Performance Testing

Based on slides created by Marty Stepp
http://www.cs.washington.edu/403/
Acceptance, performance

- **acceptance testing**: System is shown to the user / client / customer to make sure that it meets their needs.
  - A form of black-box system testing

- Performance is important.
  - Performance is a major aspect of program acceptance by users.
  - Your intuition about what's slow is often wrong.
public class BuildBigString {
    public final static int REPS = 80000;

    // Builds/returns a big, important string.
    public static String makeString() {
        String str = "";
        for (int n = 0; n < REPS; n++) {
            str += "more";
        }
        return str;
    }

    public static void main(String[] args) {
        System.out.println(makeString());
    }
}
public class Fibonacci {
    public static void main(String[] args) {
        // print the first 10000 Fibonacci numbers
        for (int i = 1; i <= 10000; i++) {
            System.out.println(fib(i));
        }
    }

    // pre: n >= 1
    public static long fib(int n) {
        if (n <= 2) {
            return 1;
        } else {
            return fib(n - 2) + fib(n - 1);
        }
    }
}
What's wrong with this? (3)

```java
public class WordDictionary {
    // The set of words in our game.
    List<String> words = new ArrayList<String>();

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

    public boolean contains(String word) {
        for (String s : words) {
            if (s.toLowerCase().equals(word)) {
                return true;
            }
        }
        return false;
    }
}
```
public class BankManager {
    public static void main(String[] args) {
        Account[] a = Account.getAll();
        for (int i = 0; i < Math.sqrt(895732); i++) {
            a[i].loadTaxData();
            if (a.meetsComplexTaxCode(2020)) {
                a[i].fileTaxes(4 * 4096 * 17);
            }
        }
    }
}

Account[] a2 = Account.getAll();
for (int i = 0; i < Math.sqrt(895732); i++) {
    if (a.meetsComplexTaxCode(2020)) {
        a2[i].setTaxRule(4 * 4096 * 17);
        a2[i].save(new File(a2.getName()));
    }
}
The correct answers

1. Who cares?
2. Who cares?
3. Who cares?
4. Who cares?

• "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."  
  -- Donald Knuth

• "We follow two rules in the matter of optimization:
  1. Don't do it.
  2. (for experts only) Don't do it yet."
  -- M. A. Jackson
Thinking about performance

• The app is only too slow if it doesn't meet your project's stated performance requirements.
  – If it meets them, DON'T optimize it!

• Which is more important, fast code or correct code?

• What are reasonable performance requirements?
  – What are the user's expectations? How slow is "acceptable" for this portion of the application?
  – How long do users wait for a web page to load?
  – Some tasks (admin updates database) can take longer
Optimization myths

• **Myth:** You should optimize your code as you write it.
  – No; makes code ugly, possibly incorrect, and not always faster.
  – Optimize later, only as needed.

• **Myth:** Having a fast program is as important as a correct one.
  – If it doesn't work, it doesn't matter how fast it's running!

• **Myth:** Certain operations are inherently faster than others.
  – $x \ll 1$ is faster to compute than $x \times 2$?
  – This depends on many factors, such as language used.
    Don't write ugly code on the assumption that it will be faster.

• **Myth:** A program with fewer lines of code is faster.
Perceived performance

• "My app feels too slow. What should I do?"
  – possibly optimize it
  – And/or improve the app's perceived performance

• perceived performance:
  User's perception of your app's responsiveness.

• factors affecting perceived performance:
  – loading screens
  – multi-threaded UIs (GUI doesn't stall while something is happening in the background)
Optimization metrics

- **runtime / CPU usage**
  - what lines of code the program is spending the most time in
  - what call/invocation paths were used to get to these lines
    - naturally represented as tree structures

- **memory usage**
  - what kinds of objects are on the heap
  - where were they allocated
  - who is pointing to them now
  - "memory leaks" (does Java have these?)

- **web page load times, requests/minute, etc.**
Benchmarking, optimization

- **benchmarking**: Measuring the absolute performance of your app on a particular platform (coarse-grained measurement).

- **optimization**: Refactoring and enhancing to speed up code.
  - I/O routines
    - accessing the console (print statements)
    - files, network access, database queries
    - `exec()` / system calls
  - Lazy evaluation saves you from computing/loading
    - don't read / compute things until you need them
  - Hashing, caching save you from reloading resources
    - combine multiple database queries into one query
    - save I/O / query results in memory for later
Optimizing memory access

• Non-contiguous memory access (bad):

```java
for (int col = 0; col < NUM_COLS; col++) {
    for (int row = 0; row < NUM_ROWS; row++) {
        table[row][column] = bulkyMethodCall();
    }
}
```

• Contiguous memory access (good):

```java
for (int row = 0; row < NUM_ROWS; row++) {
    for (int col = 0; col < NUM_COLS; col++) {
        table[row][column] = bulkyMethodCall();
    }
}
```

– switches rows \( \text{NUM\_ROWS} \) times, not \( \text{NUM\_ROWS} \times \text{NUM\_COLS} \)
Optimizing data structures

• Take advantage of hash-based data structures
  – searching an ArrayList (contains, indexOf) is $O(N)$
  – searching a HashMap/HashSet is $O(1)$

• Getting around limitations of hash data structures
  – need to keep elements in sorted order? Use TreeMap/TreeSet
  – need to keep elements in insertion order? Use LinkedHashSet

![Diagram of a hash table with elements A, B, C, D and size = 4]
Avoiding computations

• Stop computing when you know the answer:

    found = false;
    for (i = 0; i < reallyBigNumber; i++) {
        if (inputs[i].isTheOneIWant()) {
            found = true;
            break;
        }
    }

• Hoist expensive loop-invariant code outside the loop:

    double taxThreshold = \texttt{reallySlowTaxFunction}();
    for (i = 0; i < reallyBigNumber; i++) {
        accounts[i].applyTax(\texttt{taxThreshold});
    }
Lookup tables

- Figuring out the number of days in a month:
  ```java
  if (m == 9 || m == 4 || m == 6 || m == 11) {
    return 30;
  } else if (month == 2) {
    return 28;
  } else {
    return 31;
  }
  ```

- Days in a month, using a lookup table:
  ```java
  DAYS_PER_MONTH = {-1, 31, 28, 31, 30, 31, 30, ..., 31};
  ...
  return DAYS_PER_MONTH[month];
  ```

  - Probably not worth the speedup with this particular example...
Optimization is deceptive

```c
int sum = 0;
for (int row = 0; row < NUM_ROWS; row++) {
    for (int col = 0; col < NUM_COLS; col++) {
        sum += matrix[row][column];
    }
}
```

- **Optimized code:**
  ```c
  int sum = 0;
  Cell* p = matrix;
  Cell* end = &matrix[NUM_ROWS - 1][NUM_COLS - 1];
  while (p != end) {
      sum += *p++;
  }
  ```

- **Speed-up observed: NONE.**
  - Compiler was already optimizing the original into the second!
public static boolean isPrime(int n) {
    double sqrt = Math.sqrt(n);
    for (int i = 2; i <= sqrt; i++)
        if (n % i == 0) { return false; }
    return true;
}

• **dynamic programming**: Caching previous results.

```java
private static Map<Integer, Boolean> PRIME = ...;
public static boolean isPrime2(int n) {
    if (!PRIME.containsKey(n))
        PRIME.put(n, isPrime(n));
    return PRIME.get(n);
}
```
Optimization tips

• **Pareto Principle**, aka the "80-20 Rule"
  – 80% of a program's execution occurs within 20% of its code.
  – You can get 80% results with 20% of the work.

• "The best is the enemy of the good."
  – You don't need to optimize all your app's code.
  – Find the worst bottlenecks and fix them. Leave the rest.
Profiling

• **profiling**: Measuring relative system statistics (fine-grained).
  – Where is the most time being spent? ("classical" profiling)
    • Which method takes the most time?
    • Which method is called the most?
  – How is memory being used?
    • What kind of objects are being created?
    • This in especially applicable in OO, GCed environments.
  – Profiling is *not* the same as benchmarking or optimizing.
Types of profiling

- **insertion**: placing special profiling code into your program (manually or automatically)
  - *pros*: can be used across a variety of platforms; accurate
  - *cons*: requires recompiling; profiling code may affect performance

- **sampling**: monitoring CPU or VM at regular intervals and saving a snapshot of CPU and/or memory state
  - *pros*: no modification of app is necessary
  - *cons*: less accurate; varying sample interval leads to a time/accuracy trade-off; small methods may be missed; cannot easily monitor memory usage
Android Traceview

• Traceview:
  – `Debug` class generates `.trace` files to be viewed

    – `Debug.startMethodTracing();` ...
      `... Debug.stopMethodTracing();`

    – `timeline panel`: describes when each thread/method start/stops
    – `profile panel`: summary of what happened inside a method
Android profiling DDMS

- Dalvik Debug Monitor Server (DDMS):
  - Eclipse: Window → Open Perspective → Other... → DDMS
  - console: run `ddms` from `tools/` directory
  - On Devices tab, select process that you want to profile
    - Click **Start Method Profiling**
    - Interact with application to run and profile its code.
Java profiling tools

• Many free Java profiling/optimization tools available:
  – TPTP profiler extension for Eclipse
  – Extensible Java Profiler (EJP) - open source, CPU tracing only
  – Eclipse Profiler plugin
  – Java Memory Profiler (JMP)
  – Mike's Java Profiler (MJP)
  – JProbe Profiler - uses an instrumented VM

• hprof (java -Xrunhprof)
  – comes with JDK from Sun, free
  – good enough for anything I've ever needed
Using hprof

usage: java -Xrunhprof:[help] | [option=] value, ...

<table>
<thead>
<tr>
<th>Option Name and Value</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap=dump</td>
<td>sites</td>
<td>all</td>
</tr>
<tr>
<td>cpu=samples</td>
<td>times</td>
<td>old</td>
</tr>
<tr>
<td>monitor=y</td>
<td>n</td>
<td>monitor contention</td>
</tr>
<tr>
<td>format=a</td>
<td>b</td>
<td>text(txt) or binary output</td>
</tr>
<tr>
<td>file=&lt;file&gt;</td>
<td>write data to file</td>
<td>off</td>
</tr>
<tr>
<td>depth=&lt;size&gt;</td>
<td>stack trace depth</td>
<td>4</td>
</tr>
<tr>
<td>interval=&lt;ms&gt;</td>
<td>sample interval in ms</td>
<td>10</td>
</tr>
<tr>
<td>cutoff=&lt;value&gt;</td>
<td>output cutoff point</td>
<td>0.0001</td>
</tr>
<tr>
<td>lineno=y</td>
<td>n</td>
<td>line number in traces?</td>
</tr>
<tr>
<td>thread=y</td>
<td>n</td>
<td>thread in traces?</td>
</tr>
<tr>
<td>doe=y</td>
<td>n</td>
<td>dump on exit?</td>
</tr>
<tr>
<td>msa=y</td>
<td>n</td>
<td>Solaris micro state accounting</td>
</tr>
<tr>
<td>force=y</td>
<td>n</td>
<td>force output to &lt;file&gt;</td>
</tr>
<tr>
<td>verbose=y</td>
<td>n</td>
<td>print messages about dumps</td>
</tr>
</tbody>
</table>
Sample hprof usage

```
java -Xrunhprof:cpu=samples,depth=6,heap=sites
or
java -Xrunhprof:cpu=old,thread=y,depth=10,cutoff=0,format=a
ClassName
```

- Takes samples of CPU execution
- Record call traces that include the last 6/10 levels on the stack
- Only record "sites" used on heap (to keep output file small)

```
java -Xrunhprof ClassName
```

- Takes samples of memory/object usage

- After execution, open the text file `java.hprof.txt` in the current directory with a text editor
hprof visualization tools

• CPU samples
  – critical to see traces to modify code
  – hard to read - far from the traces in the file
  – HPjmeter analyzes java.hprof.txt visually
    • http://software.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPJMETER

  – another good tool called **PerfAnal** builds
    and navigates the invocation tree
  – download PerfAnal.jar, and:
  ```bash
  java -jar PerfAnal.jar ./java.hprof.txt
  ```

• Heap dump
  – critical to see what objects are there, and who points to them
  – HPjmeter or **HAT**: https://hat.dev.java.net/
TPTP

• a free extension to Eclipse for Java profiling
  – easier to interpret than raw hprof results
  – has add-ons for profiling web applications (J2EE)
Profiler results

• What to do with profiler results:
  – observe which methods are being called the most
    • These may not necessarily be the "slowest" methods!
  – observe which methods are taking most time relative to others

• Warnings
  – CPU profiling slows down your code (a lot)
    • design your profiling tests to be very short
  – CPU samples don't measure everything
    • doesn't record object creation and garbage collection time
  – Output files are very large, esp. if there is a heap dump
**Garbage collection**

- **garbage collector**: A memory manager that reclaims objects that are not reachable from a root-set

- **root set**: all objects with an immediate reference
  - all reference variables in each frame of every thread's stack
  - all static reference fields in all loaded classes
Profiling Web languages

- **HTML/CSS**

- **JavaScript**

- **Ruby on Rails**
  - ruby-prof --printer=graph_html --file=myoutput.html myscript.rb

- **JSP**
  - x.Link: [http://sourceforge.net/projects/xlink/](http://sourceforge.net/projects/xlink/)

- **PHP**
JavaScript optimization

- JavaScript is ~1000x slower than C code.
- Modifying a page using the DOM can be expensive.

```javascript
var ul = document.getElementById("myUL");
for (var i = 0; i < 2000; i++) {
    ul.appendChild(document.createElement("li"));
}

- Faster code that modifies DOM objects "offline":

```javascript
var ul = document.getElementById("myUL");
var li = document.createElement("li");
var parent = ul.parentNode;
parent.removeChild(ul);
for (var i = 0; i < 2000; i++) {
    ul.appendChild(li.cloneNode(true));
}
parent.appendChild(ul);```