recursive 'hello' method.  \*/  public void sayHello()  {  helloCount++;  System.out.println("Hello! " + helloCount);  sayHello();  }  **}** |
| **Exercise 11.5**  **Nothing will be printed.** |
| **Exercise 11.6**  **The version below, when called with a parameter of 5, prints:**  **5**  **Hello!**  **4**  **Hello!**  **3**  **Hello!**  **2**  **Hello!**  **1**  **Hello!**  **0**  **public void sayHello(int numberOfTimes)**  **{**  **System.out.println(numberOfTimes);**  **if(numberOfTimes > 0) {**  **System.out.println("Hello!");**  **sayHello(numberOfTimes - 1);**  **}**  **}** |
| **Exercise 11.7**  **The version below, when called with a parameter of 5, prints:**  **5**  **4**  **3**  **2**  **1**  **0**  **Hello!**  **Hello!**  **Hello!**  **Hello!**  **Hello!**  **public void sayHello(int numberOfTimes)**  **{**  **System.out.println(numberOfTimes);**  **if(numberOfTimes > 0) {**  **sayHello(numberOfTimes - 1);**  **System.out.println("Hello!");**  **}**  **}**  **The parameter values are printed immediately as each recursive call is made.**  **The printing of the "Hello!" strings cannot start until the recursion unwinds as a result of the base case occurring when the parameter is 0.**  **In the output from this alternative version, the printing in reverse order as the recursion unwinds is clearer:**  **Hello! 1**  **Hello! 2**  **Hello! 3**  **Hello! 4**  **Hello! 5**  **public void sayHello(int numberOfTimes)**  **{**  **if(numberOfTimes > 0) {**  **sayHello(numberOfTimes - 1);**  **System.out.println("Hello! " + numberOfTimes);**  **}**  **}** |
| **Exercise 11.10**  **Here is one set of 20 values:**  **1312 1808 2116 2474 2525 3072 3343 3499 3879 3942 6128 6371 7275 7516 8211 8590 8675 9176 9419 9974**  **Searching for 1000, "Search" was printed 6 times.**  **Searching for 1313, "Search" was printed 6 times.**  **Searching for 3500, "Search" was printed 5 times.**  **Searching for 8589, "Search" was printed 5 times.**  **Searching for 9000, "Search" was printed 6 times.** |
| **Exercise 11.11**  **Here is one set of 20 values:**  **1312 1808 2116 2474 2525 3072 3343 3499 3879 3942 6128 6371 7275 7516 8211 8590 8675 9176 9419 9974**  **Searching for 1312, "Search" was printed 5 times.**  **Searching for 6128, "Search" was printed 1 time.**  **Searching for 7516, "Search" was printed 3 times.**  **Searching for 8590, "Search" was printed 2 times.**  **Searching for 9974, "Search" was printed 4 times.** |
| **Exercise 11.12**  **The following should be listed:**  **.DS\_Store**  **images**  **Thesis.pdf**  **notes** |
| **Exercise 11.13**  **The file .DS\_Store might not appear in the folder view files with '.' as their first character are often 'hidden' by the native file system viewer.** |
| **Exercise 11.14**  **The new file should be listed.** |
| **Exercise 11.18**  **The code below results in the following output from myfolder:**  **Folder: myfolder**  **.DS\_Store**  **[images]**  **Thesis.pdf**  **[notes]**  **Folder: images**  **dover.jpg**  **london.jpg**  **gothic.jpg**  **whitstable.jpg**  **bluej-icon.png**  **Folder: notes**  **shopping.txt**  **note.txt**  **private void listFolder(File folder)**  **{**  **if(folder.isDirectory()) {**  **List<File> subfolders = new LinkedList<>();**  **System.out.println();**  **System.out.println("Folder: " + folder.getName());**  **for(File f : folder.listFiles()) {**  **if(f.isDirectory()) {**  **subfolders.add(f);**  **System.out.printf("[%s]%n", f.getName());**  **}**  **else {**  **System.out.println(f.getName());**  **}**  **}**  **for(File f : subfolders) {**  **listFolder(f);**  **}**  **}**  **}** |
| **Exercise 11.20**  **/\*\***  **\* Get the size of the given folder and all its contents.**  **\***  **\* @param folderName The name of the folder.**  **\* @return The size of the folder.**  **\*/**  **public long getFolderSize(String folderName)**  **{**  **return getSize(new File(folderName));**  **}**  **/\*\***  **\* Get the size of the given folder and all its contents.**  **\* If the parameter is not a folder, return its size.**  **\***  **\* @param folder The name of the folder.**  **\* @return The size of the folder.**  **\*/**  **private long getSize(File folder)**  **{**  **if(folder.isDirectory()) {**  **long totalSize = 0;**  **List<File> subfolders = new LinkedList<>();**  **for(File f : folder.listFiles()) {**  **if(f.isDirectory()) {**  **subfolders.add(f);**  **}**  **else {**  **totalSize += f.length();**  **}**  **}**  **for(File f : subfolders) {**  **totalSize += getSize(f);**  **}**  **return totalSize;**  **}**  **else {**  **// Not actually a folder.**  **return folder.length();**  **}**  **}** |
| **Exercise 11.21**  **Making the recursive call immediately, as in the following, does not solve the problem.**  **/\*\***  **\* Return the route from this room to the destination, if a route exists.**  **\* NB: This method does not work correctly.**  **\* @param destination The destination.**  **\* @return The route, or null if there isn't one.**  **\*/**  **public String findRoute(String destination)**  **{**  **String route = null;**  **Iterator<Entry<String, Room>> it =   exits.entrySet().iterator();**  **while(it.hasNext() && route == null) {**  **Entry<String, Room> pair = it.next();**  **if(pair.getValue().getName().equals(destination)) {**  **// Directly connected.**  **route = pair.getKey();**  **}**  **else {**  **// Try going via that exit.**  **route = pair.getValue().findRoute(destination);**  **if(route != null) {**  **route = pair.getKey() + " " + route;**  **}**  **}**  **}**  **return route;**  **}** |
| **Exercise 11.22**  **The version in Code 11.10 will find a route but the solution to Exercise 11.21 will not. The reason is that Code 11.10 defers the recursive call from Y to X and checks first the exit to Z (the required destination) and finds the route.**  **The solution to Exercise 11.21 results in X and Y recursively checking each other repeatedly without Y considering the Z exit.** |