Theory of Algorithms

ian.mcloughlin@gmit.ie

Topics

Racket

Turing machines

Computational complexity

NP completeness

Racket

Alonzo Church

- Alonzo Church (1903 – 1995) supervised PhDs of Alan Turing, Stephen Kleene, John George Kemeny, and many others.
- Introduced lambda calculus:

$$\lambda f.(\lambda x.f(xx))(\lambda.f(xx))$$

- Church–Turing thesis: lambda calculus represents all possible computations (so do Turing machines).

wikipedia.org/wiki/Alonzo_Church

John McCarthy

www-formal.stanford.edu/jmc/

- John McCarthy developed LISP at MIT in the late 1950s.
- Also famous for coining the term *artificial intelligence* at the Dartmouth workshop in 1956.
- Lisp generally considered first functional programming language: based on lambda calculus, limited side effects, first-class functions.
- Lots of dialects exist today, such as Scheme, Racket 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.

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
```

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

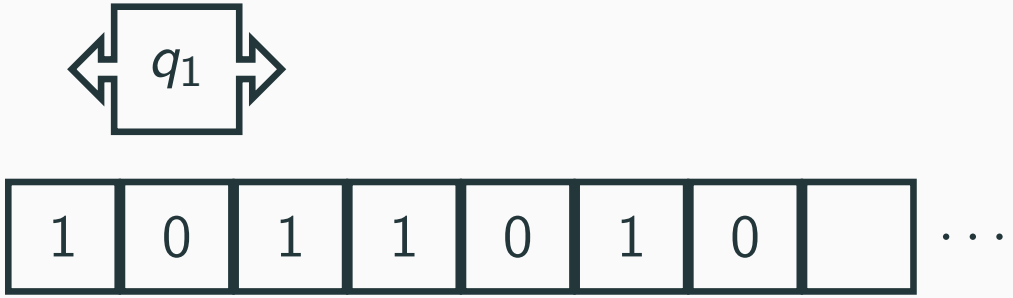
Closures

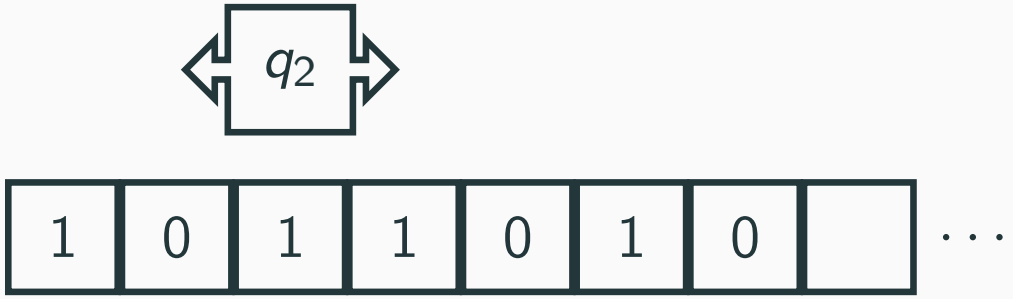
```
> (define (container value)
(lambda ()
(string-append "This container contains " value ".")))
> (define apple (container "an apple"))
> (define pie (container "a pie"))
> (apple)
"This container contains an apple."
> (apple)
"This container contains an apple."
> (pie)
"This container contains a pie."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

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

Turing machines

Visualisation





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

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

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, $\in Q$, $\neq q_a$

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 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

As a slightly more difficult example we can construct a machine to compute the sequence 00101101110111101111 . . . . *The machine is to be capable of five m-configurations, viz.* $\mathfrak{o}$, $\mathfrak{q}$, $\mathfrak{p}$, $\mathfrak{f}$, $\mathfrak{b}$ *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.*

www.cs.virginia.edu/ robins/Turing_Paper_1936.pdf

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: 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

m-config.	*symbol*	*operations*	*final m-config.*
𝔟		$Pe, R, Pe, R, P0, R, R, P0, L, L$	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

m-config.	*symbol*	*operations*	*final m-config.*
$\mathfrak{q}$	Any (0 or 1)	R, R	$\mathfrak{q}$
	None	$P1, L$	$\mathfrak{p}$

```
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 for 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?

Terminology of complexity

How does the number of operations change as n increases?

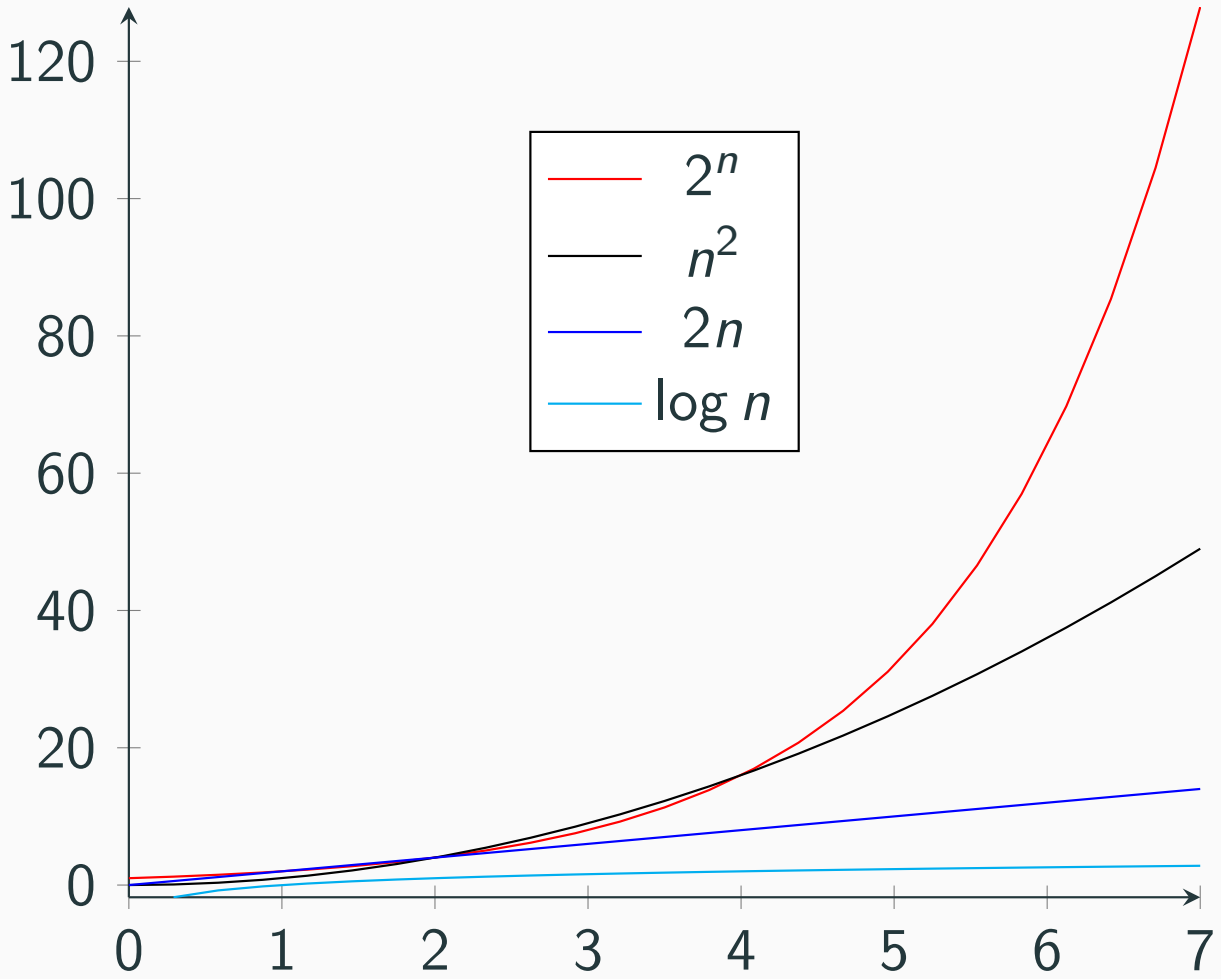
Linear Multiply n by a constant, add a constant. (Special category of polynomial time.)

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)



Linear

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

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.

Polynomial

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots$$

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.

Exponential

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

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.

Logarithmic

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

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 to we need (at a minimum)?
- The inverse operation to exponentiation is logarithm.
- Remember, $a^n = b$ means $\log_a b = n$.

Big-O (Sipser)

Definition

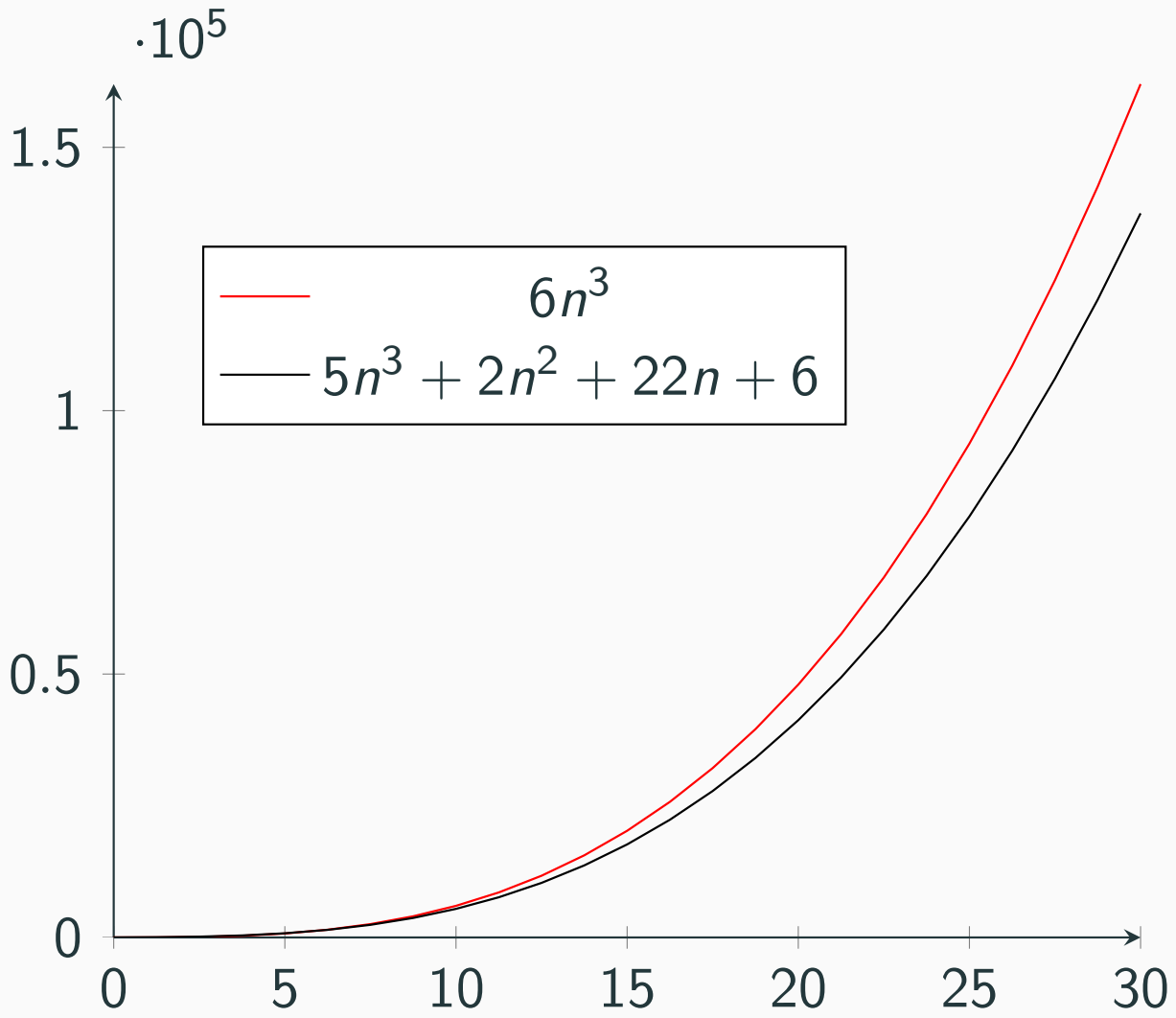
Let f and g be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$, or f is *big-O* of g, if positive integers c and n_0 exist such that, for every integer n greater than or equal to n_0, $f(n) \leq cg(n)$.

Example

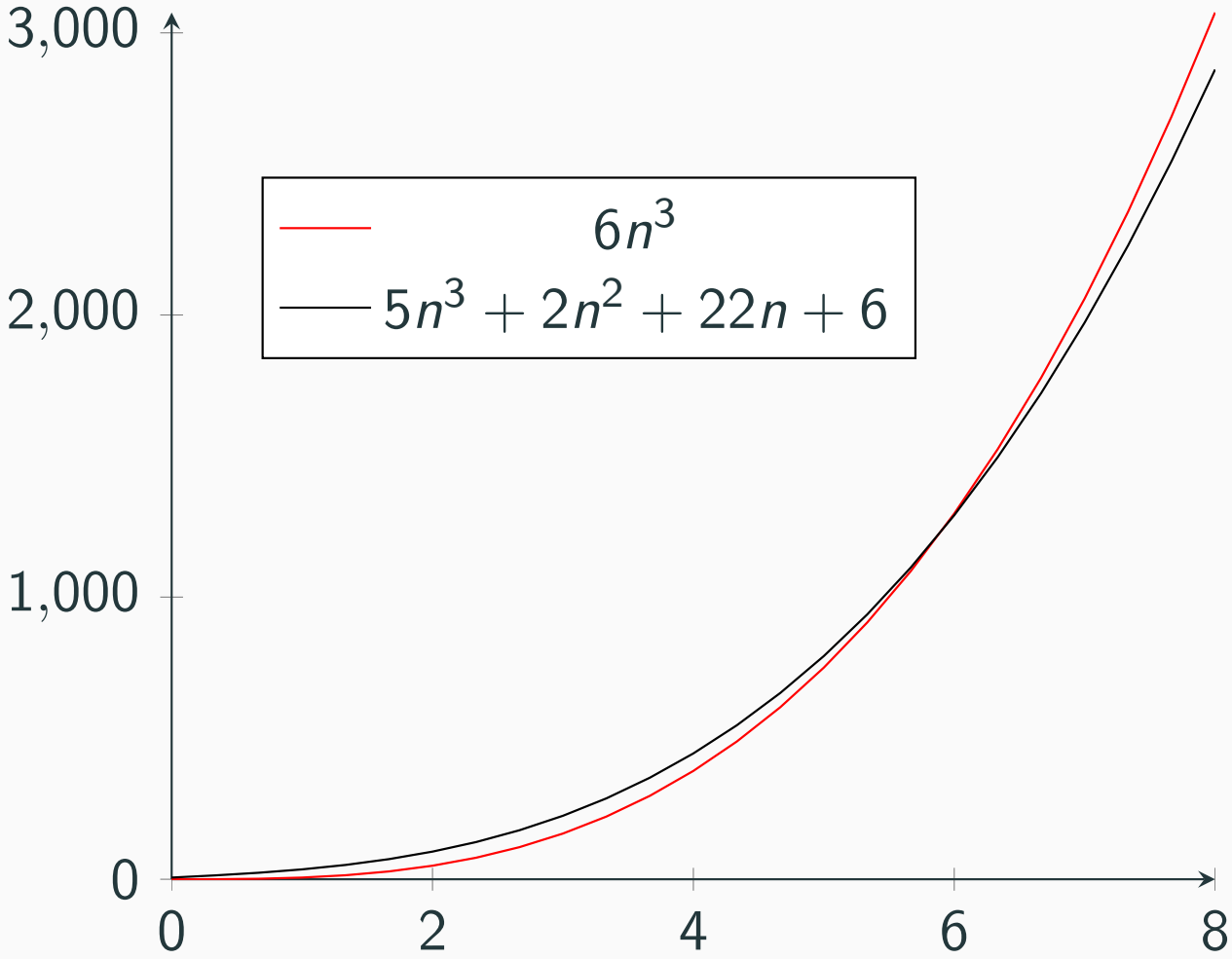
Let f be the function $f(n) = 5n^3 + 2n^2 + 22n + 6$. We'll prove that f is big-O of n^3 ($f = O(n^3)$). Let c be 6 and n_0 be 10. Is the following true, for all n greater than or equal to 10, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$? Note that as n increases ($n = 10, n = 11, n = 12, \ldots$), $f(n)$ also increases. Also note that $f(10) = 5426$ and $6g(10) = 6000$.

math.mit.edu/ sipser/book.html

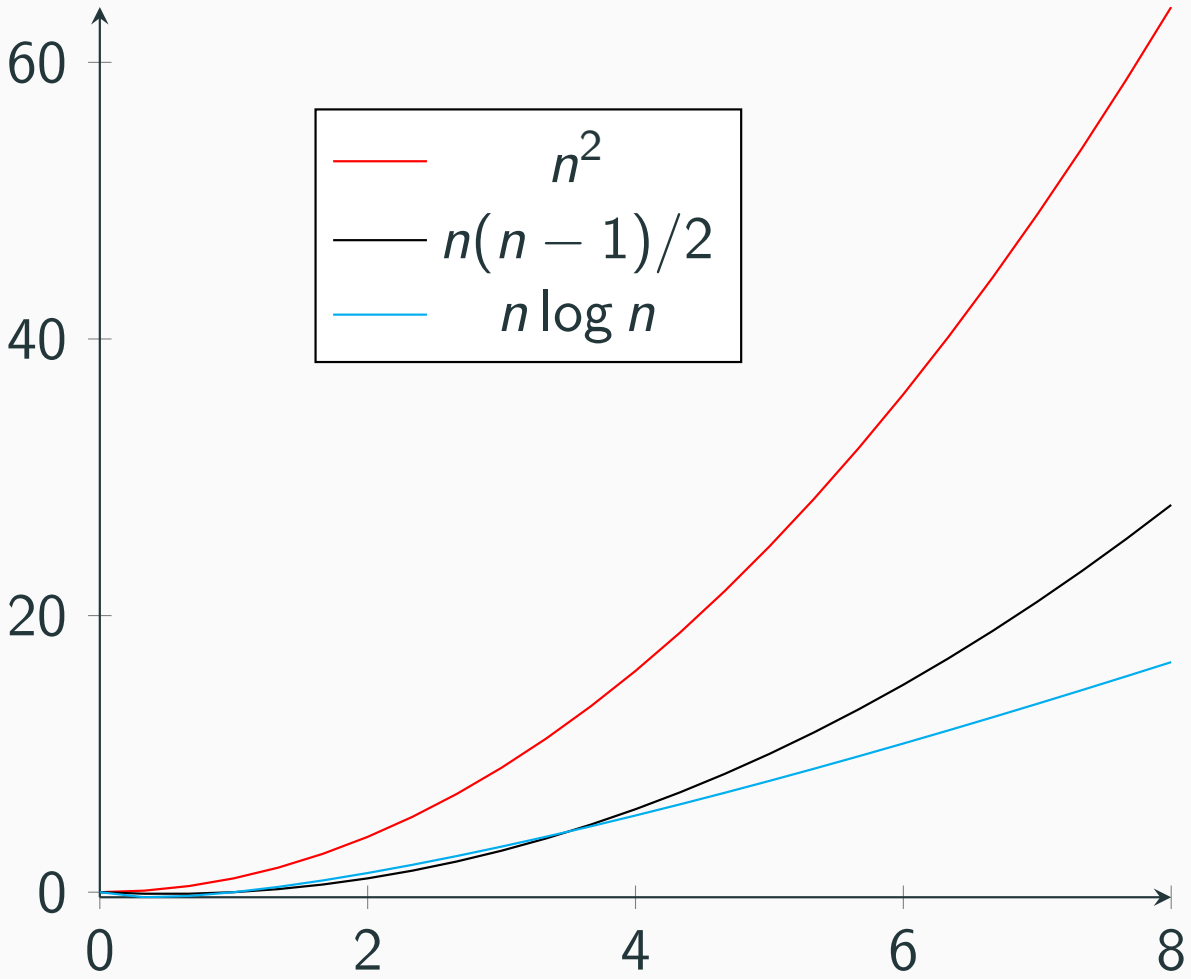
Big-O example graph



Smaller values of n



Bubble sort is $O(n^2)$



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.

Informally: 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.

P complexity class

The P complexity class is the set of languages for which there exists some Turing machine that decides the language in polynomial time.

Recap on Languages

Alphabet Finite set of symbols, denoted Σ.

String Sequence of symbols, w from Σ.

Language Set of strings, denoted L.

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

Empty string Unique string of length 0, denoted ϵ.

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 Σ as a language in itself, we get that Σ^* is the set of all words over Σ.

Example

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

Think about $\{0,1\}^*$

Consider the set:

$$\Sigma^* \quad \text{where} \quad \Sigma = \{0,1\}$$

Σ^* is the set of all strings (of all lengths) of 0's and 1's.

Documents on a computer are elements of Σ^*.

Programs on a computer are elements of Σ^*.

The entire contents of your hard drive is one big string of 0's and 1's, and so is in Σ^*.

Example of a text file

Create a text file on your computer with this single line of text:

```
Hello I'm a text file!
```

Then open it in a hex editor and see this:

```
48656C6C6F2049276D2061207465787420666696C6521
```

The hex editor is just displaying the binary as hex:

```
01001000011001010110110001101100011011110010
00000100100100100111011011010010000001100001
00100000011101000110010101111000011101000010
00000110011001101001011011000110010100100001
```

Decision problems

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

$$f : \{0,1\}^n \to \{0,1\}$$

Turing machines model decision problems by deciding languages.

Deciders are Turing machines that decide languages – they end in an accept or fail state no matter what the input, as opposed to never finishing.

Accept is 1, fail is 0.

Deciding Word documents

Can a Turing machine decide valid Word documents?

Word documents are just strings of 0's and 1's, elements of $\{0, 1\}^*$. Not all strings of 0's and 1's are valid Word documents.

If you try to open any old string of 0's and 1's in Word, it will likely tell you your file is corrupt.

Can we construct a Turing machine that accepts all valid Word documents, and fails otherwise?

Word seems to be able to decide what a valid Word document is, but it's always dealing with finite inputs. The specification might allow for infinite Word documents.

PRIMES

Definition

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

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)

www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf

Modern cryptography

Modern asymmetric key cryptography is based on prime numbers

It depends on two facts:

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

Generating versus verifying

Note that it's not necessarily easy to generate prime numbers. We know that verifying a number is prime can be done in polynomial time. That doesn't mean that we can generate prime numbers in polynomial time. You must start with the prime, and then ask the question.

Brute force prime checking

Is n a prime?

```
function is_prime(n) {
  for (var i = 2; i < n; i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

More efficient prime checking

- Suppose $a \times b = n$.
- Then $a < b$, $a > b$ or $a = b$.
- No matter what, $a \leq \sqrt{n}$ and/or $b \leq \sqrt{n}$.
- So only loop to $\sqrt{n}$.
- This still isn't that efficient.

Slightly more efficient

Is n a prime?

```
function is_prime(n) {
  for (var i = 2; i < Math.sqrt(n); i++) {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

NP completeness

Non-deterministic Turing machine

The usual Turing machines are often called deterministic Turing machines.

Deterministic Turing machines have exactly one row in their state table for every combination of (non-terminal) state and tape symbol.

This means there is only one path to follow at a given point in time.

Nondeterministic Turing machines can have any number of rows for each state/symbol (including none).

Essentially they allow for parallel computation – they can branch into two or more paths at the same time.

Non-deterministic Turing machine and languages

Languages are accepted by non-deterministic Turing machines, where an input string is accepted if any branch ends in the accept state.

Deciders – if a non-deterministic Turing machine always halts on all branches of computation, no matter what the input, then we say it decides the language it accepts.

Any language that is accepted (or decided) by a non-deterministic Turing machine has some deterministic Turing machine that accepts (or decides) it. So non-deterministic Turing machines don't really have any extra abilities over deterministic ones.

Non-deterministic polynomial time

Definition

A decision problem is in the NP complexity class if it is decidable by a non-deterministic Turing Machine in polynomial time.

P is a subset of NP

Note that every determinisitic Turing machine is also a non-deterministic one, by our definitions. The P complexity class is a subset of NP because of this.

Equivalent definition

An equivalent definition of NP that you may come across is that NP is the set of languages A that can be verified in polynomial time. By verified we mean that a deterministic Turing machine can accept a language $\{wc\}$ where w is in A and c is some string, called the certificate for w.

NP-complete problems

Definition

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

Reduction

Reduction is a way of converting one problem into another, so that a solution to one is a solution to the other. By reducing decision problem A to decision problem B, we mean that we can transform inputs to A into inputs to B in such a way that a given input to A is accepted iff the corresponding input to B is.

Definition

A problem is NP-complete if it's in NP and is 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.

SAT motivation

SAT is short for Boolean SATisfiability.

It is the archetypal NP-complete problem.

Informally it asks if there is a way to efficiently decide if an expression containing a bunch of Boolean variables combined using ands, ors, nots and brackets has any setting of the variables that makes the expression true.

SAT instance example

Consider the expression $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$, where x_1, x_2 and x_3 are true/false variables. Is there any setting of the variables that makes the expression true?

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 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)

SAT is the set of all Boolean expressions that are satisfiable.

Satisfiable expressions have some setting of their constituent variables that causes them to evaluate as true.

The SAT decision problem is the problem of deciding which Boolean expressions are in SAT.

Note that it's easy to verify that a given setting of the variables in an expression make it true.

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

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

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

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

SUBSETSUM and NP

SUBSETSUM is NP-complete.

2^n is the number of subsets. Note that 2^n is also the number of settings of n Boolean variables.

The correspondence can be seen in terms of 0's and 1's. In SUBSETSUM the elements from the set that are included in a given subset are represented by 1's.

The usual proof that SUBSETSUM is NP-complete is done by reduction to 3-SAT.