Data Structures and Algorithms

Prof. Nadeem Abdul Hamid
CSC 220 - Fall 2005
Lecture Unit 2 - Correctness and Efficiency

Program Correctness

- Preventive Measures
- Testing
- Error Handling

Kinds of Errors

- Syntax errors
- Runtime errors ("exceptions")
- Logic errors

Syntax Errors

- Mistakes in use of Java syntax (grammar)
  - Incorrect brackets, parentheses
  - Incorrect type of operation on values
  - Not declaring variable before its use
  - Multiple declarations for a variable
  - Not returning a value from non-void method
  - Typographical errors
  - …
- Most often caught by compiler

Runtime Errors

- Occur during program execution
- Table 2.1
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
  - `IllegalArgumentException`
  - `NumberFormatException`
  - `NullPointerException`
  - `NoSuchElementException`
  - …

Division by Zero

```
average = sum / count;
```

- Guarding against division by zero:

```
if (count == 0)
  average = 0;
else
  average = sum / count;
```

Array Index Out of Bounds

```
public boolean setElementOfX( int index, int val ) {
   if ( index < 0 || index >= x.length )
        return false;
   x[index] = value;
   return true;
}
```

```
public static void main( String[] args ) {
   String inputFileName;
   if ( args.length > 0 )
        inputFileName = args[0];
   else
        inputFileName = "Phone.dat";
}
```


Number Format Error

```
String speedStr =
   JOptionPane.showInputDialog( "Enter speed" );
double speed = Double.parseDouble( speedStr );
```


Logic Errors

- Mistake in design of class/method
- Incorrect implementation of algorithm

```
// page 40
private int find( String name ) {
   for ( int i = 0; i < size; i++ ) {
        if ( dir[i].getName().equals( name ) )
           return i;
        else
           return -1;
   }
}
```


Logic Errors in the News

- Mars Lander crash - feet/meters inconsistency
- ATM/billing program errors
- Operating system exploits
- Therac x-ray dosage software
- Jet fighter flips upside down crossing equator

Eliminating Logic Errors

- Reduce logic errors by
  - Careful initial design
  - Follow-up design analysis and checking
  - Program code testing
  - Formal methods and reasoning

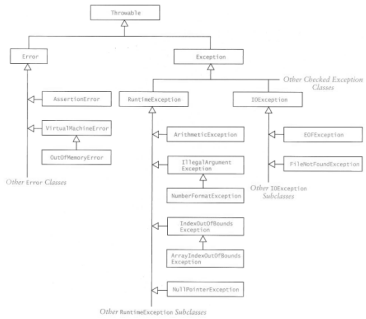
Exceptions in Java

- When runtime error/exception happens, an Exception object is instantiated and "thrown"
- **`Throwable`**: superclass of all exceptions
  - Common methods:
    - `String getMessage()`
    - `void printStackTrace()`
    - `String toString()`

Two Types of Exceptions

- **Checked**
  - Usually not programmer error
  - Must be handled in some way
- **Unchecked**
  - Error conditions due to programmer, **or**, serious external unrecoverable conditions
  - Not required (even, discouraged) from handling these
  - Subclasses of **`RuntimeException`** or **`Error`** classes

Exception Class Hierarchy



Handling Exceptions

- Default behavior:
  - Program stops executing
  - JVM prints error message and stack trace
- **Stack trace**: sequence of method calls from method causing exception back to the `main` method

try-catch Sequence

```
/**
    Method to return an integer data value
    @param prompt Message
    @return The data value read as an int
    */
 public static int readInt( String prompt ) {
   while ( true ) { // loop until valid number read
     try {
       String numStr = JOptionPane.showInputDialog( prompt );
       return Integer.parseInt( numStr );
     }
     catch ( NumberFormatException ex ) {
       JOptionPane.showMessageDialog( null,
                                      "Bad numeric string - Try again",
                                      "Error",
                                      JOptionPane.ERROR_MESSAGE );
     }
   }
 }
```


try-catch-finally Sequence

```
public static int readIntTwo( String prompt ) {
   String numStr = JOptionPane.showInputDialog( prompt );
   return Integer.parseInt( numStr );
 }

 public static void main( String[] args ) {
   final int DEFAULT_SIZE = 25;
   final int MAX_SIZE = 100;
   int size = 0;
   try {
     size = readIntTwo( "Enter new size" );
   }
   catch ( Exception ex ) {
     System.err.println( "Error occurred in call to readIntTwo" );
     size = DEFAULT_SIZE;
   }
   finally {
     if ( size > MAX_SIZE )
       size = MAX_SIZE;
   }
   System.out.println( "Read size: " + size );

   System.exit( 0 );
 }
}
```


Handling Exception Subclasses

```
.
.
.
catch ( EOFException ex ) {
  System.out.print( "End of file reached " );
  System.out.println( " - processing complete" );
  System.exit( 0 );
}
catch ( IOException ex ) {
  System.err.println( "Input/Output Error:" );
  ex.printStackTrace();
  System.exit( 1 );
}
```

More on Exceptions

- try-catch-finally Syntax: pg 74-75
- Look over:
  - Pitfall pg. 76
  - Program style pg. 76

Throwing Exceptions

- Instead of immediately handling an exception, a method may 'throw' it to a higher-level method in the stack
- Or, you may instantiate the exception object yourself and 'throw' it

The throws Clause

```
public void readData() throws IOException {
   BufferedReader console = new BufferedReader(
      new InputStreamReader( System.in ) );
   System.out.print( "Enter first name: " );
   firstName = console.readLine();
   System.out.print( "Enter last name: " );
   lastName = console.readLine();
   ...
}

public void setNewPerson() {
   try {
      readData();
      // process data read
      ...
   }
   catch ( IOException ex ) {
      ...
   }
}
```


The throw Statement

```
public String addOrChangeEntry( String name, String number ) {
   if ( ! isPhoneNumberFormat( number ) )
      throw new IllegalArgumentException
         ( "Invalid phone number: " + number );
   ...
   // add or change the number
}

...

   try {
      addOrChangeEntry( myName, myNumber );
   }
   catch ( IllegalArgumentException ex ) {
      System.err.println( ex.getMessage() );
   }
```


More on Throwing Exceptions

- Syntax: page 80
- Example 2.10
- Style: page 84

Handling Logic Errors

- Logic (Semantic) errors can be difficult to detect and isolate
- Best situation
  - Logic error is in part of code that always executes
- Worst situation
  - Error is in obscure piece of code that doesn't always execute
  - Testing may miss this code - software will be delivered with hidden defect

Structured Walkthroughs

- Hand-checking algorithm by simulating its execution *before* implementing any code
- Done together with others in a team
  - Because designer may anticipate steps that don't actually happen that way
- Effective process in industrial experience

25

Levels of Testing

- **Unit testing** - testing smallest testable piece of software (in OOD, method or class)
- **Integration testing** - testing interaction among units (e.g. interactions between classes)
- **System testing** - testing whole program in context it will be used (other programs/hardware)
- **Acceptance testing** - show program meets all functional requirements

26

Types of Testing

- Black-box testing
  - Test an item (method/class/program) based on its interfaces and functional specifications/requirements
  - Also called *closed-box* or *functional* testing
- White-box testing
  - Test a software element with knowledge of its internal structure (i.e. implementation)
  - Exercise as many paths as possible
  - Also called *glass-box*, *open-box*, or *coverage*

27

Example - Path Coverage

```
public void someMethod( char a, char b ) {
   if ( a < 'M' ) {
      if ( b < 'X' )
         System.out.println( "path 1" );
      else
         System.out.println( "path 2" );
   } else {
      if ( b < 'C' )
         System.out.println( "path 3" );
      else
         System.out.println( "path 4" );
   }
}
```

28

Testing Tips

- Developing test plan
- Defensive programming
- Documentation comments
- Trace execution using print statements
- Test data
- Boundary conditions

- Look over pages 88-90
- *ArraySearch.java*

29

Test Cases

- *SinCos.java*

30

Assertions and Loop Invariants

- Section 2.7

31

Algorithmic Efficiency

```
public static int search( int[] x, int target ) {
   for ( int i = 0; i < x.length; i++ )
      if ( x[i] == target ) return i;
   // target not found
   return -1;
}


public static boolean areDifferent( int[] x, int[] y ) {
   for ( int i = 0; i < x.length; i++ )
      if ( search(y, x[i]) != -1 ) return false;
   return true;
}
```

32

Algorithmic Efficiency (cont.)

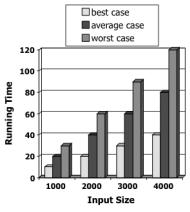
```
public static boolean areUnique( int[] x ) {
   for ( int i = 0; i < x.length; i++ )
      for ( int j = 0; j < x.length; j++ )
         if ( i != j && x[i] == x[j] )
            return false;
   return true;
}

public static boolean areUnique( int[] x ) {
   for ( int i = 0; i < x.length; i++ )
      for ( int j = i + 1; j < x.length; j++ )
         if ( x[i] == x[j] )
            return false;
   return true;
}
```

33

Running Time

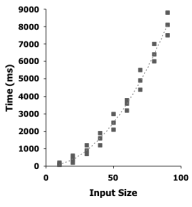
- Most algorithms transform input objects into output objects
- The running time of an algorithm typically grows with the input size
- Average case time is often difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



34

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



35

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments must be used

36

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(*A*, *n*)
  Input array *A* of *n* integers
  Output maximum element of *A*

  currentMax ← *A*[0]
  for *i* ← 1 **to** *n* − 1 **do**
    if *A*[*i*] > *currentMax* **then**
      currentMax ← *A*[*i*]
  return *currentMax*

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time to execute

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the (pseudo)code, we determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

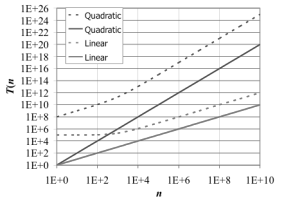
Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* − 1 **do**	2*n*
if *A*[*i*] > *currentMax* **then**	2(*n* − 1)
currentMax ← *A*[*i*]	2(*n* − 1)
{ increment counter *i* }	2(*n* − 1)
return *currentMax*	1
	Total 8*n* − 2

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



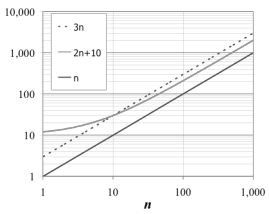
Big-Oh Notation

- Upper bound on the growth rate of a function (running time or memory)
  - "$O()$": 'order of magnitude'
- To say "$f(n)$ is $O(g(n))$" is to say that $f(n)$ is "less than or equal to" $g(n)$
- More precisely, given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for all $n \geq n_0$

Big-Oh Example

- $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$
- Remember, c must be a constant



More Big-Oh Examples

- $7n - 2$

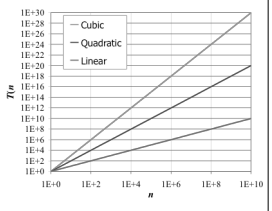
  7n-2 is O(n)
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$
- $3 \log n + 5$

  3 log n + 5 is O(log n)
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - N-Log-N $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Simple Rules for Big-Oh

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, *i.e.*,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm *arrayMax* executes at most $8n - 2$ primitive operations
  - We say that algorithm *arrayMax* "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Using Big-Oh

- The big-Oh notation expresses a relationship between functions. It does not say what the functions *are*.
- Big-Oh does not just refer to the worst-case running time
  - binary search on an array,
    - the worst-case running time is in *O(log n)*,
    - the best-case running time is in *O(1)*, and
    - the memory use is in *O(n)*

More Caveats

- Beware huge coefficients
  - $10^{100}n$ is *O(n)* and probably not as useful in practice as $10n \log n$
- Beware key lower order terms
- Beware when *n* is "small"
- Generally speaking, algorithms running in *O(n log n)* time or faster can be considered "efficient"
  - Even n^2 may be reasonable if *n* is small

Does it matter?

Let *n* = 1,000, and 1 ms / operation.

	n = 1000, 1 ms/op	max *n* in one day
n	1 second	86,400,000
$n \log_2 n$	10 seconds	3,943,234
n^2	17 minutes	9,295
n^3	12 days	442
n^4	32 years	96
n^{10}	3.17×10^{19} years	6
2^n	1.07×10^{301} years	26