1004 Final Review

Wednesday, May 7, 2014

Final Exam Information

- Two dates
  - Friday, May 9, 4:10 - 7:00 PM in 301 Pupin
  - Monday, May 12, 4:10 - 7:00 PM in 417 IAB
  - Sign-up email sent out last week, deadline was this past Monday
- Exam will be cumulative
  - Vocabulary and multiple choice, like midterm
  - Will cover material over the entire semester
  - Anything covered in class or homeworks is fair game
  - Relevant chapters
    - Schneider & Gersting: Chapters 1 - 5, 7, 12
    - Horstmann: Chapters 1 - 11, look over 12

Topics in this review

There's a lot of them and we may not get to all of them, so we'll let you choose what we go over first! Here's what we have:

- Algorithms and analysis (sorting algorithms, time complexity)
- Java: data types (primitives, Strings, expressions)
- Java: loops and conditionals (if, while, for statements)
- Java: classes and methods (overview)
- Java: objects and object oriented programming
- Java: debugging code
- Java: Exceptions and I/O
- Java: Inheritance and interfaces
- Binary and hexadecimal numbers (convert to/from decimal, etc.)
- Boolean logic (truth tables, logic gates)
- Computer systems (Von Neumann, memory, machine code)
- Networks: LAN/WAN, protocol stack, collisions, ARQ
- Networks: Dijkstra's algorithm
- Turing machines

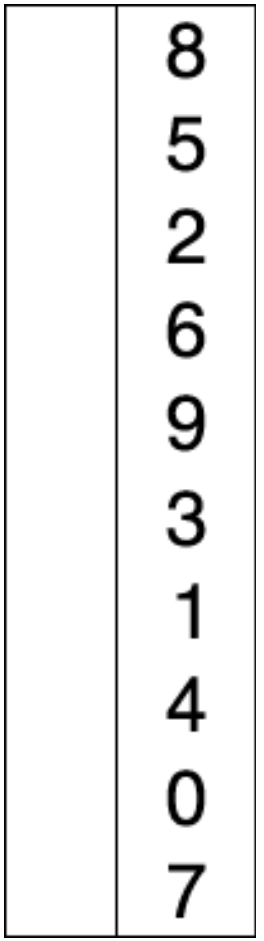
Algorithms & Analysis

- Algorithms are analyzed in Big-O notation (at least in this class)
- Takes into account worst-case performance
- Measures how much of a resource is required (i.e. operations, space) with an increasing number of inputs
- The final answer takes the fastest-growing term as the complexity

Example: Selection Sort

Algorithm:

- Iterate through list, and find minimum
- place minimum in “sorted section” of list
- repeat until the sorted section is the length of the list.



source: wikipedia

Time Complexity

- For one iteration: as many items are in the list.
- For each subsequent iteration, there is one less element in the list

Time Taken: $$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Big-O time complexity - $O(n^2)$

Bubble Sort

- Abstract idea: each iteration, the largest element “bubbles” to the top of the list
- Mechanism: switching two adjacent elements.
- Each element moves forward in the list until it encounters something bigger, in which case that element starts moving forward.

6 5 3 1 8 7 2 4

source: wikipedia

Insertion Sort

Algorithm:

- Maintain a “sorted section” of the list
- Each iteration, add the first element of the unsorted list to the sorted list, by iterating through and seeing where the correct position is.

6 5 3 1 8 7 2 4

source: wikipedia

Java: data types

8 primitive data types:

TYPE NAME	SIZE IN MEMORY	RANGE	USAGE
`byte`	1 byte (8 bits)	-2^7 to $+2^7 - 1$	`byte b = 16;`
`short`	2 bytes (16 bits)	-2^{15} to $+2^{15} - 1$	`short s = 399;`
`int`	4 bytes (32 bits)	-2^{31} to $+2^{31} - 1$	`int i = 1234;`
`long`	8 bytes (64 bits)	-2^{63} to $+2^{63} - 1$	`long l = 8675309;`
`float`	4 bytes (32 bits)	$\sim\pm10^{38}$, ~7 sig. digits	`float f = 13.37f;`
`double`	8 bytes (64 bits)	$\sim\pm10^{308}$, ~15 sig. digits	`double d = 3.14159;`
`char`	2 bytes (16 bits)	Any unicode character	`char c = '$';`
`boolean`	1 bit	**true** or **false**	`boolean h = true;`

Java: data types

Binary operators: *, /, %, +, -

- *, /, and % operations done before +, -
- All operators are left associative, e.g. 12 / 3 * 2 is (12 / 3) * 2
- If one operand is an integer data type and the other is a floating point data type, the result will be a floating point number, e.g. int * double → double

Integer division

- Division works as expected if at least one of the two operands is a floating point number
  - `7.0 / 4.0 == 7 / 4.0 == 7.0 / 4 == 1.75`
- If both operands are integer types, the fractional part is discarded
  - `7 / 4 == 1`
  - Use mod (%) to get remainder: `7 % 4 == 3`
  - To get floating point result, cast one operand to a floating point type

```
int m = 9, n = 5;
int intResult = m / n; // will be 1
double doubleResult = (double) m / n; // will be 1.8
```

Java: data types

Strings

- There will usually be a question about Strings
- The shortest String is length zero, i.e. empty quotes “”
- Concatenation
  - Strings can be combined with other Strings as well as some other data types
    - ```
      String name = “Bob”;
      double age = 17.5;
      String info = name + “: ” + age + “ ys”; // will be “Bob: 17.5 ys”
      ```
  - If a String is involved in an expression, the whole expression evaluates to a String
  - Concatenation follows order of operations
    - `System.out.print(“bob” + 8 + 2); // will print “bob82”`
    - `System.out.print(“bob” + (8 + 2)); // will print “bob10”`
    - `System.out.print(“bob” + 8 * 2); // will print “bob16”`
    - `System.out.print(“bob” + 8 - 2); // ERROR! cannot use - between String and int`
- Strings are *immutable*!
  - Once created, their contents cannot be changed
  - Methods like `s.replace(String targ, String repl)` and `s.toUpperCase()` return new Strings and do not alter the original String
  - String lengths cannot change -- concatenation produces a new String, does *not* make the original String longer
      ```

Java: data types

Example question:

- Given the following variables, what is the result of the expression below?
  - ```
    int m = 6, n = 4;
    double x = 2.5, y = 1.25;
    ```
  - Expression: `x + (m / n) * y`
  - (a) 3.75
  - (b) 4.25
  - (c) 5
  - (d) 4.375
    ```

Java: data types

Example question:

- Given the following variables, what is the result of the expression below?
  - ```
    int m = 6, n = 4;
    double x = 2.5, y = 1.25;
    ```
  - Expression: `x + (m / n) * y`
  - **(a) 3.75**
  - (b) 4.25
  - (c) 5
  - (d) 4.375
    ```

Java: conditionals & loops

- if statements
- for loops
- while loops
- do-while loops

Conditionals and loops allow you to perform certain actions *if* a specified condition is true (for if statements) or *while/as long as* a specified condition is true (for loops)

if(condition)
{
//do something once
}
	for(initial counter value; condition; change in counter value)
	{
	//do something
	}
		while(condition)
		{
		//do something repeatedly
		}
			do(condition)
			{
			//do something repeatedly, at least once
			}

Remember that the condition should always be a boolean expression (something that evaluates to true or false).

Java: conditionals & loops

property	**for loops**	**while loops**	**do while loops**
number of iterations	until counter's value meets the condition	until condition is met	until condition is met
zero iterations possible?	yes	yes	**no**
infinite iterations possible?	yes	yes	yes
when should you use it?	when you want a 'counter' variable to step through a series of values	when you don't know when the condition will be met, so you don't know how many times the loop will run at most	when you don't know when the condition will be met, **but you want the loop to run at least once**
example	1) print the first 11 multiples of 3 2) find the sum of all numbers in an array	1) keep adding consecutive numbers until you reach a certain goal	1) display instructions for a game to the user, only giving them the option to quit after seeing the instructions once

Java: conditionals & loops

Enhanced for loop

- allows you to easily iterate through all elements in an array, arraylist, etc. without worrying about counters or how big the list is
- uses a single variable name that's defined in the for loop; within the loop, that variable name can be used to refer to the current element of the array that the for loop is iterating over
- in the given example, `myName` is the variable name you can use to refer to the current element
- printing `myName` inside the loop would print every element of `names`

Syntax:

```
for(type variableName: arrayName)
{
//type should match the type of
element stored in the array
}
```

Example:

```
//names is an array of Strings
for(String myName: names)
{
//do something with myName here
}
```

Java: conditionals & loops

- At least one question dealing with your understanding of loops
- Example qs: How many times does this loop run? What is its output? etc.

Practice Questions

```
for(int i=1; i<=100; i = i*2)
{
    System.out.println(i);
}
```

Which of these does this loop print?

a) 24 b) 9 c) 64 d) 128

```
for(String word: allWords)
{
    if(word.length()==4)
        System.out.println(word);
}
```

Which of these is **false**?

a) allWords is a String[]

b) allWords is an int[]

c) The for loop might not print anything

d) allWords is an ArrayList<String>

Java: conditionals & loops

- At least one question dealing with your understanding of loops
- Example qs: How many times does this loop run? What is its output? etc.

Practice Questions

```
for(int i=1; i<=100; i = i*2)
{
    System.out.println(i);
}
```

Which of these does this loop print?

a) 24 b) 9 **c) 64** d) 128

```
for(String word: allWords)
{
    if(word.length()==4)
        System.out.println(word);
}
```

Which of these is **false**?

a) allWords is a String[]

b) allWords is an int[]

c) The for loop might not print anything

d) allWords is an ArrayList<String>

Java: Classes

- Textbook: abstract data types
- Can think of them as representations of real-world things
- Everything in Java is a class - this can be confusing
- Instance Variables: adjectives
- Methods: verbs

Java: Methods

- Access modifier
- Static vs. non-static
- return type
- parameters

- Passing Parameters

Java: OO Design

- Code reuse
- Encapsulation
- Polymorphism (inheritance and interfaces)

EVERYTHING IS AN OBJECT.

Java: debugging

Two main types of bugs:

- Compile-time errors
  - Any error that would prevent your program from compiling
  - Primarily syntax errors
  - Missing semicolons, missing quotes around a String, improper method calls, unbalanced parentheses, missing imports, etc
- Run-time errors
  - Will get past the compiler, but can cause the program to crash or simply be logic errors
  - `System.out.println(1 / 0);` will compile, but will cause the program to crash with an `ArithmeticException: / by zero`
  - Logic errors: incorrect or unexpected output, harder to find and deal with
    - Can usually only be discovered by the programmer
    - Ex: `System.out.print("The sum is " + x + y);` will *concatenate* x and y, not add them (recall order of operations and String concatenation)
- Style and design issues are NOT bugs!

Java: debugging

See if you can find all the bugs in the familiar PhoneNumber class below! Hints: There are compile-time *and* run-time bugs in this class (more than 6 of them!). The compile-time bugs (and some run-time bugs) are usually easy to spot. Other run-time bugs require intuition about what the class is supposed to do. Use the tester class for context!

```
1   //*****************************************
2   // PhoneNumber.java
3   // Written by Cannon
4   // Objects of this class determine whether a
5   // phone number is a valid Manhattan Number
6   // THIS CLASS CONTAINS BUGS
7   //*****************************************
8
9   public class PhoneNumber {
10
11      private int number;
12      private boolean valid;
13
14      public PhoneNumber(String n){
15          number=n;
16          valid=false;
17      }
18
19      private void check1() {
20          int areaCode=Integer.parseInt(number.substring(1,3));
21          if ((areaCode==212 || areaCode==917 || areaCode==646))
22              valid=false;
23      }
24
25      private void check2() {
26          int first=Integer.parseInt(number.substring(4,5));
27          if (first==0 or first=1)
28              valid=false;
29      }
30
31      private void check() {
32          check1();
33          check2();
34      }
35      // This is an accessor method for the IV valid
36       public void isValid() {
37           return valid;
38       }
39
40  }
```

Here is the test class to show how the PhoneNumber class works. ***It is bug-free!***

```
1   import java.util.Scanner;
2
3   public class PhoneNumberTester {
4
5       public static void main(String[] args) {
6
7           Scanner input = new Scanner(System.in);
8           System.out.println("Please enter a number xxx-xxx-xxxx");
9           String n=input.next();
10
11          // Now we instantiate the PhoneNumber and check it's validity
12          PhoneNumber number=new PhoneNumber(n);
13          number.check();
14          if (number.isValid())
15              System.out.println("The number is valid.");
16          else
17              System.out.println("The number is not valid.”);
18
19      }
20  }
```

Java: debugging

See if you can find all the bugs in the familiar PhoneNumber class below! Hints: There are compile-time *and* run-time bugs in this class (more than 6 of them!). The compile-time bugs (and some run-time bugs) are usually easy to spot. Other run-time bugs require intuition about what the class is supposed to do. Use the tester class for context!

```
1   //*****************************************
2   // PhoneNumber.java
3   // Written by Cannon
4   // Objects of this class determine whether a
5   // phone number is a valid Manhattan Number
6   // THIS CLASS CONTAINS BUGS
7   //*****************************************
8
9   public class PhoneNumber {
10
11      private int number;
12      private boolean valid;
13
14      public PhoneNumber(String n){
15          number=n;
16          valid=false;
17      }
18
19      private void check1() {
20          int areaCode=Integer.parseInt(number.substring(1,3));
21          if ((areaCode==212 || areaCode==917 || areaCode==646))
22              valid=false;
23      }
24
25      private void check2() {
26          int first=Integer.parseInt(number.substring(4,5));
27          if (first==0 or first=1)
28              valid=false;
29      }
30
31      private void check() {
32          check1();
33          check2();
34      }
35      // This is an accessor method for the IV valid
36      public void isValid() {
37          return valid;
38      }
39
40  }
```

Here is the test class to show how the PhoneNumber class works. ***It is bug-free!***

```
1   import java.util.Scanner;
2
3   public class PhoneNumberTester {
4
5       public static void main(String[] args) {
6
7           Scanner input = new Scanner(System.in);
8           System.out.println("Please enter a number xxx-xxx-xxxx");
9           String n=input.next();
10
11          // Now we instantiate the PhoneNumber and check its validity
12          PhoneNumber number=new PhoneNumber(n);
13          number.check();
14          if (number.isValid())
15              System.out.println("The number is valid.");
16          else
17              System.out.println("The number is not valid.”);
18
19      }
20  }
```

Java: Exceptions and I/O

- Largest Issue: design.

I/O and exception handling should generally be done in one class, typically your tester/driver class

- Note: checked vs. unchecked exceptions - I/O typically nee*ds* you to handle exceptions

Java: Exceptions and I/O

Throw, throws and try/catch

- Use **throws** to acknowledge that a method could throw an Exception. Throws tells the program to have whatever code that *called* the method deal with any Exceptions.
  - ```
    public void readFile(String filename) throws FileNotFoundException {
        Scanner read = new Scanner(new File(filename));
        while (read.hasNextLine()) System.out.println(read.nextLine());
    }
    ```
- Use **throw** to manually throw an Exception within your code.
  - ```
    public void checkFileName(String filename) {
        if (!filename.equals("somefile.txt")) {
            throw new WrongFileNameException("Wrong filename provided!");
        }
    }
    ```
- Use **try/catch** statements to ultimately handle thrown Exceptions. A **finally** statement will always run whether an Exception occurs or not.
  - ```
    try {
        readFile("somefile.txt");
    } catch (FileNotFoundException e) {
        System.out.println("Could not find the file specified!");
    } finally {
        System.out.println("file read was attempted");
    }
    ```
    ```

Java: inheritance

- Allows you to model parent-child or category-subtype relationships, just like in the real world
- Motivation: **code reuse.** If one class 'extends' another class, it inherits methods and instance variables from that class so you don't have to write them again!

What's inherited?

- only **public and protected methods and instance variables**
- that means that the child class **can't make any references to private methods or instance variables of its parents**

What else?

- child classes can define their own additional methods - these **cannot** be accessed by the parent classes.
- child classes can also **override** their parents' methods

```
public class Animal {
    private String name;
    public Animal(String n) {
        name = n;
    }
    public void makeSound()
    {
        System.out.print("Hi!");
    }
}
public class Cat extends Animal {
    public Cat(String n)
   {
        super(n);
    }
    public void makeSound()
    {
        System.out.print("Meow");
    }
    public void scratch()
    {
    //stuff
    }
}
```

Java: inheritance

Reference types vs. Object types

Cat	myCat =	new	Cat(“Snowball”)
Reference type	variable name		**Object type**
Animal	myAnimal =	new	Cat(“Grumpy”)

The **reference type** determines **what** methods can be called.

The **object type** determines **which version** of the method is called.

The **object type** must be the same as the reference type **or be a child of it**

So…

- myCat.scratch() **compiles** but myAnimal.scratch() **won’t**
- myCat.makeSound() prints “meow” and myAnimal. makeSound() **also prints “meow”**

```
public class Animal {
    private String name;
    public Animal(String n) {
        name = n;
    }
    public void makeSound()
    {
        System.out.print(“Hi!”);
    }
}
public class Cat extends Animal {
    public Cat(String n)
   {
        super(n);
    }
    public void makeSound()
    {
        System.out.print(“Meow”);
    }
    public void scratch()
    {
    //stuff
    }
}
```

Java: inheritance

Sample Questions

Which of these can Cat access?

a) makeSound()

b) scratch()

c) name

Which version of makeSound() is called for each of these objects?*

a) Animal myAnimal = new Animal(“test”);

b) Animal myPet = new Cat(“kitty”);

c) Cat myKitty = new Cat(“another kitty”);

d) Cat myCat = new Animal(“an animal”);

```
public class Animal {
    private String name;
    public Animal(String n) {
        name = n;
    }
    public void makeSound()
    {
        System.out.print(“Hi!”);
    }
}
public class Cat extends Animal {
    public Cat(String n)
   {
        super(n);
    }
    public void makeSound()
    {
        System.out.print(“Meow”);
    }
    public void scratch()
    {
    //stuff
    }
}
```

Java: inheritance

Sample Questions

Which of these can Cat access?

a) makeSound()

b) scratch()

c) name

Which version of makeSound() is called for each of these objects?*

a) Animal myAnimal = new Animal(“test”); **A**

b) Animal myPet = new Cat(“kitty”); **C**

c) Cat myKitty = new Cat(“another kitty”); **C**

d) Cat myCat = new Animal(“an animal”); **X (doesn’t compile)**

```
public class Animal {
    private String name;
    public Animal(String n) {
        name = n;
    }
    public void makeSound()
    {
        System.out.print(“Hi!”);
    }
}
public class Cat extends Animal {
    public Cat(String n)
   {
        super(n);
    }
    public void makeSound()
    {
        System.out.print(“Meow”);
    }
    public void scratch()
    {
    //stuff
    }
}
```

Java: interfaces

- Interfaces allow you to define strict templates for classes
- In Java, classes can only ‘extend’ one other class, but they can **implement** multiple interfaces: interfaces allow multiple inheritance
- If a class implements an interface it **must** implement all its methods
- Interfaces can be reference types too
- If the interface is a reference type, the only methods that object can call are the ones in the interface
- Interfaces **aren’t classes,** so you can’t create objects of interfaces

```
public interface DomesticPet
{
    void play();
    void feed();
}

public class HouseCat extends
Animal implements DomesticPet
{
//all the methods from Animal are
already inherited
    public void play()
    {
    //do stuff
    }
    public void feed()
    {
    // do more stuff
    }
}
```

Binary/hexadecimal numbers

Convert binary to decimal

- Easiest with an example: `1011.101`
- $(\mathbf{1}\times 2^{3}) + (\mathbf{0}\times 2^{2}) + (\mathbf{1}\times 2^{1}) + (\mathbf{1}\times 2^{0}) + (\mathbf{1}\times 2^{-1}) + (\mathbf{0}\times 2^{-2}) + (\mathbf{1}\times 2^{-3})$
- = 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125
- = 11.625

Decimal to binary

- Find the highest possible power of 2 less than the number and subtract
- Repeat this for each decreasing power of two, marking a 1 in each position where subtraction is possible, and a 0 where it is not (i.e. you'd get a negative value)
- Example: 13.25
- 13 - 8 = **5** (add a *1*), **5** - 4 = **1** (add a *1*), can't do **1** - 2 (add a *0*), **1** - 1 = **0** (add a *1*), for a whole number part of **1101**
- For the decimal, we have 0.25, which is $(\mathbf{0}\times 2^{-1}) + (\mathbf{1}\times 2^{-2})$, or .01
- Result: `1101.01` (we won't ask you to do a conversion with a "weird" fraction)

Hexadecimal

- Same process, using powers of 16 instead of two (digits are 0 to F)
- Note: 1 hex digit can be represented with 4 binary digits. To convert hex to binary, simply separate each digit of the hex number, convert each to a 4-digit binary number, and recombine.
  - Example: 2AF: 2 → 0010 | A → 1010 | F → 1111, so 2AF → 001010101111 (or 687)

Boolean logic

- Two states: TRUE and FALSE
- Three operations: AND, OR, NOT.
- Important to note: NOT (A and B) is not the same as NOT A and NOT B. (DeMorgan's Laws)
- Example: For what values of P and Q is the following expression *false*?

  (NOT (P and Q) and NOT(P or NOT Q))

  a) P = T, Q = T
  b) P = T, Q = F
  c) P = F, Q = F
  d) All of the above

Boolean logic

- Two states: TRUE and FALSE
- Three operations: AND, OR, NOT.
- Important to note: NOT (A and B) is not the same as NOT A and NOT B. (DeMorgan's Laws)
- Example: For what values of P and Q is the following expression *false*?

  (NOT (P and Q) and NOT(P or NOT Q))

a) P = T, Q = T
b) P = T, Q = F
c) P = F, Q = F
d) All of the above

P	Q	P ^ Q	NOT (P ^ Q)	P v NOT Q	NOT (P v NOT Q)	(NOT (P ^ Q) and NOT (P v NOT Q))
1	1	1	0	1	0	0
1	0	0	1	1	0	0
0	1	0	1	0	1	1
0	0	0	1	1	0	0

We see that the expression is only true for P=False and Q=True

Boolean logic

Logic gates

- AND gate: 3 transistors
- OR gate: 3 transistors
- NOT gate: 1 transistor

Example: For a circuit with 2 AND gates, 2 NOT gates, and 5 OR gates, how many transistors will there be?

a) 9

b) 23

c) 17

d) 15

Boolean logic

Logic gates

- AND gate: 3 transistors
- OR gate: 3 transistors
- NOT gate: 1 transistor

Example: For a circuit with 2 AND gates, 2 NOT gates, and 5 OR gates, how many transistors will there be?

a) 9

b) 23 (2 * 3 + 2 * 1 + 5 * 3 = 23)

c) 17

d) 15

Computer systems

The Von Neumann (pronounced “von NOY-mann”) architecture

- Three characteristics
  - Four major subsystems: **primary memory**, **input/output**, the **ALU (arithmetic logic unit)**, and the **control unit**
  - The **stored program concept**: instructions are represented as binary values stored in memory
  - The **sequential execution of instructions**: One instruction at a time is fetched from memory and passed to the control unit to be decoded and executed

Computer systems

Memory and cache

- Memory: stores and retrieves instructions and data
  - Consists of cells identified by addresses, that each hold a certain number of bits
  - Two memory registers
    - Memory Address Register (MAR): holds the address of a cell to be fetched or stored
    - Memory Data Register (MDR): holds the actual data value fetched from MAR, or to be stored into MAR
- Cache memory: smaller, faster memory in the processor
  - Data recently fetched from memory is copied to cache for faster future access
  - When performing a fetch, a computer will check cache memory before RAM
- Average memory access time = `(cache hit rate) x (cache access time) + (cache miss rate) x ((cache access time) + (RAM access time))`

Computer systems

ALU

- Consists of circuitry for performing operations and registers for storing operands and results

Control unit

- Machine language instructions:
  - OP CODE ADDRESS 1 ADDRESS 2 . . .
  - Example: Assume 8-bit op code and 16-bit memory addresses
    - ADD 99 100
    - 00001001 0000000001100011 0000000001100100
  - The op code and each address field use a certain number of bits, which determine how many unique instructions and addresses there can be in the language
  - Example: If the op code is 6 bits long, what is the maximum number of possible instructions? (Ans: 2^6 = 64 instructions)

Networks

Network:

- A set of independent computers (nodes) connected by links to transfer information

Local Area Networks (LAN)

- Connects hardware in close proximity
- Various network configurations: bus, ring, star, etc.

Wide Area Network (WAN)

- Nodes not in close proximity
- Many dedicated point-to-point links
- Store-and-forward, packet-switched technology
  - Packets must hop between different nodes to get to a destination
  - Advantages:
    - Packets allow large messages to be broken up and sent through the network quickly
    - Failure of one node or line does not necessarily bring down entire network

Networks

The Internet protocol stack: 5 layers

- *5* Application layer (Protocols: HTTP, FTP, etc.)
- *4* Transport layer (Protocols: TCP, UDP)
- *3* Network layer (Protocols: IP)
- *2* Data Link layer (Protocols: PPP, Ethernet)
- *1* Physical layer (Protocols: DSL, Cable, 4G, etc.)

Most important to remember what these layers are (as well as their order)

Too many slides to go over all of these in depth, but review what each layer is responsible for (Schneider 7.3)

Networks

Collisions (Data Link layer)

- Two or more messages try to get delivered on the same line
- Ethernet protocol dictates that both nodes wait a random amount of time and then retransmit, earlier node will get the line

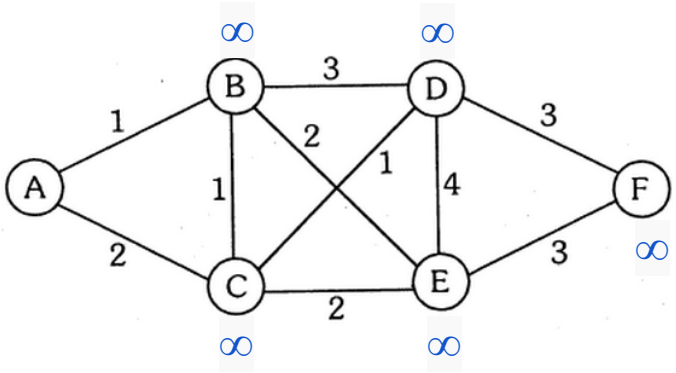
ARQ (automatic repeat request) algorithm

- Ensures that messages arrive at their destination
  - *A* sends packet to *B*
  - If *B* successfully receives the packet, sends ACK to *A*
  - If *B* does not receive the packet (or *A* does not receive the ACK), *A* will wait some arbitrary amount of time and resend
  - *B* will automatically discard any of *A*'s mistakenly resent packets

Networks

Dijkstra's shortest path algorithm (goal: A to F)

	0	1	2	3	4	5	6
A	0						
B	∞						
C	∞						
D	∞						
E	∞						
F	∞						

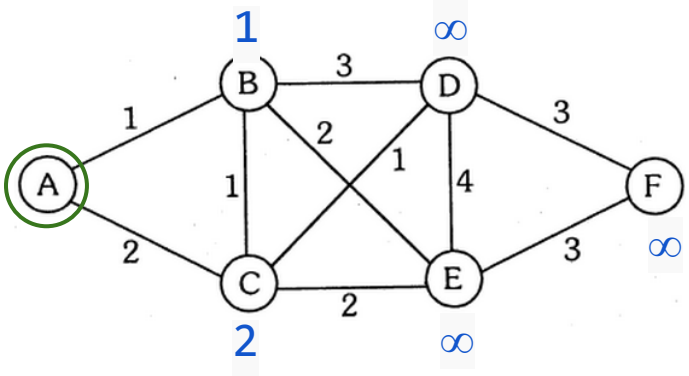


$S = \{\ \}$

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-					
B_A	∞	1 (A)					
C_A	∞	2 (A)					
D	∞	∞					
E	∞	∞					
F	∞	∞					

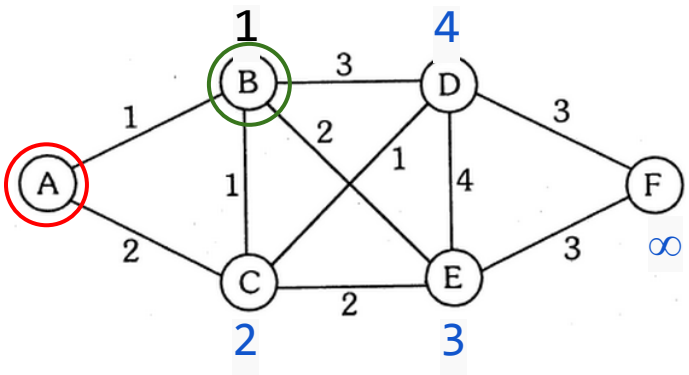


$S = \{ A \}$

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-	-				
B_A	∞	1 (A)	-				
$C_{A, B}$	∞	2 (A)	2 (A, B)				
D_B	∞	∞	4 (B)				
E_B	∞	∞	3 (B)				
F	∞	∞	∞				

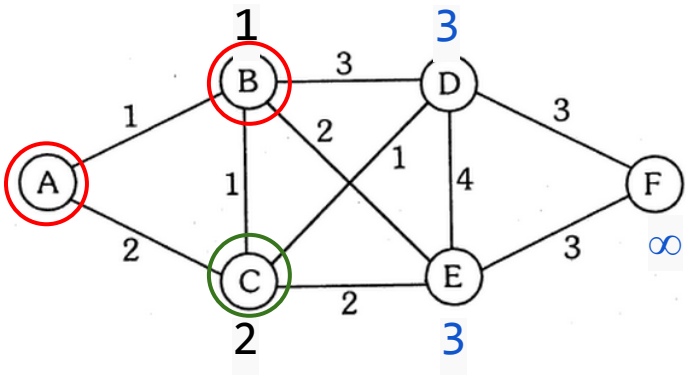


S = { A, B }

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-	-	-			
B_A	∞	1 (A)	-	-			
$C_{A, B}$	∞	2 (A)	2 (A, B)	-			
D_C	∞	∞	4 (B)	3 (C)			
E_B	∞	∞	3 (B)	3 (B)			
F	∞	∞	∞	∞			

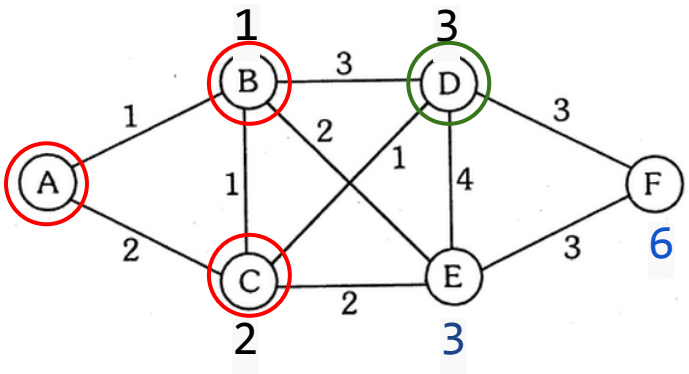


S = { A, B, C }

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-	-	-	-		
B_A	∞	1 (A)	-	-	-		
$C_{A, B}$	∞	2 (A)	2 (A, B)	-	-		
D_C	∞	∞	4 (B)	3 (C)	-		
E_B	∞	∞	3 (B)	3 (B)	3 (B)		
F_D	∞	∞	∞	∞	6 (D)		

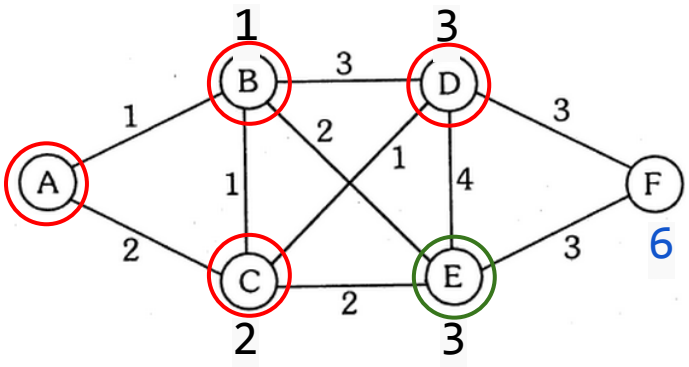


S = { A, B, C, D }

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-	-	-	-	-	
B_A	∞	1 (A)	-	-	-	-	
$C_{A, B}$	∞	2 (A)	2 (A, B)	-	-	-	
D_C	∞	∞	4 (B)	3 (C)	-	-	
E_B	∞	∞	3 (B)	3 (B)	3 (B)	-	
$F_{D, E}$	∞	∞	∞	∞	6 (D)	6 (D, E)	

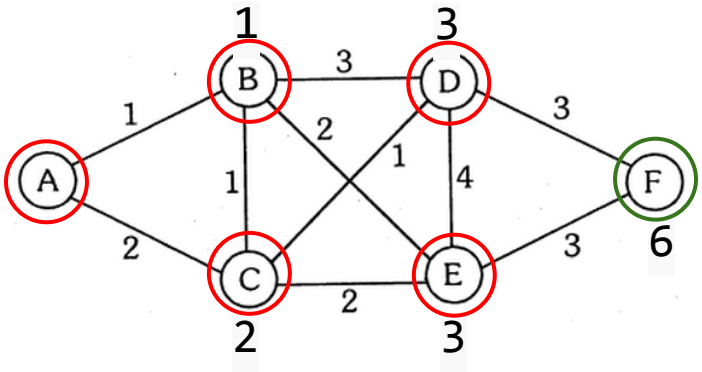


S = { A, B, C, D, E }

Networks

Dijkstra's shortest path algorithm

	0	1	2	3	4	5	6
A	0	-	-	-	-	-	-
B_A	∞	1 (A)	-	-	-	-	-
$C_{A, B}$	∞	2 (A)	2 (A, B)	-	-	-	-
D_C	∞	∞	4 (B)	3 (C)	-	-	-
E_B	∞	∞	3 (B)	3 (B)	3 (B)	-	-
$F_{D, E}$	∞	∞	∞	∞	6 (D)	6 (D, E)	-



S = { A, B, C, D, E, F }
Once destination node is in *S*, we are done!
Shortest paths are:
- ACDF, ABEF, ABCDF
- Each of length 6

Turing machines

See Anusha's review session notes:

https://courseworks.columbia.edu/access/content/group/COMSW1004_001_2014_1/ReviewSession_25_April/turing_machines.pdf

See other review session notes on CourseWorks for more information on the topics.

And obviously, have a look at the textbooks and example code.

Questions?