CS605
9 April 2025

10:55 coffee
11:10 resume lectures
11:45 break
13:15 resume lectures
14:00 end lectures
16:00 lab (proofs)
17:00 TM test 2

Computational Complexity Theory

In the module to date we have been concerned with the ultimate limits of computer programs (if we had unlimited time and memory/storage what could we compute). This area can be called computability theory, and deals with what can and cannot be computed when there are no resource constraints.

Computational complexity theory is a refinement of computability theory, in that we consider the running time of computer programs and the memory/storage requirements of computer programs. This area deals with what can and cannot be computed if we have resource constraints.

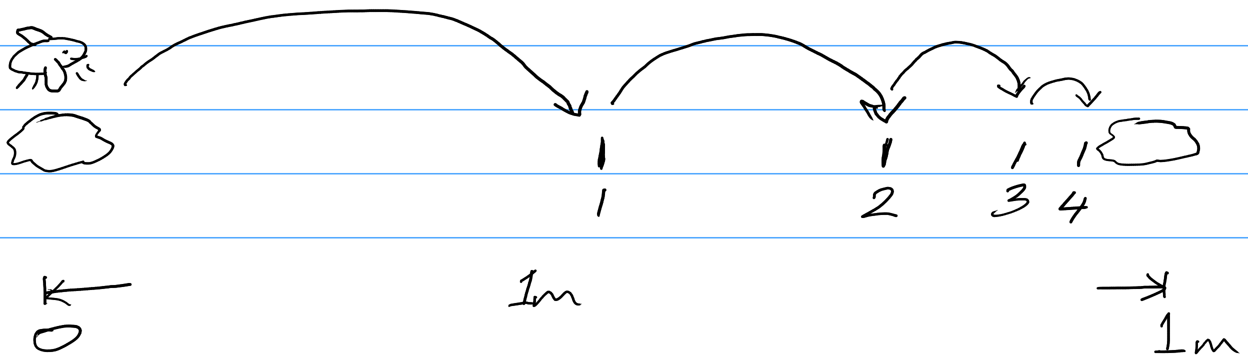
For example, if we have a limit on the running time allowed for a computer program, what problems can and cannot be solved with a computer.

One important use of this theory is to allow a computer programmer to conclude whether they should try to solve a problem perfectly with a computer program, or instead write a computer program to approximate the answer.

With this theory, we must use tools and techniques that are technology-proof (i.e. we cannot conclude that a problem can only be approximated and is too intractable to be solved directly if after 100 years technology improves sufficiently that the problem then becomes tractable and can be solved efficiently).

All of the problems we deal with in complexity theory are decidable.

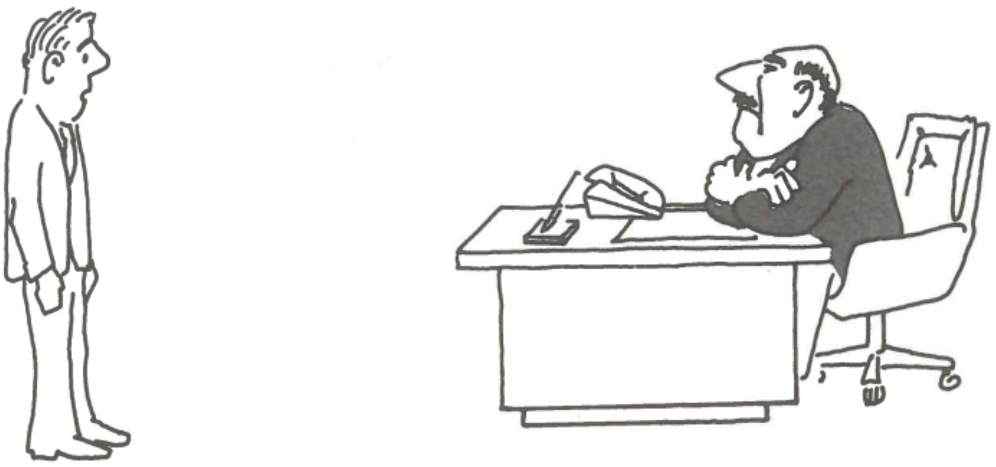
Asymptotic analysis (see separate notes & video) is used to ensure that our results are valid in the future (1. input sizes, 2. technology advances).



Copied from book:
Michael P. Garey & David S. Johnson, Bell Labs
Computers and Intractability -- A Guide to the Theory of NP-Completeness
W. H. Freeman & Co., 1999

Your boss asks you and your software development team to solve a problem related to the company's new "bandersnatch" product line. Your team spends a couple of months but can't solve it...

Rather than this...



"I can't find an efficient algorithm, I guess I'm just too dumb."

To avoid serious damage to your position within the company, it would be much better if you could prove that the bandersnatch problem is *inherently* intractable, that no algorithm could possibly solve it quickly. You then could stride confidently into the boss's office and proclaim:

...could you say this...

"I can't find an efficient algorithm, because no such algorithm is possible!"

almost as good. The theory of NP-completeness provides many straightforward techniques for proving that a given problem is "just as hard" as a large number of other problems that are widely recognized as being difficult and that have been confounding the experts for years. Armed with these techniques, you might be able to prove that the bandersnatch problem is NP-complete and, hence, that it is equivalent to all these other hard problems. Then you could march into your boss's office and announce:

...or say this...?

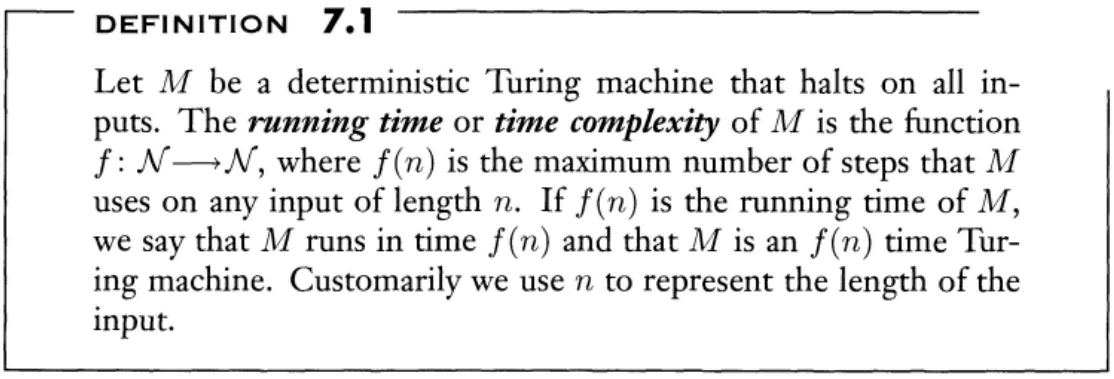
"I can't find an efficient algorithm, but neither can all these famous people."

At the very least, this would inform your boss that it would do no good to fire you and hire another expert on algorithms.

Of course, our own bosses would frown upon our writing this book if its sole purpose was to protect the jobs of algorithm designers. Indeed, discovering that a problem is NP-complete is usually just the beginning of work on that problem. The needs of the bandersnatch department won't disappear overnight simply because their problem is known to be NP-complete. However, the knowledge that it is NP-complete does provide valuable information about what lines of approach have the potential of being most productive. Certainly the search for an efficient, exact algorithm should be accorded low priority. It is now more appropriate to concentrate on other, less ambitious, approaches. For example, you might look for efficient algorithms that solve various special cases of the general problem. You might look for algorithms that, though not guaranteed to run quickly, seem likely to do so most of the time. Or you might even relax the problem somewhat, looking for a fast algorithm that merely finds designs that

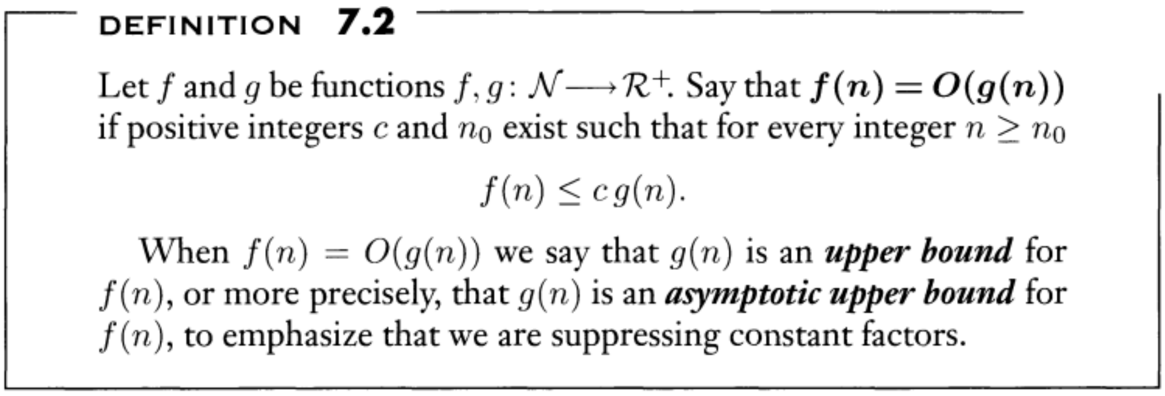
Measurement of resources (time & space)

From Sipser:



DEFINITION 7.1

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

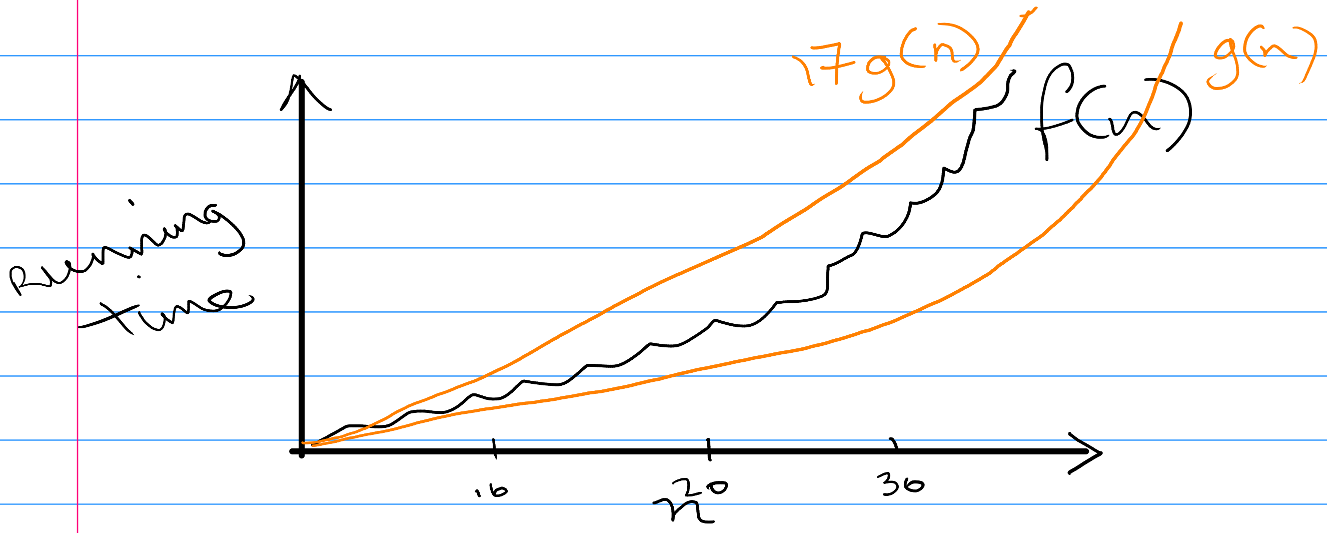


DEFINITION 7.2

Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.



$f(n) = O(g(n))$

$$f_1(n) = \cancel{5}n^3 + \cancel{2n^2} + \cancel{22n} + \cancel{6}$$

Remove lower order terms

Remove multiplicative constants

Remove additive constants

$$f_1(n) = O(n^3)$$
$$f_1(n) = O(n^4)$$

$O \quad \leq$

$o \quad <$

$\Theta \quad =$

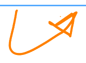
Best case, worst case, and average case complexity analysis

We'll only be concerned with worst case computational complexity analysis going forward, but it is still important to be aware of the concepts of best case and average case analysis.

Let's consider an algorithm to find an element in an unordered list of integers.

Best case

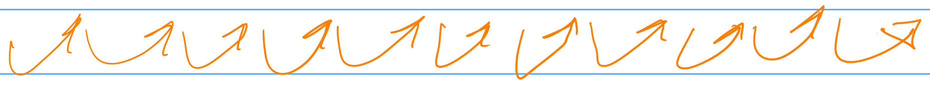
11 7 17 81 2 8 55 38 41 91



$f(n) = 1$

Worst case

11 7 17 81 2 8 55 38 41 91



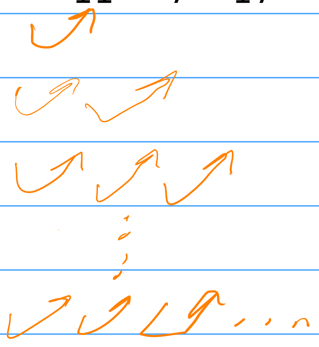
$f(n) = n + 1$

$= O(n)$

Average case

11 7 17 81 2 8 55 38 41 91

$f(n) =$



1 step

2 steps

3 steps

⋮

n+1 steps

gives $f(n) = \left(\frac{n}{2}(n+1) + n + 1\right)/n$

$f(n) = \frac{n}{2} + \frac{1}{2} + 1 + \frac{1}{n}$

$f(n) = O(n)$

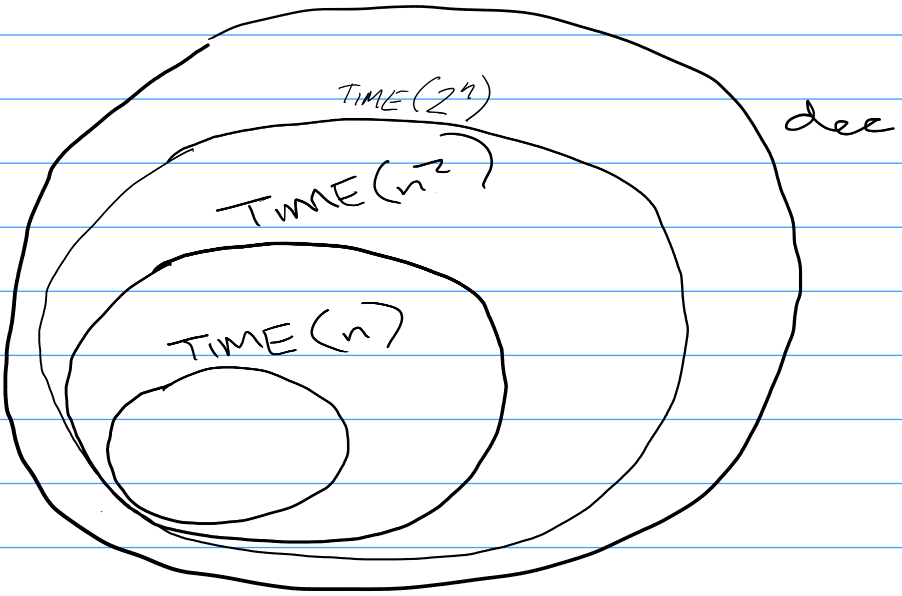
For choosing an algorithm to use in practice, average case analysis is very practical. For example, quicksort is used very often in practice because its average case running time is so low, even though its worst case running time is as bad as insertion sort/selection sort/bubble sort.

For the topics that come next, we'll only be interested in worst case analysis.

DEFINITION 7.7

Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, **TIME($t(n)$)**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

searching an ordered list $\in$ TIME(n)
sorting $\in$ TIME(n^2)
matrix multiplication $\in$ TIME(n^3)
travelling salesperson $\in$ TIME(2^n)



Complexity relationships between models

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Optional: Sipser has simulation.

The Invariance Thesis states that all reasonable (implementable) universal models of computation are polynomially related. This means that moving from one programming language to another, or moving from one computing architecture to another, we will get at most a polynomial speedup in performance, or incur at most a polynomial slow-down in performance. Another consequence of the thesis is that you can't solve a problem in polynomial time using one model of computation that provably requires exponential resources using (the best available version of) another model.

Quantum computing is a promising new technology that might falsify (break) the Invariance Thesis, but so far, it has not demonstrated solving any general-purpose problems in polynomial time that provably require exponential time with a Turing machine.