UCT

Time Complexity

Klaus Sutner

Carnegie Mellon University

Spring 2022

Getting Real

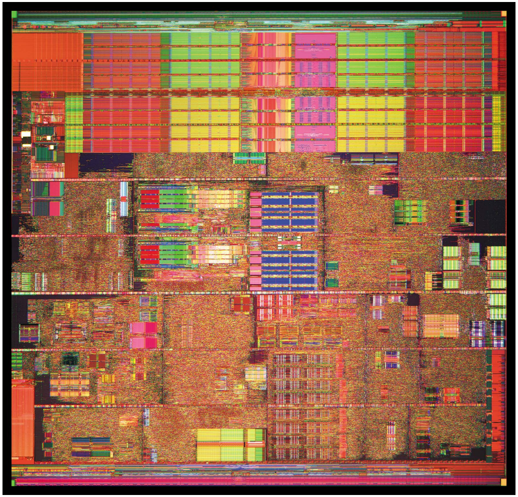
Our classification of decision problems is very useful in **math**, but it sort of misses the boat when it comes to **computer science**.

Conjecture: Digital computers will turn out to be far and away the most important development of the 20th century.

Comparable to the development of language, writing, agriculture.

Of course, that's assuming there still will be digital computers 100 years from now—which is only moderately likely.

A Better Turing Machine



Turing versus Physics

Our definition of a Turing machine is not directly based on physics, though everyone would agree that it is possible to implement a Turing machine as a real, physical machine (even using Legos).

To be sure, this would be an exercise in futility: there are much better ways to exploit the actual laws of physics (and in particular quantum physics) for the purpose of computation. Intel chips are one example.

The opposite direction is much more problematic: since we don't have anything resembling an axiomatization of physics, it is very difficult to reason about an upper bound on physically realizable computations. There are lots of bounds that suggest even some trivially computable functions cannot be realized. Then again, there are black holes and multiverses (maybe).

Physical Constraints

At any rate, in the physical world we have to worry about physical and even technical 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

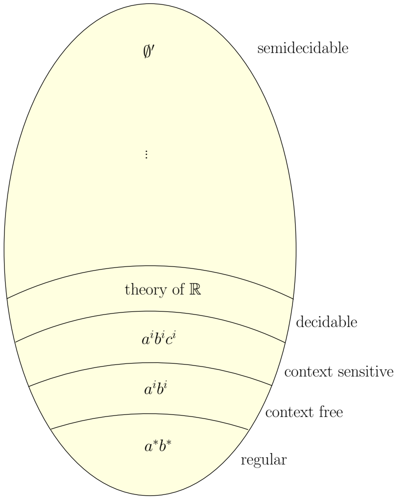
We will focus on time and space.

But note that 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 a an energy efficient way one has to compute reversibly: reversible computation does not dissipate energy, at least not in principle, more later.

Closer to Humans



Formalizing Complexity

We need to come up with ways to measure the complexity of a computation in a clear, precisely defined way. Here is an attempt to define the most general approach to this problem.

Definition (Blum 1967)

A dynamic complexity measure is a partial computable function $\Phi(e, x)$ such that

- $\{e\}(x)\downarrow$ iff $\Phi(e, x)\downarrow$
- the predicate $\Phi(e, x) \simeq y$ is decidable (uniformly in e, x and y)

$\Phi(e, x) \simeq y$ simply means: the computation of program e on input x has complexity y–measured somehow in an effective manner.

One often quietly assumes $\Phi(e, x) = \infty$ when $\{e\}(x)\uparrow$.

Time

Suppose we define $\Phi(e, x) \simeq y$ if the computation of

$\{e\}(x)$ takes y steps.

In other words,

$$\Phi(e, x) \simeq \min\big(\sigma \mid \{e\}_\sigma(x) < \sigma \big)$$

We can check that this Φ is a Blum complexity measure according to the definition.

For decidability, note that we can simply run the computation and count steps as we go along.

Space

How about this definition: $\Phi(e, x) \simeq y$ if the computation of

$\{e\}(x)$ uses y tape cells.

This is not quite right: a divergent computation could still use only a finite amount of tape. But note that we can decide whether this happens, the machine will be caught in a loop (it returns to the same configuration over and over). If we adjust the definition accordingly everything works out fine.

This is one of the elementary differences between time and space, we'll see more of this later.

Other Attempts

Would this definition work: $\Phi(e, x) \simeq y$ if the computation of

$\{e\}(x)$ changes y tape symbols.

How about this one: $\Phi(e, x) \simeq y$ if the computation of

$\{e\}(x)$ moves the tape head y times.

Exercise

Are these complexity measures? If not, fix'em. How useful are these measures?

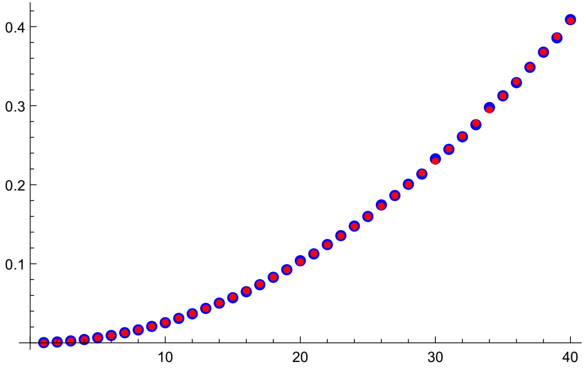
Physical Running Time

Of course, often one would like to estimate the physical running of a computation. This is a bit hard to pin down, since it depends on too many elusive factors such as processor speed, bus speed, memory speed, optimization level, humidity and so forth.

We sidestep these issues by focusing on logical time complexity instead, which comes for free from our model of computation: just count the steps from the input configuration to the output configuration.

Experience shows that, for reasonable computations, there is a very good correlation between the logical number of steps and the actual physical running time: more or less multiply by the appropriate constant.

Example: Insertion Sort



Blue: predicted, red: measured physical running time.

Reference Model: Turing Machines

Turing machines are particularly attractive when it comes to measuring resources: the basic definitions are very, very simple. Alas, as we have seen many times, details can get quite treacherous.

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

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

So time is just the length of the associated sequence of configurations

$$C_x^{\mathsf{init}} \mathrel{\vdash^{t}_{\mathcal{M}}} C_y^{\mathsf{halt}}$$

So this is an example of a dynamic complexity measure.

Worst Case Complexity

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

$$T_{\mathcal{M}}(n) = \max\big(\, T_{\mathcal{M}}(x) \mid x \text{ has size } n \,\big)$$

By size we simple mean the length of the string $x \in \Sigma^\star$.

Note that this is worst case complexity: $T_{\mathcal{M}}(n)$ is determined by the instance of size n that causes the longest computation.

Wait a Moment

Again: What does this “Turing time complexity” have to do with real running time, the kind one discusses in algorithms textbooks?

Obviously, one step in a Turing machine is not the same a CPU cycle. And the size measure does not seem quite right either, see below. This criticism is absolutely correct, but here is out big claim:

> It does not matter. Our model works fine.

This is a bit hard to swallow, but we will see that the slow-down is relatively modest, so it can be ignored in all but the lowest complexity classes. And for those we will change the model (circuits).

The point is that it is **much** more important to have a simple model of computation than having to fight with one that is closer to reality, but completely unmanageable.

Size Matters

Our size measure is the length of the input string. Clearly, this depends on the alphabet, but up to a constant factor it is the same as if we were counting bits. In this scenario, a number n has size $\log n$. This is called logarithmic size complexity and works naturally for Turing machines.

Alas, in many real applications, one can safely assume that all numbers in an instance are of bounded size (say, 64 bits). In this case it is standard in algorithm analysis to use uniform size complexity: a number is assumed to have size 1.

This is completely standard in the analysis of algorithms: a vertex in a graph (an integer) has size 1, but for a big prime we have to count bits.

Typical Example

Suppose we want to compute a product of n integers, n pretty small:

$$a = a_1 a_2 \dots a_n$$

- Under the uniform measure, the input 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 $O(n^2 k^2)$ steps and produces an output of size $O(nk)$.

No Problem

This distinction is totally standard in algorithm analysis.

For example, to say that depth-first-search takes $O(n+e)$ steps automatically assumes that the vertices are described by machine sized integers, otherwise we would pick up another factor $\log n$.

But 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 where the underlying arithmetic becomes quite expensive.

And Turing Machines?

Our Turing machines naturally use the logarithmic model: we write down numbers in binary, say.

Mostly true, but not really: we could use a tape alphabet of size 2^{64} to pretend we can deal with machine sized integers in one step.

Arguably, we really should fix our tape alphabet to be something like

$$\Sigma = \{\sqcup, 0, 1\} \qquad \text{or} \qquad \Sigma = \{0, 1\}$$

but it's very convenient to have larger alphabets lying around.

Language Recognition

As already mentioned, we focus on decision problems since they are a bit easier to handle than function and search problems (though these are arguably more important in reality).

Since Turing machines naturally operate on strings (rather than integers in classical computability theory), one usually describes decision problem in terms of languages. The instances are all strings over some alphabet Σ.

The Yes-instances are a set of words, a language $L \subseteq \Sigma^\star$. We want to recognize these languages.

Language Recognition Problem

Here is the standard decision problem that we have already encountered in several contexts. Fix some language $L \subseteq \Sigma^\star$.

Problem:	**Recognition Problem (for L)**
Instance:	A word $w \in \Sigma^\star$.
Question:	Is w in L?

Note the qualifier "fixed": there is a parametrized version of the problem where L is part of the input. This only makes sense if L can be represented by a finite data structure such as a finite state machine or context free grammar; here we are interested in the other scenario.

Example: Graphs

Suppose we want to check whether a ugraph $G = \langle V, E \rangle$ is planar.

We may safely assume that $V = [n]$, so we can represent G is an $n \times n$ binary matrix, the adjacency matrix.

By flattening out the matrix in row-major order we get a string $A \in \mathbf{2}^{n^2}$.

The language Planar $\subseteq \mathbf{2}^\star$ of all such strings representing planar graphs is our formalization of the problem.

We want to understand the complexity of Planar.

With a real algorithm we know how to do this in linear time, but a Turing machine will be a little slower.

Acceptance Languages

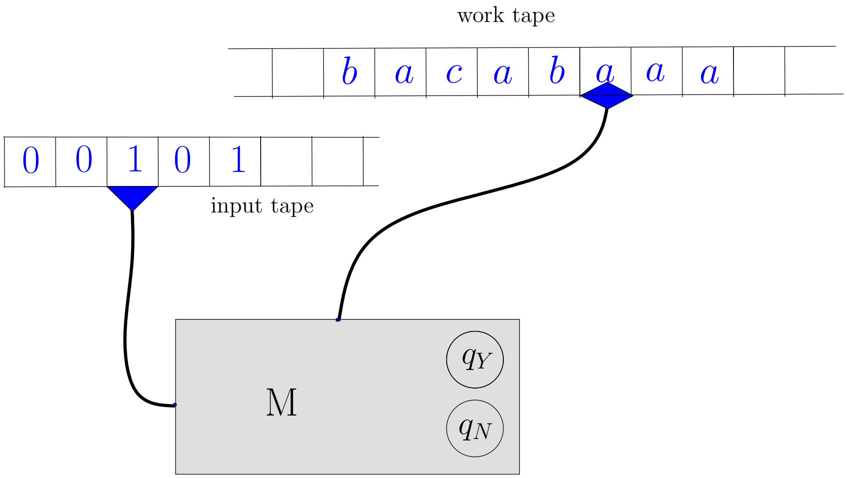
It is best to adjust our Turing machine model slightly to deal with decision problems. Say, a Turing machine $\mathcal{M}$ acceptor is a modified TM that

- halts on all inputs, and
- always halts in one of two special states q_{Y} and q_{N}, indicating "accept" or "reject".

To make life a little easier, we assume that the input is given on a separate read-only tape, and the actual computation is carried out on a work tape.

This is in particular important for space complexity, where we will charge only for the work tape.

Acceptor



Dire Warning

We already know that it is highly undecidable whether a given arbitrary Turing machine is an acceptor: we need $e \in \mathsf{TOT}$, among other things.

As a consequence, we cannot effectively enumerate these machines.

This won't be a big issue, though: in many interesting cases we can force totality, typically by adding a clock.

The Language

We can use an acceptor is to define a language, the collection of all accepted inputs:

$$\mathcal{L}(\mathcal{M}) = \{ \, x \in \Sigma^\star \mid \mathcal{M} \text{ accepts } x \, \}$$

So to solve a recognition problem we need to construct a machine $\mathcal{M}$ that accepts precisely the Yes-instances of the problem. Of course, exactly the decidable languages are recognizable in this sense.

We are interested in the case when the corresponding computation can be carried out in the RealWorld$^{\text{TM}}$.

Computable Functions

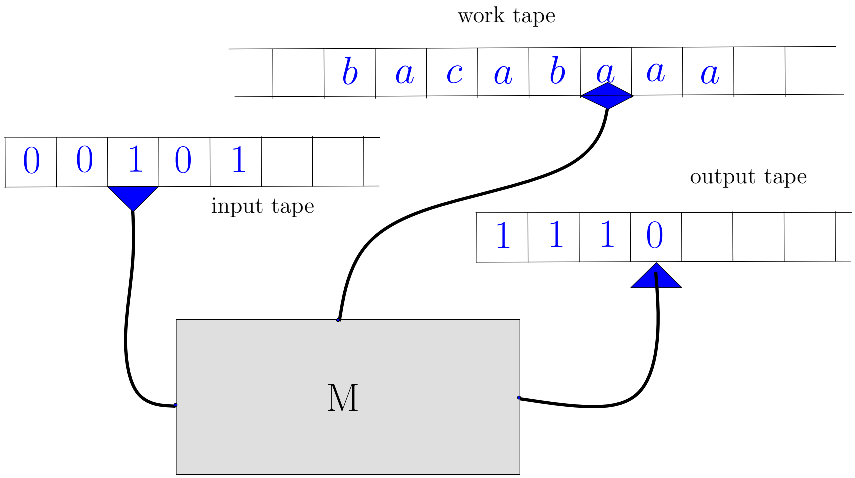
To compute a partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$ we use a slightly different type of Turing machine, a transducer.

A transducer

- has a special halting state q_H, but
- may not halt on all inputs;
- writes the output on a separate write-only tape.

In most concrete case we will have better information and will be able to assert that the machine does indeed halt on all inputs. But, as always, this property is undecidable.

Transducer



Examples

Example

There is a Turing machine $\mathcal{M}$ with acceptance language "all strings over $\{a, b\}$ with an even number of as and and even number of bs" with running time $O(n)$.

Example

There is a one-tape Turing machine $\mathcal{M}$ with acceptance language "all palindromes over $\{a, b\}$" with running time $O(n^2)$.

Example

There is a two-tape Turing machine $\mathcal{M}$ with acceptance language "all palindromes over $\{a, b\}$" with running time $O(n)$.

Time Complexity Classes

We can use time-bounds to organize decision problems into classes.

Definition

Let $f : \mathbb{N} \to \mathbb{N}$ be a function. Define

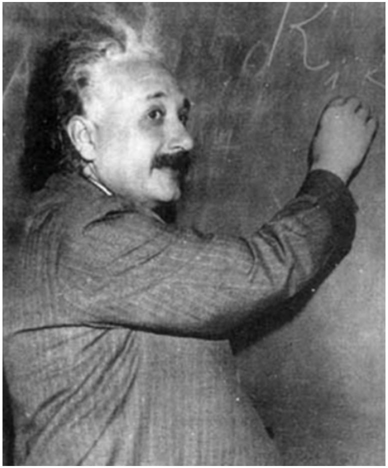
$$\mathrm{TIME}(f) = \{\, \mathcal{L}(\mathcal{M}) \mid \mathcal{M} \text{ a TM}, T_{\mathcal{M}}(n) = O(f(n)) \,\}$$

A (deterministic) time complexity class is a class

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

for some (reasonable) collection of functions $\mathcal{F}$.

Einstein



I believe in intuition and inspiration . . . at times I feel certain I am right while not knowing the reason.

Time Constructibility

There is a technical difficulty here: if the function f is wild, $\mathrm{TIME}(f)$ becomes quite unwieldy.

Definition

A function $t : \mathbb{N} \to \mathbb{N}$ is time constructible if there is a Turing machine that runs in $O(t(n))$ steps and, for any input of size n, writes $t(n)$ on the output tape (say, in binary).

We require that the TM reads its input, so time constructible automatically implies $t(n) \geq n$.

There are slightly different notions of time constructibility in the literature, but they all come down to the same thing.

No Big Deal

Other than the bound $t(n) \geq n$, just about anything goes: all your favorite arithmetic functions n^k, 2^n, $n!$, and so on are time constructible.

In fact, it is quite difficult to come up with functions that are not time constructible. Try something like $t(n) = 2n$ or $t(n) = 2n + 1$, depending on some condition of the right difficulty. Of course, this is not really an arithmetic function, it's logic in disguise.

Exercise

Verify that all the functions above are indeed time constructible.

Clocked Turing Machines

The reason one requires time constructibility is that one often needs to simulate a Turing machine $\mathcal{M}$ and halt the computation after at most $t(n)$ steps: essentially we are adding a clock to the machine and pull the plug on any computation that takes too long.

More precisely, we

- are given input $x \in \Sigma^\star$,
- compute $B = t(|x|)$,
- use a counter to compare the number of steps to B.

The Catch

No problem, right? Even on a Turing machine.

Alas, for this to be useful, the whole simulation must run in time t, and that includes the clocking mechanism.

In particular the computation of $B = t(|x|)$ cannot be hugely expensive, whence the condition of time constructibility.

Lots of Annoying Details

So how about counting a block of as?

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

Alas, on a one-tape machine, this seems to take quadratic time: the head has to zig-zag back and forth between the as and the binary counter.

This causes a problem if we are interested in sub-quadratic running times. Of course, on a two-tape machine this could be handled in linear time.

A Trick

#	a_1	a_2	a_3	a_4	a_5	a_6	$\dots$	a_n	#
			1	0	1		$\dots$		

We can achieve $n \log n$ on a single two-track tape (rather than two-tapes) by keeping the counter bits close to the tape head[†].

Exercise

Figure out the details.

[†]Recall the slide titled **TMs Suck**?

Simulation

We also need to worry about the actual simulation: we have some index e and want a master machine $\mathcal{U}$ to simulate $\mathcal{M}_e(x)$ (maybe up to σ steps).

We are going to keep track of configurations of $\mathcal{M}_e$: current tape inscription, state and head position. And, of course, we have to store the lookup table e. Again it is best to do this with multiple tracks:

a_ℓ	$\ldots$	a_2	a_1	p	b_1	b_2	$\ldots$	b_r
	e_1	e_2	$\ldots$	e_k				

One step in the simulation here would be $O(|e|)$.

Right?

Careful, we are pretending that each symbol of $\mathcal{M}_e$ corresponds to the upper track of a symbol in $\mathcal{U}$—but that machine has a fixed alphabet, and e varies over machines of arbitrary alphabets.

We may as well assume that $\mathcal{U}$ has a two-track tape alphabet $\{␣, 0, 1\}^2$. So in reality we would have to express the a_i, b_i, p and e_i as blocks of bits.

Hence we pick up a factor of $\log|\Sigma(\mathcal{M}_e)|$.

This typically does not affect any application of $\mathcal{U}$, but $|e|$ is just false.

The Key Classes

Here are some typical examples for deterministic time complexity classes that pop up quite naturally.

- $\mathbb{P} = \mathrm{TIME}(\text{poly})$, 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.

We could consider even faster growing functions such as 2^{2^n}, but they are typically not as important in the world of real algorithms.

[†]**Warning:** Some misguided authors define EXP as EXP_1.

Closure

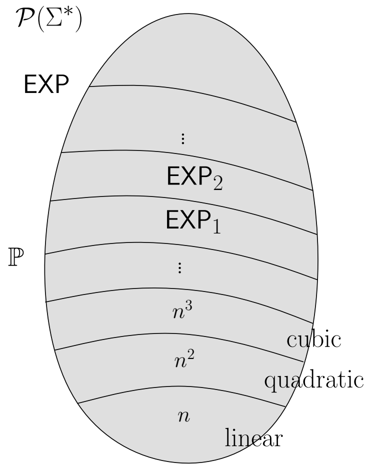
Lemma

The classes $\mathbb{P}$, EXP_k and EXP are all closed under union, intersection and complement.

This is fairly obvious, we can just flip the output bit, or run two computations without breaking the time bound.

Note that we don't need to interleave computations (as for the union of semidecidable sets): everything now halts.

Our World


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

Why $\mathbb{P}$?

Polynomial time seems to be a reasonable first formal answer to the question: what does it mean for a problem to be computable in the RealWorld$^{\mathrm{TM}}$?

One often speaks of feasible or tractable problems.

These notions are intuitively compelling, but we can't hope to prove anything without a precise formal definition—and it seems that $\mathbb{P}$ works quite well in that capacity.

Take this with a grain of salt, we will have more to say about feasibility later on.

Why Big-Oh?

Note that we require

$$T_{\mathcal{M}}(n) = O(f(n))$$

rather than $T_{\mathcal{M}}(n) \leq f(n)$ as one might expect.

This is the same trick used in asymptotic analysis of algorithms to avoid cumbersome special cases for small n. Also, it turns out that in the Turing machine model multiplicative constants are really meaningless:

Lemma (Speed Up)

If $T_{\mathcal{M}}(n) \leq f(n)$ then we can build an equivalent machine $\mathcal{M}'$ such that $T_{\mathcal{M}'}(n) \leq c \cdot f(n)$ for any constant $c > 0$.

Proof

We may assume $c < 1$. Suppose $\mathcal{M}$ has alphabet Σ. Choose a constant k such that $1/k < c$.

The new machine $\mathcal{M}'$ will use alphabet Σ^k: each super-letter consists of a block of k-many ordinary letters.

We modify the transition function accordingly, so that one step of $\mathcal{M}'$ corresponds to k steps by $\mathcal{M}$.

$\square$

Warning: This is a perfect example of a result that has little bearing on physically realizable computation: Σ^k grows exponentially, and is typically not implementable. One should always think about using $\mathbf{2}$ as the tape alphabet.

Hierarchy Theorems

It is intuitively clear that $\mathrm{TIME}(f)$ will be larger than $\mathrm{TIME}(g)$ provided that f is sufficiently much "larger" than g.

Here is a technical version of this. Alas, the proof is quite intricate–for technical reasons, not because of some deep philosophical obstructions.

Theorem (Hartmanis, Stearns 1965)

Let f be time constructible, $g(n) = o(f(n))$.

Then $\mathrm{TIME}(g(n)) \subsetneq \mathrm{TIME}(f(n)^2)$.

Note the square, this is a bit weaker than what we might hope for.

Tip of the Iceberg

One can actually squeeze down the gap quite a bit,

$$\mathrm{TIME}(g(n)) \subsetneq \mathrm{TIME}(f(n)\log f(n))$$

also works.

The proof is essentially the same, except that one needs to be much more careful in setting up the simulating machine.

Try to go through the following argument and count steps very carefully.

Cheap Example

Suppose we want to show that $\mathrm{TIME}(n) \subsetneq \mathrm{TIME}(n^2)$.

In this case, we can show that testing for palindromes is in quadratic time, but not in linear time (this is for one-tape TMs).

Hence we have separated the two complexity classes.

Unfortunately, this does not help with n^2 versus n^3 and so on, it's just a single result. We need much bigger guns.

Halting?

It is tempting to press a modified version of the Halting problem into service. How about

$$K_f = \{\, e \mid \{e\}_{f(|e|)}(e) \,\} \subseteq \mathbf{2}^\star$$

In other words, we limit all computations according to some (time constructible) bound f.

Intuitively, this should produce a set in $\mathrm{TIME}(f)$ of maximum complexity. And it should not be in $\mathrm{TIME}(g)$ since $g = o(f)$, more or less.

Proof

So the central idea is still diagonalization, but we have to make sure that the diagonalization does not push us over the $O(f^2)$ bound.

We assume an effective enumeration $(\mathcal{M}_e)$ of f-bounded Turing machines computing 0/1-valued functions.

Note: Since we are only interested in time-bounded computations here, there is no problem with Halting. There really is an effective enumeration of these machines.

Furthermore, for technical reasons, we require that each machine is repeated infinitely often in the sense that infinitely many machines are always equivalent (so this is the opposite of a repetition-free enumeration wrt the corresponding functions).

Diagonalization

Now define the function

$$\delta(x) = \begin{cases} 1 - \mathcal{M}_x(x) & \text{if } \mathcal{M}_x(x)\downarrow \text{ in time } f, \\ 0 & \text{otherwise.} \end{cases}$$

Otherwise here means: the computation tries to use too much time before producing a result, an easily decidable property.

Thus δ is total by construction and 0/1-valued.

The Machine

Since $f(n)$ is time constructible, δ can easily be computed by a Turing machine $\mathcal{M}$ in time $f^2(n)$, and with a bit more effort we can get down to time $f(n) \log f(n)$.

To this end, the machine $\mathcal{M}$ uses a three-track alphabet of the form Σ^3, which we may safely assume to be $\{\text{␣}, 0, 1\}^3$.

- one track for e, one for the clock, one for the simulation;
- e and the counter can be kept close the to tape head to avoid zigzagging.

Note that time constructibility is essential here: we first compute $f(|x|)$ (quickly), and then compare a step counter to that value.

The Contradiction

This is all easy if we don't worry too much about running time; if we want to squeeze out every little possible speed-up, then things get tricky. We want a high-performance universal machine, not just any old piece of junk.

Now assume δ can be computed by some machine $\mathcal{M}$ in time g.

Since $g = o(f)$, there is an index e for $\mathcal{M}$ such that $g(e) < f(e)$. But then $\mathcal{M}_e$ properly computes δ.

Alas, by construction $\delta(e) \simeq 1 - \mathcal{M}_e(e)$, the usual diagonal contradiction.

$\square$

Great and Unmatched Wisdom

Most proofs in computability and complexity theory are **read-once**.

Read the proof (which is really a proof-sketch at best) once or twice, just to figure out the author's general strategy, and perhaps a few technical tricks.

Then throw it in the trash, and reconstruct the argument in your own words, using your own approach. This is the only way I know to get comfortable with these results. Parroting back someone else's approach over and over is pointless.

Corollary

$$\mathbb{P} \neq \mathrm{EXP} = \bigcup \mathrm{EXP}_k;$$

To see this note that for any polynomial p we have $p(n) = o(2^n)$, so $\mathbb{P} \subseteq \mathrm{EXP}_1$.

But $\mathrm{EXP}_1 \subsetneq \mathrm{EXP}_2 \subseteq \mathrm{EXP}$, done.

OK, this is a cheap shot, we could do better (try to suggest an improvement). But you will soon see that separating classes if often very difficult.

The Gap Theorem

Just to be clear: the hierarchy theorem depends heavily on the relevant functions being well-behaved.

Theorem (Gap Theorem)

There is a computable function f such that $\mathrm{TIME}(f) = \mathrm{TIME}(2^f)$.

f can be constructed by a diagonalization argument and is highly uncivilized. It has no bearing on algorithms in the RealWorld$^{\mathrm{TM}}$.

Reality Check

Needless to say, any real computer can perform certain computations much faster than a Turing machine. There are more realistic models of computations, in particular random access machines (RAMs), but they are a whole lot more difficult to deal with.

Also, the difference is much smaller than one might think.

Claim

The speed-up on a random access machine versus a Turing machine is only a low-degree polynomial.

For an algorithms person, this gap may seem ridiculously large, but we will see practical problems where it does not matter much.

Gimme a Break

The simulation would be much easier to describe if we used a RAM instead, we could describe the simulation in fairly close detail without going insane.

But: We would then have to deal with RAMs, rather than simple little friendly snugly Turing machines.

Trust me, we're better off this way.