

# Theory of Algorithms

---

ian.mcloughlin@gmit.ie

Racket

Turing machines

Computational complexity

NP completeness

# Racket

---



- Alonzo Church (1903 – 1995) at Princeton.
- Supervised Alan Turing's PhD (among others – Kleene).
- Introduced lambda calculus.
- Church–Turing thesis.



- John McCarthy while at MIT – late 1950s.
- Created Lisp.
- Lisp generally considered first functional programming language (not really though).
- Lots of dialects exist today, such as Scheme and Common Lisp.

## Wasteful for loops

*# How do you parallelise this?*

```
inds = list(range(10))  
total = 0  
for i in inds:  
    total = total + (i * 7)
```

```
def byseven(i):  
    return i * 7
```

```
inds = list(range(10))  
sevens = map(byseven, inds)  
total = sum(sevens)
```

# State

**Imperative** programming is a programming paradigm where statements are used to change the *state*.

**State** is the name given to the current data/values related to an executing process, including internal stuff like the call stack.

**Processes** begin with an initial state and (possibly) have (a number of) halt states.

**Statements** change the state.

**Functions** in imperative programming languages might return different values for the same input at different times, because of the state.

**Functional** programming languages (try to) not depend on state.

**Functions** are said to have side effects if they modify the state (on top of returning a value).

**Static and global** variables are often good examples of side effects in action.

**Functional** programming tries to avoid side effects.

**It's tricky** to avoid them – such as when we need user input.



## Basic operators

```
> (+ 3 4)
```

```
7
```

```
> (* 3 2)
```

```
6
```

```
> (- 5 3)
```

```
2
```

```
> (/ 6 3)
```

```
2
```

## More arguments

```
> (+ 3 4 5)
```

```
(+ 3 4 5)
```

```
> (- 3 4 5)
```

```
-6
```

```
> (* 2 3 4)
```

```
24
```

```
> (/ 6 3 3)
```

```
2/3
```

```
> (/ 6 3 3 3)
```

```
2/9
```

# Functions and values

*; Define a value called foo with value 3.*

```
>(define foo 3)
```

*; Define a function f.*

```
>(define (f x)
  (+ (* 3 x) 12))
```

```
>(define (g x)
  (* 3 (+ x 4)))
```

```
>(g 2)
18
```

# Conditionals

```
> (if (< 1 2) '(y e s) '(n o))  
(y e s)
```

```
>(define (abs x)  
  (if (< x 0)  
      (- x)  
      x))
```

```
>(list 1 2 3)  
(1 2 3)
```

```
>(list 'a 'b 'c)  
(a b c)
```

```
> (length (list 1 2 3))  
3
```

## car and cdr

```
> (car (list 1 2 3))  
1  
> (cdr (list 1 2 3))  
(2 3)  
> (define l (list 1 2 3))  
> (car l)  
1  
> (cdr l)  
(2 3)  
> (car (cdr l))  
2  
> (cadr l)  
2
```

# Recursion

```
> (define (sum lv)
  (if (null? lv)
      0
      (+ (car lv) (sum (cdr lv)))))
> (sum (list 1 2 3))
6
> (define (derange n)
  (if (= 0 n)
      '()
      (cons n (derange (- n 1)))))
> (derange 12)
(12 11 10 9 8 7 6 5 4 3 2 1)
```

## Looping recursively

```
> (let loop ((i 5))  
  (print "i is " i ".\n")  
  (if (> i 0) (loop (- i 1))))  
i is 5.  
i is 4.  
i is 3.  
i is 2.  
i is 1.  
i is 0.
```



```
> (define (swap3-1-2 x)
  (list (cadr x) (car x) (caddr x)))
```

```
> (swap3-1-2 (list 1 2 3))
(2 1 3)
```

```
> (define four-over-two (list 4 '/ 2))
```

```
> four-over-two
(4 / 2)
```

```
> (eval (swap3-1-2 four-over-two))
```

## More on functions

*; Printing stuff to terminal.*

```
> (print "Ay" "-" "yo.\n")
```

*; Proper way to define a function.*

```
> (define foo (lambda (bar) (print "Bar is " bar ".\n")))
```

*; Shorthand.*

```
> (define (foo bar) (print "Bar is " bar ".\n"))
```

*; Local variables.*

```
> (define (foo bar) (let ((thing "Bar"))  
  (print thing " is " bar ".\n")))
```

```
> (foo "open")
```

```
Bar is open.
```

## Function example

```
> (define l
  (let
    ((d 4) (e 5))
    (lambda (a b c) (list a b c d e))
  )
)
> (1 1 2 3)
(1 2 3 4 5)
```

```
> (cons 1 '())  
(1)  
> (cons 1 (cons 2 null))  
(1 2)  
> (cons 1 (cons 2 (cons 3 null)))  
(1 2 3)  
> (define mylist (cons 1 (cons 2 (cons 3 null))))  
> mylist  
(1 2 3)  
> (car mylist)  
1  
> (cdr mylist)  
(2 3)
```

## More on lists

```
> (list "a" "b" "c")  
("a" "b" "c")  
> (list a b c)  
reference to undefined identifier: a  
> (list 'a 'b 'c)  
(a b c)  
  
> (equal?  
(list 1 2 3)  
(cons 1 (cons 2 (cons 3 '()))))  
#t
```

# Quoting

```
> (list a b c)
*** ERROR IN (console)@1.7--Unbound variable: a
> (quote (a b c))
(a b c)
> (quote a b c)
*** ERROR IN (console)@2.1--Ill-formed special form: quote
> '(a b c)
(a b c)
> (define forty-two '(* 6 9))
> forty-two
(* 6 9)
> (eval forty-two)
```

54

## Null list

```
> ()  
missing procedure expression  
> (list)  
( )  
> '()  
( )  
> null  
( )  
> 'null  
null
```

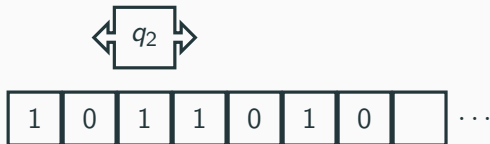
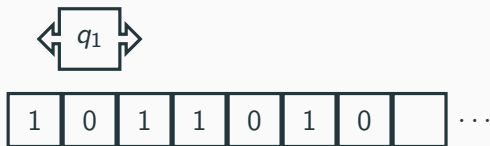
```
> (define (container value)
  (lambda ()
    (string-append "This container contains " value ".")))
> (define apple (container "an apple"))
> (define pie (container "a pie"))
> (apple)
"This container contains an apple."
> (apple)
"This container contains an apple."
> (pie)
"This container contains a pie."
```



# Turing machines

---

# Visualisation



## State Table

State	Input	Write	Move	Next
$q_0$	$\square$	$\square$	L	$q_a$
$q_0$	0	0	R	$q_0$
$q_0$	1	1	R	$q_1$
$q_1$	$\square$	$\square$	L	$q_f$
$q_1$	0	0	R	$q_1$
$q_1$	1	1	R	$q_0$

$$\delta(q_i, \gamma_n) \rightarrow (q_j, \gamma_m, L/R)$$

# Notation

$Q$  Set of states (finite).

$\Sigma$  Input alphabet, subset of  $\Gamma \setminus \{\sqcup\}$ .

$\Gamma$  Tape alphabet (finite).

$\sqcup$  Blank symbol, element of  $\Gamma$ .

$\delta$  Transition function,  $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ .

$q_0$  Start state,  $\in Q$ .

$q_a$  Accept state,  $\in Q$ .

$q_r$  Reject state,  $\in Q, \neq q_a$ .

$M$  Turing Machine:  $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ .

# Blank symbol

**Blank** is generally the only difference between the tape alphabet and the input alphabet.

**Definitions** of Turing machines generally don't disallow other symbols in the difference, but there's always at least a blank.

**Empty cells** of the tape in the machine are said to contain the blank symbol.

**Importantly** the blank symbol marks the end of the input on the tape.

**That's why** the input cannot contain the blank symbol.

# Sets and alphabets

**Recall** sets are just collections of objects called elements.

**Sets** have two important attributes – an object is either in the set or not, and all elements are distinct.

**Alphabets** in the definition of Turing machines are just sets, and their elements are called symbols.

**Strings** are finite sequences (i.e. ordered lists) of symbols over alphabets.

$\epsilon$  is the empty string.

$A^*$  is the set containing all of the strings over the alphabet  $A$ , including the empty string.

$|w|$  denotes the length of a string  $w$ , e.g. if  $w = xxyzxy$  then  $|w| = 6$ .

# Alphabets and strings

## Examples

The following are examples of strings over the alphabet  $\{0, 1\}$ :

- 100110
- 111
- 0
- $\epsilon$

## Single character strings

Note the distinction between a symbol in an alphabet and the string containing a single string. They look the same, but one is a symbol and one is a string. This is akin to the distinction in C between the character 'a' and the string literal "a".

**Language** is a set of strings.

**Turing machines** accept some strings as inputs.

**Accepted** language of a Turing machine is the set of strings it accepts.

**Halting** – given a string, a Turing machine will either accept it, reject it, or never stop (fail to halt).

**Decide** – a Turing machine that halts on all inputs is called a decider for the language it accepts.

**Turing-decidable** – a language that some Turing machine decides.



## Turing's Second Example

*As a slightly more difficult example we can construct a machine to compute the sequence 001011011101111011111 . . . . The machine is to be capable of five m-configurations, viz. o, q, p, f, b and of printing e, x, 0, 1. The first three symbols on the tape will be ee0; the other figures follow on alternate squares. On the intermediate squares we never print anything but x. These letters serve to keep the place for us and are erased when we have finished with them. We also arrange that in the sequence of figures on alternate squares there shall be no blanks.*

## Turing's Second Example: Table

<i>m-config.</i>	<i>symbol</i>	<i>operations</i>	<i>final m-config.</i>
b		<i>Pe, R, Pe, R, P0, R, R, P0, L, L</i>	o
o	1	<i>R, Px, L, L, L</i>	o
	0		q
q	Any (0 or 1)	<i>R, R</i>	q
	None	<i>P1, L</i>	p
p	x	<i>E, R</i>	q
	e	<i>R</i>	f
	None	<i>L, L</i>	p
f	Any	<i>R, R</i>	f
	None	<i>P0, L, L</i>	o

## Turing's Second Example: Initial state and tape

```
// The contents of the tape.  
var tape = [];  
// The current position of the machine on the tape.  
var pos = 0;  
// The current state.  
var state = b;
```

## Turing's Second Example: Reading and writing

```
// The blank symbol.
```

```
var blank = undefined;
```

```
// Write symbol sym to the current cell on the tape.
```

```
function write(sym) {  
    tape[pos] = sym;  
}
```

```
// Return true iff the symbol in the current cell is sym.
```

```
function read(sym) {  
    return sym == tape[pos] ? true : false;  
}
```

## Turing's Second Example: Moving the machine

*// Move the machine head right.*

```
function right() {  
    pos++;  
}
```

*// Move the machine head left.*

```
function left() {  
    pos--;  
}
```

## Turing's Second Example: State b

<i>m-config.</i>	<i>symbol</i>	<i>operations</i>	<i>final m-config.</i>
b		$P_e, R, P_e, R, P_0, R, R, P_0, L, L$	o

```
function b() {  
    write('e'); right();  
    write('e'); right();  
    write('0'); right(); right();  
    write('0'); left(); left();  
    state = o;  
}
```

## Turing's Second Example: State q

<i>m-config.</i>	<i>symbol</i>	<i>operations</i>	<i>final m-config.</i>
q	Any (0 or 1)	<i>R, R</i>	q
	None	<i>P1, L</i>	p

```
function q() {  
    if (read('0') || read('1')) {  
        right(); right(); state = q;  
    }  
    else if (read(blank)) {  
        write('1'); left(); state = p;  
    }  
}
```

# Computational complexity

---



# Bubble sort for fixed size input

How many comparisons does bubble sort make for different inputs of the same size?

- Suppose we have a list with 5 elements.
- Best case scenario: list already sorted.
- Worst case scenario: list reverse sorted.

[1, 2, 3, 4, 5]

[5, 2, 1, 4, 3]

[5, 4, 3, 2, 1]

## Bubble sort different sized inputs

How many comparisons does bubble sort make for inputs of increasing length?

- Suppose we have a list with  $n$  elements.
- Compare the worst case scenario for each value of  $n$ .

[3, 2, 1]

[4, 3, 2, 1]

[5, 4, 3, 2, 1]

[6, 5, 4, 3, 2, 1]

## Average vs. worst case

Input	Algorithm A	Algorithm B
(1,2,3)	1ms	1ms
(1,3,2)	1ms	5ms
(2,1,3)	2ms	4ms
(2,3,1)	2ms	5ms
(3,1,2)	2ms	5ms
(3,2,1)	10ms	4ms
Average	3ms	4ms
Worst	10ms	5ms

Would you choose Algorithm A or Algorithm B?

## How does the number of operations change as $n$ increases?

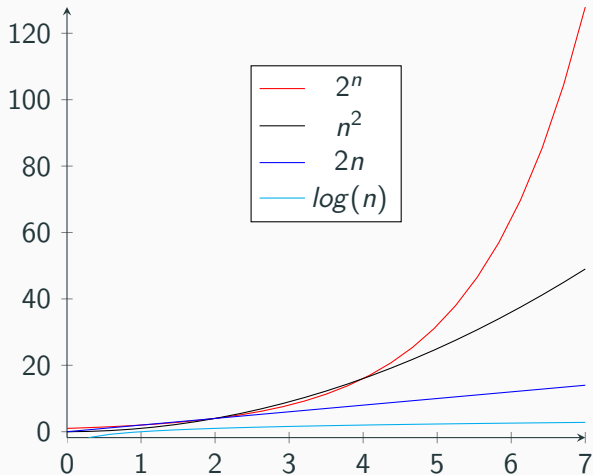
**Linear** Multiply  $n$  by a constant, add a constant. (Special category of polynomial time.)

**Polynomial** Multiply  $n$  by itself a fixed number of times, multiply by a constant. (Can add such terms together.)

**Exponential** Raise a constant to the power of  $n$ . (Rate of change is still exponential.)

**Logarithmic** Opposite of exponential.

## Terminology of complexity (graph)



$$f(n) = a_0 + a_1 n$$

## How many pairs of shoes does a centipede need?

- Let's say a centipede has 100 feet.
- Then every centipede needs 100 shoes.
- That's 50 pairs of shoes.
- So 2 centipedes need 100 pairs, 3 need 150 pairs, etc.
- So  $n$  centipedes need  $50n$  pairs of shoes.
- Linearity is familiar, and most people's default assumption.
- You take the input, multiply by a constant, and add another constant.

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots$$

## What is the volume of a cube of side $n$ ?

- Suppose we have a cube with sides of length 1 metre.
- The volume of the cube is  $1 \times 1 \times 1 = 1$  metres cubed.
- Suppose the cube has sides of length 2 metres instead.
- The volume of the cube is  $2 \times 2 \times 2 = 8$  metres cubed.
- In general, for sides of length  $n$ , the volume is  $n^3$ .

$$f(n) = a^n$$

## How many numbers can we represent with $n$ bits?

- Consider the case of four bits – imagine four placeholders

?	?	?	?
---	---	---	---

- Each placeholder can contain either 0 or 1.
- There are  $2 \times 2 \times 2 \times 2 = 2^4 = 16$  different numbers.
- Add another bit, how many numbers is it now?
- It's  $2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 32$ .
- Generally  $n$  bits can represent  $2^n$  numbers.



$$f(n) = \log_a n$$

## How many bits do we need to represent $n$ numbers?

- If we have  $n$  bits we can represent  $2^n$  numbers.
- If we want to represent  $n$  numbers, how many bits do we need (at a minimum)?
- The inverse operation to exponentiation is logarithm.
- Remember,  $a^n = b$  means  $\log_a b = n$ .

## Definition

An algorithm is said to be solvable in *polynomial time* if the number of steps required to complete the algorithm for a given input is  $O(n^k)$  for some nonnegative integer  $k$ , where  $n$  is the complexity of the input.

## **$P$ complexity class**

The  $P$  complexity class is the set of problems for which there exists, for each such problem, at least one algorithm to solve that problem in polynomial time.

# Polynomial time on a Turing machine

**Sorting** algorithms are usually compared in terms of comparisons.

**Other algorithms** might be compared in terms of something else, like iterations.

**With Turing machines** we can use the number of times we look up the state table.

**The size** of the input can be the length of the input on the tape initially.

# Recap on Languages

**Alphabet** Finite set of symbols, denoted  $\Sigma$ .

**String** Sequence of symbols,  $w$  from  $\Sigma$ .

**Language** Set of strings, denoted  $L$ .

**Length** Of a string, denoted  $|w|$ .

**Empty string** Unique string of length 0, denoted  $\epsilon$ .

# Kleene star

**Word concatenation:**  $w_1 w_2$  is the concatenation of strings  $w_1$  and  $w_2$ .

**String concatenation:**  $L_1 L_2$  is the language resulting from the concatenation of all strings in  $L_1$  and all words in  $L_2$ , in that order.

**Powers:**  $L^0 = \{\lambda\}$ ,  $L^1 = L$  and  $L^{n+1} = L^n L$  for all  $n > 1$ .

## Kleene Star

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Note that treating the alphabet  $\Sigma$  as a language in itself, we get that  $\Sigma^*$  is the set of all words over  $\Sigma$ .

## Example

$$\Sigma \{0, 1\}$$

$$L \{00, 01, 10, 11\}$$

$$w_1 \ 01$$

$$w_3 \ 11$$

$$w_1 w_3 \ 0111$$

$$\Sigma^* \{\lambda, 0, 1, 00, 01, 10, 11, 001, 010, \dots\}$$

$$L^* \{\lambda, 00, 01, 10, 11, 0000, 0001, \dots\}$$

$$L^+ \{00, 01, 10, 11, 0000, 0001, \dots\}$$

# Decision problems

**Decision problems** are problems where the answer is 0 or 1.

**Restricting** ourselves to decision problems is convenient and fair.

$f : \{0, 1\}^n \rightarrow \{0, 1\}$  is useful notation for considering decision problems.

**Turing machines** model decision problems by deciding languages.

## Definition

$$PRIMES = \{2, 3, 5, 7, 11, 13, \dots\}$$

**PRIMES** is the set of all primes numbers.

**Can** a Turing Machine be designed to decide PRIMES?  
(Yes)

**Some** people say PRIMES is the decision problem for the  
set of primes.

**Can** it do it in polynomial time? (Yes)

**Is** PRIMES in P? (Yes, 2002)



**Cryptography** depends on primes numbers, in the main.

**Asymmetric** cryptography depends on two facts:

- It's easy to find primes ( $P$ ).
- It's hard to decompose a composite number into primes (Not known to be  $P$ ).

# NP completeness

---

# Non-deterministic polynomial time

## Definition

A problem is in the  $NP$  complexity class if it is solvable by a non-deterministic Turing Machine in polynomial time. A non-deterministic Turing Machine is one which may not have a unique action to take for some or all states and inputs.

### **$P$ is a subset of $NP$**

The  $P$  complexity class is a subset of  $P$  because all polynomial time solvable problems can be modelled using Nondeterministic Turing Machines.

## Definition

An problem is *NP*-hard if each problem in *NP* can be reduced to it in polynomial time.

### *NP*-hard problems are hard

*NP*-hard problems are at least as hard to solve as the hardest *NP* problems. Note that *NP*-hard problems don't have to be *NP*.

## Definition

An problem is *NP*-complete if it's in *NP* and in *NP*-hard.

# Subset sum problem

## Problem

Given a set of integers  $S$ , is there a non-empty subset whose elements sum to zero?

## Example

Does  $\{1, 3, 7, -5, -13, 2, 9, -8\}$  have such a subset?

## Note

If somebody suggests a solution, it is very quick to check it. Being able to quickly verify a solution is a characteristic of *NP* problems.

# Propositional logic

**Literals** are Boolean variables (can be True or False), and their negations. They are represented by lower case letters like  $a$  and  $x_i$ .

**Clause** are expressions based on literals, that evaluate as True or False based on the literals. We use not, and and or on the literals. We'll sometimes call them expressions.

**Not** is depicted by  $\neg$ . So “not  $a$ ” is denoted by  $\neg a$ .

**Or** is depicted by  $\vee$ . So “ $a$  or  $b$ ” is denoted by  $a \vee b$ .

**And** is depicted by  $\wedge$ . So “ $a$  and  $b$ ” is denoted by  $a \wedge b$ .

**CNF** A clause is in Conjunctive Normal Form if it is a  
“conjunction of disjunctions”:  $(a \vee b) \wedge (\neg a \vee c) \wedge d$ .

**DNF** A clause is in Disjunctive Normal Form if it is a  
“disjunction of conjunctions”:  $(a \wedge b) \vee (\neg a \wedge c) \vee d$ .

### Converting to CNF and DNF

Every Boolean expression can be converted to CNF, and every Boolean expression can also be converted to DNF.

## Four laws

The following four laws can be used to convert expressions to CNF and DNF. The first two are known as De Morgan's laws, and the latter two are called the distributivity laws.

### Conversion laws

$$\neg(a \vee b) = \neg a \wedge \neg b$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$c \wedge (a \vee b) = (c \wedge a) \vee (c \wedge b)$$

$$c \vee (a \wedge b) = (c \vee a) \wedge (c \vee b)$$



# Boolean Satisfiability Problem (SAT)

The problem is the prototypical NP-complete problem. Note that it's quick to check the correctness of a solution for a given expression.

## Only using CNF

All Boolean expressions can be converted in CNF. However, it's not always possible to do that in polynomial time. We can though, in polynomial time, create new expressions using some extra(neous) variables that are satisfiable if and only if the original expression is.

## ***k*-SAT**

*k*-SAT is like SAT except that all expressions must be in CNF and each clause must be a disjunction of *k* literals.

**2-SAT**  $(a \vee b) \wedge (\neg c \vee d) \wedge \dots$

**3-SAT**  $(a \vee b \vee c) \wedge (\neg c \vee d \vee \neg a) \wedge \dots$

**2-SAT** is not NP-complete. There are polynomial time algorithms that solve it.

**3-SAT** is NP complete.

## 3-SAT is NP-Complete

### Reduction

3-SAT is a special case of SAT, so 3-SAT must be in NP. We can reduce SAT to 3-SAT in polynomial time. First take the expression and convert it to a CNF expression (in polynomial time). Then we just need to convert each clause into a CNF expression with 3 literals per clause.

Suppose we have a clause with 1 literal:  $a$ . Convert this to  $(a \vee u_1 \vee u_2) \wedge (a \vee u_1 \vee \neg u_2) \wedge (a \vee \neg u_1 \vee u_2) \wedge (a \vee \neg u_1 \vee \neg u_2)$ .  
Suppose we have a clause with 2 literals:  $a \vee b$ . Convert this to  $(a \vee b \vee u_1) \wedge (a \vee b \vee \neg u_1)$ .

Now suppose we have a clause with  $n$  literals:  $a \vee b \vee c \vee \dots$   
Convert this to  $(a \vee b \vee u_1) \wedge (c \vee \neg u_1 \vee u_2) \wedge \dots \wedge (i \vee \neg u_{n-4} \vee u_{n-3}) \wedge (j \vee k \vee \neg u_{n-3})$ .