CS161 Programming Section

June 27, 2013

Hi everyone. My name is Andy; I'll be teaching these programming sections which are designed to supplement the 161 lectures. I'm afraid I'm not nearly as charismatic a lecturer as Keith, but that's OK because we'll be spending most of our time coding. As a reminder, while I hope these sections will be useful in helping you better understand the material, they are completely optional. Also, the plan is for section to be about 50 minutes long, but we do have the room reserved for another hour after that, so if you guys are interested, we can use that extra hour to cover additional problems.

Prerequisites

- Familiarity with C++ or Java (CS106 level)
- Basic knowledge of Linux command line (managing files and directories)

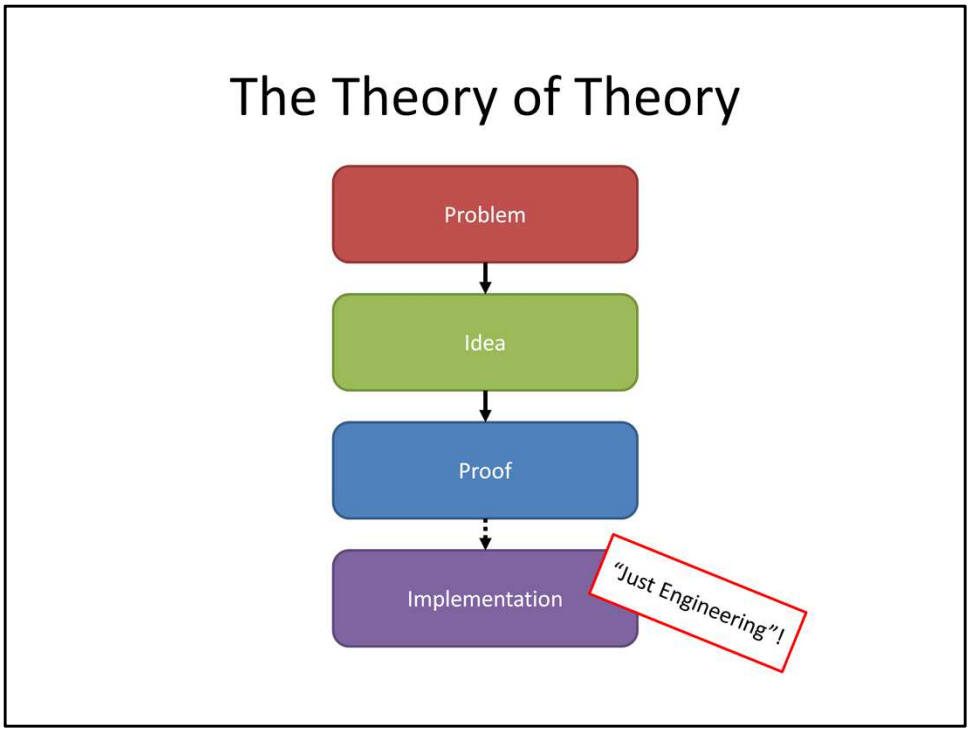
As Keith mentioned, this course is designed assuming that you have taken CS106B or X beforehand. What this means for this section is that we're going to assume that you're comfortable writing basic programs in C++ and Java, and have at least some idea of how to use their standard libraries. This includes knowing how to use templates in C++ or generics in Java. It's ok if you don't remember how to create classes that allow for them; for the most part, we'll only need to be end users of templatized library classes. Also, since section is being held in this computer lab, where Linux is what is installed on all the machines, we're going to assume that you have or can pick up some basic knowledge of the Linux command line.

Why Code Algorithms?

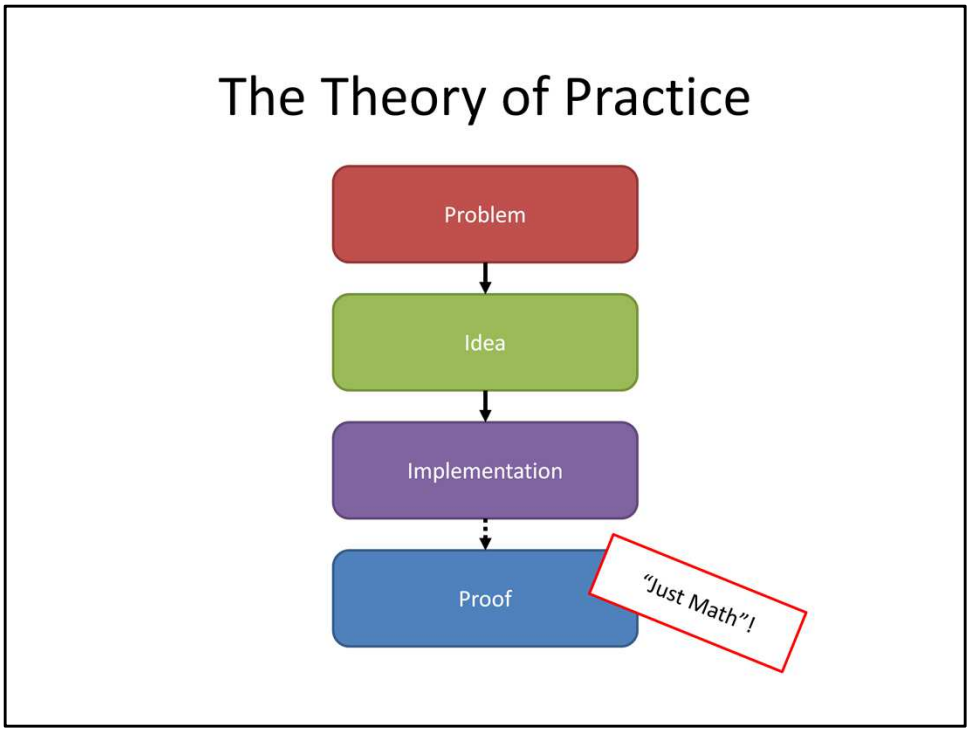
"Beware of bugs in the above code; I have only proved it correct, not tried it."

--Don Knuth

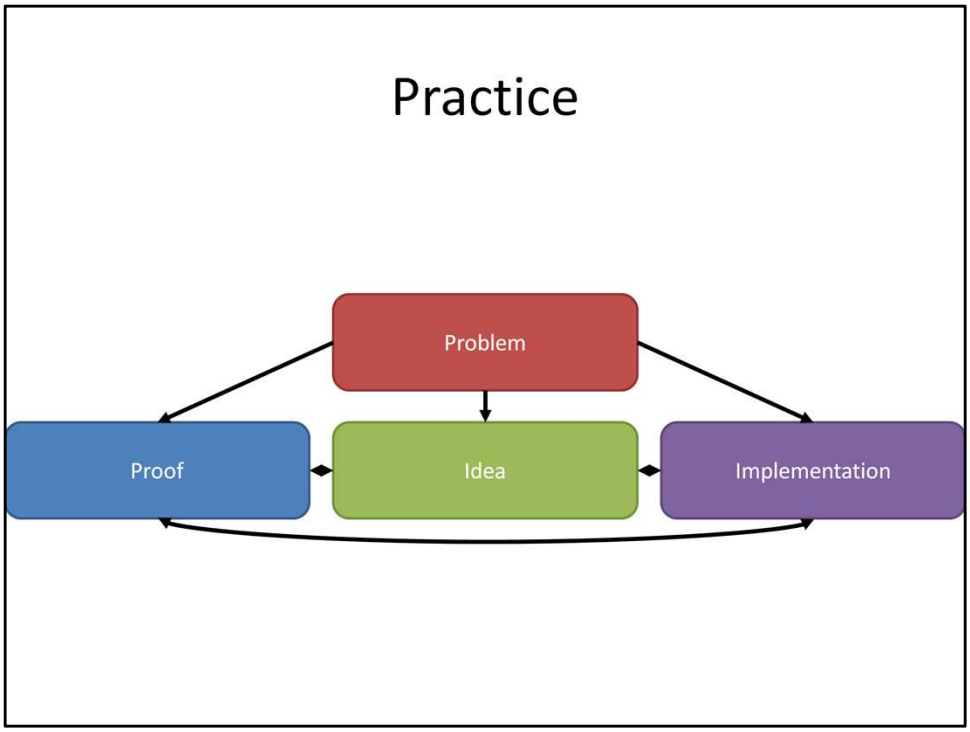
Now, as you know, CS161 is a theory course, so before we jump into this section, we might ask, "Why, in a theory class, do we care about coding up the algorithms we learn? Isn't that part just mechanical?" Well, that's a fair question. After all, as this quote by the famed Donald Knuth would suggest, it IS often the case that we'll stop after we've successfully discovered our algorithm.



See, there's this ideal approach to problem solving that computer science theory courses would have you believe. You're given a problem, which you think about for a while until you come up with an idea to solve it. You then formalize your idea and prove its correctness, and then at that point you have an algorithm that you know would work if you ever sat down and coded it up. Everything after that is "just engineering".



But if you were to ask around a bit more, you would find plenty of people who would tell you that when you have an idea, you should code it up and see how it works. After all, if it works well, you can use it as is and worry about why it works later; and if it doesn't work, then your proof wouldn't have gone anywhere anyway. Once you have code that backs up your idea, proving it works is "just math". Now, this shouldn't surprise us that there are these two trains of thought; after all, we're computer scientists, and we can think of computer science as living halfway between math and engineering. The thing is, there's something wrong with both of these pictures. Note that in both cases, we have this magic step where we come up with an idea for how to solve the problem. How does this step even work? Is it inspiration? I think Keith mentioned "beams of light" coming down from above, granting insights to algorithmic problems. If one of you guys found a beam of light that gives you answers to the problem set that came out yesterday, let me know, because I really want to see that in action.

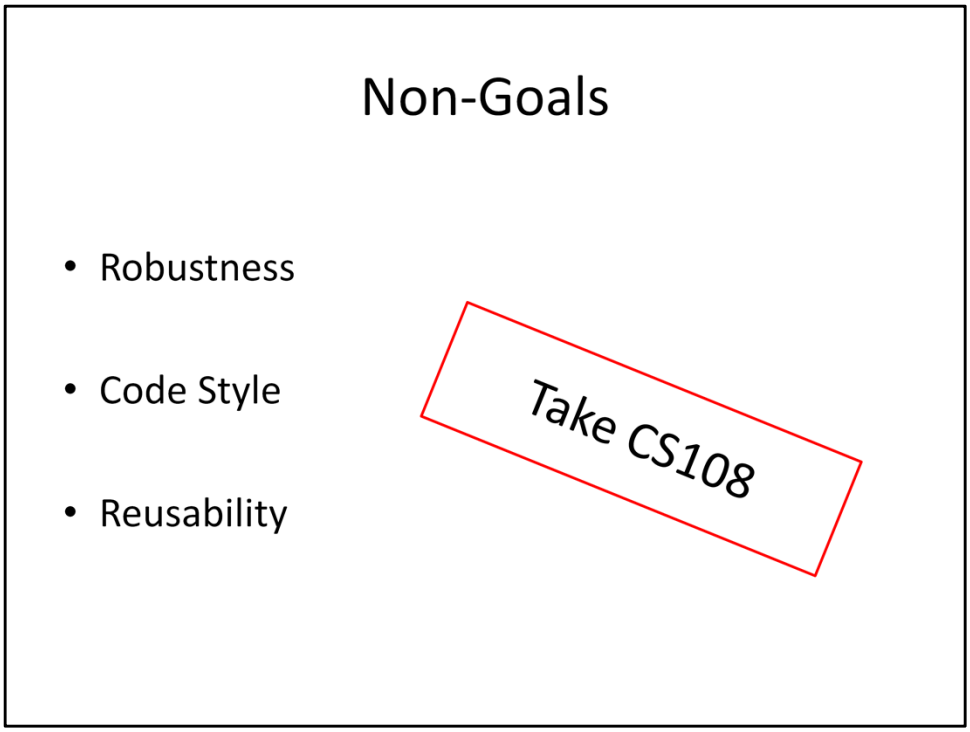


Seriously, though, these ideas don't come from nowhere; in fact, when we're trying to solve a problem, often we'll have to attack at it from all sides, hoping that some direction will allow us to make some progress. Sometimes we might write some experimental code to see if we can discover some patterns in the problem that we can exploit. Other times we'll research similar problems and read through their proofs to see if anything there applies to our problem. All of these parts feed back into each other, so even if all we care about is finding a provably correct solution, we can still benefit from coding up our ideas.

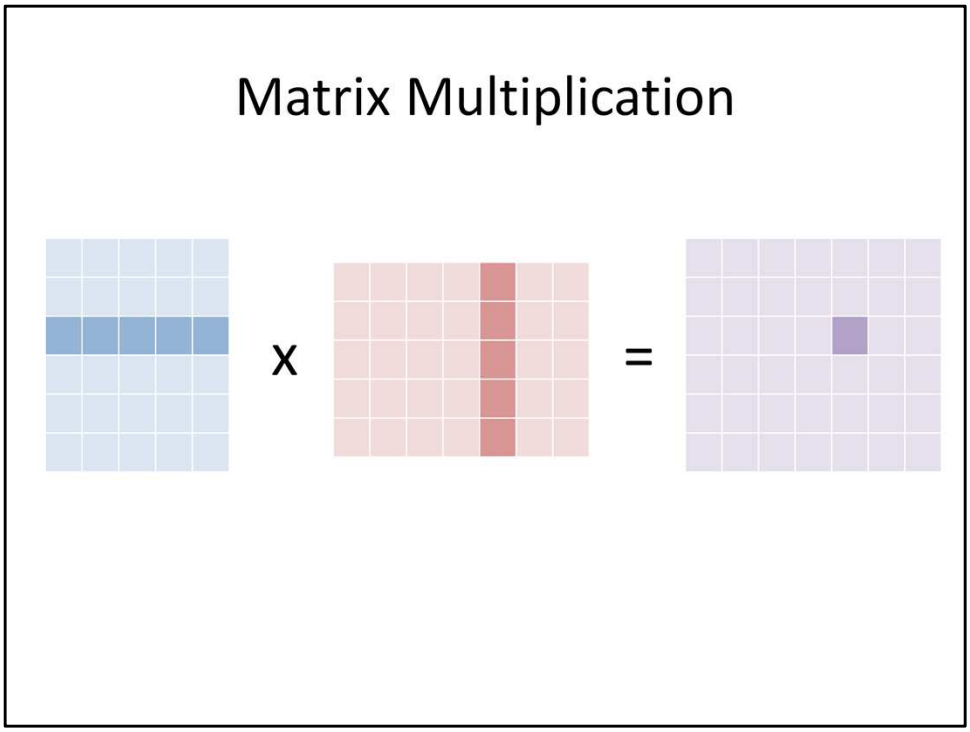
Goals

- Turn algorithm descriptions into real code
- Efficiently write code
- Write efficient code
- Learn theory from code

So here's what I'm hoping you'll get out of this section. First, we're going to spend a good chunk of time turning the descriptions of algorithms that we come across in class into actual runnable code. Often we'll discover that there are some nasty implementation details that we shoved under the rug in the process of our high-level description. We'll also take a look at some algorithms not covered in class. Next, we'll cover some tricks for being able to write what we call "scratch code" quickly and easily. In this section, we're going to focus on using code as a tool for understanding the algorithm, so the emphasis will be on how to do so with minimal effort. We're also going to spend some time covering practices that will make our code more efficient, even if it's just by a constant factor that would get swallowed up in big-O analysis. And finally, we're going to use some coding experiments to motivate refining our theoretical understanding of a problem to better fit what we observe in practice.



Before we get going, let's just briefly mention what we WON'T be covering in this section. We are NOT going to be writing industrial strength code. That means, for example, that we will assume that our input is always well-formed, and we'll handle input in a way that's convenient for us, even if it's not scalable. We're also not going to place any focus on coding style, nor are we going to spend time designing our code in such a way that we'll be able to plug it into other projects easily. These are all valuable skills, and if you want to pick these skills up, there's a great class here called CS108 that will teach them to you.



All right, now let's go start on the first algorithm we're going to code up. Since we also want to get you folks used to the setup, we're going to choose a pretty basic algorithm, and that is matrix multiplication. Now, I suspect it might have been a while since some of you last worked with matrices, so let's go over how matrix multiplication is defined. Remember that a matrix is a two-dimensional table of numbers, and if we say that a matrix is m by n, we mean that it has m rows and n columns. In order to multiply two matrices, the number of columns in the first matrix has to equal the number of rows in the second matrix. So let's say we have an m by n matrix A, and we want to multiply it with an n by p matrix B. We'll get an m by p matrix C. Where does each entry come from? Well, the entry in the ith row and jth column of matrix C is equal to the dot product of the ith row of A and the jth column of B. What do we mean by a dot product? We line up the row and column, multiply the pairs of entries together, and then add them all up. Let's run through an example on the whiteboard, to make sure we're all on the same page.

Matrix Multiplication

- Input
  - An m-by-n matrix A
  - An n-by-p matrix B
  - m, n, p all between 1 and 1000
  - all entries between -1000 and 1000
- Input Format
  - Line 1: 3 integers m, n, p
  - Lines 2 to m+1: n integers
  - Lines m+2 to m+n+2: p integers
- Sample

```
1 2 3
3 4
-1 0 1
0 1 2
```

- Output
  - The m-by-p matrix C = A x B
- Output Format
  - Lines 1 to m: p integers
- Sample

```
-3 4 11
```

OK, now let's start coding this up. For every problem we code up in section, I'm going to provide a slide that looks like this, specifying what your program will take as input, and what it is supposed to produce as output. Notice that we've placed some additional bounds on the problem; for example, the matrices that your program has to handle will never be more than 1000 wide or tall. This is to avoid various gotchas that we don't want to focus on in section, such as numeric overflow or running out of memory. This slide, along with the next few slides, are reproduced for your convenience in the handout.

Input Reference

```
#include <iostream>
#include <string>
using namespace std;
…
int i; double d; string str, line;
cin >> i >> d >> str;
getline(cin, line);

import java.util.*;
…
Scanner s = new Scanner(System.in);
int i = s.nextInt();
double d = s.nextDouble();
String str = s.next();
String line = s.nextLine();
```

All of our programs will read in from standard input, also known as the keyboard. This slide has some basic examples of how you can read from standard in; the top half is how you do so in C++, and the bottom half is how you do so in Java. Some of you might notice that this is rather inefficient compared to other tools available in these languages, but these tools are easy to code with, so we'll use them for this class.

Output Reference

```
#include <iostream>
#include <string>
#include <cstdio>
using namespace std;
…
cout << str << " " << i << endl;
printf("%.2f\n", d);

import java.text.*;
…
System.out.println(str + " " + i);
DecimalFormat fmt = new DecimalFormat("0.00");
System.out.println(fmt.format(d));
```

As for output, we'll always be writing to standard output, also known as the console. Again, the top half is C++, and the bottom half is Java. Note that I've included an example for how to print out a floating point number to two decimal places; this will come in handy for making our output consistent in spite of roundoff errors.

Compilation/Execution/Verification

```
cp -r /usr/class/cs161/public/section/week1 .
g++ -O2 mmult.cpp
./a.out < week1/mmult.in > mmult.out
diff -w mmult.out week1/mmult.ans
```

```
cp -r /usr/class/cs161/public/section/week1 .
javac MMult.java
java MMult < week1/mmult.in > mmult.out
diff -w mmult.out week1/mmult.ans
```

Now, something you might not have experienced before if you've taken the systems core classes here is that we are NOT giving you any starter code today. You'll be writing your code completely from scratch and putting it in a single file. Remember that one of the goals of this section is to enable you to quickly put together some code to test your ideas. Here are the commands you'll need to compile, run, and check your program once you're done writing it. Again, the top half is C++, and the bottom half is Java. Let me take a bit of time to explain these lines. The first line copies all of the data files we've provided for you to your current directory. The second line compiles your code, assuming that you've named your source file to match this slide. The third line executes your program. Notice that we've used input and output redirection for our convenience; using these carets allows us to run our program as though the contents of the mmult.in file were typed in from the keyboard, and send everything we would have written to the console to the file mmult.out instead. The last line runs a whitespace-insensitive comparison of our output to the reference output. If after running the last line you see nothing printed out to the console, the two files match. If, on the other hand, you see stuff printed out, the stuff that's printed out explains the difference between the two files.

Scratch Coding Tips

- Don't worry about memory management
  - Don't bother with delete
  - Avoid using pointers
  - Preallocate arrays (we'll always give you bounds)
- Use global variables when convenient
- Organization and readability still matter!

One last slide before I let you guys go code. Remember that for our purposes, we don't need to worry about memory management. That means you shouldn't worry about deleting the things you allocate. It also means that you C++ coders should avoid making pointer soup, because that's never fun to debug. In fact, if you want to get around some of the annoyances of memory management, just preallocate your arrays. We've given you size bounds on the problem, so you could just allocate arrays that are large enough to accommodate 1000 by 1000 matrices. Also, if you find it convenient, go ahead and use global variables. It can be a pain to pass parameters like the sizes of the matrices through all of your functions, so it's fine to make those values global instead. That said, remember that you're going to need to debug your code, so decomposing your code into reasonable functions and choosing descriptive variable names are still useful. Scratch code is NOT obfuscated code. All right, I'm going to give you guys some time to write your code now. Remember that you can run your code without redirection so that you can type in small inputs to test your program. I'll give you guys until about 2:50 to write up your code. If you run into any problems while debugging, I'll be right here.

Runtime Analysis

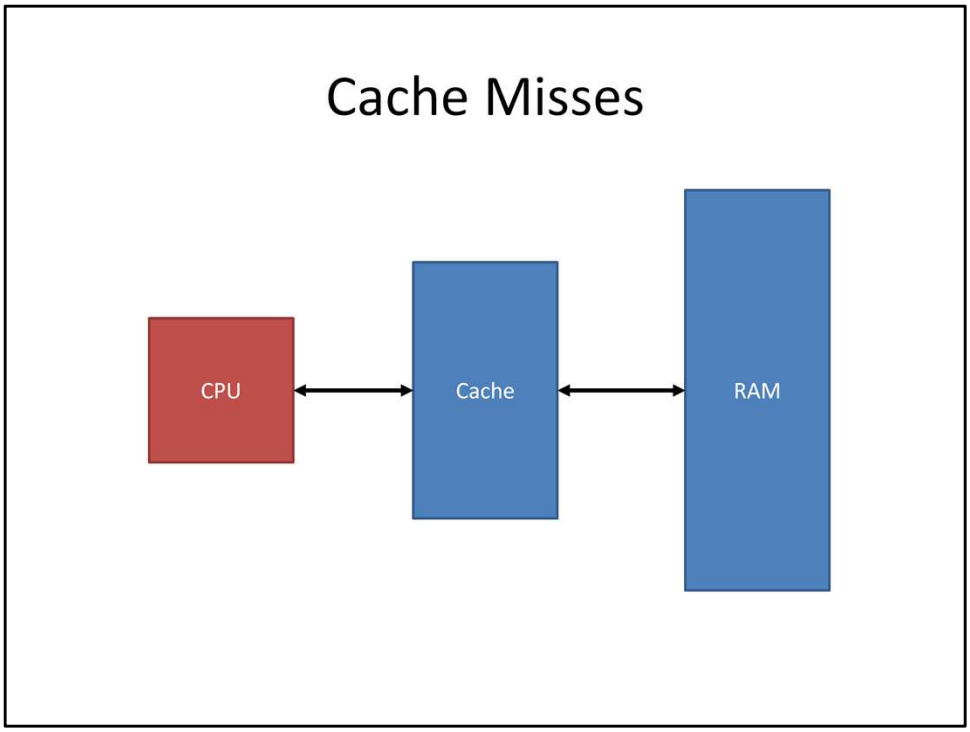
- C has $m \times p$ entries
- Each entry takes n multiplications and $n - 1$ additions
- $\Theta(mnp)$ operations
- Focus on the worst case:
  - Let $N = \max(\{m, n, p\})$
  - $O(N^3)$ time

All right, before we break for today, I'd like to tie what we've worked on in with what we've been covering in class. First, let's analyze the runtime of our algorithm. Our matrix C has m times p entries, and computing each entry takes n multiplications and n – 1 additions. This gives us on the order of m times n times p operations. This is a little unwieldy, so let's just focus on the worst case. Let's let capital N denote the largest dimension of our input matrices. Then our runtime is cubic in N.

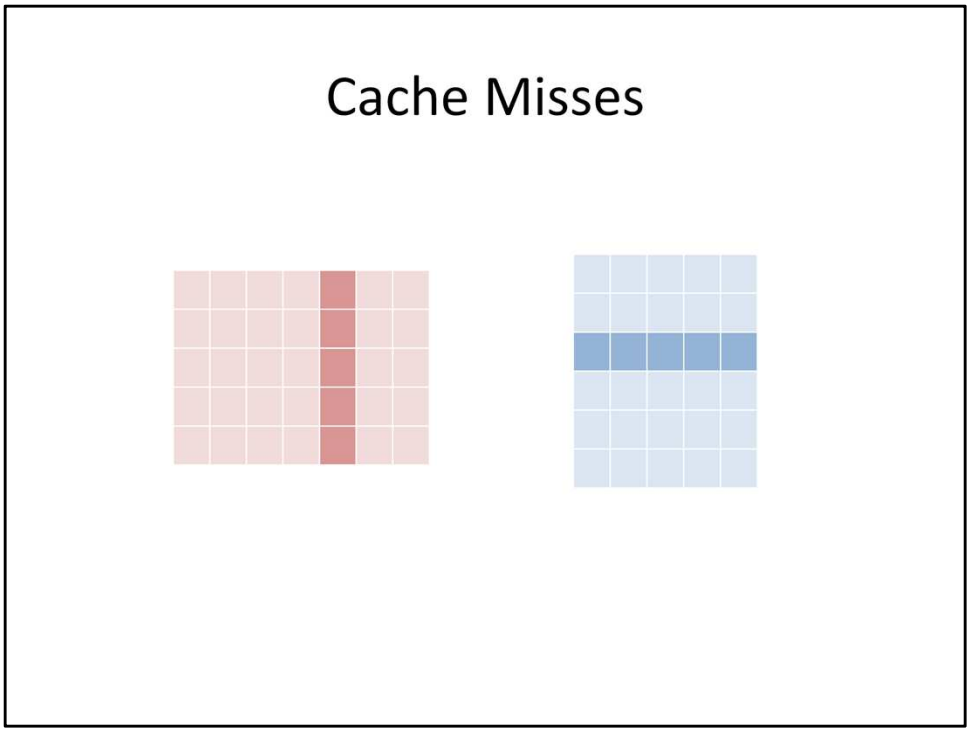
Experiment

What happens if you store B^T instead of B?

However, there's an experiment that I'd like to try. What would happen if we stored the transpose of the second matrix, instead of storing it as is? That is, what would happen if we swapped the rows and columns of the second matrix? Notice that our algorithm doesn't really change; all we're doing is swapping a couple indices. So the runtime analysis we just did shouldn't change at all. But take a look at the difference it makes.



What happened here? Well, let's take a detour into systems land. Remember that our code is executed on the processor, but the data that it operates on is stored in memory. It turns out that it's pretty expensive to move data all the way from memory to our processor; far more expensive, in fact, than performing additions and multiplications on the data once it's there. So what machines do is they keep smaller amounts of memory closer to the processor, and they load data from memory in batches into the cache. This means if you access a bunch of elements sequentially in an array, the first access is slow because it needs to get fetched from memory, but the next several accesses will be much faster because they were in the same block as the first access, so they're already in cache. Now, since our cache is much smaller than all of memory, if we try to access data from all different parts of memory, eventually our cache will be filled, and we'll need to overwrite or evict something we brought in earlier to make room for our next access. That means when we go back to that piece of data that was evicted, it'll be missing from cache, so we'll have to bring it in from memory again. Each time we have to fetch data from memory, we call a cache miss.

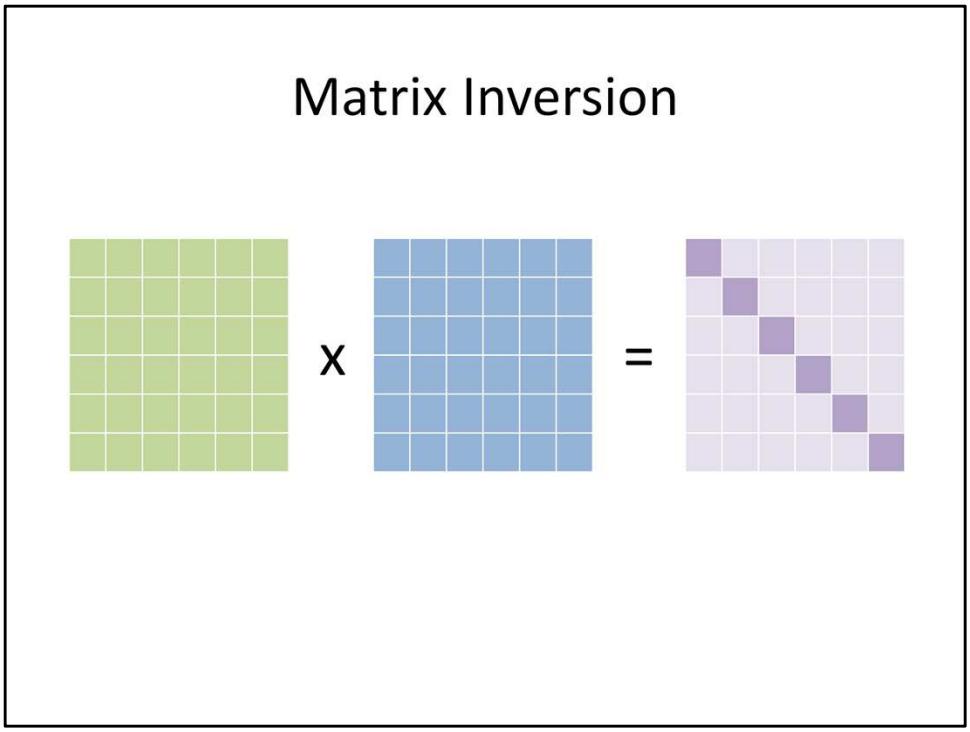


OK, so let's see how this observation helps explain our differences in runtime. In our first implementation, every time we compute an entry of matrix C, we need to loop over the entries of a column of matrix B. Let's imagine for a moment that our matrices are sized such that each row is as big as one cache block. Then each entry lives in a different cache block, so if we can't store the whole matrix in cache, we'll end up incurring a cache miss for every single entry of B. In our second implementation, we loop over the entries of a row of matrix B transpose. All of the entries live in the same cache block, so we'll incur at most one miss for the entire row.

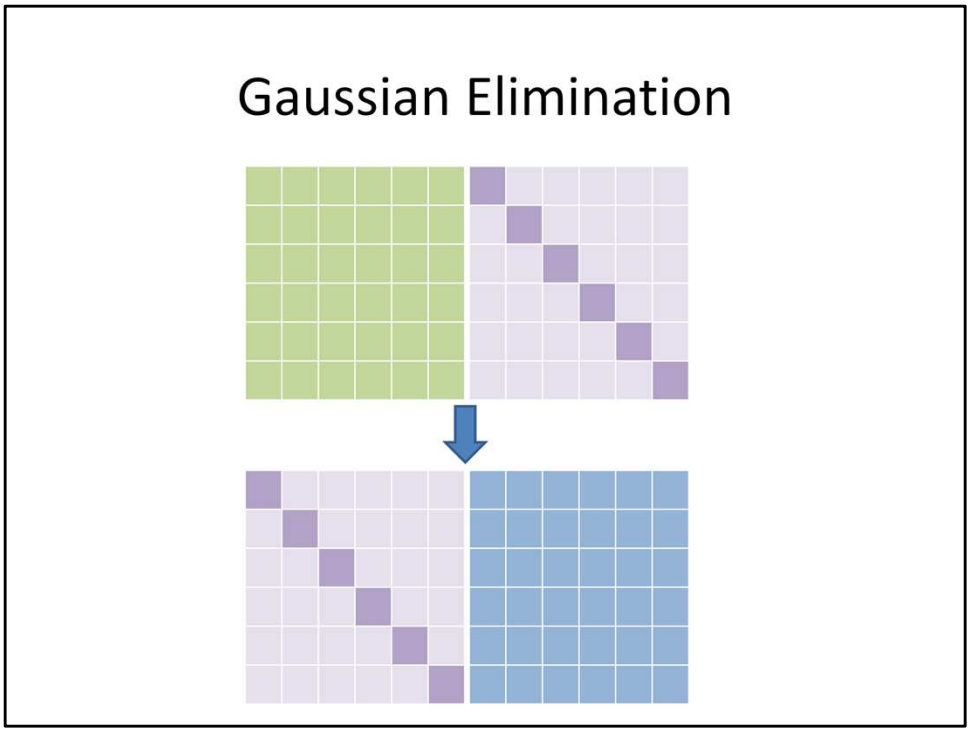
From Practice to Theory

- Assume block size is $\Theta(N)$ ints
- Assume cache size $\leq \alpha N$ ints for some $\alpha < 1$
- Storing B: $O(N^3)$ ops, $O(N^3)$ misses
- Storing B^T: $O(N^3)$ ops, $O(N^2)$ misses

This is a nice observation, but can we turn this intuition for why our second implementation runs faster than our first into something a bit more formal? Well, it turns out we can. Remember that big-O notation is defined in terms of functions; it just so happens that most of the time we're talking about functions that represent the overall runtime of an algorithm. What we can do is we can split our runtime into two parts: the time spent performing arithmetic operations, and the time spent loading things from memory due to cache misses. For simplicity, we're gonna make a couple assumptions about our cache, to make the illustration clearer. It turns out that in this setting, while the number of arithmetic operations remains the same for both approaches, our second approach saves an entire factor of N when it comes to cache misses. When it comes to recombining these expressions to compute our runtime, we still get order N cubed, but since one cache miss takes up a lot more time than one arithmetic operation, we see that the constant factors that the big-O notation hides end up being very different. So here we've used an experiment in code to encourage us to go back and take a closer look at the runtime analysis we did for our algorithm, factoring in an aspect of computation we wouldn't necessarily have considered while just working in theory land. All right, that's all we're going to officially cover for today. If you have an hour to kill and want some more practice coding up algorithms like this, stick around and we'll cover matrix inversion.



For additional practice: The inverse of a square matrix A is defined to be the (square) matrix A^-1 such that when multiplied with A yields the identity matrix, or the matrix with 1s on the diagonal and 0s everywhere else. This isn't a linear algebra course, so we're just going to describe the algorithm for computing the inverse of a matrix, and leave it up to you to go learn why it works.



The algorithm we're going to use is called Gaussian elimination. Given an n by n matrix A, we're going to concatenate an n by n identity matrix to the right of A to get an n by 2n augmented matrix M. We then perform row operations like the ones used to solve systems of linear equations in order to transform M such that its left half is now an identity matrix. What remains on the right half is our inverse A^-1. The row operations we may use are the following:

1) We may scale a row by multiplying it by any nonzero constant we wish.
2) We may swap 2 rows.
3) We may add (a multiple of) any row to any other row.

Note that using these three rules, we can systematically convert our matrix into the desired form by starting from the far left, fixing the entire column, and then moving on to the next column. If we ever end up creating a column on the left half that is all zeros, then we can conclude that the matrix A is not actually invertible.

Note that each row operation can be used to fix a single entry in the matrix, provided we proceed from left to right. Each row operation takes order n work to actually perform, so fixing all n^2 entries in the matrix takes order n^3 time.

Matrix Inversion

- Input
  - An n-by-n matrix A
  - n between 1 and 1000
  - all entries doubles between -1000 and 1000
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+1: n doubles
- Sample

```
2
1.0000 2.0000
3.0000 4.0000
```

- Output
  - The n-by-n matrix A^{-1}
- Output Format
  - Lines 1 to n: n doubles, accurate to 4 decimal places
- Sample

```
-2.0000 1.0000
1.5000 -0.5000
```

Here are the specifications for the problem. Feel free to do some additional reading on Gaussian elimination if you need a clearer picture of how this works.