

---

# COMPUTABILITY

*E.K.Burke*

## 1. Effective Procedure

An ideal formulation of an effective procedure involves:

1. A language in which a set of rules can be written

---

2. A rigorous description of a single machine which can interpret the rules in the given language and carry out the steps of a specified process.

If the Church - Turing Thesis is accepted this ideal state of affairs can be achieved.

---

## **Church - Turing Thesis**

An effective procedure is one that can be realised by a Turing machine. We will define a Turing machine in 1.2.

---

**Note:** There is no way of proving or disproving the Church - Turing Thesis. It is a 'reasonable' assumption to make precise the vague idea of 'Computability'.

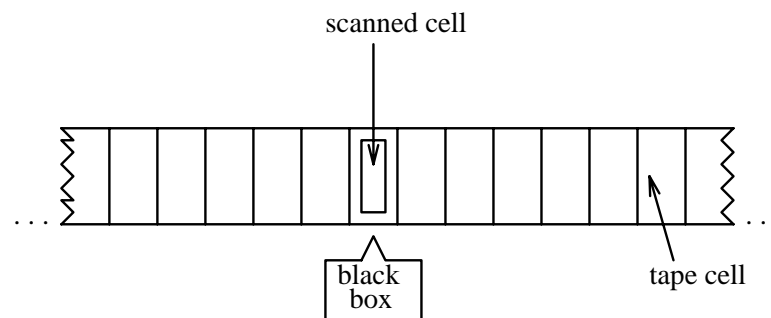
---

All the attempts (listed above) to classify the set of effective procedures have been shown to be equivalent to Turing's. This is one of the major arguments in favour of accepting the Church - Turing Thesis.

---

## 1.1. Turing Machines

A Turing machine consists of a black box (which can be in any of a finite number or internal states) and a linear tape which is infinite (or at least arbitrarily large) in both directions



---

Each Turing Machine has 2 finite alphabets:

1. A set of tape symbols denoted by  $\{s_0, s_1, \dots, s_m\}$ . Each cell of the tape contains one of these symbols.
2. A set of internal states denoted by  $\{q_0, q_1, \dots, q_n\}$ .

---

### **1.1.1. The Operation of a Turing Machine**

Suppose we have a Turing Machine  $T$ . We divide the operation of  $T$  into discrete time steps starting at  $t=0$ . To describe the operation of  $T$  from step  $t$  to step  $t + 1$  we require the tape configuration at time  $t$  together with:

- 
1. The internal state of T at time t (the current state  $q(t)$ )
  2. The symbol in the scanned cell at time t (the current symbol  $s(t)$ )

---

Depending on  $q(t)$  and  $s(t)$   $T$  will be instructed to:

3. Enter a new state  $q(t+1)$ , not necessarily different from  $q(t)$ .
4. Replace  $s(t)$  with a new symbol  $s(t+1)$ , not necessarily different from  $s(t)$ .

---

5. Move one cell to the left or right. Thus step  $t \rightarrow t+1$  of the operation of  $T$  can be described by the quintuple  $(q(t), s(t), q(t+1), s(t+1), \text{dir})$  where  $\text{dir} = \text{Left}$  or  $\text{Right}$ .

---

Now we can specify the complete computation of  $T$  by giving:

(a) The initial tape configuration of  $T$ . This includes the entire contents of the tape together with the scanned cell at  $t=0$  and  $q(0)$ .

(b) A list of quintuples of the above form. These quintuples dictate the exact step-by-step operations of  $T$ .

---

**Note:** We usually assume that each cell of the tape contains 0 unless we indicate otherwise.

---

## **1.1.2. Examples of Turing machines**

### **1.1.2.1. A Parity Counter**

Here the input of the machine is a binary number sequence. The output is 0 or 1 according as the number of 1's in the input sequence is even or odd respectively.

---

Initially the binary sequence is written onto the tape with the symbol 'A' indicating the 'beginning' and 'end' points of the sequence.

---

Also the machine initially scans the left-most digit of the sequence (in state  $q_0$ ).

For example we could have:

...0A10110101A0...  
    ↑  
     $q_0$

---

The Turing Machine is specified by the following set of quintuples:

$(q_0 0 q_0 0 R)$      $(q_1 0 q_1 0 R)$

$(q_0 1 q_1 0 R)$      $(q_1 1 q_0 0 R)$

$(q_0 A \text{Halt } 0)$      $(q_1 A \text{Halt } 1)$

The Tape alphabet is  $\{0,1,A\}$

The State alphabet is  $\{q_0, q_1\}$

---

**Note:**

(a) if we replace the above quintuples by the following ones we would not erase the sequence:

$(q_0 0 q_0 0 R)$      $(q_1 0 q_1 0 R)$

$(q_0 1 q_1 \mathbf{1} R)$      $(q_1 1 q_0 \mathbf{1} R)$

$(q_0 A \text{Halt } 0)$      $(q_1 A \text{Halt } 1)$

The **Bold** symbol denotes a change to the machine.

---

(b) We could tidy the tape up further by printing the output after the second A (on the right) and then printing a third A after that, before halting. This machine would be specified by:

$$\begin{aligned} (q_0 0 q_0 0 R) & \quad (q_1 0 q_1 0 R) \\ (q_0 1 q_1 1 R) & \quad (q_1 1 q_0 1 R) \\ (q_0 A q_2 A R) & \quad (q_1 A q_3 A R) \\ (q_2 0 q_4 0 R) & \quad (q_3 0 q_4 1 R) \\ (q_4 0 \text{Halt } A) & \end{aligned}$$

---

For states  $q_2, q_3, q_4$  we do not need to specify what happens when they scan 1 because this can never occur. Continuing our example with this new machine we have: Here the tape alphabet is still  $\{0,1,A\}$  but the state alphabet is  $\{q_0, q_1, q_2, q_3, q_4\}$

---

(c) We can represent the Turing Machine in tabular form as follows:

'old' state	'old' symbol	'new' state	'new' symbol	direction
$q_0$	0	$q_0$	0	R
	1	$q_1$	1	R
	A	$q_2$	A	R
$q_1$	0	$q_1$	0	R
	1	$q_0$	1	R
	A	$q_3$	A	R
$q_2$	0	$q_4$	0	R
	1	never occurs		
	A	never occurs		
$q_3$	0	$q_4$	1	R
	1	never occurs		
	A	never occurs		
$q_4$	0	Halt	A	
	1	never occurs		
	A	never occurs		

---

### **1.1.2.2. A parenthesis checker**

Here the input is a sequence of left and right brackets. The output is a 1 or 0 according as the sequence is properly formed or not respectively.

---

Initially the machine scans the left most bracket symbol in state  $q_0$ . The symbol 'A' represents the 'beginning' and 'end' points of the sequence. The machine can be represented in tabular form as follows:

---

'old' state	'old' symbol	'new' state	'new' symbol	direction
q <sub>0</sub>	)	q <sub>1</sub>	X	L
	(	q <sub>0</sub>	(	R
	A	q <sub>2</sub>	A	L
	X	q <sub>0</sub>	X	R
q <sub>1</sub>	)	q <sub>1</sub>	)	L
	(	q <sub>0</sub>	X	R
	A	Halt	0	
	X	q <sub>1</sub>	X	L
q <sub>2</sub>	)	Never occurs		
	(	Halt	0	
	A	Halt	1	
	X	q <sub>2</sub>	X	L

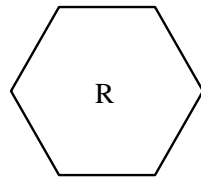
---

---

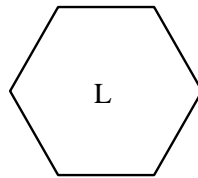
**Note:**  $q_1$  scanning a ")" does not occur because we start at the leftmost bracket symbol. It would be needed if we did not impose this condition. In the above Turing Machines (and in most Turing Machines) we can associate a single direction with each state. For 1.2.2.2  $q_0$  is associated with R,  $q_1$  with L and  $q_2$  with L.

---

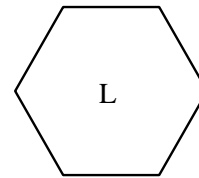
This can be expressed diagrammatically by:



$q_0$



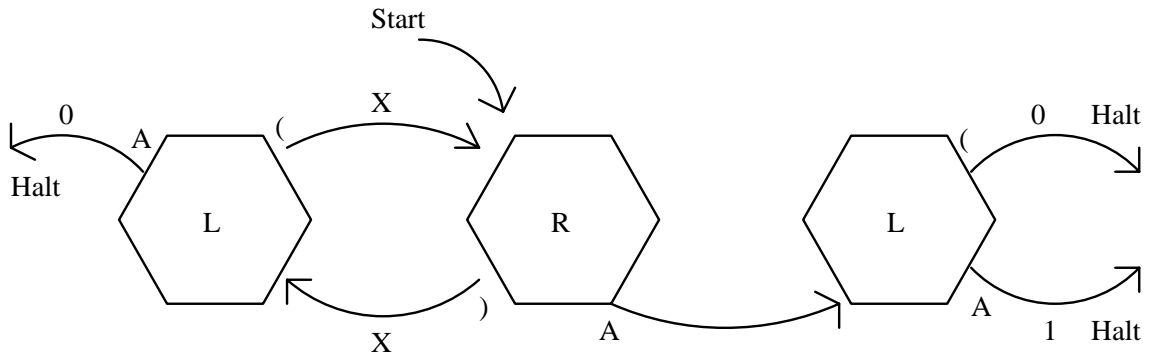
$q_1$



$q_2$

---

Now we can remove the labels  $q_0, q_1, q_2$  and link the states together by arrows which indicate the interactions between the states as follows:



---

**Remark:**

$q_0$  is the middle state,  $q_1$ , is on the left and  $q_2$  on the right.

---

**Exercise:**

(a) Alter this machine to leave the tape in the following format:

A "0 or 1" A

(b) Construct a flow diagram for the parity checker (1.2.2.1).

---

We now have 3 ways of representing Turing Machines:

(1) A set of quintuples

(2) Tabular form

(3) Diagrammatic form.

When constructing Turing Machines it is usually easier to do so using the diagrammatic form.

---

### 1.1.2.3. A Unary Multiplier

The input of the machine is to be two numbers  $m, n$  written (in unary notation) on the tape as follows:

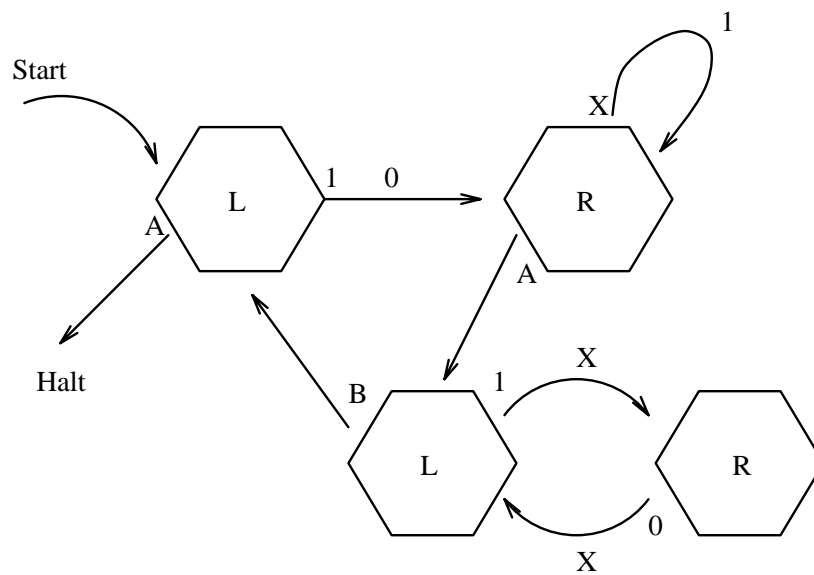
$$\dots 0 A \overbrace{1 \dots 1}^m B \overbrace{1 \dots 1}^n A 0 \dots$$

$\wedge$

The 'B' is the symbol initially scanned.

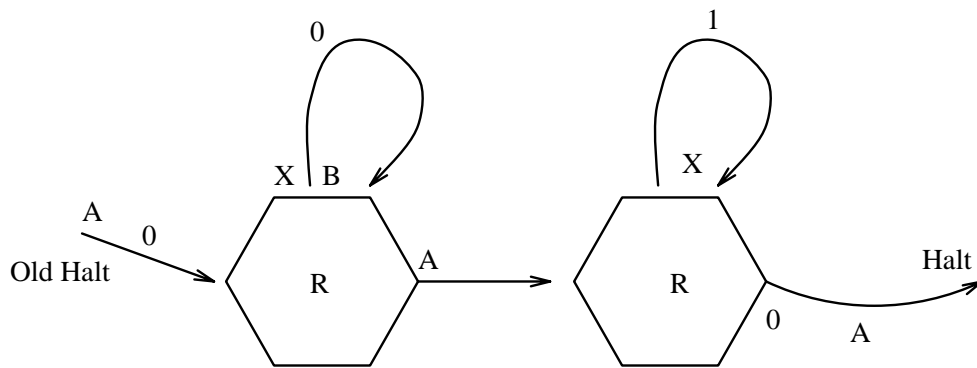
---

The following Turing Machine will calculate  $m.n$  and halt:



---

We could now hand over to the following  
'Clean up' Turing machine:



---

This would leave us with

..0A11111A0...

as the final configuration.

---

### 1.1.2.4. Example

Consider the Turing machine with the following tables of quintuples:

	0	Halt			0	B	1
	1	1	X	0	1	1	0
0	A	0	A	1	0	B	1
	B	0	B	1	1	A	0

---

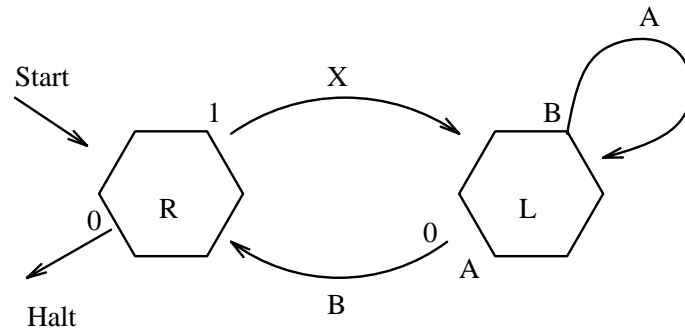
The initial configuration is

$$\begin{array}{c} \dots 001 \dots 100 \\ \uparrow \underbrace{\hspace{1.5cm}}_n \\ 0 \end{array}$$

Convert these tables into a state diagram  
and consider the case when  $n=3$ .

---

# Solution



---

**Note:**

If we consider  $B \leftrightarrow 1$ ,  $A \leftrightarrow 0$  this Turing Machine is designed to turn a unary number into a binary number.

---

### 1.1.2.5. Example

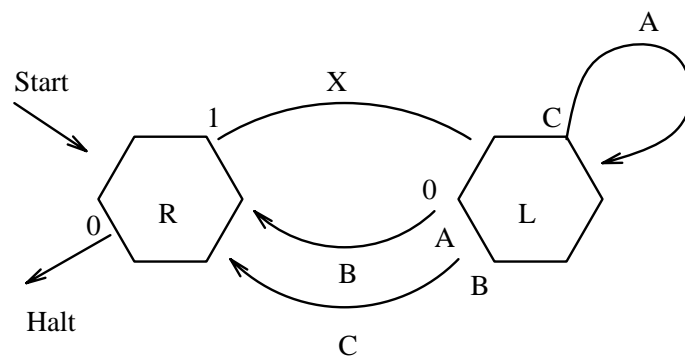
(a) Convert the state diagram in Example 4 into a state diagram for a Turing Machine to record a unary number in base 3 using  $A \leftrightarrow 0$ ,  $B \leftrightarrow 1$ ,  $C \leftrightarrow 2$ .

(b) Consider the case when  $n=3$

---

## Solution:

(a)



---

### **1.1.3. Turing Machines with just two tape symbols**

Every Computation that can be carried out by a T.M. with a finite alphabet can be (equivalently) carried out by a T.M. with just two tape symbols 0,1.

---

This can be carried out as follows:

(i) Suppose a T.M.  $T$  has  $k$  tape symbols and that  $k$  has  $n$  binary digits.

(ii) Assign a distinct  $n$  digit binary number to each tape symbol of  $T$ .

(iii) Replace  $T$  by a new machine  $T^*$  that will treat its tape as blocks of  $n$  cells grouped together.

---

Suppose  $T$  has tape symbols  $0, 1, A, B$  ( $k=4$   
hence  $n=2$ )

We set up the correspondence as follows:

$$\begin{array}{cccc} \text{C} & 1 & \text{A} & \text{B} \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 00 & 01 & 10 & 11 \end{array}$$

These are the binary digits for the operation  
of  $T^*$ .

---

Suppose  $T$  has initial configuration.

...01AB0....

↑

$q_0$

---

Then  $T^*$  has the following initial configuration:

...00|01|10|11|00....

↑

\*

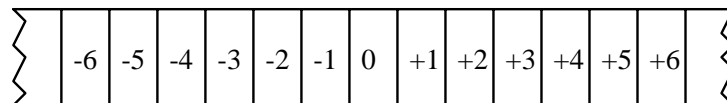
$q_0$

The technicalities of constructing  $T^*$  from  $T$  are left as an exercise.

---

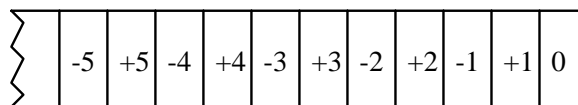
## 1.1.4. Single and Doubly infinite tapes

The tape for a T.M.,  $T$  is doubly infinite  $\therefore$  the cells of the tape can be numbered by integers as follows:



---

Then it is possible to construct a T.M,  $T^*$   
whose tape is;



and which carries out a corresponding  
operation to  $T$ . Again the technicalities of  
constructing  $T^*$  are left as an exercise.

---

We say that  $T^*$  has a single infinity on the left.

---

## Remark

We can modify the Church-Turing thesis to obtain the following statement:

"An effective procedure is one that can be realised by a T.M. whose tape alphabet is simply  $\{0,1\}$  and whose tape is singly infinite on the left".

---

## **1.1.5. Turing Computable Functions**

A T.M. can be thought of as a function which processes the input data and produces an output if the T.M. stops after carrying out a computation.

---

### 1.1.5.1. Definition:

A function  $f(x_1, \dots, x_n)$  is said to be **TURING COMPUTABLE** if there exists a T.M.,  $T$  such that when  $(x_1, \dots, x_n)$  is put onto  $T$ 's tape in some standard form then  $T$  computes  $f(x_1, \dots, x_n)$  on its tape and halts.

---

### 1.1.5.2. Example

Show that the function  $U_3^2(x_1, x_2, x_3) = x_2$ , the projection function is Turing computable.

---

## Solution

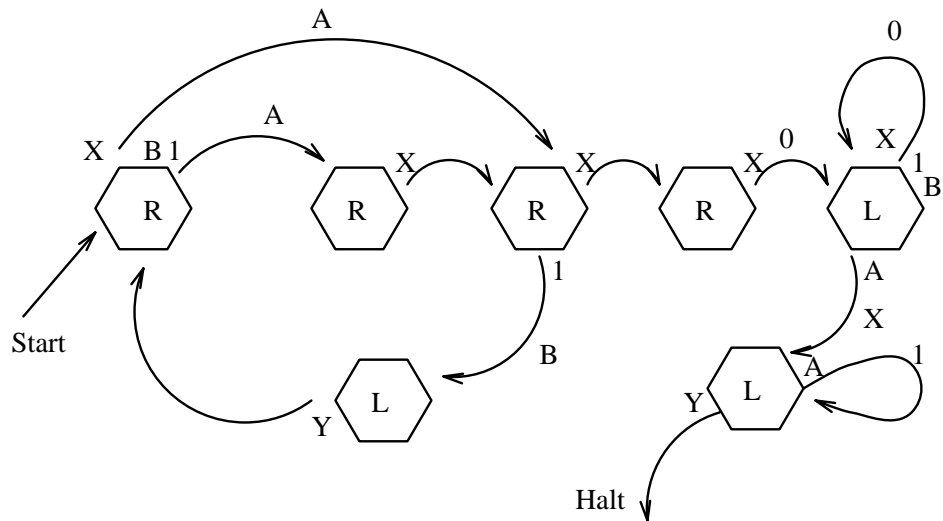
We must ‘come up’ with a T.M. that does the same thing as the given function. Let the ‘standard form’ of  $(x_1, x_2, x_3)$  on the tape be

$$\dots 0 \overset{\wedge}{Y} 1 \dots 1 \overset{\underbrace{\hspace{1cm}}}{X} 1 \dots 1 \overset{\underbrace{\hspace{1cm}}}{X} 1 \dots 1 \overset{\underbrace{\hspace{1cm}}}{X} 0 \dots$$

Start       $x_1$                    $x_2$                    $x_3$

---

The State diagram for the T.M. is



---

### **1.1.6. A Universal Turing Machine**

A Universal Turing Machine is one that is capable of ‘mimicking’ the action of any other Turing Machine.

---

We will consider a fixed Turing Machine  $U$  with the property that, for each and every Turing Machine  $T_f$  (that computes  $f(x_1, \dots, x_n)$  say) there is a string of symbols (the quintuples of  $T_f$ ) which can be written onto  $U$ 's tape together with  $x_1, \dots, x_n$  (in some standard notation) and after executing  $U$ ,  $f(x_1, \dots, x_n)$  will be written on its tape.

---

Suppose we have a Turing Machine  $T$  which we can assume has alphabet  $\{0,1\}$  and has a tape with single infinity on the left.

---

The tape description of  $T$  (written  $d_T$ ) is the string of all quintuples of  $T$  separated by  $X$ 's with a terminator  $Y$ , but not including any quintuples instructing the machine to halt.

---

For example, suppose we have  $T$  with the quintuples:

$(q_0 1 q_1 0 L)$

$(q_1 1 q_2 0 L)$

$(q_2 0 q_2 0 L)$

Using the coding  $q_0 \leftrightarrow 00$ ,  $q_1 \leftrightarrow 01$ ,  $q_2 \leftrightarrow 10$ ,  $L \leftrightarrow 0$ ,  $R \leftrightarrow 1$ ,  $d_T$  would be

X0010100X0111000X1001000Y

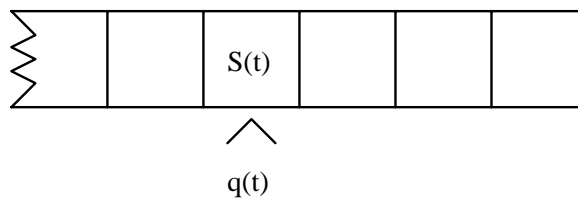
---

## Minsky's Universal Turing Machine

$U$

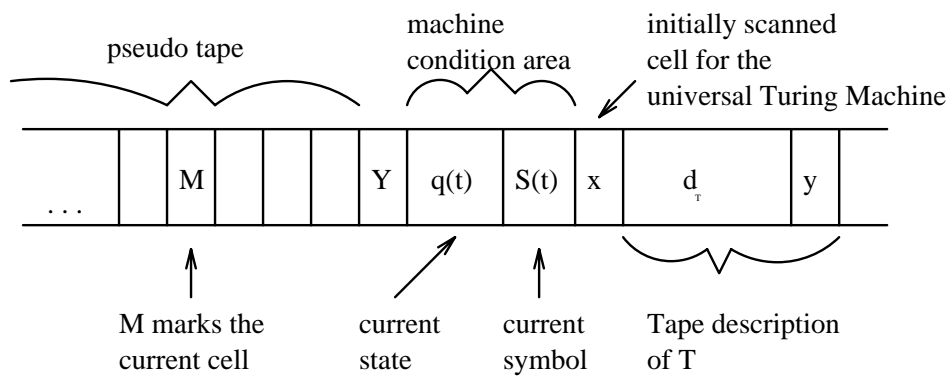
Suppose we have  $T$  as just described. After

$t$  cycles  $T$ 's tape will be of the form:



---

The corresponding tape for Minsky's Universal Turing Machine will be of the form:



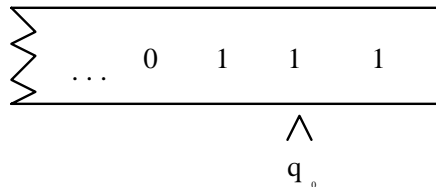
---

The Universal Turing Machine will read  $d_T$  and mimic the action of  $T$  on the pseudo tape.

The Universal Turing Machine can be thought of in terms of modern general purpose computers. These interpret programs (corresponding to  $d_T$ ) and carry out the instructions of the program (corresponding to the operation of  $T$ ).

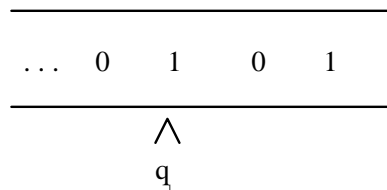
---

Suppose the Turing Machine  $T$  presented on page 24 is started on (singly infinite) tape in state  $q_0$  with the following initial configuration:



---

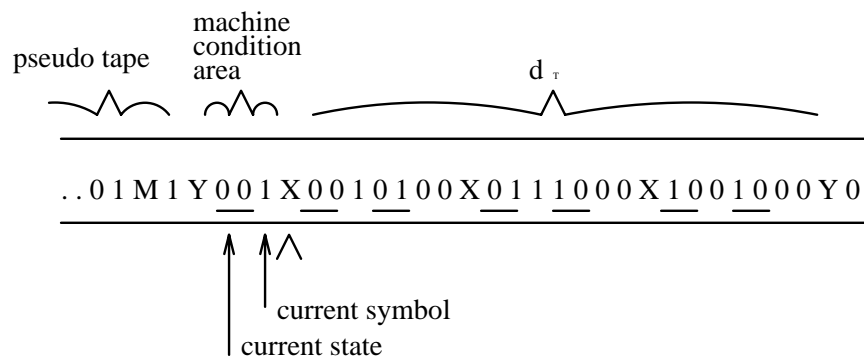
We will convert this information to the tape for Minsky's Universal Turing Machine and work through 1 cycle of the machine to obtain



Which is the configuration after 1 step of  $T$ .

---

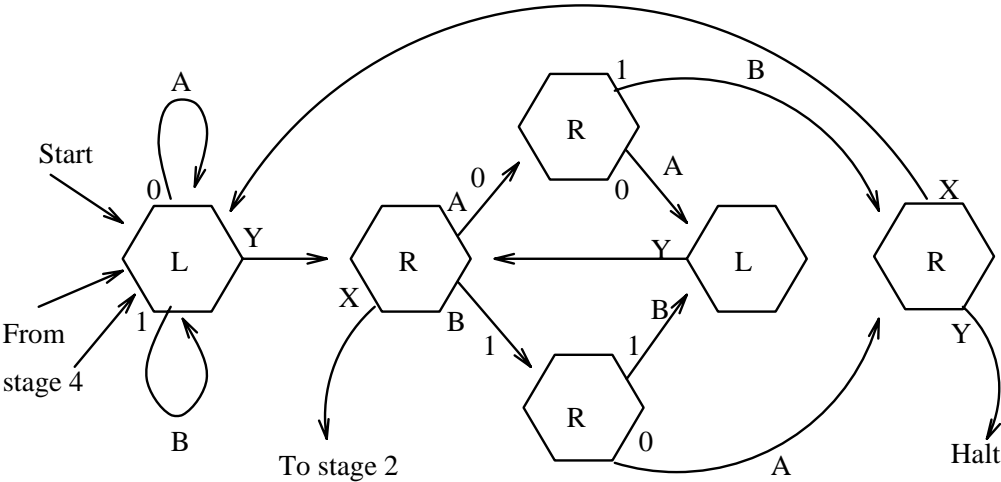
The tape of Minsky's Universal Turing Machine will be as follows:



We will now work through the 4 stages in the operation of 1 cycle of the Universal Turing Machine.

---

# Stage 1: The Location Machine



---

This machine searches to the right (changing 0's to A's and 1's to B's respectively) to find a state-symbol pair that matches the 'current' state-symbol pair stored in the machine condition area.

---

Upon finding a match it changes the appropriate 0's and 1's to A's and B's respectively (in order to 'remember' the correct quintuple). The machine then locates the left most X and moves on to Stage 2.

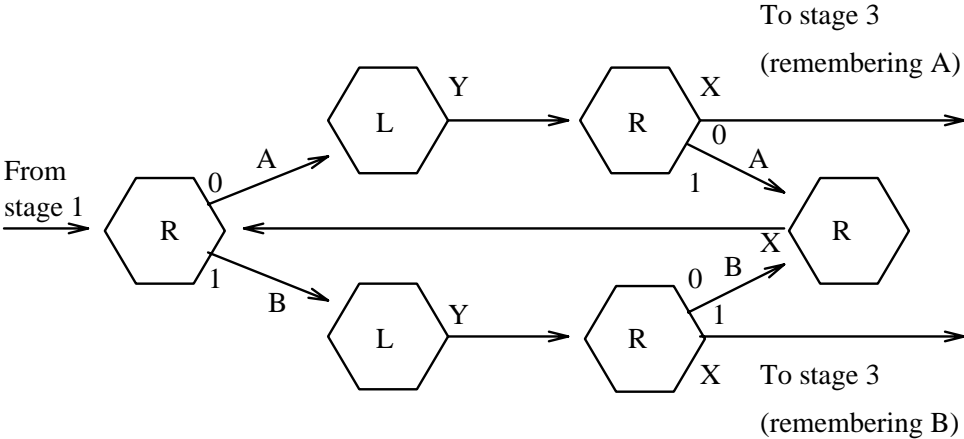
The absence of a matching state-symbol pair indicates that  $T$  will halt. Correspondingly  $U$  will halt.

After Stage 1 the tape configuration is:

$$\begin{array}{c} \text{..01M1Y001X\underline{A}A\underline{B}0\underline{1}00X\underline{0}1\underline{1}1\underline{0}00X\underline{1}00\underline{1}000Y0..} \\ \hline \wedge \end{array}$$

---

# Stage 2: The Copying Machine



---

This machine moves to the right until it locates 0's or 1's. These indicate the 'new' state-symbol pair and the direction in which to move. The machine copies the new state-symbol pair into the machine condition area of the tape. It then moves on to Stage 3 remembering an A or B according to whether the direction was left (0) or right (1) respectively.

The configuration after Stage 2 is:

---

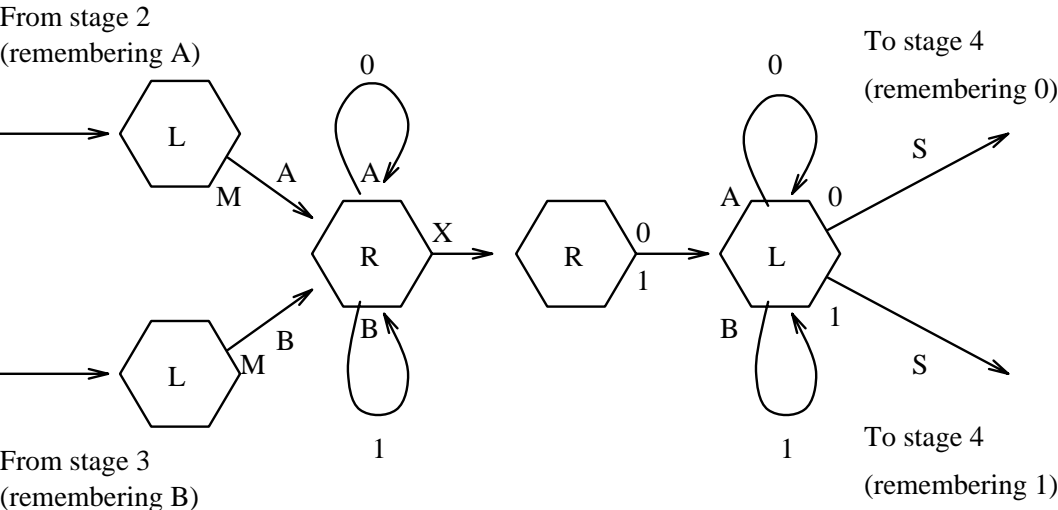
..01M1YABAXAABABAAX0111000X1001000Y0..

---

^

---

# Stage 3: The Restore Machine



---

This machine moves left until it locates  $M$  on the pseudo tape (which stands for the currently scanned cell of  $T$ 's tape). It then temporarily replaces  $M$  by the remembered  $A$  or  $B$  (that are indicators of the direction in which to move). The machine then moves right, restoring the  $A$ 's and  $B$ 's in the machine condition area to  $0$ 's and  $1$ 's until it locates the first  $X$ .

---

It then moves right looking for the first 0 or 1 and, having found it, moves left restoring the A's and B's in  $d_T$  to 0's and 1's. Finally it reads and remembers the new tape symbol to be printed in  $M$ 's old location. For the moment it replaces this by an  $S$  and then moves on to Stage 4.

---

The configuration after Stage 3 is:

---

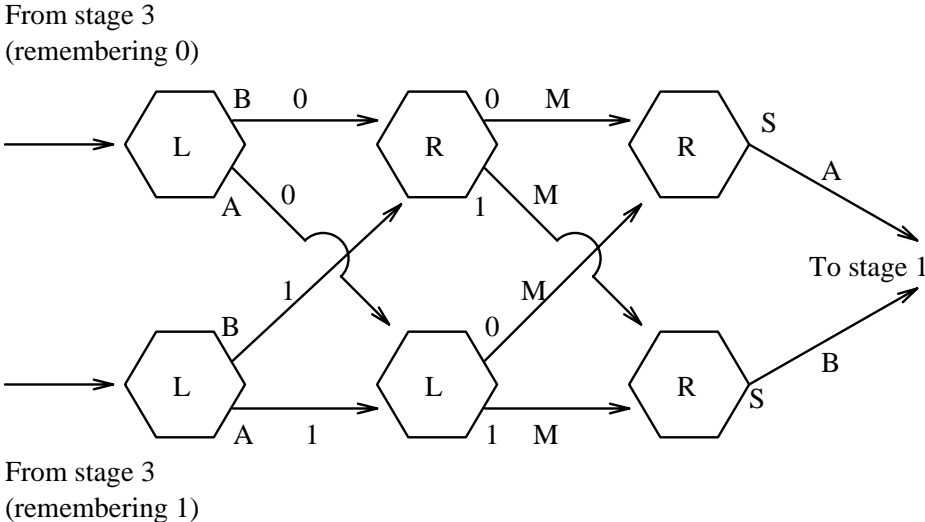
...01A1Y01SX0010100X0111000X1001000Y0...

---

^

---

# Stage 4: The Operating Machine



---

This is the final stage of one complete cycle of  $U$ . The machine moves left until it locates A or B on the tape in  $M$ 's old location. It then prints the new symbol (the remembered 0 or 1 from Stage 3) in this cell and moves left or right according as the cell remembers A or B respectively.

---

It then prints  $M$  in its new location and remembers the symbol in this cell. It now moves right until it finds the 'S' and replaces it by an A or B corresponding to the remembered symbol. Finally it hands over to Stage 1 to continue with the next cycle.

The configuration after Stage 4 is:

---

..0M01Y01BX0010100X0111000X1001000Y0..

---

^

---

Notice that

Pseudo tape  
⌋  
.. 0 M 0 1 Y 0 1 B X  
          ⌋  
          machine  
          condition area

corresponds to the configuration on  $T$ 's tape  
after 1 step of the operation of  $T$  i.e.

...	0	1	0	1
-----	---	---	---	---

^  
 $q_1$

---

## 1.1.7. Primitive Recursive Functions:

### 1.1.7.1. The Initial Functions:

The following are called initial functions:

- (a) The zero function:  $Z(x) = 0$  for all  $x$   
(where  $x$  is a natural number).
- (b) The successor function:  $S(x) = x+1$  for  
all  $x$  (where  $x$  is a natural number).

---

(c) The projection functions:

$U_i^n(x_1, \dots, x_n) = x_i$  for natural numbers

$x_1, \dots, x_n$ .

---

## 1.1.7.2. Rules for Obtaining New Functions:

### (d) Substitution:

Given functions  $g(x_1, \dots, x_n)$  and  $h_1(x_{(1,1)}, \dots, x_{(1,m_1)}), \dots, h_n(x_{(n,1)}, \dots, x_{(n,m_n)})$  we can define a new function  $f$  by substitution as:

$$f = g(h_1(x_{(1,1)}, \dots, x_{(1,m_1)}), \dots, h_n(x_{(n,1)}, \dots, x_{(n,m_n)}))$$

For example let  $g(z, w) = z + w$ ,

$h_1(x, y) = x \cdot y$  and  $h_2(x) = x^2$  then

$$f = x \cdot y + x^2.$$

---

## (e) Recursion

Given functions  $g, h$  of  $n, n+2$  variables respectively we can define a new function  $f$  by recursion in the variable  $y$  as:

$$\begin{aligned}f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n) \\f(x_1, \dots, x_n, y+1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y))\end{aligned}$$

---

### **1.1.7.3. Definition**

A function is called primitive recursive (p.r.) if it can be obtained from the initial functions (a), (b), (c) by a finite number of applications of rules (d), (e).

The set of all such functions is called the set of primitive recursive functions.

---

## 1.1.8. The Development of Primitive Recursive Functions

### 1.1.8.1. The Constant Function:

$$C_n(x) = n \text{ for all } x \text{ (for } n \geq 0).$$

We show this function is primitive recursive by induction on  $n$ .

**Base Case:**  $n = 0$

$$C_0(x) = Z(x)$$

This is p.r. because  $Z$  is an initial function.

---

## Induction Step

We assume for  $k \geq 0$  that  $C_k(x)$  is p.r. Then

$$C_{k+1}(x) = k+1 = S(C_k(x))$$

$\therefore C_{k+1}$  is p.r. by substitution since  $S$  and  $C_k$  are p.r.

The result follows for all  $n$  by induction.

---

### 1.1.8.2. $x+y$

We can define this function recursively in  $y$ , using  $u_1^1$  and  $s$  as:

$$x+0 = x \quad (=u_1^1(x))$$

$$x+(y+1) = (x+y)+1 \quad (=s(x+y))$$

Since  $u_1^1$  and  $s$  are p.r.  $x+y$  is p.r.

---

### 1.1.8.3. $x.y$

We can define  $x.y$  recursively in  $y$ , using

$z, +, u_1^1$  as:

$$x.0 = 0 \quad (= z(x))$$

$$x.(y+1) = x.y+x \quad (= x.y+u_1^1(x))$$

Since  $z, +, u_1^1$  are p.r.  $x.y$  is p.r.

---

#### 1.1.8.4. $x^y$

We have

$$\begin{aligned}x^0 &= 1 \\x^{y+1} &= x^y \cdot x\end{aligned}$$

This is a recursive definition of  $x^y$  which is p.r. since  $1, \dots, x$  are p.r.

---

### 1.1.8.5. The Predecessor function

$\delta(x)$

The function  $\delta(x)$  is defined as:

$$\delta(x) = \begin{cases} x-1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

This can be defined recursively as using p.r.

functions as:

$$\delta(0) = 0$$

$$\delta(x+1) = x(+0.\delta(x))$$

$\therefore \delta(x)$  is p.r.

---

### 1.1.8.6. $x \dot{-} y$

The function  $x \dot{-} y$  is defined as:

$$x \dot{-} y = \begin{cases} x-y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

This can be defined recursively as:

$$x \dot{-} 0 = x$$

$$x \dot{-} (y+1) = \delta(x \dot{-} y)$$

Since  $x$ ,  $\delta$  and p.r. so is  $x \dot{-} y$ .

---

### 1.1.8.7. $|x-y|$

The function  $|x-y|$  is defined as:

$$|x-y| = \begin{cases} x-y & \text{if } x \geq y \\ y-x & \text{if } x < y \end{cases}$$

Now  $|x-y|$  can be defined by substitution as  $(x \dot{-} y) + (y \dot{-} x)$ . Since  $+$ ,  $\dot{-}$  are p.r. So is  $|x-y|$ .

---

### 1.1.8.8. $sg(x)$

The function  $sg(x)$  is defined as

$$sg(x) = \begin{cases} 0 & \text{if } x=0 \\ 1 & \text{if } x>0 \end{cases}$$

This can be defined recursively as

$$\begin{aligned} sg(0) &= 0 \\ sg(x+1) &= 1 \end{aligned}$$

Since 0, 1 are both p.r. so is  $sg$ .

---

### 1.1.8.9. $\overline{sg}(x)$

This is defined as:

$$\overline{sg}(x) = \begin{cases} 1 & \text{if } x=0 \\ 0 & \text{if } x > 0 \end{cases}$$

It can be defined recursively as

$$\begin{aligned} \overline{sg}(0) &= 1 \\ \overline{sg}(x+1) &= 0 \end{aligned}$$

Since 1,0 are both p.r. so is  $\overline{sg}$ .

---

### 1.1.8.10. $x!$

We can define this recursively as:

$$0! = 1$$

$$(x+1)! = s(x) \cdot x!$$

Since  $1, \cdot, s$  are all p.r. so is  $x!$

---

### 1.1.8.11. $\min(x,y)$

This is defined as:

$$\min(x,y) = \begin{cases} x & \text{if } x \leq y \\ y & \text{if } x > y \end{cases}$$

We can define this using substitution as

$x \dot{-} (x \dot{-} y)$ . Since  $\dot{-}$  is p.r. so is min.

---

### 1.1.8.12. $\max(x, y)$

This is defined as

$$\max(x, y) = \begin{cases} y & \text{if } x \leq y \\ x & \text{if } x > y \end{cases}$$

Again using substitution we see that

$\max(x, y) = y + (x \dot{-} y)$ . Since  $+$ ,  $\dot{-}$  are p.r.

so is  $\max$ .

---

### 1.1.8.13. The remainder function

$rm(x, y)$

$rm(x, y)$  is defined as the remainder when  $y$  is divided by  $x$ .

Given  $x, y$  there exists numbers  $q$  and  $r$  such that  $y = q \cdot x + r$  (where  $r < x$ ).  $q, r$  are the quotient and remainder respectively.

---

Now

$$y+1 = \begin{cases} q \cdot x + (r+1) & \text{if } r+1 < x \\ (q+1) \cdot x & \text{if } r+1 = x \end{cases}$$

These properties are used in the recursive definition of  $rm(x,y)$  (and of **1.3.2.14**  $qt(x,y)$ ).

We have

$$rm(x, 0) = 0$$
$$rm(x, y+1) = (rm(x, y)+1) \cdot sg(x \dot{-} (rm(x, y)+1))$$

Since  $0, +, 1, \cdot, sg, \dot{-}$  are p.r. so is  $rm(x,y)$ .

---

### 1.1.8.14. The quotient function

$qt(x, y)$

$qt(x, y)$  is the quotient when  $y$  is divided by  $x$ . This can be defined recursively as:

$$qt(x, 0) = 0$$

$$qt(x, y+1) = qt(x, y) + \overline{sg}(x \dot{-} (rm(x, y)+1))$$

Since  $0, +, \overline{sg}, \dot{-}, rm, 1$  are p.r. so is  $qt$ .

---

## 1.1.9. Bounded Sums and Products

### 1.1.9.1.

$$\sum_{y < z} f(x, y) = \begin{cases} 0 & \text{if } z = 0 \\ f(x, 0) + \dots + f(x, z-1) & \text{if } z > 0 \end{cases}$$

We can define this by recursion in  $z$  as:

(notice that  $\sum_{y < z} f(x, y)$  is a function in  $x$

and  $z$ )

$$\begin{aligned} \sum_{y < 0} f(x, y) &= 0 \\ \sum_{y < z+1} f(x, y) &= f(x, 0) + \dots + f(x, z-1) + f(x, z) \\ &= \sum_{y < z} f(x, y) + f(x, z) \end{aligned}$$

---

$\therefore$  **if**  $f(x,y)$  is p.r. then  $\sum_{y < z} f(x,y)$  is p.r.

---

### 1.1.9.2.

$$\sum_{y \leq z} f(x, y) = \begin{cases} f(x, 0) & \text{if } z = 0 \\ f(x, 0) + \dots + f(x, z) & \text{if } z > 0 \end{cases}$$

### Exercise:

Show that if  $f$  is p.r. then  $\sum_{y \leq z} f(x, y)$  is p.r.

### 1.1.9.3.

$$\prod_{y < z} f(x, y) = \begin{cases} 1 & \text{if } z = 0 \\ f(x, 0) \cdot \dots \cdot f(x, z-1) & \text{if } z > 0 \end{cases}$$

---

## Exercise:

Show that if  $f$  is p.r. then  $\prod_{y < z} f(x, y)$  is p.r.

---

### 1.1.9.4.

$$\prod_{y \leq z} f(x, y) = \begin{cases} f(x, 0) & \text{if } z = 0 \\ f(x, 0) \cdot \dots \cdot f(x, z) & \text{if } z > 0 \end{cases}$$

We can define this by recursion in  $z$  as:

$$\begin{aligned} \prod_{y \leq 0} f(x, y) &= f(x, 0) \\ \prod_{y \leq z+1} f(x, y) &= f(x, 0) \cdot \dots \cdot f(x, z) \cdot f(x, z+1) \\ &= \prod_{y \leq z} f(x, y) \cdot f(x, z+1) \end{aligned}$$

$\therefore$  **if**  $f(x, y)$  is p.r. then  $\prod_{y \leq z} f(x, y)$  is p.r.

---

### 1.1.9.5.

$$\sum_{u < y \leq v} f(x, y) = f(x, u+1) + \cdots + f(x, v)$$

This can be defined by substitution as

$$\sum_{y \leq v} f(x, y) \doteq \sum_{y \leq u} f(x, y)$$

$\therefore$  if  $f$  is p.r. then  $\sum_{u < y \leq v} f(x, y)$  is p.r.

---

### 1.1.9.6.

$$\prod_{u \leq y < v} f(x, y) = f(x, u) \cdot \dots \cdot f(x, v-1)$$

This can be defined by substitution as:

$$qt\left(\prod_{y < u} f(x, y), \prod_{y < v} f(x, y)\right)$$

$\therefore$  if  $f$  is p.r. so is  $\prod_{u \leq y < v} f(x, y)$ .

---

### 1.1.9.7. The number of factors of $x$ (denoted by $D(x)$ )

$$D(x) = \begin{cases} 1 & \text{if } x=0 \\ \text{number of factors of } x & \text{if } x > 0 \end{cases}$$

To show  $D(x)$  is p.r. we have to consider how to count the factors of  $x$ . If  $z \leq x$  it is a factor of  $x$  if

$$rm(z, x) = 0$$

$z$  is not a factor of  $x$  if

$$rm(z, x) \neq 0$$

---

To count the factors we consider

$$\overline{sg}(rm(z, x)) = \begin{cases} 1 & \text{if } z \text{ is a factor of } x \\ 0 & \text{if } z \text{ is not a factor of } x \end{cases}$$

$$\therefore D(x) = \sum_{0 < z \leq x} \overline{sg}(rm(z, x))$$

$\therefore$  Since  $\overline{sg}$ ,  $rm$ ,  $\sum$  are p.r. so is  $D(x)$  (by substitution).

---

## 1.1.10. Primitive Recursive Relations

### 1.1.10.1. Definitions:

Given any number-theoretic relation  $R$  there is an associated function, known as the characteristic function of  $R$  (written  $C_R$ ), which is defined by:

$$C_R = \begin{cases} 0 & \text{iff } R \text{ is true} \\ 1 & \text{iff } R \text{ is false} \end{cases}$$

---

A relation  $R$  is called primitive recursive iff its characteristic function  $C_R$  is a primitive recursive function.

---

## Examples

### 1.1.10.2. $x=y$

The characteristic function of "=" is:

$$C_{=}(x,y) = \begin{cases} 0 & \text{if } x=y \\ 1 & \text{if } x \neq y \end{cases}$$

$$\therefore C_{=}(x,y) = \text{sg}(|x-y|)$$

$\therefore C_{=}$  is p.r. by substitution so "=" is a p.r. relation.

---

### 1.1.10.3. $x < y$

The characteristic function of  $<$  is

$$C_{<}(x,y) = \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \geq y \end{cases}$$

$\therefore C_{<}(x,y) = \overline{sg}(y \dot{-} x)$  and so is a p.r.

function by substitution.

$\therefore x < y$  is a p.r. relation.

---

**1.1.10.4.  $x$  is a factor of  $y$  (denoted by  $x|y$ )**

$$C_{|}(x,y) = \begin{cases} 0 & \text{if } x|y \\ 1 & \text{if } x \not|y \end{cases}$$

i.e.

$$C_{|}(x,y) = \begin{cases} 0 & \text{if } rm(x,y)=0 \\ 1 & \text{if } rm(x,y)\neq 0 \end{cases}$$

$\therefore C_{|}(x,y) = sg(rm(x,y))$  which is a p.r.

function by substitution.

$\therefore x|y$  is a p.r. relation.

---

**1.1.10.5.  $x$  is a prime number  
(denoted by  $Pr(x)$ )**

$$C_{Pr(x)} = \begin{cases} 0 & \text{if } x \text{ is prime} \\ 1 & \text{if } x \text{ is not prime} \end{cases}$$

$$\text{i.e. } C_{Pr(x)} = \begin{cases} 0 & \text{if } x \text{ has exactly 2 factors} \\ 1 & \text{if } x \text{ doesn't have exactly 2 factors} \end{cases}$$

---

i.e.

$$C_{Pr}(x) = \begin{cases} 0 & \text{if } D(x)=2 \\ 1 & \text{if } D(x)\neq 2 \end{cases}$$

$\therefore C_{Pr}(x) = sg |D(x)-2|$  which is a p.r.

function by substitution.

$\therefore Pr(x)$  is a p.r. relation.

---

### 1.1.10.6. $\sim R$

$$\begin{aligned} C_{\sim R} &= \begin{cases} 0 & \text{if } R \text{ is F} \\ 1 & \text{if } R \text{ is T} \end{cases} \\ &= \begin{cases} 0 & \text{if } C_R=1 \\ 1 & \text{if } C_R=0 \end{cases} \end{aligned}$$

$\therefore C_{\sim R} = \overline{sg}(C_R)$  which is a primitive recursive function if  $C_R$  is.

$\therefore$  If  $R$  is a p.r. relation then  $\sim R$  is a p.r. relation.

---

### 1.1.10.7. $R \wedge S$

Given relations  $R, S$ :

$$\begin{aligned} C_{R \wedge S} &= \begin{cases} 0 & \text{if } R \text{ is T \& } S \text{ is T} \\ 1 & \text{if } R \text{ is F or } S \text{ is F} \end{cases} \\ &= \begin{cases} 0 & \text{if } C_R=0 \text{ \& } C_S=0 \\ 1 & \text{if } C_R=1 \text{ or } C_S=1 \end{cases} \end{aligned}$$

$\therefore C_{R \wedge S} = sg(C_R + C_S)$  which is p.r. if  $C_R$  and  $C_S$  are.

$\therefore R \wedge S$  is a p.r. relation if  $R, S$  are p.r. relations.

---

### 1.1.10.8. $R \vee S$

$$C_{R \vee S} = \begin{cases} 0 & \text{if } C_R=0 \text{ or } C_S=0 \\ 1 & \text{if } C_R=1 \text{ \& } C_S=1 \end{cases}$$

$\therefore C_{R \vee S} = C_R \cdot C_S$  which is p.r. if  $C_R, C_S$  are.

$\therefore$  If  $R, S$  are p.r. relations then  $R \vee S$  is a p.r. relation.

---

## 1.1.11. Definition by Cases

### 1.1.11.1. Definition:

Let a function  $f$  be defined as follows:

$$f = \left\{ \begin{array}{l} g_1 \text{ if } R_1 \text{ is true for all values of variables occurring} \\ \cdot \quad \cdot \\ \cdot \quad \cdot \\ \cdot \quad \cdot \\ \cdot \quad \cdot \\ g_n \text{ if } R_n \text{ is true for all values of variables occurring} \end{array} \right.$$

---

where  $g_1, \dots, g_n$  are functions and  $R_1, \dots, R_n$  are disjoint relations. This defines the function properly and is called a definition by cases of  $f$ .

---

### 1.1.11.2. Theorem

If  $g_1, \dots, g_n$  are p.r. functions and  $R_1, \dots, R_n$  are p.r. relations then  $f$  is p.r.

#### **Proof:**

Let  $C_{R_i}$  be the characteristic function of  $R_i$

(for  $1 \leq i \leq n$ ) then

$$C_{R_i} = \begin{cases} 0 & \text{when } R_i \text{ is true} \\ 1 & \text{when } R_i \text{ is false} \end{cases}$$

---

$\therefore$  we can define  $f$  as

$$f = g_1 \cdot \overline{sg}(C_{R_1}) + \cdots + g_n \cdot \overline{sg}(C_{R_n})$$

$\therefore f$  is p.r. by substitution since all the functions used are p.r.

---

### 1.1.11.3. Example

$$\text{Let } f(x) = \begin{cases} x^2 & \text{if } x \text{ is even} \\ x+1 & \text{if } x \text{ is odd} \end{cases}$$

Show that  $f$  is p.r.

### **Solution:**

We know that  $x^2$ ,  $x+1$  are p.r. functions.

---

It remains to show that  $E(x)$  ( $x$  is even) and  $O(x)$  ( $x$  is odd) are p.r. relations:

$$C_E(x) = \begin{cases} 0 & \text{if } 2 \mid x \\ 1 & \text{if } 2 \nmid x \end{cases}$$

$\therefore C_E(x) = rm(2, x)$  which is p.r.

---


$$C_0(x) = \begin{cases} 0 & \text{if } 2 \nmid x \\ 1 & \text{if } 2 \mid x \end{cases}$$

$\therefore C_0(x) = \overline{sg}(rm(2,x))$  which is p.r.

$\therefore$  By Theorem **1.3.5.2**  $f$  is p.r.

$$f(x) \stackrel{i.e.}{=} x^2 \cdot \overline{sg}(rm(2,x)) + (x+1) \cdot \overline{sg}(\overline{sg}(rm(2,x)))$$

In this case we can simplify it to

$$f(x) = s^2 \cdot \overline{sg}(rm(2,x)) + (x+1) \cdot rm(2,x)$$

(Since  $rm(2,x) = 0$  or  $1$  anyway)

---

**1.1.12. Are all number theoretic functions p.r.?**

The answer to this question is 'no' as can be seen in the following Lemma.

---

### 1.1.12.1. Lemma

Let  $f_0(x), f_1(x), f_2(x), \dots$  be an enumeration of all p.r. functions in the variable  $x$ .

We can define the enumerating function  $g(n, x)$  by  $g(n, x) = f_n(x)$ .  $g(n, x)$  is not a p.r. function.

---

**Proof:**

By way of contradiction suppose  $g(n,x)$  is p.r.

Then  $g(x,x)+1$  must be p.r.

As this is a function in the variable  $x$  it must occur in the above sequence.

---

Suppose  $g(x,x)+1 = f_i(x)$

Let  $x = i$  and we have

$$g(i,i)+1 = f_i(i) = g(i,i),$$

a contradiction.

$\therefore g(n,x)$  is not p.r.

---

## 1.1.13. Recursive Functions

### 1.1.13.1. The unrestricted least operator

A function  $g(x_1, \dots, x_n, y)$  is called regular iff for each sequence  $x_1, \dots, x_n$  there exists a  $y$  such that  $g(x_1, \dots, x_n, y) = 0$ .

---

We looked at the bounded least operator in Exercise 2. We showed this to be a p.r. function. This is not the case if we take away the bound.

---

Let the unbounded least operator,

$$\mu y (g(x_1, \dots, x_n, y) = 0)$$

be the least  $y$  such that  $g(x_1, \dots, x_n, y) = 0$  if

there is one. If no such  $y$  exists we let

$\mu y (g(x_1, \dots, x_n, y) = 0)$  be undefined.

---

## 1.1.13.2. Recursive Functions

A function is said to be totally recursive iff it can be obtained from the initial functions by a finite number of applications of substitution, recursion or of the unbounded least operator on regular functions.

---

It is partially recursive if we do not restrict the unbounded least operator to regular functions. A function is usually said to be recursive if it is either totally or partially recursive.

---

Most of the computable functions arising in practice are primitive recursive. However, all the p.r. functions are total functions and a satisfactory treatment of the computable functions in general has to allow for the consideration of partial functions.

---

Also the p.r. functions do not ‘grow’ (see Section 3) quite as fast as arbitrary computable ones which has to do with the fact that they are defined by less complicated recursions than in the general case. We will discuss the computability of p.r. and recursive functions more fully in **1.5**.

---

## 1.2. Register Machines

### 1.2.1. Definition

A register machine  $R$  has a finite number  $r$  of registers. These can be thought of as positions which, during the running of a machine, contain a natural number.

$R$  has registers  $1, 2, \dots, r$ .

We will denote the contents of the  $n$ th register by  $\bar{n}$ .

---

A register machine program consists of a finite sequence of instructions.

The instructions will be denoted by  $\hat{1}, \hat{2}, \dots, \hat{m}$  (where  $\hat{m}$  is halt) and are of 2 particular types (other than halt).



---

(b)  $(n, p, q)$ : This means that if  $\bar{n} > 0$ , subtract 1 from  $\bar{n}$  and goto instruction  $\hat{p}$ .

Otherwise goto instruction  $\hat{q}$ .

---

**A program to add the contents of  
two registers.**

$\hat{1}$  (2, 2, 3)  
 $\hat{2}$  (1, 1)  
 $\hat{3}$  Halt

(Here  $r=2$  and  $m=3$ ).

---

Suppose we have  $\bar{1}=1$  and  $\bar{2}=3$ .

We will denote this by  $[1,3]$  (i.e. the contents of the registers are given in square brackets).

---

Operating the program on [1,3] gives

$\hat{1}$  [1,3]  
 $\hat{2}$  [1,2]  
 $\hat{1}$  [2,2]  
 $\hat{2}$  [2,1]  
 $\hat{1}$  [3,1]  
 $\hat{2}$  [3,0]  
 $\hat{1}$  [4,0]  
 $\hat{3}$  Halt

---

Given  $[a, b]$  the program terminates with  $[a+b, 0]$ . We can prove this by showing that just before the  $(k+1)$ th execution of instruction  $\hat{1}$  the contents of the registers are  $[a+k, b-k]$ . We do this by induction.

---

**Base Case:**  $k=0$

This is the initial situation:

$$\bar{1} = a + 0, \quad \bar{2} = b - 0$$

---

## Induction Step:

We assume the result for  $k$

i.e.  $\bar{1} = a + k, \bar{2} = b - k$ .

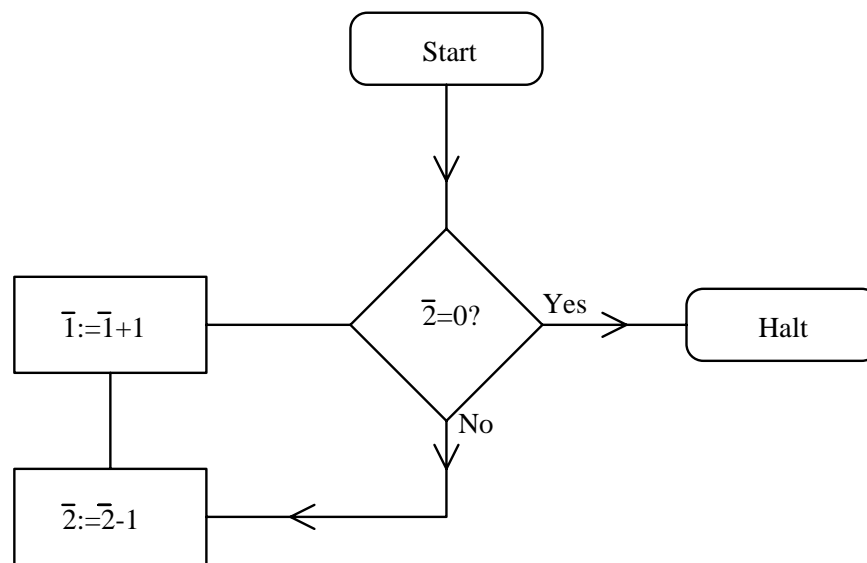
When  $\hat{1}$  is executed for the  $(k+1)$ th time  $\bar{1} = a + k, \bar{2} = b - k - 1$  and we go to instruction  $\hat{2}$ . After  $\hat{2}$  is executed we have  $\bar{1} = a + k + 1, \bar{2} = b - k - 1$  and  $\hat{1}$  is the current instruction.

---

It then follows that  $\bar{2}$  only becomes 0 after  $b$  executions of  $\hat{1}$  and when  $\hat{1}$  is the current instruction for the  $(b+1)$ th time  $\bar{1} = a+b$  and the program halts.

---

We can represent the program in the form of a flowchart:



---

**Consider the following program**

$\hat{1}$  (2, 2, 3)  
 $\hat{2}$  (1, 1, 3)  
 $\hat{3}$  Halt

Where  $r = 2$  and  $m = 3$ .

---

Starting with  $[3,2]$  at instruction  $\hat{1}$  we have

$\hat{1}$   $[3,2]$   
 $\hat{2}$   $[3,1]$   
 $\hat{1}$   $[2,1]$   
 $\hat{2}$   $[2,0]$   
 $\hat{1}$   $[1,0]$   
 $\hat{3}$   $[1,0]$  halt.

---

Starting with  $[3,5]$  at instruction  $\hat{1}$  we have

$\hat{1}$   $[3,5]$   
 $\hat{2}$   $[3,4]$   
 $\hat{1}$   $[2,4]$   
 $\hat{2}$   $[2,3]$   
 $\hat{1}$   $[1,3]$   
 $\hat{2}$   $[1,2]$   
 $\hat{1}$   $[0,2]$   
 $\hat{2}$   $[0,1]$   
 $\hat{3}$   $[0,1]$  halt.

---

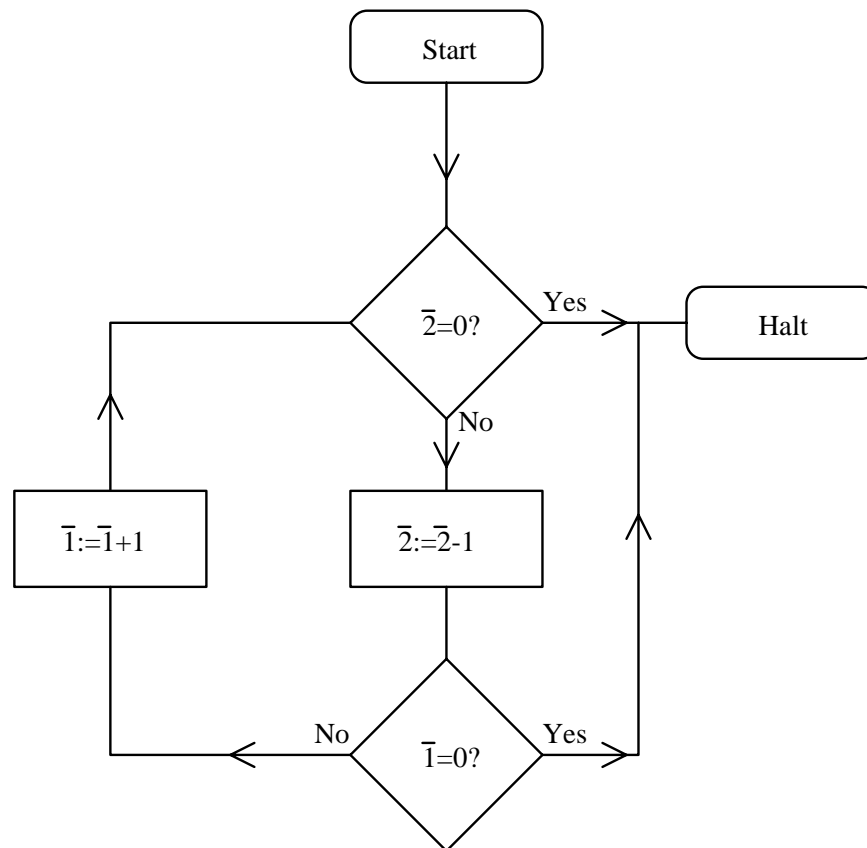
In general this program computes the function  $f$  where

$$f(a,b) = \begin{cases} a-b & \text{if } a \geq b \\ b-a-1 & \text{if } a < b \end{cases}$$

i.e. Starting with  $[a,b]$  we finish with  $[a-b,0]$  if  $a \geq b$  and  $[0,b-a-1]$  if  $a < b$ .

---

The flowchart for this program is



---

We can quite easily modify this program to compute  $a \div b$  and  $(a - b)$ .

For  $a \div b$  we have:

$\hat{1}$  (2, 2, 4)    Exit from loop by  $\hat{1}$  if  
 $\hat{2}$  (1, 1, 3)     $a \geq b$  and by  $\hat{3}$  if  $a < b$   
 $\hat{3}$  (2, 3, 4)    Instruction  $\hat{3}$  deletes the  
 $\hat{4}$  halt.        unwanted contents of register 2

---

For  $|a-b|$  we have:

$\hat{1}$  (2, 2, 5)

$\hat{2}$  (1, 1, 3)

$\hat{3}$  (1,4)      Instructions  $\hat{3}$ ,  $\hat{4}$  transfer

$\hat{4}$  (2, 3, 5)      the contents of register 2 to

$\hat{5}$  Halt.      register 1 and add 1.

---

## Exercise

Construct a register machine to add  $r \times \bar{2}$  to  $\bar{1}$ .

---

## **Codes for Register Machine Programs**

Before we consider coding a register machine program, we will consider a code for an arbitrary sequence of natural numbers.

---

If we have a sequence  $(x_1, \dots, x_n)$  of natural numbers then the code for the sequence (denoted by  $\langle x_1, \dots, x_n \rangle$ ) is given by  $2^{x_1+1} \cdot 3^{x_2+1} \cdot \dots \cdot P_n^{x_n+1}$  where  $P_i$  is the  $i$ th prime number.

---

We add 1 to the powers in order to detect zeros at the end of the sequence. (Strictly speaking it is only necessary to add 1 to the final power, but the above method is more standard.)

---

## Example

Sequence (2,0,1) we have  $\langle 2,0,1 \rangle = 2^3 \cdot 3^1 \cdot 5^2 = 8 \cdot 3 \cdot 25 = 600$ .

Suppose we are given the code first and asked to determine the sequence. We write the number in the form  $2^{y_1} \cdot 3^{y_2} \dots P_m^{y_m}$  and the sequences is  $y_1-1, y_2-1, \dots, y_m-1$ .

---

**Note:**

This coding relies upon ‘the fundamental theorem of arithmetic’ which says that any integer  $\geq 2$  can be uniquely written as a product of powers of distinct primes.

---

## Example

Given the number 150 the prime factorisation is  $2^1 \cdot 3^1 \cdot 5^2$ , therefore, the sequence is (0,0,1).

Given a particular number  $n \geq 2$  we need to be able to decide what it codes.

---

## Seq( $n$ ):

$n$  codes a sequence. We write  $n$  as  $2^{x_1} \cdot 3^{x_2} \cdot \dots \cdot P_m^{x_m}$  with  $x_m > 0$ . If  $a_i > 0$  for  $i \leq m$  then  $n$  codes a sequence. E.g.  $30 = 2 \cdot 3 \cdot 5$  is a sequence number  $(0,0,0)$  but  $14 = 2 \cdot 7$  is not (there is a gap in the prime numbers).

---

## Instruction ( $n$ ):

$n$  codes a register machine instruction.

This holds iff  $n=2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3}$  with

$x_1, x_2 \geq 2$  and  $x_3 \neq 1$ .

---

## Prog ( $p$ ):

$p$  codes a program. This holds iff  $p = 2^{x_1} \cdot$

$3^{x_2} \cdot \dots \cdot P_r^{x_r}$  where each  $x_i > 0$  and  $x_i - 1$

codes a register machine instruction (i.e.

Instruction ( $x_i - 1$ ) holds). We can check if

$p$  codes a program by decoding twice.

---

If  $\text{prog}(p)$  holds the number of registers required by the program is the greatest number  $y$  occurring in an instruction  $(y,p)$  or  $(y,p,q)$ , we could call this  $R(p)$ .

---

**State  $(m,p)$ :**

$m$  codes a state for the program with code  $p$ . (A state is a sequence  $j, r_1 \dots r_s$  where  $r_i$  denotes the contents of register  $i$  and  $j$  denotes the current instruction.)

This holds iff  $\text{prog}(p)$  holds and  $m$  codes a sequence of length  $R(p) + 1$ .

---

A complete computation of a program is a finite sequence of states, therefore, it is possible to code the complete course of a computation.

---

## A Universal Register Machine

The input for a universal register machine is a pair  $(p, m)$  where  $p$  encodes a program and  $m$  the current input to the program. We can define a function  $f(p, m)$  which returns the value of program  $p$  applied to input  $m$ .

---

We now need to construct a register machine to compute  $f(p,m)$  (if  $p$  does not code a program we will let  $f$  be undefined).

---

## Outline of the Universal Register Machine

(1) Test for whether or not  $p$  is a program number. If no go into an infinite loop. If yes load

$$\langle 1, m, \underbrace{0, \dots, 0}_{R(p)-1} \rangle$$

into register 1. This is the code for the initial state of the program.

---

(2) A loop to simulate the steps of the computation of the program encoded by  $p$ . (Denote this program by  $L$ .) Examine  $\bar{2}$  to find the current instruction of  $L$ . If this is halt, exit the loop. Otherwise interpret the instruction (no) as an instruction of the program  $L$  (which is coded in register 2).

---

Replace  $\bar{2}$  by the code for the next state.

(3) Exit

Examine  $\bar{2}$  and read from it the output of the simulated behaviour of  $L$ .

**Note:**

We have considered the machine to have 2 registers. Other ones (one?) will be needed to carry out decoding and

encoding.

### **1.3. The equivalence of the three**

---

### **'types' of computability we have considered**

(i.e. Turing computable, recursive, register machine computable.)

#### **1.3.1. Theorem**

Any primitive recursive function is register machine computable.

---

## **Proof**

We show this by induction on the definition of a primitive recursive function.

---

## Base Cases

The base cases are  $Z(x)$ ,  $S(x)$ ,  $U_i^n(x_1, \dots, x_n)$ .

The register machines are

$Z(x)$ :             $\hat{1}$  (1,1,2)

$\hat{2}$  Halt

$S(x)$ :             $\hat{1}$  (1,2)

$\hat{2}$  Halt

---

$U_1^n:$                      $\hat{1}$  Halt

$U_i^n(x_1, \dots, x_n):$      $\hat{1}$  (1,1,2)

(for  $i > 1$ )                 $\hat{2}$  ( $i,3,4$ )

$\hat{3}$  (1,2)

$\hat{4}$  Halt

---

## Induction Steps

### (1) Composition

Suppose  $f(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$  where  $g, h_1, \dots, h_m$  are primitive recursive. By the induction hypothesis  $g, h_1, \dots, h_m$  are register machine computable.

---

By appropriate renumbering of the registers and instructions we may suppose the computations for  $h_1, \dots, h_m$  are performed on disjoint sets of registers beyond  $m, n$ .

---

Sketch of program to compute  $f$ :

$\hat{1}$  (copy  $x_1, \dots, x_n$  to input registers for  $h_1, 2$ )

$\hat{2}$  (copy  $x_1, \dots, x_n$  to input registers for  $h_2, 3$ )

$\hat{m}$  (copy  $x_1, \dots, x_n$  to input registers for  $h_m, m+1$ )

$\hat{m+1}$  (compute  $h_1(x_1, \dots, x_n), m+2$ )

$\hat{2m}$  (compute  $h_m(x_1, \dots, x_n), 2m+1$ )

$\hat{2m+1}$  (transfer the outputs  $y_1, \dots, y_m$  of the computations  
of  $h_1, \dots, h_m$  to registers  $1, \dots, m; 2m+2$ )

$\hat{2m+2}$  (compute  $g(y_1, \dots, y_m), 2m+3$ )

$\hat{2m+3}$  Halt

---

## (2) Recursion

Suppose  $f$  has been defined from  $g$  and  $h$  by recursion, i.e.

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= h(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y+1) &= \\ g(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \end{aligned}$$

By the induction hypothesis  $h$  and  $g$  are register machine computable.

---

Again we can renumber registers and instructions and suppose that the computations of  $h, g$  are carried out in registers  $n+5, \dots, s$  where  $n+5$  is the output register (for  $h$  and  $g$ ),  $n+5, \dots, 2n+4$  the input registers for  $h$  and  $n+5, \dots, 2n+6$  the input registers for  $g$ .

---

**Note:**

Registers  $1, \dots, n, n+1$  hold  $x_1, \dots, x_n, y$ . To compute  $f(x_1, \dots, x_n, y)$  (following the recursive definition) we calculate  $f(x_1, \dots, x_n, 0)$  then  $f(x_1, \dots, x_n, 1)$  and so on until we have  $y$  in the  $(n+1)$ th argument place.

---

Registers  $n+2, n+3, n+4$  hold respectively the current value of the  $(n+1)$ th argument place (we call this  $Z$ ),  $y - Z$  and  $f(x_1, \dots, x_n, Z)$ .

---

The outline of the register machine is:

- $\hat{1}$  (copy register  $n+1$  to register  $n+3, 2$ )      initially  $Z=0$   
 $y-Z=y$
- $\hat{2}$  (copy registers  $1, \dots, n$  to registers  
 $n+5, \dots, 2n+4; 3$ )
- $\hat{3}$  (compute  $h$  using  $\overline{n+5}, \dots, \overline{2n+4}; 4$ )
- $\hat{4}$  (copy register  $n+5$  to  $n+4, 5$ )
- } Compute  
 $f(x_1, \dots, x_n, 0)$

---

$\hat{5}$  ( $n+2, 6$ )

new value of  $Z$

$\hat{6}$  ( $n+3, 7, 10$ )

new value of  $y-Z$

$\hat{7}$  (copy registers  $1, \dots, n, n+2, n+4$  to registers  $n+5, \dots, 2n+6; 8$ )

$\hat{8}$  (compute  $g$  using  $\overline{n+5}, \dots, \overline{2n+6}; 9$ )

$\hat{9}$  (copy register  $n+5$  to  $n+4, 5$ )

$\hat{10}$  (copy register  $n+4$  to  $1, 11$ )

$\hat{11}$  Halt

---

We have shown that every primitive recursive function is register machine computable. However, there do exist register machine computable functions which are not primitive recursive (for example, the primitive recursive enumerating function and the function given in question 5 of Exercise 2).

---

Let us consider the function which enumerates the 1-variable primitive recursive function.

Recall  $g(n, x) = f_n(x)$  where  $f_0(x), f_1(x), \dots$  is an enumeration of all the 1-variable primitive recursive functions.

---

Now  $g(x,x) = f_x(x)$  and since  $f_x(x)$  is register machine computable (by Theorem 1.5.1) then  $g$  is. However,  $g$  is not primitive recursive.

---

To obtain the class of all (partial or total) register machine computable functions from the primitive recursive functions, we need to use the unbounded least number operator, i.e. we need to consider the class of recursive functions.

---

We have the following theorem.

### 1.3.2. Theorem

The (partial or total) function  $f(x_1, \dots, x_n)$  is register machine computable iff there are primitive recursive functions  $g, h$  such that  $f(x_1, \dots, x_n) = h(\mu y(g(x_1, \dots, x_n, y) = 0))$ .

*The proof of this theorem is beyond the scope of the course*

---

Given Theorem **1.5.2** we have that the class of recursive functions is equivalent to the class of register machine computable functions.

---

### 1.3.3. Theorem 1.5.3

Let  $f$  be an  $n$ -argument (partial or total) function.  
 $f$  is Register machine computable iff  $f$  is Turing  
machine computable.

---

**Proof:**

Suppose we have a Turing machine  $T_f$  to compute  $f$ . We need to construct a register machine  $R_f$  to compute  $f$ . Recall that the current configuration of  $T_f$  is given by the contents of the tape (always finite although there is an infinite number of 0's), which cell is being scanned and the current state of  $T_f$ .

---

We code the contents as a number sequence and put the code into the first non-input register. We then code the number of the scanned cell and the current state and hold these in the next registers respectively.

---

Now for each pair  $(q(t), s(t))$  we need a register program to mimick the action of  $T_f$ , i.e.  $(q(t+1), s(t+1), L \text{ or } R)$ . We need to perform the correct transformations on the three registers coding the current configuration.

---

This is easy since, for example, if  $q(t) = q_i$  and  $q(t+1) = q_{i \pm s}$  then we need a register machine program to add  $\pm s$  to the contents of the state register.

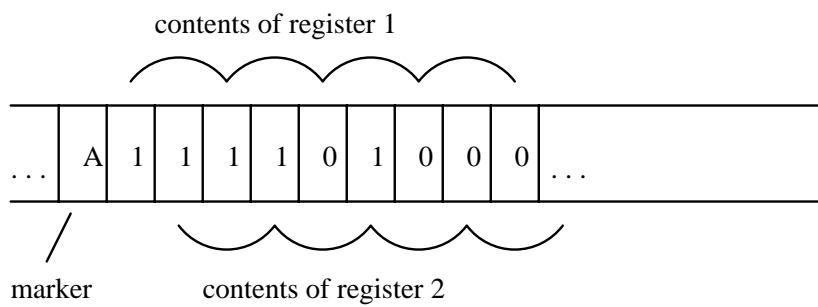
When  $T_f$  halts the register machine  $R_f$  should decode the tape contents (see Exercise 2, Question 5) and put the result in the output register.

---

Conversely, suppose we have a register machine  $R_f$ . We show how to construct  $T_f$  (which simulates the action of  $R_f$ ). Suppose  $R_f$  has  $r$  registers. We use cells on the tape which are a multiple of  $r$  apart to act for each register, e.g. If  $r = 2$  the odd cells handle the contents of register 1 and the even cells handle the contents of register 2.

---

Let us adopt a unary notation for the input, e.g. suppose  $R_f$  has input  $[2,3]$ . We will code this as  $110 \dots, 1110 \dots$ . This will be written on  $T_f$ 's tape as follows:



---

Note that, whatever form the input of  $R_f$  is put onto the tape we can construct a Turing machine to put it into the above form.

---

Now we have to simulate the register machine instructions  $(n,p)$ ,  $(n,p,q)$ . The marker A shows where the contents of the registers start on the tape. For  $(n,p)$  we move  $n$  cells to the right (to find the part of the tape standing for  $\bar{n}$ ).

---

Then we move to the right in blocks of  $r$  (the number of registers of  $R_f$ ) until we find a 0 which we replace by 1. We then return to A. The instruction  $(n,p,q)$  is handled in a similar manner.

---

When 'halt' is reached  $\bar{1}$  can be put in a more readable form and the rest of the tape cleared (using a 'clean up' Turing machine).

---

## 2. Unsolvable Problems

### 2.1. The Halting Problem for Turing Machines

The computation of any Turing machine can be specified by a machine tape pair of form  $(T,t)$  where  $T$  is a description of the Turing machine (i.e. set of quintuples, table, diagram). We will restrict to the tape description  $d_T$  defined on page 24.  $t$  is the initial tape configuration of the Turing machine so we will consider any Turing machine to be specified by  $(d_T,t)$

---

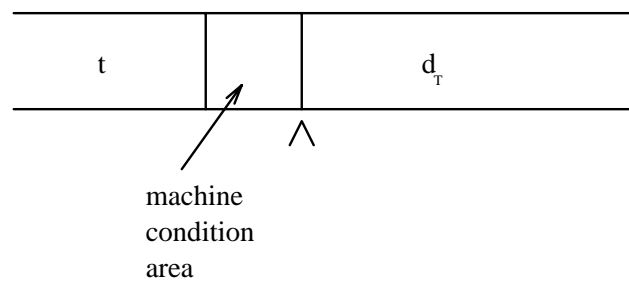
The halting problem is:

“Does there exist an effective procedure (computable function) for deciding, for every machine-tape pair  $(d_T, t)$ ; does  $T$  halt for  $t$ ?”

---

**Note:**

This problem could be made more specific in terms of the Universal Turing machine  $U$ . We would say, “For each initial input of the form:



Does  $U$  halt?”

---

## 2.2. Theorem

The halting problem is unsolvable.

### **Proof:**

By way of contradiction we assume that the halting problem is computable. Assuming the Church-Turing thesis there must exist a Turing machine  $D'$  to decide, for each  $(d_T, t)$ , whether or not  $T$  halts.

---

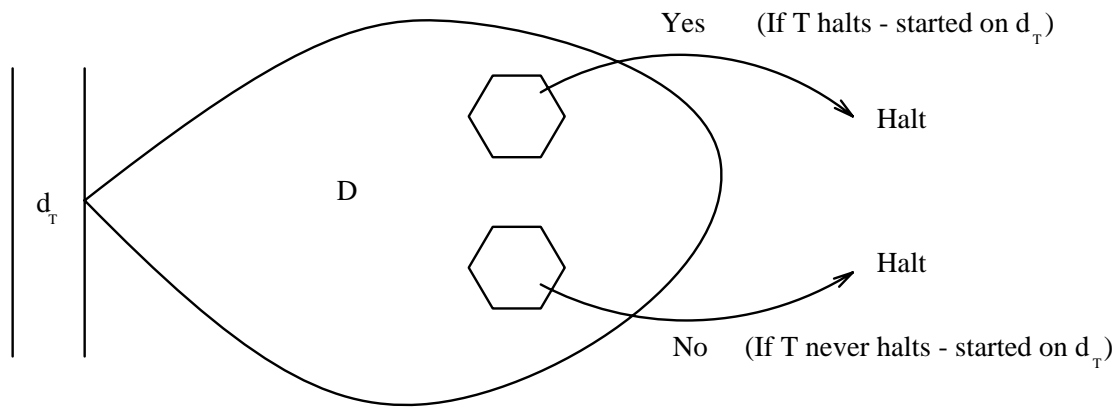
Now if  $D'$  can solve the halting problem for all tape pairs  $(d_T, t)$  then it can certainly do it for the particular tape pairs  $(d_T, d_T)$ : we need not consider why anyone would want to perform such peculiar operations (is there anything wrong with a man contemplating a description of his own brain?). The reason we do this is so that the input to the machine can be  $d_T$  rather than  $(d_T, t)$ .

---

It is easy to construct a new machine  $D$  which accepts only  $d_T$  but otherwise acts like  $D'$  (i.e.  $D$  is just  $D'$  together with some states which will copy  $d_T$  into the cells allocated for  $t$ ). Note that we are restricting the problems we can look at but if  $D$  can be shown not to exist then  $D'$  cannot have existed. (i.e. it would follow immediately that we cannot have a machine to answer the general question, “Does  $T$  halt for  $t$ ?” because this would include the solution of the problem “Does  $T$  halt for  $d_T$ ?”).

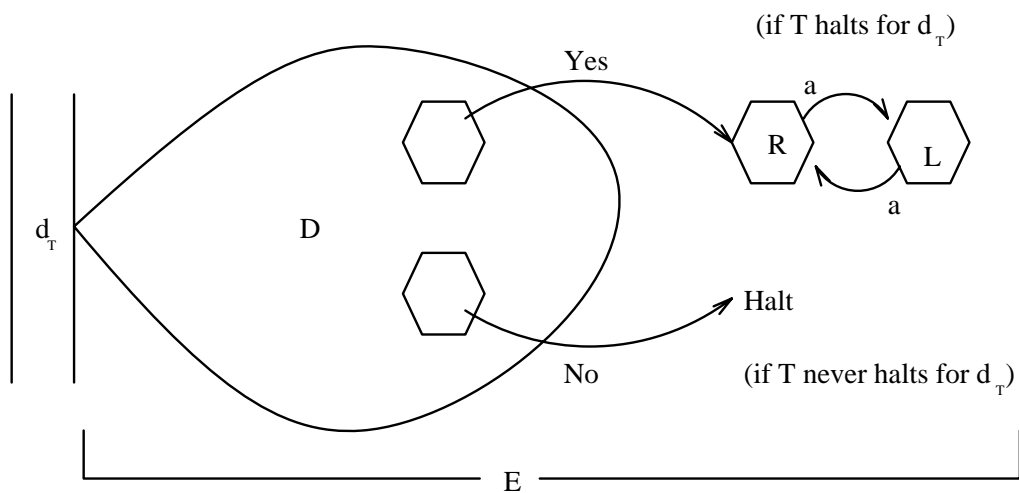
---

So  $D$  will be of the form:



---

We now convert  $D$  to a new machine  $E$  by adding two further states as follows:



In the diagram  $a$  is taken to be any symbol.

---

Now  $E$  itself is a Turing machine so we can let  $E$  look at itself (i.e. let the input to  $E$  be  $d_E$ ).

---

If  $E$  halts for  $d_E$  then  $D$  answers ‘yes’ and then  $E$  goes into a non-terminating right-left loop, therefore,  $E$  never halts for  $d_E$  which is a contradiction. If  $E$  never halts for  $d_E$  then  $D$  answers ‘no’ and halts, i.e.  $E$  halts for  $d_E$  which is a contradiction. Therefore, neither  $E$ ,  $D$  or  $D'$  can exist so the halting problem is effectively unsolvable.

---

**Note:**

We have shown that we cannot construct a Turing machine to decide whether or not for *any* tape pair  $(T,t)$   $T$  will halt for  $t$ . However, given a particular tape pair  $(T_o,t_o)$  it is *impossible* to prove that you *cannot* decide whether or not to halt for  $t_o$ .

---

## 2.3. Theorem

Given any particular  $(T_o, t_o)$  we cannot prove that there is no way to find out if  $(T_o, t_o)$  halts.

---

**Proof:**

Suppose by way of contradiction that we can *prove* that there is *no* way to decide if  $(T_o, t_o)$  halts.

Now suppose we set  $(T_o, t_o)$  running.

Either:

- (i) it halts;
- (ii) it never halts.

---

If (i) then when we set it running it eventually halts which contradicts the hypothesis (therefore, there was a way to decide if it halts, namely set it running). Therefore, (ii) must be the case.

In other words, the fact that there is no way to find out if it halts or not is used to show it never halts — a contradiction.

---

## **2.4. Problems Related to the Halting Problem**

We will show that the following problems are unsolvable by showing that they are equivalent to a problem we already know to be unsolvable.

We are considering decision problems, i.e. problems which answer ‘yes’ or ‘no’.

---

Two decision problems  $A, B$  are equivalent ( $A \Leftrightarrow B$ ) if  $A$  answers ‘yes’ iff  $B$  answers ‘yes’ and  $A$  answers ‘no’ iff  $B$  answers ‘no’.

---

## 2.4.1. The Printing Problem

Given any tape pair  $(T,t)$  does  $T$  ever print the symbol  $s$  when applied to  $t$ ?

## 2.4.2. Theorem

The printing problem is unsolvable.

---

**Proof:**

Suppose  $T$  does not have  $s$  in its alphabet. Then we can alter it to a new machine  $T^*$  which prints  $s$  before each of its halting states. Then Halting Problem for  $T \Leftrightarrow$  Printing Problem for  $T^*$ . Suppose  $T$  does have  $s$  in its alphabet. Then we can easily alter it to a new machine  $T'$  which does not by changing  $s$  to a symbol not already used.

Then halting Prob. for  $T \Leftrightarrow$  Halting Prob. for  $T'$   
 $\Leftrightarrow$  Printing Prob. for  $T'^*$ .

---

If the Printing Problem is solvable for all  $T'^*$   
then the halting problem is solvable for all  $T$ .

$\therefore$  By Theorem **2.2** the Printing Problem is  
unsolvable.

---

### **2.4.3. The Blank Tape Halting Problem**

For any machine  $T$ , “Does  $T$  halt when started on a blank tape?”

### **2.4.4. Theorem**

The blank tape halting problem is unsolvable.

---

**Proof:**

Consider any machine tape pair  $(T, t)$ . We construct a further machine  $M_{(T, t)}$  from  $(T, t)$  as follows:

Suppose  $t$  is

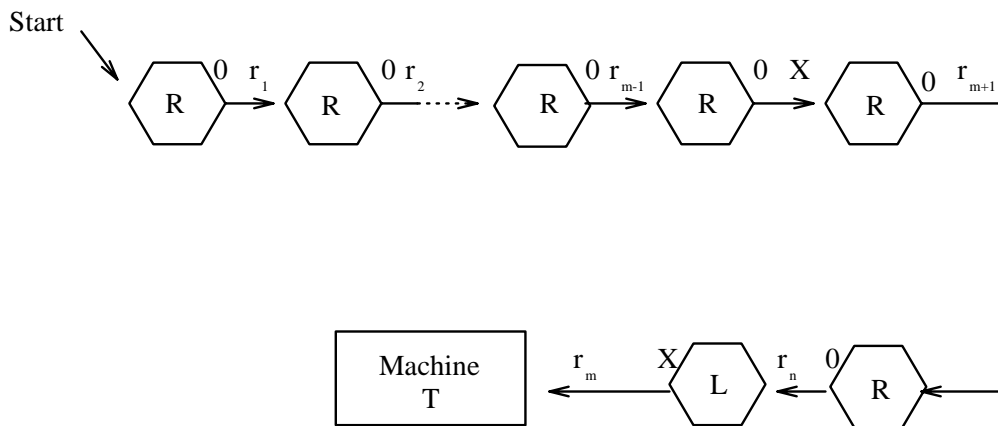
$$\dots 0 r_1 \dots r_m \dots r_n 0 \dots$$

$\wedge$   
Initial state of T

where  $r_1, \dots, r_n$  represent the non-zero part of  $t$ .

---

$M_{(T,t)}$  is constructed by preceding  $T$  with the following states:



So when  $M_{(T,t)}$  is applied to a blank tape it first writes out  $t$  and then hands over to  $T$ .

$\therefore$  The blank tape halting problem for  $M_{(T,t)}$

$\Leftrightarrow$  the halting problem for  $(T,t)$ .

---

---

∴ By Theorem **2.2** the blank tape halting problem is unsolvable.

**Note:**

This is the case when the blank tape is replaced by any other fixed initial tape.

---

## 2.4.5. The Uniform Halting Problem

For any machine  $T$  “Does  $T$  halt for every input tape?”

## 2.4.6. Theorem

The Uniform Halting Problem is unsolvable.

---

**Proof:**

Suppose we have any machine  $T$ . We construct a further machine  $T'$  as follows:

---

Let  $A, B$  be two symbols not used by  $T$ . For each state  $q_i$  of  $T$  we add the quintuples

$$(q_i A q_i' O L) \quad (q_i' Q q_i A R)$$

$$(q_i B q_i'' O R) \quad (q_i'' Q q_i B L)$$

where  $q_i', q_i''$  are two new states, added to  $T$  for each  $q_i$ , and  $Q$  denotes any symbol in the alphabet of  $T$ .

---

Suppose  $T$  starts in state  $q_0$ . Then we also add the following sets of quintuples of  $T$  to form  $T'$ :

$$(q_0^* Q q_1^* A R) \quad (q_1^* Q q_2^* O R) \quad (q_2^* Q q_0 B L)$$

For each symbol  $Q$  of the alphabet of  $T$  and where  $q_0^*, q_1^*, q_2^*$  are new states.  $T'$  is now started in  $q_0^*$ .

---

Consider the operation of  $T'$  on any tape:

(1)                   ...QQQQQ...  
                           $\wedge^*$   
                           $q_0$

(2)                   ...QAOBQ...  
                           $\wedge$   
                           $q_0$

Then  $T'$  hands over to  $T$ .

---

Suppose  $T$  carries out the step:

(3)

...QAS<sub>1</sub>BQ...

^

$q_i$

---

Then  $T'$  takes over and carries out the steps:

(4)

...QAS<sub>1</sub>OQQ...

^

$q'_i$

(5)

...QAS<sub>1</sub>OBQ...

^

$q_i$

---

Suppose  $T$  carries out the steps:

(6)

...QAS<sub>1</sub>S<sub>2</sub>BQ...

^ ,  
q<sub>j</sub>

---

then  $T'$  takes over and carries out the steps:

(7)

...QQOS<sub>1</sub>S<sub>2</sub>BQ...

^,

$q_j$

(8)

...QAOS<sub>1</sub>S<sub>2</sub>BQ...

^

$q_j$

---

Notice that  $T'$  sets up  $T$  so that it always appears to be operating on a “ $O$ ” in its current cell (i.e. on a blank tape).

$\therefore$  we have  $T$  will halt on a blank tape iff  $T'$  halts on every tape.

$\therefore$  by Theorem **2.4.4** the Uniform Halting Problem is unsolvable.

---

## 3. Complexity Theory

### 3.1. Introduction

We have considered the class of ‘computable’ functions and we have looked at some unsolvable problems. In this section we attempt to separate the ‘practically’ computable functions from those that cannot be carried out on a computer in a reasonable time.

---

When constructing a computer program it is important to consider how expensive the program is in terms of storage and time.

---

When considering the complexity of an algorithm we measure the number of ‘primitive’ steps taken. ‘Primitive’ could mean the number of steps on a Turing Machine or Register Machine (though usually it is not at so low a level).

---

It is better to consider Turing Machines for the following because register machines allow a number of arbitrary size in each register whereas each cell on a Turing Machine contains just one symbol. Register Machines were ideal when we were considering ‘computability’ without regard for efficiency but Turing Machines become more useful when we wish to consider these issues.

---

Now for a Turing Machine the number of cells required on the tape cannot be greater than the number of steps required because we never scan more than one cell at a time. For this reason we can look exclusively at the time efficiency of algorithms.

---

Given an algorithm  $A$  and inputs of size  $n$  we define the time complexity function  $T_A(n)$  to be the greatest number of primitive steps taken by  $A$  for all  $n$ .

---

We are usually more interested in the ‘growth’ of  $T_A(n)$  rather than actual values. Thus to determine whether or not it is feasible to implement  $A$  we require a (not too large) upper bound for  $T_A(n)$ . We assume we implement each algorithm on a Turing Machine.

---

We say an algorithm  $A$  can be performed in polynomial time if  $T_A(n) \leq f(n)$  and  $f$  is a polynomial, i.e. is of form  $a_1 n^p + a_2 n^{p-1} + \dots + a_m$ .

---

We say it can be performed in exponential time if  $T_A \leq f(n)$  and  $f(n)$  is of the form  $a_1^n + a_2^{n-1} + \dots + a_n$  where  $a_i > 1$ .

For most purposes it is the polynomial time functions that are feasible and the exponential ones that are not. If a problem is solvable in polynomial time the power  $p$  is usually quite small (less than 5, say).

---

### 3.1.1. Definitions

We say  $g = O(f)$  if, for some  $N$ ,  $g(x) \leq kf(x)$  for all  $x > N$  and some constant  $k$ . (This is read ‘ $g$  is of the order of  $f$ .’) The constant  $k$  is used to allow for the fact that computer technology will get faster. In a year computers might be five times as fast so we let the constant be five times as big.

---

### 3.1.2. Example

$$3x^2 + 2x + 2 = O(x^2)$$

$$\begin{aligned} \text{Since } 3x^2 + 2x + 2 &\leq 3x^2 + 2x^2 && \text{for } x \geq 2 \\ &\leq 5x^2 && \text{for } x \geq 2. \end{aligned}$$

We allow some initial failures of  $f$  to exceed  $g$  (in this case  $x=1$ ). What is *usually* important is the eventual behaviour. However, this is not always the case (see example 3.1.3).

---

Note that  $3x^3 + 2x + 2 \neq O(x^2)$ .

We say that  $f \approx g$  ( $f$  is “equivalent” to  $g$ ) if  $f = O(g)$  and  $g = O(f)$ .

We have that  $x^2 = O(3x^2 + 2x + 2)$  since  $x^2 \leq 3x^2 + 2x + 2$  for all  $x \geq 0$ .

---

We could say  $3x^2 + 2x + 2 \approx x^2$ .

That is if we have an algorithm  $A$  such that  $T_A(x) = 3x^2 + 2x + 2$  and an algorithm  $B$  such that  $T_B(x) = x^2$  then the two algorithms are considered to have the ‘same’ complexity (at least one is not significantly worse than the other).

Algorithm  $A$  is said to be better than algorithm  $B$  if  $T_A(x) = O(T_B(x))$  but  $T_B(x) \neq O(T_A(x))$ .

---

However, as was indicated earlier this definition of ‘better’ might not always hold.

### 3.1.3. Example

$$\text{Let } g(x) = \begin{cases} 1,000,000 & \text{for } x \leq 1,000,000 \\ x & \text{for } x > 1,000,000 \end{cases}$$
$$\text{Let } f(x) = \begin{cases} x & \text{for } x \leq 1,000,000 \\ x^2 & \text{for } x > 1,000,000 \end{cases}$$

---

Now

$g(x) = O(f(x))$  since

$g(x) < f(x)$  for all  $x > 1,000,000$  and for the same reason

$f(x) \neq O(g(x))$ .

---

However, suppose we have an algorithm  $A$  such that  $T_A(x) = g(x)$  and  $B$  such that  $T_B(x) = f(x)$ .

We would say  $A$  is better than  $B$  by the above definition but it may be that for practical purposes we never consider inputs as large as 1,000,000. If this is the case algorithm  $B$  would be preferable to algorithm  $A$ .

---

In practice though, such occasions rarely arise and the O-notation and ‘better than’ definition are useful. Constants appearing in time complexity functions are usually relatively small.

---

### 3.1.4. Definition

We write  $g = o(f)$  if

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \rightarrow 0.$$

This says that  $f$  grows more rapidly than  $g$  (i.e.  $g$  is of smaller order than  $f$ ).

Clearly  $g = o(f) \Rightarrow g = O(f)$

However  $g = O(f) \ \& \ f \neq O(g) \Rightarrow g = o(f)$

---

## Exercise:

Prove this. Hint:

$$\text{let } g(x) = \begin{cases} x^2 & \text{if } x \text{ is even} \\ 2x & \text{if } x \text{ is odd} \end{cases}$$
$$f(x) = x^2.$$

---

## 3.2. Some Standard Complexity Functions

### 3.2.1. Theorem

(a) 
$$a_p n^p + a_{p-1} n^{p-1} + \dots + a_0 \approx n^p$$
 provided  $a_p > 0$ .

(b)  $n^p = o(n^q)$  if  $0 \leq p < q$ .

(c)  $n^p = o(q^n)$  if  $q > 1$ .

**Proof:**

*Exercise*

---

We can see that  $\log n = o(n^p)$  for positive  $p$  since if we let  $N = \log n$

$$\frac{\log n}{n^p} = \frac{N}{(2^N)^p} \quad (\text{as convention we take logs to base 2}).$$

---

This tends to  $o$  as  $N \rightarrow \infty$  by Theorem 3.1 (c).  
Hence it tends to  $o$  as  $n \rightarrow \infty$ . Thus  $\log n$  grows more slowly than any positive power of  $n$ . A more slowly growing function than  $\log n$  is  $\log \log n$ .

---

## 3.3. Some Algorithms

### 3.3.1. (Binary) Addition

Suppose we have two binary numbers with  $n$  digits,  $a = a_{n-1}a_{n-2}\dots a_0$ ,  $b = b_{n-1}b_{n-2}\dots b_0$ .

---

We have  $a + b = d = d_n d_{n-1} \dots d_0$ .

$d_i$  is obtained by adding and carrying in the usual way, i.e.  $d_i = a_i + b_i + c_i \pmod 2$  where  $c_{i-1}$  is the 'carry' from the previous sum.

$c_0 = 0, c_i = \lfloor (a_{i-1} + b_{i-1} + c_{i-1}) / 2 \rfloor$  ( $0 < i \leq n$ ) ( $d_{n+1} = c_n$ ).

---

It is impossible to calculate the  $d_i$  in parallel (because of the ‘carry’). Each must await its turn in the order of computation.

---

Therefore, the complexity of the standard binary addition algorithm is  $O(n)$ . (Similar ideas pass over to base 10 addition. Complexity is  $O(n)$ .)

However, it is possible (using parallelism) to calculate the  $c_i$  in time complexity  $O(\log n)$ .

Then (again using computers working in parallel) to compute the  $d_i$ .

---

## 3.3.2. Finding the Maximum (or Minimum) Element of a List

### 3.3.2.1.

Suppose we have a list of  $n$  elements  $x_1, \dots, x_n$  and suppose we require an algorithm to pick out the maximum element.

---

The following algorithm does this in  $n - 1$  steps:

Step 1:     Let  $x = \max(x_1, x_2)$

Step 2:     Let  $x = \max(x, x_3)$

Step  $n - 1$    Let  $x = \max(x, x_n)$ .

---

Clearly the algorithm does the job. The algorithm is optimal in the sense that it is not possible to construct an algorithm which will do the job using less comparisons. To see this we suppose that there is an algorithm which determines the maximum using  $r$  comparisons.

---

We can form a graph on  $\{1, \dots, n\}$  by joining  $i, j$  if they have been compared.

The graph has  $r$  edges.

---

This graph must be connected (ie there must be a path from 1 vertex to any other).

---

To see this suppose there are  $x_i, x_j$  that we are connected and suppose that  $x_i$  has been determined to be maximum. Now take the same list except that we replace  $x_j$  by  $x_{j'}$  such that  $x_{j'} > x_i$  and run the algorithm. It would still return  $x_i$  since there is no path between  $x_i, x_j$  in the graph. So the graph must be connected.

---

A well known result from Graph Theory states that a connected graph on  $n$  vertices has at least  $n-1$  edges (See any standard book on graph theory).

$$\therefore r \geq n-1.$$

In this example the most obvious method was optimal. This is unusual.

---

**Note:**

To pick out the minimum element we use a trivial adaptation of the same algorithm.

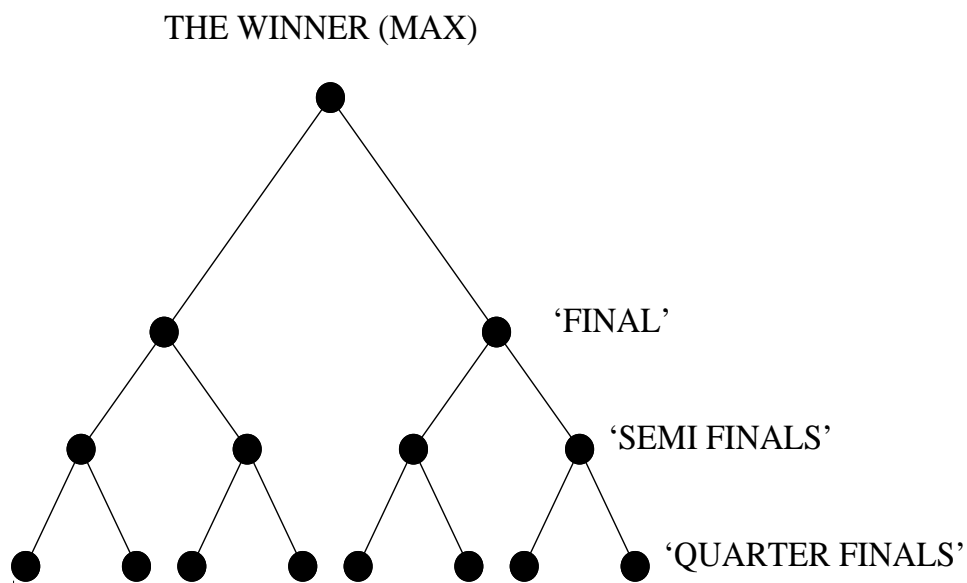
---

### **3.3.2.2.**

This is an alternative way of finding the maximum (minimum) element of a list where we imagine the elements to be playing a ‘knockout’ tournament. We make an arbitrary ‘draw’ which splits the list into pairs (some elements might get a ‘free passage’ into the next round).

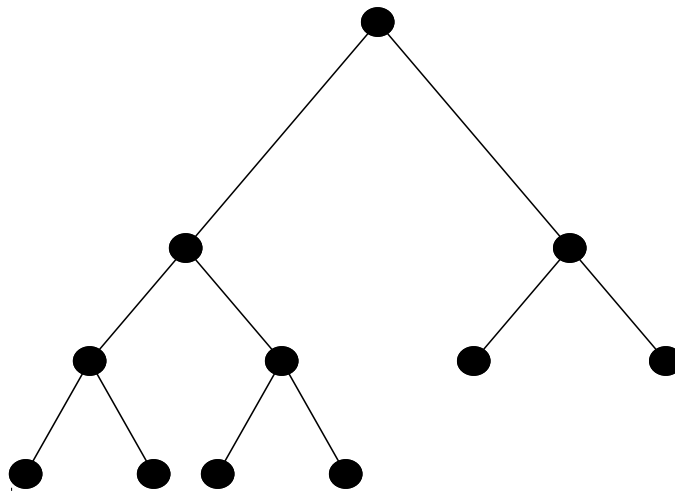
---

The results can then be arranged in a binary tree as follows:



---

This is for a list of 8 elements. For 6 elements we would have:



---

Notice that 2 elements get drawn (free passage) into the semi-finals (because the size of the list is not a power of 2). Clearly if the number of elements is  $n$  then the number of comparisons (matches played) is  $n - 1$ .

This is an alternative algorithm to **3.2.2.1** but is still optimal.

---

### 3.3.3. Finding the maximum and minimum members of a list

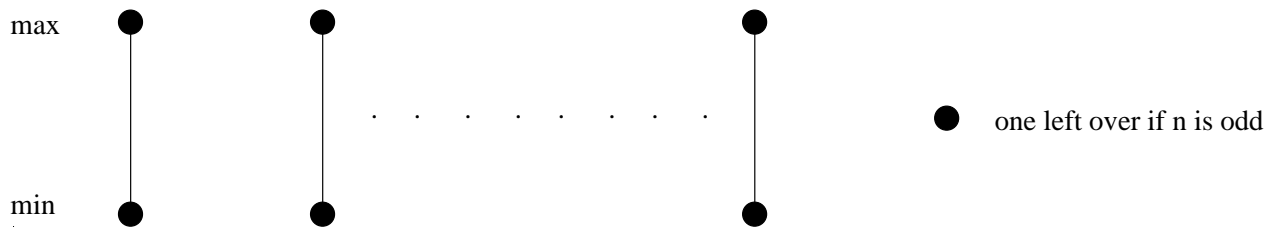
The obvious method is an application of **3.3.2.1** giving  $n-1$  comparisons and then another application (but for min) of **3.3.2.1** on the remainder, giving  $n-2$  comparisons. In total,  $2n-3$  comparisons.

We can improve this method as follows:

---

Divide the list into disjoint pairs (with one left over if  $n$  is odd).

Apply  $\max(x,y)$  to each pair and arrange as in the diagram (we have made  $\lfloor \frac{1}{2}n \rfloor$  comparisons):



---

Now the greatest element of the list is on the top and the least is on the bottom. So we apply **3.3.2.1** to these subsets (with the one left over if  $n$  is odd).

---

The total number of comparisons is:

$$\lfloor \frac{1}{2}n \rfloor + 2(\lfloor \frac{1}{2}(n+1) \rfloor - 1)$$

If  $n$  is even:  $\frac{1}{2}n + 2(\frac{1}{2}n - 1) = \frac{3n}{2} - 2$

If  $n$  is odd:  $\frac{1}{2}(n-1) + 2(\frac{1}{2}n - \frac{1}{2}) = \frac{3n}{2} - 3$

This is about  $\frac{3}{4}$  of the number required by the obvious method.

---

### **3.3.4. Finding the 2 greatest elements in a list**

Again the obvious method would be 2 applications of **3.3.2.1** giving  $2n-3$  comparisons. An improvement is to first apply **3.3.2.2** (giving  $n-1$  comparisons).

---

The overall ‘winner’ is the largest element. The next greatest element must have been ‘beaten’ at some point by the winner. So we follow the winner’s path back down the tree. We now apply **3.3.2.1** (or **3.3.2.2**) to the  $\lfloor \log_2(n-1) \rfloor + 1$  elements defeated by the winner. This is done in  $\lfloor \log_2(n-1) \rfloor$  comparisons.

---

$\therefore$  the overall number of comparisons is  $n + \lfloor \log_2(n-1) \rfloor - 1$ .

For large  $n$  this is proportionally only a little larger than  $n$ .

---

## 3.4. NP-Complete Problems

### 3.4.1. Introduction

Here we investigate the class of problems which are computable but for which no efficient algorithm is known. This class of problems is known as NPC. All known algorithms for problems in this class take exponential time.

---

However, it has not yet been proven that they take greater than polynomial time but most people think this is the case. P is taken to be the class of polynomial time problems.

---

## Example:

TRAVELLING SALESMAN: Suppose we have a set  $\{c_1, \dots, c_n\}$  of cities and for each  $c_i, c_j \in \{c_1, \dots, c_n\}$  a distance  $d_{ij}$  between the cities.

---

Can a travelling salesman starting at  $c_1$  visit each of the other cities just once and only then return to  $c_1$ , with a total distance  $\leq b$  (for some given integer  $b$ ?). (Can we construct an (efficient) algorithm which will return the smallest possible (optimal) distance travelled?)

---

An algorithm is traditionally viewed as being deterministic. This means that at each step of the algorithm the next step is uniquely defined.

It is easy to construct an algorithm which will (eventually) solve the problem: consider all  $(n-1)!$  orderings of  $c_1, \dots, c_n$  evaluate the total distance of the tour starting and ending at  $c_1$  for each ordering and pick the smallest.

---

This algorithm is clearly very inefficient.

If we had unlimited parallelism we could solve the problem in polynomial time. We could assign a machine to each of the  $(n-1)!$  orderings and set them running in parallel. Again, this is not a reasonable suggestion.

---

With only one machine we need to be able to ‘guess’ (in some way) which of the  $(n-1)!$  orderings is best, i.e. the algorithm has to ‘choose’ its next possible move. This is what we mean by a non-deterministic machine.

---

The Turing Machines we considered were ‘deterministic’ ones, i.e. given a state symbol pair there was a unique triple (new state, new symbol and direction). With a non-deterministic Turing Machine we have a set of triples.

---

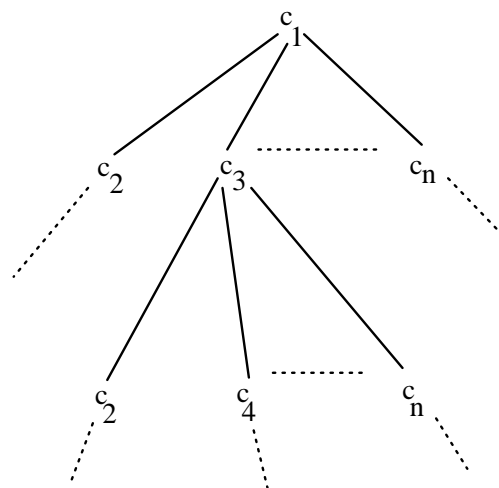
A deterministic Turing Machine (DTM) solves a problem if it always gives the correct answer to a valid input. The class  $P$  is the set of decision problems that can be solved in polynomial time by a DTM. A non-deterministic Turing Machine (NDTM) solves a problem if there is ANY sequence of 'choices' which would allow it to give a correct answer to a valid input.

---

The class NP is the set of decision problems which can be solved in polynomial time by a non-deterministic Turing Machine. The NP stands for non-deterministic polynomial time. We deal with decision problems. Most problems have a related decision form. Let us consider the Travelling Salesman problem in order to get an intuitive idea of what is going on.

---

We are asking whether or not there is a route with distance  $\leq b$  for some given  $b$  (we have formulated it as a decision problem). Now given a guess at the problem we could quickly check whether or not the answer was yes or no. There are, of course, many guesses  $((n-1)!)$ . The guesses correspond naturally to the branches of a tree with a fixed degree of branching at each vertex.



---

The significant point is that the depth of the tree is relatively small. The depth of the tree corresponds to the number of stages needed in the checking stage which is bounded by a polynomial in the size of the input.

---

A NDTM can solve the problem in polynomial time because there is a sequence of choices (which a NDTM can be thought of as making) which will return the correct answer in polynomial time.

---

Suppose we have two decision problems  $A$  and  $B$ .  $A$  is said to be polynomially reducible to  $B$  (written  $A \leq B$ ) if from any instance of  $A$  we can construct, in polynomial time, an instance of  $B$  such that the two instances are either both ‘yes’ instances or ‘no’ instances.

---

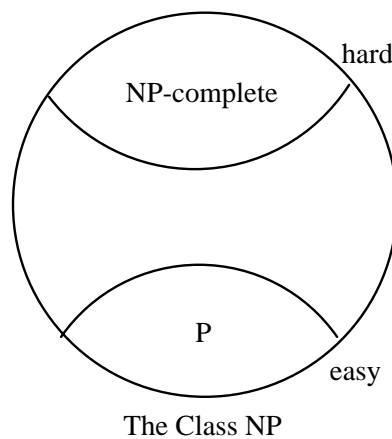
(This is the same idea that we used when showing the unsolvability of the Printing Problem, blank tape problem and the uniform halting problem. We reduced them to a problem we already know to be unsolvable.)

---

We now introduce Cook's idea of NP-complete. Intuitively, this defines the idea of the hardest problems in NP, or rather, any problem which is NP-complete is as hard as any other problem in NP.

---

More formally, a problem  $A$  is NP-complete if  $A \in \text{NP}$  and for any  $B \in \text{NP}$ ,  $B \leq A$ . The diagram shows the class NP as it is thought of by most people:



---

Clearly  $P \subset NP$ . However, it has not yet been shown that  $NP \not\subset P$ . This is one of the most important ‘unanswered’ questions in theoretical computer science.

---

### 3.4.1.1. Lemma

$\alpha$  is transitive, i.e.  $A \alpha B \ \& \ B \alpha C \Rightarrow A \alpha C$ .

---

**Proof:**

Suppose we have Turing Machines  $M_{(A,B)}$  and  $M_{(B,C)}$  which polynomially reduce  $A$  to  $B$  and  $B$  to  $C$  respectively. Let  $M_{(A,C)}$  be the Turing Machine formed by applying  $M_{(A,B)}$  and then  $M_{(B,C)}$ . Clearly  $M_{(A,C)}$  reduces  $A$  to  $C$  and since  $T_{M_{(A,C)}} = T_{M_{(A,B)}} + T_{M_{(B,C)}}$  it does it polynomially.

---

### 3.4.1.2. Theorem

Suppose  $A$  is NP-complete. If  $B \in \text{NP}$  and  $A \leq B$  then  $B$  is NP-complete.

---

**Proof:**

Suppose  $C$  is any problem in NP. Then  $C \leq A$  (because  $A$  is NP-complete). We are given that  $A \leq B$ . By Lemma 3.4.1.1 we have  $C \leq B$ , i.e.  $B$  is NP-complete.

---

### 3.4.1.3. Theorem

Let NPC denote the class of NP-complete problems. If  $\text{NPC} \cap \text{P} \neq \emptyset$  then  $\text{NP} = \text{P}$ .

---

**Proof:**

Let  $A \in \text{NPC} \cap P$ . For any  $B \in \text{NP}$ ,  $B \alpha A$ .

However,  $A \in P$ , therefore,  $B \in P$ , i.e.  $\text{NP} \subset P$ .

---

**Note:**

All the evidence suggests that  $NP \neq P$ . So it appears that *no* NP-complete problem is going to be executable by a polynomial time algorithm. In other words, if we showed that any one NP-complete problem was polynomial we would have shown that  $NP = P$ .

---

## **3.4.2. Examples of NP-Complete Problems**

### **3.4.2.1. SAT:**

For a given propositional formula in conjunctive normal form, there is an assignment of truth values to the propositional variables giving it the value true?

---

**Note:**

The ‘guesses’ here would be assignments of truth values to the propositional variables of the formula. Given a particular guess it can be checked very quickly whether or not the formula is true. Once again these guesses correspond to the depth of an appropriate tree.

---

### **3.4.2.2. Theorem (Cook's Theorem):**

SAT is NP-complete.

#### **Proof:**

Not part of the course. See Garey & Johnson or  
Rayward-Smith.

---

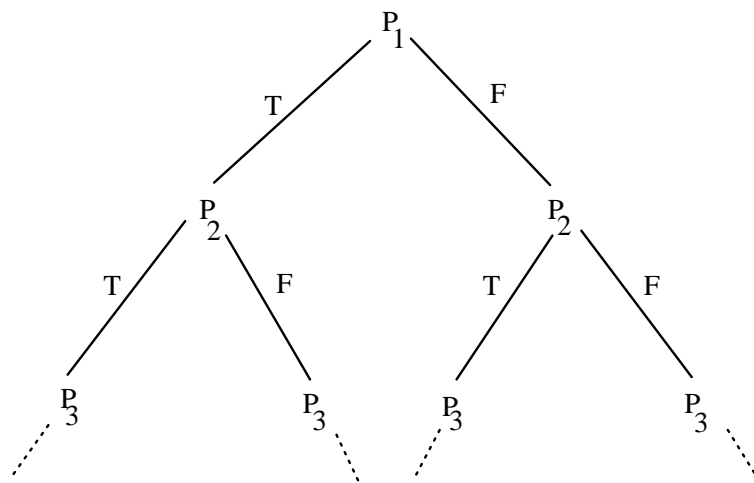
**Note:**

**(a)**

Intuitively, the reason SAT is ‘hard’ is that given a propositional formula in conjunctive normal form, the Turing Machine must make ‘choices’ of truth values  $T$  or  $F$  for each variable occurring.

---

Suppose we have propositional variables  $p_1, \dots, p_n$ . The corresponding tree is:



---

For each choice (branch) the TM checks by the usual truth table method. The proof that SAT is NP-complete is beyond the scope of the course.

---

(b)

The standard method for showing that a problem  $A$  is NP-complete is to use **Theorem 3.4.1.2**, i.e. we have that one typical problem (SAT) is NP-complete and by showing  $\text{SAT} \leq A$  we show  $A \in \text{NPC}$ .

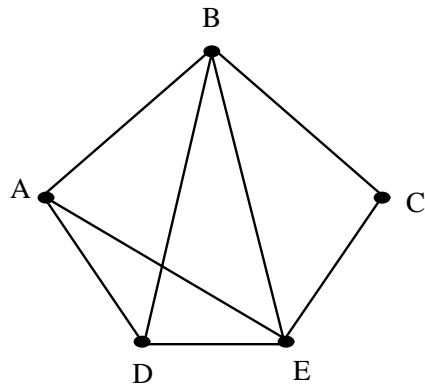
---

### 3.4.2.3. CLIQUE:

Suppose we have a graph  $G$  on a set of vertices  $V$ . A clique is a *complete* subgraph of  $G$  (any pair of the subgraph are joined by an edge).

---

For example suppose we have the graph:



---

Then  $ADE$  is a clique of size 3.  $ABDE$  is a clique of size 4. (Size is taken to be the number of vertices.)

The problem CLIQUE is: given  $G$  and a positive integer  $n \leq |V|$ , does  $G$  have a clique of size  $n$ ?

---

### 3.4.2.4. Theorem

CLIQUE is NP-complete.

#### **Proof:**

CLIQUE  $\in$  NP since we can construct a NDTM which will ‘non-deterministically’ choose a set of  $n$  vertices and then in polynomial time check whether or not they form a clique.

---

To show that CLIQUE is NP-complete (using Theorems 3.4.1.2 and 3.4.2.2) we find a polynomial reduction of SAT to CLIQUE.

---

Suppose we are given an instance of SAT:

$$(q_{1,1} \vee q_{1,2} \vee \dots \vee q_{1,m_1}) \wedge \dots \wedge (q_{n,1} \vee \dots \vee q_{n,m_n})$$

Each  $q_{i,j}$  is either  $p_s$  or  $\sim p_s$  for some propositional variable  $p_s$ .

---

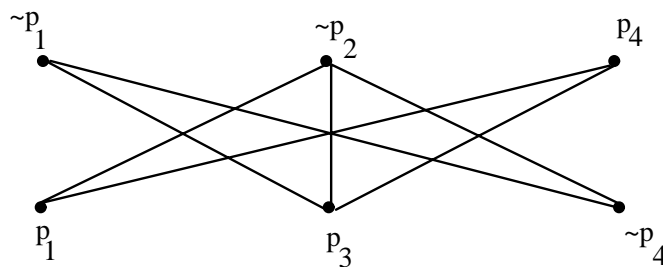
We associate with this instance a graph  $G$  with vertex set:

$$\{(1,1), (1,2), \dots, (1, m_1), (2,1), (2,2), \dots, (2, n_2), \dots, (k,1), (k,2), \dots, (k, n_k)\}$$

Vertices  $(i,j)$  and  $(i',j')$  have an edge between them if  $i \neq i'$ , and if  $q_{ij}$  and  $q_{i',j'}$  are not negations of each other. Suppose  $t$  is the total number of propositional variables then the graph has at most  $nt$  vertices, and can be defined in polynomially many steps.

---

We think of this graph as instances of literals (propositional variables or the negation of propositional variables). For example  $p_s$  may occur several times and each time it does it is counted as a separate vertex of  $V$ . Suppose we have the formula  $(\sim p_1 \vee \sim p_2 \vee p_4) \wedge (p_1 \vee p_3 \vee \sim p_4)$ . The corresponding graph is:



---

We now show that the formula is satisfiable iff  $G$  has a clique of size  $n$ . Assume we have an assignment of truth values making the formula true. These literal occurrences then define a clique in  $G$  of size  $n$  because each of the clauses (conjuncts) is true so for each  $i$ , some  $q_{ij}$  is true and these form the clique. The significant point is that no two can be contradictory so must be joined.

---

Conversely, assume that  $G$  has a clique of size  $n$ .

Literals in the same clause were not joined so just one literal was chosen from each clause.

We make all such literals true and this defines an assignment which makes the formula true.

Some variables will not have occurred in the clique but these can be given an arbitrary value.

---

Looking at our example ( $n=2$ ):

If we were given an assignment

$\{P_1:=T, P_2:=T, P_3:=T, P_4:=T\}$  then the clique

defined is either  $\{P_4, P_1\}$  or  $\{P_4, P_3\}$ .

---

Conversely if the clique were  $\{\sim P_1, P_3\}$  then any assignment making  $P_1$  false and  $P_3$  true ( $P_2, P_4$  arbitrary) will make the formula true.

We have that any formula in conjunctive normal form with  $n$  conjuncts is satisfiable iff the graph constructed as above has a clique of size  $n$ .

---

### 3.4.2.5. VERTEX COVER:

Let  $G$  be a graph. A set  $C$  of vertices of  $G$  is a vertex cover for  $G$  if every edge has an end point in  $C$ .

The problem is: Given  $G$  and  $k$  does  $G$  have a vertex cover of size  $k$ ?

---

### 3.4.2.6. Theorem

VERTEX COVER is *NP*-complete.

#### **Proof:**

We show  $\text{CLIQUE} \in \text{VERTEX COVER}$  and appeal to **Theorem 3.4.1.2**. (Actually, in this case it is easy to also show  $\text{VERTEXCOVER} \in \text{CLIQUE}$ ).

---

We first show that  $C$  is a vertex cover for  $G$  iff its complement is a clique for the graph  $\bar{G}$  (where  $\bar{G}$  has the same vertex set as  $G$  but  $x,y$  are joined in  $\bar{G}$  iff they are not joined in  $G$ ).

---

Assume  $C$  is a vertex cover for  $G$ . Let  $x, y$  be vertices of  $G$  not in  $C$ . Then there is no edge joining  $x, y$ . So they are joined in  $\bar{G}$ .

Hence  $G - C$  is a clique in  $\bar{G}$ .

---

Conversely, assume  $G-C$  is a clique in  $\bar{G}$ . Suppose  $x, y$  are vertices of  $G$  with an edge between them.

Then not both  $x, y \in G-C$ , so one is in  $C$ .  $\therefore C$  is a vertex cover for  $G$ .

---

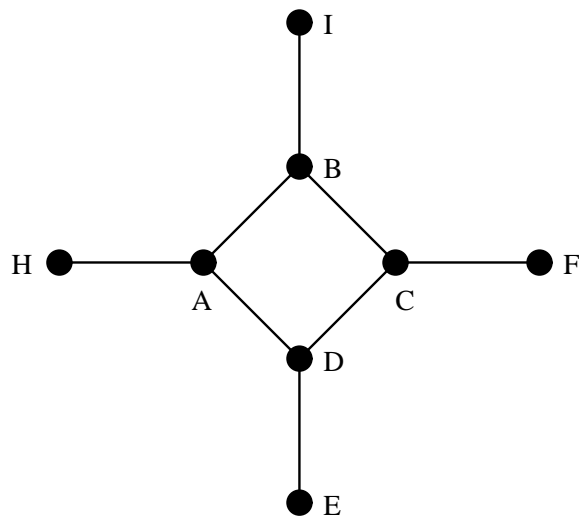
We can now polynomially reduce CLIQUE to VERTEX COVER as follows:

To decide whether or not  $G$  has a clique of size  $m$  we decide whether or not  $\overline{G}$  has a vertex cover of size  $|G| - m$ . (VERTEX COVER  $\propto$  CLIQUE can be shown similarly).

---

## Example

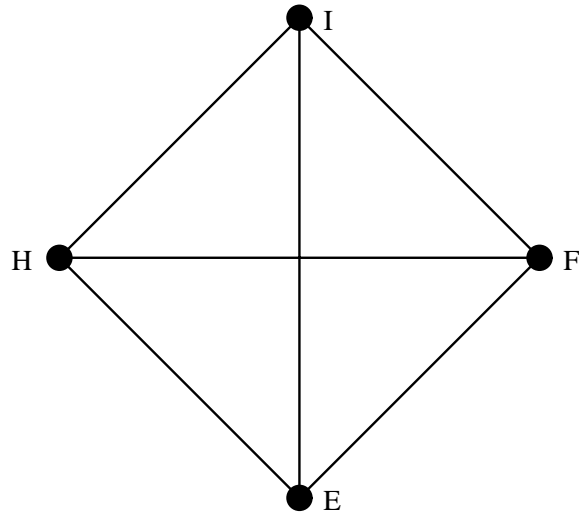
Suppose we have the graph given by:



$C = \{A, B, C, D\}$  is a vertex cover.

---

$G - C$  in  $\bar{G}$  is:



which is a clique of size 4.