Theory of Algorithms

ian.mcloughlin@gmit.ie

Topics

Racket

Turing machines

Computational complexity

NP completeness

Racket

Alonzo Church

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

wikipedia.org/wiki/Alonzo_Church

John McCarthy

www-formal.stanford.edu/jmc/

- 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

```python
# 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.

Side effects

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

```scheme
; 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
```

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

Conditionals

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

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

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

Lists

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

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

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

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

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
```

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

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)
```

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

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.
```

gambitscheme.org/wiki/images/a/a7/A_Tour_of_Scheme_in_Gambit.pdf

Data and code

```
> (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))
```

www.artificialworlds.net/presentations/scheme-01-intro/scheme-01-intro.html

More on functions

```scheme
; 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.
```

gambitscheme.org/wiki/images/a/a7/A_Tour_of_Scheme_in_Gambit.pdf

Function example

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

gambitscheme.org/wiki/images/a/a7/A_Tour_of_Scheme_in_Gambit.pdf

cons

```
> (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)
```

www.artificialworlds.net/presentations/scheme-02-basics/scheme-02-basics.html

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
```

www.artificialworlds.net/presentations/scheme-02-basics/scheme-02-basics.html

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
```

www.artificialworlds.net/presentations/scheme-05-quotation/scheme-05-quotation.html

Null list

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

www.artificialworlds.net/presentations/scheme-05-quotation/scheme-05-quotation.html

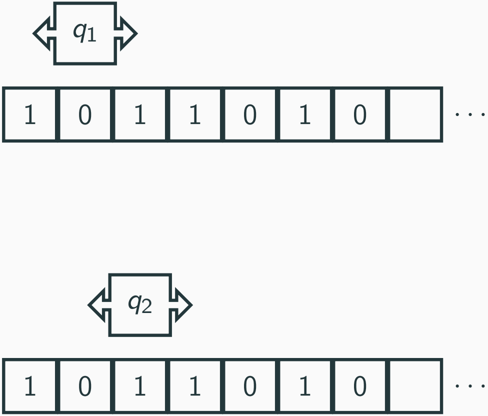
Closures

```
> (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."
```

gambitscheme.org/wiki/images/a/a7/A_Tour_of_Scheme_in_Gambit.pdf

Turing machines

Visualisation



State Table

State	Input	Write	Move	Next
q_0	$\sqcup$	$\sqcup$	L	q_a
q_0	0	0	R	q_0
q_0	1	1	R	q_1
q_1	$\sqcup$	$\sqcup$	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).
- Σ Input alphabet, subset of $\Gamma \setminus \{\sqcup\}$.
- Γ Tape alphabet (finite).
- $\sqcup$ Blank symbol, element of Γ.
- δ Transition function, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
- 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.

ϵ is the empty string.

$\mathbb{A}^*$ 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
- ϵ

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".

Languages

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

We can construct a machine to compute the sequence

$$001011011101111011111\ldots$$

The machine is to be capable of five m-configurations, viz. $\mathfrak{o}$, $\mathfrak{q}$, $\mathfrak{p}$, $\mathfrak{f}$, $\mathfrak{b}$ *and of printing* a, x, 0, 1*. The first three symbols on the tape will be* $aa0$*; 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

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{b}$		$Pe, R, Pe, R, P0, R, R, P0, L, L$	$\mathfrak{o}$
$\mathfrak{o}$	1	R, Px, L, L, L	$\mathfrak{o}$
	0		$\mathfrak{q}$
$\mathfrak{q}$	Any (0 or 1)	R, R	$\mathfrak{q}$
	None	$P1, L$	$\mathfrak{p}$
$\mathfrak{p}$	x	E, R	$\mathfrak{q}$
	e	R	$\mathfrak{f}$
	None	L, L	$\mathfrak{p}$
$\mathfrak{f}$	Any	R, R	$\mathfrak{f}$
	None	$P0, L, L$	$\mathfrak{o}$

Turing's Second Example: JavaScript 1

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

Turing's Second Example: JavaScript 2

```
// Writes a symbol to the current cell on the tape.
function write(sym) {
  tape[pos] = sym;
}

// Returns true iff the current cell contains sym.
function read(sym) {
  return sym == tape[pos] ? true : false;
}
```

Turing's Second Example: JavaScript 3

```
// Erases the symbol in the current cell of the tape.
function erase() {
  delete tape[pos];
}

// Returns true iff the current cell is blank.
// Returns true iff the current cell is blank.
function blank() {
  return typeof(tape[pos])
    == 'undefined' ? true : false;
}
```

Turing’s Second Example: JavaScript 4

```
function b() {
  write('e');
  pos++;
  write('e');
  pos++;
  write('0');
  pos++;
  pos++;
  write('0');
  pos--;
  pos--;
  state = o;
}
```

Computational complexity

Languages

Alphabet Finite set of symbols, denoted Σ.

Word Sequence of symbols, w from Σ.

Language Set of words, denoted L.

Length Of a word, denoted $|w|$.

Empty word Unique word of length 0, denoted λ.

Kleene star

Words $w_1 w_2$ is the concatenation of words w_1 and w_2.

Languages $L_1 L_2$ is the language resulting from the concatenation of all words 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 Σ as a language in itself, we get that Σ^* is the set of all words over Σ.

Example

$$\begin{aligned}
\Sigma &\quad \{0, 1\} \\
L &\quad \{00, 01, 10, 11\} \\
w_1 &\quad 01 \\
w_3 &\quad 11 \\
w_1 w_3 &\quad 0111 \\
\Sigma^* &\quad \{\lambda, 0, 1, 00, 01, 10, 11, 001, 010, \ldots\} \\
L^* &\quad \{\lambda, 00, 01, 10, 11, 0000, 0001, \ldots\} \\
L^+ &\quad \{00, 01, 10, 11, 0000, 0001, \ldots\}
\end{aligned}$$

Exponential

How many numbers can we represent with k bits?

- Consider the case of four bits.
- Imagine four placeholders: $\boxed{\textbf{?}}\boxed{\textbf{?}}\boxed{\textbf{?}}\boxed{\textbf{?}}$
- 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 k bits can represent 2^k numbers.
- This grows **exponentially** relative to k.

Polynomial time

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.

mathworld.wolfram.com/P-Problem.html

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 solvabled problems can be modelled using Nondeterministic Turing Machinses.

mathworld.wolfram.com/P-Problem.html

Decision problems

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

Restricting ourselves to decision problems is convenient and fair.

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

Other problems can be easily (polynomial time) adapted into decision problems.

NP completeness

NP-complete problems

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 soltuion, 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 $a \wedge b$.

Normal forms

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 cluse 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.

CNF SAT

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

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 $(a \vee b) \wedge (\neg c \vee d) \wedge \ldots$

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

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 $(a \vee b \vee u_1) \wedge (c \vee \neg u_1 \vee u_2)) \wedge \dots \wedge (i \vee \neg u_{n-4} \vee u_{n-3}) \wedge (j \vee k \vee \neg u_{n-3})$.