somethingsomething

premature optimization

somethingsomething

Fred Wenzel, Mozilla

Math, meet code

I was solving some **Project Euler** problems recently

Project Euler is a collection of math-related problems that you need to write code to solve.

Turns out, this taught me something about **optimization**.

So consider the following problem:

projecteuler.net

Problem 1

Work out the **first ten digits** of the **sum** of the following **one-hundred 50-digit numbers**.

```
37107287533902102798797998220837590246510135740250
46376937677490009712648124896970078050417018260538
74324986199524741059474233309513058123726617309629
91942213363574161572522430563301811072406154908250

(...)

82616570773948327592232845941706525094512325230608
22918802058777319719839450180888072429661980811197
77158542502016545090413245809786882778948721859617
72107838435069186155435662884062257473692284509516
20849603980134001723930671666823555245252804609722
53503534226472524250874054075591789781264330331690
```

Simple Python solution:

Project Euler: Problem 13. http://projecteuler.net/problem=13

```
numbers = open('numbers.txt').read()
print str(sum(int(n) for n in numbers.splitlines()))[:10]
```

Anything wrong with this?

But what if...

What if I need this again to sum up 1000 numbers?

Or a bunch of 1000-digit numbers?

Or one thousand 1000-digit numbers?

No problem! We can optimize this!

How about we do it like we'd add manually?

Start with the rightmost digit, adding those up, then "carry over" to the next one

And once we have accumulated 10 digits we can start dropping the last one(s) as we go, because we only need to end up with 10 digits!

Awesome!!! ... or is it

Why is this overoptimized?

Python, unlike C, can handle long numbers just fine, usually without overflows.

There's no indication the Python solution was inefficient in the first place.

Educated guess: With twice, or 10 times, as many numbers, the simple Python solution would still work just fine.

Problem 2

A **path** through a triangle like this is defined by starting at the top and moving either left or right to the next number below, until we reach the bottom. The **total** of a path through the triangle is the sum of all the numbers in such a path.

Find the **maximum total** from **top to bottom** of the triangle below:

```
                75
               95 64
              17 47 82
             18 35 87 10
            20 04 82 47 65
           19 01 23 75 03 34
          88 02 77 73 07 63 67
         99 65 04 28 06 16 70 92
        41 41 26 56 83 40 80 70 33
       41 48 72 33 47 32 37 16 94 29
      53 71 44 65 25 43 91 52 97 51 14
     70 11 33 28 77 73 17 78 39 68 17 57
    91 71 52 38 17 14 91 43 58 50 27 29 48
   63 66 04 68 89 53 67 30 73 16 69 87 40 31
  04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

Project Euler: Problem 18. http://projecteuler.net/problem=18

Intuitive solution:

- Generate all paths
- Determine their totals
- Pick the maximum

Anything wrong with this?

What if...

For this triangle, there are **16,384 possible paths**.

If the triangle **doubles** in height, the number of paths **quadruples**.

i.e., for a triangle of height $\mathbf{n}$, there are $\mathbf{2^{n-1}}$ possible paths.

Let's fix this.

From the bottom up, assign to every node its own weight plus the maximum of its two children's weights.

When done, every node will have a weight assigned to it based on its maximum potential of reaching higher numbers below.

The **top node** contains the **maximum potential** of the entire triangle. **Solved!**

(If we want to find the ideal path, we just need to start at the top, always moving on to the maximum child node.)

Why was this a good optimization?

Our intuitive algorithm has **exponential** complexity of **$O(2^n)$**.

For a triangle with 100 lines, we'd need to analyze
633,825,300,114,114,700,748,351,602,688 paths!

The better algorithm is a flavor of the Bellman-Ford Algorithm and has **linear complexity** of **`O(n)`**.

The improved algorithm works great even for triangles hundreds of lines high.

Summary

Simple >> complex

Don't optimize for the heck of it

unless you have evidence that your solution is inefficient and won't survive expectable changes in problem size.

If your solution is already inefficient now, it won't be long before it breaks.

So look for better methods, including (obviously) efficient algorithms, dynamic programming, memoization, caching, deferred execution, ...

In short:

Avoid premature optimization.

But avoiding premature optimization is not an excuse for bad algorithms.

Thanks!