CS151

Exceptions

Complexity Analysis
Exceptions

- Unexpected events during execution
  - unavailable resource
  - unexpected input
  - logical error
- In Java, exceptions are objects
  - because almost everything is an object
- 2 options with an Exception
  - “Throw” it
    - this says that the exception must be handled elsewhere
  - “Catch” it.
    - handle the problem here and now
    - Once caught, you can re-throw
Catching Exceptions

- Exception handling
- try-catch
- An exception is caught by having control transfer to the matching catch block
- At most one catch block is executed
- If no exception occurs, all catch blocks are ignored

```java
try {
    guardedBody
} catch (exceptionType_1 variable_1) {
    remedyBody_1
} catch (exceptionType_2 variable_2) {
    remedyBody_2
} ...
...
Throwing Exceptions

- An exception is thrown
  - implicitly by the JVM because of errors
  - explicitly by code
- Method signature – throws
- For example, the constructor of the class FileReader can throw an exception
- From Java documentation
  public FileReader(String fileName) throws FileNotFoundException
Handling — Bad

- All exceptions thrown within your code must be caught by your code

```java
public class ExBad1 {
    public void doo(String filename) throws IOException {
        FileReader bb = new FileReader(filename);
    }

    public static void main(String[] args) {
        ExBad0 ex = new ExBad1();
        ex.doo("20");
    }
}
```

Does not compile, exception not handled
Handling — still bad

- NEVER throw from main
- There is no one (nothing) to catch

```java
public class ExBad2 {
    public void doo(String filename) throws IOException {
        FileReader bb = new FileReader(filename);
    }
    public static void main(String[] args) throws IOException {
        ExBad2 ex = new ExBad2();
        ex.doo("20");
    }
}
```

Throws from main!!
NEVER DO THIS
public class ExGood {
    public void doo(String filename) throws IOException {
        FileReader bb = new FileReader(filename);
    }
    public static void main(String[] args) {
        ExGood ex = new ExGood();
        try {
            ex.doo("20");
        } catch (IOException ioe) {
            System.err.println("Problem calling doo with arg 20" + ioe);
        }
    }
}

Exception is caught

Inside of catch clause use System.err.println
Exceptions Options

- Within code you write ... choice
- Catch and handle exception here
- Throw exception
- Catch exception and return error value
  - Not available from constructor

```java
public class ReadCSV implements Iterable<String[]> {
    public ReadCSV(String name, int maxSplit) throws IOException {
        if (name.contains("://")) {
            uu = new URL(name);
        } else {
            File file = new File(name);
            URI uri = file.toURI();
            uu = uri.toURL();
        }
        URLConnection huc = uu.openConnection();
        huc.setConnectTimeout(3 * 1000);
        new BufferedReader(new InputStreamReader(huc.getInputStream()));
        this.maxSplit = maxSplit;
    }

    public static void main(String[] args) {
        try {
            ReadCSV csvReader;
            csvReader = new ReadCSV("https://cs.brynmawr.edu/cs151/Data/HW1/us.csv", 4);
        } catch (IOException ioe) {
            System.err.println("Ending. Cannot read. " + ioe.toString());
            return;
        }
    }
}
```
Java’s Exception Hierarchy

- RuntimeException
  - ArithmeticException
  - NullPointerException
  - ClassCastException
  - IndexOutOfBoundsException
    - ArrayIndexOutOfBoundsException
    - StringIndexOutOfBoundsException
- IOException
- SQLException
- AWTException
- StackOverflowError
- VirtualMachineError
- OutOfMemoryError
- InterruptedException
- Assertion error
- ExceptionInInitializerError
- IOMonitor
- AWTError

RuntimeError & its sub-classes and Error & its sub-classes are Unchecked Exception; All other exceptions are Checked Exception
Running Time

• The run time of a program depends on

<table>
<thead>
<tr>
<th>Factor</th>
<th>Effect on time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Efficiency</td>
<td>Better ==&gt; less time (inverse)</td>
</tr>
<tr>
<td>Size of Input</td>
<td>Bigger ==&gt; more time (direct)</td>
</tr>
<tr>
<td>Other?</td>
<td>speed of computer, startup</td>
</tr>
</tbody>
</table>

• How do you measure running time?
  • Clock time?
  • CPU usage?
    • What effect reliability of these measures?
public class Timer {
    private static final int REPS = 2; // number of trials
    private static final int NANOS_SEC = 1000000000; // nanosec per sec

    public double doSomething(int[] data) {
        double k = 0;
        for (long i = 0; i < data.length; i++) {
            for (long j = 0; j < data.length; j++) {
                k += Math.sqrt(i * j);
            }
        }
        return k;
    }

    public static void main(String[] args) {
        Timer timer = new Timer();
        long data[] = new long[REPS];
        for (int j = 1000; j < 65000; j = j*2) {
            for (int i = 0; i < REPS; i++) {
                long start = System.nanoTime();
                timer.doSomething(new int[j]);
                long finish = System.nanoTime();
                data[i] = (finish - start) / NANOS_SEC;
            }
        }
    }
}
Experimental Studies

• Write a program implementing the algorithm
• Run it with different input sizes and compositions
• Record times and plot results
Limitation of Experiments

- You have to implement the algorithm
  - How good is your implementation??
- You have to generate inputs (data)
  - Do your inputs capture everything?
- Comparing two algorithms requires exact same hardware and software environments
  - Even then timing is hard
    - multiprocessing
    - file i/o
Theoretical Analysis

• Use a high-level description of algorithm
  □ pseudo-code
• Running time as a function input size, \( n \)
  • assume worst possible inputs
• Ignore other details of the input
• Independent of the hardware/software environment
Primitive Operations

- Basic computations
  - *, /, +, -
- Comparisons
  - ==, >, <
- Setting
  - x<-y
- Assume all of these to take constant time
  - exact constant is not important
  - Because constant is not important, can do more than just this list
Example

Time required to compute an average

```java
public double allAverage(long[] data) {
    double res = 0;
    for (int i=0; i<data.length; i++) {
        res = res + data[i];
    }
    return res/data.length;
}

public double posAverage(long[] data) {
    double res = 0;
    long pCount = 0;
    for (int i=0; i<data.length; i++) {
        long datum = data[i];
        if (datum > 0) {
            res = res + datum;
            pCount = pCount + 1;
        }
    }
    return res/pCount;
}
```

How many operations? (In terms of the length of data)
Estimate Running Time

- `allAverage` executes 5N+3 operations
- `posAverage` executes a total of 9N+3 primitive operations in the worst case, 5N+3 in the best case.
- Let \( a \) be the fastest primitive operation time, \( b \) be the slowest primitive operation time
- Let \( T(n) \) denote the worst-case time of `allAverage`. Then:
  \[ a(5n+3) < T(n) < b(5n+3) \]
- \( T(n) \) is bounded by two functions
  - both are linear in terms of \( n \)
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 both algorithms.
Comparison of Two Algorithms

- insertion sort: $\frac{n^2}{4}$
- merge sort: $2n\log n$
- suppose $n=10^8$
  - insertion sort: $10^8 \times 10^8 / 4 = 2.5 \times 10^{15}$
  - merge sort: $10^8 \times 26 \times 2 = 5.2 \times 10^9$
  - or merge sort can be expected to be about $10^6$ times faster
  - so if merge sort takes 10 seconds then insertion sort takes about 100 days
Asymptotic Notation

• Provides a way to simplify analysis
• Allows us to ignore less important elements
  □ for example, constant factors
• Focus on the dominant term
How do these functions grow?

- $f_1(x) = 43n^2 \log^4 n + 12n^3 \log n + 52n \log n$
- $f_2(x) = 15n^2 + 7n \log^3 n$
- $f_3(x) = 3n + 4 \log_5 n + 91n^2$
- $f_4(x) = 13 \cdot 3^{2n+9} + 4n^9$
Big $\Theta$

- Constant factors are ignored
- Upper bound on time
- Goal is to have an easily understood summary of algorithm speed
  - not implementation speed
Sublinear Algorithms

- $O(1)$
  - runtime does not depend on input

- $O(\log_2 n)$
  - algorithm constantly halves input
Linear Time Algorithms: $O(n)$

- The algorithm’s running time is at most a constant factor times the input size.
- Process the input in a single pass spending constant time on each item:
  - max, min, sum, average, linear search
- Any single loop
$O(n \log n)$ time

Frequent running time in cases when algorithms involve:

• Sorting
  • only the “good” algorithms
    • e.g. quicksort, merge sort, ...
Quadratic Time: $O(n^2)$

- Nested loops, double loops
  - The `doSomething` program
- Processing all pairs of elements
- The less-good sorting algorithms
  - e.g., insertion sort
Slow!!!! Times

- polynomial time: $O(n^k)$
  - All subsets of $n$ elements of size $k$

- exponential time: $O(2^n)$
  - All subsets of $n$ elements (power set)

- factorial time: $O(n!)$
  - All permutations of $n$ elements
# Algorithm Run Times

<table>
<thead>
<tr>
<th>N</th>
<th>log(n)</th>
<th>n</th>
<th>n log(n)</th>
<th>n*n</th>
<th>n<em>n</em>n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>10</td>
<td>33</td>
<td>100</td>
<td>1000</td>
<td>10^5</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
<td>100</td>
<td>664</td>
<td>10000</td>
<td>10^6</td>
<td>10^94</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>1000</td>
<td>9966</td>
<td>10^6</td>
<td>10^9</td>
<td>10^1435</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
<td>10000</td>
<td>132877</td>
<td>10^8</td>
<td>10^12</td>
<td>10^19355</td>
</tr>
<tr>
<td>100000</td>
<td>17</td>
<td>100000</td>
<td>1660964</td>
<td>10^10</td>
<td>10^15</td>
<td>10^(10^6)</td>
</tr>
</tbody>
</table>
Analyzing StuffBag

- add
- remove one
- count
- remove all of X

- Can these times be improved?
  - at what cost?