GTI

Time Complexity

A. Ada, K. Sutner

Carnegie Mellon University

Spring 2018

1. Resource Bounds

- Asymptotics

- Time Classes

A Historical Inversion

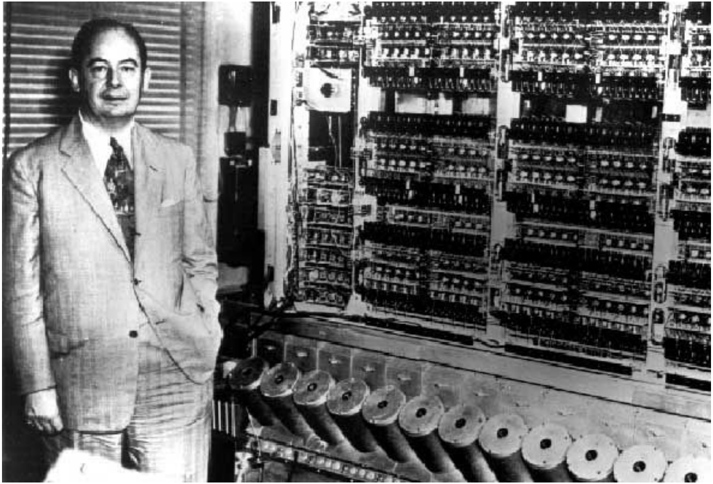
The mathematical theory of computation was developed in the 1930s by Gödel, Herbrand, Turing, Church and Kleene.

The motivation was purely foundational, no one wanted to actually carry out computations in the λ-calculus.

Pleasant surprise: all models define the exact same class of computable functions.

Get rock solid ToC concepts: computable, decidable, semidecidable.

Usable Computers



Feasible Computation

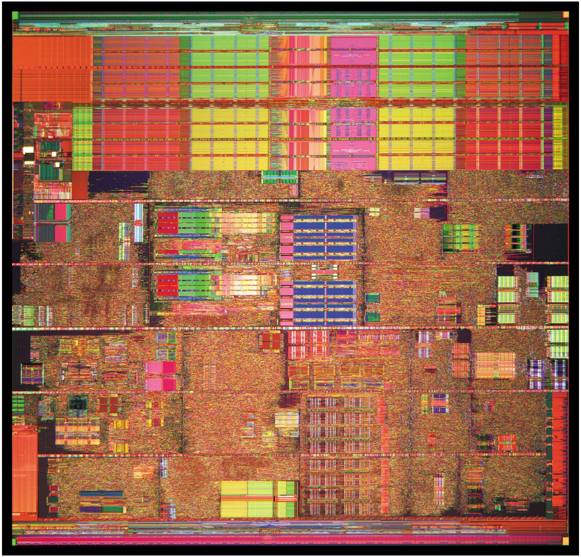
With actual digital computers becoming widely available in the 1950s and 60s, it soon become clear that a mathematically computable function ("recursive function") may not be computable in any practical sense.

So there is a three-level distinction:

- not computable at all,
- computable in principle, and
- computable in practice.

Unsurprisingly, abstract computability is easier to deal with than the concrete kind. Much. The RealWorld$^{\mathrm{TM}}$is a mess.

The Real Target Now



Physical Constraints

So we have to worry about physical and even technological constraints, rather than just logical ones.

So what does it mean that a computation is practically feasible?

There are several parts. It

- must not take too long,
- must not use too much memory, and
- must not consume too much energy.

So we are concerned with time, space and energy.

Time, Space and Energy

We will focus on time and space.

Energy is increasingly important: data centers account for more than 3% of total energy consumption in the US. The IT industry altogether may use close to 10% of all electricity.

Alas, reducing energy consumption is at this point mostly a technology problem, a question of having chips generate less heat.

Amazingly, though, there is also a logical component: to compute in an energy efficient way, one has to compute reversibly: reversible computation does not dissipate energy, at least not in principle.

Complexity Classification

- Algorithms: give upper and lower bounds on performance, ideally matching.
- Problems: give upper and lower bounds for all possible algorithms, ideally matching (intrinsic complexity).

Determining the complexity of a problem (rather than an algorithm) is usually quite hard: upper bounds are easy (just find any algorithm) but lower bounds are very tricky.

This is essentially the search for the best possible algorithm.

Warning

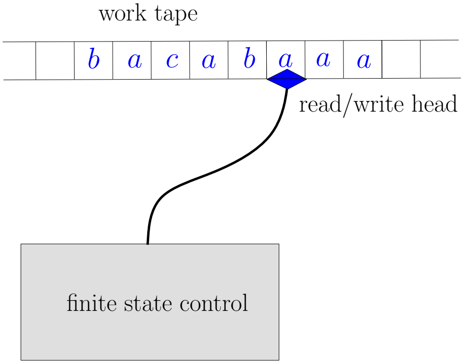
There is a famous theorem by M. Blum that says the following:

> There are decision problems for which any algorithm admits an exponential speed-up.

This may sound impossible, but the precise statement says: “for all but finitely many inputs . . . ”

These decision problems are entirely artificial and do not really concern us. We will encounter lots of problems that do have an optimal algorithm (in some technical sense).

Turing Rules



An algorithm is a Turing machine: a beautifully simple, clean model.

Time Complexity

Note that time complexity is relatively straightforward in every model of computation: there is always a simple notion of “one-step” of a computation.

For Turing machines this is particularly natural:

$$T(x) = \text{length of the computation on input } x$$

Technically, we want to understand the function $T : \Sigma^\star \to \mathbb{N}$.

We are only interested in deciders here, we don’t care about $T(x) = \infty$.

Why Not Physical Time?

It many cases it is interesting to understand how much physical time a computation consumes. Why not measure physical time?

- Requires tedious references to an actual technological model; tons and tons of parameters; results ugly and complicated.
- And, understanding the logical running time provides very good estimates for physical running time, in different situations.

Theory Wins

Reference Model: Turing Machines

Given some Turing machine M and some input $x \in \Sigma^\star$ we measure "running time" as follows:

$$T_M(x) = \text{length of computation of } M \text{ on } x$$

Just to be clear: Turing machines are mathematically simple, but that does not mean that counting steps is trivial. Far from it.

Just take your favorite TM (say, a palindrome recognizer) and try to get a precise step count for all possible inputs.

Worst Case Complexity

Counting steps for individual inputs is often too cumbersome, one usually lumps together all inputs of the same size:

$$T_M(n) = \max\big(T_M(x) \mid x \text{ has size } n \big)$$

Note that this is worst case complexity.

What is the size of an input? Just the number of characters (the length of tape needed to write down x).

You should think of size $|x|$ as the number of bits needed to specify x.

Aside: Average Time

Alternatively we could try to determine

$$T_M^{\text{avg}}(n) = \sum \left(p_x \cdot T_M(x) \mid x \text{ has size } n \right)$$

the average case complexity, where p_x is the probability of instance x.

This is more interesting in many ways, but typically much harder to deal with: it is generally not clear with probability distribution is appropriate, and solving the equations becomes quite hard.

Aside: Amortized Time

Very often a particular operation on a data structure is executed over and over again.

In order to assess the cost for a whole computation, one should try to understand the cumulative cost, not just the single-shot cost.

For example, consider a dynamic array: whenever we run out of space, we double the size of the array.

Every once in a while, a push operation will be very expensive, but usually it will just cost some constant time. A careful analysis shows that the total damage is still constant per operation.

More on this in 15-451, we'll stick to worst case complexity.

Pinning Down Complexity I

Let's say we have an algorithm A (really a Turing machine). We would like to find

upper bounds: Show that A runs in time at most such and such.

lower bounds: Show that A runs in time at least such and such.

In an ideal world, the upper and lower bounds match: we know exactly how many steps the algorithm takes.

Alas, in the RealWorld$^{\mathrm{TM}}$there may be gaps.

Pinning Down Complexity II

Let's say we have a decision problem Π. We would like to find

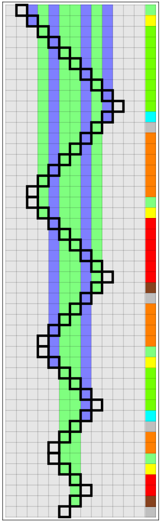
upper bounds: Show that there is some algorithm (read: TM) that can solve Π in such and such time.

lower bounds: Show that every algorithm (read: TM) that solves Π requires such and such time.

Again, we would like bounds to match.

In general, upper bounds are easier than lower bounds.

Palindromes



Has to zigzag, requires quadratic time.

For palindromes one can actually prove a lower bound. And, it matches the upper bound.

Not So Fast

In general, figuring out lower bounds is quite hard. Try

$$L = \{ a^n b^n \mid n \geq 0 \}$$

It might be tempting to try to prove that this is also quadratic: we have to zigzag to match up a's and b's.

Exercise

Find a sub-quadratic TM for this problem. Warmup: figure out how to count a block of n a's.

$$\#aaa\ldots aa\# \quad \rightsquigarrow \quad \#aaa\ldots aa\#100110$$

Getting Real

Anyone familiar with any programming language whatsoever knows that one can check palindromes in linear time: just put the string into an array, and then run two pointers from both ends towards the middle.

Why the gap between quadratic and linear?

Because the program uses a different model of computation: random access machine (RAM).

RAMs are much closer to real computers, but they are much harder to deal with in any serious mathematical analysis.

RAM

We are not going to give a formal definition of a RAM, just use your common sense intuition from programming. Think about counting steps in a C program.

Here are the key points:

- All arithmetic operations (plus, times, comparisons, assignments, $\ldots$) on integers are constant time.
- There are arrays, and we can access elements in constant time.

The insertion sort algorithm from above fits very nicely into this model.

Disaster Strikes

Recall that all models of computation are equal in the sense that they define the exact same computable functions.

All true, but they may disagree about running time.

Computability is a very robust notion, time complexity is much more frail.

Actually, between reasonable models there is usually a mutual polynomial bound, but that's about it. The model matters.

Logarithmic versus Uniform

In a sense, low lever models like Turing machines force you to count every single bit that is manipulated during the computation.

This is clearly justified at some level, but clashes with the practical observation that one can manipulate fixed-size blocks of bits in one step (words in memory).

A slightly more robust way to deal with this is to pretend that numbers are constant as long as they are bounded by some polynomial in the input size

$$x \leq p(n)$$

Fudging Size

Similarly we can simplify the problem of measuring input size:

- in the strict, logarithmic model every bit counts,
- in the relaxed, uniform model numbers have size 1.

For example, when sorting a list of n integers the size in the uniform model is just n.

But in the logarithmic model it is something like kn where k is the number of bits in each integer.

Typical Example

Suppose we want to compute a product of n integers:

$$a = a_1 a_2 \dots a_n$$

- Under the uniform measure, a list of integers of length n has size n. Multiplication of two numbers takes constant time, so we can compute a in time linear in n.

- Under the logarithmic measure, the same list has size essentially the sum of the logarithms of the integers. Suppose each a_i has k bits. Performing a brute-force left-to right multiplication requires some $n^2 k^2$ steps and produces an output of size around nk.

The logarithmic measure is indispensable when dealing with arbitrary precision arithmetic: we cannot pretend that a k-bit number has size 1. This is important for example in cryptographic schemes such as RSA.

- Resource Bounds

2 Asymptotics

- Time Classes

Fudging It

One annoying problem in the analysis of algorithms is that even simple programs often behave in a rather complicated fashion; the running time depends in a rather messy way on the input x.

Keeping track of all the gory details induces insanity, and, on top, is really utterly useless: all that matters are the “higher order terms.”

We introduce some notation that systematically eliminates all the irrelevant details (asymptotic notation, big-oh, big-omega, big-theta).

Insertionsort

```c
void  insertion_sort( int *A, size_t len )
{
   size_t  i, j;
   for( i = 1; i < len; i++ )
   {
      int x = A[i];
      j = i;
      while( j > 0 && x < A[j-1] )
      {
         A[j] = A[j-1];
         j--;
      }
      A[j] = x;
   }
}
```

Analysis

So what is the running time of this program?

On an input of size n, the outer loop (4) executes n times, but the inner loop (8) depends on the actual input.

There is no simple answer, it's a total mess.

But remember, we only care about worst case: for insertion sort, we actually know what that looks like. We wind up with something like

$$T(n) = c_1 + \sum_{i=1}^{n-1} (c_2 i + c_3)$$

Asymptotics

To avoid drowning in umpteen cases and dozens of constants, we ignore lower-order terms and constants. Suppose $f : \mathbb{N} \to \mathbb{N}$ is some arithmetic function. Define

$$O(f) = \{\, g : \mathbb{N} \to \mathbb{N} \mid \exists\, c > 0, n_0 \,\forall\, n \geq n_0 \,(g(n) \leq c \cdot f(n)) \,\}$$

$$\Omega(f) = \{\, g : \mathbb{N} \to \mathbb{N} \mid \exists\, c > 0, n_0 \,\forall\, n \geq n_0 \,(g(n) \geq c \cdot f(n)) \,\}$$

$$\Theta(f) = O(f) \cap \Omega(f)$$

Ordering Things

big-oh	upper bound, no-worse-than
big-omega	lower bound, no-better-than
big-theta	upper and lower bound, exactly-as

Big-Oh

Let's focus on the first class, the others are very similar.

First, we should write $g \in O(f)$, but no one does that; instead one writes

$$g = O(f) \quad \text{or} \quad g(n) = O(f(n))$$

The quadratic polynomial from above now collapses to

$$-0.000772027 + 0.000260301n + 0.000242921n^2 = O(n^2)$$

Make sure you understand why.

Tight Bounds

We could have written

$$-0.000772027 + 0.000260301n + 0.000242921n^2 = O(n^{10})$$

but that is much frowned upon: we want our bounds as tight as possible.

In this particular case,

$$-0.000772027 + 0.000260301n + 0.000242921n^2 = \Theta(n^2)$$

But What About the Constants?

The big-oh notation introduces two fudge factors: there exist constants n_0 and c such that

$$\forall\, n \geq n_0 \left(g(n) \leq c \cdot f(n)\right)$$

Why not just say $g(n) \leq f(n)$?

Because it's actually not so clear what counts for a single step: on step in a Turing machine, one step of a register machine, one clock cycle on some Intel chip, ...

It is best to ignore these details.

Why not just say $\forall\, n \left(g(n) \leq c \cdot f(n)\right)$?

Because often small values of n behave differently, there is no point in keeping track of a few special cases.

But What If They Are Huge?

What if these constants are huge? Let's say c is a trillion digits?

Would this not ruin our analysis completely?

Standard Answer: True, but this simply does not happen: for practical problems it is a matter of experience that the constants are easy to determine and are always very reasonable.

Truth in advertising: this is a white lie. There are some very, very rare cases where we don't know what the constants are; it is safe to ignore these cases for now.

Taxonomy

$O(\log n)$	logarithmic	
$O(n)$	linear	
$O(n \log n)$	log-linear	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	still OK, but . . .
$O(n^k)$	polynomial	
$O(2^n)$	simply exponential	
$O(2^{n^k})$	exponential	

Basic Rules

- $\sqrt{n}$ clobbers $\log n$
- n clobbers $\sqrt{n}$
- n^ℓ clobbers n^k for $\ell > k$
- 2^n clobbers n^k for all k

To prove these things pretend that you have functions $\mathbb{R} \to \mathbb{R}$ and use calculus.

Ignore floors and ceilings, they are the devil's work.

- Resource Bounds
- Asymptotics

3. Time Classes

Time Complexity Classes

We can use time-bounds to organize decision problems into groups. We fix some model of computation once and for all.

Let $f : \mathbb{N} \to \mathbb{N}$ be an arithmetic function.

First, all problems that admit a $O(f)$ solution.

Definition

$$\mathrm{TIME}(f) = \{\, \mathcal{L}(A) \mid A \text{ algo}, T_A(n) = O(f(n)) \,\}$$

Again, to get solid results we should be using Turing machines, but we will usually fudge things (uniform measure, RAMs).

Polynomial Time

Usually we are dealing with a whole family of functions $\mathcal{F}$.

Definition

A (deterministic) time complexity class is a class

$$\mathrm{TIME}(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} \mathrm{TIME}(f).$$

Arguably, the most important case is

$$\mathcal{F} = \text{ all polynomials}$$

Why?

- First off, there are tons of practical algorithms that have polynomial running time.
- Getting from brute-force to polynomial often requires some essential insight.
- Polynomial time decision problems are closed under union, intersection and complement.
- Polynomials are closed under substitution, so any composition of polynomial time algorithms is still polynomial.
- Any real algorithm with polynomial running time already runs in time $O(n^6)$, for some value of 6.
- Reasonably robust under changes in the underlying model.

Some Important Classes

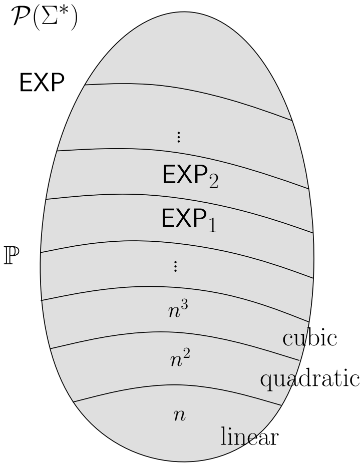
Here are some typical examples for deterministic time complexity classes regarding decision problems (essentially just sets of Yes-instances).

- $\mathbb{P} = \mathrm{TIME}($poly$)$, problems solvable in polynomial time.
- $\mathrm{EXP}_k = \bigcup \mathrm{TIME}(2^{c\,n^k} \mid c > 0)$, kth order exponential time.
- $\mathrm{EXP} = \bigcup \mathrm{EXP}_k$, full exponential time.

Warning: Some misguided authors define EXP as EXP_1.

There are other decidable problems that are much, much worse, but the RealWorld they are not as important.

Our World


$\mathcal{P}(\Sigma^*)$
EXP
⋮
EXP_2
EXP_1
$\mathbb{P}$
⋮
n^3
cubic
n^2
quadratic
n
linear

Example: Shortest Paths

You are given a directed graph $G = \langle V, E \rangle$ with labeled edges $\lambda : E \to \mathbb{N}$.

Think of $\lambda(e)$ as the length (or cost) of edge e.

The length $\lambda(\pi)$ of a path π in G is the sum of all the edges on the path.

The distance from node s to node t is the length of the shortest path:

$$\mathsf{dist}(s,t) = \min\big(\, \lambda(\pi) \mid \pi : s \longrightarrow t \,\big)$$

If there is no path, let $\mathsf{dist}(s,t) = \infty$.

Computing Distance

The standard problem is to compute dist(s, t) for a given source/target pair s and t in G.

If you prefer decision problems write it this way:

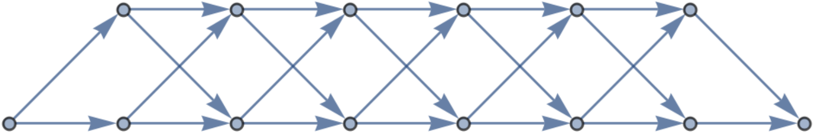
Problem:	**Distance (decision version)**
Instance:	A labeled digraph G, nodes s and t, a bound B.
Question:	Is dist$(s, t) \leq B$?

This decision problem is essentially the same as the problem of actually computing distances.

Upper Bound

A brute-force algorithm is to implement the definition directly: compute all simple paths from s to t, determine their lengths, find the minimum.

Straightforward, but there is a glitch: the number of such paths is exponential in general.



Dijkstra

But one can get away with a polynomial amount of computation by acting greedy: at every step, always take the cheapest possible extension.

```
shortest_path( vertex s ) {
    forall x in V do
        dist[x] = infinity; add x to Q;

    dist[s] = 0;                    // s reachable

    while( Q not empty ) {
        x = delete_min( Q );            // x reachable
        forall  (x,y) in E  do
             if( (x,y) requires attention )
                dist[y] = dist[x] + cost(x,y);
    }
}
```

Dijkstra Contd.

Here Q is a queue (first-in-first-out container), and an edge (x, y) requires attention if

```
    dist[y] > dist[x] + cost(x,y);
```

Since all distance values are associated with an actual path, this means that the current value for y must be wrong.

It is a small miracle that updating only obviously wrong estimates in a systematic manner is enough to get the actual distances in the end.

Definition: Polynomial Time

intuition

formal def

examples

counterexpl

results