Introduction to Algorithms (1)

Michael L. Littman

September 1st, 1998

1 ADMINISTRATION

1.1 Course Background

Basic Information:

- CPS130: Introduction to the Design and Analysis of Algorithms
- Instructor: Michael Littman (me)
- Class meets twice a week in this room at this time.

Prerequisites:

- Four semesters of math listed. Should know notation, basic probability theory, the idea of limits, the concept of proofs.
- Knowledge of programming: How to express the solution to a problem in a computer language, how to structure complex programs in modules, recursion.
- Some modifying and running of C++ programs.

1.2 Grading

Class grade will be based on:

- written homework assignments (30%),
- participation (10%),
- two in-class exams (40%), and
- a final exam (20%).

1.3 Resources

Traditional:

- Textbook: "Big White Book" (useful resource)
- Handouts of lecture notes.
- TA office hours.
- My office hours (or by appointment).

Electronic:

- web page: http://www.cs.duke.edu/%7emlittman/courses/cps130/
- mailing list: send me your address!
- newsgroup: duke.cs.cps130 (for class discussion)
- email: Me (Michael Littman) (mlittman@cs.duke.edu), TA and UTAs (see web page)

1.4 Homeworks

Homework cycle:

- Homework problems mentioned during lecture (Tuesday or Thursday).
- Written (or online) answers due at the *beginning of class* the following Thursday.
- Partial credit for late homeworks at my discretion.
- We'll try to get them back to you by the following Thursday.

Philosophy:

- Practice trying out the concepts we discuss.
- Opportunity to get feedback on your work.
- Chance for me to see how you are participating.

Collaboration:

- homework: discuss, but write up separately!
- tests: do not discuss!!

1.5 Teaching Goals

Algorithms:

- Separate us from the "beasts."
- Are important practically.
- Give us a chance to "think" (puzzles, fun!).

Take Home Lessons:

- Algorithms matter,
- Some classic algorithms and their analyses,
- How to analyze algorithms
- How to invent new algorithms
- Empirical evaluation of algorithms.

2 ALGORITHM BACKGROUND

2.1 Need for Speed

World-wide web:

- Driving directions to Long Beach Island, NJ.: Database lookup, shortest path, range computation
- Zip+4 for my new address in Durham: spell correction, fast lookup, rule-based.
- Search for web pages mentioning a particular article.
- Compute and manipulate travel information for an upcoming trip: another kind of shortest path, database lookup.

It's gotta be fast or it's nearly useless.

2.2 Writing Fast Programs

How do you make a program fly?

- Careful program design is important.
- Tweaking loops and stuff can help.
- Most important and general thing is good *algorithm design*.

We'll study this process throughout the semester.
(Algorithm just means a way of doing something.)

2.3 Algorithm Development Algorithm

```
AlgorithmDesign(informal problem)
formalize problem (mathematically) [Step 0]
repeat
        devise algorithm [Step 1]
        analyze correctness [Step 2]
        analyze efficiency [Step 3]
        refine
   until algorithm good enough
return algorithm
```

2.4 The Need for Mathematics

Theory (math stuff) helps in three ways:

- formal notation for a problem,
- analyze correctness,
- analyze efficiency.

Sounds important, yes?
Efficiency:

- **time**,
- memory,
- processors.

2.5 Kinds of Studied Algorithms

The algorithms studied by researchers can be categorized as:

- A. How to do something faster.
- B. How to do something at all.
- C. Prove a point (to help us understand the mathematical underpinnings).

Courses often focus on C, for obvious reasons. I'm going to try hard to focus on A, which I think is the one that is the most useful for the average computer scientist.

3 ALGORITHM CASE STUDY

3.1 What Does This Do?

Assume a and b positive integers.

```
int Naive(int a, int b)
// Naive algorithm
{
   int x = a, y = b, z = 0;
   while (x > 0)
   {
      z += y;
      x--;
   }
   return z;
}
```

Animated example.

3.2 Algorithm Development

- 1. Devise algorithm.
- 2. Analyze correctness.
- 3. Analyze efficiency.

Homework: For Naive multiplication, carry out Step 2.

3.3 Russian Peasants' Algorithm

```
int Russian(int a, int b)
//  implements Russian Peasant Algorithm
{
   int x = a, y = b, z = 0;
   while (x > 0)
   {
      if ( 1 == x % 2)
         z += y;
      y <<= 1;
      x >>= 1;
   }
   return z;
}
```

3.4 Concrete Example

Compute 71×43:

x	y
71	[43]
35	[86]
17	[172]
8	344
4	688
2	1376
1	[2752]

Now sum.

$$\begin{array}{r} 2752 \\ 172 \\ 86 \\ 43 \\ \hline 3053 \end{array}$$

Animated example.

3.5 Why Does It Work? [Step 2]

- 1. Initially $xy + z = ab$.
- 2. Inductive Step
  - Let x, y, z be values at the top of the loop.
  - Assume $xy + z = ab$ holds.
  - Let x', y', z' be the values after the loop.
  - If x is odd:
    - $x' = (x - 1)/2$
    - $y' = 2y$
    - $z' = z + y$
    - $x'y' + z' = (x - 1)/2(2y) + z + y$
      $= (x - 1)y + z + y = xy - y + z + y = xy + z = ab$
  - if x is even:
    - $x' = x/2$
    - $y' = 2y$
    - $z' = z$
    - $x'y' + z' = (x/2)(2y) + z = xy + z = ab$
  - Therefore $x'y' + z' = ab$.
- 3. Variable x starts at at least one and decreases on each iteration down to zero. Therefore, algorithm terminates.
- 4. At termination, $xy + z = ab$, but $x = 0$, so $z = ab$.

3.6 Which Is Better?

How can we choose between the two algorithms (Naive and Russian)?
What do you think?
Animated comparison.

3.7 Issues

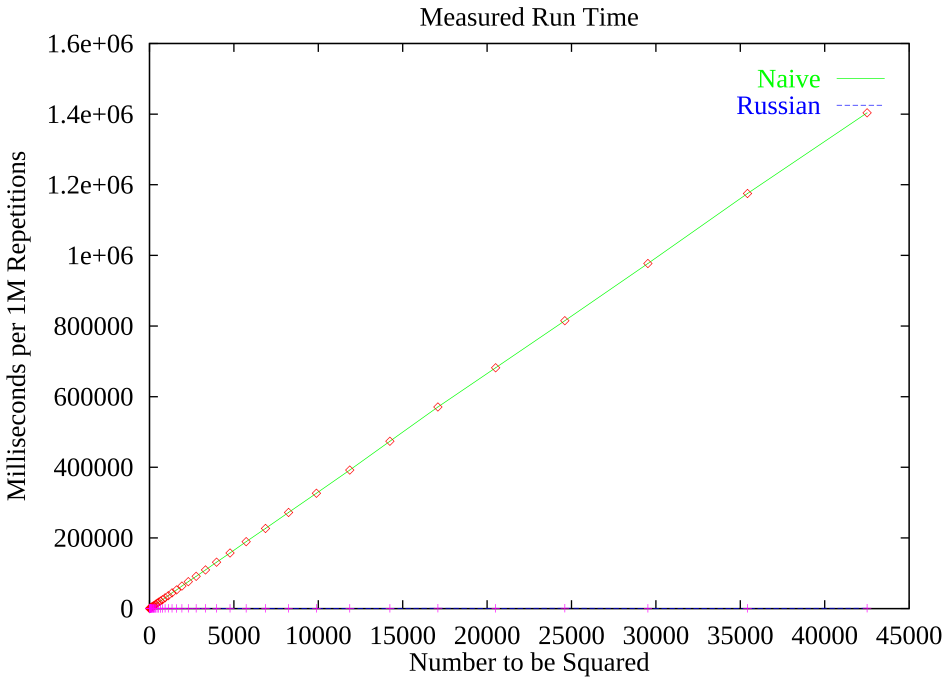
Depends on what it is for. Naive is perhaps clearer. Probably what matters is speed.
Which is faster? How can we find out? [Step 3]

4 EMPIRICAL EVALUATION

4.1 Experiment

Choose a bunch of numbers and square them. Measure run time.
For practical reasons, we'll rerun the algorithm a million times and measure the total time.

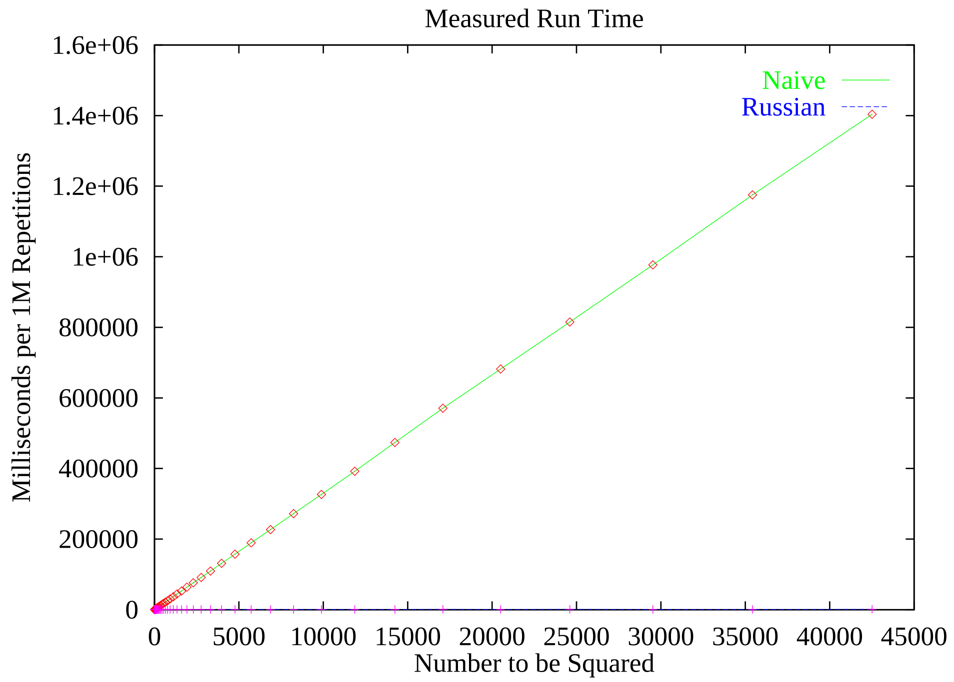
4.2 Results

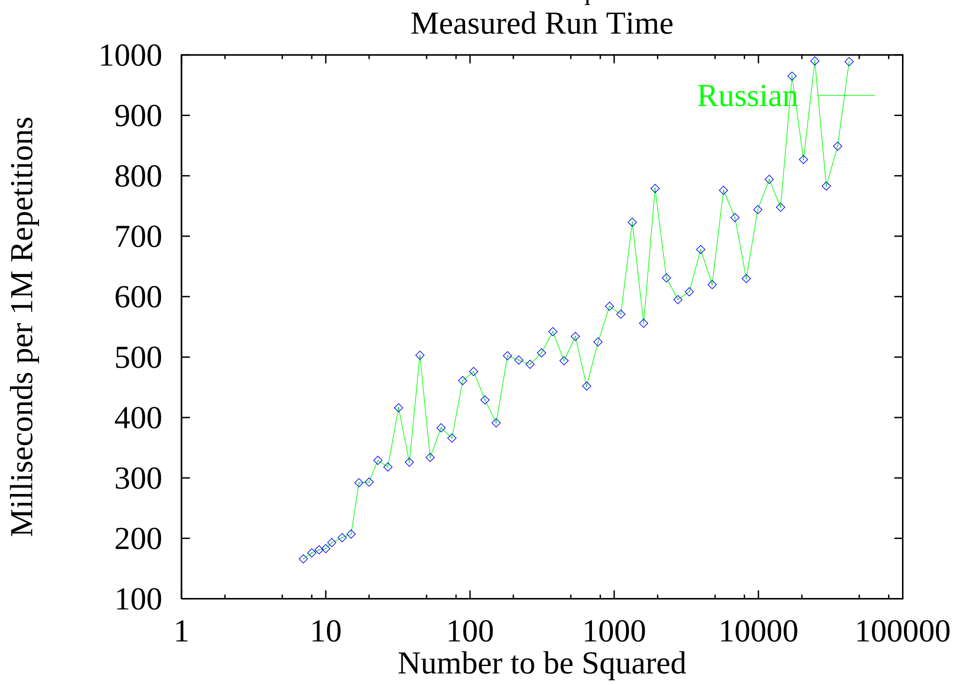


4.3 Discussion

For small numbers, Naive performs slightly better. For larger numbers Russian performs *much* better.
In fact, there's a detectable trend. Can we characterize it?

4.4 Growth Rates

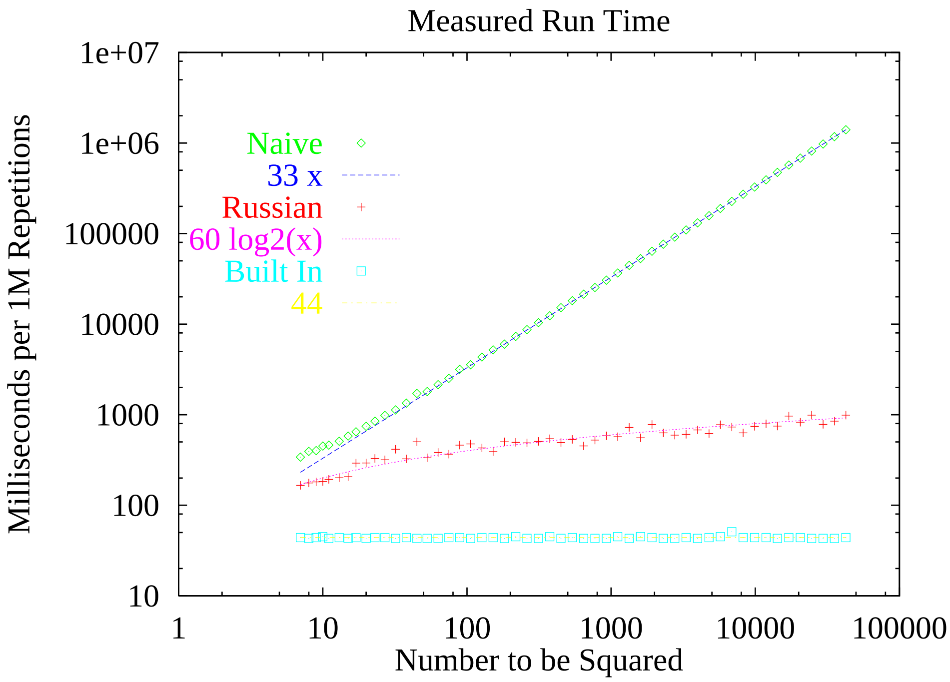




Guess the growth functions?

4.5 Direct Route

What if we just compute ab by letting the computer multiply the two numbers together? Takes approximately constant time (about 45 milliseconds per a million multiplications, independent of the size of the numbers being multiplied).



Rule #1 of Good Algorithm Design: If you already have a good subroutine to solve the problem, use it!

5 EFFICIENCY ANALYSIS

A simple experiment suggested that the time it takes to multiply a by itself

- using Naive is some linear function of a,
- using Russian is some logarithmic function of a.

As a result, it appears Russian outperform Naive on large problems.

5.1 Explaining the Growth Rates

No analytic function predicts these curves exactly: too many unmeasurable influences and complex relationships (caching, compiler, pipelining).
Furthermore, we can't use these algorithms to square numbers larger than a million or so because we run out of room in our integer variables.

5.2 Measuring Time

We'd like a method for taking a program and producing a mathematical function representing the amount of time it takes to run it.
We can't do this, in general.
However, we can begin to make simplifying assumptions that let us do this approximately.
In order of decreasing accuracy:

- Simplifying assumption #1: The time it takes to execute a simple statement like $x \leftarrow x + 1$ depends only on the statement (not the processor load, the amount of memory in the computer, the time of day, the number of users, the value of x...).
- Simplifying assumption #2: The time it takes to execute a pair of statements is the sum of the times to execute the individual statements (ignoring issues like pipelining, etc.).
- Simplifying assumption #3: Any simple statement takes "unit" time (evaluating any arithmetic expression, assigning a variable, initializing a loop, etc.).
- Simplifying assumption #4 (for later): Any constant-length (i.e., independent of the input) sequence of statements takes "unit" time.

This will be enough to get us started. We'll need to make some even stronger assumptions later!

5.3 Simple Programming Model

Time(Prog):

- Prog = statement; Prog': Time(statement) + Time (Prog')
- Prog = "var = expr": 1 + Time(expr)
- Prog = "expr1 op expr2": 1 + Time(expr1) + Time(expr2) for primitive ops
- Prog = "array[expr]": 1 + Time(expr)
- Prog = "for i = 1 to n, do Prog' ": n times Time(Prog')
- Prog = function call: Time(function)
- Prog = "while(expr), do Prog' ": Time(expr) + Time(Prog') times executions

Analyze by inspection, multiplication, recurrences, potential functions.

5.4 Algorithms as a Science

Science:

- Make Observations
- Build Theories

Theories are useful inasmuch as they predict reality.
In fact, no single theory is right for all classes of algorithms.

5.5 Experiment vs. Theory

Theory pros:

- general
- implementation independent
- input independent
- can be carried out before anything is built

Experiment pros:

- matches reality (time, task, computer system)
- less mathematical

6 ANALYZING RUSSIAN

6.1 Counting Iterations

From the simplifying assumptions, we can reason that the important determinant of runtime in Naive is the number of times the body of the loop is executed.
So, how many iterations does Naive do, as a function of a? Well, we subtract one from it over and over again until we get down to zero. How many times can we do that?
What about Russian? That one's a bit more interesting. There, we divide a by two (and round down) over and over again until we get down to zero. How many times can we do that?

6.2 Fun With Logarithms

Base 2 logarithms come up everywhere in this class (and CS).
Some characterizations of $\log(n)$:

- The number of times we can divide n by 2 before we get 1 or less.
- The number of bits in the binary representation of n.
- A way to turn multiplication into addition: $\log(ab) = \log(a) + \log(b)$.
- The inverse function of 2^n.

The function $\log(n)$ grows much more slowly than n or $1/100n$ or even $\sqrt{n}$.
The book lists a bunch of amazing properties of this function. I'll introduce them as needed.

6.3 Iteration Count for Russian

From this little aside, it is apparent that the number of iterations it takes to square a is $\log(a)$.
To be more precise, the exact number of iterations is $1 + \lfloor \log(a) \rfloor$.
Here, the "floor" function, $\lfloor x \rfloor$ is the largest integer not exceeding x (rounding down). This is in contrast to the "ceiling" function $\lceil x \rceil$, which rounds up.
We will prove this bound next time, and introduce another (recursive) way of writing the algorithm.