COMP250

Midterm 2

Series: $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$, $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$
$\sum_{i=1}^{n-1} r^i = \frac{r^n-1}{r-1}$
Logarithms:
$\log_b(M^k) = k\log_b M$, $\quad k^{\log_b M} = M^{\log_b k}$
Asymptotic notations:

```
Big-Oh: an asymptotic upper bound. We say that f is O(g(n)) if, for sufficiently large n, the function f(n) is bounded above by a constant multiple of g(n).
Big-Omega: an asymptotic lower bound. We say that f is OMEGA(g(n)) if, for sufficiently large n, the function f(n) is bounded below by a constant multiple of g(n).
Big-Theta: a tight bound. If we can show that a running time f(n) is both O(g(n)) and OMEGA(g(n)), then g(n) is a tight bound for f(n). That is, f(n) grows like g(n) to within a constant factor.
1 < log2n < n < nlog2n < n^2 < n^3 <...< 2^n < n!
```

Interfaces:

```
All methods are by default public and abstract.
All fields are by default public, static, and final
Interfaces cannot be instantiated. Syntax:
public interface myInterface {}
To use an interface you first need a class that implements it.
public class Dragon extends Enemy implements MonsterLike, FireBreather {}
A class can implement more than one interface.
Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface
Classes can extend at most one class, but they can implement multiple interfaces.
Interfaces define new data types, we can create variables of those type and assign to them any value referencing to instances of classes that implement the specified interface.
A class that implements an interface must implement every method from such interface.
```

Generics:

```
A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter.
Cannot put primitive types inside the brackets, must use the wrapper classes.
public class Cage<T> {}
BOUNDED TYPE:
To restrict the types that can be used in the class
public class Cage<T extends MonsterLike>
This limits the types we can use to instantiate a generic type and allows us to use methods defined in the bounds.
```

Comparable:

```
The Java Comparable interface is used to define an ordering on objects of user-defined class.
public class T implements Comparable<T>{}
A class that implements Comparable must declare the method compareTo(T obj).
t1.compareTo(t2) returns negative int if t1<t2, 0 if t1=t2, analogous for t1>t2.
(t1.compareTo(t2)==0) == (t1.equals(t2))
```

Iterable and Iterator:

```
The use of a for-each loop is made possible by the use of two interfaces: Iterator and Iterable.
Iterable objects are representations of a series of elements that can be iterated over.
Iterator objects allow you to iterate through objects that represent a collection (a series of elements).
A class that implements Iterable needs to implement the iterator() method. The iterator() method returns an object of type Iterator that can iterate through the elements of this instance.
A class that implements Iterator needs to implement the methods hasNext() and next().
iterator() returns an iterator to the start of the collection. Using hasNext() and next() you can move forward in the collection.
Iterator class is generally an inner class of Iterable

public interface Iterable<T> {
   public Iterator<T> iterator();}
public interface Iterator<T> {
   boolean hasNext();
   T next(); // returns current, advances to next}
```

Mergesort:

```
mergesort(list){
   if (list.size() == 1)
      return list
   else {
      mid = (list.size() - 1) / 2
      list1 = list.getElements(0,mid)
      list2 = list.getElements(mid+1, list.size()-1)
      list1 = mergesort(list1)
      list2 = mergesort(list2)
      return merge(list1, list2)}}
merge(list1, list2){
   list = ...initialize with empty list...
   while (!list1.isEmpty() && !list2.isEmpty()){
      if (list1.get(0) < list2.get(0))
         list.addlast(list1.removeFirst())
      else
         list.addlast(list2.removeFirst())
   }
   while (!list1.isEmpty())
      list.addlast(list1.removeFirst())
   while (!list2.isEmpty())
      list.addlast( list2.removeFirst())
   return list
   }
```

QuickSort:

```
Pick an element of the array (the pivot).
Move the pivot to its correct position making sure that all the smaller elements are on its left and all the larger elements are on its right.
Sort the left part AND the right part
Keep doing it until there is nothing left to sort.
quickSort(list, leftIndex, rightIndex) {
   // Base case:
   if(leftIndex >= rigthIndex) {
      return; // done!
   } else { // recursive step:
      i<-placeAndDivide(list, leftIndex, rightIndex)
      // i = index where the pivot is placed
      quickSort(list, leftIndex, i-1)
      quickSort(list, i+1, rightIndex)} }
placeAndDivide(list, leftIndex,rightIndex) {
   // pick the right most element
   pivot <- list.get(rigthIndex)
   // place the wall to the left
   wall <- leftIndex -1
   // go through all elements and compare to pivot
   for(int i=leftIndex; i< rigthIndex; i++) {
      if(list.get(i)<pivot) {
         wall++; // move wall
         swap (list.get(i),list.get(wall));}}// move element behind wall
   swap(list.get(rigthIndex),list.get(wall+1)) // move pivot next to wall
   return wall+1;}
```

Binary Tree Traversal

```
Pre order:
preorderBT(root){
      if (root is not empty){
            visit root
            preorderBT(root.left)
            preorderBT(root.right)}}
visit parent, then subtrees
In order:
inorderBT(root){
      if (root is not empty){
            inorderBT(root.left)
```

```
            visit root
            inorderBT(root.right)}}
Visit left subtree, then parent, then right right
    ↪ subtree
Post order
postorderBT(root){
      if (root is not empty){
            postorderBT(root.left)
            postorderBT(root.right)
            visit root} }
treeTraversalUsingStack(root){
   initialize empty stack s s.push(root)
   while s is not empty {
      cur = s.pop()
      visit cur
      for each child of cur
         s.push(child) }}
treeTraversalUsingQueue(root){ \\Breadth first
   initialize empty queue q
   q.enqueue(root)
   while q is not empty {
      cur = q.dequeue()
      visit cur
      for each child of cur
         q.enqueue(child)} }
```

Binary search tree:

```
Binary tree with comparable and unique keys. For each
    ↪ node, all descendants in left subtree are less
    ↪ than the node, and all descendants in the node'
    ↪ s right subtree are greater than the node.
An INORDER traversal visits the nodes in increasing
    ↪ order.
add(root, key){ // returns root node
   if (root == null)
      root = new BSTnode(key)
   else if (key < root.key){
      root.left = add(root.left,key)}
   else if (key > root.key){
      root.right = add(root.right,key)}
   return root
}
remove(root, key){ // returns root node
   if( root == null )
      return null
   else if ( key < root.key )
      root.left = remove(root.left, key)
   else if ( key > root.key )
      root.right = remove(root.right, key)
   else if (root.left == null)
      root = root.right
   else if (root.right == null)
      root = root.left
   else {
      root.key = findMin(root.right).key
      root.right = remove(root.right, root.key)}
   return root}
```

Recursive

```
Tower of Hanoi
tower(n, start, finish, other) { // e.g. tower(5,A,B,C)
   if(n==1) {
       move from start to finish.
   } else {
       tower(n-1, start, other, finish)
       tower(1, start, finish, other)
       tower(n-1, other, finish, start)
} }
Power iterative
power(x, n) {
   int result =1;
   for(int i=1; i<=n; i++) {
      result = result *x; }
   return result;
}
Power Recursive 1
power(x, n) { if(n==0) {
   return 1;
   } else {
      return x*power(x,n-1);}}
Power recursive 2
power( x, n) { if (n == 0)
   return 1;
   else if (n == 1)
      return x;
   else{
      tmp = power(x, n/2);
      if (n\%2==0)
         return tmp*tmp; // one mult.
      else
         return tmp*tmp*x; // 2 mult.
}}
```

Binary Search

```
binarySearch(list, key, left, right){
   if(left <= right){
      mid = (left + right)/2
      if(list[mid]==key)
         return mid
      else {
         if(key<list[mid])
            return binarySearch(list, key, left, mid
                ↪ -1)
         else
         return binarySearch(list, key, mid+1, right)
   }}return -1 }
```

ok