Theory of Algorithms

ian.mcloughlin@gmit.ie

Topics

Functional programming

Turing machines

Computational complexity

Exponential

How many numbers can we represent with k bits?

- Consider the case of four bits.
- Imagine four placeholders: $\boxed{?}\boxed{?}\boxed{?}\boxed{?}$
- 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.

NP complete problems

Permutations

Permutations

Permutations are rearrangements of ordered collections of items.

Example: "abcd" is a rearrangement of "bacd".

Think of having a four boxes, where we have to place one of the items in each box.

What are all the different ways of doing this?

What are all the different ways of associating items with boxes?

Abstraction

We can consider permutations in abstraction. For instance, if we have four items to rearrange, we can label the first item 1, the second 2, and so on. Then we can represent the various permutations, in terms of the numbers associated with the items. The permutations "abcd" and "bacd" could be represented by:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$$

In this way we can consider permutations in their own right.

Counting permutations

With four items, how many distinct permutations are there?
Consider having four placeholders where we can place the items:

$$\boxed{?}\ \boxed{?}\ \boxed{?}\ \boxed{?}$$

When we place an item in the first box, we have four choices, then we are left with only three choice for the second, two choices for the third, and one choice (i.e. not a choice at all) for the last.

So for there are $4 \times 3 \times 2 \times 1 = 4!$ choices in total.

Counting anagrams

Repitition

What happens when we consider two of our items to be the same? For instance, what if we are looking for all distinct rearrangements of “aacd” as opposed to “abcd”? In that case we need to account for the rearrangements of those items by dividing by the factorial of the number of times each item is repeated: $\frac{4!}{2!}$.

If more than one item is repeated a number of times we just keep dividing by the factorials of the numbers of repititons. The distinct number of rearrangements of “aaabbcd” is $\frac{7!}{3!2!}$.

Exercise

Calculate the number of distinct rearragements of the word “Mississippi”.

Heap’s algorithm

Heap	published an algorithm in November 1963 for generating permutations.
Published	in The Computer Journal.
Read	the article in the link below – it’s an easy read.
Pairs	of items are interchanged to generate each new permutation.
Induction	is used to show the algorithm works.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Heap's algorithm description

- Suppose we know how to permute $(n-1)$ items.
- That is, we know all of the different ways of slotting $(n-1)$ items in $(n-1)$ boxes.
- Let's add another, n^{th} item, and another box, an n^{th} box.
- First, place the n^{th} item in the n^{th} box.
- Permute all the the other items, which you know how to do.
- Then swap another item with the n^{th} item.
- Again permute the items in boxes 1 to $(n-1)$.
- Swap the item in the n^{th} box with another, different item.
- Repeat until all items have been in box n.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Steinhaus-Johnson-Trotter algorithm

Johnson	published another algorithm in 1963 for generating permutations.
Attributed	to three people: Steinhaus, Johnson and Trotter.
See	the article in the link below – it's a trickier read.
Pairwise	– the algorithm can be done pair-wise, like Heap's.
Induction	is used to show the algorithm works.

www.ams.org/journals/mcom/1963-17-083/S0025-5718-1963-0159764-2/S0025-5718-1963-0159764-2.pdf

Steinhaus-Johnson-Trotter algorithm description

- Start with two items, 1 and 2, and generate their list of permutations 12 and 21.
- Use the two-item list to generate the three item list in the following way:
  - Place 3 at the right of the first element in the two-item list.
  - Then move 3 one place to the left continuosly until its on the left.
  - Then place 3 at the left of the next element in the two-item list.
  - move 3 one place to the right continuously until it reaches the right.
- Use the three item list to generate the four item list in the same way, and so on.

comjnl.oxfordjournals.org/content/6/3/293.full.pdf

Conundrum – Naive method

```
for permutation in permutations(letters):
  checkIfWord(permutation)
```

Conundrum – Quick method preparation

```python
worddict = {}

for word in dictionaryOfWords:
  sortword = sorted(list(word))
  hashword = hash(sortword)
  allWords = worddict.get(hashword, set())
  allWords.update({word})
  worddict[hashword] = allWords
```

Conundrum – Quick method checking

```
word = "conundrum"
sortword = sorted(list(word))
hashword = hash(sortword)

worddict.get(hashword, None)
```

Timing Algorithms

timeit command line

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])
10000 loops, best of 3: 27.5 usec per loop
$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

docs.python.org/3/library/timeit.html

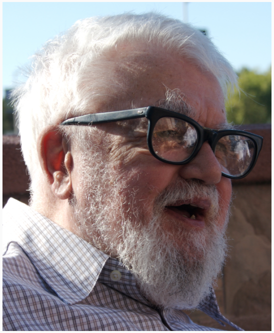
timeit module

```python
import timeit

def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()",
            setup="from __main__ import test"))
```

docs.python.org/3/library/timeit.html

Functional Programming

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

```
; 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

```
; 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

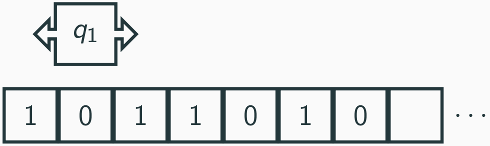
Closures

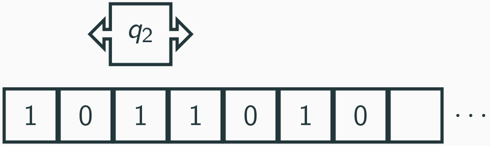
```
> (define (container value)
(lambda ()
(string-append "This container contains " value ".")))
> (define apple (container "an apple"))
> (define pie (container "a pie"))
> (apple)
"This container contains an apple."
> (apple)
"This container contains an apple."
> (pie)
"This container contains a pie."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

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

Turing Machines

Visualisation





Symbols

Q	Set of states (finite).
G	Tape alphabet (finite).
B	Blank symbol, element of G.
S	Input alphabet, subset of $G \setminus \{B\}$.
δ	Transition function.
q_0	Initial state, $\in Q$.
q_a	Accept state, $\in Q$.
q_r	Reject state, $\in Q$.
M	Turing Machine: $[Q, G, B, S, \delta, q_0, q_a, q_r]$.

State Table

State	Input	Write	Move	Next
	B	B		Accept
0	0	0	L	0
	1	1	L	1
	B	B		Fail
1	0	0	L	1
	1	1	L	0

$$\delta(q_i, g_n) \rightarrow (q_j, g_m, L/R)$$

Turing's Second Example

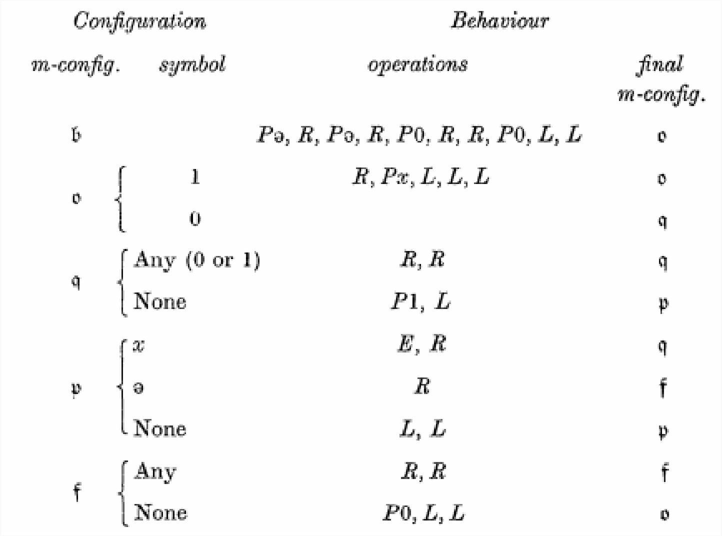
A slightly more difficult example

We can construct a machine to compute the sequence

$$001011011101111011111\ldots$$

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



Configuration		*Behaviour*	
m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{b}$		$P\text{ə}, R, P\text{ə}, R, P0, R, R, P0, L, L$	$\mathfrak{o}$
$\mathfrak{o}$	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	$\mathfrak{q}$
	ə	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;
}
```

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 $\lvert w \rvert$.
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

Σ	$\{0, 1\}$
L	$\{00, 01, 10, 11\}$
w_1	01
w_3	11
$w_1 w_3$	0111
Σ^*	$\{\lambda, 0, 1, 00, 01, 10, 11, 001, 010, \ldots\}$
L^*	$\{\lambda, 00, 01, 10, 11, 0000, 0001, \ldots\}$
L^+	$\{00, 01, 10, 11, 0000, 0001, \ldots\}$

Complexity Classes

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-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

$$\neg(a \lor b) = \neg a \land \neg b$$

$$\neg(a \land b) = \neg a \lor \neg b$$

$$c \land (a \lor b) = (c \land a) \lor (c \land b)$$

$$c \lor (a \land b) = (c \lor a) \land (c \lor b)$$

Boolean Satisfiability Problem (SAT)

The problem

We're often interested in knowing if there is any setting of the variables in a Boolean expression that makes the expression true. Another way of asking the question is: is the expression satisfiable? The real question is, is there a polynomial time algorithm that takes as input *any* Boolean expression and outputs true if there is any setting, and false otherwise.

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: $a \vee b \vee c \vee \ldots$. Convert this to $(a \vee b \vee u_1) \wedge (c \vee \neg u_1 \vee u_2)) \wedge \ldots \wedge (i \vee \neg u_{n-4} \vee u_{n-3}) \wedge (j \vee k \vee \neg u_{n-3})$.