Coding Techniques

Code Complete + extras

Layout of Source Code

• Good layout affects how easy it is to understand the code, review it, and revise it months after it was written
• Picky details?
  – Attention to such details makes a difference in the quality and maintainability of the code!
  – Must be done during initial construction
• Team should agree on a style before coding begins
import java.io.*; class OutputStars {
    public static void main(String[] args) throws Exception {
        BufferedReader inFromUser = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        int n;
        int i, j;
        System.out.println("Enter a number greater than 1:");
        s = inFromUser.readLine();
        n = Integer.parseInt(s);
        if (n < 1) {
            System.out.println("Number must be positive.");
        } else {
            for (i = 1; i <= n; i++) {
                for (j = 1; j <= i; j++) {
                    System.out.print("*");
                }
                System.out.println();
            }
        }
    }
}

Layout

- Fundamental theorem of formatting
  - Good visual layout shows the logical structure of the program
  - Making the code look pretty is worth something, but less than showing the code's structure
  - Lots of techniques, match up curly braces, group related items together, e.g. 3*4+4*5, indent code, use whitespace such as spacing and blank lines
  - Bad: if (x<3)
    a=1;
    b=2;
    c=3;
    d=4;
Religion Wars

• Layout approaches religion on how formatting should be done among programmers
• Key points:
  – Accurately represent the logical structure of the code
  – Consistently represent the logical structure of the code
  – Improve readability
  – Withstand modifications

Whitespace

• Could you read a book as easily if there were no whitespace? Consider the gutters and margins
• Blank Lines
  – Help group related statements
  – Study in 1990 found that the optimal number of blank lines in a program is 8-16%
    • Above 16% and debug time increased dramatically
• Alignment
  – Align elements that belong together, e.g. = in columns
• Indentation
  – 1983 study, subjects scored 20-30% higher on a comprehension test when programs had 2-4 space indentation scheme vs. no indentation
  – Second lowest scores on six-space indentation! Even though subjects thought it looked most pleasing, less effective than 2-4 space indentation
Layout Styles

• Pure blocks
  if (...) begin
     statement1;
     statement2;
  end if

• Shortened Pure blocks
  if (...) begin
     statement1;
     statement2;
  end

• Endline
  while (x==y) do {
     statement 1;
     statement 2;
  }

Using Only One Statement Per Line

• Several statements on one line
  – Takes up fewer screens of space
  – Might be OK for related statements, e.g. initialization

• Better to leave one statement per line
  – Doesn't hide complexity of a line
  – Some optimizing compilers use syntax as clues for optimization
  – Programmer forced to read left-right and top-bottom
  – Harder to find errors if in the middle of the line
  – Harder to step through code with a debugger
  – Harder to comment out individual statements
Declaring Functions

• Recommendation: Put each parameter on a line

```c
int SomeFunction(int numEmployees,
                 EList Employees,
                 File InputFile,
                 Rec dataRecord )
```

Extra work but holds up better under modification

Debugging

• For many programmers, debugging is the hardest part of programming

• First bug, Mark I computer:
Some are better than others

- Study of twelve programmers with at least four years of experience
- Fastest three programmers:
  - Average debug time: 5 minutes
  - Average number of errors not found: 0.7
  - Average number of errors made correcting errors: 3.0
- Slowest three programmers:
  - Average debug time: 14.1 minutes
  - Average number of errors not found: 1.7
  - Average number of errors made correcting errors: 7.7
- Use errors as opportunities
  - Learn about the program, kinds of mistakes you make, how you fix errors

Ineffective Debugging

- Guessing
  - Scatter print statements and logic changes until it works
  - More exciting without making backups
  - Learn to use your integrated debugger!
- Don’t waste time trying to understand the problem
- Fix the error with the most obvious patch

\[
X = \text{Compute}(Y); \\
\text{If } (Y == 17) \ X=25.15; \quad \text{// Was getting wrong answer for 17}
\]

- Debugging by superstition
  - Full moon?
  - Re-type program, mysterious whitespace?
  - Compiler’s fault?
Scientific Debugging

• In general
  – Gather data through repeatable experiments
  – Form a hypothesis that accounts for as much relevant data as possible
  – Design an experiment to test your hypothesis
  – Prove or disprove the hypothesis
  – Repeat as needed
• For programming
  – Stabilize the error
  – Locate the source of the error
  – Fix the error
  – Test the fix
  – Look for similar errors

Tips on Finding Errors

• Refine the test cases that produce the error
• Reproduce the error several different ways
• Use the results of negative tests
• Brainstorm for hypothesis
• Narrow the suspicious region of code
• Be suspicious of routines that have had errors before
• Check code that’s changed recently
• Expand the suspicious region of code
• Integrate incrementally
• Check for common errors
• Talk to someone else about the problem
• Take a break
Tips of Fixing Errors

• Understand the problem before you fix it
• Understand the program, not just the problem
• Confirm the error diagnosis
• Relax
• Save the original source code
• Fix the problem, not the symptom
• Make one change at a time
• Check your fix
• Look for similar errors

Code-Tuning Strategies

• Code tuning is the practice of modifying correct code to make it run more efficiently
• Less of a factor in many of today’s systems, particularly business software
• Problem: Efficient code isn’t necessarily better code
Code Tuning Misconceptions

• Reducing the lines of code in a HLL improves the speed of the resulting machine code
  – FALSE
  – Usually more lines is faster due to pipelining
  – Example:
    ```c
    for (i=0; i<5; i++) a[i]=i;  Time: 0.379
    vs.
    a[0]=0;  Time: 0.051
    a[1]=1;
    a[2]=2;
    a[3]=3;
    a[4]=4;
    ```

Code Tuning Misconceptions

• Certain operations are probably faster or smaller than others
  – FALSE!
  – No room for probably, changes with compilers and languages
  – Can reduce portability
• You should optimize as you go
  – FALSE!
  – Almost impossible to identify bottlenecks before a program is working
  – Focusing on performance detracts from other objectives
• A faster program is just as important as a correct one
  – FALSE!
  – Easy to make a fast program that is not correct
Pareto Principle

- **80/20 Rule**
  - You can get 80 percent of the result with 20 percent of the effort

- **Also applies to program optimization**
  - Usually the part that needs to be perfected and optimized is quite small
  - Working toward perfection may prevent completion

- **Measurement**
  - It pays to measure your code to find the hot spots
  - Don’t assume the hot spots are in a particular place

Matrix Summation Example

- **C example of straightforward code**

```
sum = 0;
for (row=0; row<rowCount; row++)
{
    for (col = 0; col < colCount; col++)
    {
        sum += matrix[row][column];
    }
}
```

Every access to a 2D array requires computing base + row*sizeof(row)+column
For a 10x10 matrix, that is 100 multiplications and additions plus loop overhead!

```
sum = 0;
elemeentPtr = matrix;
lastElemPtr = matrix[rowCount-1][colCount-1]+1;
while (elementPtr < lastElemPtr)
{
    sum += *(elementPtr++);
}
```

Speedup results? 0. Even with bigger matrices. Compiler had already optimized the first code well enough to match the second.
Common Sources of Inefficiency

- I/O operations
  - Move to random if sequential? Cache?
- Formatted print routines
- Floating point ops
  - Use integers when possible
- Paging
  Consider a machine that stores data by rows, with 1K pages

```plaintext
for col:=1 to 1000
  for row:=1 to 5
    table[row,col]:=0;
for row:=1 to 5
  for col:=1 to 1000
    table[row,col]:=0;
```

- System calls

Code-Tuning Techniques

- Loops
  - Good source of hotspots since loops may run many times
  - Unswitching
    - Make a decision outside the loop if possible
    - Usually means turning the loop inside-out
    - Example:

```plaintext
Unswitched loop: (Disadvantages?)

```
Loop Optimization

• Jamming or Fusion
  – Combining two loops that operate on the same set of elements

```c
for (i=0; i<count; i++) {
    name[i] = ";
}
```

```c
for (i=0; i<count; i++) {
    num[i] = 0;
}
```

Dangers of jamming? Relatively small time increase, up to 4%

Loop Optimization

• Loop Unrolling
  – Directly compute code that would normally be done by the loop
  – Good way to exploit parallelism, pipelining

```c
for (i=1; i<count; i++) {
    a[i] = i;
}
```

```c
for (i=1; i<count-4; i+=4) {
    a[i] = i;
    a[i+1]=i+1;
    a[i+2]=i+2;
    a[i+3]=i+3;
}
for (; i<count; i++)
    a[i]=i; // Catch leftovers
```

21-28% increase in speed
Loop Optimization

- Minimize Work Inside Loops
  - Precompute as much as possible

```c
for (i=1; i<count; i++) {
    a[i] = i*j*k*l*m*n;
}
```

```c
z = j*k*l*m*n;
for (i=1; i<count; i++) {
    a[i] = i*z;
}
```

Generally small increase in performance, most compilers can do a similar optimization on its own

- Also can try strength reduction

```c
for (i=1; i<count; i++) {
    a[i] = i*j*k*l*m*n;
}
```

```c
increment = j*k*l*m*n;
incAmount = increment;
for (i=1; i<count; i++) {
    a[i] = incAmount;
    incAmount += increment;
}
```

Logic

- Stop testing when you know the answer

```c
for (i=1; i<count; i++) {
    if (a[i] == target)
        found=true;
}
```

```c
for (i=1; i<count; i++) {
    if (a[i] == target)
        found=true;
        break;
}
```

- Order tests by frequency

```c
switch (inputChar) {
    case '+': ...
    case '-': ...
    ...
}
```

Appplies to switch and if-then-else

Particularly noticeable inside loops
Logic

• Substitute Table Lookup for Complicated Expressions

• Example:

```c
If ((A && !c) || (A && B && C))
    Class = 1;
else if ((B && !A) || (A && C && !B))
    Class = 2;
else if (C && !A && !B)
    Class = 3;
Else
    Class = 0;
```

```c
static int ClassTable[2][2][2] =
{  /* B/C   B/C   C   */
    0,      3,   2,     2,    /*  !A  */
    1,      2,    1,     2 }; /*  A    */
Class = ClassTable[A][B][C];
```

Arrays

• Minimize Array References

```c
for (discount = 0; discount < numD; discount++)
{
    for (rate = 0; rate < numR; rate++)
    {
        rateTbl[rate] = rateTbl[rate] * discountTbl[discount];
    }
}
```

`discountTbl[discount]` is unchanged the entire inner loop:

```c
discountTbl[discount] is unchanged the entire inner loop:

for (discount = 0; discount < numD; discount++)
{
    thisDiscount = discountTbl[discount];
    for (rate = 0; rate < numR; rate++)
    {
        rateTbl[rate] = rateTbl[rate] * thisDiscount;
    }
}
```
Precompute

• Initialize at compile time, reduce strength, eliminate common sub expressions

```javascript
for (i=1; i<count; i++) {
    a[i] = Math.pow(log(i) / log(3), 2);
}
const LOG3 = log(3);
for (i=1; i<count; i++) {
    a[i] = (log(i) / LOG3) * (log(i) / LOG3);
}

const LOG3 = log(3);
for (i=1; i<count; i++) {
    double unSquared = (log(i) / LOG3);
    a[i] = unSquared * unSquared;
}
```

Assembly

• Use inline assembly for critical routine
• Generally preferred to avoid hard-to-read code
Managing Construction

• Tips if you are ever in charge of other programmers
  – And also some data you can use to report to your manager
• Encourage Good Coding
  – Standards shouldn’t be imposed if can be avoided
    • Assign two people to every part of the project
      – Buddy system, guarantee at least two people think it works and is readable
    • Review every line of code
      – Peer review → peer pressure to get it right
      – Best known practice, share coding techniques
    • Require code sign-offs
      – Before code considered complete, senior tech must sign-off
    • Route good code examples for review
    • Emphasize that code listings are public assets
      – Lessens the “my code” as private property view
    • Reward good code

Configuration Management

• The practice of handling changes systematically so that a system can maintain its integrity over time
• Change control
• Software Design Changes
  – Follow a formal change-control procedure
  – Establish a change-control board
    • Submit change request, review requests, approve/reject/defer
  – Handle change requests in groups
  – Estimate the cost of each change
  – Be wary of major changes
• Code Changes
  – Use source code version control software
What If You’re Behind

- Hope you’ll catch up
  - Hardly ever the case
- Add more people
  - Usually makes the software later, not faster
  - More people to train, communicate with, familiarize
  - Exception: if new people can be partitioned in a way to perform tasks independently, the people can help
- Reduce the scope of the project
  - Eliminating a feature eliminates design, coding, debugging, testing, documentation
  - Another option is to back off requirements or implement a cheaper version of the same functionality

Treating Programmers as People

- How do programmers spend their time?
- Bell Labs study

<table>
<thead>
<tr>
<th>Activity</th>
<th>Code</th>
<th>Business</th>
<th>Personal</th>
<th>Meetings</th>
<th>Training</th>
<th>Misc/Misc</th>
<th>Manuals</th>
<th>Operating Misc</th>
<th>Test</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Talk or Listen</td>
<td>4%</td>
<td>17%</td>
<td>7%</td>
<td>3%</td>
<td></td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td>32%</td>
</tr>
<tr>
<td>Manager</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>Phone</td>
<td>2%</td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3%</td>
</tr>
<tr>
<td>Read</td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2%</td>
<td>2%</td>
<td></td>
<td></td>
<td>18%</td>
</tr>
<tr>
<td>Write</td>
<td>13%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1%</td>
<td></td>
<td></td>
<td>14%</td>
</tr>
<tr>
<td>Away</td>
<td>4%</td>
<td>1%</td>
<td>4%</td>
<td>6%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15%</td>
</tr>
<tr>
<td>Walk</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td></td>
<td></td>
<td>1%</td>
<td></td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>Misc</td>
<td>2%</td>
<td>3%</td>
<td>3%</td>
<td></td>
<td></td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>11%</td>
</tr>
<tr>
<td>Totals</td>
<td>35%</td>
<td>29%</td>
<td>13%</td>
<td>7%</td>
<td>6%</td>
<td>5%</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Variation in Performance and Quality

- Not surprisingly, some people are better programmers than others
- Study of programmers with 7 years experience
  - Ratio of initial coding time between best and worst programmers was 20:1
  - Debugging time 25:1
  - Execution speed 10:1
- Order of magnitude differences among programmers
- Team Variation
  - Good programmers tend to cluster, as do bad programmers (1985 study)
  - Study of seven identical projects
    - Efforts expended varied by a factor of 3.4:1 and program size by 3:1
  - Boehm: 80% of the contribution comes from 20% of the contributors
    - 15th percentile of programmers ranked by ability take four times as long as a team ranked in the 90th percentile
- Bottom line
  - Pick your personnel carefully, immediate payoff depending on the hire

Managing Your Manager

- Telling your manager what to do rather than the other way around
  - The trick is to do it so your manager thinks that you are the one being managed!
  - Some approaches
    - Refuse to do what your manager tells you, and insist on doing your job the right way
    - Pretend to do what your manager wants you to do, and secretly do it the right way.
    - Plant ideas for what you want to do, then wait for your manager to have a brainstorm (your idea) about doing what you want to do.
    - Find another job.
  - The best one
    - Educate your manager about the right way to do things.
Personal Character

• The personal character of programmers has received only a little attention
  – Dijkstra, 1965: “Programming Considered as a Human Activity”

• But this should receive more attention
  – Electrical Engineer: Knowledge of circuits, conductivity, how to use an oscilloscope, etc.
  – Software Engineer: Primary tool is YOU to design and construct the system

Personal Character Off Topic?

• Inwardness of programming makes personal character especially important
  – Ever program at odd hours? Burned out?
  – Programming work is mostly unsupervisable because nobody really knows what you’re working on unless looking over your shoulder all day
  – Often employer not in a position to judge if you’re good, it’s up to you to be responsible to be good or great
  – Character makes a difference; if you can’t change your intelligence at least you can change your character!
You don’t have to be super intelligent?

- Nobody is really smart enough to understand everything
  - Most programming compensates for limited size of stuff in our skulls
  - Best programmers realize how small their brains are; they are humble
  - Worst programmers refuse to accept that their brains aren’t equal to the task; egos keep them from being great programmers
  - The more you learn to compensate for your small brain, the better programmer you will be and the more humble you are the faster you will improve

Compensation Examples

- Decomposing a system
  - Makes it easier for humans to comprehend, whether structured, top-down, or object-oriented
- Conducting reviews, inspections, tests compensates for human fallibilities
  - Originated as part of “egoless” programming
- Keeping routines short helps reduce mental workload
- Using conventions can help free your brain from relatively mundane aspects of coding

- The humble programmers who compensate for their fallibilities write code that’s easier for themselves and others to understand and with fewer errors.
**Curiosity**

- Ok, so hopefully you admit your brain is too small to understand most programs and you need a way to compensate…

- Curiosity about technical subjects is a must to become a superior programmer
- Technical environment changes every 5-10 years, if you aren’t curious to keep up with the changes you will go the way of COBOL and punch cards

**Actions to exercise curiosity**

- Build your awareness of the development process
  - From reading, own observations
- Experiment
  - With development process and coding, write tests for new concepts, execute in debugger
- Analyze and plan before you act
- Learn about successful projects (or why projects were unsuccessful)
  - Rarely done, most people wouldn’t use their recreational time to scrutinize long code listings that work (or don’t work)
  - But engineers study the Tacoma Narrows bridge, or architects study Frank Lloyd Wright
- Read manuals, books, periodicals
Intellectual Honesty

• Maturing as a programming professional is developing an uncompromising sense of intellectual honesty. Examples:
  – Refusing to pretend you’re an expert when you’re not
  – Admitting your mistakes
  – Trying to understand a compiler warning rather than suppressing the message
  – Clearly understand your program – not compiling to see if it works
  – Provide realistic status reports
  – Provide realistic schedule estimates and holding your ground when management asks you to change them (or tricking management to win a project).

Communication and Cooperation

• Truly excellent programmers learn how to work and play with others
  – This includes writing readable code
• Most good programmers enjoy making programs readable, given enough time, although there are a few holdouts
  – Level 1: Beginner
    • Capable of using basic capabilities, e.g. loops, conditionals, write routines
  – Level 2: Intermediate
    • Capable of basic routines of multiple languages
  – Level 3: Specialist
    • Expertise in a language or environment or both, many stuck here
  – Level 4: Guru
    • Level 3 plus recognizes 85% of programming is communicating with other people
    • Only 30% of an programmer’s time is spent working alone, on average
    • Guru writes crystal clear code, documents it, results in guru status
Creativity and Discipline

• “When I got out of school, I thought I was the best programmer in the world. I could write an unbeatable tic-tac-toe program, use five different computer languages, and create 1000 line programs that WORKED. Then I got out into the Real World. My first task was to read and understand a 200,000 line Fortran program, then speed it up by a factor of two. Any Real Programmer will tell you that all the structured coding in the world won’t help you solve a problem like that – it takes actual talent.”
  – “Real Programmers Don’t Write Pascal”

Creativity and Discipline

• Tools and methods to emphasize human discipline (e.g. standards, conventions) have been especially effective
  – 15 year NASA study, 1990
• Highly creative people can still have discipline
  – Myth that discipline stifles creativity
  – Michelangelo divided the Sistine Chapel into symmetric collections of geometric forms, zones corresponding to Platonic stages. Self-imposed structure for human figures
  – Software engineers can impose similar discipline for requirements, design, testing
Laziness

- Laziness manifests itself in several ways
  - Deferring an unpleasant task
    - E.g., defer data entry, futz on other items first
    - True laziness
  - Doing an unpleasant task quickly to get it out of the way
    - Enlightened laziness – spending smallest possible time on something unpleasant
  - Writing a tool to do the unpleasant task so you never have to do the task again
    - Most productive if you ultimately save time, long-term laziness
- Don’t mask laziness as “hustle” or just doing something to look busy

Characteristics that don’t matter as much as you think

- Persistence
  - Depending on the situation, can be an asset or a liability
  - Stuck on a new piece of code: hardly ever a virtue, try redesigning or try an alternative approach, or come back later
  - Good idea to take a break if no progress after 15 minutes
  - Hard to know when to give up but it’s essential that you ask
Characteristics that don’t matter as much as you think

• Experience
  – Value of hands-on experience compared to book learning is smaller in software development compared to many other fields
    • Basic knowledge changes rapidly in SW Dev
    • Coding habits effective for COBOL not necessarily effective for Java
  – Easy to draw wrong conclusion from experience
    • “Five years of C++” mostly meaningless, if C++ not learned after a year or two, another three years makes little difference
  – Advantage goes to the young, hungry programmer!

Characteristics that don’t matter as much as you think

• Gonzo Programming
  – “If you haven’t spent at least a month working on the same program – working 16 hours a day, dreaming about it during the remaining 8 hours of restless sleep, working several nights straight through trying to eliminate that “one last bug” from the program – then you haven’t really written a complicated computer program. And you may not have the sense that there is something exhilarating about programming.”
    • Edward Yourden
  – Tribute to programming machismo is bunk and even a recipe for failure. May help your ego but how about the time spent fixing all the bugs you wrote during those all nighters?
Habits

• How often do you think about how you format indented loops?
  – Probably you thought about it when you first learned the technique, but now it is habit
  – Once habits are learned, they are hard to break or question
• Examples
  – Programmers tend to check loop indices carefully, but not check assignment statements so carefully
  – You’re looking for ways to make code readable, or fast, or you’re not
  – You’re regularly testing code incrementally as changes are made
• When you learn something new, it will be to your benefit to learn it the right way so it becomes an easy good habit instead of a bad habit

Job Advice

• Still CS jobs out there at decent salaries
• National Association of Colleges and Employers (NACE)
  – Ave starting salary for CS graduates in 2006 is $50,046
  – In 2004 : $48,656
  – In 2003 : $44,679
  – Down from the boom of 2001 when it was $52,723
Looks like supply of CS grades getting tighter

- www.cra.org/statistics

Job Seeking Advice

- Regularly update your resume
- Internet presence
  - Employers will google you, build a web page
  - Could include projects you’ve worked on, e.g. expose your senior project
  - Postings to mailing lists, discussion boards
- Learn a hot technology
  - Helps marketability, bot resume searches
- Learn a hot methodology
  - E.g. UML
Job Seeking Advice

• Pitch in on an open source project
  – Tons of projects out there looking for programmers

• Statistically, big companies pay more but don’t forget the little companies or freelance work

• Learn to use software tools
  – Version control, IDE, bug trackers, profilers

Job Seeking Advice

• Read every day about the field
  – Tons of programming and technology based mailing lists, news services

• Write some code every day
  – Or your skills will decline

• Build and rely on your network of people
  – Ask for help and give help when you can

• If invited for an interview, do your homework
  – Common interview questions
    http://maxnoy.com/interviews.html
  – Research the firm, generate questions
Or…… there’s Graduate School

- MBA
  - Good choice to help move up the corporate ladder, particularly into management
- MS
  - Good choice for technical path, potential for management, higher starting salary
- Ph.D.
  - Potentially highest starting salary but fewer career choices

Grad school is not for everyone

- Plenty of smart people go right to industry
  - Learn on the job
  - Advance within company or hop jobs
  - Entrepreneurs cannot afford to wait
- For some, grad school provides
  - A way to one-up your peers on the resume (degree as status symbol)
  - A fast track to a job (faster than working your way up)
  - Unique opportunities (no other way to be professor)
  - A great opportunity to focus
What is Graduate School Like?

• A professor’s perspective…
  – At research universities, the professor runs a small company
    • Product: Invents and develops long-range research
    • Customer: typically Federal Government (National Science Foundation, Defense Advanced Research Projects Agency – DARPA)
    • Annual Revenue: $300,000 - $1,000,000
    • Employees: Grad students
  – At teaching universities, the professor runs a small fiefdom
    • Serfs: Students
    • Most time spent teaching courses, performing university service (committees, curriculum, etc.)
    • Some of the above but at a smaller scale

What is Graduate School Like?

• M.S. Degree
  – 1.5 - 3 years
  – Coursework similar to senior-level undergraduate courses
  – Usually provided an opportunity to specialize
    • Can easily start degree without selecting area of focus
    • Good schools provide opportunity to take many focused courses in your favorite area
  – Research (in form of thesis) may be required
  – Tuition and stipend are possible
    • (full tuition + $15k / 9 mos + summer job)
What is Graduate School Like?

• Ph.D. Degree
  – 4 - 7 years
  – Similar coursework to Master’s Degree plus seminars and courses related to research
  – Research required
  – Tuition and stipend scholarships are standard
    • ($15k / 9 mos + some summer jobs)
  – Required to specialize
    • Helpful to know research interests from day one to expedite selection of research focus
    • Many select research focus after starting school

Is Graduate School for You?

• What are your career goals?
  – Sick and tired of school
  – Learn on the job (job hopping)
  – Entrepreneur
  – Technology management (manage engineers)
  – Professor
Is Graduate School for You?

– Do you enjoy learning - becoming an expert?
  • PhD makes you world’s expert in foo
– Do you like being a big fish in a small pond?
  • Question applies to job and school options
– Do you prefer constancy or change?
  • Higher degrees are entree to management and provide you with more control
– Financial and family situation

Is Graduate School for You?

• Degree pros and cons
  – Bachelor’s Degree
    • Good starting salary ($45-$55k) but peaks early
    • More job openings
    • Opportunity to swap jobs or move to management
    • But many jobs are entry level
    • Less control of day-to-day tasks
    • Employer usually benefits from not promoting you
    • May become bored – have to hop jobs
Is Graduate School for You?

• Master’s Degree Benefits
  – Better starting salary ($50k and up)
  – Many job openings
  – Potential to start at management level
  – Opportunity to swap jobs
  – More control of day-to-day tasks

Is Graduate School for You?

• Master’s Degree Cons
  – Still not in charge of project
  – 1.5 - 3 years of lost wages (less if paid during school)
  – May become bored by repetitive tasks
  – May become frustrated by poor employees and lack of support from upper-level management
Is Graduate School for You?

- Ph.D. Degree Benefits
  - Potentially higher starting salary ($50k+)
  - Large amount of control over work
  - Opportunity to teach in a university
  - Management skills assumed
  - You’ll be an expert in ________

- Ph.D. Degree Cons
  - 3 - 5 years of income beyond the masters is lost
  - Overqualified to make large jumps between fields
  - It’s a lot of hard work with few clear paths
Total numbers in grad school

• www.cra.org/statistics

![Figure 3. Ph.D. Pipeline corrected for year of entry](image)

Total numbers in grad school

• www.cra.org/statistics

![Figure 2. PhD Production](image)
Total numbers in grad school

- www.cra.org/statistics

<table>
<thead>
<tr>
<th>Department, Rank</th>
<th>CS</th>
<th>CE</th>
<th>CS &amp; CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>US CS 1-12</td>
<td>781</td>
<td>80</td>
<td>10.4%</td>
</tr>
<tr>
<td>US CS 13-24</td>
<td>837</td>
<td>1</td>
<td>11.1%</td>
</tr>
<tr>
<td>US CS 25-36</td>
<td>566</td>
<td>0</td>
<td>7.5%</td>
</tr>
<tr>
<td>US CS Other</td>
<td>4,618</td>
<td>370</td>
<td>61.3%</td>
</tr>
<tr>
<td>Canadian</td>
<td>717</td>
<td>45</td>
<td>9.5%</td>
</tr>
<tr>
<td>US CE</td>
<td>11</td>
<td>425</td>
<td>0.1%</td>
</tr>
<tr>
<td>Total</td>
<td>7,530</td>
<td>921</td>
<td>8,451</td>
</tr>
</tbody>
</table>

How Do I Apply?

- Application packet generally consists of the following
  - Transcript
    - Important, but not much you can do about this now…
  - Letters of Recommendation
    - Important – make or break marginal cases
    - Establish relationships with professors, one might be from employer
  - Personal Statement
    - Somewhat important – think about what you like
  - GREs
    - Somewhat important - subject test is hard, but many do poorly.
How Do I Apply?

• Transcript
  – Your school’s reputation,
  – your grades
  – and your courses will speak for themselves
  – Schools are sympathetic to GPAs that improve over time and weaknesses in outside areas

How Do I Apply?

• Letters of Recommendation
  – These carry a great amount of weight
  – Help your letter writer by reminding him/her of significant interactions you have had
  – Help your letter writer by sharing your research interests so he/she may find ways to write a letter that complements your personal statement
How Do I Apply?

• Contact person at other school
  – This is very difficult
  – Might strike up an email conversation with prof from another school
  – Us profs get many such emails from Chinese and Indian students
    • Don’t sound desperate
    • Ask a reasonable question about the professor’s research – showcase your qualifications

How Do I Apply?

• Personal Statement
  – This is a great opportunity to stand out
  – Research the schools in which you are interested
  – Ask professors to explain research areas
  – Try to sound like a student with experience, focus, and initiative
  – Don’t limit your choices by writing something that makes you sound too focused (unless you are)
How Do I Apply?

• GRE’s
  – General test always required
    • General test is like SAT’s but slightly harder
  – Subject test frequently required
    • Subject test is very detail oriented
  – Study! Purchase old tests for practice!

Where Do I Apply

• US News and World Report Top 50
  – Try to upgrade
    • CS Grad School List
  – Try not to worry about the money
    • Most schools have similar packages for their students. Those who want funding can usually find it.
Soapbox (Kenrick’s Opinion)

• A Master’s Degree is most flexible
  – On average you’ll earn more over your lifetime with a MS than with a BS or perhaps even a Ph.D.
  – You’ll have more control over your day to day tasks and have a leg up in management
• Only get the Ph.D. if you are strongly compelled to get what it provides
• Don’t go to work and think you’ll come back to school – it’s too hard and almost never happens
• Always remember to consider cost of living adjustments when comparing salaries
  – Silicon Valley is expensive

Special Case

• Get employer-paid M.S. while working
  – Consider quality of school
    • If you weren’t working, is a better school possible?
    • A MBA degree from UAA might not be worth much to you if you are capable of CMU (won’t open doors)
  – Difficult to work and study – but you’re young and might not have time commitments
  – Consider that school will likely pay you too