Comparing Algorithms

Computability and Complexity

The final part of the course deals with the issue of assessing how

difficult

specific computational problems are to solve.

Given a description of a particular problem a number of questions arise:

A. Is it *possible* to design an algorithm which solves the problem?

B. Is it possible to design an *efficient algorithm* which solves the problem?

C. How can one tell if a given algorithm is the '*best possible*'?

These questions are imprecisely formulated:

- What exactly is meant by the term 'problem'?
- What distinguishes algorithms which can be *implemented* from those which cannot?
- What is meant by terms such as:
  - Difficulty
  - Efficient
  - Best

COMPUTABILITY THEORY

and

COMPUTATIONAL COMPLEXITY THEORY

are the fields of Computer Science concerned with the questions A, B, and C, raised earlier.

Computability Theory

is concerned with identifying one particular class of

INTRACTABLE PROBLEMS

(Those for which no 'effective' algorithm exists)

One aim of
Computational Complexity Theory

is to identify *solvable problems* that are intractable in the sense that
No Efficient Algorithm Exists

which solves them.

Both areas require a formal definition of the concepts:

Problem

Algorithm

Complexity Theory also proposes definitions for the ideas of

Problem Difficulty

Algorithm Efficiency

Best Algorithm

Computational Problems

At the lowest level of abstraction any computer program may be viewed as a means of producing

Specific Output Values

from

Given Input Data

i.e. Programs compute *functions* whose domain is the set of valid inputs and whose range is the set of possible outputs.

In practice the input and output are (ultimately) represented in *binary*

Hence any program may be viewed as (computing) some *function*

$$f : \{0,1\}^* \rightarrow \{0,1\}^*$$

($\{0,1\}^*$ is the set of all finite strings (or *words*) containing only 0s and 1s.

It is only necessary to consider such functions as have range $\{0,1\}$. These are known as

Decision Problems

Summary

- All computer programs may be viewed as computing *functions*
- Since all computers employ binary notation, such functions are defined over sets of binary strings.
- In considering the question 'what problems can be solved by computers' it is sufficient to concentrate on *decision problems*

Hence the question

What problems can be solved by computers?

is *equivalent* to

What *decision problems* can be solved?

Decision Problems

Any binary string can be viewed as the representation of some natural number. Thus for decision problems on binary strings we can concentrate on the set of functions of the form

$$f : \mathbf{N} \rightarrow \{0,1\}$$

That is 'problems' of the form

Input: n a natural number

Output: 1 if n satisfies some property; 0 if n does not satisfy it.

Examples:

EVEN: Returns 1 if n is an even number; 0 if n is an odd number

PRIME: Returns 1 if n is a prime number; 0 if n is a composite number.

ADA_PROGRAM: Returns 1 if n written in binary corresponds to the ASCII representation of a syntactically correct ADA program; 0 otherwise.

Notice that the last example shows that we are not restricting attention to merely 'arithmetic' decision problems.

What is a 'reasonable' algorithm?

In all programming systems

- Programs have a *finite* number of instructions.
- Progams make use of instructions and operations from a finite set of *basic operations*
- Programs should come to a halt for any valid input data.

An *effective algorithm* is one which meets these criteria.

Suppose we can show that an ADA program cannot be written which solves some problem.

Why should one take this as evidence that NO PROGRAM (ALGORITHM) **AT ALL** exists for the problem?

In other words

How can it be argued that results on computability apply to *all* types of computer? (Past, Present, and Future)

The answer is that such results are proved with respect to abstract *models of computation* that capture all the essential elements of a typical computer.

A 'typical computer' has

- Memory
- Input and Output Mechanisms
- A processor handling a finite instruction set.

In an abstract model we can permit an *infinite* supply of memory. The only restraint on an algorithm is that it can only use a finite amount of memory when dealing with any single input (which, if the algorithm terminates, is all that could be accessed).

This is **not** an unrealistic assumption: we do not want to classify a problem as unsolvable just because on *some* real machines there may not be enough memory available.

The first abstract model of computation was defined by Alan Turing in 1936. This had:

- An infinite memory of single bit cells.
- Its 'processor' dealt with only two types of 'instruction'

  A **stop** instruction which would return a result (0 or 1).

  A **test** instruction which would read and write to a single bit of memory, examine an adjacent memory location, and jump to another instruction.

Although the instruction set of the *Turing Machine* is extremely limited Turing machine programs can be written to solve **any** decision problem that can be solved on a 'normal' computer.

The Church-Turing Hypothesis

> The class of decision problems that can be solved by *any reasonable model of computation* is **exactly the same** as the class of decision problems that can be solved by Turing machine programs.

Since 1936 several thousand different models of computation have been proposed, e.g.

- Post Systems
- Markov Algorithms
- λ-calculus
- Gödel-Herbrand-Kleene Equational Calculus
- Horn Clause Logic
- Unlimited Register Machines
- ADA programs on unlimited memory machines

Every single one of these satisfies the Church-Turing hypothesis, i.e. anything that can be solved by a Turing machine program can be programmed in any one of the system above and *vice-versa*

The importance of the Church-Turing hypothesis is that it allows us, without any loss of generality, to consider computability results solely with respect to

ADA programs on
unlimited memory computers

Suppose P is some *decision problem*
The Church-Turing hypothesis asserts

There is no Turing machine program
that computes P

IF AND ONLY IF

There is no any ADA program
which solves P

Because we can (in principle) write a Turing machine program which can exactly simulate the behaviour of *any* given ADA program.

Similarly, we can write an ADA program which can exactly simulate the behaviour of *any* given Turing machine program.

The discussion above has, casually, assumed that there are decision problems for which algorithms do not exist.

Why should we believe that there are
actually any such problems?

(nothing above has given any evidence for the assumption made).

We can prove that there *exist* decision problems for which no algorithm exists by employing a method of argument called

Diagonalisation

This can used to show that there are 'more' different decision problems than there are different ADA programs.

If there are 'more' decision problems than programs then it must be the case that *some* decision problems cannot be solved by programs since since a single program can solve only a single decision problem.

Suppose we are given two sets of items, A and B, say, e.g.

A = Set of all students registered for the course 2CS46

B = Set of all students registered for the course 2CS42

How can we tell if there are more items in A than there are in B?

By 'counting' the number of items in each set.

> How many different decision problems are there?

A decision problem, f, is specified by saying for each number n what the value of $f(n)$ is. $f(n)$ can either be 0 or 1. So we have:

$$\begin{array}{c} 2 \text{ ways of choosing the value of } f(1) \\ \times \\ 2 \text{ ways of choosing the value of } f(2) \\ \times \\ 2 \text{ ways of choosing the value of } f(3) \\ \times \\ \cdots \\ \times \\ 2 \text{ ways of choosing the value of } f(n) \\ \times \\ \cdots \end{array}$$

Thus there are an infinite number of distinct decision problems.

How many different ADA programs are there which take a single number, n, as input and return a 0 or 1 as their result?

Infinitely many. Since there are clearly infinitely many such programs which contain a loop of the form

```
for i in 1..m loop
   s := 0;
end loop;
```

'Counting' is all right when the sets being compared are *finite*.

'Counting' does not appear to tell us anything when both sets are *infinite*.

Q: What do we *actually do* when we 'count' the number of items in a set?

A: To each item we attach a unique 'label': 1,2,3,4,...

Thus, in comparing the *sizes* of two finite sets, A and B, we attach 'labels' to the items in A, and 'labels' to the items in B and we say that

`A is larger than B'

if we use more 'labels' (i.e. *numbers*) with A than we do with B.

But why use 'numbers' as the 'labels' to count with?

We could use items in A to *label items in B* (i.e. count) and, if A is larger than B, then at least one item in A will *not have been used* to label any item in B.

$$A = \left\{ \begin{array}{l} ALF \\ BABS \\ CALUM \\ DORA \\ ED \\ FIONA \\ GARY \end{array} \right\} \quad \left\{ \begin{array}{l} AVA \\ BOB \\ CAROL \\ DAVE \\ ELLA \\ FRANK \end{array} \right\} = B$$

And we can 'count' B by using the 'labels'

ALF for AVA
BABS for BOB
CALUM for CAROL
DORA for DAVE
ED for ELLA
FIONA for FRANK

and now GARY has *not been used* to label any element of B and so we can say that

`A is larger than B'

What if both A and B are *infinite* sets?

If it can be shown that for *every way* of labelling items in B with items in A there is *at least one* item of A which is *not* used as a label then, again, we can state that:

`A is larger than B'

Diagonalisation

is a method of argument that allows this to be shown.

Let:

$A =$ The set of *all* decision problems, $f : \mathbf{N} \to \{0,1\}$.

$B =$ The set of all *ADA* programs.

We can *enumerate* (i.e. list) every single ADA program. That is there is an algorithm which will output every single different ADA program in turn, e.g.

```
i := 1;
loop
  if ADA_PROGRAM ( i ) = 1 then
    put ( Binary representation of i );
  end if;
  i := i + 1;
end loop;
```

(Recall that *ADA_PROGRAM* (n) is the decision problem which tests if n written in binary corresponds to the ASCII character representation of a compiling *ADA* program)

Thus we can talk about a

`First' ADA program, P_1
`Second' ADA program, P_2
`Third' ADA program, P_3
$\cdots$
`n''th ADA program, P_n
$\cdots$

i.e. the 1st, 2nd, 3rd, $\cdots$, $n'th$, $\cdots$, program that is output by the procedure above.

And so we can form an infinite table, T:

	1	2	3	..	n	..
P_1	$*P_1(1)$	$P_1(2)$	$P_1(3)$	..	$P_1(n)$	..
P_2	$P_2(1)$	$*P_2(2)$	$P_2(3)$	..	$P_2(n)$	..
P_3	$P_3(1)$	$P_3(2)$	$*P_3(3)$	..	$P_3(n)$	..
..		..		..	..	..
..		..		..	..	..
P_n	$P_n(1)$	$P_n(2)$	$P_n(3)$	..	$*P_n(n)$	..
..		..		..	..	..
..		..		..	..	..

where $P_i(j)$ is the result returned by the program P_i on input j. Thus $P_i(j)=0$ or $P_i(j)=1$

If it *is* the case that *every* decision problem, f, is computed by *at least one* ADA program, then in the table above which 'labels' each ADA program with a decision problem:

For every decision problem f

There is an ADA program, P_f, which is labelled with f.

i.e. There is a row, P_f, of the table such that $P_f(n) = f(n)$ for all n.

What about the following decision problem?

$$DIAG(n) = \begin{cases} 1 & \text{if } P_n(n) = 0 \\ 0 & \text{if } P_n(n) = 1 \end{cases}$$

DIAG is well-defined, since our table, *T*, above exists.

BUT

DIAG cannot be the decision problem

that is labelling P_1 ($DIAG(1) \neq P_1(1)$)

OR that labelling P_2 ($DIAG(2) \neq P_2(2)$)

OR that labelling P_3 ($DIAG(3) \neq P_3(3)$)

. . .

OR that labelling P_n ($DIAG(n) \neq P_n(n)$).

And so, the decision problem, *DIAG*, does not label any P_n, in the table *T*.

Now matter how the ADA programs are ordered, we can always define a decision problem, *DIAG*, as above, which does not label any program in the ordering.

In consequence:

There exist decision problems
which cannot be solved by
ANY
ADA-program

There exist decision problems
for which
NO ALGORITHM AT ALL EXISTS

Decision problems for which no algorithm exists are called:

Undecidable

Diagonalisation indicates that undecidable decision problems

exist

It does not help in *finding* an
explicitly defined
undecidable decision problem.

That is:

It does not identify an undecidable decision problem whose input and output behaviour can be given with a *finite* description.

The decision problems

EVEN
PRIME
ADA_PROGRAM

are explicitly defined (we can give a finite description of each problem; we do not have to state each individual output for each individual input: that would be an infinite length description)

But, all three of these are obviously,

decidable

i.e. an algorithm (ADA program) can be written to solve them.

Alan Turing in 1936 exhibited the first explicit example of an undecidable decision problem:

The Halting Problem

The Halting Problem, *HP*, is the following decision problem:

Input: An ADA program, *P*, (which takes as input a single number, *n*) and an input *w* for *P*.

Output: 1 if *P* comes to a halt when run with input *w*; 0 if *P* never halts when started with input *w*.

It may be objected that *HP* is not a decision problem in the sense that we have been using to date since:

- It has 2 items of input instead of one.
- One of its inputs is a ‘program’ not a number.

However we can define an *equivalent* decision problem *HP* 2 to which such objections do not apply.

HP 2 takes as input a single number, *m*. This number is a *valid* input if

1. *m* has a binary representation of the form

   x 00000000 y

   where **x** and **y** are binary strings;
2. **x** is the ASCII binary representation of an ADA program, P_x say, which takes a single number as input and returns a zero or one as output.

Clearly

- The problem of checking whether an input is valid is decidable.
- No algorithm exists for *HP* if and only if no algorithm exists for *HP* 2 (since we can translate from inputs for *HP* to inputs for *HP* 2 and vice-versa).

(Note: the 'encoding trick' described above can be applied to transform any decision problem with 'multiple inputs' into an equivalent one with a **single number** as input).

Given this fact we will deal with the more 'natural' problem *HP* instead of the precisely encoded version *HP* 2.

THEOREM
The Halting Problem (*HP*)
is
undecidable

Thus there is no algorithm which solves the Halting Problem.

PROOF

Suppose that there *is* an ADA program, *HALT*, which solves the Halting Problem, *HP*. That is:

There is an ADA program, *HALT*, which

Reads an ADA program, *P*, and a single number, *w*, as its input.

If *P* halts when given *w* as input then *HALT* returns 1 as output.

If *P* does not halt when given *w* as input then *HALT* returns 0 as output.

Since *HALT* is assumed to exist we can write an ADA program, *BAD* say, which uses *HALT* as a sub-procedure.


```
procedure BAD is
  Q : integer;
  function HALT (P integer;
                 w integer)
                 return boolean is
   if ADA_PROGRAM(P) then
    --*********************************
    -- Halting problem program goes
    -- here.
    --*********************************
   end if;
  end HALT;
  begin
   get (Q);
   if HALT (Q,Q) then
    loop
     put ("NOT OK");
    end loop;
   else
    put ("OK");
   end if;
  end BAD;
```


What does *BAD* do?

- *BAD* reads a single number, *Q*, as input.
- If *Q*, written in binary, corresponds to an ADA program which takes a single number as input then:
  - *BAD* tests whether the program represented by *Q* halts when given the **number** *Q* as input (using the Halting Problem procedure, *HALT*, to perform the test).
  - If the Halting problem procedure says that *Q* halts when started with input *Q* then *BAD* goes into a **non-terminating loop**.
  - If the Halting problem procedure says that *Q* fails to halt when started with input *Q* then *BAD* prints a message and **stops**.

- *HP* (which *HALT* solves) takes an ADA program and single number as input.
- The input for *BAD* should be a number which (when written in binary) is an ADA program which takes a single number as input.
- *BAD* is an ADA program (since *HALT* is an ADA program).

THEREFORE

There is a number,
BAD_NUMBER,
which when written in binary
corresponds exactly to the
ASCII representation of the
ADA program BAD

BAD_NUMBER is therefore a valid input for the ADA program *BAD*.

What happens if *BAD* is given
BAD_NUMBER
as input?

EITHER

BAD halts on input
BAD_NUMBER

OR

BAD does NOT halt on input
BAD_NUMBER

Suppose
BAD halts on input *BAD_NUMBER*.
This can *only* happen if
HALT(*BAD_NUMBER*, *BAD_NUMBER*)
is **false**,

But what does this mean? It means that

The program corresponding to *BAD_NUMBER* does not halt on input *BAD_NUMBER*.
But 'the program corresponding to *BAD_NUMBER*' is *BAD* i.e.

HALT(*BAD_NUMBER*, *BAD_NUMBER*)
is **false**,

means that
BAD does **NOT** halt on input
BAD_NUMBER

On the other hand, suppose

BAD does NOT halts on input *BAD_NUMBER*.

This can *only* happen if
HALT(*BAD_NUMBER*, *BAD_NUMBER*) is **true**,

But what does this mean? It means that

The program corresponding to *BAD_NUMBER* halts on input *BAD_NUMBER*.
But 'the program corresponding to *BAD_NUMBER*' is *BAD* i.e.

HALT(*BAD_NUMBER*, *BAD_NUMBER*) is **true**,

means that
BAD DOES halt on input *BAD_NUMBER*

In summary we have shown that:

- If a program to solve the Halting problem, *HP*, exists then the program, *BAD* exists.
- *BAD* on *any* valid input either halts or does not halt.
- *BAD_NUMBER* is a valid input for *BAD*.

  BAD on input *BAD_NUMBER* fails to halt only if *BAD* on input *BAD_NUMBER* halts.

  BAD on input *BAD_NUMBER* halts only if *BAD* on input *BAD_NUMBER* fails to halt.

We can therefore conclude that

No program to solve the Halting Problem, *HP*, can be written

i.e.

The Halting Problem is undecidable.

□

Computational Complexity Theory

Computability Theory is concerned with the question

For which decision problems do algorithms **exist**

Computational Complexity Theory is concerned with the question

For which decision problems do **efficient** algorithms exist

This raises the questions:

- What 'resources' do we wish to be employed 'efficiently'
- What do we mean by 'efficient'?

  The two significant 'resources' (or *complexity measures*) of interest are

TIME

and

SPACE

i.e. The number of *steps* an algorithm takes (in the worst-case) to return its answer. The amount of memory needed in which to run the algorithm.

Recall that a decision problem takes as input a *finite length* binary string and returns as output a 0 or a 1, hence a decision problem

$$f : \{ 0, 1 \}^* \rightarrow \{ 0, 1 \}$$

may be seen as an *infinite sequence* of decision problems [f_n], where

$$f_n : \{ 0, 1 \}^n \rightarrow \{ 0, 1 \}$$

is the problem f limited to inputs containing exactly n bits.

We measure the running time of an *algorithm* as the number of steps taken on inputs of length n

That is, if A is an algorithm (ADA program, say) that solves a decision problem, f, the *running time* of A is a function $R : \mathbf{N} \rightarrow \mathbf{N}$, so that R is the *worst-case number of steps taken on inputs of length n*.

The **Time Complexity** of a decision problem, f, is the *running time* of the 'best' algorithm A for f.

Thus we say that a decision problem, f, has

Time Complexity $T(n)$

if there is an an algorithm, A, for f such that

The number of steps taken by A on inputs of length n is always less than or equal to $T(n)$

The **Space Complexity** of a decision problem, f, is the *amount of memory* used by the 'best' algorithm A for f.

Thus we say that a decision problem, f, has

Space Complexity $S(n)$

if there is an an algorithm, A, for f such that

The number of memory locations used by
A
on inputs of length n
is always
less than or equal to $S(n)$

These definitions allow us to *classify* decision problems according to their Time (Space) Complexity. Thus we can define *complexity classes*, e.g.

$TIME(n)$ is the set of decision problems for which an algorithm running in at most n steps on inputs of length n exists.

$TIME(n^2)$ is the set of decision problems for which an algorithm running in at most n^2 steps on inputs of length n exists.

$TIME(2^n)$ is the set of decision problems for which an algorithm running in at most 2^n steps on inputs of length n exists.

And in general,

$TIME(T(n))$
is
the set of decision problems
for which an algorithm
running in
at most $T(n)$ steps
on all inputs of length n
exists.

Which complexity classes correspond to decision problems with 'efficient' (for run-time) algorithms?

Intuitively we regard an algorithm with worst-case run-time of n, n^2, n^3, etc as being 'efficient' since the time taken to return an answer for input of length n is 'acceptable' even for large values of n; and the amount of additional computation needed for inputs of length $n+1$ does not 'greatly exceed' the time on inputs of length n.

On the other hand, intuitively, we would regard an algorithm with worst-case run-time of 2^n, 3^n, n^n, $n!$, $n^{\log n}$, etc as being *'inefficient'* since even for moderate values of n the time taken to return an answer on input of length n is 'unacceptable'; and the amount of additional computation needed for inputs of length $n+1$ considerably exceeds the time on inputs of length n.

How can we distinguish run-times in the second category from run-times in the first?

All of the run-times in the first category have the following property:

For each 'efficient' run-time there is a constant, k, such that the running time is at most n^k when n is large enough.

A decision problem, f, for which there is an algorithm taking at most n^k steps on inputs of length n (i.e. $f \in TIME(n^k)$) is called a

Polynomial Time Complexity Decision Problem

All of the run-times in the second category do *not* have this property, i.e.

For each 'inefficient' run-time there is **NO** constant, k, such that the running time is at most n^k when n is large enough.

A decision problem, f, for which there is **NO** algorithm taking at most n^k steps on inputs of length n (i.e. $f \notin TIME(n^k)$) is called a

Non-Polynomial Time Complexity Decision Problem

It is possible to show (by diagonalisation) that for all run-time functions, $T(n)$:

There *exist* decision problems, f,
such that
the Time-Complexity of f
is greater than $T(n)$

(thus there are decision problems which *do* have non-polynomial time complexity)

There are very few 'natural' examples of decision problems for which such *lower bounds* have been proved.

A major open problem in Computational Complexity Theory is to develop arguments by which important computational problems can have their time complexity described *exactly*.

For 'almost all' 'interesting' problems no lower bound on Time Complexity larger than $\Omega(n)$ has been obtained.

Example Open Problems

We have seen that multiplication of 2 n-digit numbers has Time Complexity at worst $n^{\log_2 3}$. It is conjectured that

$$Multiplication \in TIME(n \log n)$$

This conjecture remains unproven.

Multiplication of 2 $n \times n$ integer matrices can be done in time $O(N^{1.81})$ (where $N = n^2$). It is conjectured that

$$Matrix{-}Multiplication \in TIME(N \log N)$$

Again this remains unproven.

It is unknown whether there is a polynomial time algorithm that can determine if a given n-digit number is a Prime Number.

It is unknown whether there is a polynomial time algorithm that will factorise an n digit number.

Computational Complexity Theory involves a large number of subfields each of which is ultimately concerned with problems such as those above, e.g.

Algebraic Complexity Theory
Complexity of Probabilistic Algorithms
Complexity of Parallel Algorithms
Machine-Based Complexity Theory
Structural Complexity Theory
Complexity of Approximation Algorithms
Axiomatic Complexity Theory
Complexity of Counting and Enumeration
Average Case Complexity
Complexity Theory of Boolean Functions

The study of the last field is over 50 years old.

It is convenient to use a single symbol to denote the class of all decision problems for which time efficient algorithms exist.

$$P = \{ f : f \text{ has a polynomial time algorithm} \} = \bigcup_{k=1}^{\infty} TIME(n^k)$$

The Complexity Class *NP*

Consider the following:

Composite:
Input: n-bit number, x
Output: 1 if x is a composite number; 0 otherwise.

Satisfiability: (*SAT*)
Input: A Boolean expression, F, over n variables $(x_1, \ldots, x_n)$
Output: 1 if the variables of F can be fixed to values which make F equal **true**; 0 otherwise.

Hamiltonian Cycle (*HC*)
Input: n-node graph, $G(V, E)$
Output: 1 if there is a cycle of edges in G which includes each of the n nodes exactly once; 0 otherwise.

CLIQUE
Input: n-node graph $G(V, E)$; positive integer k
Output: 1 if there is a set of k nodes, W, in V such that every pair of nodes in W is joined by an edge in E; 0 otherwise.

3-Colourability (3−*COL*)
Input: n-node graph $G(V,E)$
Output: 1 if each node of G can be assigned one of three colours in such a way that no two nodes joined by an edge are given the same colour; 0 otherwise.

All 5 of these (and infinitely many other) decision problems share a certain property. They are

'Checkable' by efficient
(polynomial time) algorithms

What does this mean?

In order to solve each of the decision problems above we have to demonstrate the *existence* of a particular object which has a certain relationship to the input.

That is, to show that:

x, is composite we 'only' have to find a factor of x.
$F(X)$ is satisfiable we 'only' have to find an assignment to X that makes F true.
$G(V,E)$ is hamiltonian we 'only' have to find a suitable cycle in G.
$G(V,E)$ has a k-clique we 'only' have to find a suitable set of k nodes in V.
$G(V,E)$ is 3-colourable we 'only' have to find a suitable colouring of V

So for the examples: Given an *input instance*, I, (number, Boolean expression, graph) one has to find a *witness*, W (number, assignment, cycle, set of nodes, colouring) such that a particular relationship holds between the witness and the input.

e.g.

That W divides I with no remainder.
That W makes I take the value **true**.
That W is a cycle in I containing each node of I exactly once.
That W is a set of k nodes in I each pair of which is connected by an edge of I.
That W does not assign the same colour to two nodes in I which are joined by an edge of I.

How difficult is it to *decide* if something *is indeed* a valid witness for instances to these decision problems?

It is easy.

One can check if :
W is a factor of I (by division)
W satisfies I (by evaluating the expression)
W is a hamiltonian cycle in I (by testing if I contains all the edges in W)
W is a k-clique in I (test if each pair in W is joined by an edge of I)
W is a colouring of I (test if each edge in I joins nodes with different colours in W)

All of these can be carried out by efficient algorithms.

In summary:

For any decision problem, f, with each possible input size, n, there is a set of *potential* witnesses W_n.

To solve a decision problem 'all' that is needed is an algorithm to determine if there is a *genuine* witness, $W \in W_n$ for any given input instance I of size n.

In principle, one could do this by going through each element, W of W_n, in turn and *checking* if W is a genuine witness for I.

However, the question
'Is W a genuine witness for I'
(for the decision problem f)

(e.g. Is y a factor of x? (for Composite)
Does α make $F(X)$ equal **true**? (for *SAT*)
Is C a hamiltonian cycle in $G(V,E)$? (for *HC*)
Is K a k-clique in $G(V,E)$ (for *Clique*)
Is χ is 3-colouring of $G(V,E)$ (for *3-Col*)
)

is **itself** a decision problem (based on f)

So for any decision problem, f, we can define another decision problem, $CHECK_f$

$CHECK_f$:
Input: I input instance for f; W possible witness that $f(I)$=1.
Output: 1 if W is a genuine witness that $f(I)$=1; 0 otherwise

Now we have seen that
$CHECK_{Composite}$
$CHECK_{SAT}$
$CHECK_{HC}$
$CHECK_{Clique}$
$CHECK_{3-Col}$

are all *polynomial time decidable*, i.e. all of these have efficient algorithms.

The notation, NP, denotes the class of all decision problems, f, with the property that $CHECK_f$ is polynomial time decidable,

$$NP = \{ f : CHECK_f \in P \}$$

Thus for decision problems we have two 'variants'

Finding version of f
(f itself)
which asks
Is there *any* witness W for I (with f)
and
Verifying version of f
($Check_f$)
which asks
Is this **specific** W a witness for I (with f)

P is the class of problems with efficient 'finding' algorithms.

NP is the class of problems with efficient 'verifying' algorithms.

How are P and NP related?

If $f \in P$ then certainly $Check_f \in P$ and so $f \in NP$ also. Thus,

$$P \subseteq NP$$

Undoubtedly the most important open question in modern Computational Complexity Theory is:

Is P a proper subset of NP?

i.e. Are there decision problems f such that $Check_f \in P$ (i.e. $f \in NP$) but for which $f \notin P$

Or, in informal terms,

Are there decision problems f such that *checking a solution* (witness) is much easier (polynomial time) than *finding a solution*?

The following problems are a fraction of the tens of thousands of problems which are known to be in *NP* but for which no efficient algorithm has been found (despite, in some cases, centuries of work on them)

Hamiltonian Cycle
Satisfiability
3-Colouring
Clique
Travelling Salesman Problem
Integer Knapsack Problem
Longest Path in a Graph
Edge Colouring
Optimal Scheduling
Minimising Boolean Expressions
Minimum Graph Partition
Composite Number
Primality

With the exception of the last 2, it is generally believed that no efficient algorithm exists for any of these decision problems.

i.e. They are all **intractable**.

This has yet to be **proved**.

The concept of **NP-complete** decision problems offers a possible route to resolving whether $P = NP$ or $P \neq NP$.

NP-completeness

The idea underlying the concept of *NP*-complete problems is to describe formally what can be understood as the

'most difficult' problems in *NP*

That *NP*-complete problems *are* (intuitively) the most difficult in *NP* follows from the fact that we may prove

$P = NP$
if and only if
some *NP*-complete problem, *f*,
has a polynomial-time algorithm

Suppose S and T are two decision problems and suppose we can construct an ADA program (algorithm) to solve T that is of the form


```
procedure T is
  x : T_input_type;
  y : S_input_type;
  function S ( z : S_input_type)
           return boolean is
  --****************************************
  -- ADA function for decision problem S
  -- goes in here.
  --****************************************
  end S;
  function transform ( t : T_input_type)
           return S_input_type is
  --******************************************
  -- ADA function which transforms an
  -- input instance for T into an input
  -- instance for S.
  --******************************************
  end transform;
  begin
    get (x);
    y := transform (x);
    if S(y) then
      put (1);
    else
      put (0);
    end if;
 end T;
```


In this *transform* is a function which takes an input instance for T and changes it into an input instance for S. For example if T is a decision problem on numbers and S is one on graphs, then *transform* given a number would construct a graph using that number.

Now suppose that both:

transform **and** S
have efficient (polynomial)
algorithms

Then the procedure above gives an efficient algorithm for T.

It follows that (with the above procedure form)

There is an efficient algorithm for T
if
there is an efficient algorithm for S

But, more importantly,

If we can *prove* that there is
NO efficient algorithm for T
then
we have *also* proved that there is
NO efficient algorithm for S

This means that if we have of an *efficient* procedure, τ which

- transforms input instances x for T into input instances, $\tau(x)$, for S
- is such that $T(x) = 1$ if and only if $S(\tau(x)) = 1$

Then we have established a relationship between the Time-Complexity of S and the Time-Complexity of T.

If S and T are decision problems for which such an efficient transformation procedure from inputs for T to inputs for S exists then we say that

The decision problem T is
polynomially reducible
to the decision problem S

This relationship denoted

$$T \leq_p S$$

How does the idea of '*polynomial reducibility'* help in addressing the question of whether P equals NP?

Suppose we have a decision problem, f, for which we can prove the following:

1. $f \in NP$, i.e. f there is an efficient *checking* algorithm for f.
2. For every other decision problem, g in NP, it is the case that $g \leq_p f$, i.e. for every decision problem, g with an efficient checking algorithm there is an efficient procedure, τ_g, that transforms inputs for g into inputs for f.

Then if there **are** decision problems in NP for which efficient algorithms **do not exist** then

the decision problem *f* must be one such problem

In other words, for such a decision problem f:

$$P = NP$$

if and only if

f has an efficient algorithm (i.e. $f \in P$)

A decision problem, f, which has these properties, i.e.

1. $f \in NP$
2. $\forall\ g \in NP \quad g \leq_p f$

(f has an efficient checking algorithm and every problem with an efficient checking algorithm can be efficiently transformed to f)

is called an

NP-complete decision problem

The NP-complete decision problems are the 'most difficult' problems in NP in the sense that if

Any **single** NP-complete problem does not have an efficient algorithm
then
NO NP-complete problem
has an efficient algorithm

Conversely

If we find an efficient algorithm for
just **one** NP-complete problem
then
we can construct efficient algorithms
for all
decision problems in NP

Problem: The definition of NP-completeness requires the following to be done in order to show that a problem f is NP-complete

- We have to find an efficient checking algorithm for f (i.e. prove $f \in NP$)
- We have to show that *every* problem with an efficient checking algorithm can be transformed into f (i.e. prove $\forall\ g \in NP \ \ g \leq_p f$)

The first is usually (relatively) easy.

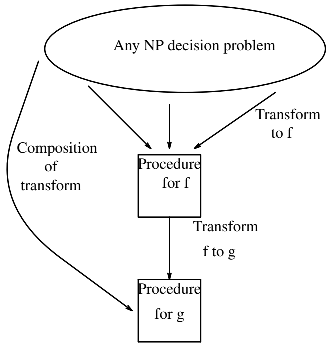
But for the second, how can we reduce **infinitely** many decision problems to a **single** problem?

Before dealing with this we can note the following: suppose we have shown for two decision problems, f and g say, that:

- f is NP-complete.
- $g \in NP$
- $f \leq_p g$

Then we have **also** shown that g **is NP-complete**.

Why?



So we have a polynomial reduction from every problem in NP to g by

1. Converting an input instance x for h (in NP) to one $\tau_h(x)$ for f.
2. Converting the input $\tau_h(x)$ for f to an input instance for g.

The first is possible since f is NP-complete. The second since there is a polynomial reduction from f to g.

This argument shows that once we have identified a **single** NP-complete decision problem, f, in order to prove that another decision problem, g, is NP-complete 'all' we have to do is find an efficient algorithm for transforming input instances for f into input instances for g, i.e. prove

$$f \leq_p g$$

But, this still leaves the problem of finding a ‘first’ *NP*-complete problem, that is of finding some way of

transforming infinitely many
decision problems
into
just **one**
decision problem

In 1971 Steven Cook proved one of the most important results in modern computational complexity theory. This result is (now) known as

Cook’s Theorem

SAT is NP-complete

Proof: Omitted (not difficult, just long)

Within a few months of Cook’s result dozens of new *NP*-complete problems had been identified. By 1994 well over 10,000 basic *NP*-complete decision problems were known.

Thus it is known that the following problems are all *NP*-complete

CLIQUE
Hamiltonian Cycle
Travelling Salesman Problem
3-Colouring
Set Partitioning
Edge Colouring
Longest Path
Integer Knapsack

A proof that a decision problem is *NP*-complete is accepted as evidence that the problem is **intractable** since a fast method of solving a single *NP*-complete problem would immediately give fast algorithms for **all** *NP*-complete problems.

It generally though to be **extremely unlikely** that any *NP*-complete problem has a fast algorithm.

It has still to be **proved** that any *NP*-complete problem does not have an efficient algorithm, i.e. the question of whether *P*=*NP* is still open.

While most of the classical decision problems have now been determined as being in *P* or being *NP*-complete a number of important problems are still open. In particular

Primality
Composite Number

Both are known to be in *NP*. Neither has been shown to be *NP*-complete. It is believed that both are in fact solvable by *efficient algorithms*.

The structure of P, NP, and PSPACE