5. LIMITS OF COMPUTATION: Undecidable Problems

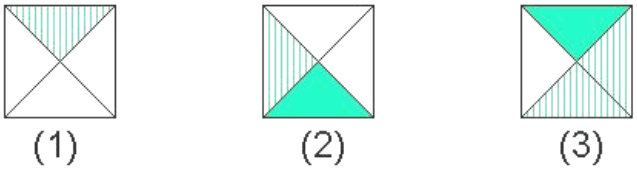
Unfortunately the hierarchy of difficulty at the end of the last section didn't tell the whole story. There are some problems so hard they are beyond even the 'non-elementary' class, and which are effectively *undecidable* (outside of the class of decision problems considered here, such algorithmically insoluble problems are referred to as *non-computable*).

To say that a problem is 'undecidable' means that there is no way, even given unlimited resources and an infinite amount of time, that the problem can be decided by algorithmic means.

There are many well-known examples of undecidable problems. We will look at just two: **tiling problems**, and the canonical example of an undecidable problem, the **halting problem**.

Example: tiling problems

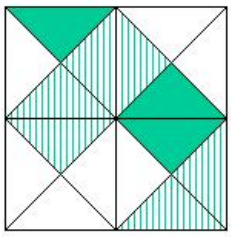
Consider the set of three tiles below:



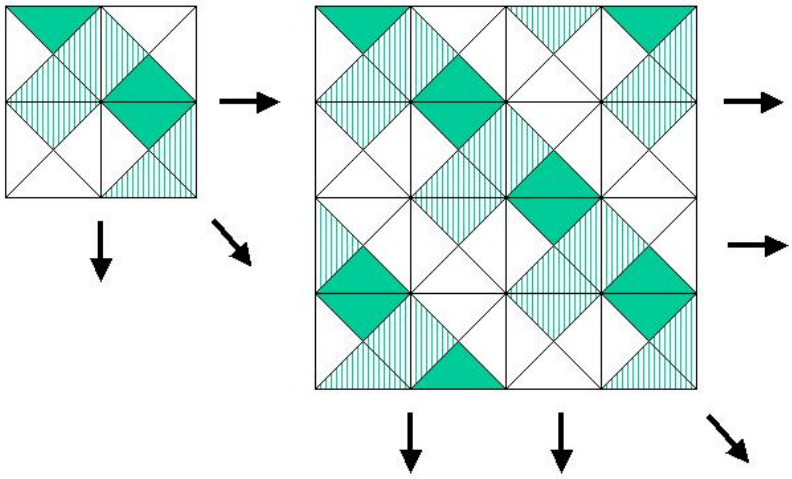
Could these be used to tile an arbitrary n×n area? The rules that have to be obeyed are:

- Only these three tile types can be used, but each can be used arbitrarily often.
- The edges of the tiles have to match up when they are used to cover an area.
- The tiles can't be rotated.

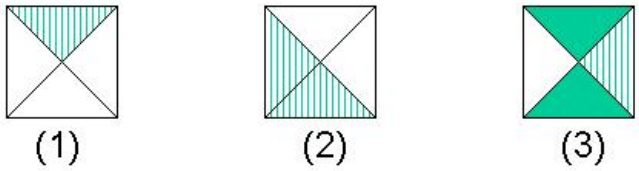
It's clear that they can when n=2:



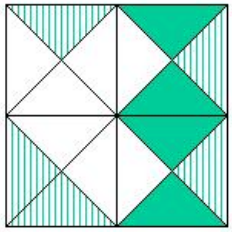
Also in this simple case one can see that the solution above extends easily to any n×n area:



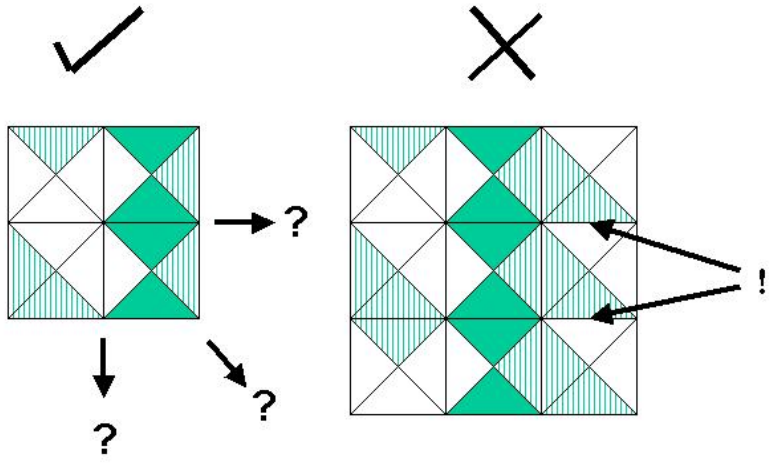
Now suppose that the bottom colours of tiles (2) and (3) are exchanged:



The new set of tiles can still cover a 2×2 area as below



but this particular solution can't be extended:



Moreover there are no others which will work; this set can't tile any area of size greater than 2×2.

The most common forms of the tiling problem are a generalisation of the problem above with a supplied set T of tile types (which cannot be added to or modified).

Bounded tiling problem:

Given a set T of tiles, can these be used to cover a specified n×n area?

This problem is in fact in the set NPC: it has a short certificate as a putative tiling of the n×n area can be checked (to see that the 'matching edges' and 'no rotations' rules have been obeyed) in time in $O(n^2)$, so is in NP; the problem also can be shown to be such that $B \leq_p$ (bounded tiling) for some $B \in$ NPC, so bounded tiling is in addition NP-complete. There appears to be no algorithm that can do significantly better than just checking through all possible arrangements of the tiles, something that takes an exponential amount of time.

Unbounded tiling problem:

Given a set T of tiles, can these be used to cover <u>any</u> nxn area? (Or equivalently, can they be used to tile the entire integer grid?)

This problem is much worse than bounded tiling; it is in fact undecidable as the display of no finite tiled area can prove—in general, we aren't just talking about especially easy tile sets such as the one first considered above—that any area can be tiled.

However it would be wrong to assume that the essence of the difficulty was that the number of things that might need to be checked (all possible areas) was infinite. It might have been that there was some rule that could look at a set of tiles and say "Yes, these have Property X, and so they can tile any area."

Hamiltonian and Eulerian Paths

A 'Property X' parallel for the case of intractability:

The **Hamiltonian Path Problem (HPP)** is in NPC but the **Eulerian Path Problem**, which looks like very similar—on the surface seeming to need a super-polynomial number of checks to be carried out—is actually in P.

The HPP is a variant on the HCP which relaxes the requirement that the route linking all n nodes and visiting each just once should also end up back where it began. (This seems to make it a bit easier but doesn't stop it joining the HCP, TSP and many other variations on these in NPC.)

The Eulerian Path Problem is identical to the HPP except that now it is edges can that be used once and only once, not nodes.

However this problem is definitely not in NPC; a p-time algorithm for it has been known since 1736, when Leonhard Euler showed a connected graph contains an Eulerian path *if and only if the number of edges emanating from all points (with a possible exception of just two) is even*.

This then, in the language used to discuss undecidable problems, is 'Property X', and can in fact be checked in just linear time. There is no need in the case of this problem to check through all possible paths (there are about $(n!)^2$ of them) even though it might at first appear this would be the case.

Example: the halting problem

Consider the program

```
while x ≠1 do
        x <− x - 2
```

For odd x (supposing for simplicity that the program's inputs are positive integers) this decrements by 2 until 1 is reached, when the program terminates. However when x is even decrementing by 2 each time misses 1 and so the program continues to decrease the original value forever.

Now consider

```
while x ≠1 do
        if even(x) then
                x <− x/2
        else
                x <− 2x + 1
```

Again there is no problem seeing for which inputs this terminates (this time only for x's that are powers of 2) and for which it runs forever.

For the variation below, however

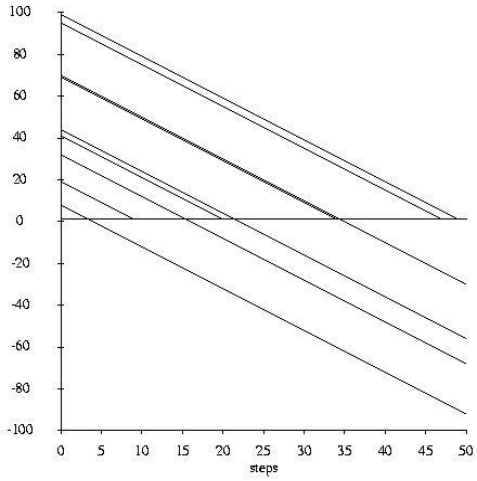
```
 while x ≠1 do
        if even(x) then
                x <− x/2
        else
                x <− 3x + 1
```

the situation is very different. Although in practice it does seem this program terminates for all positive integers x—though sometimes reaching very high and unpredictable values before it does so—no-one knows why. Consequently it's impossible to say *with certainty* before the program is run on a new input whether it will terminate or not. (If it appeared for some input not to, how would we know if it simply hadn't been given enough time?)

The examples below take random integers from {1,...,100} and run the three programs above for 50 steps, showing at each step the value reached:

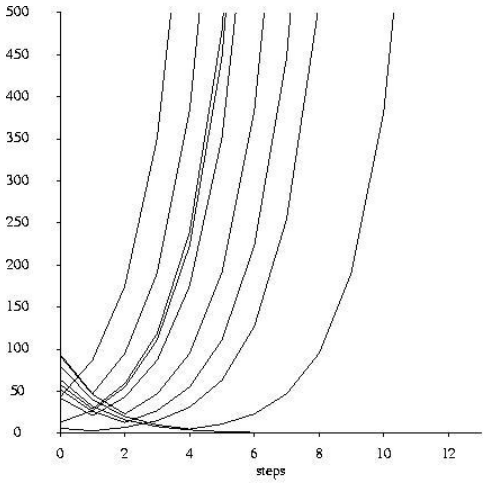
while x ≠ 1
x <− x - 2

(terminates for *odd x*)



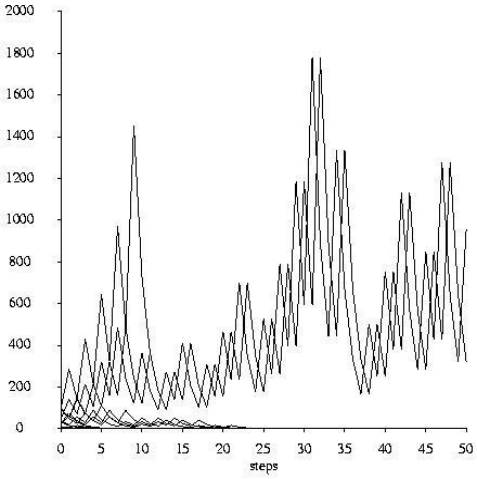
while x ≠ 1
if even(x) then
x <− x/2
else
x <− 2x + 1

(terminates when *x is a power of 2*)



while x ≠ 1
if even(x) then
x <− x/2
else
x <− 3x + 1

(terminates when ???)



All three of the programs have an infinite number of possible inputs (all positive integers) but for the first two cases we can say when the program will terminate without checking through all possible cases (in the first case Property X is 'being an odd number', in the second it's 'being a power of 2'). So this is more evidence that the presence of an unbounded, potentially infinite input set, with a seeming necessity to check all cases, is not the fundamental factor that determines decidability.

The halting problem is defined to be the problem, for any program R run on a legal input X (for example in the cases above that would mean X had to be a positive integer), of deciding if R halts (terminates) on X.

Note that this is about *any* program and not just some specific one (in the same way that the tiling problem was about *any* set of tiles T, not just some specific, and possibly easy to decide, set)—but that the halting problem is not, once R has been chosen, asking whether R terminates on *all* inputs, just on a given one, X.

The halting problem is the best known example of a problem that is not algorithmically decidable. Morever it has a special status because it's possible to prove its undecidability without reference to another undecidable problem, playing the same role with respect to undecidability that PSAT plays with respect to NP-completeness.

Fortunately it's a lot easier to explain why the halting problem is undecidable that it is to explain why PSAT is in NPC.

Showing the halting problem is undecidable

To show that the halting problem is not decidable it is necessary to show that there is no algorithmic procedure, or, loosely, 'program', which we'll here call H (in some appropriate language) that takes as its inputs the text of another legal program R in the same language, and an input X to R, and after some finite time halts and outputs 'yes' if R halts on X, and 'no' if R does not.

(Any uncertainty as to what is meant by a 'program' and a 'language'—and any suspicion that the result obtained here might be subtly dependent on some unreasonable assumptions about these—can be shown using the properties of **Turing Machines**, a simple but universal model of computation, to be unfounded as the argument below can then be made very precise in a way that shows that the conclusion to be reached can be extended to apply to all computers and languages, now and in the future.)

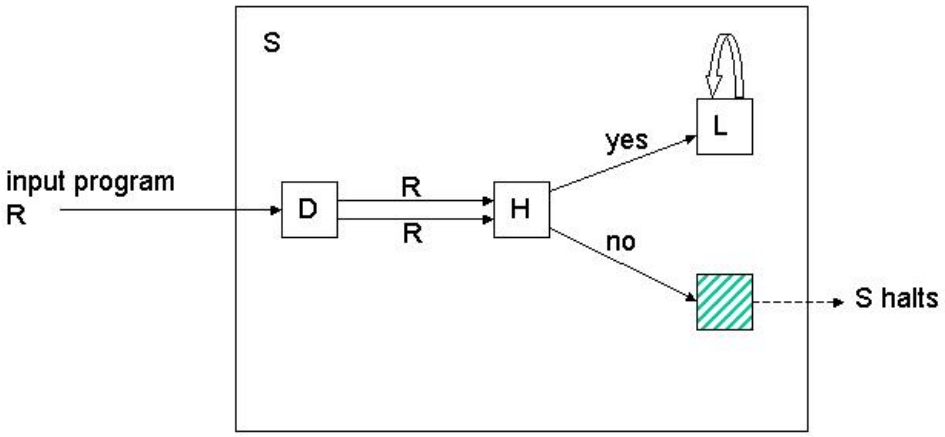
For the moment assume that there *is* such a decision program H; later, though, we will see that such a program in fact *can't* exist because it would lead to a logical contradiction.

Now assume that there is some other program S that does the following:

- Takes as its input the text of any legal program R.
- Makes a copy of it.
- Passes both copies to the hypothetical program H, which is thus being asked 'Does R halt when it is given its own text as input?'
- Goes into an infinite loop if the output of H is 'yes' (ie if R *does* halt on text-input R), and halts if the output of H is 'no' (ie if it has discovered that R *does not* halt on its own text as input).

You might think that it's strange to give a program its own text as input—perhaps that's where the hidden flaw in the argument could be..? But remember the way in which instructions and data are both binary-coded; without knowing the context it's impossible to know whether the contents of a given memory location are a piece of data or part of the executable code that is working on this data.

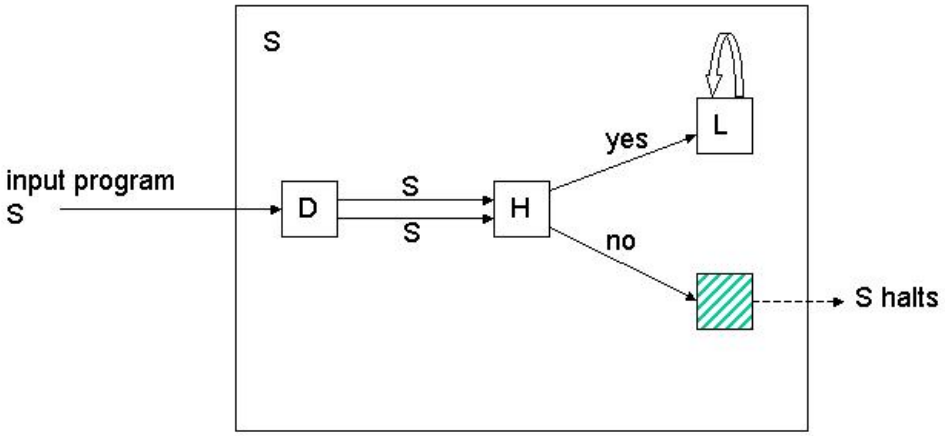
The situation so far then is as follows:



S has the following components:

D copies the input program R
H is the hypothesised 'halting checker'
L goes into an infinite loop if H does halt

Now let S have **its own text as input**:



This is where the contradiction arises.

First, assume that S **does** halt:

H(S,S) = yes, so L goes into an infinite loop, and hence S, the program that contains L, also fails to terminate. ***The assumption that S halts has led to the conclusion that it doesn't.***

Assume instead, then, that S **does not** halt:

H(S,S) = no, so S then halts. Now, even more straightforwardly, ***the assumption that S does not halt has led to the conclusion that it does.***

Clearly there is something wrong!

The suspect element has to be H, the hypothesised halting-checker. Everything else in the argument was perfectly legal and (by constructing the argument in the language of Turing machines) can in fact can be made very precise, removing any concerns about how 'programs' are defined. (If you are still uncomfortable with the proof there are other, somewhat differently constructed proofs you could look at—see Harel pp 210-211 for an example.) It just isn't possible, in general, to say whether a program terminates on a given input.

Showing other algorithmic problems are undecidable

As hinted above this is done for all other problems by giving 'reductions' that ultimately lead back to the halting problem, shown above to be undecidable in an argument that stands independently of any other algorithmic problem. These (possibly lengthy) chains of reductions are reminiscent of how new problems are recruited to the class NPC by being ultimately linked to PSAT. The difference is that here the reductions don't need to take p-time; provided that they take only a *finite* time the arguments about decidability still hold. The basic principal is that if an input to an algorithmic problem P can be turned into an input to another problem Q, and Q solved to give an answer that can be converted back into a P-answer ('P reduces to Q'), then *if P is undecidable Q must be too* (otherwise an answer to P could be obtained by passing the problem to Q).

Finite certificates

There is another sense in which questions about decidability are reminiscent of the questions of tractability dealt with in Section 2.1. In that section it was discussed how even though a problem might take an unfeasibly large (super-polynomial) time to be solved, in some cases—for problems in NP and NPC—a candidate solution could be checked for correctness in only polynomial time; this was referred to as the problem's having a 'short certificate'.

The concept that parallels this for questions of decidability is that of a **finite certificate**:

A decision problem has a finite yes-certificate (equivalently, no-certificate) if an assertion that the answer to the question posed in the problem is 'yes' (equivalently, 'no') can be checked in a finite amount of time.

Problems that have finite certificates of one kind or the other, even though they don't have finitely terminating algorithms, are referred to as **semi-decidable** or **partially decidable**. The existence of a finite certificate doesn't contradict the claim that the problem is effectively insoluble as there's no way to know, for a given input, whether the answer that will need to be checked will be a 'yes' or 'no'**—**this is the very thing that takes an infinite amount of time to discover.

Both the examples of undecidable problems discussed above are, in fact, semi-decidable in this sense.

The **halting problem** has a **finite yes-certificate** because one could show the assertion true by following through the steps of the program on the given input X and thus demonstrating explicitly that the program, when run on this input, does indeed terminate. However it does not have a finite no-certificate because to show a program did *not* terminate on a given input would by definition demand an infinite amount of time.

For the **unbounded tiling problem** the situation is the opposite: the problem has a **finite no-certificate** because one could simply exhibit an area for which there was no possible tiling (by the brute force method of considering all possible arrangements, if necessary**—**we aren't here concerned with how long things take, just whether they can be done at all). However there is no finite yes-certificate as this would involve showing there was a valid tiling of all possible n×n areas (or equivalently, one for the whole integer grid).

Problems which lack a finite certificate in either direction are 'less decidable'…

These are cases for which it's not possible to check the validity of either a yes-answer or a no-answer in a finite amount of time.

An example would be the **totality problem**, which instead of asking if a program R halts on a *given* input X asks if it halts on *all* inputs X. For the case of an asserted no-answer the argument that establishes that there can be no finite certificate is the same as for the halting problem, namely that the non-termination of R for at least one input case would have to be proven, taking an infinite amount of time by definition. For the case of a yes-answer, however there is now no finite certificate either because *all possible cases would have to be checked* since just one exception would make the claim untrue.

…and there are problems that are even worse

Although it's clear that the totality problem must in some sense be 'less decidable' than the halting problem, because in general there's no way to demonstrate the truth of a claim that it's either true or untrue for some program R, problems like this are not as bad as it gets! There are problems that share with the totality problem the property of not having any sort of finite certificate but which are still harder: these can generically be described as **highly undecidable problems**.

An example would be the **recurring unbounded tiling problem**. In this version of the problem a specified tile from the set is required to occur infinitely often. This problem certainly lacks a finite certificate in either direction. There is no finite yes-certificate for the same reason (the need to check absolutely everywhere in order to demonstrate that there were no exceptional regions that couldn't be covered) as for ordinary unbounded tiling. However there isn't now a no-certificate either because although it would be possible to show from a finite area that a no-assertion was correct because there would have to be a tiling rule violation of the usual kind (unmatched edges) it would *not* be possible to show on the basis of a finite area that the the rule about 'the special tile occurring infinitely often' had been broken**—**no-one has said that it should recur periodically, or with some minimum frequency, so its non-presence in a finite area means nothing.

However it's not just the lack of a finite certificate in either direction that causes the recurring unbounded tiling problem to be classed as highly undecidable, there is a quantifiable sense in which this problem really is harder, more undecidable, than the totality problem, which also lacks finite certificates. (You may be familiar with the idea that the number of real numbers is 'more infinite' than the number of integers and that this difference can be quantified, in spite of the idea of something's being 'more infinite' seeming to defy common sense; the situation here with these 'more undecidable' problems is comparable.)

Hierarchies of decidability

Just as decidable problems could be grouped into complexity classes (P, NP, NPC, EXPTIME, EXP(EXPTIME)…) undecidable problems can be grouped into classes with the same *degree* of undecidability. In each of these *computability classes*

- The problems are **computationally equivalent**, meaning that any one can be reduced to any other by a chain of **finite-time reductions** (of the kind used to discover if a newly-posed problem is undecidable, as discussed above).
- The problems are in some well-defined way 'less decidable' than all problems at preceeding levels.

The lowest level of the hierarchy of undecidability is occupied by the semi-decidable problems that have a finite certificate for either the 'is it the case that..?' or 'is it *not* the case that..?' version of the problem, such as the halting and unbounded tiling problems. Beyond that are problems like the totality problem which admit no finite certificates so can't be shown to be either true or untrue without unlimited resources. And beyond even that there are the 'highly undecidable' problems such as the recurring unbounded tiling problem. (In fact there is an infinite hierarchy of these last such that every new level contains problems still *more* highly undecidable than those below…)

Wrapping up:

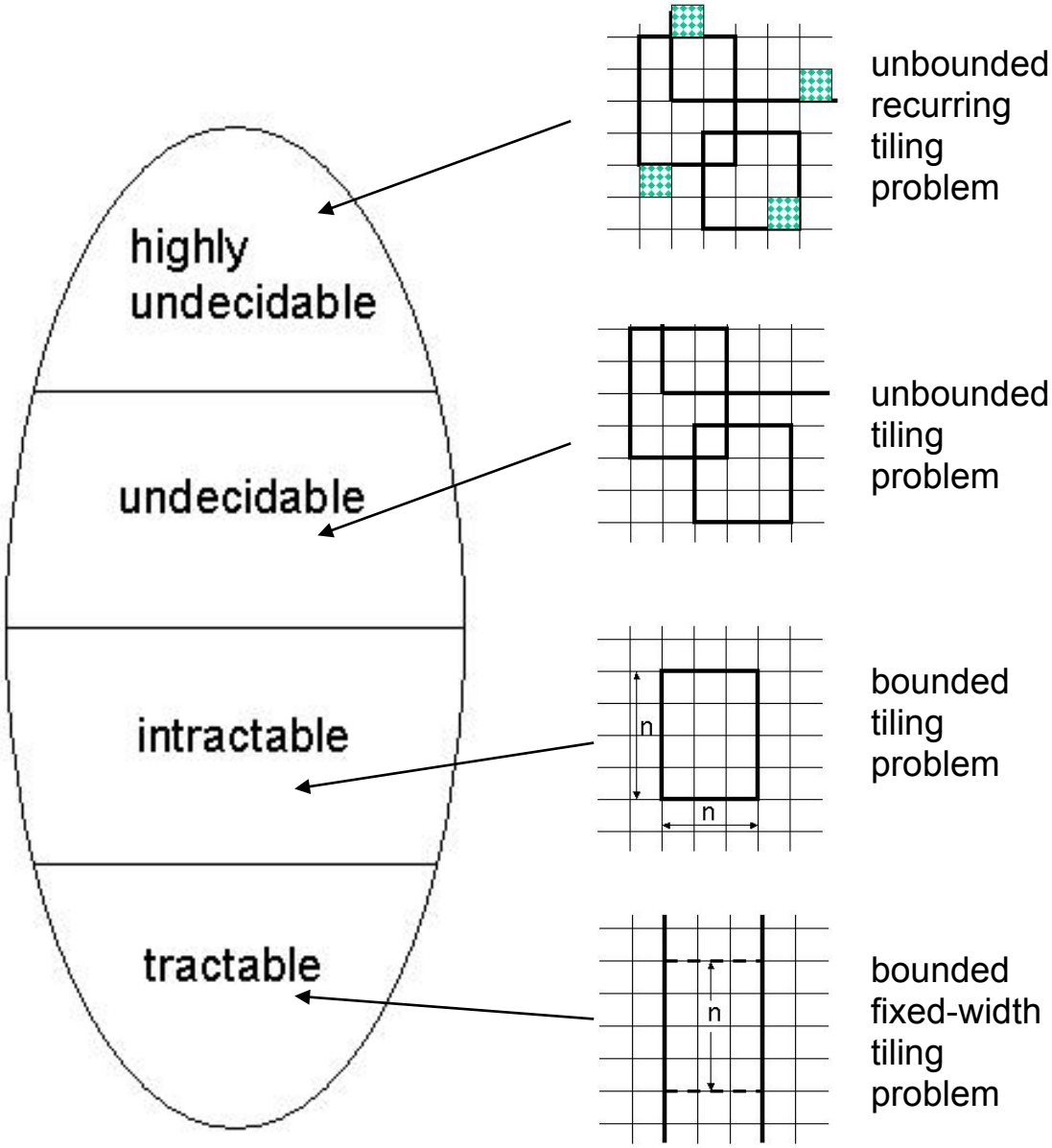
The situation with respect to what can be known in practice—the question of whether a problem is tractable—and what can be known in principle—whether a problem is even decidable—can be well summarised by variations on the tiling problem:

- Tiling an area in which only *one* dimension is variable (an n×w, w fixed, strip) can be shown to be a **tractable** p-time problem.

- Tiling a *given* n×n area is a **(probably) intractable** problem.

- Tiling *any* n×n area is (or equivalently, the whole integer grid) is **semi-decidable**.

- Tiling the integer grid so that a specified tile occurs infinitely often is **highly undecidable**.


highly
undecidable
undecidable
intractable
tractable
unbounded
recurring
tiling
problem
unbounded
tiling
problem
n
n
bounded
tiling
problem
n
bounded
fixed-width
tiling
problem