Administrative Slide #1

• New Office Hours:
  – Monday 10:00-12:30
  – Tuesday 1:00-3:30
  – Wednesday 10:00-11:00
  – Thursday 11:00-12:00
  – Friday 2:30-5:30
Administrative Slide #2

• Penn State Behrend 24th Annual Undergraduate Student Research and Creative Accomplishment Conference
  
• [http://psbehrend.psu.edu/research-outreach/student-research/annual-conference-1](http://psbehrend.psu.edu/research-outreach/student-research/annual-conference-1)

• Abstract deadline: February 25
• Notification: March 9
• Conference: April 11 (Saturday), 8-noon + lunch
Last Time

- ThreeSum.java
- ThreeSumWithTiming.java
- ThreeSumFast.java

- Binary Search
- Calculating theoretical runtime
- Timing program execution
What Can Complicate Our Analysis?

• Large constants
• Non-dominant loops
• Instruction time
• System considerations
• Too close to call
• Dependence on inputs
• Multiple problem parameters
Large Constants

• I told you to ignore lower-order terms...
  \[ \frac{1}{2} n^2 + cn = \sim \frac{1}{2} n^2 \]
  – What if \( c = 1000 \), or \( c = 1000000 \), or \( c = 10^{100} \)?
    • It’s still better than a \( \sim n^2 / 2 \) function for a large enough \( n \)...
  – Consider a program that runs in linear time, vs a program that does the same thing in constant time. But the constant is 1000000...
    • Constant is still better with a large enough \( n \)...

• If the constant is unknown it’s *generally* safe to ignore.
Non-dominant Loops

```
int sum = 0;
for (int i = 1; i < n; i *= 2) {
    for (int j = 0, j < i; j++) {
        sum++;
    } //for
} //for
```

**Geometric sum:** \[1 + 2 + 4 + 8 + \cdots + n = 2n - 1 \sim 2n\]

**Harmonic sum:** \[1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{n} = \ln(n) \sim \log(n)\]
Instruction Time

• We cannot assume that every instruction takes the same amount of time to execute. (Remember the $t_0$ through $t_4$ variables from the table?)

```cpp
int sum = 0, product = 0, i;
for (i = 0; i < n; i++) {
    sum++;
} //for

for (i = 0; i < n; i++) {
    product = product * product;
} //for
```
System Considerations

![Windows Task Manager screenshot](image)

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>User Name</th>
<th>CPU</th>
<th>CPU Time</th>
<th>Memory (Pr.)</th>
<th>Page Faults</th>
<th>PF Delta</th>
<th>Threads</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firefox.exe *32</td>
<td>156520</td>
<td>John</td>
<td>07</td>
<td>00:30:15</td>
<td>1,158,484 K</td>
<td>28,520,518</td>
<td>117</td>
<td>72</td>
<td>Firefox</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>1164</td>
<td>SYSTEM</td>
<td>00</td>
<td>07:58:44</td>
<td>161,028 K</td>
<td>33,372,960</td>
<td>0</td>
<td>30</td>
<td>Host Process for Windows Services</td>
</tr>
<tr>
<td>aim.exe *32</td>
<td>4364</td>
<td>John</td>
<td>01</td>
<td>03:44:29</td>
<td>76,100 K</td>
<td>14,472,666</td>
<td>0</td>
<td>66</td>
<td>AOL Instant Messenger</td>
</tr>
<tr>
<td>BTSync.exe *32</td>
<td>4624</td>
<td>John</td>
<td>00</td>
<td>20:12:32</td>
<td>70,684 K</td>
<td>41,115,297</td>
<td>17</td>
<td>27</td>
<td>BitTorrent Sync</td>
</tr>
<tr>
<td>explorer.exe</td>
<td>2116</td>
<td>John</td>
<td>00</td>
<td>01:47:29</td>
<td>44,276 K</td>
<td>77,019,606</td>
<td>2</td>
<td>58</td>
<td>Windows Explorer</td>
</tr>
<tr>
<td>dwm.exe</td>
<td>3924</td>
<td></td>
<td>00</td>
<td>13:44:17</td>
<td>39,364 K</td>
<td>8,356,879</td>
<td>0</td>
<td>5</td>
<td>Desktop Window Manager</td>
</tr>
<tr>
<td>POWERPNT.EXE *32</td>
<td>66160</td>
<td>John</td>
<td>00</td>
<td>00:21:22</td>
<td>38,044 K</td>
<td>4,619,599</td>
<td>0</td>
<td>15</td>
<td>Microsoft PowerPoint</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>1236</td>
<td>SYSTEM</td>
<td>00</td>
<td>03:15:53</td>
<td>30,980 K</td>
<td>98,642,880</td>
<td>8</td>
<td>48</td>
<td>Host Process for Windows Services</td>
</tr>
<tr>
<td>SearchIndexer.exe</td>
<td>49216</td>
<td>SYSTEM</td>
<td>00</td>
<td>01:51:14</td>
<td>27,412 K</td>
<td>4,753,480</td>
<td>0</td>
<td>15</td>
<td>Microsoft Windows Search Indexer</td>
</tr>
<tr>
<td>Steam.exe *32</td>
<td>88152</td>
<td>John</td>
<td>00</td>
<td>03:40:44</td>
<td>22,512 K</td>
<td>3,660,440,254</td>
<td>0</td>
<td>23</td>
<td>Steam Client Bootstrapper</td>
</tr>
<tr>
<td>Dropbox.exe *32</td>
<td>37852</td>
<td>John</td>
<td>00</td>
<td>01:10:20</td>
<td>16,984 K</td>
<td>14,974,656</td>
<td>0</td>
<td>47</td>
<td>Dropbox</td>
</tr>
<tr>
<td>avgnsa.exe</td>
<td>115048</td>
<td>SYSTEM</td>
<td>00</td>
<td>00:42:48</td>
<td>12,948 K</td>
<td>129,610,635</td>
<td>0</td>
<td>16</td>
<td>AVG Online Shield Service</td>
</tr>
<tr>
<td>sidebar.exe</td>
<td>4336</td>
<td>John</td>
<td>00</td>
<td>26:46:37</td>
<td>12,140 K</td>
<td>587,339,874</td>
<td>157</td>
<td>21</td>
<td>Windows Desktop Gadgets</td>
</tr>
<tr>
<td>audiodg.exe</td>
<td>43684</td>
<td>LOCAL</td>
<td>00</td>
<td>14:23:03</td>
<td>12,120 K</td>
<td>171,562</td>
<td>0</td>
<td>9</td>
<td>Windows Audio Device Graph Isolator</td>
</tr>
<tr>
<td>avgrsa.exe</td>
<td>116300</td>
<td>SYSTEM</td>
<td>00</td>
<td>00:06:55</td>
<td>11,904 K</td>
<td>464,423</td>
<td>0</td>
<td>78</td>
<td>AVG Resident Shield Service</td>
</tr>
</tbody>
</table>

Processes: 127  CPU Usage: 14%  Physical Memory: 80%
System Considerations

- Caching
- Java garbage collector
- Internet download
- Just-in-time compiler

- Scientific method: “Experiments should be reproducible!”
  - What is happening in your system during a program run is unlikely to be produced again.
  - Other system processes should *in principle* be negligible or under your control.
Multiple Problem Parameters

• It is not unusual to have more than one input parameter.
  – Example: an algorithm that builds a data structure, then performs operations on that data structure.
    • Parameters: size of the structure (m), number of operations on the structure (n).
  – Is there a difference between the runtime $O(m \times \log(n))$ and $O(n \times \log(m))$?
    • What if $n = 2^m$?
Too Close to Call

• Program A is faster on Input 1, but Program B is faster on Input 2. Which is the better algorithm?
• It depends...
• Don’t devote an extreme amount of energy towards finding the “best” implementation.
  – Order of growth is most important to our analysis in this class.
Dependence on Inputs

```java
foreach item in items {
    for (i = item, i < 100; i++) {
        System.out.println(i);
    } //for
} //foreach
```

- Input 1: 99, 115, 107, 98, 100, 101, 110, 111
- Input 2: 7, 1, 12, 16, 17, 21, -8325827
Dependence on Inputs

• We cannot make the assumption that running time will be insensitive to inputs.

• Consider a modified version of ThreeSum, which reports true/false whether or not a triple exists, rather than counting them.
  – If the first three numbers are a triple, the runtime will be $O(1)$.
  – If the last three numbers are a triple, the runtime will be $O(n^3)$. 
Coping with Input Dependence

• In many problems, running time will vary greatly depending on input. Solutions:
  – We can model the kind of input that will be processed.
    • Challenge: model may be unrealistic.
  – We can analyze using worst-case.
    • Challenge: worst case probability might be very, very small.
  – We can introduce randomness.
    • Challenge: guarantees are not absolute.
  – Amortized analysis (keep track of total cost of all operations divided by number of operations).
    • Challenge: can have significant overhead.
Back to Implementation Dependence

• Algorithm Analyst – Discover as much relevant information about an algorithm as possible.

• Application Programmer – Apply that knowledge to develop programs that effectively solve the problems at hand.

• Ideal goal: generate algorithms that lead to clear and compact code
Memory

• Analyzing memory usage is simultaneously easier and harder than analyzing time complexity.
  – Every integer is 4 bytes.
  – Every double is 8 bytes.
  – Every String is...... well......

• As an additional complication, Java’s memory allocation system is both implementation dependent and hardware dependent.
Memory and Objects

- Objects typically have an overhead of 16 bytes.
- Memory usage is padded to a multiple of 8 bytes. (why?)
- Things get more complicated when we delve into data structures.
Memory and Arrays

• An array of n integers needs 24 bytes of overhead (16 for the object, 4 for the length, 4 for padding before the data).
  – Total memory requirement = 24+4n, rounded up to the next multiple of 8, so \( \sim 4n \). (for doubles?)

• A two-dimensional array (array of arrays) of integers of size \( M \times N \) needs \( \sim 4MN \):
  – 24 bytes for the overall overhead,
  – 8\( M \) bytes of overhead referencing each of the row arrays,
  – 24\( M \) bytes of overhead for each of the row arrays,
  – 4\( MN \) bytes for each integer (with possibly more padding).
Memory and Strings

• Java 7 and later: (ALERT error in book page 202)
  – Two instance variables:
    • Reference to a char array,
    • An int value hash code.
  – A String of length n takes $56 + 2n$ bytes (16 object overhead, 8 reference to char[], 4 hash code, 4 padding, $24 + 2n$ for the char[] itself.

• Java 6 and earlier:
  – As above, but with two extra integer variables, offset and count, used to make substring computation more efficient. New total = $64 + 2n$ bytes.
Memory and Substrings

• In Java 7+, implementing the `substring()` method involves creating a new `char[]`, taking linear time and linear space.

• In Java 6-, we can alias the new substring using the original object – the `offset` and `count` fields identify the substring, taking constant time and constant space.

• Extracting a substring takes either constant or linear memory depending on the underlying implementation.
Memory and Substrings

Extracting a substring takes either constant or linear memory depending on the underlying Java implementation.
Any Questions?