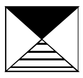
Beyond NP

- Provably intractable problems:
  - We have seen various NP-complete problems that take up exponential time for their solution.
    - But we cannot seem to prove that this is necessary.
  - There are other problems which are intractable but can be proven to be intractable.
    - That is: there is a proof that the minimum time must be exponential in the size of the input.
  - Examples:
    - The Towers of Hanoi problem.
    - Chess: The game tree for chess grows exponentially.

Non-computability

- Computation has its limits:
  - There are problems that cannot be solved by *any* algorithm (even if exponential time is allowed).
- Notions of computability:
  - In our discussion of computability we will utilize some programming language L.
    - L is to be defined later.
    - We will show that there are problems for which there is no algorithm possible, expressed in this language L.
  - We will also have an associated set of legal inputs for a problem.
    - A proposed solution of a problem must successfully work for *all* such inputs.

- Non-computability:
  - There are problems that can be easily stated but no algorithm can solve them.
    - Note: we do *not* mean “no efficient algorithm”.
    - Difficulty goes beyond simply requiring exponential time.
  - Example: “The Tiling Problem”.
    - We are given a finite set of square tiles of the same size, with four colours (not necessarily different) associated with each edge.
    - We wish to tile a rectangular area that has edge lengths of integer size.
      - The colours must match on an edge.
      - Rotations are not permitted.
      - We may use as many tiles of a given pattern as we wish.



- For some sets of tiles it is fairly easy to see that any size of rectangle could be properly tiled.
- For other sets of tiles it is fairly easy to establish that no such solutions exist
- Statement of The Tiling Problem:
  - Given a set T of tiles, determine whether it is possible to use this set T to tile **any** rectangle in the plane.
    - Note: the dimensions of a specific rectangle is **not** part of the problem instance.
  - There is no such algorithm.
    - We do not prove this: we just make some observations.

Non-computability of Tiling

- Can we solve the problem by brute force?
    - For some tile sets we can group tiles to make up repeating patterns that will guarantee success for any larger rectangle (result is TRUE).
    - For some tile sets we can find rectangles that cannot be tiled (result is FALSE due to this counterexample).
  - But there are tiling sets with no repeating patterns (this can be proven but it is very difficult to do so).
    - In these cases, we would have to work with all tilings for rectangles of size 1x1, 1x2, 2x2, 1x3, 2x3, 3x3 …*ad infinitum.*
    - There is no way to bound the time of this computation and so the problem is undecidable (in finite time).

Non-computability

- Definition:
  - A problem that has no algorithm for its solution is termed **non-computable**.
  - If the non-computable problem is a decision problem we use a more narrow term: we say it is **undecidable**.

- Problems similar to the tiling problem:
  - Can the tiles be laid down in a “domino fashion” so as to make a path from point v in the plane to another point w in the plane?
    - If we restrict the path to lie in a given rectangle the problem is decidable (exhaustive search).
    - If we do not restrict the path (it can wander the entire plane before arriving at w), then again the problem is decidable.
    - If we restrict the path to a half infinite plane the problem is undecidable!

Word Correspondence is Undecidable

- The word correspondence problem:
  - We are given two sequences of words: $(U_1, U_2, \ldots, U_n)$ and $(V_1, V_2, \ldots, V_n)$ each word made up of symbols taken from a finite alphabet.
  - The problem:
    - Can we find a sequence of words taken from the U set and a corresponding sequence of words taken from the V set (same indices being used) such that the concatenation of words from the U set is the same as the concatenation of words from the V set.
      - There is no algorithm that can do this for all possible U, V combinations.
      - The problem is undecidable.

Program Verification

- Can we design an automated program verifier?
  - We would like to design an algorithm that has as input:
    - a program implementing an algorithm A
    - a description of the problem that A is to solve.
  - Output of program verifier: “yes” if the program can solve the problem and “no” if the program cannot solve the problem.
    - By “not solving the problem” we mean that there exists at least one problem instance that algorithm A does not answer correctly or A does not terminate at all.
    - This problem is also undecidable.

- Are we perhaps asking too much?
  - Would it be undecidable because our program language is too complicated?
    - No. The problem is undecidable even for very simple programming languages.
  - Would it be undecidable because we want to investigate the computing actions of an algorithm that is too complicated?
    - No. We can make the program verification question even simpler by changing it to a more basic question:
  - Can we decide whether a program always terminates?
    - Even this “simpler” decision problem is undecidable!

The Halting Problem

– Before we give a more detailed description of the halting problem let us consider various algorithms and try to decide on whether they do or do not halt:

```
trivial_function(x)
while x!= 1 do
   x := x – 2;
stop;
```

– So, if *x* is odd the program will stop otherwise it loops forever.

– That was easy.
  - Now consider this one:

```
mystery_function(x)
while x != 1 do
   if (x is even) then x := x/2;
   else x := 3*x + 1;
stop;
```

– People have run this program for many, many positive integers.
  - Empirical evidence: The program may take a long time and it has always stopped.
  - BUT: no one has ever been able to *prove* that the program will stop for any given positive integer.
    – SO: even assessing whether a *specific* program will halt is not necessarily an easy thing to do.

What is an Algorithm?

– We will need a more precise specification of what it means to compute.

– Turing machines:
  - Turing machines are a mathematical model of computation developed by Alan Turing in the 1930's.

– The Church-Turing Thesis:
  - Any process that could be naturally called an effective procedure (or *algorithm*) can be realized by a Turing Machine (TM).

The Church-Turing Thesis

– We cannot prove this in a rigorous fashion but it is supported by a lot of evidence:
  - Many other models of computation have been developed and proved to be equivalent to TMs.
  - The class of functions computable by TMs is insensitive to numerous modifications of the TM definition.
  - There is no known effective procedure that would not have its formal counterpart in terms of a TM.

The RAM Model of Computation

- Motivation:
  - In this course we will use the RAM model of computation.
    - It is more like a conventional machine and we so we can present reasonably simple arguments related to the halting problem without getting involved with the more abstract specification of a TM.
    - We will still have enough precision in the model to state arguments without ambiguity.
    - It can be demonstrated that the RAM model of computation is equivalent to a TM.

Important Aspects of the RAM Model

- We have access to an arbitrarily large (not infinite) amount of time and storage.
  - Arbitrarily large means “as large as you need”.
    - The time and space requirements are not subject to some pre-specified upper bound.
    - So: large enough to do the job but still finite.
  - As we go through this material, keep in mind:
    - We are not concerned with efficiency of computation.
    - Our major concern is whether we can solve a problem by any means whatsoever (lots of time, lots of memory).

Other RAM model features

- Memory:
  - An array of registers: *R*1, *R*2, …
  - Each register can store an arbitrarily large integer.
- Program:
  - The program has a fixed number of lines.
  - Instruction set:

	INC op	Increment operand.
	DEC op	Decrement operand.
L:	IFZERO op	If operand is zero go to line L+1, otherwise go to line L+2.
	GOTO k	Go to line k.

- Operands:
  - Memory addressing capability of the machine.
    - The operand designation can be:

Ri	the instruction works with register *Ri*
@*Ri*	the instruction works with a register that is specified by register *Ri*.

  - We assume:
    - The input is stored in *R*1.
    - The output is stored in *R*2 after the computation finishes.
    - Apart from *R*1, the memory will always start with zeros.

Strength of the RAM Model

- Equivalence issues:
  - The model has enough features to simulate a TM.
  - A TM can simulate the RAM model.

  - So, considering the Church-Turing thesis, we may consider the RAM model to be as strong as any other model of computation.

Example

- Given n, compute 2^n:

```
1.  INC     R2          // R2 := 1
2.  IFZERO  R1          // while R1 != 0
3.  GOTO    17
4.  IFZERO  R2          // R3 := R2; R2 := 0
5.  GOTO    9
6.  INC     R3
7.  DEC     R2
8.  GOTO    4
9.  IFZERO  R3          // R2 := 2*R3; R3 := 0
10. GOTO    15
11. INC     R2
12. INC     R2
13. DEC     R3
14. GOTO    9
15. DEC     R1          // R1 := R1 – 1
16. GOTO    2
17.                     // If we get to here we are done
```

Lists, Characters, and Strings

- Values in the RAM model are natural numbers.
- More ideas:
  - Strings or lists of integers are represented numerically (as in a standard computer)
  - Characters can be represented as integers (think ASCII code).
  - A string can be represented as an integer: concatenation of ASCII numbers.
  - Program:
    - The program itself is a string and hence an integer.
  - Working with a more capable program:
    - From now on we will not write code in the above “assembly language”.
    - Instead, we will use the Church-Turing Thesis and assume that our pseudocode can be rewritten as a RAM program.

Recursive Functions

- Since everything is a natural number (integer) we can assume that the solution of a problem is defined as a function f mapping the natural numbers to the natural numbers:

$$f: N \rightarrow N$$

  - We will define $f(x) = 0$ if x does not represent a valid input.
- Definition:
  - Total function $f: N \rightarrow N$ is a **recursive function** if and only if there exists a RAM program that computes the function.
    - In this case, a total function is defined for *all* numbers in N.

The Halting Problem

- The Halting Problem is a decision problem:
  - Given a RAM program P and input x, does P halt on x?
  - Expressed as a function:

$$HALT\_CHECKER(P, x) = \begin{cases} 1, & if\ \ P\ halts\ on\ x \\ 0, & otherwise \end{cases}$$

  - We will show that this function cannot be computed.
  - Proof by contradiction:
    - We assume that $HALT\_CHECKER$ exists and then derive a contradiction.
    - Notes:If such a function exists, it must be some legal program in our RAM model. It can use as much memory space and time as it needs, but it must work as described for *every* pair (P, x).

The Halting Problem Theorem:

Theorem: No RAM program can compute function *HALT_CHECKER*.

- Proof:
  - We construct a new RAM program *FIDO*(*W*).
  - *FIDO* acts follows:
    - It takes a program *W* as input, replicates it, and calls *HALT_CHECKER(W,W), i.e.* with first and second arguments both equal to *W*.
    - A program as *input??* Sure, why not. A compiler takes a program as input. So does `fmt` (the unix formatter for C code)

The Halting Problem Theorem:

- Proof:
  - We construct a new RAM program *FIDO*(*W*).
  - *FIDO* acts follows:
    - It takes a program *W* as input, replicates it, and calls *HALT_CHECKER(W,W), i.e.* with first and second arguments both equal to *W*.
    - Since *HALT_CHECKER* can do its computation in finite time it will have a result for *FIDO*.
    - If the result from *HALT_CHECKER* is 0, *FIDO* is to halt.
    - Otherwise, *FIDO* should go into an infinite loop.

- *FIDO* can take any program as input, so let us see what happens when we feed *FIDO* to *FIDO*.
- A program that takes itself as input?
- Sure, why not… for example we can run the formatter **`fmt`** on itself:

```
%fmt fmt.c
```

- *FIDO* can take any program as input, so let us see what happens when we feed *FIDO* to *FIDO*.
  - The big question is: Does *FIDO* halt?
- Suppose *FIDO* does halt:
  - This means that the *HALT_CHECKER* returned a 0.
  - This implies that *HALT_CHECKER* has determined that when *FIDO* is fed *FIDO*, it will *not* halt.
  - Contradiction, so *FIDO* cannot halt (but see next point):
- Suppose *FIDO* does not halt:
  - This means that the *HALT_CHECKER* returned 1.
  - This implies that *HALT_CHECKER* has determined that when *FIDO* is fed *FIDO*, it will halt.
  - Again a contradiction!

- SO: *FIDO* cannot halt but it cannot not halt.
- What is wrong here?
- In the syntactical construction of the *FIDO* code we did nothing incorrect.
  - If *HALT_CHECKER* actually existed, *FIDO* could be built with the RAM model.
- So the occurrence of this "strange behaviour" is due to our belief in the workability of *HALT_CHECKER*.
  - The only conclusion we can draw is that *HALT_CHECKER* cannot act as claimed.
  - More precisely: it is impossible to have a program that works in the way that we have specified for *HALT_CHECKER*.

Diagonalization

- Another proof that *HALT_CHECKER* is noncomputable:
  - This proof will be more pictorial.
  - Recall that a program in the RAM model could be represented by a (large) number and its input could also be represented by a number.
  - So, the program *HALT_CHECKER*(P, x) is essentially taking two (usually large) numbers as input.
  - Assume *HALT_CHECKER* is computable:
    - Since it is a decision program we could then represent the set of all its outputs in a huge matrix of results:

– Each row holds results for a particular program P.

– Each column holds results for a particular input instance x.

- That is: entry i, j is the result of $HALT\_CHECKER(P_i, x_j)$:

input #

Program #

	0	1	2	3	4	…
0	1	0	1	1	1	…
1	0	0	1	1	1	
2	1	1	1	1	0	…
3	0	0	1	0	0	…
4	1	0	1	1	0	…
.	.	.			.	…

– So, if *HALT_CHECKER* is a computable function, then this table would hold the *HALT_CHECKER* output for *all* possible program P and input x combinations.

– If this is the case, then we see that the program *FIDO* (because of the way it is defined) will be a program that has its outputs completely specified as the complement of the main diagonal.

- In the given example:

FIDO input #

	0	1	2	3	4	…
FIDO	0	1	0	1	1	…

- The big question is: Would the table contain the *FIDO* program?
  - No, because *FIDO* is different from every row in the table.
    - If we compare *FIDO* and program P_i, then we see that their results are different in the i^{th} column.
- So, our assumption that the table contains all possible programs is incorrect.
- This contradiction implies that *FIDO* is noncomputable and so *HALT_CHECKER* is noncomputable.

  - Note the similarity between this proof technique and that of Cantor's theorem:
    "The set of real numbers is non-denumerable".

Proving a Function Q is Noncomputable

- If we can show that function Q can be used to solve *HALT_CHECKER*, then we can conclude that Q is noncomputable.

- Definition:
  - We say that function A is Turing reducible to function B (or $A \leq^T B$) if there exists an algorithm computing A using B as a subroutine.
  - Note the differences between $A \leq^T B$ and $A \leq_P B$:
    - $\leq^T$ is defined for all problems, not just decision problems.
    - There are no restrictions on the running time of the reduction algorithm.
    - There are no restrictions on the number of calls to B.

- Lemma:
  - If A is non-recursive (noncomputable) and $A \leq^T B$, then B is non-recursive.
- An example:
  - Consider:

$$HALT\_ALL(P) = \begin{cases} 1 & \text{if } P \text{ halts on all inputs} \\ 0 & \text{otherwise.} \end{cases}$$

  - Claim:
    - $HALT\_ALL$ is non-recursive.
  - Proof:
    - By reduction from $HALT\_CHECKER$, that is: $HALT\_CHECKER \leq^T HALT\_ALL$.

– Consider the following code for $HALT\_CHECKER$:

```
HALT_CHECKER(P, x):
   Q := encoding of the program “Q(y): return P(x)”;
   return HALT_ALL(Q);
```

– Notes:
- This is just a string manipulation.
- Program Q has argument y but it is ignored and P is run with the parameter x.

– We need to show that this algorithm is an implementation of $HALT\_CHECKER$.
- Assume P halts on x;
  then Q halts on all inputs y, $HALT\_ALL$ returns 1.
- Assume P does not halt on x;
  then Q never halts as well, so $HALT\_ALL$ returns 0.

– So: $HALT\_CHECKER \leq^T HALT\_ALL$ and so $HALT\_ALL$ is non-recursive (i.e. it is noncomputable).

Noncomputability Proof Template

- If we want to prove Q is non-recursive:
  1. Choose a function P that is known to be non-recursive.
  2. Write pseudocode for function P using Q as a subroutine.
  3. Justify that this pseudocode will compute P.
  4. Conclude that $P \leq^T Q$ and so Q is noncomputable.

Another Turing Reduction Example

- Problem:
  - Given two programs (P_1, P_2), do they exhibit the same behaviour?
  - Same behaviour is defined as:
    - If $P_1(x) = y$ then $P_2(x) = y$.
    - If $P_2(x) = y$ then $P_1(x) = y$.
  - Consider the program $EQUIV(P_1, P_2)$:

$$EQUIV(P_1, P_2) = \begin{cases} 0 & if\ \exists\, x \ni P_1(x) \neq P_2(x) \\ 1 & \text{otherwise.} \end{cases}$$

- Claim:
  - *EQIV* is non-recursive.
- Proof:
  - By reduction from *HALT_ALL*:

```
HALT_ALL(P):
   Q := encoding of the program “Q(x): return ∅”;
   R := encoding of the program “R(x): P(x); return ∅”;
   return EQUIV(Q, R);
```

  - We now show that the above program is an implementation of *HALT_ALL*:
    - Program *Q* always halts and returns ∅ for any input.
    - Assume *P* halts on all inputs:
      - Then program *R* always halts and returns ∅.
      - So, *Q* and *R* are equivalent.
    - Assume *P* does not halt for some input *x*:
      - Then program *R* does not halt on input x either; but $Q(x) = \emptyset$.
      - So, *Q* and *R* are not equivalent.

– Therefore:

$$HALT\_ALL \leq^T EQUIV$$

and so:

EQUIV is non-recursive.