CSC 421: Algorithm Design and Analysis

Spring 2014

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

cohesion 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

coupling 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

```
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

```
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

3

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, …`

```
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;
}
```

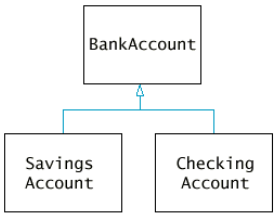
```
ArrayList<String> words1 = new ArrayList<String>();
…
int num1 = this.longest(words1);


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


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

4

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.nextNumber;
      this.nextNumber++;
  }

  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;
      }
  }
}
```

5

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);
  }
}
```

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

```
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;
  }
}
```

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

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


BankAccount acct1 = new BankAccount();
…
showAccount(acct1);

SavingsAccount acct2 = new SavingsAccount(3.0);
…
showAccount(acct2);

CheckingAccount acct3 = new CheckingAccount();
…
showAccount(acct3);
```

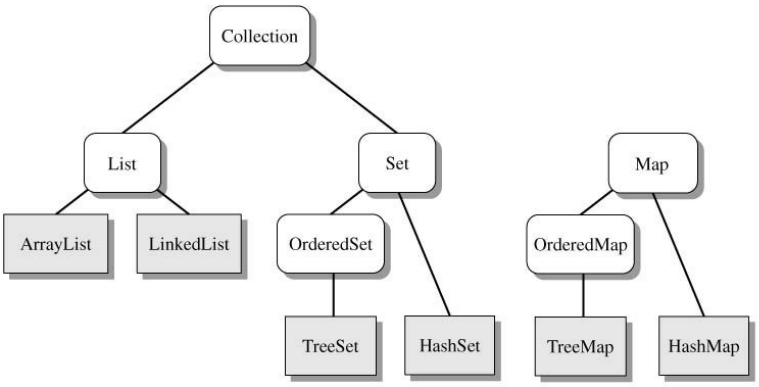
7

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



8

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

9

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)

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private Set<String> words;

    public Dictionary() {
        this.words = new TreeSet<String>();
    }

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

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

10

Maps

java.util.Map interface: a collection of key → value mappings

```
public interface Map<K, V> {
    boolean put(K key, V value);   // adds key→value to Map
    V remove(Object key);          // removes key→? 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 containsValue(Object value); // returns true if value 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 utlizes a HashSet to store key/value pairs
- ✓ HashSet provides O(1) put, get, and containsKey (on average, but can degrade)

11

Word frequencies

a variant of Dictionary is WordFreq

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

```
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() {
        this.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);
            }
            infile.close();
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

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

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

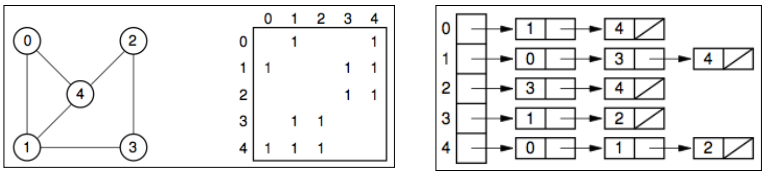
12

Graphs

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

```
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



13

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;
    }

    ...
```

to create adjacency matrix, constructor must be given a list of all vertices

however, matrix entries are not directly accessible by vertex name

- index → vertex requires `vertices` array
- vertex → index requires `lookup` map

14

DiGraphMatrix

```
    ...

    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];
    }

    public Set<E> getAdj(E v) {
        int row = this.lookup.get(v);

        Set<E> neighbors = new HashSet<E>();
        for (int c = 0; c < this.adjMatrix.length; c++) {
            if (this.adjMatrix[row][c]) {
                neighbors.add(this.vertices[c]);
            }
        }
        return neighbors;
    }
}
```

`containsEdge` is easy
- lookup indices of vertices
- then access row/column

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

15

DiGraphList

```
public class DiGraphList<E> implements Graph<E>{
    private Map<E, Set<E>> adjLists;

    public DiGraphList() {
        this.adjLists = new HashMap<E, Set<E>>();
    }

    public void addEdge(E v1, E v2) {
        if (!this.adjLists.containsKey(v1)) {
            this.adjLists.put(v1, new HashSet<E>());
        }
        this.adjLists.get(v1).add(v2);
    }

    public Set<E> getAdj(E v) {
        if (!this.adjLists.containsKey(v)) {
            return new HashSet<E>();
        }
        return this.adjLists.get(v);
    }

    public boolean containsEdge(E v1, E v2) {
        return this.adjLists.containsKey(v1) &&
               this.getAdj(v1).contains(v2);
    }
}
```

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

16

GraphMatrix & GraphList

can utilize inheritance to implement simple graph variants

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

```
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

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

```
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);
    }
}
```

17

Depth-first & Breadth-first searches

```
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);
            }
        }
    }
}
```

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

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

```
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);
            }
        }
    }
}
```

18

Stacks & Queues

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

```
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

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

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

19

Delimiter matching

```
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`

20

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)+1) + 2s = 10s log s + 12s

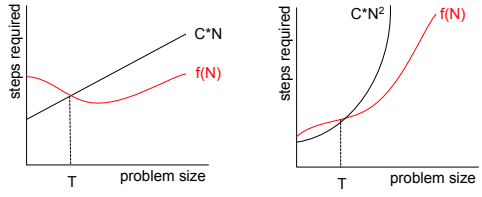
- for an O(N^2) algorithm, doubling the problem size quadruples the amount of work

  e.g., suppose Cost(N) = $5N^2 – 3N + 10$
  - Cost(s) = $5s^2 – 3s + 10$
  - Cost(2s) = $5(2s)^2 – 3(2s) + 10 = 5(4s^2) – 6s + 10 = 20s^2 – 6s + 10$

21

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 ≥ T, # of steps required ≤ C*f(N)



for example, selection sort:

N(N-1)/2 inspections + N-1 swaps = ($N^2/2$ + N/2 -1) steps

if we consider C = 1 and T = 1, then

$N^2/2 + N/2 - 1 \leq N^2/2 + N/2$ *since added 1 to rhs*

$N^2/2 + N/2 \leq N^2/2 + N(N/2)$ *since 1 ≤ N at T and beyond*

$N^2/2 + N(N/2) = N^2/2 + N^2/2 = N^2$

by transitivity, $N^2/2 + N/2 – 1 \leq 1N^2$ ➔ $O(N^2)$

in general, can use C = sum of positive terms, T = 1 (if log terms, then T = 2)

22

General rules for analyzing algorithms

1. for loops: the running time of a for loop is at most
   running time of statements in loop × number of loop iterations

```
for (int i = 0; i < N; i++) {
    sum += nums[i];
}
```

2. nested loops: the running time of a statement in nested loops is
   running time of statement in loop × product of sizes of the loops

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        nums1[i] += nums2[j] + i;
    }
}
```


General rules for analyzing algorithms

3. consecutive statements: the running time of consecutive statements is
   sum of their individual running times

```
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
   running time of the test + maximum running time of the if and else cases

```
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
            → O(L), assuming a smart encoding
    • test to see if contained in the dictionary
            → O(D), assuming sequential search
    • if so, add to the list of anagrams
            →O(1)
```

since L! different permutations, will loop L! times

➔O(L! × (L + D + 1)) ➔ O(L! × D)

5! × 117,000 = 120 × 117,000 = 14,040,000
10! × 117,000 = 3,628,800 × 117,000 = 424,569,600,000

25

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
    • test to see if identical to sorted given word
    • if so, add to the list of anagrams
```

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

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

since dictionary is size D, will loop D times

➔O(L log L + (D × (L log L + L + 1))) ➔ O(L log L × D)

5 log 5 × 117,000 ≈ 12 × 117,000 = 1,404,000
10 log 10 × 117,000 ≈ 33 × 117,000 = 3,861,000

26

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(S) = Cost(S/2) + C	can unwind Cost(S/2)
= (Cost(S/4) + C) + C	
= Cost(S/4) + 2C	can unwind Cost(S/4)
= (Cost(S/8) + C) + 2C	
= Cost(S/8) + 3C	can continue unwinding
= …	(a total of $\log_2 S$ times)
= Cost(S/S) + ($\log_2$S)*C	
= Cost(1) + ($\log_2$S)*C	
= C $\log_2$ S + C'	where C' = Cost(1)
➔ O(log S)	

27