UCT

Time Complexity

Klaus Sutner

Carnegie Mellon University

Spring 2024

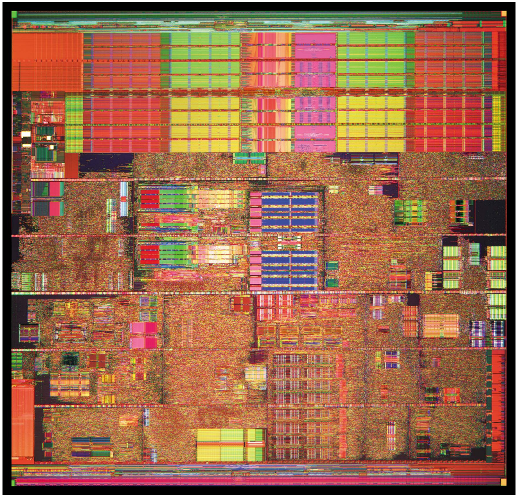
Getting Real

Our current classification of decision problems (arithmetical hierarchy) is very useful in **math**, but it mostly misses the boat when it comes to **computer science**.

Undecidability results are useful in that one need not bother to find a general algorithm for, say, Diophantine equations. On the other hand, decidability results mean that one can at least try to find a decent algorithm.

But what we really need is a much more fine-grained classification of problems that are decidable.

A Better Turing Machine



Physical Constraints

When computing with a microchip 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.

We will mostly deal with time and space.

Forget 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.

[†]Crypto mining is a crime against mankind.

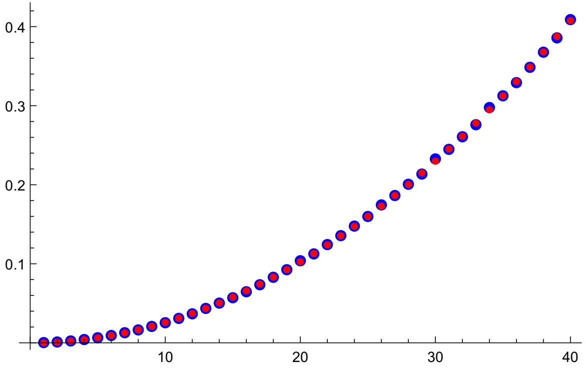
Analysis of Algorithms

Actual physical running time depends on many accidental factors and in particular processor technology.

In the analysis of algorithms one usually employs a (rather loosely specified) model of computation where we somehow "count steps," and only asymptotically. A natural question is whether this step-counting has anything to do with real physical time complexity.

Claim: 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, just multiply by the appropriate constant.

Example: Insertion Sort



Blue: predicted, red: measured physical running time.

RAMs

If one wants to be a bit more carefuly about this step-counting business one can come up with models of computation that are reasonably close to what happens in a digital computer. The standard model is a random access machine (RAM).

For our purposes, the gold standard will be Turing machines, and those are more primitive and will produce higher time complexity. However, 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 that it does not matter much for our purposes.

Why Not RAMs?

It might be tempting to suggest we should redo all of complexity and simply use RAMs or some other, more realistice model everywhere.

Tempting, but leading to disaster. Our arguments will be messy enough with Turing machines, the added complications of RAMs would make life much worse. Remember, we want real proofs, not just some more or less plausible intuition and gut feeling.

Trust me, we're better off sticking to Turing machines all the way, even though that introduces a minor mismatch between the concrete results.

Formalizing Complexity

Before we dive into the realm of standard complexity measures (time, space), it's a good idea to spend a minute trying to figure out what complexity means in general.

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. Convergence is clear.

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

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?

Reference Model: Turing Machines

Let's introduce the concrete complexity measures we are going to use from now on. 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 leading from the initial configuration to the final one:

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

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.

Size Matters

Our size measure is the length of the input string. We need $\Theta(\log|\Sigma|)$ bits to represent a symbol in the tape alphabet Σ, so up to a constant we are basically counting bits. For example, to represent a natural number n we need $\Theta(\log n)$ symbols (unless we write n in unary). This is called logarithmic size complexity and works naturally for Turing machines.

In algorithmic analysis, one often assumes that numbers have size 1: this is justified if we know ahead of time that the numbers won't require more than, say, 64 bits. In this case it makes perfect sense to use a uniform size complexity: all numbers are 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:

$$a = a_1 a_2 \ldots a_n$$

- 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 $O(nk)$.

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

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

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

Problems and Languages

Since Turing machines naturally operate on strings (rather than integers in classical computability theory), a decision problem $x \in A?$ now turns into a language recognition problem: we want to recognize the language $L \subseteq \Sigma^\star$ of all Yes-instances.

In automata theory one often talks about building an acceptor that recognizes a language. Same thing, just a change in terminology.

The Mother of All Problems

Fix some language $L \subseteq \Sigma^\star$ (the Yes-instances of some decision problem).

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 the recognition problem for Planar.

Planarity can be tested in linear time with a real algorithm, 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 TM that

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

We interpret halting in q_{Y} or q_{N} as indicating "accept" or "reject," respectively.

Exercise

Show that we could equivalently consider TMs that always halt and, in the end, leave a 0 or a 1 as output.

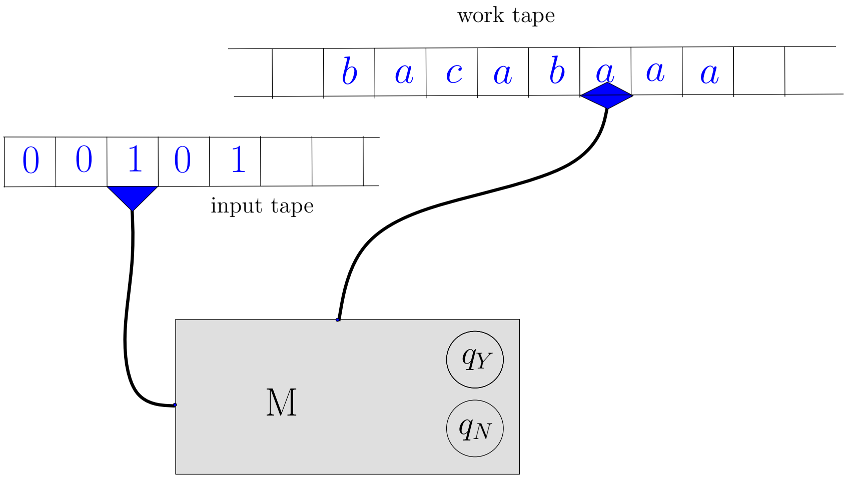
Input Tape

Recall that our machines have just one tape, and the input, output and intermediate results are all written on that tape at various moments during the computation (and we need to erase input and scratch space before halting).

This is perfectly fine, but it makes life a little easier if we assume that the input is given on a separate read-only input tape whereas the actual computation is carried out on a work tape. Ultimately we will also add a special write-only output tape, but for acceptors this is not necessary.

Separate input/output tapes are particularly 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 $e \in \mathrm{TOT}$, plus conditions on the permissible output. Acceptors are a semantical class, not a syntactical one.

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.

Exercise

Show that it is undecidable whether a Turing machine is an acceptor.

The Language

We can use an acceptor 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. Hence, 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

Decision problems are hugely important, but in general one also needs to worry about computing some function (the output is a data structure, rather than just a single bit).

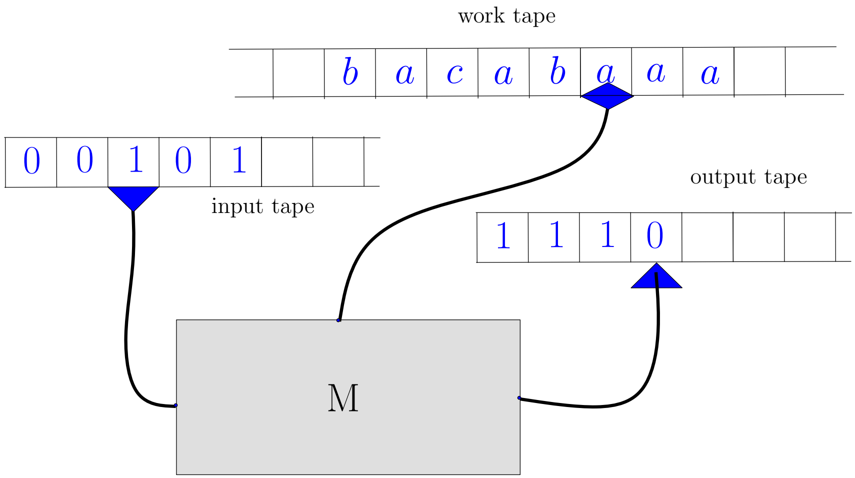
To compute a partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$, we use a slightly different type of Turing machine, a transducer.

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 according to their level of difficulty, their complexity.

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 $\mathcal{F}$.

Time Constructibility

There is a technical difficulty here: if the function f is wild, $\mathrm{TIME}(f)$ becomes quite unwieldy and not particularly interesting.

Definition

A function $t : \mathbb{N} \to \mathbb{N}$ is time constructible if there is a transducer 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 our machines always read all their 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 $2n + 1$, 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 $\mathcal{M}$ on a bigger machine $\mathcal{U}$ 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, the simulator $\mathcal{U}$ does the following:

- given input $x \in \Sigma^\star$ for $\mathcal{M}$,
- compute $B = t(|x|)$,
- use a counter to keep track of the number of simulated steps,
- stop the simulation when that number exceeds B,
- and do whatever is appropriate.

The Catch

No problem, right? It would be a royal pain to implement this on an actual Turing machine, but it is not difficult conceptually.

Right, but, for this to be useful, the whole simulation must run in time $t(|x|)$, and that includes the use of the clocking mechanism.

Alas, a malicious logician can cook up examples where the computation of $B = t(|x|)$ is hugely expensive, so the clocked simulation takes much longer. Whence the condition of time constructibility.

Lots of Annoying Details

If one tries to make constructions like the added clock efficient, things can get quite annoying. For example, here is a simple task: determine the length of a block of as.

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

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	$\ldots$	a_n	#
				0	0	1	$\ldots$		

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

Exercise

Figure out the details.

Simulation

There is more trouble brewing: we also need to worry about the actual simulation: we have some index e and want a master machine $\mathcal{U}$ to simulate $\mathcal{M}(x)$ for some machine $\mathcal{M} = \mathcal{M}_e$ (maybe up to t steps).

We are going to keep track of configurations of $\mathcal{M}$: current tape inscription, state and head position. And, of course, we have to store the lookup table e (the index for $\mathcal{M}$). 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 keeps the lookup table near the cell containing the state in the configuration of $\mathcal{M}$ to avoid zigzagging.

Alphabet Size

Another issue is that, in general, the alphabet of $\mathcal{M}$ can be much larger than the alphabet of the simulator $\mathcal{U}$—which is fixed once and for all. So a single symbol for $\mathcal{M}$ may require a whole block of symbols in $\mathcal{U}$[†].

We may as well assume that $\mathcal{U}$ has a two-track tape alphabet $\{␣, 0, 1\}^2$. Hence we pick up a factor of $\log|\Sigma(\mathcal{M})|$.

This typically does not affect any application of $\mathcal{U}$, but one needs to be aware of all these issues.

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

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} (doubly exponential), but they are typically not as important in the world of real algorithms.

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

Closure

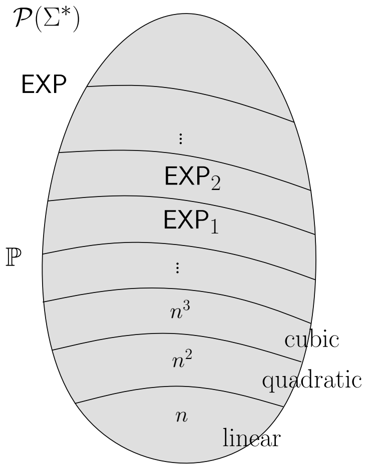
Lemma

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. We are relying on closure properties of polynomials here.

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$^{\text{TM}}$?

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, to first-order approximation.

Take this with a grain (a pound) of salt, we will have much more to say about feasibility later on. In the end, $\mathbb{P}$ does not really work, but it's a good first step.

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)

Suppose $T_{\mathcal{M}}(n) \leq f(n)$. 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 Sketch

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.

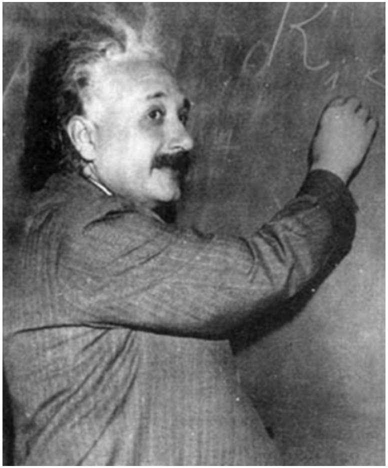
We modify the transition function accordingly[†], so that one step of $\mathcal{M}'$ corresponds to k steps by $\mathcal{M}$.

$\square$

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 very small tape alphabets.

[†] This requires quite a bit of work, try it.

Einstein



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

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 shrink 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

Just to be clear: on rare occasions one can directly exhibit a problem that separates the two classes.

Key example: $\mathrm{TIME}(n) \subsetneq \mathrm{TIME}(n^2)$.

In this case, we can show that testing for palindromes is in quadratic time, but not in linear time. Careful, though, this result is exceedingly brittle: it only works for one-tape TMs.

Unfortunately, this approach does not help with n^2 versus n^3 and so on, it's just a single, isolated result.

We need much bigger guns.

Halting?

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

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

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 by using clocks.

Furthermore, for technical reasons, we require that each machine is repeated infinitely often (there is a bigger index for a machine that performs the same computation in the sense that we get the same answers. So this is the opposite of a repetition-free enumeration, an idea that seems much more natural intuitively.

Diagonalization

Now define the diagonal function

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

Otherwise here means: the computation tries to use too much time before producing a result, something we can easily detect by clocking the machine.

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 $f(n) \log f(n)$.

Sketch: the simulator uses a three-track alphabet of the form Σ^3, which we may safely assume to be $\{\textvisiblespace, 0, 1\}^3$.

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

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

The Contradiction

Now assume δ can be computed by some machine in time g.

Since $g = o(f)$, there is an index e such that $\mathcal{M}_e$ also computes δ and $g(e) < f(e)/2$. But then $\mathcal{M}_e$ properly computes δ.

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

The real challenge here is to squeeze out every little possible speed-up for the simulator. We want a high-performance universal machine, not just any old piece of junk.

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.

[†] For some small value of 2.

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 is often very difficult and runs into open problems in complexity theory.

The Gap Theorem

Just to be clear: the hierarchy theorem depends heavily on the relevant time complexity 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$^{\mathrm{TM}}$.