CSC 421: Algorithm Design and Analysis
Spring 2013

221/222/321 review
- object-oriented design
  - cohesion, coupling
  - interfaces, inheritance, polymorphism
- data structures
  - Java Collection Framework, List (ArrayList, LinkedList)
  - Set (TreeSet, HashSet), Map (TreeSet, HashMap),
  - Graph, Stack, Queue
- algorithm efficiency
  - Big Oh, rate-of-growth
  - analyzing iterative & recursive algorithms

OO design issues

describes how well a unit of code maps to an entity or behavior
in a highly cohesive system:
- each class maps to a single, well-defined entity – encapsulating all of its internal
  state and external behaviors
- each method of the class maps to a single, well-defined behavior
- leads to code that is easier to read and reuse

describes the interconnectedness of classes
in a loosely coupled system:
- each class is largely independent and communicates with other classes vi a small,
  well-defined interface
- leads to code that is easier to develop and modify
Interfaces

an interface defines a generic template for a class
- specifies the methods that the class must implement
- but, does not specify fields nor method implementations

```java
public interface List<T> {
    boolean add(T obj);
    boolean add(int index, T obj);
    void clear();
    boolean contains(Object obj);
    T get(int index);
    T remove(int index);
    boolean remove(T obj);
    T set(int index, T obj);
    int size();
    // ...
}
```

advantage: can define different implementations with different tradeoffs

```java
public class ArrayList<T> implements List<T> { … }
// uses array, so direct access
// but must shift when add/remove
public class LinkedList<T> implements List<T> { … }
// uses doubly-linked list, so
// sequential access but easy
// add/remove
```

- so, can write generic code that works on a List → either implementation will work

Interfaces & polymorphism

polymorphism: the capability of objects to react differently to the same method call

leads to more general-purpose code
- when called with an ArrayList, its version of a list iterator (array indexing) is used
- when called with a LinkedList, its version of an iterator (list referencing) is used

Collections class contains static methods that manipulate generic Lists
- shuffle, reverse, …
- max, sort, binarySearch, …

```java
public int longest(List<String> words) {
    if (words.size() == 0) {
        throw new java.util.NoSuchElementException();
    } int sofar = 0;
    for (String w : words) {
        if (w.length() > sofar) {
            sofar = w.length();
        }
    }
    return sofar;
}
```

ArrayList<String> words1 = new ArrayList<String>(); …
```java
int num1 = this.longest(words1);
```

```
LinkedList<String> words2 = new LinkedList<String>(); …
int num2 = this.longest(words2);
```

List words3 = words1; …
```java
int num3 = this.longest(words3);
```
Inheritance

inheritance is a mechanism for enhancing existing classes

- one of the most powerful techniques of object-oriented programming
- allows for large-scale code reuse

Here, a static field is used so that each account has a unique number

```
public class BankAccount {
    private double balance;
    private int accountNumber;
    private static int nextNumber = 1;
    public BankAccount() {
        this.balance = 0;
        this.accountNumber = this.nextIntNumber;
        this.nextIntNumber++;
    }
    public int getAccountNumber() {
        return this.accountNumber;
    }
    public double getBalance() {
        return this.balance;
    }
    public void deposit(double amount) {
        this.balance += amount;
    }
    public void withdraw(double amount) {
        if (amount >= this.balance) {
            this.balance -= amount;
        }
    }
}
```

```
public class SavingsAccount extends BankAccount {
    private double interestRate;
    public SavingsAccount(double rate) {
        this.interestRate = rate;
    }
    public void addInterest() {
        double interest = this.getBalance()*this.interestRate/100;
        this.deposit(interest);
    }
}
```

```
public class CheckingAccount extends BankAccount {
    private int transCount;
    private static final int NUM_FREE = 3;
    private static final double TRANS_FEE = 2.0;
    public CheckingAccount() {
        this.transCount = 0;
    }
    public void deposit(double amount) {
        super.deposit(amount);
        this.transCount++;
    }
    public void withdraw(double amount) {
        super.withdraw(amount);
        this.transCount++;
    }
    public void deductFees() {
        if (this.transCount > NUM_FREE) {
            double fees = TRANS_FEE*(this.transCount – NUM_FREE);
            super.withdraw(fees);
            this.transCount = 0;
        }
    }
}
```

Derived classes

```
public class SavingsAccount extends BankAccount {
    private double interestRate;
    public SavingsAccount(double rate) {
        this.interestRate = rate;
    }
    public void addInterest() {
        double interest = this.getBalance()*this.interestRate/100;
        this.deposit(interest);
    }
}
```

```
public class CheckingAccount extends BankAccount {
    private int transCount;
    private static final int NUM_FREE = 3;
    private static final double TRANS_FEE = 2.0;
    public CheckingAccount() {
        this.transCount = 0;
    }
    public void deposit(double amount) {
        super.deposit(amount);
        this.transCount++;
    }
    public void withdraw(double amount) {
        super.withdraw(amount);
        this.transCount++;
    }
    public void deductFees() {
        if (this.transCount > NUM_FREE) {
            double fees = TRANS_FEE*(this.transCount – NUM_FREE);
            super.withdraw(fees);
            this.transCount = 0;
        }
    }
}
```

**a derived class automatically inherits all fields and methods (but private fields are inaccessible)**

- can override existing methods or add new fields/methods as needed
Inheritance & polymorphism

polymorphism applies to classes in an inheritance hierarchy

- can pass a derived class wherever the parent class is expected
- the appropriate method for the class is called

```java
public void showAccount(BankAccount acct) {
    System.out.println("Account "+ acct.getAccountNumber()+": "+ acct.getBalance());
}
```

BankAccount acct1 = new BankAccount();
showAccount(acct1);
SavingsAccount acct2 = new SavingsAccount();
showAccount(acct2);
CheckingAccount acct3 = new CheckingAccount();
showAccount(acct3);
```

Java Collection classes

a collection is an object (i.e., data structure) that holds other objects

the Java Collection Framework is a group of generic collections

- defined using interfaces abstract classes, and inheritance

```
Collection
  - List
    - ArrayList
    - LinkedList
  - Set
    - OrderedSet
      - TreeSet
    - HashSet
  - Map
    - OrderedMap
      - TreeMap
    - HashMap
  - Graph
    - DiGraph
```
Sets

java.util.Set interface: an unordered collection of items, with no duplicates

public interface Set<E> extends Collection<E> {  
  boolean add(E o); // adds o to this Set  
  boolean remove(Object o); // removes o from this Set  
  boolean contains(Object o); // returns true if o in this Set  
  boolean isEmpty(); // returns true if empty Set  
  int size(); // returns number of elements  
  void clear(); // removes all elements  
  Iterator<E> iterator(); // returns iterator  
  . . . }

implemented by TreeSet and HashSet classes

TreeSet implementation
✓ implemented using a red-black tree; items stored in the nodes (must be Comparable)
✓ provides O(log N) add, remove, and contains (guaranteed)
✓ iteration over a TreeSet accesses the items in order (based on compareTo)

HashSet implementation
✓ HashSet utilizes a hash table data structure
✓ HashSet provides O(1) add, remove, and contains (on average, but can degrade)

Dictionary revisited

note: Dictionary class could have been implemented using a Set

- Strings are Comparable, so could use either implementation
- TreeSet has the advantage that iterating over the Set elements gives them in order (here, alphabetical order)
Maps

java.util.Map interface: a collection of key \(\rightarrow\) value mappings

```java
public interface Map<K, V> {
    boolean put(K key, V value); // adds key\rightarrow value to Map
    V remove(Object key); // removes key\rightarrow entry from Map
    V get(Object key); // returns true if o in this Set
    boolean containsKey(Object key); // returns true if key is stored
    boolean isEmpty(); // returns true if empty Set
    int size(); // returns number of elements
    void clear(); // removes all elements
    Set<K> keySet(); // returns set of all keys
}
```

implemented by TreeMap and HashMap classes

- TreeMap implementation
  - utilizes a red-black tree to store key/value pairs; ordered by the (Comparable) keys
  - provides \(O(\log N)\) put, get, and containsKey (guaranteed)
  - keySet() returns a TreeSet, so iteration over the keySet accesses the key in order

- HashMap implementation
  - HashSet utilizes a HashSet to store key/value pairs
  - HashSet provides \(O(1)\) put, get, and containsKey (on average, but can degrade)

Word frequencies

a variant of Dictionary is WordFreq

- stores words & their frequencies (number of times they occur)
- can represent the word\rightarrow counter pairs in a Map
- again, could utilize either Map implementation
- since TreeMap is used, showAll displays words + counts in alphabetical order

```java
import java.util.Map;
import java.util.TreeMap;
import java.util.Scanner;
import java.io.File;

public class WordFreq {
    private Map<String, Integer> words;

    public WordFreq() {
        words = new TreeMap<String, Integer>();
    }

    public WordFreq(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        } catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        String cleanWord = newWord.toLowerCase();
        if (words.containsKey(cleanWord)) {
            words.put(cleanWord, words.get(cleanWord)+1);
        } else {
            words.put(cleanWord, 1);
        }
    }

    public void showAll() {
        for (String str : words.keySet()) {
            System.out.println(str + ": " + words.get(str));
        }
    }
}
```
Graphs

graphs can be simple (w/ bidirectional edges) or directed

```java
import java.util.Set;

public interface Graph<E> {
    public void addEdge(E v1, E v2);
    public Set<E> getAdj(E v);
    public boolean containsEdge(E v1, E v2);
}
```

- can be implemented using an adjacency matrix or an adjacency list

```
@DiGraphMatrix
public class DiGraphMatrix<E> implements Graph<E> {
    private E[] vertices;
    private Map<E, Integer> lookup;
    private boolean[][] adjMatrix;
    public DiGraphMatrix(Collection<E> vertices) {
        this.vertices = (E[])(new Object[vertices.size()]);
        this.lookup = new HashMap<E, Integer>();
        int index = 0;
        for (E nextVertex : vertices) {
            this.vertices[index] = nextVertex;
            lookup.put(nextVertex, index);
            index++;
        }
        this.adjMatrix = new boolean[index][index];
    }
    public void addEdge(E v1, E v2) {
        Integer index1 = this.lookup.get(v1);
        Integer index2 = this.lookup.get(v2);
        if (index1 == null || index2 == null) {
            throw new java.util.NoSuchElementException();
        }
        this.adjMatrix[index1][index2] = true;
    }
...
```
public boolean containsEdge(E v1, E v2) {
    Integer index1 = this.lookup.get(v1);
    Integer index2 = this.lookup.get(v2);
    return index1 != null && index2 != null &&
        this.adjMatrix[index1][index2];
}

going getAdj is more work
- lookup index of vertex
- traverse row
- collect all adjacent vertices in a set

we could implement the
adjacency list using an
array of LinkedLists
- simpler to make use of
  HashMap to map each
  vertex to a Set of
  adjacent vertices
  (wastes some memory,
  but easy to code)
- note that constructor
does not need to know
all vertices ahead of
time
GraphMatrix & GraphList

can utilize inheritance to implement simple graph variants

- can utilize the same internal structures, just add edges in both directions

```java
public class GraphMatrix<E> extends DiGraphMatrix<E> {
    public GraphMatrix(Collection<E> vertices) {
        super(vertices);
    }
    public void addEdge(E v1, E v2) {
        super.addEdge(v1, v2);
        super.addEdge(v2, v1);
    }
}
```

constructors simply call the superclass constructors

```java
addEdge adds edges in both directions using the superclass addEdge
```

```java
public class GraphList<E> extends DiGraphList<E> {
    public GraphList() {
        super();
    }
    public void addEdge(E v1, E v2) {
        super.addEdge(v1, v2);
        super.addEdge(v2, v1);
    }
}
```

Depth-first & Breadth-first searches

DFS utilizes a Stack of unvisited vertices

- results in the longest path being expanded
- as before, uses a Set to keep track of visited vertices & avoid cycles

```java
public static <E> void DFS(Graph<E> g, E v) {
    Stack<E> unvisited = new Stack<E>();
    unvisited.push(v);
    Set visited = new HashSet<E>();
    while (!unvisited.isEmpty()) {
        E nextV = unvisited.pop();
        if (!visited.contains(nextV)) {
            System.out.println(nextV);
            visited.add(nextV);
            for (E adj : g.getAdj(nextV)) {
                unvisited.push(adj);
            }
        }
    }
}
```

BFS is identical except that the unvisited vertices are stored in a Queue

- results in the shortest path being expanded
- similarly uses a Set to avoid cycles

```java
public static <E> void BFS(Graph<E> g, E v) {
    Queue<E> unvisited = new LinkedList<E>();
    unvisited.add(v);
    Set visited = new HashSet<E>();
    while (!unvisited.isEmpty()) {
        E nextV = unvisited.remove();
        if (!visited.contains(nextV)) {
            System.out.println(nextV);
            visited.add(nextV);
            for (E adj : g.getAdj(nextV)) {
                unvisited.add(adj);
            }
        }
    }
```
Stacks & Queues

The `java.util.Stack` class defines the basic operations of a stack:

```java
public class Stack<T> {
    public Stack<T>() { ... }
    T push(T obj) { ... }
    T pop() { ... }
    T peek() { ... }
    boolean isEmpty() { ... }
}
```

The `java.util.Queue` interface defines the basic operations of a queue:

- `LinkedList` implements the `Queue` interface.

```java
public interface Queue<T> {
    boolean add(T obj);
    T remove();
    T peek();
    boolean isEmpty();
}
```

Queue<Integer> numQ = new LinkedList<Integer>();

---

Delimiter matching

```java
import java.util.Stack;

public class DelimiterChecker {
    private static final String DELIMITERS = "{}|\[|\]<>";

    public static boolean check(String expr) {
        Stack<Character> delimStack = new Stack<Character>();
        for (int i = 0; i < expr.length(); i++) {
            char ch = expr.charAt(i);
            if (DelimiterChecker.isLeftDelimiter(ch)) {
                delimStack.push(ch);
            } else if (DelimiterChecker.isRightDelimiter(ch)) {
                if (!delimStack.empty() &&
                    DelimiterChecker.match(delimStack.peek(), ch)) {
                    delimStack.pop();
                } else {
                    return false;
                }
            }
        }
        return delimStack.empty();
    }
}
```

How would you implement the helpers?

- `isLeftDelimiter`
- `isRightDelimiter`
- `match`
Big-Oh and rate-of-growth

Big-Oh classifications capture rate of growth

- for an O(N) algorithm, doubling the problem size doubles the amount of work
  - e.g., suppose Cost(N) = 5N – 3
    - Cost(s) = 5s – 3
    - Cost(2s) = 5(2s) – 3 = 10s - 3

- for an O(N log N) algorithm, doubling the problem size more than doubles the amount of work
  - e.g., suppose Cost(N) = 5N log N + N
    - Cost(s) = 5s log s + s
    - Cost(2s) = 5(2s) log (2s) + 2s = 10s log s + 12s

- for an O(N^2) algorithm, doubling the problem size quadruples the amount of work
  - e.g., suppose Cost(N) = 5N^2 – 3N + 10
    - Cost(s) = 5s^2 – 3s + 10
    - Cost(2s) = 5(2s)^2 – 3(2s) + 10 = 5(4s^2) – 6s + 10 = 20s^2 – 6s + 10

Big-Oh (formally)

an algorithm is O(f(N)) if there exists a positive constant C & non-negative integer T such that for all \( N \geq T \), # of steps required \( \leq C f(N) \)

for example, selection sort:

\[
\frac{N(N-1)}{2} \text{ inspections} + N-1 \text{ swaps} = \left( \frac{N^2}{2} + \frac{N}{2} - 1 \right) \text{ steps}
\]

if we consider \( C = 1 \) and \( T = 1 \), then

\[
\frac{N^2}{2} + \frac{N}{2} - 1 \leq \frac{N^2}{2} + \frac{N}{2} \quad \text{since added 1 to rhs}
\]

\[
\leq \frac{N^2}{2} + N(N/2) \quad \text{since } 1 \leq N \text{ at } T \text{ and beyond}
\]

\[
= \frac{N^2}{2} + \frac{N^2}{2} = N^2 \Rightarrow O(N^2)
\]

in general, can use \( C = \text{sum of positive terms}, T = 1 \) (but other constants work too)
General rules for analyzing algorithms

1. **for loops**: the running time of a for loop is at most
   \[ \text{running time of statements in loop} \times \text{number of loop iterations} \]
   
   ```java
   for (int i = 0; i < N; i++) {
       sum += nums[i];
   }
   ```

2. **nested loops**: the running time of a statement in nested loops is
   \[ \text{running time of statement in loop} \times \text{product of sizes of the loops} \]
   
   ```java
   for (int i = 0; i < N; i++) {
       for (int j = 0; j < M; j++) {
           nums1[i] += nums2[j] + i;
       }
   }
   ```

3. **consecutive statements**: the running time of consecutive statements is
   \[ \text{sum of their individual running times} \]
   
   ```java
   int sum = 0;
   for (int i = 0; i < N; i++) {
       sum += nums[i];
   }
   double avg = (double)sum/N;
   ```

4. **if-else**: the running time of an if-else statement is at most
   \[ \text{running time of the test} + \text{maximum running time of the if and else cases} \]
   
   ```java
   if (isSorted(nums)) {
       index = binarySearch(nums, desired);
   } else {
       index = sequentialSearch(nums, desired);
   }
   ```
EXAMPLE: finding all anagrams of a word (approach 1)

for each possible permutation of the word
  • generate the next permutation
  • test to see if contained in the dictionary
  • if so, add to the list of anagrams

efficiency of this approach, where \( L \) is word length & \( D \) is dictionary size?

for each possible permutation of the word
  • generate the next permutation
    \( \rightarrow O(L) \), assuming a smart encoding
  • test to see if contained in the dictionary
    \( \rightarrow O(D) \), assuming sequential search
  • if so, add to the list of anagrams
    \( \rightarrow O(1) \)

\[ O(L! \times (L + D + 1)) \rightarrow O(L! \times D) \]

\[
5! \times 117,000 = 120 \times 117,000 = 14,040,000 \\
10! \times 117,000 = 3,628,800 \times 117,000 = 424,569,600,000
\]

EXAMPLE: finding all anagrams of a word (approach 2)

sort letters of given word
traverse the entire dictionary, word by word
  • sort the next dictionary word
    \( \rightarrow O(L \log L) \), assuming an efficient sort
  • test to see if identical to sorted given word
    \( \rightarrow O(L) \)
  • if so, add to the list of anagrams
    \( \rightarrow O(1) \)

\[ O(L \log L + (D \times (L \log L + L + 1))) \rightarrow O(L \log L \times D) \]

\[
5 \log 5 \times 117,000 \approx 12 \times 117,000 = 1,404,000 \\
10 \log 10 \times 117,000 \approx 33 \times 117,000 = 3,861,000
\]
Analyzing recursive algorithms

recursive algorithms can be analyzed by defining a recurrence relation:

cost of searching N items using binary search =
cost of comparing middle element + cost of searching correct half (N/2 items)

more succinctly: Cost(N) = Cost(N/2) + C

Cost(N) = Cost(N/2) + C
= (Cost(N/4) + C) + C
can unwind Cost(N/2)
= Cost(N/4) + 2C
can unwind Cost(N/4)
= (Cost(N/8) + C) + 2C
can unwind Cost(N/4)
= Cost(N/8) + 3C
can continue unwinding
... (a total of log₂ N times)
= Cost(1) + (log₂ N)*C
= C log₂ N + C'
where C' = Cost(1)

⇒ O(log N)