Jake Wiseberg

Intro to Data Structures

Writing Assignment 1

1.

i)

**public class** Book<T> {  
 **private** T **id**;  
 **private boolean available**;  
  
 **public** Book(T id) {  
 **this**.**id** = id;  
 **available** = **true**;  
 }  
  
 **public** T getId() { **return id**; }  
 **public boolean** isAvailable() { **return available**; }  
  
 **public boolean** setAvailable(**boolean** a) {  
 **available** = a;  
 **return available**;  
 }  
}

**public class** Library<T> {  
 **private** Book[] **lib**;  
 **private int len**;  
  
 **public** Library(**int** l) {  
 **lib** = **new** Book[l];  
 **len** = 0;  
 }  
  
 **public boolean** onLoan(T t) {  
 **for** (**int** i=0; i<**lib**.**length**; i++) {  
 **if** (**lib**[i].getId() == t) {  
 **lib**[i].setAvailable(**false**);  
 **return true**;  
 }  
 }  
 **return false**;  
 }  
  
 **public boolean** addBook(T t) {  
 **if** (**len** < **lib**.**length** && **lib**[**len**] == **null**) {  
 **lib**[**len**] = (Book)t;  
 **len**++;  
 **return true**;  
 }  
 **return false**;  
 }  
  
 **public boolean** removeBook(T t) {  
 **for** (**int** i=0; i<**lib**.**length**; i++) {  
 **if** (**lib**[i].getId() == t) {  
 **lib**[i] = **null**;  
 **return true**;  
 }  
 }  
 **return false**;  
 }  
  
 **public** Book getBook(T t) {  
 **for** (**int** i=0; i<**lib**.**length**; i++) {  
 **if** (**lib**[i].getId() == t)  
 **return lib**[i];  
 }  
 **return null**;  
 }  
}

ii)

**public class** Test {  
 **public static void** main(String[] args) {  
 Library books = **new** Library(2);  
 books.addBook(**new** Book<String>(**"Ready PLayer One"**));  
 books.addBook(**new** Book<Integer>(121432));  
 System.***out***.println(books.getBook(**"Ready Player One"**));  
 }  
}

iii)

**public class** Test {  
 **public static void** main(String[] args) {  
 **int** l = 2;  
 Library books = **new** Library(l);  
 books.addBook(**new** Book<String>(**"Ready PLayer One"**));  
 books.addBook(**new** Book<Integer>(121432));  
 **for** (**int** i=0; i<l; i++) {  
 System.***out***.println(books.getBook(i));  
 }  
 }  
}

2.

This program would not work because the recursion would never end. Since the base case is y==0, it would be necessary to include some type of decrease in y to get to that point so the program doesn’t run forever. Instead, on each run through the program adds 1 to y, so with the invocation of pow(2,3), y would forever increase and there would be too many recursive sections that would never end. The result of pow(2,3) is actually an error, a StackOverflowError. According to the javadocs this error is caused when “an application recurses too deeply.” In this situation the program would check if y is 0, and since it isn’t it would return x times the method with x and y+1. Using this y would never become 0 so the base case would never be met.

3.

String reversei(String str) {  
 **char**[] temp = str.toCharArray();  
 String str2 = **""**;  
 **for** (**int** i=temp.**length**-1; i>=0; i--) {  
 str2 += temp[i];  
 }  
 **return** str2;  
}

4.

**int** min(**int**[] a, **int** index) {  
 **if** (index == a.**length** - 1)  
 **return** a[index];  
 **int** min = *min*(a, index + 1);  
 **if** (a[index] < min)  
 **return** a[index];  
 **else  
 return** min;  
}

5.

f1(n) + f2(n) = O[ max( g1(n) , g2(n) )]

f1(n) = O(g1(n))

f2(n) = O(g2(n))

O(g1(n)) + O(g2(n)) = O[ max( g1(n) , g2(n) )]

O( g1(n) + g2(n) ) = O[ max( g1(n) , g2(n) )]

Based on the mathematical definition of Big O functions, O( g1(n) + g2(n) ) would be equal to which ever function, either g1(n) or g2(n), grows faster. Therefore f1(n) + f2(n) = O[ max( g1(n) , g2(n) )].

6.

1. O(n^1.1 + nlog(n)) = O(nlogn)
2. O(log2(n^2) + ln(n) + 21) = O(ln(n))
3. O(15n^2+17n^1.2+4n) = O(n^2)

7.

1. O(n) - This algorithm searches through an array to find any value that is equal to -1 in the array, a, and then changes it to the variable, val. For worst-case runtime, the algorithm would need to run through the entire array once, therefor the runtime complexity would be described as O(n) where n is the length of the array.
2. O(n^3) - This algorithm calculates the sum of all of the factors of every number from 0 to n. This means that the runtime complexity for the worst-case scenario would be O(n^3) since 3 for loops are involved so there would be 3 levels of iteration.
3. O(n^2) - This algorithm calculates the sum of all of the numbers from 0 to n and adds the number, i, multiplied by the following number (so i\*i+1). Since this requires 2 for loops there would be 2 levels of iterations at the worst-case making the runtime complexity O(n^2)