The Art and Craftsmanship of Computer Programming\textsuperscript{1}

Xi Tan (tan19@purdue.edu)

October 22, 2013

\textsuperscript{1}Many resources are from the Web.
Outline

1. The Art of Computer Programming: A Structural Overview
   - Elegant in Theory: Data Structure and Algorithm
   - Beautiful in Language: Code and Document
   - Efficient in Performance: Compilers and Micro-architecture

2. The Craftsmanship of Computer Programming: Some Examples
   - C/C++
   - Python, Matlab, and R
   - Perl and Bash
The Art of Computer Programming

The software you make should be a piece of art, that is

- Elegant in Theory (Data Structure and Algorithm)
- Beautiful in Language (Code and Document)
- Efficient in Performance (Compilers and Micro-architecture)
Part I. Elegant in Theory: Data Structure and Algorithm
Data Structure

- array / list
- stack / queue
- tree / graph
- hash tables

Algorithms

- dynamic programming
- tree traversal (pre/in/post-order)
- graph search (BFS, DFS, shortest distance)
- sorting (bubble/merge/quick/selection/shell/heap/insertion)
- selection (median, $k^{th}$ element)
Two (main) types of algorithms:

- Non-recurrent algorithms
- Recurrent algorithms (easy for programmers but hard for computers)

\[
T(n) = aT(n/b) + f(n), \quad a \geq 1, b > 1
\]

\[\text{a subproblems of size } n/b\]
First Example

**Input**: a sequence of a large number (say, $10^6$) of *small* integers.

**Output**: a sequence of sorted numbers of the input.
Second Example

Given an unsorted array of length $N$.

- Find the $1^{st}$ smallest element.
- Find the $2^{nd}$ smallest element.
- Find the $k^{th}$, $k \leq N$ smallest element.

Recurrence

Runtime for problem size $n$

$$T(n) = aT(n/b) + f(n), \quad a \geq 1, b > 1$$

Cost of conquer step

$a$ subproblems of size $n/b$
Constants matter too!
Some good books to read:

- Concrete Mathematics
- Introduction to Algorithms
- Computer Systems: A Programmer’s Perspective
- Hacker’s Delight
- Unix Power Tools

Suggestions:

- Memorize 30 most useful algorithms (sorting, searching, etc. write them down without thinking.)
- Read good code (STL, open source code, etc.)
- Know 10 Problems that are NP (so you don’t waste your time)
Part II. Beautiful in Language: Code and Document
Code

- Choose the right tools: programming languages, editors, OS, and etc.
- The Unix Philosophy: write code that does one thing and does it well.
- Code Level: clear, consistent.
- Software Level: modularity, abstraction.

Document

- Write for others, and for yourself.
- Write descriptions for your functions, and comment on important (and/or obscure) statements.
Write good code.

If the probability of the stock going down is 0.8, I would like to sell it all and buy the stocks in my watch list that are climbing. There are many things you can improve in the following C statement:

\[
\text{if}(a=0.8)\{b=c/d\}
\]
When do you need to write a document?

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalves = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;          // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );  // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalves - ( x2 * y * y ) );  // 1st iteration
    //
    y = y * ( threehalves - ( x2 * y * y ) );  // 2nd iteration, this can be removed

    return y;
}
```
When do you need a document? (cont.)

The "magic number" [edit]

The selection of 0x5f3759df as a constant prompted much of the original speculation surrounding the fast inverse square root function. In an attempt to determine how a programmer might have originally determined that constant as a mechanism to approximate the inverse square root, Charles McEnery first determined how the choice of any constant R could give a first approximation for the inverse square root. Recalling the integer and floating point comparison from above, note that \( x \), our floating point number, is \( x = (1 + m_x)2^{e_x} \) and \( I_x \), our integer value for that same number, is \( I_x = E_x L + M_x \). These identities introduce a few new elements, which are simply restatements of values for the exponent and significand.

\[
m_x = \frac{M_x}{L} \quad \text{where} \quad L = 2^{n-1-b},
\]

\[
e_x = E_x - B \quad \text{where} \quad B = 2^{b-1} - 1.
\]

The illustration from McEnery 2007 proceeds:

\[
y = \frac{1}{\sqrt{x}}
\]

\[
\log_2(y) = -\frac{1}{2} \log_2(x)
\]

\[
\log_2(1 + m_y) + c_y = -\frac{1}{2} \log_2 (1 + m_x) - \frac{1}{2} c_x
\]

Taking the binary logarithm or \( \log_2 \) of both sides. The binary logarithm is the inverse function of \( f(n) = 2^n \) and makes the multiplied terms in the floating point numbers \( x \) and \( y \) reduce to addition. The relationship between \( \log_2 (x) \) and \( \log_2 (x^{-1/2}) \) is linear, allowing an equation to be constructed which can express \( x \) and \( y/0 \) (The input and first approximation) as a linear combination of terms.\(^{12}\) McEnery introduces a term \( \sigma \) which serves as an approximation for \( \log_2 (1+x) \) in an intermediate step toward approximating \( f \).\(^{13}\) Since \( 0 \leq x < 1 \), \( \log_2 (1+x) \approx x \), \( \log_2 (1+x) \approx x + \sigma \) can be defined. This definition offers a first approximation of the binary logarithm. For our purposes, \( \sigma \) is a real number bounded by \([0,1/3]\)—for an \( R \) equal to 0x5f3759df, \( \sigma = 0.0450461875791687011756 \).\(^{15}\)

\[
m_y + \sigma + c_y = -\frac{1}{2} m_x - \frac{1}{2} \sigma - \frac{1}{2} e_x
\]

Using the identities for \( M_x, E_x, B \) and \( L \):

\[
M_x + (E_x - B)L = -\frac{3}{2} \sigma L - \frac{1}{2} M_x - \frac{1}{2} (E_x - B)L
\]

Rearranging of terms leads to:

\[
E_x L + M_y = \frac{3}{2} (B - \sigma)L - \frac{1}{2} (E_x L + M_x)
\]
A Brief History of Programming Languages

- 1950: FORTRAN (good at numerical computation)
- 1964: BASIC (easy to learn but slow)
- 1970: PASCAL (good coding practices)
- 1972: C (efficient and powerful)
- 1983: C++ (C with Classes, slightly slower but evolutionary)
- 1984: Matlab (written in C with LAPACK)
- 1987: Perl (good at text processing)
- 1989: Bash (a very effective scripting language)
- 1991: Python (compact and structured code)
- 1993: R (written in C and FORTRAN)
- 1995: JAVA (simpler object model and fewer lower-level facilities; slow)
Speed Comparison of Programming Languages
Speed Comparison of Programming Languages (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Fortran GCC 4.5.1</th>
<th>Julia 12b1d5a7</th>
<th>Python 2.7.3</th>
<th>Matlab R2011a</th>
<th>Octave 3.4</th>
<th>R 2.14.2</th>
<th>JavaScript V8 3.6.6.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>0.28</td>
<td>1.97</td>
<td>46.03</td>
<td>1587.03</td>
<td>2748.74</td>
<td>275.63</td>
<td>2.09</td>
</tr>
<tr>
<td>parse_int</td>
<td>9.22</td>
<td>1.72</td>
<td>25.29</td>
<td>846.67</td>
<td>7364.87</td>
<td>353.48</td>
<td>2.55</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.65</td>
<td>1.37</td>
<td>69.20</td>
<td>133.46</td>
<td>3341.94</td>
<td>708.76</td>
<td>4.95</td>
</tr>
<tr>
<td>mandel</td>
<td>0.76</td>
<td>1.45</td>
<td>34.88</td>
<td>74.61</td>
<td>988.74</td>
<td>184.71</td>
<td>7.62</td>
</tr>
<tr>
<td>pi_sum</td>
<td>1.00</td>
<td>1.00</td>
<td>33.64</td>
<td>1.46</td>
<td>457.26</td>
<td>253.45</td>
<td>1.12</td>
</tr>
<tr>
<td>rand_mat_stat</td>
<td>2.23</td>
<td>1.95</td>
<td>29.01</td>
<td>7.71</td>
<td>31.04</td>
<td>12.66</td>
<td>5.53</td>
</tr>
<tr>
<td>rand_mat_mul</td>
<td>1.14</td>
<td>1.00</td>
<td>1.75</td>
<td>1.08</td>
<td>1.93</td>
<td>9.58</td>
<td>45.82</td>
</tr>
</tbody>
</table>

**Figure:** benchmark times relative to C (smaller is better).

C compiled by Clang 4.0, taking best timing from all optimization levels (-O0 through -O3).

The Python implementations of `rand_mat_stat` and `rand_mat_mul` use NumPy (v1.5.1) functions; the rest are pure Python implementations.
Conclusion

- If you want to implement something efficient, use C/C++.
- If you are at the design stage and want to play with various models, use Matlab/R.
- If you want a good balance in between, use Python.
- For routine work, use Bash.
- If you feel it hard to understand OOP, read a book on Java, but don’t use it unless your boss ask you to.

My personal pick: C, Matlab/R, Bash
Part III. Efficient in Performance: Compilers and Micro-architecture
Abstracted Microarchitecture: Example Core (2008)

- Throughput is measured in doubles/cycle
- Latency in cycles for one double
- 1 double = 8 bytes
- Rectangles not to scale

**Core 1**

- out of order execution
- superscalar
- fadd, fmul, ALU, load, store
- execution units
- instruction pool (up to 96 “in flight”)
- instruction decoder (up to 5 ops/cycle)

**ISA**

- internal registers
- 16 FP register
- L1 DCache
- L1 cache
- L2 cache
- Main Memory (RAM)
- Hard disk ~500 GB

**Memory hierarchy:**
- Registers
- L1 cache
- L2 cache
- Main memory
- Hard disk

**Core 2 Duo:**

- on die
- RAM
- Core 1
- Core 2
Linking

- Source file #1 → Assembler → Object file #1
- Source file #2 → Assembler → Object file #2
- Source file #3 → Assembler → Object file #3
- ... → Assembler → ...
- Program library
- Linker
- Executable file
Optimize Your Code (1)

Example: Data Type for Vectors

```c
/* data structure for vectors */
typedef struct{
    int len;
    double *data;
} vec;

/* retrieve vector element and store at val */
int get_vec_element(*vec, idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```
Optimize Your Code (2)

Example: Summing Vector Elements

```c
/* retrieve vector element and store at val */
int get_vec_element(*vec, idx, double *val)
{
    if (idx < 0 || idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

/* sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}
```

Overhead for every fp +:
- One fct call
- One <
- One >=
- One ||
- One memory variable access

Slowdown: probably 10x or more
Optimize Your Code (3)

Removing Procedure Call

```c
/** sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double val;

    for (i = 0; i < n; i++) {
        get_vec_element(v, i, &val);
        *res += val;
    }
    return res;
}

/** sum elements of vector */
double sum_elements(vec *v, double *res)
{
    int i;
    n = vec_length(v);
    *res = 0.0;
    double *data = get_vec_start(v);

    for (i = 0; i < n; i++)
        *res += data[i];
    return res;
}
```
Optimize Your Code (4)

Code Motion

- Reduce frequency with which computation is performed
  - If it will always produce same result
  - Especially moving code out of loop (loop-invariant code motion)

- Sometimes also called precomputation

```c
void set_row(double *a, double *b, 
    long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```c
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```
Optimize Your Code (5)

Strength Reduction

- Replace costly operation with simpler one
- Example: Shift/add instead of multiply or divide
  \[ 16 \times x \rightarrow x << 4 \]
  - Utility machine dependent
- Example: Recognize sequence of products

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```c
int ni = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
```
Optimizing Compilers

- Use optimization flags, *default is no optimization* (-O0)!
- Good choices for gcc: -O2, -O3, -march=xxx, -m64
- Try different flags and maybe different compilers
Example

double a[4][4];
double b[4][4];
double c[4][4]; // set to zero

/* Multiply 4 x 4 matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++)
            for (k = 0; k < 4; k++)
                c[i*4+j] += a[i*4 + k]*b[k*4 + j];
}

■ Compiled without flags:
  ~1300 cycles

■ Compiled with -O3 -m64 -march=... -fno-tree-vectorize
  ~150 cycles

■ Core 2 Duo
Optimizing Compilers

- Compilers are **good** at: mapping program to machine
  - register allocation
  - code selection and ordering (instruction scheduling)
  - dead code elimination
  - eliminating minor inefficiencies

- Compilers are **not good** at: algorithmic restructuring
  - For example to increase ILP, locality, etc.
  - Cannot deal with choices

- Compilers are **not good** at: overcoming “optimization blockers”
  - potential memory aliasing
  - potential procedure side-effects
The Craftsmanship of Computer Programming

Characteristics of a Pragmatic Programmer:
- Early adopter / Fast adapter
- Inquisitive
- Critical thinker
- Jack-of-all-trades (*master of a few*)

We will show some examples in:
- C/C++
- Python, Matlab, and R
- Perl and Bash
Part I. C/C++
Question: You start with $1. With each move, you can either double your money or add $1 to it. What is the smallest number of moves you have to make to get to exactly $200?
Answer: 200 is 11001000 in binary, so we need 9 moves.

1\textsuperscript{st} move: 1 \rightarrow 10
2\textsuperscript{nd} move: 10 \rightarrow 11
3\textsuperscript{rd} move: 11 \rightarrow 110
4\textsuperscript{th} move: 110 \rightarrow 1100
5\textsuperscript{th} move: 1100 \rightarrow 11000
6\textsuperscript{th} move: 11000 \rightarrow 11001
7\textsuperscript{th} move: 11001 \rightarrow 110010
8\textsuperscript{th} move: 110010 \rightarrow 1100100
9\textsuperscript{th} move: 1100100 \rightarrow 11001000
C/C++ (cont.)

Example 1: Some interesting binary tricks.
Use the following formula to turn off the rightmost 1-bit in a word, producing 0 if none (e.g., 01011000 → 01010000):

\[ x \&(x-1) \]
Example 2: Some interesting binary tricks. (cont.)
Similarly, the following formula can be used to test if an unsigned integer is of the form $2^n - 1$ (including 0 or all 1's):

$$x \& (x+1)$$
C/C++ (cont.)

Example 3: Some interesting binary tricks. (cont.)
Use the following formula to isolate the rightmost 1-bit, producing 0 if none (e.g., 01011000 → 00001000):

\[ x \& (-x) \]
Example 4: Some interesting binary tricks. (cont.)
Use the following formula to isolate the rightmost 0-bit, producing 0 if none (e.g., 10100111 → 00001000):

\[ \sim x \& (x+1) \]
Example 5: Some interesting binary tricks. (cont.)
Swap two integers (without the third variable):

\[
x = x \hat{\,} y;
\]
\[
y = x \hat{\,} y;
\]
\[
x = x \hat{\,} y;
\]
/* Bubble sort code */

#include <stdio.h>

int main()
{
    int array[100], n, c, d, swap;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);

    for (c = 0; c < (n - 1); c++)
    {
        for (d = 0; d < n - c - 1; d++)
        {
            if (array[d] > array[d+1]) /* For decreasing order use < */
            {
                swap = array[d];
                array[d] = array[d+1];
                array[d+1] = swap;
            }
        }
    }

    printf("Sorted list in ascending order:\n");

    for (c = 0; c < n; c++)
        printf("%d\n", array[c]);

    return 0;
}
Part II. Python, Matlab, and R
Python, Matlab, and R

Python

```python
1. myList=[43,21,12,80,3,2,35]
2. end=len(myList)-1
3. while (end!=-1):
4.     swapped=-1
5.     for i in range(0,end):
6.         if myList[i]>myList[i+1]:
7.             temp=myList[i]
8.             myList[i]=myList[i+1]
9.             myList[i+1]=temp
10.            swapped=i
11.        end=swapped
12.    print(myList)
```
Python, Matlab, and R (cont.)

Matlab

clc;
clear;

N = 10000000;
x = rand(N,1);
y = rand(N,1);
tic;
s = x'*y;
toc;
display(s);

tic;
s = 0;
for i = 1:N
    s = s + x(i)*y(i);
end;
toc;
display(s);

Elapsed time is 0.008378 seconds.
s =

2.4998e+06

Elapsed time is 0.140675 seconds.

s =

2.4998e+06
Python, Matlab, and R (cont.)

R

```r
> summary(lmfit)
Call:
  lm(formula = change ~ setting + effort)
Residuals:
     Min      1Q  Median      3Q     Max
-10.3475 -3.6426  0.6384  3.2250 15.8530
Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)  -14.4511    7.0938  -2.037 0.057516 .
setting        0.2706    0.1079   2.507 0.022629 *
effort         0.9677    0.2250   4.301 0.000484 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.389 on 17 degrees of freedom
Multiple R-Squared: 0.7381,  Adjusted R-squared: 0.7073
F-statistic: 23.96 on 2 and 17 DF,  p-value: 1.132e-05
```
Part III. Perl and Bash
Perl and Bash

Perl

```perl
1. # extract hours, minutes, seconds
2. if ($time =~ /\d\d:\\d\d:\\d\d\d/) { # match hh:mm:ss format
3.   $hours = $1;
4.   $minutes = $2;
5.   $seconds = $3;
6. }
```
Perl and Bash (cont.)

Bash

```
$ ls
this_has_text_to_find_1.txt
this_has_text_to_find_2.txt
this_has_text_to_find_3.txt
this_has_text_to_find_4.txt

$ rename 's/text_to_find/been_renamed/' *.txt
$ ls
this_has Been_renamed_1.txt
this_has Been_renamed_2.txt
this_has Been_renamed_3.txt
this_has Been_renamed_4.txt
```
Summary

1. The Art of Computer Programming: A Structural Overview
   - Elegant in Theory: Data Structure and Algorithm
   - Beautiful in Language: Code and Document
   - Efficient in Performance: Compilers and Micro-architecture

2. The Craftsmanship of Computer Programming: Some Examples
   - C/C++
   - Python, Matlab, and R
   - Perl and Bash
Thank You!