Intro to Methods of Software Engineering Fall 2016

Lecture 9 — November 8, 2016

Patrick Lam *version 1*

Program Efficiency

Theoretical View

Computer science uses big-O notation to analyze time and space requirements of algorithms. (Technically, big-O asymptotically bounds a function from above.)

time	
$O(\log n)$	searching in a sorted array
$O(n)$	finding the largest element in an array
$O(n \log n)$	average-case quicksort
$O(n^3)$	naive matrix multiplication
$O(2^n)$	boolean satisfiability

We are counting execution steps, and I'm leaving aside here the question of what is an execution step. On modern computers, it's not as simple as one might hope, e.g. due to caches.

Exponential-time algorithms. Boolean satisfiability, or SAT, is a good example of a problem which has no known algorithm that is faster than exponential in general.

Input: boolean variables $x, y, z, a, b, c, \ldots$, possibly negated, ($\neg$) and joined with logical-or ($\vee$) or logical-and ($\wedge$).

Question: Does there exist a satisfying assignment to the variables?

Example: instance

$$x \wedge \neg y \wedge (z \vee \neg x),$$

has satisfying assignment $\langle x : T, y : F, z : T \rangle$.

The naive algorithm tries all possible assignments, which takes time $O(2^n)$. We have heuristics that work better for most "interesting" cases but still $O(2^n)$ in the worst case.

Many practical applications can be encoded as SAT problems, for instance railway signalling systems. Prof. Vijay Ganesh does research on SAT solvers.

Curriculum preview: CS 240 & CS 341. CS 240: learn many important algorithms and analyze their time and space complexity. For instance, you'll learn about priority queues, where you can add elements along with weights, and then you can quickly extract the highest-weight element. CS 341: algorithm design techniques and limits for algorithms.

Limit 1 (hard): Undecidability. Sometimes there is no general solution for a problem.

Let's ask: given a program P, can you tell me if it always terminates?

What about this program?

```
int foo(int N) {
  for (int i = 0; i < N; i++) {
    printf("SEXXI\n");
  }
}
```

Yes, it clearly finishes in $O(n)$ steps. That was easy! Is it always this easy?

```
void bar(int N) {
  if (N == 1) {
    return;
  } else if (N % 2 == 0) {
    bar(N/2);
  } else {
    bar(3*N+1);
  }
}
```

This certainly terminates for all N that we can represent in 32 bits. However, it's not known to terminate even for 64-bit integers; the Collatz conjecture says that it does.

The *halting problem* says that there is no algorithm that works for all programs P and answers whether P halts or not. Of course, as we've seen, we can tell in many cases and just say "I don't know" otherwise. [Intuition: if you had an algorithm A which solved the halting problem, you could construct a program P_A which forced A to give the wrong answer; *Godel, Escher, Bach* is supposed to be an accessible read about it, though I've never read it myself.]

Limit 2 (less hard): NP-completeness. I gave you an $O(2^n)$ algorithm for SAT. Is there a better (polynomial-time) one?

We don't know!

Why should we think there might not be one? The notion of NP-completeness is a hint. If there is an efficient solution to any of these problems (among a set of hundreds):

- integer linear programming;
- subset sum (e.g. is there a subset of { -7, -3. -2, 5, 8} which sums to 0?);
- graph colouring.

then there is an efficient solution to all of them.

It is literally a million-dollar problem; the Clay Mathematics Institute will pay $1M to someone who solves it.

For many of these problems, though, it is possible to efficiently compute approximate solutions.

Practical implications. For real systems, algorithmic issues are often unimportant.

- Constant factors matter in practice (but invisible in big-O notation); and,
- Many programs are not operating on large enough input sizes for algorithms to matter.

If your list has a maximum of 7 elements, go ahead and use an exponential algorithm on it. Just be careful about when requirements change and your list now has 200 elements in it.

We do emphasize theoretical efficiency in any CS program; I think it is key to being an educated computer scientist. It's like this:

> You are not required to go through this check-off, but if you leave the Pavilion to sail elsewhere, and you don't know this material, don't embarass us by letting everyone know that you are an MIT sailor.

(Except we won't let you graduate without knowing what undecidability is.)

Real-world performance

Adapted from `bitfunnel.org/strangeloop`, credit Dan Luu.

Usually performance doesn't matter. But sometimes it does. And when it does, a couple of months of up-front design can make your system perform 10× to 100× better than a badly-designed system. It takes some back-of-the-envelope calculations and simple arithmetic, as we'll see.

You can improve a poorly-designed system by profiling to see what parts take a long time, but you may not get order-of-magnitude improvements without a redesign.

Scale. Let's search a corpus of documents, each of which are 5k.

How many? Let's consider each of:

10k 10M 10G

The first case, 10k documents, is easy. We have 50MB of data in total. In 2016, a $50 phone from Amazon has 1GB of RAM, i.e. 20× more RAM than needed. We can use a naive algorithm:

```
for all docs {
  for all terms in doc {
    // does the doc have all requested terms?
  }
}
```

This is going to run fast enough.

Let's now consider 10M documents. We're now up to 50GB of data, which doesn't fit on a laptop, but does fit on a $2000 server with 128GB of RAM. This server has 25GB/s memory bandwidth. The same algorithm will then take

$$\frac{50\text{GB}}{25\text{GB/s}} = 2\text{s per query.}$$

or 1/2 query per second (during which time the server is otherwise unavailable).

Is 1/2 query per second good enough? Yes, if you have 20 developers typing in queries. No, for an Internet service; latency = $$$, so we'd want to do better.

Finally, for 10G documents, we have 50TB. That's hard to put on a single machine today. You'd want to use 10M docs per machine, which would require 1,000 machines. Can also distribute geographically and add redundancy for better performance.