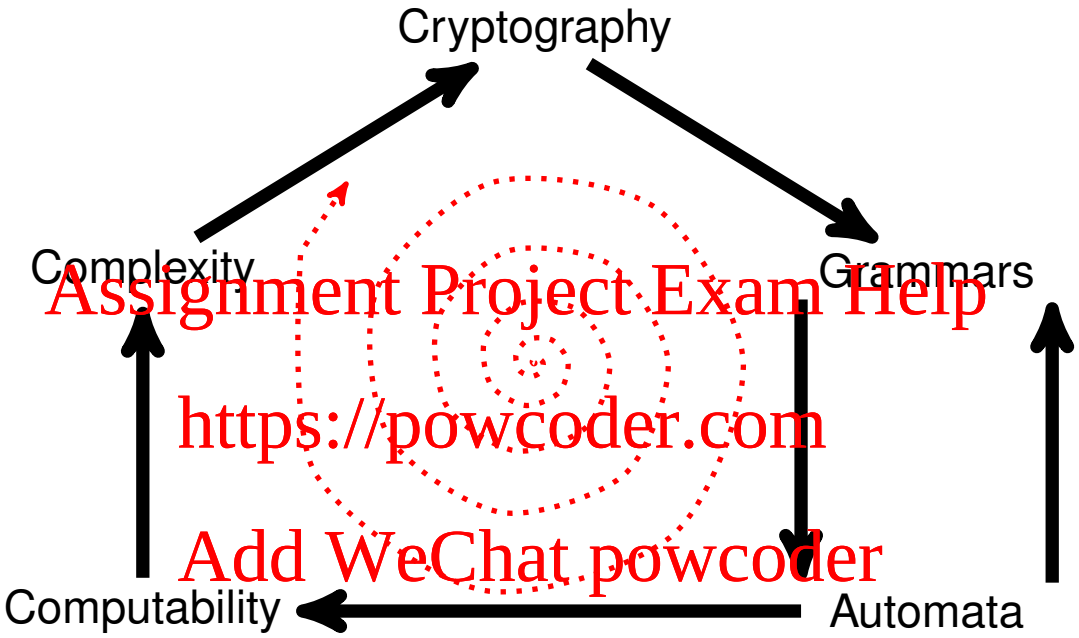
COMPUTING THEORY

COSC1107/1105: A Computable Odyssey

Course Material

Semester 2, 2020



Acknowledgements

The persons mainly responsible for assembling this material are James Harland and Sebastian Sardina.

However, there are a number of people who have contributed to the preparation of this material, including Nitin Yadav, Abhijeet Anand, Angus Kenny, Andy Kitchen, Alex Bowe, Ken Gardiner, Lin Padgham, John Thangarajah, Lawrence Cavedon, Simon Duff, Lavindra de Silva, Mark Ryall, Geoff Crompton, Kwong Yuen Lai, Scott Buszard, Daniel Patterson, and Joseph Antony.

Since 2012, this notes uses tikz LaTex drawing package (www.texample.net/tikz/).

Please send any corrections or comments to James Harland.

Last Edited: Tuesday 3rd August, 2021 14:41

Table of Contents

Some Initial Questions

Below are three problems, all to do with tiles, which will hopefully set the scene for this subject. How would you go about attacking these problems?

Tiling Problem 1:

Given a set of tile designs T, is it possible to cover *any* rectangular area using only designs from T? Rotations are **not** allowed, and the patterns on each of the four sides of a tile **must** match the neighbouring tiles.

Informally, can we tile *any* bathroom this way? (including some enormous examples as the bathrooms at Versailles, or the Taj Mahal, or ...)

Tiling Problem 2:

Given a set of tile designs T and a bathroom wall of size $s \times t$, is it possible to cover the wall using only designs from T? Rotations are **not** allowed, and the patterns on each of the four sides of a tile **must** match the neighbouring tiles.

Informally, can we tile *my* bathroom this way? This is clearly a more specific question that the earlier version.

Tiling Problem 3:

Given a set of tile designs T and a bathroom wall of size $s \times t$, is it possible to cover the wall using only designs from T? Rotations **are** allowed, but the patterns on each of the four sides of a tile must still match the neighbouring tiles.

Informally, can we tile my bathroom by turning the tiles around, if necessary?

Introduction

There are a number of topics, but there are five main ones:

- Complexity.
- Cryptography.[1]
- Grammars.
- Automata.
- Computability.

Complexity is concerned with measuring the cost of computations. There are many possible aspects to this; however, we concentrate on maximum execution time. Even then, it is not hard to find problems which cannot be solved in any reasonable time, no matter how fast hardware improves.

Cryptography is concerned with encrypting information so that no unauthorised person can read it. The main technical issue is how we can be sure of this; the answer is to rely on problems which no-one can solve in any reasonable time.

Grammars are concerned with the specifications of syntax rules — in particular, how do you indicate to a machine what programming language constructs are legal and which are not?

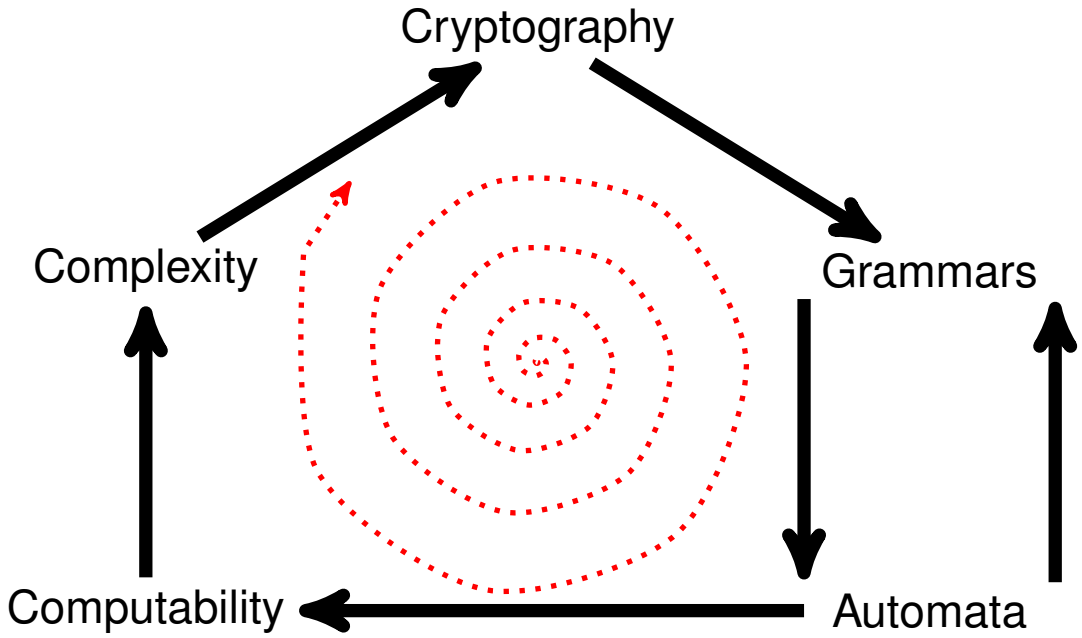
Automata are concerned with the modelling of computation, and in particular how we can reason about computation. In particualr, the key role of memory is emphasised in three different approaches to memory, being none (finite state machines), a single stack (a pushdown automaton) and a semi-infinite tape (a Turing machine).

Computability is concerned with the fundamental limits of computation (in contrast with the resources view used in complexity). In particular, it may be surprising to learn that there are some problems for which it can be proved that computers (at least in their current conception) cannot solve.

As may be noticed, there is an inter-dependence between these topics.

Also, observe that the order of the topics in this course notes may not correspond to the order delivered during the weeks. However, it should be straightforward to find the relevant section for each week.

[1] No longer covered in the course.



This picture depicts the general topics touched during the course. Notice, however, that **in the current course we do NOT include Cryptography**. Nonetheless, the section on Cryptography was left in the course notes for interest of the reader and as an example of when difficult computational problems are very useful.

We will mostly use a sequence compatible with several textbooks, starting with Automata and finishing with Complexity.

Specific Skills Covered

- analyze algorithmic complexity
- read and write automata
- NFA $\rightarrow$ DFA conversion
- read and write grammars
- parsing and derivations

Concepts

- decidability
- tractability and O(n) notation
- intractability and NP-completeness
- grammar and language classes
- automata relationships
- grammar relationships

Motivating Questions

- How can we reason about programs?
- What is possible and impossible?
- How can we tell the difference?
- How do we show that some things are impossible?
- Can we ever say "Never"? "Always"?
  (e.g. How do we reason about cryptographic attacks?)
- What *precisely* are programs anyway?
- How can we be precise about complexity costs?

> "Put the right kind of software into a computer, and it will do whatever you want it to. There may be limits on what you can do with the machines themselves, but there are no limits on what you can do with software." – Time, April, 1984.

- What limits are there to computation?
- How do we establish them?
- Can we do anything about such limits?
- What is an algorithm?
- Do algorithms always exist?
- What can and cannot be computed?
- What can be feasibly computed?

We can establish claims about "exists", "sometimes", "possible", etc. by **construction**, such as "there is at least one bearded lecturer at RMIT".

It is generally harder to establish claims about "all", "always", "impossible", etc., such as "Humans will never reach the moon".

"Never" is a very long word! Or, as Treebeard the Ent puts it "*Never* is too long a word, even for me". [2]

Overview Problem

The problem described below is intended as an overview of the material studied in this subject. You are **not** required to provide a solution to it; it is intended as a means of showing how the material fits into the "big picture". In particular, you will not be required to know any of the details of quantum computing.

Problem Statement

You are employed as an IT professional at a company called Incandescent Technologies. One day your boss comes into your office and says: "I hear that there is a new kind of computer called a quantum computer, which can apparently perform some computations much faster than those we have now. No such machines are currently commercially available, but we need to be prepared for the possibility that they will be in the near future. I want you to investigate this topic and prepare a comprehensive report on it, specifically addressing the points below:

1. Will the security of our systems be affected? Could some hacker utilise a quantum computer to break into our systems?
2. Can we harness the power of quantum computers to make our compilers run faster?
3. Are the benefits of quantum computing limited to faster processing alone? Or can we now solve problems which were unsolvable before?

Please, have your report on my desk by the end of the semester."

[2] J.R.R. Tolkien, 'The Lord of the Rings', p.1016, Harper-Collins, 1991.

Addressing this problem

In order to tackle this problem (which is (deliberately :-) very broad), you may find the following questions useful.

1. (a) How do modern-day cryptographic systems work?
   (b) What relationship is there between high complexity problems and cryptography?
   (c) How are factorisation and an algorithm for finding primes linked to cryptography?
   (d) How do we estimate the execution time of a program?
   (e) What are intractable problems and why are they important?
   (f) How much faster hardware do we need to solve problems with exponential complexity?
   (g) What approaches can we use to deal with difficult problems?
   (h) What are NP-complete problems and why are they important?
2. (a) How does parsing work?
   (b) What are grammars used for?
   (c) What are some different classes of grammars and how do they differ?
   (d) What can regular expressions express?
   (e) What are finite state machines and how do they work?
   (f) what are the differences between deterministic and non-deterministic finite state automata?
   (g) What are pushdown automata and how do they work?
   (h) What are turing machines and how do they work?
3. (a) Can we always find an algorithm to solve a problem?
   (b) What are the relationships between different machines and grammars?
   (c) What formal models of computation are there which are more powerful than Turing machines?
   (d) What is the most powerful kind of Turing machine?
   (e) Can we show without any doubt whether all problems could or couldn't be solved by computers? How?

Questions such as these will be discussed at lectures and answered by the problems set for each week of tutorials.

Tutorials

Whereas lectures are meant to give the "big picture" of each aspect of the course, tutorials are meant to deal with the *actual technical details*. Both lectures and tutorials are important, but **tutorials are more personal, and a tutor will expect students to attend them unless there is a very good reason not to**. **It is extremely difficult, if not impossible, to get the technical training required for passing the course without attending tutorials.**

During tutorials you will be working in **learning groups of around 2-4 people**. Your tutor will spend time with each group, assisting as necessary to ensure that you are on the right track. You should discuss and work actively together to ensure that everyone fully understands the subject material. In some tutorials you will have specific exercises to do — these are designed to assist you in mastering the various skills and concepts. Tutorial time can also be spent discussing assignments as well as material from lectures or the textbook. In the last part of each tutorial, **tutors will go over some questions of the previous tutorial in the whiteboard**, so make sure to attend and stay around until the end!

During each week you should:

- **Attend lectures** to get the general big picture of the material. Sometimes lectures provide essential core information not covered in tutorials.
- **Attend tutorials** to dive into the technical details and get the necessary training.
- After tutorials, **complete by yourself the remaining questions** that were left incomplete in the tutorial. You may want to collaborate with other fellow students. However, you are meant to develop yourself the solutions. Just knowing the solution (e.g., because another student explained it to us) means very little. Indeed, a moderate form of collaboration is encouraged as a useful part of any educational process. Nevertheless, students are expected to do their own thinking and writing. Never copy another student's work, even if the other student "explains it to you first." Never give your written work to others. Do not work together to form a collective solution, from which the members of the group copy out the final solution. Rather, walk away and recreate your own solution later.
- **Use the online forum** to help complete each tutorial as well as clarifying aspects covered in the lecture that you have not fully grasped.
- **Ask yourself**: *would I be able to solve similar exercise completely by myself?*

After each week's tutorial, it is a very good idea to write out the *complete solutions* to the problems set that week, particularly as for some tutorials your group may not complete all the problems in class. This will help you gather the content of the course up to the necessary details. This means that you can commence the following week's class with a brief discussion of the solutions, possibly by asking your tutor for feedback on your written work. Such regular clarification of your own understanding of the topic is the best form of preparation for the final exam.

Background Material

The material in this section is not examinable. However, you should make yourself familiar with this material, and it will be useful for reference.

Mathematical Formalism

To answer such questions, we use mathematical formalisms to describe computation.

- We can *reason* about the program and its behaviour.
- We can estimate the *cost* of the program.
- We can be rigorous and precise.
- We can (sometimes) establish "negative" results, e.g. "There is no algorithm to solve problem X".
- For this subject, mathematics is generally "read-only".

Mathematical Language

Definition A precise mathematical statement of what we mean, such as "A prime number is a positive integer which has no factors other than itself and 1".

Theorem/Proposition A property which we can show always holds and is interesting by itself, such as "All eger numbers except 2 are not prime" or "There are an infinite number of primes".

Lemma A property that always holds and that is used as a stepping stone to a larger result rather than as a statement of interest by itself.

Corollary A property that always holds and is interesting by itself, but that can be readily deduced from another result, generally a Theorem or Proposition.

Proof The justification for our statement. This *must* cover all cases and leave no room for doubt. Techniques used here include proof by *induction* and proof by *contradiction*.

Mathematical Methodology

We will often use mathematical methods to state precise results, usually in the form of a theorem and a proof. This requires us to

- write a precise summary,
- provide guaranteed behaviour (proof),
- concentrate on important parts,
- prove it can be applied elsewhere.

Two common techniques used for proofs are proof by induction and proof by contradiction.

Mathematical Induction

Let P be a property defined on N. How can we show that $\forall x \in N$, $P(x)$ is true? A generally applicable method is the *principle of mathematical induction*. If

1. $P(0)$ is true, and
2. if $P(n)$ is true, then $P(n+1)$ is true.

Then by the principle of mathematical induction, $\forall x \in N, P(x)$ is true. Note that step 2 is an abbreviation of

$P(0) \Rightarrow P(1)$
$P(1) \Rightarrow P(2)$
$P(2) \Rightarrow P(3)$
...

Proof by Contradiction

To prove a statement P, it is sometimes easier to assume its opposite, written as $\neg P$, and derive a contradiction.

Key Idea: if $\neg P$ cannot hold, then P must be true.
Step 1 Assume $\neg P$ is true.
Step 2 Based on this assumption draw a number of conclusions.
Step 3 Establish that the conclusions lead to a contradiction.
Step 4 Therefore $\neg P$ cannot be true (as was assumed).
Step 5 Therefore P is true.

Limits of Computation

Hardware speeds currently tend to double every two years (Moore's Law).

"Computers can do *anything*!"

- Some problems cannot be solved *even in principle* (i.e. given unlimited resources)
- Some problems are too difficult to be solved in practice (i.e. the needed resources are too vast)
- Some problems are too difficult to solve on a given hardware platform.

Some Example Problems

You may like to consider which category these problems fall into:

Problem 1 Write a program which finds the cheapest route to visit a given list of cities. Assume you have a list of all cities and the cost of flying between each two cities. The program should take as input a start point and a list of cities to visit.

Problem 2 Write a program which determines whether another program terminates on all possible inputs (i.e. is there an input that puts the program into an infinite loop?).

Problem 3 Write a program to index and make available all full text articles from the last 10 years of The Age newspaper.

Random Number Generation

A nice example of the need to think things through before coding is that of *random number generation*. A sequence of seemingly random numbers has many applications.

- Simulation
- Program testing
- Numerical analysis
- Decision making and scheduling
- Games (Tattslotto Quickpick)

Many of these problems can be mapped into the problem of generating a random sequence of integers between 0 and a given number M.

- Random 10 letter words: sequences of length 10 between 0 and 25. 0 becomes 'a', 1 becomes 'b' etc. ($n-1$ becomes nth letter of alphabet).
- Random arrival times, a sequence of real numbers in the interval $[1, 10)$: for $0 \leq X < M$, use $Y = 1 + 9(X/M)$.
- Random words from a dictionary: if there are N items in the dictionary, use the nth word, where $n = \lceil 1 + N(X/M) \rceil$.

Problem: To generate a sequence of seemingly random integers between 0 and M from a computer program.

Examples:
23, 654, 1, 34, 787, 233, 4232, ...
3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, ...
Non-examples:
1, 2, 3, 4, 5, ...
1, 4, 9, 16, 25, 36, ...

"Spaghetti attempt"

This was an attempt by Don Knuth to write a program tha was as unpredictable as possible, and hence would (hopefully) produce a sequence of random numbers.

Input: A 10-digit number X
Output: Next number in the sequence
K1 $Y \leftarrow \lfloor X/10^9 \rfloor$
K2 $Z \leftarrow \lfloor X/10^8 \rfloor \bmod 10$. Goto Step K$(Z+3)$
K3 If $X < 5 \times 10^9$ then $X \leftarrow X + (5 \times 10^9)$
K4 $X \leftarrow \lfloor X^2/10^5 \rfloor \bmod 10^{10}$
K5 $X \leftarrow (1001001001 \times X) \bmod 10^{10}$
K6 If $X < 10^8$ then $X \leftarrow X + 9814055677$, else $X \leftarrow 10^{10} - X$
K7 Swap the lower 5 digits of X with the higher 5 digits
K8 Same as K5
K9 Decrease each non-zero digit of X by one
K10 If $X < 10^5$ then $X \leftarrow X^2 + 99999$, else $X \leftarrow X - 99999$
K11 If $X < 10^9$ then $X \leftarrow 10 \times X$. Goto K11
K12 $X \leftarrow \lfloor X(X-1)/10^5 \rfloor \bmod 10^{10}$
K13 If $Y > 0$ then $Y \leftarrow Y - 1$; Goto K2

The hope was that starting with a number such as 1234567890 and repeatedly applying this algorithm would produce a random sequence of numbers. Unfortunately, what came out was a sequence that soon just repeatedly produced the number 6065038420 — not very random at all!

Moral (Knuth): Random numbers should not be generated with a method chosen at random. Some theory should be used.

Desirable properties of a random sequence:

- each integer $0, 1, \ldots, M-1$ occurs equally often,
- each pair, triple, etc. occurs consecutively equally often,
- the sequence should have mean $(M-1)/2$ and variance $M^2/12$, and
- successive numbers should be independent.

Linear congruential generator

One method to do this is to use a *linear congruential generator*.

$$X_{n+1} = (a * X_n + c) \text{ mod } M, \text{ where } a, c \leq M \text{ and } 0 \leq X_n < M.$$

Some good choices for a and b are (see Knuth's book for details):

1. $M = 65535\ (= 2^{16} - 1)$, $a = 25173$, $c = 13849$
2. $M = 262143\ (= 2^{18} - 1)$, $a = 4229$, $c = 56687$

A sample program:

```
int random(int);

main() {
    int seed = 12345;
    int num = 100;

    while (num-- > 0) {
        seed = random(seed);
        printf("%d\n", seed);
    }
}

int random(int seed){
    return(((25173*seed + 13849) % 65535));
}
```


Generated Numbers:

7564, 43246, 43522, 44560, 25669, 4486, 23122, 47620, 51424, 62881, 50407, 20590, 9604, 16726, 60607, 19060, 29494, 20296, 14197, 32575, 50404, 10606, 9097, 33340, 40459, 8821, 32302, 59350, 30004, 13666, 34852, 26200, 2209, 47326, 56017, 13195, 40204, 12136, 54742, 29770, 21334, 60841, 11392, 3505, 35104, 12901, 44797, 27985, 44539, 21316, 937, 8350, 37654, 45286, 17002, 61645, 169, 8311, 38932, 38695, 36379, 61861, 63667, 44815, 22354, 47581, 52702, 56290, 4249, 20806, 7567, 53230, 44029, 27946, 45817, 14725, 20314, 8566, 35617, 16255, 424, 4996, 16492, 2740, 45049, 14686, 21592, 1975, 54994, 16471, 63922, 41500, 65449, 11626, 61372, 9115, 27709, 43501, 40207, 22120, ...

Moral: There is no substitute for carefully thinking about the problem in an organised way.

Section 1: Complexity

Computation "Size"

- How do we estimate the cost of a computation?
- What cost is deemed "acceptable"?
- Given that input size varies, how do we "standardise" costs?
- What properties of such costs are important?
- If the cost is too high, what can we do?

All computing resources are finite, no matter how large they may seem:

- processor speed,
- memory (RAM, ROM),
- disk, tape, etc.,
- screen size.

Hence, there is *always* some problem which is too large to be solved by a given platform. For example, the number of operations need to solve the travelling salesman problem for 100 cities is a 161-digit number. The number of protons in the known universe is a 79-digit number. How can the resources needed by a program be measured? What resources are required by the computation?

The actual resources used by a program depend upon the programming language, CPU speed, available memory, disk speed, operating system, bus traffic, etc. The theoretical analysis of computational time

- should not depend on a particular implementation;
- should not limit available memory or time;
- must allow for all possible algorithms.

Time Complexity

The measurement of complexity is usually described in terms of how the time (or possibly space) requirements increase as the size of the input to the program increases. The most common measure of complexity is the amount of time the program takes to run.

"*The difference between time and space is that you can re-use space*"

In general, we perform a *worst-case analysis*; i.e. we work out the maximum number of operations that could occur. Worst case analysis:

- guarantees termination within a certain number of steps;
- provides an upper bound on the resources needed.

An average (or even minimum) performance measurement may sometimes be more useful, but in general it makes the analysis significantly more difficult. You have to ask yourself, what is average? How often do "bad" cases occur? etc.

Consider an electronic phone directory.

Aristotle
Archimedes
DaVinci
Einstein
Gallileo
Newton
Plato
Russell
Tesla
Turing

Consider the following "pseudo-code" to find a supplied name:

```
i := 1;
while (name <> book[i] and i < n) do
    i := i + 1;
```

This has a complexity n, as in the worst case it will take n comparisons to find an entry (i.e. the last one in the book). A better method is to use a *binary search* (assuming that the address book is sorted).

Aristotle
Archimedes
DaVinci
Einstein
Gallileo
Newton
Plato
Russell
Tesla
Turing

```
left := 1; right := n;
found := false;
while (not found and left <= right) do
     i := (left + right)/2;
     if (name = book[i]) then found := true
     else if (name > book[i] then
          left := i+1
     else right := i-1
```

This algorithm has a complexity $\log_2 n$. In order to calculate the maximum number of operations that will be done, we analyse the algorithm as explained below.

Get array[ind-id]

will always take constant time no matter how big the array is for a particular problem.

```
For i = 1 to n
    print i
```

will increase linearly as n increases.

```
For i = 1 to n
    for j = 1 to n
        print i - j
```

will have n^2 steps and will increase quadratically.

We often only count the most expensive operation involved.

- Sorting programs — comparisons
- Numerical programs — floating point operations
- Graph algorithms — edge traversals

Rates of Growth

We measure the time complexity of a program by a function. The *rate of growth* of the function is usually what is most important.

n	0	10	100	1000
$20n + 500$	500	700	2,500	20,500
n^2	0	100	10,000	1,000,000
$n^2 + 2n + 5$	5	125	10,205	1,002,005

Note that the lower order terms have progressively less influence as n gets large.

A function f is of *order* g, written $f \in O(g)$ if g is an approximation of the rate of growth of f for large n.

For example:

- $n^2 + 2n + 5$ is $O(n^2)$
- $20n + 500$ is $O(n)$
- $n^3 + 500n^2 + 6$ is $O(n^3)$
- $2^n + 30n^6$ is $O(2^n)$

To determine the order we always take the fastest growing term.

$O(1)$	constant
$O(\log_a n)$	logarithmic
$O(n)$	linear
$O(n \log_a n)$	“n log n”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^r)$	polynomial
$O(a^n)$	exponential
$O(n!)$	factorial

Often anything larger than exponential is referred to simply as exponential (or sometimes hyperexponential). $O(a^{\sqrt{n}})$ is becoming more important in practice.

n	$\log n$	n^2	2^n	$n!$
5	2	25	32	120
10	3	100	1,024	3,628,800
20	4	400	1,048,576	$\sim 10^{18}$
30	4	900	$\sim 10^{9}$	$\sim 10^{32}$
40	5	1,600	$\sim 10^{12}$	$\sim 10^{48}$
50	5	2,500	$\sim 10^{15}$	$\sim 10^{64}$
100	6	10,000	$\sim 10^{30}$	$\sim 10^{161}$
200	7	40,000	$\sim 10^{60}$	$\sim 10^{374}$

Consider a problem with factorial complexity. Assume 1 computation per 10^{-12} seconds. Computing *continuously* from 1 A.D. until now, we would still not have solved the problem for $n = 23$ (!?!). At a rate 1000 times faster, we would have barely finished solving the problem for $n = 21$ (!?!?!?).

Polynomial time

The complexity of a problem is measured by the *most efficient* solution to it. For example, a problem with a linear solution and a quadratic solution is *linear*. A problem is *decidable in polynomial time* if there is a program which solves the problem with time complexity $O(n^r)$ for some r. The class of all such problems is denoted as $\mathcal{P}$. A problem for which there is no efficient solution is said to be *intractable*.

Generally, it is considered that any problem outside $\mathcal{P}$ is intractable. Note that this is a broad categorisation only, because:

- sometimes a polynomial-time algorithm is not efficient enough, or
- sometimes an exponential algorithm will suffice, or
- sometimes an algorithm is "in-between", such as $O(2^{\sqrt{n}})$.

For example, an $O(n^2)$ algorithm is often sufficiently less efficient than an $O(n \log n)$ algorithm. For $n = 25$, $n^2 = 625$, whereas $n \log n \approx 80$. Also, the constant factors can make a difference. For example, $\frac{1}{55}2^n < 1000n^2$ for $n < 25$.

Section 2: Cryptography

Note: this section will not be covered during the 2016 edition. However, this section was left in the course notes for interest of the reader and as as an example of when difficult computational problems are very useful.

Intractable problems can actually be very useful. Consider the problem of keeping information private on a public network, such as transmitting credit card numbers over the Internet. We want to make sure that only the right people can access this information. One way to ensure this is to ensure that unauthorised decryption is *intractable*.

Consider the problem of *encryption*. Write a message as a (large) number — encoding letters as numbers as in ASCII. A message M, is encrypted by applying E, so the encrypted message is $E(M)$. To decrypt, we apply a decryption key, D, so that $M = D(E(M))$. The idea is that no "unauthorised" person can see the message M. In order to do this, we need to ensure that "cracking" the system is sufficiently hard that no-one will attempt it.

Simple Ciphers

Substitution Ciphers

Shift each letter n places.

A	B	C	D	E	F	G	...	Z
E	F	G	H	I	J	K	...	D

The message "SINK THE BISMARCK" become "WMRO XLI FMWQEVGO". Such codes are very easy to break — there are only 25 possible substitution ciphers (why?). This is sometimes also known as Caesar's Cipher. Julius Caesar used $n = 3$.

A	B	C	D	E	F	G	...	Z
D	E	F	G	H	I	J	...	C

Rot 13 (moving 13 places either way) was often used to hide offensive material, quiz answers, punch lines, etc. Historically, many ciphers were based on this idea.

Polyalphabetic Ciphers

Thse are multiple substitution ciphers. These were invented in 1568, and were still in use in the US Civil War (1860-5). For example, one using 5 substitution ciphers might work as follows:

1st letter	4	to the right
2nd letter	3	to the left
3rd letter	10	to the left
4th letter	7	to the right
5th letter	2	to the right
6th letter	4	to the right
...		

Rotors

- Machines with a mechanical wheels (rotors), one per substitution cipher

- Rotors move at different rates; period for an n-rotor machine was 26^n
- Enigma machine used by Germans in World War II
- Enigma codes broken by a British team including Alan Turing

The Unix utility crypt is based on a similar idea. This provides minimal security and sophisticated attack methods are known.

Moral: substitution ciphers of any form are insecure since the invention of the computer.

A more secure system would be to use an arbitrary mixing of the letters.

A	B	C	D	E	F	G	...	Z
X	C	J	Q	B	Z	R	...	Y

There are $25! = 1.5 \times 10^{25}$ combinations, making it difficult to check them all. However, this makes it more difficult to specify the key. Other factors come into play, such as the distribution of letters, keywords, contextual knowledge, ...

There are many other old techniques like these, however, all of them are insecure when computers get involved. Note, that ciphers are *secret key* systems, i.e. the key must be kept private.

The DES Cryptosystem

One cipher which has been used in more modern times is the Data Encryption Standard (DES). It is a

- widely used cryptosystem (certified 1976)
- 64-bit "block" cipher (i.e. text encoded in 64-bit blocks);
- 56-bit (secret) keys;
- complex combination of bitwise permutations and substitutions;
- encryption and decryption algorithms are same;
- decryption key = reverse of encryption key;
- usually implemented in hardware;
- intended for 5-10 year span.

At the time, 56-bit keys were considered outside the cracking range of all but very large organisations. 128-bit numbers were considered more appropriate. Today DES is considered insecure; various credible cracking claims have surfaced. Secure variations include triple-DES (i.e. using 3 different keys).

Moral: Always make the numbers bigger than is strictly needed.

Public key cryptosystems

In general, we wish to be able to ensure secure communication between two arbitrary users. The standard way to do this is *public key cryptosystems*. In public key encryption,

- individual encryption keys E are published;
- decryption keys D are kept secret;
- anyone can encrypt a message for Fred;
- only Fred can decrypt it.

Hence we publish E, and keep D secret. To work, this requires that D be *not* deducible from E. We can ensure this if the task of computing D from E is intractable. This gives us

confidence that *no-one* can find D from E. To sum up, requirements of a public key encryption are:

- D must not be (easily) deducible from E;
- increase the numbers until this happens;
- computing $E(M)$ must be simple enough.

Hence, we want $E(M)$ to be a **one-way trapdoor function**. For authentification, as well as security, we require $E(D(M)) = M$ (as well as $D(E(M)) = M$ to decrypt the message). This allows us to have *signatures*, i.e. a way of determining whether or not a message is authentic.

For example, B sends a secret message to A. B sends $E_A(D_B(M))$ to A, rather than just $E_A(M)$. A decrypts with $D_A(E_B(M))$ rather than just $D_A(M)$.
$D_A(E_B(E_A(D_B(M)))) = D_A(E_B(D_B(E_A(M)))) = D_A(E_A(M)) = M$

The RSA Cryptosystem

How do we find the appropriate functions? One method is to use the *RSA cryptosystem*. This method chooses two large primes p and q, and let $n = p \times q$. Now, find k, g such that

$$k \times g \equiv 1 \bmod ((p-1) \times (q-1)), \text{where}$$

- g and n are made public;
- k, p and q are kept secret;
- divide message into chunks $\leq n-1$;
- $E(m) = m^g (\bmod n)$;
- $D(m) = m^k (\bmod n)$.

So $D(E(m)) = E(D(m)) = m^{(g \times k)} (\bmod n)$, and we can show that:

$$m^{(g \times k)} (\bmod n) \equiv m (\bmod n).$$

To break this system, it is necessary to determine k from n and g. If we can factor n into p and q, k can easily be determined from g, p and q. The way that this is done is to solve the equation $(k \times g) + (x \times r) = 1$, where $r = (p-1) \times (q-1)$. This is known as a *Diophantine* equation, and k and x can be found from g and r quite easily using a variation of *Euclid's algorithm* for greatest common divisor. Determining k from g, p and q like this is how the system is set up in the first place.

Hence modern cryptography uses an algorithm 2,500 years old ... Hence security depends on *not* being able to factor n in any reasonable time. By making n large enough to defeat the best algorithms known, the method is secure. Factoring is known to be a very intractable

problem, and so it is unlikely that a “smart” algorithm will be able to defeat the system. Note also that probabilistic methods can be used to easily find appropriate primes, and so setting up the system is computationally tractable.

Section 3: Grammars and Parsing

Consider the following encrypted message.

WKHVHYHQWLHV

The 25 possible decryptions of this as a simple cipher are below.
xliwizirxmiw
ymjxjajsynjx
znkykbktzoky
aolzlcluaplz
bpmamdmvbqma
cqnbnenwcrnb
drocofoxdsoc
espdpgpyetpd
ftqeqhqzfuqe
gurfriragvrf
hvsgsjsbhwsg
iwthtktcixth
jxuiuludjyui
kyvjvmvekzvj
lzwkwnwflawk
maxlxoxgmbxl
nbymypyhncym
ocznzqziodzn
pdaoarajpeao
qebpbsbkqfbp
rfcqctclrgcq
sgdrdudmshdr
theseventies
uiftfwfoujft
vjgugxgpvkgu

How do we find the "right" one? For this, we need to solve two problems:

1. Find the right way to organise streams of characters into "building blocks" (*lexical analysis).*
2. *Determine whether the current sequence is a "legal" one (*parsing*).*

This makes it important to know about *formal languages*.

Languages

Languages come in all kinds of different forms:

- Natural Languages: English, Chinese, heiroglyphics, Russian, Greek, ...
- Computer Languages: Java, C, HTML, SQML, Cobol, (programming, formatting, querying, etc) ...
- Mathematical Languages: predicate calculus, sets, relations, functions, ...

Given the nature and expanse of such languages, how do we specify such languages? What can we express in such languages? How do we describe "legal" phrases in the language? We need much more than a dictionary. How do we write programs to recognise and manipulate the language?

A *language* is a *set of symbols* that can be combined together in *particular ways* to form the *strings* of the language.

Definition 1 *An* alphabet *is a finite set of* symbols.

Examples:

- Roman: {a,b,c,d,e,f, ... z}
- Greek: $\{\alpha, \beta, \gamma, \delta, \epsilon, \ldots \omega\}$
- Binary: {0,1}
- Numeric: {0,1,2,3,4,5,6,7,8,9}
- Alphanumeric: {a-z, A-Z, 0-9}
- "Keyboard": {a-z, A-Z, 0-9, :'?!()&@ ...}
- C Tokens: {if, while, main, return, +=, ==, case, ...}

Definition 2 *A* string *over an alphabet* Σ *is a finite sequence of symbols from* Σ.

Examples:

- *watermelon* and *banana* are strings over {a,b,c,d,e,f, ... z}
- 1011010111 and 110 are strings over {0,1}
- "if ((x += 1) >= y) while z" is a string of C tokens

Strings can be *empty*; in this notes, we denote the empty string by λ. Another symbol commonly used to denote the empty string is ϵ. The set of all strings over the alphabet Σ (including λ) is denoted by Σ^*. Note that Σ^* represents all possible strings, some of which may not make sense. A *language* then places restrictions on what set of strings are valid (or legal). For example, this sentence is not a valid English sentence, although almost is it.

Also, the syntax of programming languages places restrictions on the ordering of constructs such as **if**, **while**, and **return**. Natural languages can be very difficult to get right. For e.g.,

incorrect syntax	An arrow like flies time
correct syntax	An arrow flies like time
sensible meaning	Time flies like an arrow
sensible meaning (?)	Fruit flies like a banana

Definition 3 *A* language *is a subset of* Σ^*.

Hence a language is just a "certain class" of strings over Σ.

Examples:

- $\Sigma = \{0, 1\}$, $L = \{0, 01, 011, 0111, 01111, \ldots\}$

- $\Sigma = \{a, \ldots z\}$, $L = \{ab, cd, efghi, s, z\}$
- $\Sigma = \{0, 1\}$, $L =$ { (representations of) prime numbers }
- $\Sigma =$ C Tokens, $L =$ { legal C programs }
- $\Sigma = \{0, 1\}$, L = {strings containing at least 2 0's}
- $\Sigma = \{a, \ldots z\}$, $L = \emptyset$

In general, languages are specified via

$$L = \{w \in \Sigma^* \mid w \; has \; property \; P\}$$

Describing a Language

We need to have some mechanism to describe what are the valid strings within a language. Consider the "language" of correct mathematical expressions (infixnotation), involving variable names, *, + (,). How can we describe legal phrases in this language?

Example of valid strings:

- a * (b * c + d)
- a + b
- (c + d)

Example of invalid strings:

- * + a
- + b * *
- (* c d

Language Rules

In order to formally specify languages, we follow certain conventions/forms. For example,

Textbook form

E → E + E
E → E * E
E → (E)
E → id

BNF form

<expr> ::= <expr> + <expr>
— <expr> * <expr>
— (<expr>)
— <id>
<id> ::= string

Other conventions for specifying language rules include:

- DTD's (Data Type Definitions) for languages such as HTML
- Regular Expressions - simple languages with only union, concatenation, repetition.

Regular Expressions

We introduce $\oslash$ to represent the empty language. Hence regular expressions are strings over the alphabet $\{(,), \oslash, \cup, ^* \} \cup \Sigma$.

Definition 4

1. *$\oslash$, λ and each member of Σ is a regular expression.*
2. *If α and β are regular expressions, then so is $(\alpha\beta)$.*
3. *If α and β are regular expressions, then so is $(\alpha \cup \beta)$.*
4. *If α is a regular expression, then so is α^*.*
5. *Nothing else is a regular expression.*

Regular expressions are used as a finite representation of languages. Hence the two languages above can be represented as

$$0^*10^*10^* \; and \; 0^*10^*10^* \cup 0^*1110^*$$

We define the language $\mathcal{L}(\alpha)$ of a regular expression α as follows:

Definition 5

1. *$\mathcal{L}(\oslash) = \emptyset$. $\mathcal{L}(a) = \{a\}$ for each $a \in \Sigma$.*
2. *If α and β are regular expressions, then $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$.*
3. *If α and β are regular expressions, then $\mathcal{L}(\alpha \cup \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$.*
4. *If α is a regular expression, then $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$.*

Examples:

- $\mathcal{L}((a \cup b)^*a) = \{a, aa, ba, aaa, aba, baa, bba, \ldots\}$
  $= \{w \in \{a, b\}^* \mid w \text{ ends in } a\}$

- $\mathcal{L}((01 \cup 1)^*) = \{1, 01, 11, 011, 101, 1011, \ldots\}$
  $= \{w \mid w \text{ does not contain } 00\}$

- $\mathcal{L}(0^*(10^*)^*) = \{0, 1, 00, 01, 10, 11, 001, \ldots\}$
  $= \{0, 1\}^*$
  To see why, consider $w \in \{0, 1\}^*$. Each occurrence of 1 in w is preceded or followed by a (possibly empty) sequence of 0's.
- $\mathcal{L}(0^*(1 \cup \oslash)0^*) = \{0, 1, 00, 01, 010, 000, \ldots\}$
  $= \{w \in \{0, 1\}^* \mid$
  w contains at most one $1\}$

- Let $L = \mathcal{L}(c^*(a \cup (bc^*))^*)$. What is L?
  Note that no string in this language contains ac — each occurrence of a is followed either by another a or by b. So $L \subseteq A = \{w \in \{a, b, c\}^* \mid w \text{ does not contain } ac\}$.
  Is $A \subseteq L$?
  Consider $w \in A$. If w starts with a or b, then $w \in (a \cup (bc^*))^*$, as any sequence of c's must be preceded by a b. Hence it only remains to allow for a sequence of c's at the beginning.
  Hence $\mathcal{L}(c^*(a \cup (bc^*))^*)$ is exactly the strings in $\{a, b, c\}^*$ which do not contain ac, i.e. $L = A$.

- "The language over $\{0,1\}^*$ in which all strings contain 11".
  Does this mean
  1. 0^*110^* (*exactly* two 1's)?
  2. $(0 \cup 1)^*11(0 \cup 1)^*$ (*at least* two 1's)?
  Hence informal descriptions can be ambiguous, but there is no ambiguity in the formal verisons.
- "The language over $\{0,1\}^*$ in which all strings begin with 00 or end with 11".
  $00(0 \cup 1)^* \cup (0 \cup 1)^*11$

There can be many different regular expressions for a given language. For example,

$$\begin{aligned}(0 \cup 1)^* &= ((0 \cup 1)^*)^* \\ &= (0 \cup 1)^*(0 \cup 1)^* \\ &= (\oslash \cup 0 \cup 1)(0 \cup 1)^* \\ &= (0^* \cup 1^*)^* \\ &= 1^*(01^*)^*.\end{aligned}$$

Hence we often want the "simplest" expression (usually minimal number of nested *'s).

- Can be used to represent some (simple) languages.
- Representation is itself a string (and hence can be typed at a keyboard).
- Used in search facilities (vi, emacs, grep, egrep, fgrep, ...) and in compilers.
- Cannot handle some "easy" languages — for example, there is no regular expression for $\{0^n1^n \mid n \geq 0\}$.

Grammar Rules

Grammars are a way of describing language rules, defining the strings that are allowed within a language. Given the *alphabet* of symbols, the grammar tells us how these symbols can be combined. The grammar itself also uses some language form. Some (simple) languages can be described using the mathematical language of *regular expressions* Other languages require a more powerful specification format. If we try to specify the expression (+,*,(,), etc.) as a regular expression, we might try

$$id((+ \cup *)id)^*$$

But how can we capture the brackets? And get the matching number of each? No way to do this as we can't "count" brackets. Similarly we can't express $\{w \in a^nb^n \mid n \geq 0\}$ as a regular expression. We will discuss classes of languages that can be expressed by different kinds of grammars.

Equivalent Grammars

The same language can be specified in different ways, even within the same specification language. For example, consider the language of strings over $\{a,b\}$ which do not contain the substring aa

Regular Expressions:

$$\begin{aligned} & \mathbf{b}^*(\mathbf{abb}^*)^* \cup \mathbf{b}^*(\mathbf{abb}^*)^*\mathbf{a} && (1) \\ = \quad & \mathbf{b}^*(\mathbf{abb}^*)^*\lambda \cup \mathbf{b}^*(\mathbf{abb}^*)^*\mathbf{a} && (2)\end{aligned}$$

$= \quad \mathbf{b}^*(\mathbf{abb}^*)^*(\lambda \cup \mathbf{a}) \qquad (3)$
$= \quad (\mathbf{b} \cup \mathbf{ab})^*(\lambda \cup \mathbf{a}) \qquad (4)$

Equivalent grammars:

$S \to A \mid B, \quad A \to CD, \quad C \to bC \mid \lambda$
$D \to abC, \quad B \to Aa \quad$ (from 1)

$S \to AB, \quad A \to AC \mid \lambda, \quad C \to b \mid ab$
$B \to a \mid \lambda \quad$ (from 4)

Two different grammars which use the same specification language and specify the same set of strings are *equivalent*.

Generating Language Strings

Consider the language defined by $a(a^* \cup b^*)b$. First output is 'a'. Then output a string of 'a''s or string of 'b''s. Then output 'b'.

Let S = a string, M = the "middle part", A = a string of a's, B = a string of b's.

$S \to aMb \quad (1)$
$M \to A \quad (2)$
$M \to B \quad (3)$
$A \to aA \quad (4)$
$A \to e \quad (5)$
$B \to bB \quad (6)$
$B \to \lambda \quad (7)$

To generate the string $aaab$

S
aMb rule (1)
aAb rule (2)
$aaAb$ rule (4)
$aaaAb$ rule (4)
$aaab$ rule (5)

This is an example of *Context Free Grammar*. They are an important class of grammars, in which rules can be applied to symbols (e.g. A) regardless of context of symbol. This makes them computationally useful.

Example Grammars

- Even length strings over $\{a, b\}$

  $S \to aO \mid bO \mid \lambda$
  $O \to aS \mid bS$

  $S \Rightarrow aO \Rightarrow abS \Rightarrow abbO \Rightarrow abbbS \Rightarrow abbb$

- Strings with an even number of b's
  (i.e. $\mathbf{a}^*(\mathbf{ba}^*\mathbf{ba}^*)^*$)

  $$S \to aS \mid bB \mid \lambda$$
  $$B \to aB \mid bS$$

  $S \Rightarrow aS \Rightarrow abB \Rightarrow abbS \Rightarrow abbbB \Rightarrow abbbaB \Rightarrow abbbabS \Rightarrow abbbab$
- Strings not containing abc

  $$S \to aB \mid bS \mid cS \mid \lambda$$
  $$B \to aB \mid bC \mid cS \mid \lambda$$
  $$C \to aB \mid bS \mid \lambda$$

  $S \Rightarrow aB \Rightarrow abC \Rightarrow abaB \Rightarrow abacS \Rightarrow abac$
- Palindromic strings, $w = w^R$, over $\{a, b\}$

  $$S \to a \mid b \mid \lambda$$
  $$S \to aSa \mid bSb$$

  $S \Rightarrow aSa \Rightarrow abSba \Rightarrow ababa$

Formal Definition

A *context-free grammar* G is a quadruple (V, Σ, R, S) where

- V is a finite set of non-terminals;
- Σ is the set of terminals (the symbols of the language);
- S is a distinguished element of V called the *start symbol*; and
- R is a set of rules.

Note that the set of non-terminals V and the set of terminals Σ are assumed to be disjoint.

Example Grammar

Grammar $G = (V, \Sigma, R, S)$, where $V = \{S\}$, $\Sigma = \{a, b\}$ and R consists of rules $\{S \to aSb$ and $S \to \lambda\}$. Using these rules we can get:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

The *language* of G, denoted $L(G)$, is the set $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$. This is clearly $\{a^n b^n \mid n \geq 0\}$. Recall that this is *not* a language which can be specified by a regular expression.

Derivations

Rule application: Given the string uAv and the rule $A \to w$, we obtain the string uwv. We denote this as $uAv \Rightarrow uwv$. A string w is derivable from v if there is a finite sequence of rule applications form v to w.

$$v \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n = w$$

Usually written as $u \overset{*}{\Rightarrow} w$. For example, $S \Rightarrow A\underline{A} \Rightarrow \underline{A}bA \Rightarrow abA$, so $S \overset{*}{\Rightarrow} abA$. The *length* of a derivation $v \overset{*}{\Rightarrow} w$ is the number of rule applications in the derivation.

Derivations

S	$\to$	aMb	(1)
M	$\to$	A	(2)
M	$\to$	B	(3)
A	$\to$	aA	(4)
A	$\to$	e	(5)
B	$\to$	bB	(6)
B	$\to$	λ	(7)

$aaab$ is derivable from S:
$S \Rightarrow aMb \Rightarrow aAb \Rightarrow aaAb \Rightarrow aaaAb \Rightarrow aaab$
$S \overset{*}{\Rightarrow} aaab$.
The length of this derivation is 5.

Parse Trees and Derivations

Derivations can be written in a graphical form as a parse tree. Given a grammar and a string, there may be different derivations to get the same string. Equivalent derivations (same meaning) have the same parse tree. Any parse tree has unique *leftmost* and *rightmost* derivations. Grammars with strings having 2 or more parse trees are *ambiguous*. Some ambiguous grammars can be rewritten as equivalent unambiguous grammars.

Please refer to Chapter 3 in Sudkamp's course book (page 71 in 3rd edition; page 60 in 2nd edition) for definition of left-most derivations, as well as Definition 3.5.2 (3rd edition) or Definition 4.1.2 (2nd edition) for precise definition of ambiguous grammars.

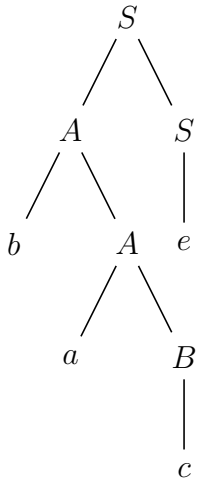
Derivation as Parse Tree

Consider the following grammar.

$S \to AS \mid SB \mid \lambda$
$A \to aB \mid bA \mid \lambda$
$B \to bS \mid c \mid \lambda$

The string abc can be derived by applying the above rules in the following order.
$S \Rightarrow AS \Rightarrow bAS \Rightarrow baBS \to bacS \to bac$
A *parse tree* of $S \overset{*}{\Rightarrow} w$ is then obtained as follows:
- The parse tree has root S

- If $S \to AS$ is the rule applied to S, then add A and S as children of S. $A \to bA$, then add b and A as children of A ...
- If $A \to e$ is the rule applied to A, then add e as the only child of A.



Equivalent Derivations

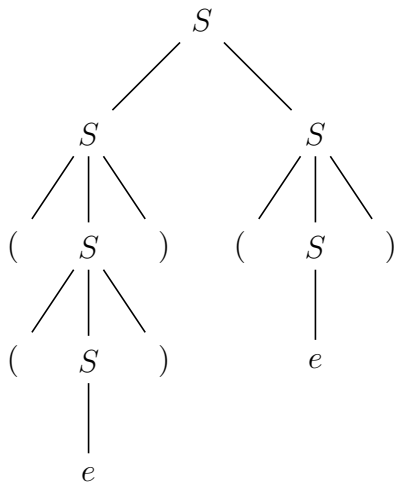
A grammar can admit different derivations yielding the same string. For example, consider this simple grammar $G = (V, \Sigma, R, S)$, where

- $V = \{S\}$.
- $\Sigma = \{(,)\}$.
- $R = \{S \to SS \mid (S) \mid \epsilon\}$.

The string "(())()" can be derived from S by several derivations, e.g.:

(D_1) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$
(D_2) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$
(D_3) $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$

In the derivations above, D_1 and D_2 differ in that steps 4 and 5 are reversed. These all have the same parse tree, namely:



Leftmost/Rightmost Derivations

A *left-most derivation* is one in which the left-most variable is used for expansion at all points in the derivation. In other words, when there is a choice to be made between two or more variables, the left-most one is always chosen. Similarly, a *right-most derivation* is one where the right-most variable is always chosen. If there is only one variable at every derivation step, then the derivation is both left-most and right-most.

See Chapter 3 in Sudkamp's course book (page 71 in 3rd edition; page 60 in 2nd edition) for definition of left-most derivations.

Let G be the following grammar:

$$S \to AB$$
$$A \to aA \mid Aa \mid a$$
$$B \to bB \mid Bb \mid b$$

A left-most derivation for $aaabb$: $\underline{S} \Rightarrow \underline{A}B \Rightarrow a\underline{A}B \Rightarrow a\underline{A}aB \Rightarrow aaa\underline{B} \Rightarrow aaa\underline{B}b \Rightarrow aaabb$

A right-most derivation for $aaabb$: $\underline{S} \Rightarrow A\underline{B} \Rightarrow A\underline{B}b \Rightarrow Abb \Rightarrow a\underline{A}bb \Rightarrow a\underline{A}abb \Rightarrow aaabb$

A derivation that is neither left nor right most: $\underline{S} \Rightarrow A\underline{B} \Rightarrow \underline{A}Bb \Rightarrow a\underline{A}Bb \Rightarrow aAa\underline{B}b \Rightarrow a\underline{A}abb \Rightarrow aaabb$

Ambiguous Grammars

If you can have two left-most derivations for the same string from a given grammar then that grammar is ambiguous. **Please see Sudkamp's book section on left-most derivations and ambiguous grammars: Definition 3.5.2 in 3rd edition or Definition 4.1.2 in 2nd edition.**

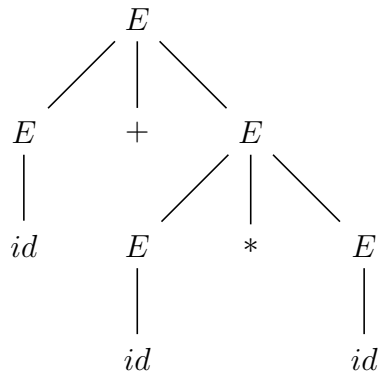
Recall the grammar $G = (V, \Sigma, R, S)$, where

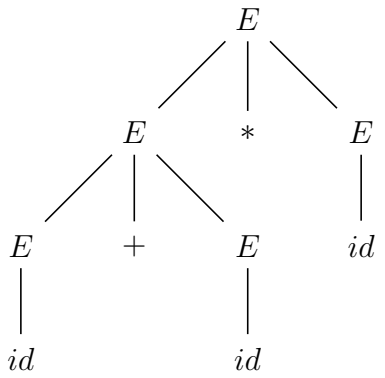
- $V = \{S, E\}$;
- $\Sigma = \{*, +, (,)\}$; and
- $R = \{S \to E, E \to E + E \mid E * E \mid (E) \mid id\}$.

The string $id + id * id$ can be generated by this grammar with two different left-most derivations according to two *different parse trees*.

- $E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$
- $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$

Accordingly, these derivations yield two *different parse trees*, namely:





Observe that one of these corresponds to the "natural" semantics of $id + id * id$, where $*$ takes precedence over $+$. *Which one?*

Ambiguous Languages

Many ambiguous grammars (such as the one on the previous slide) can easily be converted to an unambiguous grammar representing the same language. Some context free languages have the property that *all* grammars that generate them are ambiguous. Such languages are *inherently ambiguous*.

Inherently ambiguous languages are not useful for programming languages, formatting languages or other languages which must be parsed automatically.

Parsing

Given a context-free grammar G and input w, determine if $w \in L(G)$. The problem is:

- how do we determine this for **all** possible strings?;
- multiple derivations may exist;
- must also discover when no derivation exists.

A procedure to perform this function is called a *parsing algorithm* or *parser*. Some grammars allow *deterministic parsing*, i.e. each sentential form has at most one derivation.

Ambiguity and Parsing

A grammar is unambiguous if at each step in a leftmost derivation there is only one rule which can lead to the desired string. Deterministic parsing is based upon determining which rule to apply.

- **top-down parsing** From start symbol generate a leftmost derivation of w guided by w.
- **bottom-up parsing** From w generate in reverse order a rightmost derivation of $w \overset{*}{\Leftarrow} S$ guided by rules of G.

Top Down Parsing

Leftmost derivation from S with choice of rule guided by w. For example, a grammar for $\{a^i b^i \mid i \geq 0\}$ is $S \to aSb \mid \lambda$

Derivation for $aaabbb$

Derivation	terminal prefix	next char	rule
S	e	a	$S \to aSb$
aSb	a	a	$S \to aSb$
$aaSbb$	aa	a	$S \to aSb$
$aaaSbbb$	aaa	b	$S \to \lambda$
$aaabbb$			

Grammar for arithmetic expressions

$S \to A$
$A \to T \mid A + T$
$T \to b \mid (A)$

Derivation for $(b + b)$

Derivation	terminal prefix	Derivation	terminal prefix
S	λ	S	λ
A	λ	A	λ
T	λ	T	λ
(A)	$($	(A)	$($
$(A + T)$	$($	(T)	$($
$(T + T)$	$($	(b)	(b)
$(b + T)$	$(b+$		
$(b + b)$	$(b + b)$		

Grammar for arithmetic expressions

$$S \to A$$
$$A \to T \mid A + T$$
$$T \to b \mid (A)$$

S
A
T
(A)
(A + T)
(T + T)
(b + T)
(b + b)

Bottom Up Parsing

Construct a rightmost derivation in reverse by doing the leftmost production backwards. A *reduction* (reverse production) is of the form $w = u_1 q u_2$ and $A \to q$ gives $v = u_1 A u_2$.

Reductions for $(b) + b$

Reduction	Rule
$(\underline{b}) + b$	$T \to b$
$(\underline{T}) + b$	$A \to T$
$\underline{(A)} + b$	$T \to (A)$
$\underline{T} + b$	$A \to T$
$A + \underline{b}$	$T \to b$
$\underline{A + T}$	$A \to A + T$
$\underline{A}$	$S \to A$
S	

Rightmost derivation in reverse

$$\begin{aligned} S &\Rightarrow \underline{A} \\ &\Rightarrow A + \underline{T} \\ &\Rightarrow \underline{A} + b \\ &\Rightarrow \underline{T} + b \\ &\Rightarrow (\underline{A}) + b \\ &\Rightarrow (\underline{T}) + b \\ &\Rightarrow (b) + b \end{aligned}$$

Again there are multiple possibilities. For string $w \in (V \cup \Sigma)^*$ we can split it into $w = uv$ and apply reductions to the rightmost part of u.

	u	v	Rule	Reduction
	e	$(A+T)$		
$shift$	$($	$A+T)$		
$shift$	$(A$	$+T)$	$S \to A$	$(S+T)$
$shift$	$(A+$	$T)$		
$shift$	$(A+T$	$)$	$A \to A+T$	(A)
			$A \to T$	$(A+A)$
$shift$	$(A+T)$	e		

Current derivation string $w = u \cdot v$. Apply reductions to rightmost part of u, or shift one character from v to u. For e.g. $u \cdot az$ becomes $ua \cdot z$.

Further Parsing Methods

More sophisticated forms of parsing:

- **top-down parsing**
  - Lookahead
  - LL(k) grammars
- **bottom-up parsing**
  - LR(0) grammars (and machines)
  - LR(1) grammars (and machines)

Section 4: Finite State Machines

Our model of computation comes down to two parts: *input (and output) language* and *processor*. For example:

- Coke machine (input money, output drinks + change if right coins inserted).
- Hardware adder (input bits, output bits representing sum of input).
- Credit card checker (input card number + expiry date, output validity as 'yes' or 'no').
- Compiler (input program, output yes/no answer + compiled code).
- Operating System (input commands, output could be anything!).

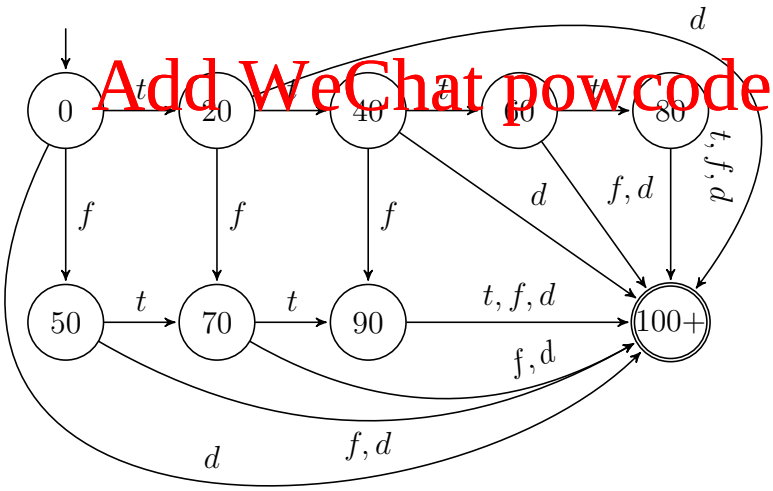
Finite State Automata are a method of specifying a process. We look at a simple machine (and more complex ones later) which has following properties.

- simple, fixed amount of memory;
- driven by input;
- transitions between states;
- simple output, often yes/no.

This model is simple, but useful — applications include lexical analysis in compilers and string search algorithms. Consider a (simple) Coke machine.

- accepts 20c (t), 50c (f) and $1.00 (d) coins;
- if given $1.00 or more it delivers a coke.

States: Fed 0, 20, 40, 50, ... 100+
Initial state: Fed 0
Transitions: e.g. fed 20, insert 20, new state is fed 40
Final state: Fed 100+



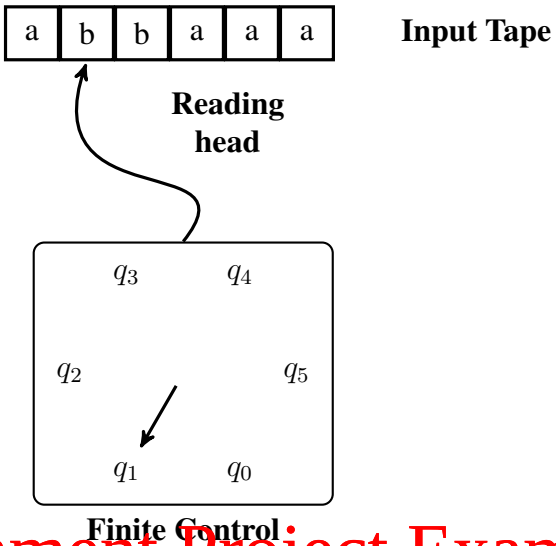
	t	f	d
0	20	50	100
20	40	70	100
40	60	90	100
50	70	100	100
60	80	100	100
70	90	100	100
80	100	100	100
90	100	100	100

Here, *states* are denoted with circles and *transitions* with edges between states. State 0 is the *initial* state, whereas the double circled state $100+$ is the *final* state. A finite state machine has:

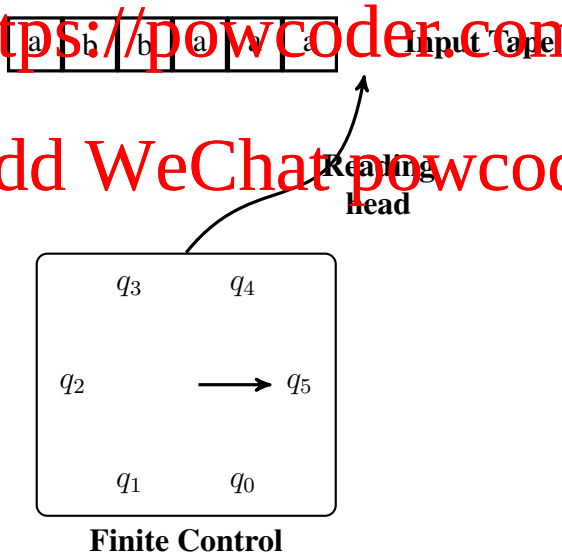
- an input as a string on a *tape*;
- a head which reads the tape left-to-right;
- an internal register with distinct internal *states*;

- instructions for changing to a new state depending on the input (a transition table);
- a distinguished initial state; and
- a number of *final* states.

An input string is accepted if the machine is in a final state when the input finishes; otherwise it is rejected.



Result is determined when the input is finished.



Definition 6 *A* deterministic finite automaton *is a quintuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where,*

- Q *is a finite set of states;*
- Σ *is an alphabet;*
- $q_0 \in Q$ *is the* initial state*;*
- $F \subseteq Q$ *is the set of* final states*;*
- δ*, the* transition function *is a function from* $Q \times \Sigma$ *to* Q*.*

Note that for every state in Q and every input symbol in Σ, there must be a *unique* new state. Computations in such a machine are a sequence of *configurations*. As the tape must be read from left to right, this will depend only on the current state and remaining input.

We repesent this by a tuple from $Q \times \Sigma^*$, e.g. $(q_3, abbab)$. We say $(q, w) \vdash_M (q', w')$ if (q, w) and (q', w') are configurations of M, $w = aw'$ for some $a \in \Sigma$ and $\delta(q, a) = q'$. We say $(q, w) \vdash_M^* (q', w')$ if there is a sequence of transitions from (q, w) to (q', w'). M accepts w if there is a state $q \in F$ such that $(q_0, w) \vdash_M^* (q, \lambda)$. $L(M)$ is the language accepted by the machine M.

Hence the computation cycle repeats the following 3 steps:

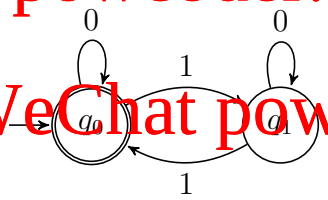
1. Read current input symbol x. Halt if end-of-input.
2. Determine the new state from x and the current state.
3. Move the tape head one position to the right.

Note that in step 2 there are no choices; hence the name deterministic. Output is a yes/no answer. There are variations of such machine which produce more expressive output known as *transducers*.

Let M be the DFA $(Q, \Sigma, \delta, q_0 F)$ where $Q = \{q_0, q_1\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$ and δ is

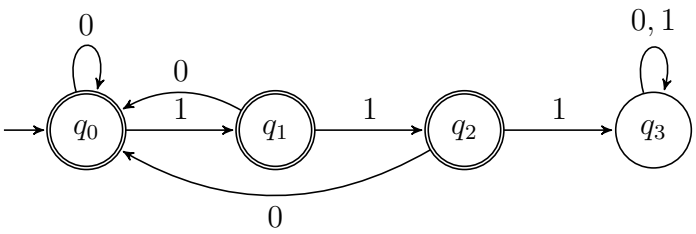
δ	0	1
q_0	q_0	q_1
q_1	q_1	q_0

$L(M)$ is strings over $\{0, 1\}$ which contain an even number of 1's. This is much easier to understand from a *state diagram*.



Consider a machine M which accepts strings over $\{0, 1\}$ which do not contain three consecutive 1's, i.e., $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{q_0, q_1, q_2\}$, and δ is:

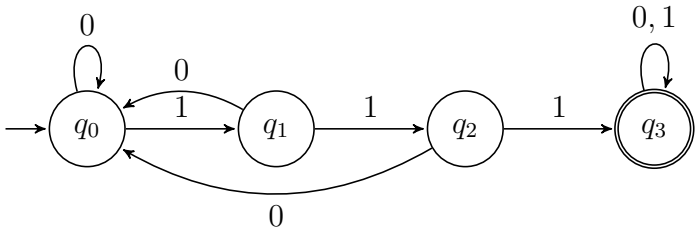
δ	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3



Note that q_0, q_1 and q_2 are all final states, and that q_3 is a *dead* state; once computation reaches q_3, it stays there from then on.

Here is a machine which accepts strings over $\{0,1\}$ which **do** contain three consecutive 1's: $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$, $F = \{q_3\}$ and δ is

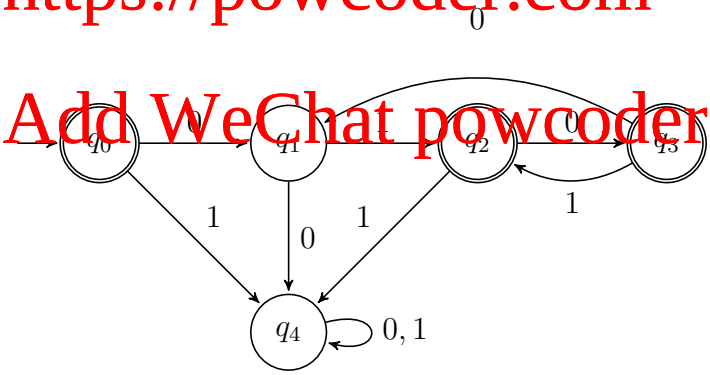
δ	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3



Hence it is simple to convert a DFA into a DFA for the complement of the language.

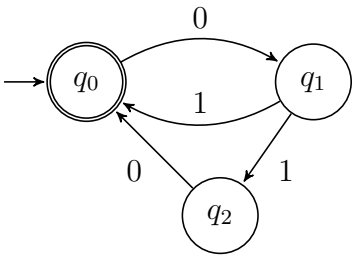
Non-determinism

DFAs are precise, but can be overly "clumsy". Consider the following DFA for the language $(01 \cup 010)^*$:



This is the smallest DFA which accepts the language above. Can we specify this machine in a simpler way? Often we can, by not fully specifying δ and not insisting on a unique new state for each old state and character. This introduces *nondeterminism*, i.e. unspecified choices, into the machine.

Here is a nondeterministic version of the same machine:



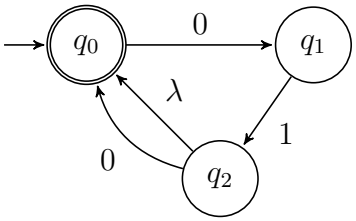
Note that there is no restriction on transitions here. A string is accepted if there is *some* way finishing in a final state; not all computational paths will work. Also note that there is no state to enter in the above machine if the first character is 1; here the machine just rejects the string immediately.

Hence the computation cycle is now:

1. (Same) Read current input symbol x. Halt if end-of-input.
2. **(NEW)** Choose a new state from x and the current state. If there are no such states, halt with failure.
3. (Same) Move the tape head one position to the right.

Note that step 2 is now nondeterministic and that the machine can halt with failure before the end of the input is reached. Acceptance only requires that there is *at least one* computation which succeeds.

Another nondeterministic machine for the same language is the one below.



Note that the empty string can further simplify the design of such machines. Hence nondeterministic machines can be simpler for humans to understand. Nondeterministic machines are also simpler to combine than deterministic ones.

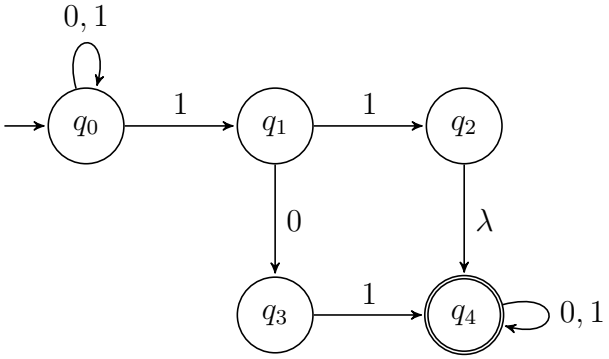
Definition 7 *A nondeterministic finite automaton is a quintuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where,*

- *(same)* Q *is a finite set of states;*
- *(same)* Σ *is an alphabet;*
- *(same)* $q_0 \in Q$ *is the* initial state;
- *(same)* $F \subseteq Q$ *is the set of* final states;
- **(NEW)** δ*, the* transition relation *is a subset of* $Q \times (\Sigma \cup \{\lambda\}) \times Q$.

Note that δ is now a relation on $Q \times (\Sigma \cup \{\lambda\}) \times Q$, rather than a function from $Q \times \Sigma$ to Q. δ can also be considered as a *partial function* from $Q \times (\Sigma \cup \{\lambda\})$ to 2^Q. Configurations are much as before, with two new features:

1. For a transition labelled with e, no input is read (i.e. the tape head does not move).
2. For a given configuration (q, w) there can be 0, 1, or several new possible configurations.

M accepts w if there is a state $q \in F$ such that $(q_0, w) \vdash_M^* (q, e)$. Note that this means there need be only *one* computation that succeeds.



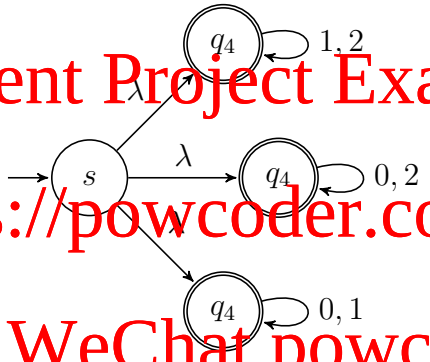
Note that

$$\begin{aligned}(q_0, 1010101) &\vdash_M (q_0, 010101)\\ &\vdash_M (q_0, 10101)\\ &\vdash_M (q_0, 0101)\\ &\dots\\ &\vdash_M (q_0, e)\end{aligned}$$

$$\begin{aligned}(q_0, 1010101) &\vdash_M (q_1, 010101)\\ &\vdash_M (q_3, 10101)\\ &\vdash_M (q_4, 0101)\\ &\dots\\ &\vdash_M (q_4, e)\end{aligned}$$

Another example is the machine below.

Let $L = \{w \mid$ there is a symbol $a_i \in \Sigma$ *not* appearing in w. L may be considered the complement of the strings which contain every symbol in Σ. For the case when $\Sigma = \{0, 1, 2\}$ the machine M is as follows:



Intuitively, M "guesses" which character is missing.

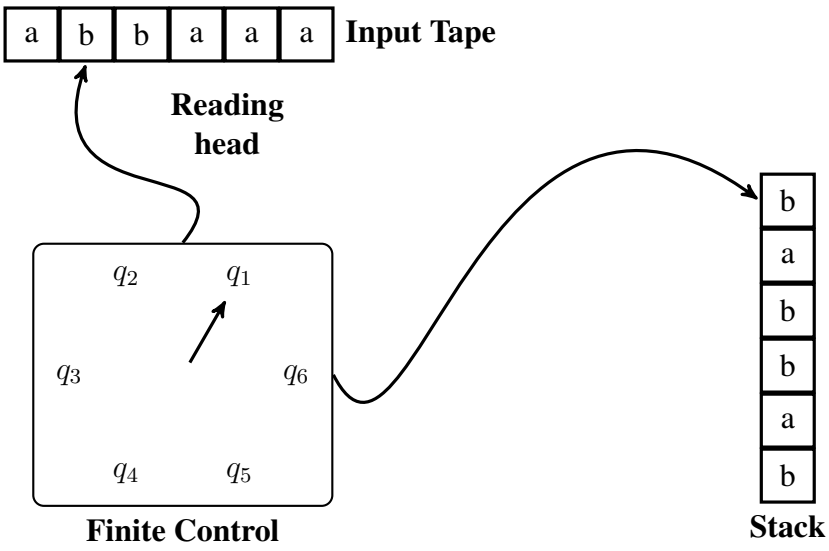
Pushdown Automata

Context-free grammars are more powerful than regular expressions or NFAs. Is there an equivalent machine? If so, what extra features do we need to add to NFAs?

Consider $\{ww^R \mid w \in \{a, b\}^*\}$. This is context-free, but not regular.

- Needs to "remember" 1st half of the string.
- Requires some memory "in reverse".

One such model is a *pushdown automaton* (PDA).



Like an NFA, but with a stack (or pushdown store) memory. How to balance parentheses?

- Count the parentheses from left to right
- +1 for each (
- -1 for each)
- If the count becomes negative, reject the string
- Accept if the count is 0 at the end of the string

PDAs work like this. To accept a string computation must finish in a final state, and the stack must be empty. $A \to aB$ can be easily modelled by an NFA. For $A \to aBb$, we need somewhere to store the b.

Definition 8 *A* pushdown automaton *is a sextuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *where*

- Q *is a finite set of states;*
- Σ *is an alphabet;*
- **(NEW)** Γ *is the* stack alphabet*;*
- $q_0 \in Q$ *is the* initial state*;*
- $F \subseteq Q$ *is the set of* final states*;*
- **(NEW)** δ*, the* transition relation *is a finite subset of* $(Q \times (\Sigma \cup \{\lambda\}) \times \Gamma^*) \times (Q \times \Gamma^*)$.

A transition of the form $\delta((p, a, \beta), (q, \gamma))$ means:

- Old state is p, input is a, β on stack
- New state is q
- Replace β with γ on stack

Configurations of M are elements of $Q \times \Sigma^* \times \Gamma^*$. If (p, x, α) and (q, y, ζ) are configurations of M, then $(p, x, \alpha) \vdash_M (q, y, \zeta)$ if there is a transition $((p, a, \beta), (q, \gamma)) \in \Delta$ such that $x = ay, \alpha = \beta\eta$ and $\zeta = \gamma\eta$ for some $\eta \in \Gamma^*$. M *accepts* w iff $(s, w, \lambda) \vdash_M^* (p, \lambda, \lambda)$ for some $p \in F$.

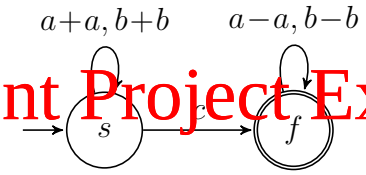
Push means to add a symbol to the top of the stack; **pop** means to remove the top symbol from the stack. Note that an empty stack cannot be popped. If this is attempted, the PDA halts abnormally. It is also important to understand that PDAs are nondeterministic, just like context-free grammars.

PDA for $\{wcw^R \mid w \in \{a,b\}^*\}$. $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where $Q = \{s, f\}$, q_0 is s, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$ and $\delta(\cdot, \cdot)$ is as follows:

$$\begin{array}{ll} ((s,a,e),(s,a)) & ((f,a,a),(f,e)) \\ ((s,b,e),((s,b)) & ((f,b,b),(f,e)) \\ ((s,c,e),((f,e)) & \end{array}$$

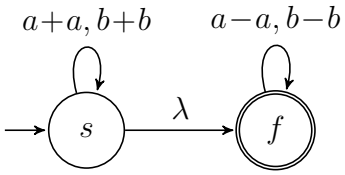
State	Unread Input	Stack
s	$abbcbba$	e
s	$bbcbba$	a
s	$bcbba$	ba
s	$cbba$	bba
f	bba	bba
f	ba	ba
f	a	a
f	e	e

We can also represent PDAs via state diagrams.



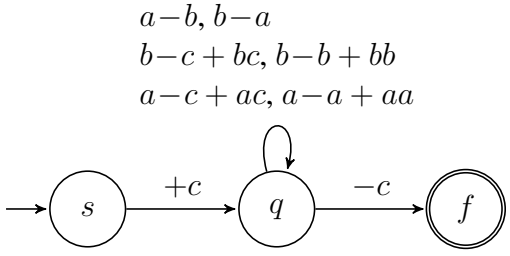
As before, this generally makes this a lot simpler to understand. In general, a transition is represented as $a - \alpha + \beta$, meaning pop α from the stack and push β onto it.

Let $L = \{ww^R \mid w \in \{a,b\}^*\}$. We have the same transitions as before except that $((s, c, \lambda), (f, \lambda))$ becomes $((s, \lambda, \lambda), (f, \lambda))$.



Note that this PDA is now nondeterministic, as it has to "guess" when to transfer to state f. As before, some computations will not accept the string, but as long as at least one does, the string is accepted overall.

Consider $L = \{w \in \{a,b\}^* \mid w$ has the same number of a's as b's$\}$. A PDA for this is as follows.



Here the transitions are:

$((s, \lambda, \lambda),\ (q, c))$
$((q, a, c),\ (q, ac))$
$((q, a, a),\ (q, aa))$
$((q, a, b),\ (q, \lambda))$
$((q, b, c),\ (q, bc))$
$((q, b, b),\ (q, bb))$
$((q, b, a),\ (q, \lambda))$
$((q, \lambda, c),\ (f, \lambda))$

Limitations of PDAs

PDAs are as powerful as context-free grammars, but hence share their limitations. The language $\{a^n b^n c^n \mid n \geq 0\}$ is not context-free, and hence cannot be recognised by any PDA. PDAs are also noted for being one class of machine in which the deterministic version is strictly less powerful than the non-deterministic one.

Extensions of PDAs

We can extend PDAs in a number of ways — adding an extra stack, or allowing random access to the stack (possibly with restrictions, such as accessing only a bounded amount of it). However, many such additions turn out to give us the most powerful machine of all .... (see next section).

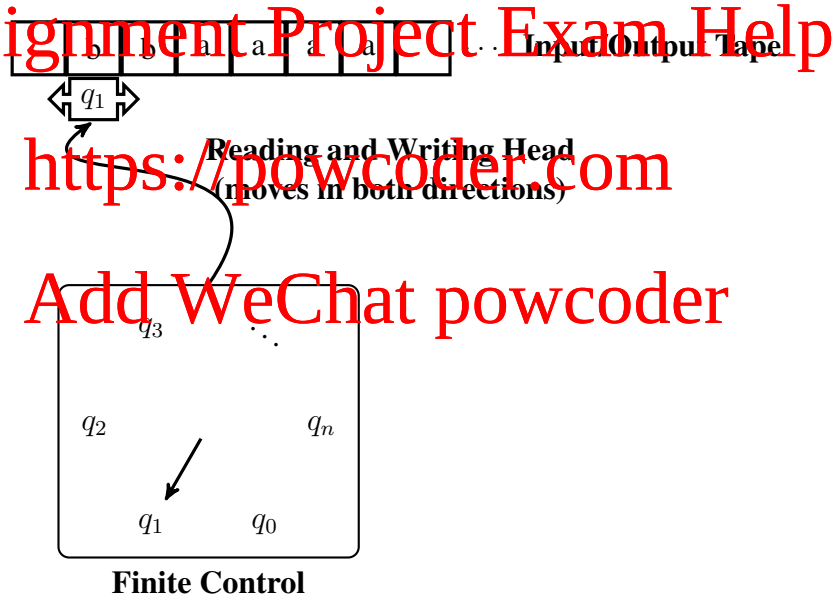
Section 5: Turing Machines

Turing Machine is a conceptual machine similar to FSMs and PDAs. It was invented (and conceived) by Alan Turing in 1936, shortly before the first computers. It is the most powerful formal machine known to computer scientists.

Some of the properties of a Turing machine are:

- unbounded memory using an *infinite tape*;
- can be an *acceptor* or a *transducer*;
- input as a string on a *tape*;
- **(NEW)** output as a string on the *tape*;
- internal register for distinct internal states;
- instructions for changing to a new state, depending on the input;
- a distinguished initial state; and
- a number of *final* states.

A Turing machine is conceptually similar to a modern computer (as we will see). Other formal machines which we have seen previously, can be modified such that they have similar computational power as a Turing machine. Such as, FSA + random access "queue" is similar, PDA + queue (Post machine) or PDA + 2 stacks are all similar in power. It should be noted that non-determinism does *not* add to the power of a Turing machine.



The computation sequence is,

1. change to a new state (specified by the current state and input symbol),
2. write a new symbol on the tape,
3. move the tape head one place left or right.

There are variations on this, such as allowing the machine only to move left or right *or* write a new symbol, but not both; as we shall see, there is no essential difference between these versions of Turing machines.

Note that:

- the tape is bounded to the left, but is infinite to the right;

- the tape alphabet includes input alphabet Σ and blank B and extra internal symbols;
- initially the tape head is pointing at the left end of the tape;
- input w is a string in Σ^*, $BwBBBB\ldots$

Definition 9 *A* Turing machine *is a quintuple* $M = (Q, \Sigma, \Gamma, \delta, q_0)$ *where*

- Q *is a finite set of states;*
- Γ *is the* tape alphabet*, which must include the symbol* B *for a blank space;*
- Σ *is the* input alphabet*, which is a subset of* $\Gamma - \{B\}$ *;*
- $q_0 \in Q$ *is the* initial state*;*
- δ*, the* transition function *is a function from* $Q \times \Gamma$ *to* $Q \times \Gamma \times \{L, R\}$).

Note, that if the machine attempts to move to the left from the leftmost tape position, the machine is said to ***terminate abnormally***. A Turing machine **halts** if it encounters a state and symbol for which no transition is defined. A transition depends on two things

1. current state;
2. current symbol under tape head.

A transition itself is three actions

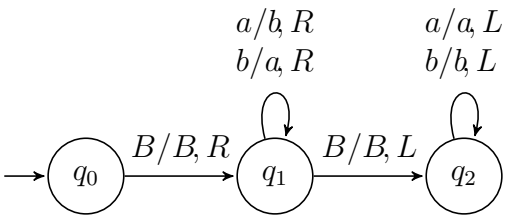
1. change state;
2. write symbol on tape;
3. move the tape head.

Turing machines compute on strings from the input alphabet. They have **output**, i.e. what is on the tape when they halt. Note that a Turing machine, as defined above, is deterministic. This is not a restriction; as we shall see, adding non-determinism does not add more power (similar to NFAs and DFAs).

Examples

A Turing machine to interchange a's and b's
$Q = \{q_0, q_1, q_2\}, \mathcal{S} = \{a, b\}, \Gamma = \{a, b, B\}$

δ	B	a	b	
q_0	q_1, B, R			move off start state
q_1	q_2, B, L	q_1, b, R	q_1, a, R	swap and start backwards
q_2		q_2, a, L	q_2, b, L	go back to beginning



Consider $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where,

$K = \{q_0\}$, $\Sigma = \{a\}$, $\Gamma = \{a, B\}$ and δ is

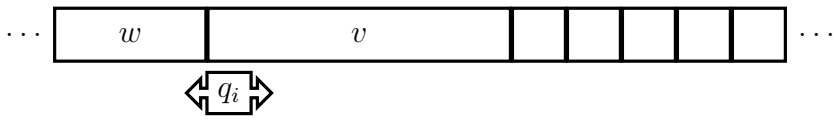
δ	a	B
q_0	q_0, a, L	q_0, B, R

Computation beginning with q_0BaaaB is

$$\begin{aligned} q_0BaaaB &\Rightarrow Bq_0aaaB \\ &\Rightarrow q_0BaaaB \\ &\Rightarrow Bq_0aaaB \\ &\Rightarrow q_0BaaaB \\ &\Rightarrow Bq_0aaaB \\ &\Rightarrow q_0BaaaB \\ &\Rightarrow Bq_0aaaB \\ &\dots \end{aligned}$$

As before, computations are a sequence of configurations. A configuation consists of the current state, the tape, and the tape head position. Note that at any step of the computation only a finite segment of the tape is not blank.

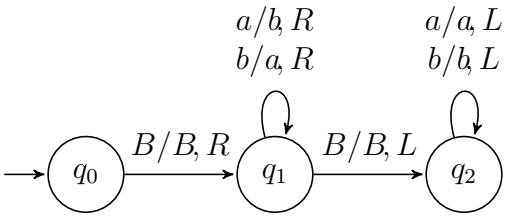
Denote configuration as uq_ivB to mean:



- uv is the string on the tape from left boundary to rightmost non-blank;
- q_i is the current state;
- tape head is assumed to be scanning the first symbol of v.

Denote a single step as $uq_ivB \vdash xq_jyB$. Denote multiple steps (including zero) as $uq_ivB \overset{*}{\vdash} xq_jyB$. Note that as in the example above, a machine may never reach a halted configuration.

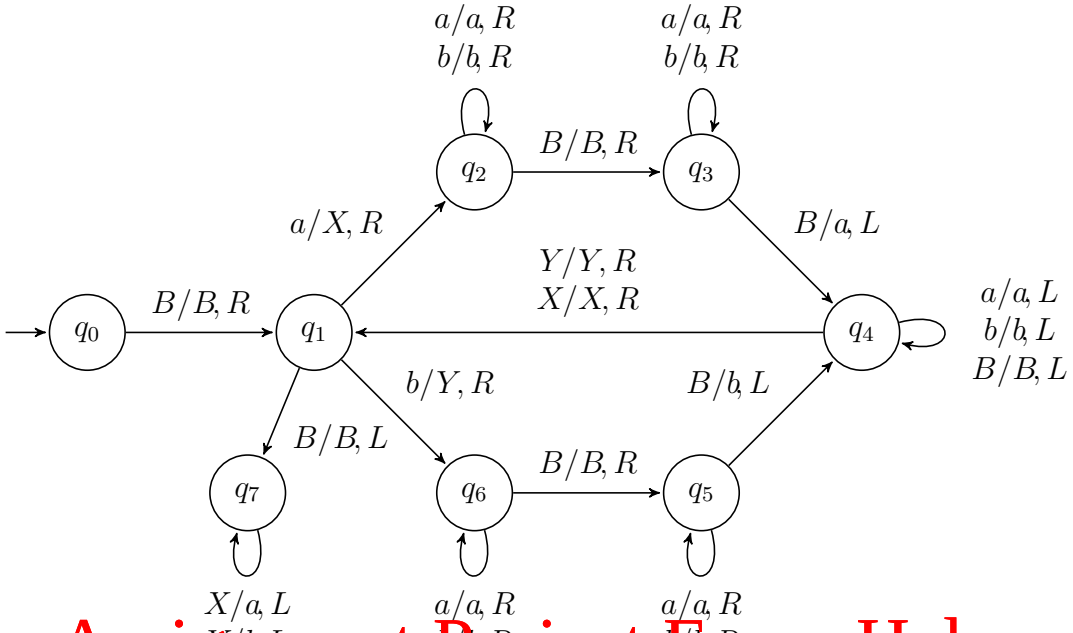
A Turing machine to interchange a's and b's:



$$\begin{aligned} & q_0BababB \\ \vdash\ & Bq_1ababB \\ \vdash\ & Bbq_1babB \\ \vdash\ & Bbaq_1abB \\ \vdash\ & Bbabq_1bB \\ \vdash\ & Bbabaq_1B \\ \vdash\ & Bbabq_2aB \\ \vdash\ & Bbaq_2baB \\ \vdash\ & Bbq_2abaB \\ \vdash\ & Bq_2babaB \\ \vdash\ & q_2BbabaB \end{aligned}$$

Machine to COPY string with input alphabet $\{a, b\}$.

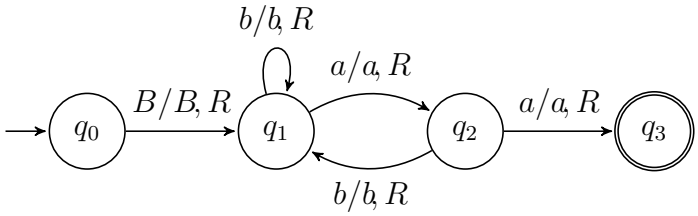
$BuB \overset{*}{\vdash} BuBuB$



Turing Machines as Language Acceptors

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a Turing machine. A string $u \in \Sigma^*$ is *accepted by final state* if M with input u halts in a state $q \in F$. Unlike PDAs and DFAs a Turing machine does not have to read the entire string to accept it.

Machine for $(a \cup b)^* aa(a \cup b)^*$ (assuming $\Sigma = \{a, b\}$):

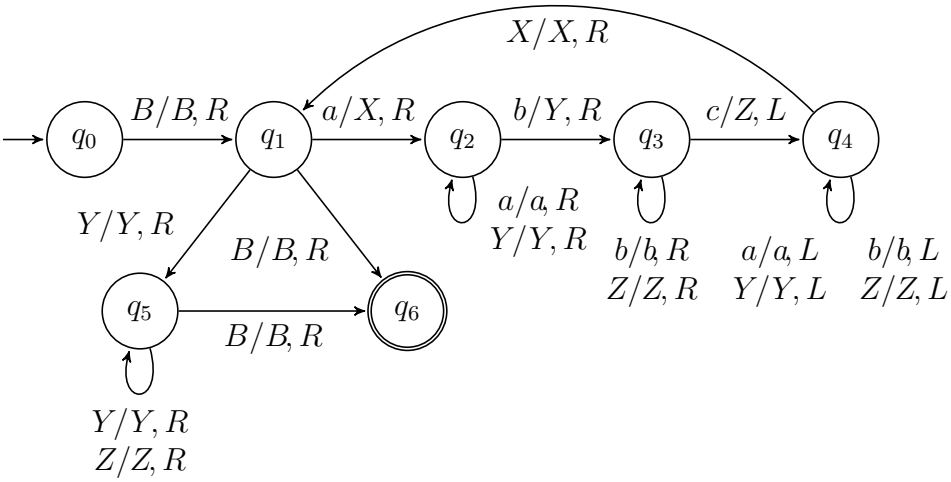


Computation for input $aabb$:

$$
\begin{aligned}
& q_0 Baabb B \\
\vdash\ & B q_1 aabb B \\
\vdash\ & B a q_2 abb B \\
\vdash\ & B a a q_3 bb B
\end{aligned}
$$

A language accepted by a Turing machine is called a *recursively enumerable language*. If the Turing machine halts for all input the language is *recursive*.

Here is a Turing machine for the language $\{a^n b^n c^n \mid n \geq 0\}$.



This machine works as follows:

- Start at left end of string;
- Search for a, skipping X's;
- If found, mark found a with X, and then search for b, skipping a's and Y's;
- If found, mark found b with Y, and then search for c, skipping b's and Z's;
- If found, mark found c with Z, and then go back left to first X.

Turing Machines as Computing Output

Turing machines can also use the tape as an *output*. Let $M = (Q, \Sigma, \Gamma, \delta, q_0)$ be a Turing machine. A string $u \in \Sigma^*$ is *accepted by halting* if M with input u halts.

Theorem: Every language L accepted by a TM by halting is accepted by a TM by final state and vice versa.

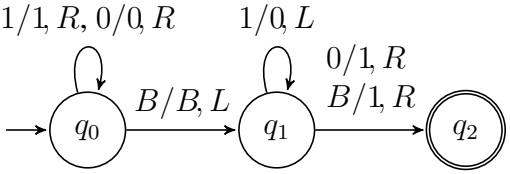
Proof: ($\Rightarrow$) Make every state of the TM accepting L by halting a final state. ($\Leftarrow$) To a TM which accepts L by final state, add

1. for each non-final state q_i, if $\delta(q_i, x)$ is undefined, then $\delta(q_i, x) = (q_f, x, R)$
2. for each $x \in \Gamma$, $\delta(q_f, x) = (q_f, x, R)$.

Note that $M(w)$ is defined only if M halts on w; this point will be important later. Consider a machine add 1 to a binary representation of an integer.

- Move to the right end of the input string, that is, to the least significant digit.
- If 0 read, replace it with 1
- If 1 read, replace it with 0 and move left
- If B read, replace it with 1 and shift 1 place to the right

This gives us the machine below



Unlike NFAs or DFAs, Turing machines may loop forever. Note that if L is recursive, then L is recursively enumerable. However, there are recursively enumerable languages which are not recursive. Examples of such languages will follow soon.

Variations of Turing Machines

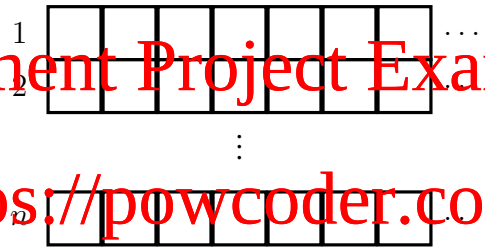
Turing machines appear to be simple but dumb! How can we improve them? Some possibilities are to include:

- Multiple "tracks" on a tape.
- Two-way infinite tape.
- Multiple tapes.
- Two-Dimensional tapes.
- Non-determinism.

None of these improve on the original model!

Multitrack Turing Machines

The tape is divided into n tracks



The machine reads an entire tape position. Two-track transitions,

$$\delta(q_i, [x, y]) = (q_j, [z, w], d), \quad d \in \{L, R\}$$

means:

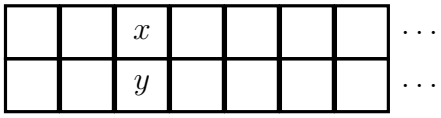
- x on track 1 and y on track 2;
- replaced with z on track 1, w on track 2

A two-track machine clearly simulates a one-track machine (just ignore the second track). To simulate a two-track machine M on a one-track machine M', encode each *pair* of symbols in M's tape by a *single* symbol on M's tape. Hence we construct M' as

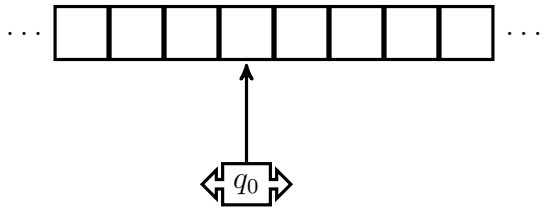
$$M' = (Q, \Sigma \times \{B\}, \Gamma \times \Gamma, \delta', q_0, F)$$

and

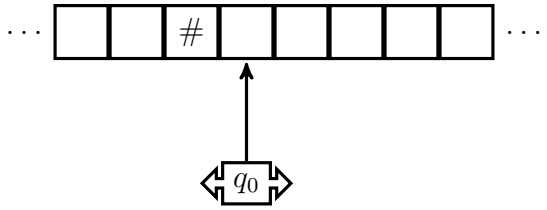
$$\delta'(q_i, [x, y]) = \delta(q_i, [x, y])$$



Two-Way Tape Turing Machines

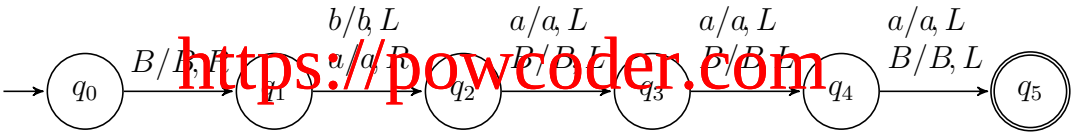


Two-way tape TMs can simulate a standard TM.

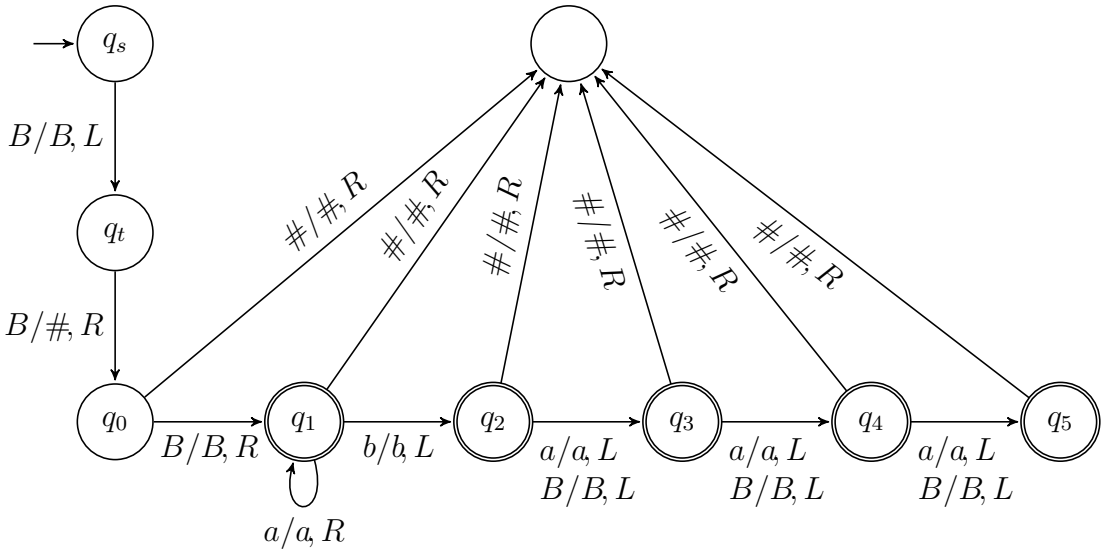


New tape symbol # represents the left end of the tape. When the machine reads the symbol # it enters a non-accepting state, *simulating* the result of terminating abnormally, i.e. moving off the left end of the tape.

Machine accepting strings over $\{a, b\}$ with first b preceded by 3 or more a's.

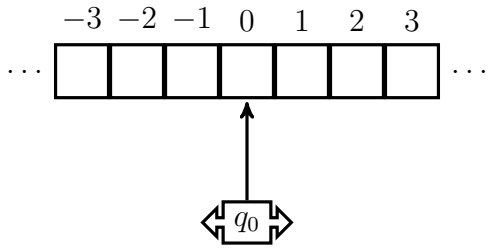


Two-way tape version:

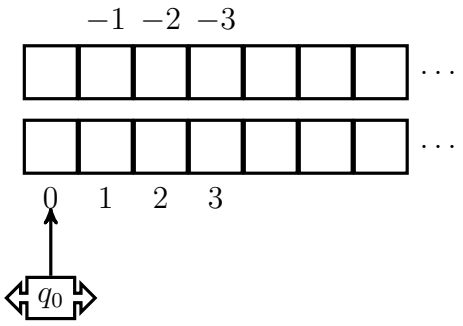


A standard TM can simulate a two-way tape TM (actually use a two-track TM).

Tape of a two-way tape TM:



Tape of a two-track TM:



Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a two-way tape TM. $M' = (Q', \Sigma, \Gamma \cup \{\#\}, \delta', q_0, F')$ is a two track TM.

- States Q' are states of Q plus U (up) or D (down) meaning whether we are reading the upper or lower track.
- Transitions δ' reflect δ but either processing the upper track or lower track.
- # marks the boundary between upper and lower track.

Two-Way Tape Turing Machines

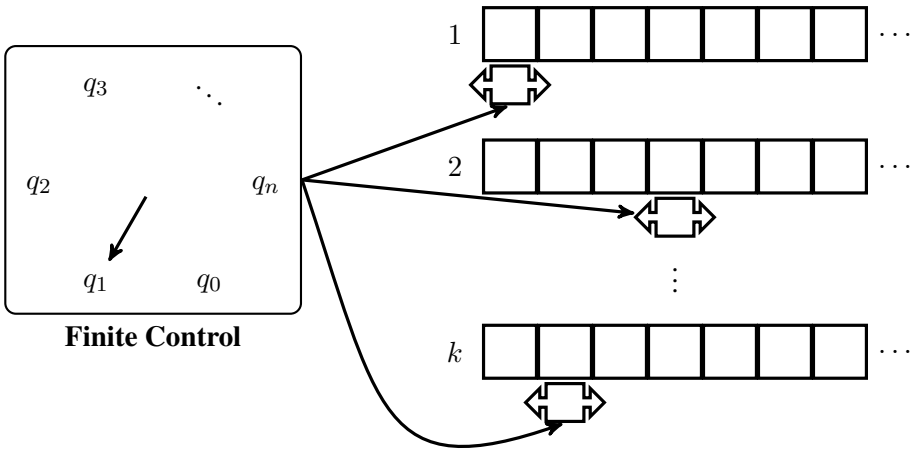
States of M': $[q_i, D]$ and $[q_i, U]$. Start state $[q_s, D]$. New transition function δ'

1. $\delta'([q_s, D], [B, B]) = ([q_s, U], [B, \#], R)$. Place # on tape.
2. $\delta'([q_s, U], [x, B]) = ([q_0, D], [x, B], L)$ for all x. Return to position 0.
3. $\delta'([q_i, D], [x, z]) = ([q_j, D], [y, z], d)$ for all $z \in \Gamma - \{\#\}$ if $\delta(q_i, x) = (q_j, y, d)$. Simulate right of tape.
4. $\delta'([q_i, U], [z, x]) = ([q_j, U], [z, y], d')$ for all $z \in \Gamma - \{\#\}$ if $\delta(q_i, x) = (q_j, y, d)$ where d' is reverse of d. Simulate left of tape.
5. $\delta'([q_i, D], [x, \#]) = ([q_j, U], [y, \#], R)$ if $\delta(q_i, x) = (q_j, y, R)$. Crossing right to left.
6. $\delta'([q_i, U], [x, \#]) = ([q_j, D], [y, \#], R)$ if $\delta(q_i, x) = (q_j, y, R)$. Crossing left to right.
7. $\delta'([q_i, D], [x, \#]) = ([q_j, D], [y, \#], R)$ if $\delta(q_i, x) = (q_j, y, R)$. Simulate 0
8. $\delta'([q_i, U], [x, \#]) = ([q_j, U], [y, \#], R)$ if $\delta(q_i, x) = (q_j, y, L)$. Simulate 0

Multitape Turing Machines

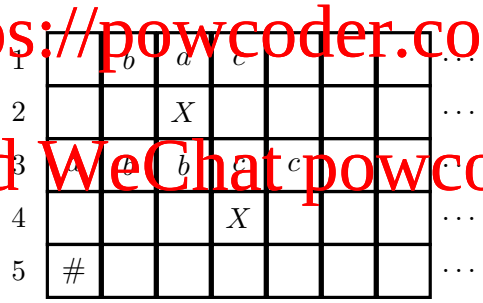
A k-tape machine has:

- k tapes and k *independent* heads:



- Transitions:
  - change the state;
  - write a symbol on each tape;
  - independently reposition tape heads: left, right, stationary.
  - Transition $\delta(q_i, x_1, x_2) = (q_j; y_1, d_1; y_2; d_2)$:
    - reads x_1 on tape 1 and x_2 on tape 2;
    - writes y_1 on tape 1 and y_2 on tape 2;
    - moves tape 1 to direct on d_1, move tape 2 to direction $d_2 \in \{L, R, S\}$.

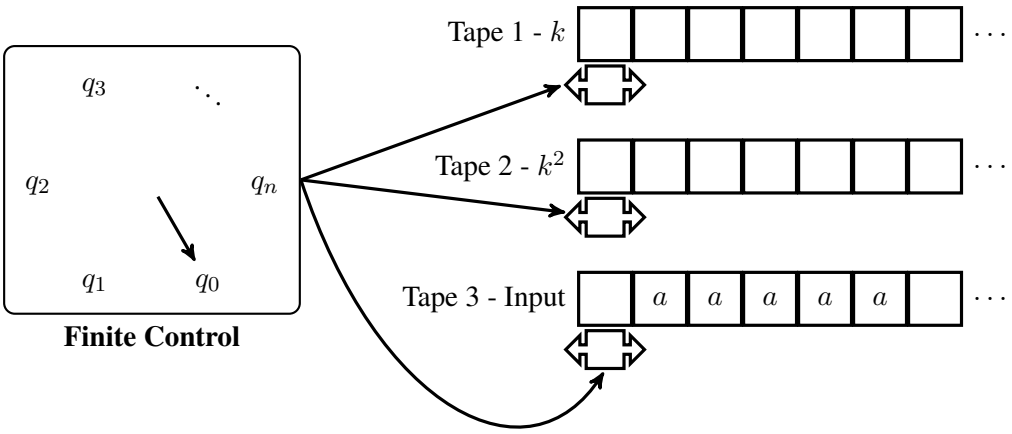
Simulate a two tape machine with a fivetrack machine:



States are $(s, q_i, x_1, x_2, y_1, y_2, d_1, d_2)$; initially $(f1, q_i, U, U, U, U, U, U)$.

1. Find symbol x_1 on first tape, store in state $(f1, q_i, x_1, U, U, U, U, U)$;
2. Find symbol x_2 on second tape, store in state $(f2, q_i, x_1, x_2, U, U, U, U)$;
3. Enter state $(p1, q_j, x_1, x_2, y_1, y_2, d_1, d_2)$;
4. Print symbol y_1 and move first tape;
5. Print symbol y_2 and move second tape;
6. New state $(f1, q_j, U, U, U, U, U, U)$.

$\{a^k \mid k = n^2, n \geq 0\}$



1. Write X onto tape 2 and 3;
2. Check the length of input versus tape 2,
   - if *equal* halt with success;
   - if *less* halt with failure.
3. Add an X to tape 3;
4. Add two copies of tape 3 to tape 2;
5. Add an X to tape 3 [$(k+1)^2 = k^2 + 2k + 1$];
6. Repeat (from check).

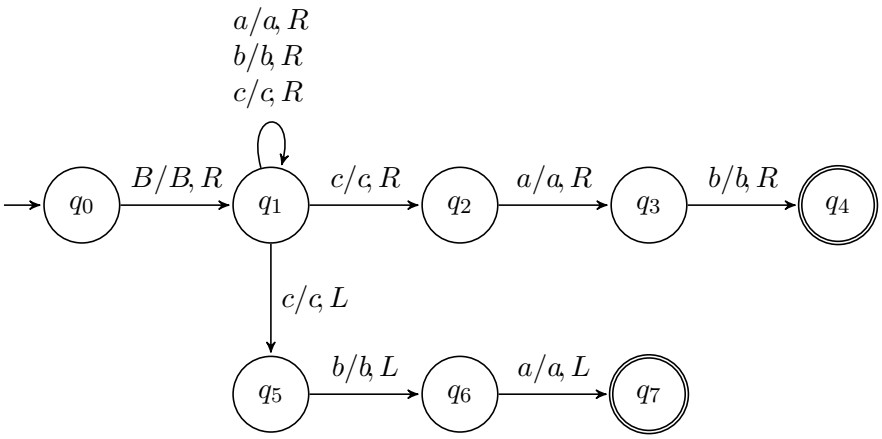
Nondeterministic Turing Machines

Nondeterministic Turing Machines allow a finite number of transitions to be specified for a given (state, tape symbol) pair.

$$\delta(q_i, x) = \{(q_j, y, d), (q_k, z, e)\}$$

$$\delta : Q \times \Gamma \to \wp(Q \times \Gamma \times \{L, R\})$$

Nondeterministic TM for strings with a c preceded or followed by ab:



- Maximum n of different transitions for (state, symbol) pair.
- A sequence $(m_1, m_2, \ldots, m_k)$ where $1 \leq m_i \leq n$ defines a possible computation (of k or fewer transitions).

q_0	B	$1:(q_1,B,R)$	q_1	b	$1:(q_1,b,R)$
		$2:(q_1,B,R)$			$2:(q_1,b,R)$
		$3:(q_1,B,R)$			$3:(q_1,b,R)$
q_1	a	$1:(q_1,a,R)$	q_1	c	$1:(q_1,c,R)$
		$2:(q_1,a,R)$			$2:(q_2,c,R)$
		$3:(q_1,a,R)$			$3:(q_5,c,L)$
q_2	a	$1:(q_3,a,R)$	q_3	b	$1:(q_4,b,R)$
		$2:(q_3,a,R)$			$2:(q_4,b,R)$
		$3:(q_3,a,R)$			$3:(q_4,b,R)$
q_5	b	$1:(q_6,b,L)$	q_6	a	$1:(q_7,a,L)$
		$2:(q_6,b,L)$			$2:(q_7,a,L)$
		$3:(q_6,b,L)$			$3:(q_7,a,L)$

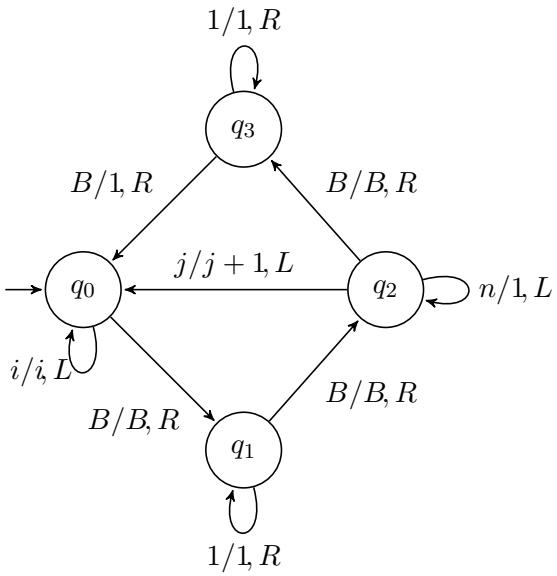
Derivations for sequence (1,1,1,1,1), (1,1,2,1,1), (2,2,3,3,1)

	$q_0BacacB$	1		$q_0BacacB$	1		$q_0BacacB2$
$\vdash$	Bq_1acabB	1	$\vdash$	Bq_1acabB	1	$\vdash$	$Bq_1acabB2$
$\vdash$	Baq_1cabB	1	$\vdash$	Baq_1cabB	2	$\vdash$	Baq_1cabB3
$\vdash$	$Bacq_1abB$	1	$\vdash$	$Bacq_2abB$	1	$\vdash$	Bq_5acabB
$\vdash$	$Bacaq_1bB$	1	$\vdash$	$Bacaq_3bB$	1		
$\vdash$	$Bacabq_1B$		$\vdash$	$Bacabq_4B$			

Simulate a nondeterministic Turing machine M with a three tape TM:

1. Sequence $(m_1,\ldots,m_k)$ written on tape 3.
2. Input on tape 1 is copied onto tape 2.
3. The computation of M using sequence on tape 3 is simulated on tape 2.
4. If the simulation halts, the computation halts and accepts input.
5. A new sequence is generated on tape 3. Repeat 2-5.

TM (deterministic) for generating $\{1, 2, \ldots, n\}^*$



Thus the deterministic version is exponentially slower than the nondeterministic one. Is this exponential factor necessary? Is there a more efficient way? Does $P = NP$?

Limitations of Turing Machines

Turing machines are powerful, but in practice no one would program this way. It is possible to introduce more convenient languages (such as the random access model in the book, pp. 210-221). This is more like a real programming language, but makes negative results harder to prove.

Turing machines are sometimes criticised for not being able to model *communication*. Such extensions are often desired to model systems such as the Internet. Process algebras and concurrent games are often used as ways to model such complex interactive systems.

Extensions of Turing Machines

As above, there are many ways to extend the operations of a Turing machine. However, none have yet been shown to be more powerful. Quantum computing may change this, although it is too early to tell what will happen here. Modelling interactive computation is another interesting direction of research.

Section 6: Computability

One of the aspects of computer science is to understand what is our limit. What can be computed? What are the limits of computation? Are there problems for which no algorithms can ever exist? How can we show this? Intuitively, computation has to be,

- complete (i.e. cover all cases),
- mechanistic (i.e. precise),
- deterministic (i.e. same input always leads to the same result).

Some problems are obviously too vague to be solvable by an algorithm. For e.g. *'How big is the universe'*? Some practical problems are hard, although easy to define precisely. For e.g., *'Can a compiler determine if a program contains the possibility of an infinite loop?'* or *'Are there any uninitialised variables'*?

Intuitively any statement should either be *true* or *false* (provided we have all the facts). Some problems **do not have any algorithmic solution!** These are called *undecidable*. Some problems are *semi-decidable*; we can determine when a property **is** satisfied, but not when it is not.

For example, is there an odd perfect number (a perfect number equals the sum of its proper divisors, e.g. 6 = 1+2+3)?
Is 1 perfect?
Is 3 perfect?
Is 5 perfect?
...

Church-Turing thesis

Standard Turing machines are mechanistic and deterministic. If a Turing machine M halts on all inputs, it is also complete. Hence, Turing machines which always halt correspond to the intuitions above. However, there can be many similar formal models of the informal notion of computable:

- Turing machines
- Grammars
- Numerical functions
- λ-calculus
- ...
- + more yet to be thought of ...

All the following are equivalent to standard Turing machines:

- multiple tapes
- two-way infinite tapes
- multiple heads
- two-dimensional tapes
- random access
- nondeterministic machines
- unrestricted grammars
- computable functions
- λ-calculus
- + every formal model yet found!

Church-Turing thesis: All *possible* models of computation are equivalent to a standard Turing machine. Strong statement – any *future* models of computation will also be no more powerful than a standard Turing machine! However, this statement,

- cannot be proved; it is **not** a theorem!
- can only be experimentally verified;

- can potentially be disproved, but no-one considers it likely.

Will quantum computing disprove this? Too early to tell perhaps! Assuming the Church-Turing thesis, a computation that cannot be done on a standard Turing machine $\Rightarrow$ cannot be computed in any sense.

Universal Turing Machine

So how good are Turing machines? What kind of limitations do they have? How do we explore them? Turing machines are like hardware — each has a fixed program. As we can represent Turing machines as strings, we can even use them as input to other Turing machines. These 'other' machines are known as *Universal Turing Machines*. A *Universal Turing Machine*

- takes another Turing machine M as input;
- simulates the actions of M;
- model for "real" computers (i.e. input is both data *and* instructions);
- one of the top 10 ideas of the 20th Century;

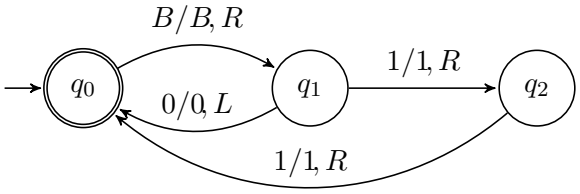
How do we represent a Turing machine as a string? Assume that the input alphabet of M is $\{0, 1\}$, and the tape alphabet is $\{0, 1, B\}$.

Symbol	Encoding
0	1
1	11
B	111
q_0	1
q_1	11
.	.
.	.
.	.
q_i	1^{i+1}
L	1
R	11

We encode the transition $\delta(q_i, x) = (q_j, y, d)$ as $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$. 00 is used to separate transitions. 000 indicates the beginning and end of the representation.

The Universal Turing Machine

For example, the TM below accepts the empty string or strings commencing with 0 or 11.



Transition	Encoding
$\delta(q_0, B) = (q_1, B, R)$	101110110111011
$\delta(q_1, 0) = (q_0, 0, L)$	1101010101
$\delta(q_1, 1) = (q_2, 1, R)$	110110111011011
$\delta(q_2, 1) = (q_0, 1, L)$	1110110101101

Machine is represented as

000101110110111011001101010101001101101110110110011101101011010000

We can build a machine to recognise valid encodings which,

- begin and end with 000;
- in between a finite sequence of encoded transitions are separated by 00;
- encoded transition is 1's separated by 0.

The encoded machine is deteministic if the first two parts of each encoded transition are never repeated. Hence a representation of a Turing machine with input alphabet $\{0,1\}$ is itself a string over $\{0,1\}$, and may be used as an input to another Turing machine. A universal Turing machine U with input $R(M)w$ simulates M with input w.

U is a three-tape machine:

- Tape 1 contains the input;
- Tape 2 contains the current state of M;
- Tape 3 simulates action of M on w.

This machine functions as below:

1. If the input is not $R(M)w$, U moves to the right forever.
2. Write w on tape 3.
3. Write 1 (= $en(q_0)$) on tape 2.
4. Let x be the symbol on tape 3, q_i the encoded state on tape 2. Look for matching transition on tape 1. If none, halt and accept.
5. $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$ on tape 1.
   - $en(q_i)$ is replaced by $en(q_j)$ on tape 2.
   - y is written on tape 3.
   - The tape head of tape 3 is moved according to d.
6. Repeat steps 4-5.

Note that U has two degrees of freedom:

1. Choice of M.
2. Choice of w.

Note that w could be "M"! (ever used a compiler to compile itself?) This may be somewhat silly, but it is perfectly possible. For that matter, it is possible for M to be U itself, and hence we can simulate the action of U on itself using a representation of itself as input ...

U("U" "U") = ????

Paradoxes

"This sentence is false" or "I am lying"
If this is true, then it is false ...
If this is false, then it is true ...

A king decrees:

1. All men who *do not* shave themselves **must** be shaved by the barber.
2. All men who *do* shave themselves **must not** be shaved by the barber.

So who shaves the barber?

- Barber shaves himself ⇒ not shaved by Barber ⇒ doesn't shave himself.
- Barber doesn't shave himself ⇒ shaved by Barber ⇒ shaves himself ????

Resolutions:

1. Don't shave!
2. Exempt the barber from the rules
3. The barber is a woman ...

Self-Reference

- The problem with many paradoxes is that they are self-referential.
- If we avoid self-referential statements logic and mathematics don't have paradoxes.
- However in computation it is sometimes hard to avoid self-referential statements.
- The "Halting Problem". *Give an algorithm to determine if a given turing machine halts on a given input* is self-referential.

The Halting problem

Suppose we had a Turing machine halts(P, X), which was able to take a program P and input X and detemine whether or not P halts on input X. This machine

- *always* answers correctly;
- *always* terminates.

Assuming we can represent the source code for P as input, we could define another program silly as: silly(P). If halts(P,P) then loop forever else halt. Note that silly(P)

- *doesn't halt* if P halts on itself as input;
- *does halt* if P doesn't halt on itself as input.

Crunch: What happens to silly(silly)?

- silly(silly) doesn't halt if silly(silly) halts;
- silly(silly) does halt if silly(silly) doesn't halt.

This is a contradiction, like the paradox of the barber. We conclude that halts(P, X), as specified, cannot exist.

The Halting Problem: Given an arbitrary Turing machine M, does the computation of M with input w halt? This is *undecidable*; there is no algorithm for this problem. Hence the universal Turing machine will sometimes not terminate (i.e. there's no 'smart' version of U which always detects loops). A solution to this problem must work for *all* possible Turing machines. It is possible to solve the problem in particular cases, but not in general. This result was first shown by Turing in the 1930's.

An Undecidable Problem

Theorem: The Halting Problem is undecidable.
Proof: Assume that there is such a Turing machine called H. A string is accepted by H if it is the encoding of a TM M with input w and M halts on input w. We write this as halts$(R(M), w)$ (i.e. H with the input being $R(M)w$). Note that as $R(M)$ is a string over $\{0, 1\}$, w could be $R(M)$, so that we could see if M halts with its own representation as input.

Define the machine strange as follows:

strange$(R(M))$
    if halts$(R(M), R(M))$ then loop forever
    else halt

This halts when M does not halt on $R(M)$, and does not halt when M halts on $R(M)$. We can clearly construct a TM S that implements strange by using H. Consider the action of S with input being $R(S)$, i.e. strange$(R(S))$. Then,

- S with input $R(S)$ halts if S with input $R(S)$ does not halt
- S with input $R(S)$ does not halt if S with input $R(S)$ halts

This is a contradiction. Hence, there is no such Turing machine H, and so the Halting Problem is undecidable.

Corollary: The language $L_H = \{R(M)w \mid R(M)$ is the representation of a TM M and M halts with input $w\}$ is not recursive.

Note the strength of this result – as there can be no standard Turing machine for the halting problem, by the Church-Turing thesis, this is the same in *any* model of computation! We have just shown that the Halting problem is *undecidable*. We can generate further undecidable problems via *reduction*.

- If P is reducible to P' and P' is decidable, then P is decidable.
- If P is reducible to P' and P is undecidable, then P' is undecidable.

Why? If P' is decidable, with say M' deciding it, then we can decide P by transforming it to P' and using M'.

Undecidable problems for machines

- Does M halt on w?
- Does M halt on an empty tape?
- Is there any string on which M halts?
- Does M halt on every string?
- Do M_1 and M_2 halt on the same strings?
- Is $L(M)$ regular? context-free? recursive?

The first case was proved above. The others follow from it by reduction.

The Halting problem

- Arises when we try to construct a Turing machine that can determine if ANY Turing machine halts on ANY input.
- We can solve the the halting problem for many Turing machines on many inputs, but not for ALL Turing machines.

- Being unable to write such a Turing machine is not terribly interesting. However many practical problems can be reduced to the Halting problem.
- The same problem as writing a machine to determine if an **arbitrary** program in C or Java or ... has an infinite loop.

- Also equivalent to writing a program to determine whether an arbitrary program contains any uninitialised variables - i.e. *is a variable ever read before it is written?*

Undecidable problems for grammars

As Turing machines are equivalent to unrestricted grammars, we expect undecidable problems there. Let G be an unrestricted grammar.

- Is $w \in L(G)$?
- Is $e \in L(G)$?
- Is $L(G_1) = L(G_2)$?
- Is $L(G) = \emptyset$?

Perhaps surprisingly, there are some context-free language properties which are undecidable:

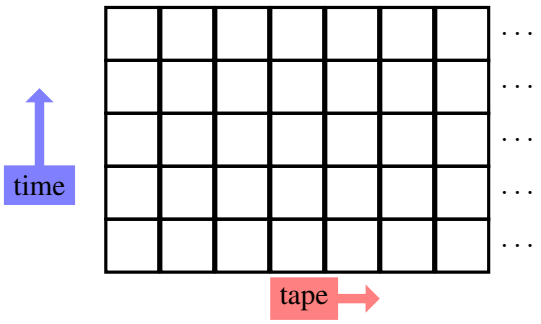
- For a context-free grammar G, is $L(G) = \Sigma^*$?
- For context-free grammar G_1, G_2, is $L(G_1) = L(G_2)$?
- For PDAs M_1 and M_2, is $L(M_1) = L(M_2)$?
- For PDA M, does M have the minimum number of states?

Some properties are decidable for CFGs, though — $w \in L(G)$ and $L(G) = \emptyset$ are decidable (as we have implicitly seen earlier).

Tiling Problem 1 Revisited

Tiling Problem 1: Given a set of tile designs T, can we cover *any* rectangular area using only designs from T? Rotation is **not** allowed. The solution **must** be able to cover all possible rectangular shapes.

This problem is undecidable. We show this by constructing a Turing machine which halts iff there is no tiling. We generate tiling from the Turing machine by taking configurations of M's and placing them on top of each other.



Edge rules:

- horizontal: elements of configuration
- vertical: state information, if present

The idea is that a tiling pattern corresponds to an infinite computation in M. If there is no such tiling, then M will eventually halt (and hence get 'stuck'). The details of the argument can be found in Lewis & Papadimitrou, pp. 263-6.

Limitations

Undecidable problems are often interesting and useful. Whilst termination of programs dealing with such problems cannot be guaranteed, this is often not a problem in practice. Programmers generally write programs which terminate. The only tricky part tends to come with programs which take other programs as input.

There are actually classes of undecidable problems — the differences being what information is needed in order to decide the problem.

Section 7: Complexity (revisited)

The reason for the high complexity of many programs is due to the number of alternative solutions. For example, in a complete graph with n vertices, there are $n!$ circuits in the graph. Hence searching all the possibilities takes $O(n!)$ time.

It is often useful to analyse such problems by asking "If some oracle found me a path, how hard would it be to check it?". In other words, how much of the complexity of the problem is tied up in the search through the alternatives. One way to express this is through *nondeterminism*. Essentially, an algorithm is nondeterministic if it makes an unspecified choice at some point. One way to think of this is: what is the best possible choice that I could make?

For e.g., cost of minimal path from A to B via way-points can be achieved by

1. chossing minimal path;
2. calculating path cost;
3. outputting an answer.

Note: This is purely about how to specify the algorithm. It is **not** about reducing the cost of running the algorithm.

In terms of cost, it corresponds to the cost of *checking the answer*. In the words of a former Computing Theory student: "*Nondeterminism means never having to say that you are wrong*". Nondeterministic computations can have complexities which appear much lower than determinisitic ones.

Problem: Is n a prime?

Deterministic solution: Test all numbers $2, 3, \ldots n-1$ as divisors of n. This has complexity $O(n)$, as we may test all such numbers and find that none of them divide n.

Nondeterministic solution: Pick a number in the range $2, 3, \ldots n-1$ and test if it is a factor of n. If it is a composite number, then one of these choices will work. Hence, the cost is that of checking the answer rather than finding one.

Nondeterminism often seems puzzling at first — why introduce "magic" like this? The reason is that it

- simplifies specification of problems and algorithms;
- can often be "compiled away" (i.e. replaced with a deterministic procedure);
- localizes costs;
- can sometimes help the discovery of more efficient algorithms.

If you like, nondeterminism is to determinism as *specification is to computation*.

Nondeterminism	Determinism
Specification	Computation

A problem is *decidable in nondeterministic polynomial time* if there is a nondeterministic program that solves the problem which has time complexity $O(n^r)$ for some r. The class of all such problems is denoted as $\mathcal{NP}$. As a deterministic program is a special case of a nondeterministic program, $\mathcal{P} \subseteq \mathcal{NP}$.

What about the other direction, i.e. is $\mathcal{NP} \subseteq \mathcal{P}$? **No one knows for sure!**. It is strongly suspected that $\mathcal{P} \subset \mathcal{NP}$, but there is no proof for this (yet!). It is possible to convert a nondeterministic program to a deterministic one (see later), but this does not preserve polynomial time complexity (i.e. the deterministic program takes exponential time).

A Problem in NP

A *Boolean variable* is a variable which may have values 0 or 1. A *clause* is a disjunction of variables or their negations. For example, $x \vee \neg y$ and $x \vee y \vee \neg z$ are clauses. A formula is in *conjunctive normal form* if it has the form $u_1 \wedge u_2 \wedge \ldots \wedge u_n$ where each u_i is a clause.

The *satisifiability problem* is to determine if a given formula in conjunctive normal form is satisfied by a truth assignment. For example, $(x \vee y) \wedge (\neg y \vee \neg z)$ is satisfied by the assigment $x = 1, y = 0, z = 0$. Note that $\neg x \wedge (x \vee y) \wedge (\neg y \vee x)$ is not satisfied by this assignment (nor by any other).

The satisfiability problem can be shown to be in $\mathcal{NP}$ (very long!). For n variables, there are 2^n possible truth assignments. Hence, exhaustive search is exponential, but checking a particular truth assignment is polynomial.

Key Idea:

A problem is in $\mathcal{NP}$ if a proposed solution to it can be *checked* in polynomial time. A problem is in $\mathcal{P}$ if a solution to it can be *found* in polynomial time.

Hence if $\mathcal{P} = \mathcal{NP}$, finding a solution is not essentially more difficult than checking one. Fame (and fortune?) awaits anyone who can give a polynomial time algorithm for satisfiability (or for various other problems in NP)

More Problems in NP

There are many other problems which fall in $\mathcal{NP}$. For example,

- 3-satisifiability: the satisfiability problem in which each clause has exactly three literals (i.e. a variable or its negation).
- vertex cover: A *vertex cover* of the nodes of a graph G is a set of nodes VC such that for each arc (u, v) in G, at least one of u and v is in VC. The *vertex cover problem* is to determine if a graph has a vertex cover with k vertices. zz
- Hamiltonian circuit: A *Hamiltonian circuit* is a path in a graph which visits each node exactly once and returns to its starting point. The *Hamiltonian circuit problem* is to determine if a given graph has a Hamiltonian circuit.
- Travelling Salesman problem: Given a complete graph with weights (i.e. all points connected with a distance on each edge), find the minimal path which visits every node and returns to the starting point.
- Integer linear programming + many others (see Garey & Johnson).

"Most" interesting and useful problems seem to be NP-complete (or worse!).

Problem reduction

We classify new problems by *reducing* them to known problems. To solve problem X, transform the problem to an input to a known problem Y, and then solve Y.

For example, to solve the Hamiltonian circuit problem, construct a new graph from the old one as follows:

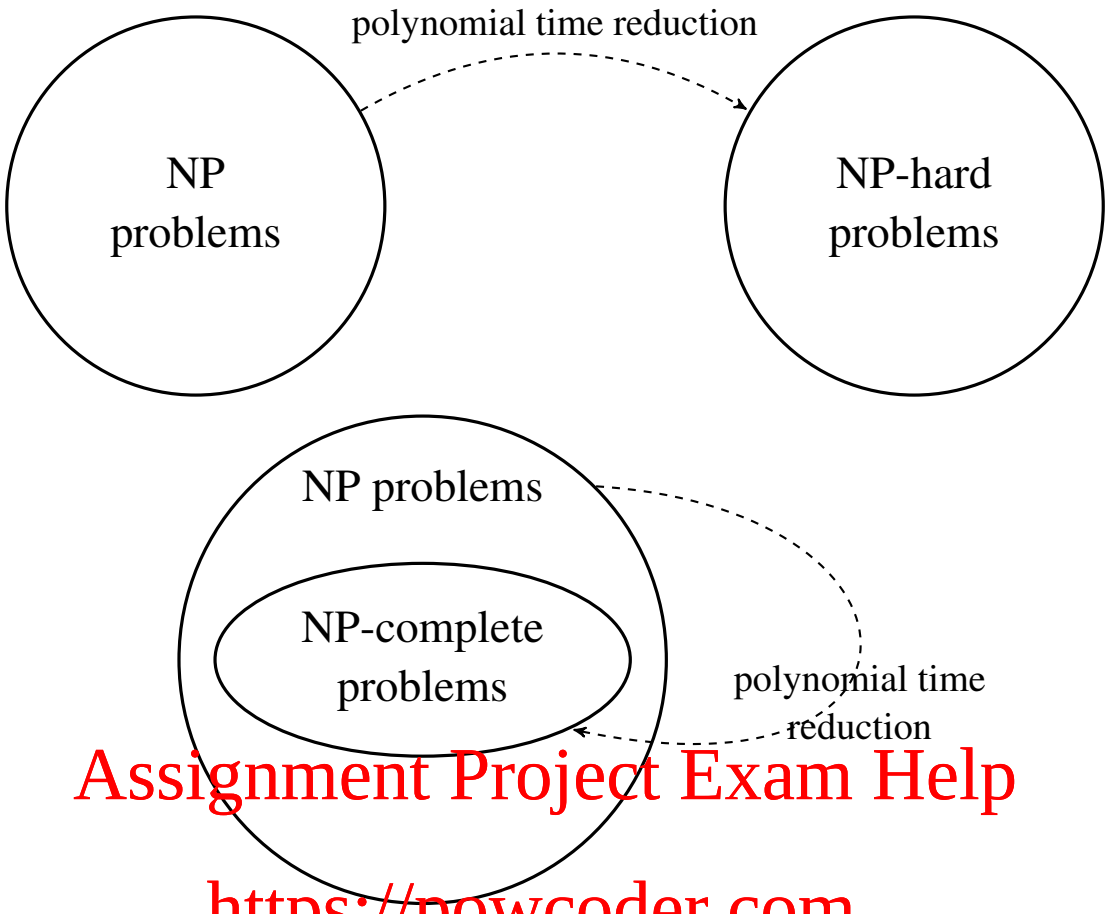
- for each edge in the original graph, add an edge with weight 1 to the new graph;
- for all other pairs of vertices in the original graph, add an edge with weight 10 to the new graph.

Clearly there is a Hamiltonian circuit in the original graph iff there is a tour of weight $n + 1$ or less in the new graph. Hence we solve the Hamiltonian circuit problem by *reducing* it to the TSP.

NP-Completeness

Remarkably, all of the problems above can be reduced to each other and, moreover, the transformation can be done in polynomial time. Hence a polynomial-time solution to one such problem is a polynomial-time solution to **all** of them!

These problems represent completely the whole set of NP problems in that **all** NP problems can be reduced to them in polynomial time. They are called NP-complete. Not all hard NP problems (i.e. ones we don't have a polynomial algorithm for) are NP-complete.



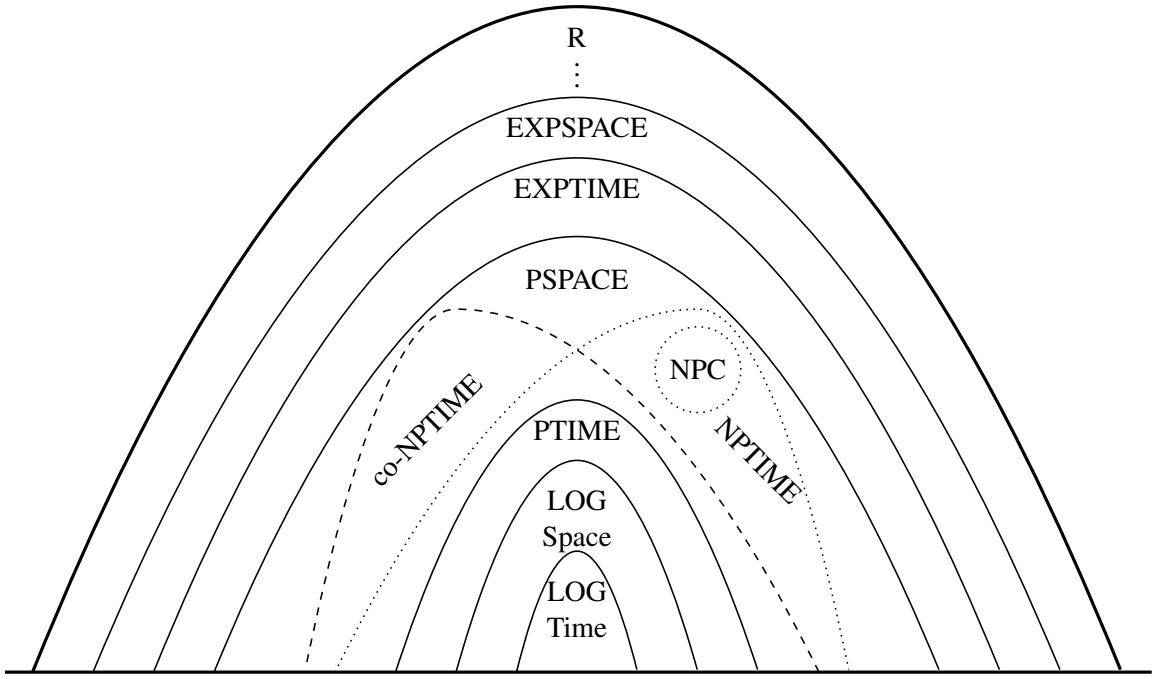
NP-complete problems:

- have solutions which can be *checked* in polynomial time;
- a solution cannot necessarily be *found* in polynomial time;
- stand or fall together (i.e. either all exponential or all polynomial);
- "straddle" border between tractable and intractable.

The first problem shown to be NP-complete was the Boolean satisfiability problem. This was done by Stephen Cook in 1971 and is considered one of the most important results in computer science. This has a very complex proof (which we will not cover).

Since then, other problems are shown to be NP-complete by finding a reduction of some NP-complete problem to them. All of the problems in NP mentioned above are NP-complete. The currently known number of NP-complete problems is well over 1,000. See the book by Garey and Johnson for more details.

Complexity Classes



Algorithm Classes

Undecidable problems
(no algorithms)

Intractable problems
(algorithms worse than polynomial)

NP-complete problems???

Tractable problems
(algorithms polynomial or less)

Dealing with NP-Completeness

The traveling salesman problem is a hard problem, for which:

- a naive algorithm is $O(n!)$;
- best known algorithm is $O(2^{\sqrt{n}})$;
- it is suspected that no tractable algorithm exists.

How to solve it in practice? There are:

- approximation algorithms: settle for less than optimum;
- heuristic algorithms: probably efficient;

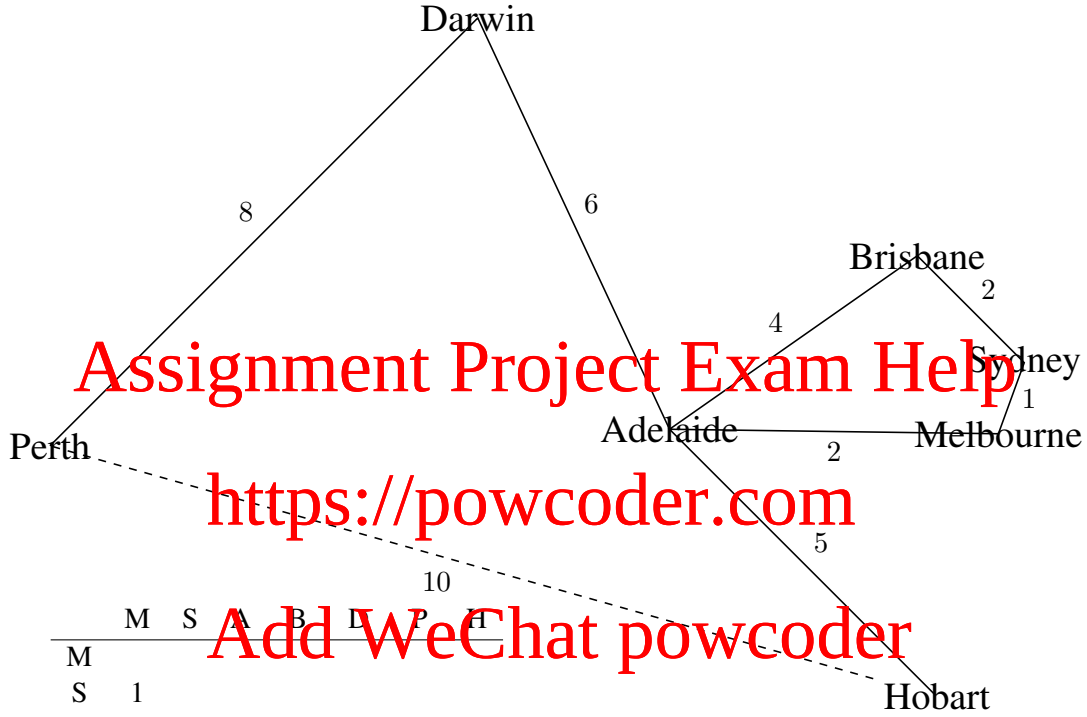
- randomised algorithms: probably correct.

Approximation Algorithms

Such algorithms, relax the constraints — settle for sub-optimal (but not bad) solutions.

- $O(n^2)$ algorithm with solution $\leq 2\times$ optimum:
  - finds minimal spanning tree ($O(n^2)$);
  - preorder traversal ($O(n)$);
  - "forgets" some edges.
- $O(n^3)$ algorithm with solution $\leq 1.5\times$ optimum (more complex than above).

Hence can trade quality for speed.



	M	S	A	B	D	P	H
M							
S	1						
A	2	3					
B	4	2	4				
D	8	7	6	6			
P	8	9	6	10	8		
H	2	3	5	7	10	10	

Output: Melb-Syd-Bris-Adel-Darwin-Perth-Hobart, cost 33
Min. path is 29

Heuristic Algorithms

A heuristic based algorithm tries to make "good" choices. Some special cases of the TSP have the following properties:

- graph is a tree;
- triangle inequality holds: d(A,B) $\leq$ d(A,C) + d(C,B) for all C;
- planar graph with Euclidean distance.

Heuristics attempt to solve the simple cases first and only use the full version when these fail. Heuristics are most useful when worst-cases are rare and average-cases occur often.

Limitations

The classic "big O" concept used here is the worst-case complexity. This gives a guaranteed upper bound, but can be a little crude. The *simplex algorithm*, used in linear programming, is exponential, but in practice most cases are not, and this algorithm is widely used in real systems.

Average-case or minimal-case complexity is generally more useful, but more difficult to derive. Some cases are known — for example, it is known that sorting takes *at least* $\mathrm{O}(n \log n)$ operations, and also at most $\mathrm{O}(n \log n)$ operations (for mergesort).

Randomised Algorithms

There are some other ways of finding an answer, for e.g., with high probability, but not certainty.

Probabilistic algorithms:

- incomplete search;
- can measure the maximum error.

Other forms of randomized algorithms make random choices in various ways.

- local improvement;
- simulated annealing;
- genetic algorithms.

Genetic Algorithms

Genetic algorithms are often:

- a general search technique;
- applied when exhaustive search is too hard;
- modelled on human genetics;
- randomized, but far from totally random.

There are three main elements:

- chromosomes to represent the problem (e.g. paths in a graph);
- fitness function to measure quality of chromosomes;
- mutation rate for certain random fluctuations.

The process is as follows:

Step 1 Start with initial set of chromosomes.
Repeat steps 2-5 until satisfied.
Step 2 Use fitness function to select "good" chromosomes.
Step 3 Bias parent selection towards good chromosomes.
Step 4 Swap genetic material between parents.
Step 5 Mutate some genetic material.

However, there are certain problems with such an approach, such as:

- setting parameters,
- details of swapping process, and
- knowing when to stop!

TSP solutions have had some considerable success with such methods.

Extensions

There is an amazingly intricate hierarchy of complexity classes (parallel classes, probabilistic classes, "oracle" classes, ...) — we have only scratched the surface here.

A new direction is *parametric complexity*, which can be thought of as making more detailed analyses of algorithms in order to explain properties such as "exponential in theory, efficient in practice".

Another interesting notion is *Kolgomorov complexity* — the minimum length encoding of the problem. For example, the even numbers are a simpler class than the digits of π, as the former have a very simple recogniser, but the latter can only be represented by an infinite string of digits.

There are people working on the $\mathcal{P} = \mathcal{NP}$ question — we may have an answer one day!

Complexity Summary

- Measured by worst-case time on a given length input
- Measure rate of growth of time with respect to length of input
- Tractable $\equiv$ polynomial time or less
- Intractable $\geq$ polynomial time
- Undecidable problems are problems with no algorithm
- NP problems are an interesting subset of (probably) intractable problems.
- To deal with hard problems we can approximate the problem, use heuristics and use probabilistic algorithms.

Section 8: Cryptography (revisited)

Probabilistic Prime Testing

Primality testing is an important problem in computer science and has many applications. We generally tend to ask 'is n prime'? It was shown to be in $\mathcal{P}$ in 2002 (see "Primes is in $\mathcal{P}$" section of the printed notes). It was known for a long time before that that there is a polynomial time algorithm if a deep mathematical conjecture is true (the extended Riemann hypothesis). Note that factorisation is *not* $\mathcal{NP}$-complete!

There exists a probabilistic algorithm as below:

1. Pick a number, k, less than n.
2. If k is a factor ofll n, n is not prime. Halt with failure.
3. Choose another value of k and repeat step 2.
4. When sufficient values of k have been tried, halt with probabilistic success.

The possible outcomes of such an approach are:

- definitely composite;
- probably prime, with a specific probability.

In the algorithm above, choice of k is crucial. It is possible to choose k such that the probability of correctness is $1 - \frac{1}{2^n}$ for n trials (Solovay-Strassen test). Hence, we can increase our confidence in the result by increasing n. The procedure for choosing k is a bit complex (see Harel's book). There is a similar procedure with correctness $1 - \frac{1}{4^n}$ for n trials (Rabin-Miller Test).

Pragmatics of RSA

RSA is considered secure if large enough numbers are used. It is used in systems such as PGP. Encryption and decryption are costly compared to other schemes, though. Increasing efficiency can lead to a loss of security (e.g. small exponents, special kinds of keys, etc.).

RSA is often used for secure transmission of secret keys, with the secret key method then used for efficient encryption of a number of transaction messages. Other cryptographic methods include discrete logarithms and elliptic curves (not covered in this subject).

Secure Dealings

Consider the problem of playing cards over the phone. In this,

- some information must be kept secret,
- some information must be shared,
- no "umpire" to keep the players honest.

How can we ensure that the game is played fairly? We can use a public key cryptosystem if we have that $E_B(E_A(M)) = E_A(E_B(M))$, i.e. encryption is commutative. This means that the order of encryption doesn't matter, as it leads to the same result. The RSA system has this property. Here all keys are kept *private* (until after the game).

Key Idea: Pass around information which is unintelligible to others. To deal two cards to each player:

$1, 2, 3, 4, 5, 6$	original
$E_A(4, 1, 2, 5, 6, 3)$	A shuffles+encrypts
$E_A(4, 5), E_B(E_A(2, 3)))$	B picks 4, encrypts 2
$4, 5, E_B(2, 3)$	A decrypts all 4
$E_B(2, 3)$	A sends cards to B
$2, 3$	B decrypts

The RSA system is not quite adequate here, as the encrypted form of a card can give out some information. Other, more complex, systems can be used which do not let this happen.

Limitations

RSA encryption is costly compared to other forms of encryption, especially when the key sizes get large (as they are currently). It is also a potential victim of a successful search for an efficient factorisation algorithm. if such an algorithm is ever found, RSA becomes vulnerable immediately.

Extensions

One threat to RSA and similar methods is via *quantum computing*. Such a machine uses some quantum physics to generate many solutions to a given problem in parallel, and it has been shown that such a machine, if ever built, will be able to factorize efficiently. It is not clear that such a machine will ever be built, much less be commercially viable. however, it is a potential threat to RSA.

Zero-Knowledge Proofs

Recall that a problem is in $\mathcal{NP}$ if a solution to it can be checked, but not found, in polynomial time. We can use this as a security protocol as follows:

- the prover (customer) sends a solution to an $\mathcal{NP}$-complete problem to the doubter (bank teller);
- the doubter verifies that the solution indeed works.

An extension of this idea is *zero-knowledge proofs*.

Key Idea: Convince someone that you know something, without telling them what it is. This would allow access to systems without the possibility of your password being spotted by some malevolent user.

Here is one way of doing that.

- A shows B a graph, and claims it can be 3-coloured.
- A secretly colours the graph.
- B randomly selects two adjacent nodes.
- B checks that the colours differ.
- A re-colours the graph.
- B randomly checks two adjacent nodes.
- ...until B is satisfied.

A mathematical explanation to such an approach is as follows. Let the graph have n edges. Probability of passing 1st test falsely is $(n-1)/n$. Probability of passing 1st and 2nd falsely is $((n-1)/n)^2$. Probability of passing 1st, 2nd and 3rd falsely is $((n-1)/n)^3$. And so on so forth. The probability of error can hence be made arbitrarily low.

Cryptography Summary

- Cryptography depends on high complexity.
- Public key systems allow secure communication between arbitrary users.
- RSA is a public key system which uses large prime numbers.
- RSA systems secure as long as factorisation is intractable.
- RSA makes use of the existence of a probabilistic algorithm for finding large primes.
- Security protocols can make great use of encryption.

Section 9: Grammars (revisited)

CFG and Regular Languages

A *regular grammar* is a context-free grammar in which the rules are all of the form

- $A \to a$
- $A \to aB$
- $A \to e$

A *regular language* is a language which is expressible by a regular grammar. *Some context free languages are not regular*, e.g. $a^n b^n : n \geq 0$. However, *all regular languages are context-free* (see book p. 91).

Hierarchy of Grammars

Chomsky Hierarchy

1. Regular Grammars
2. Context Free Grammars
3. Context Sensitive Grammars
4. Unrestricted Grammars

Context sensitive grammars also allow rules of form: $BC \to bC$ - i.e. LHS contains more than one symbol. Whereas, unrestricted grammars can have any kind of rules.

Which Languages are Context-Free?

- *Context-free languages are closed under union, concatenation and Kleene star*
  - $CFG_1 \cup CFG_2 = CFG$
  - $CFG_1 CFG_2 = CFG$
  - $CFG_1^* = CFG$
- *The intersection of regular with context-free language is context-free*
  - $CFG_1 \cap RG_2 = CFG$

Chomsky Normal Form

Chomsky normal form is a useful way to "standardise"context-free grammar rules.

- RHS of a rule in Chomsky normal form must have length 1 or 2. (e.g. $A \to BC$ or $A \to a$ or $S \to e$)
- Any context free grammar can be converted to equivalent Chomsky normal form in *polynomial time*. (except that strings of length ≤ 1 cannot be generated).

Chomsky Normal Form Conversions

Three ways a rule may violate constraints of Chomsky normal form:

- *Long rules*. E.g. $A \to aBC$.
- *Short rules*. E.g. $A \to B$.
- *e-rules*. E.g. $A \to e$.

We will show how to convert each of these classes of rules to rules in Chomsky normal form.

Converting Long Rules

Let $A \to B_1B_2B_3B_4$ be a long rule we want to convert. Replace this rule with 3 new rules:

- $A \to B_1C_1$
- $C_1 \to B_2C_2$
- $C_2 \to B_3B_4$

where each C_n is a new non-terminal not used elsewhere in the grammar.

This principle can be used for all long rules. Each long rule with RHS length = n will be replaced by $n-1$ new rules in Chomsky normal form.

Converting e-Rules

- Determine the set ε of erasable non-terminals. ($A : A \stackrel{*}{\Rightarrow} e$).
  $\varepsilon := \emptyset$
  while there is a rule $A \to B$ with $B \in \varepsilon^*$ and $A \notin \varepsilon$
  do add A to ε.
- Delete all $e - rules$.
- For each $B \in \varepsilon$ and each rule of form $A \to BC$ or $A \to CB$
  add a rule of form $A \to C$.
- This adds some *short rules* that then need to also be transformed.

Converting Short Rules

- Get set of symbols $D(A)$ derivable from each symbol A (terminal and non-terminal).
  For each symbol A in language $D(A) := \{A\}$
  while there is a rule $B \to C$, with $B \in D(A)$
  and $C \notin D(A)$
  do add C to $D(A)$.

- Replace each rule $A \to BC$ with all possible rules $A \to B_1C_1$ where $B_1 \in D(B)$ and $C_1 \in D(C)$.
  E.g. $D(B) = \{B, E, F\}$ and $D(C) = \{C, G\}$ Then $A \to BC$ becomes:
  $A \to BC$, $A \to BG$, $A \to EC$, $A \to EG$, $A \to FC$, $A \to FG$
- Add $S \to BC$ for each A (except S itself) where $A \to BC$ and $A \in D(S)$.
  E.g. if $D(S) = \{S, B\}$ and we have $B \to CD$ then we add rule $S \to CD$

Greibach Normal Form

A grammar is in *Greibach Normal Form* if the rules are of the form:

- $A \to aA_1A_2 \ldots A_n$, for $n \geq 0$;
- The only rule with e as the body is $S \to e$.

All context-free grammars can be transformed into Greibach normal form (but the procedure is very complex). Derivations are not left recursive (i.e. $A \Rightarrow Aw$ doesn't happen), and are of length k for a derived string of length k.

PDAs and Grammars

PDAs accept the same class of languages as context-free grammars.

Grammar $\Rightarrow$ PDA. Given $G + (V, \Sigma, R, S)$, we construct a PDA with two states p and q. V is the stack alphabet. Hence, M is $(\{p, q\}, \Sigma, V, \Delta, p, \{q\})$ where Δ has the transitions:

- $((p, e, e), (q, S))$;
- $((q, e, A), (q, x))$ for each rule $A \to x$;
- $((q, a, a), (q, e))$ for each rule $a \in \Sigma$.

This simulates derivations of G, replacing nonterminals with the body of the rule, and finishing when only terminal symbols are left.

PDA $\Rightarrow$ grammar. This is tricky! "track" the computation states (q, A, p) with a nonterminal $< q, A, p >$ in the grammar.

1. $S \to < s, Z, f >$, s is the start state, f final state.
2. For each $((q, a, B), (r, e))$, $q, r \in K$, $a \in \Sigma \cup \{e\}, B, C \in \Gamma \cup \{e\}$, for each $p \in K$, $< q, B, p > \to a < r, C, p >$
3. For each $((q, a, B), (r, C_1C_2))$, $q, r \in K$, $a \in \Sigma \cup \{e\}, B \in \Gamma \cup \{e\}$ and $C_1, C_2 \in \Gamma$, for each $p, p' \in K$, $< q, B, p > \to a < r, C_2, p' >< p', C_2, p >$.
4. For each $q \in K$, $< q, e, q > \to e$.

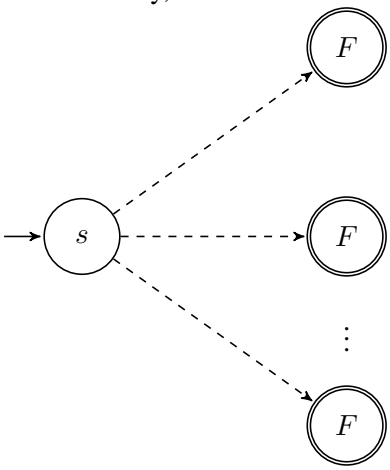
The full procedure is a bit complex; details can be found on pp.139-142.

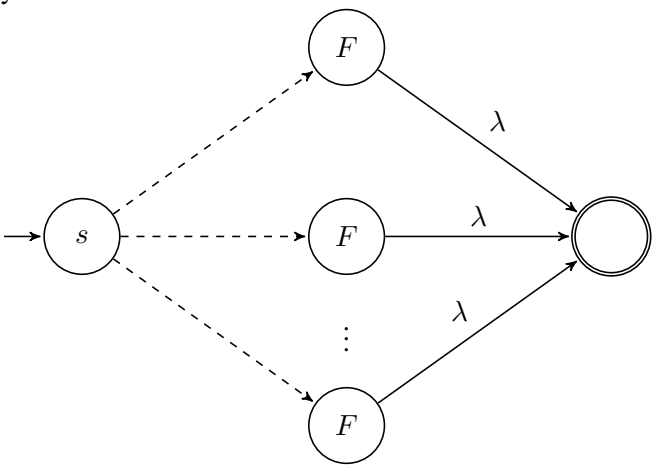
Extensions

There are more powerful grammar formalisms (*context-sensitive* grammars). There are also useful tools such as DTDs (Document Type Definitions) which make use of context-free grammars (as well as regular expressions). Grammars for natural language is a more difficult problem, but useful tools include *definite clause grammars* and *phrase-structure grammars*. These involve more intricate rules than context-free grammars.

Section 10: Automata (revisited)

In Automate theory, we can assume that any NFA has only one final state.





We can also assume that there are no transitions into the initial state, nor out of the final state.

NFA vs DFA

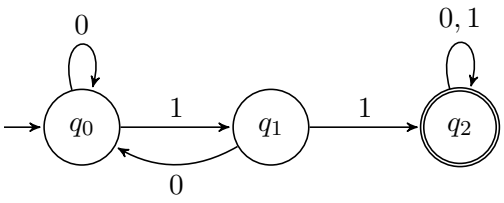
DFAs:

- Simpler computational model;
- Simple to implement;
- Easier to reason about;
- More likely to be used to show "negative" results.

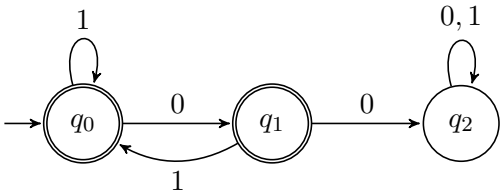
NFAs:

- Easier to specify machines;
- Simple to combine machines;
- Harder to implement;
- More likely to be used to show "positive" results.

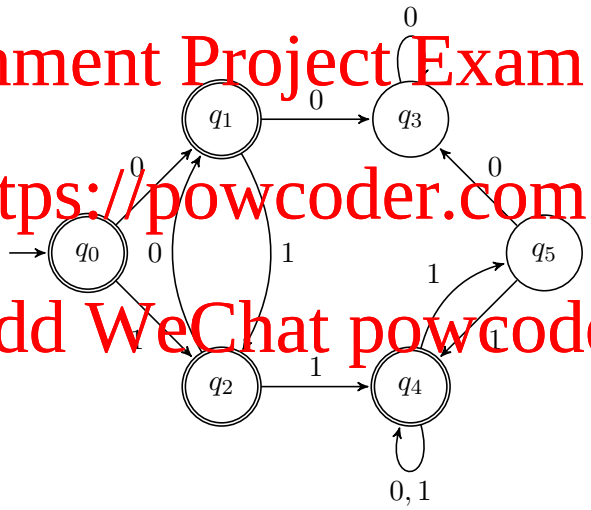
For example, M_1: accepts strings containing 11



M_2: accepts strings **not** containing 00

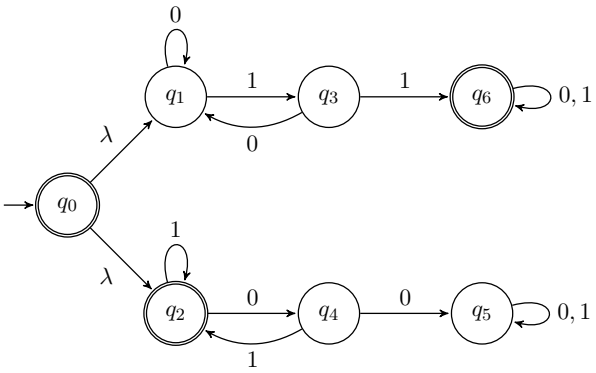


M_3: DFA for $L(M_1) \cup L(M_2)$



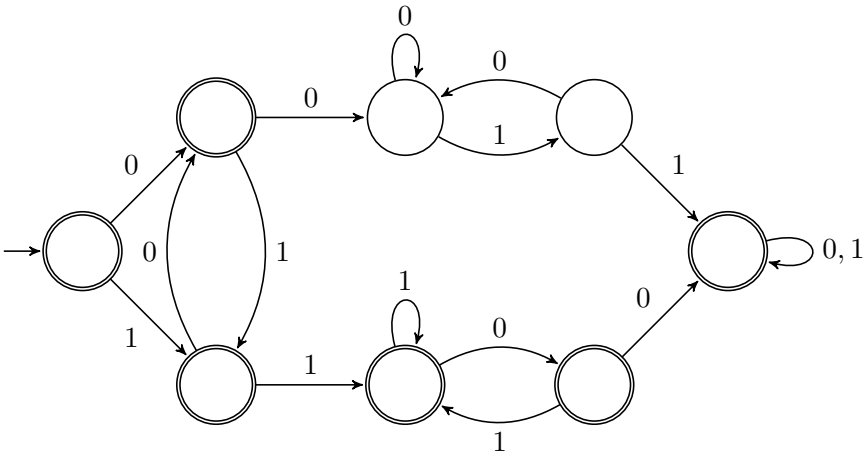
Note that this has no direct relationship to M_1 and M_2.

M_4: NFA for $L(M_1) \cup L(M_2)$



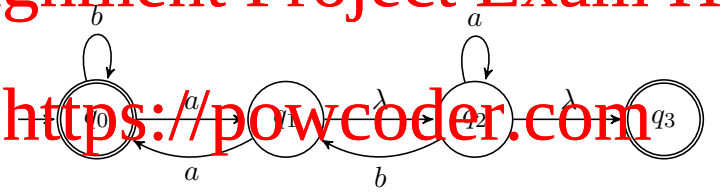
Clearly M_4 is a simple combination of M_1 and M_2.

Note that a machine equivalent to M_3 can be obtained by converting M_4 to a DFA.



Converting NFAs to DFAs

Key Idea: trace *all* computation paths at once (rather than choose one)



Input aab

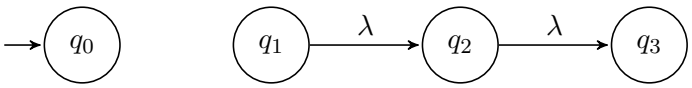
$\{q_0\} \xrightarrow{a} \{q_1, q_2, q_3\} \xrightarrow{a} \{q_0, q_2, q_3\} \xrightarrow{b} \{q_0, q_1, q_2, q_3\}$

Idea: replace NFA sets of states with DFA states. Each state in the DFA will represent the set of possible states in the NFA.

Lambda closure

If we can be in state q_i and it has a λ arc to q_j we can be in state q_j. The λ *closure* of a state q_i, denoted $\lambda - closure(q_i)$, is defined as:

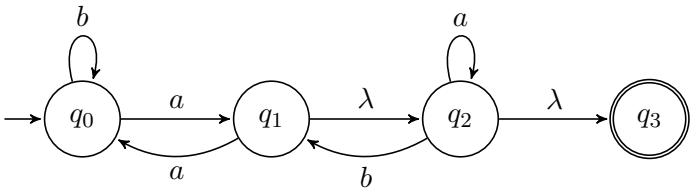
1. $q_i \in \lambda - closure(q_i)$;
2. Let $q_j \in \lambda - closure(q_i)$. If $q_k \in \delta(q_j, \lambda)$ then $q_k \in \lambda - closure(q_i)$.



q_0	$\{q_0\}$
q_1	$\{q_1, q_2, q_3\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$

The *input transition function* $t : Q \times \Sigma \to \wp(Q)$ of an NFA-λ is defined:

$$t(q_i, a) = \bigcup_{q_j \in \lambda-closure(q_i)} \lambda - closure(\delta(q_j, a))$$

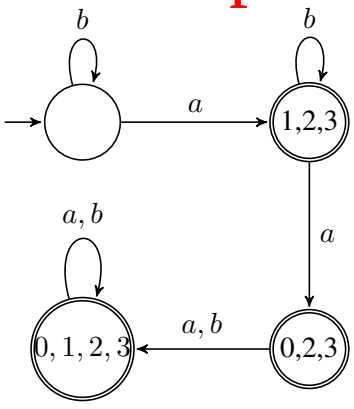


t	a	b
q_0	$\{q_1, q_2, q_3\}$	$\{q_0\}$
q_1	$\{q_0, q_2, q_3\}$	$\{q_1, q_2, q_3\}$
q_2	$\{q_2, q_3\}$	$\{q_1, q_2, q_3\}$
q_3	$\emptyset$	$\emptyset$

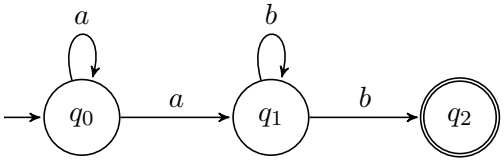
Algorithm for Constructing a DFA from an NFA-λ

Initialise Q' to $\{\lambda - closure(q_0)\}$
repeat
  if $X \in Q'$ is a node with no arc labelled a leaving X
    $Y = \bigcup_{q_i \in X} t(q_i, a)$
    if $Y \notin Q'$ **then** $Q' := Q' \cup \{Y\}$
    add an arc from X to Y labelled a
  else *done* := true
until *done*
start state of DM is $\lambda - closure(q_0)$
final states of DM are $F' := \{X \in Q' \mid X \cap F \neq \emptyset\}$

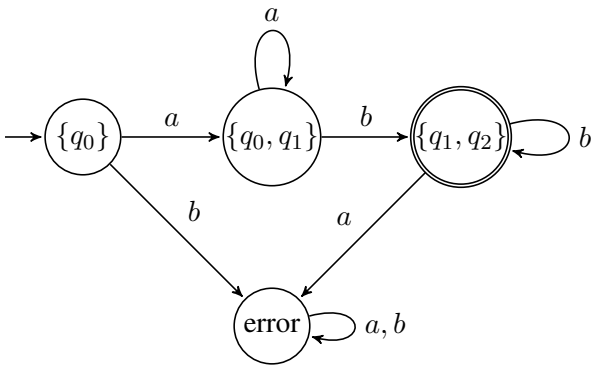
For the above example, applying this algorithm gets the DFA below.



For another example, consider the NFA below for the regular expression $\mathbf{a}^+\mathbf{b}^+$.



An equivalent DFA is below.

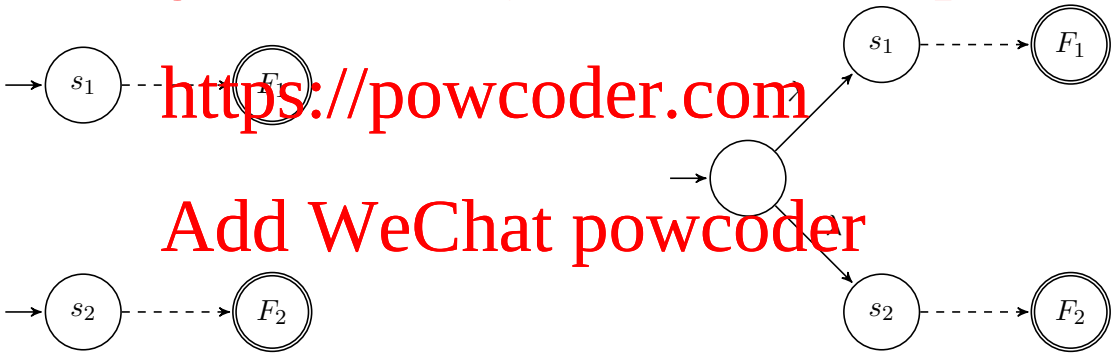


FSMs and Regular Expressions

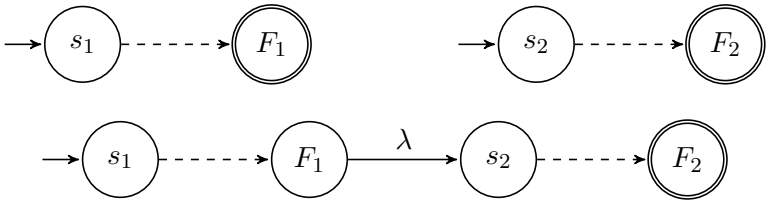
Given a regular expression, can we find an NFA for it?

- $\oslash$ and a:

- $X \cup Y$:



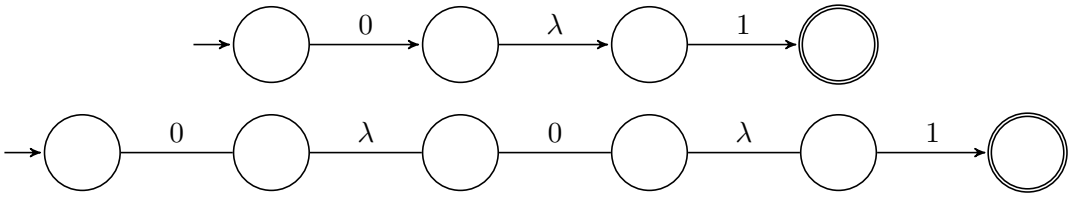
- XY:



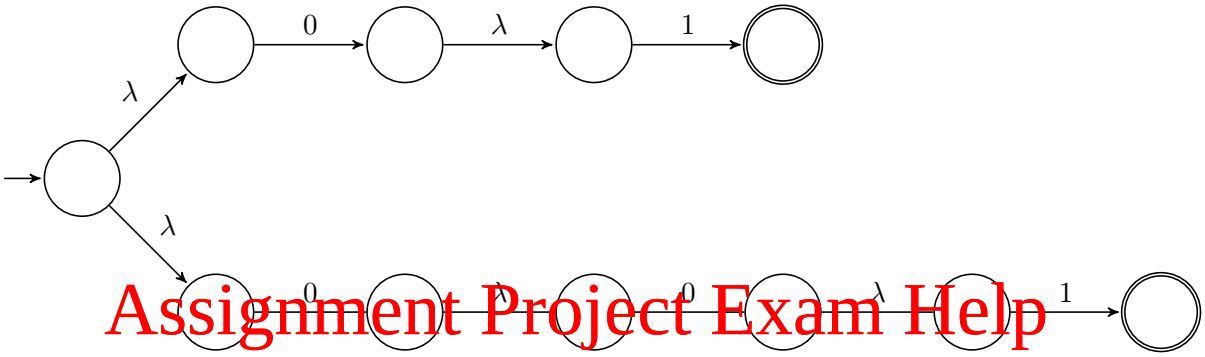
- X^*: (next page)

Hence for any regular expression, we can find an equivalent NFA. For example, consider the expression $(01 \cup 001)^*$. We start as follows:

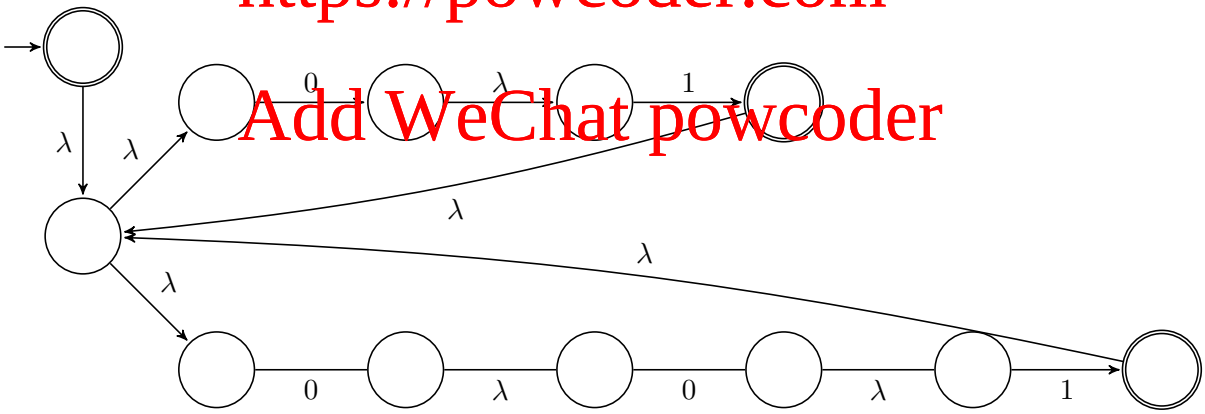
We then get:



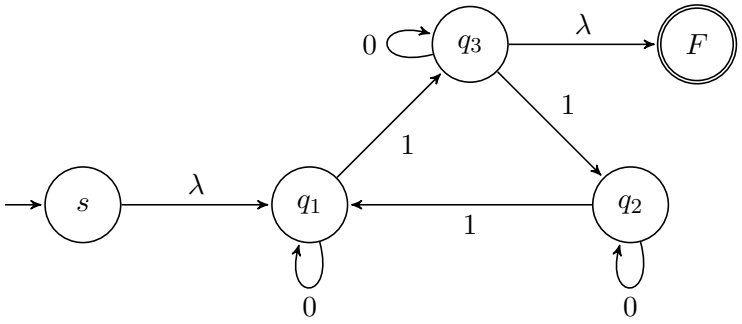
We then combine the two machines:



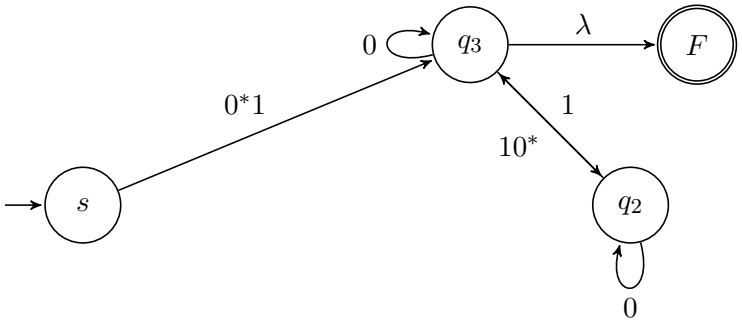
And finally add the transitions for the $*$:



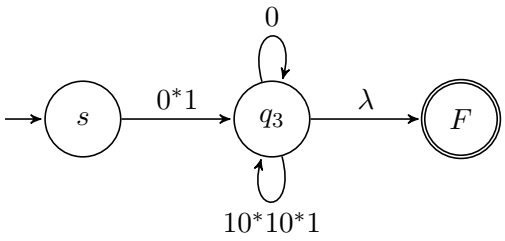
Whilst this is a rather convoluted machine, it shows how we can systematically construct a machine for $(01 \cup 011)^*$. For any NFA, can we find a regular expression for it? Consider the NFA below.



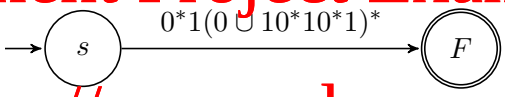
First we eliminate q_1.



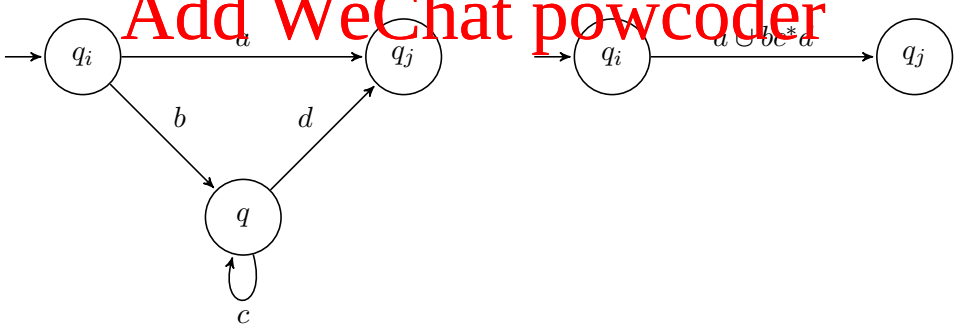
Note that the paths eliminated are $s \to q_1 \to q_3$ and $q_2 \to q_1 \to q_3$. Then we eliminate q_2.



Here we have eliminated the path $q_3 \to q_2 \to q_3$. Finally we eliminate q_3.



Hence the regular expression for $L(M)$ is $0^*1(0 \cup 10^*10^*1)^*$. Note that these machines are not NFAs; these are merely convenient steps used in order to find the regular expression. These are all just instances of the following general transformation.



This reduced the number of states from 3 to 2. We apply this transformation repeatedly until we are left with just the initial state and the final state. Note that the size of the regular expressions generated can be quite large. Hence, NFAs (and DFAs) and regular expressions are equivalent in power.

Theorem 1 *A language L is accepted by a DFA iff L is accepted by an NFA iff there is a regular expression for L.*

A language L as above is known as a *regular* language. We can show that a language is regular by giving one of the following for it:

- a regular expression;
- an NFA;
- a DFA.

Limitations of FSAs

Not all languages are regular — consider $\{a^i b^i | i \geq 0\}$. How do we know this, i.e. cannot build a FSM for it? How can we show there can be no NFA for this language?

- FSAs only have "bounded" memory.
- Infinite strings $\rightarrow$ cycles $\rightarrow$ predictable structure.

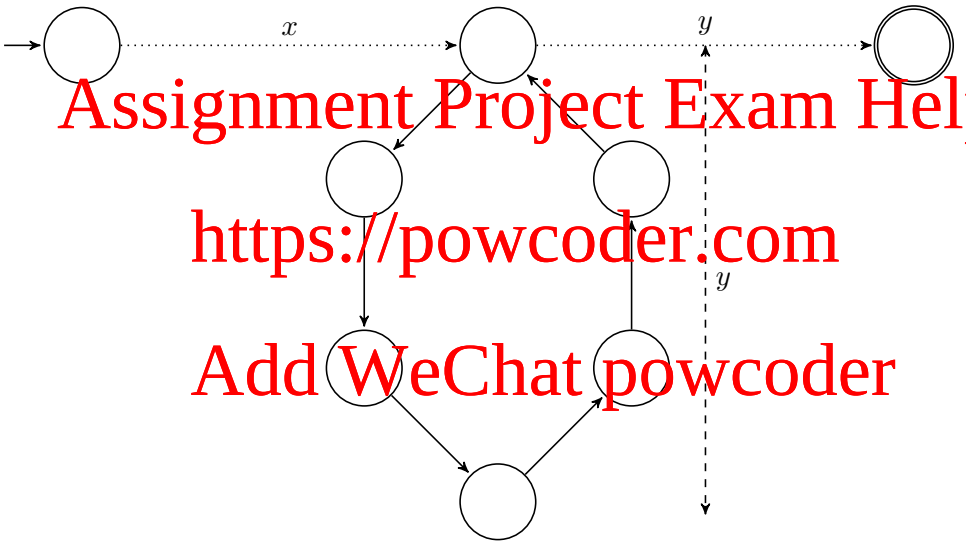
The *Pumping Lemma* is a technical way of using these observations. It is so named because of the way it indicates strings can be "pumped" into a given string and still have it accepted by the machine.

Pumping Lemma

Theorem 2 (The Pigeon-hole Principle) *Let M be a DFA with k states. Every computation of length k contains a cycle.*

$$q_0 \rightarrow q_1 \rightarrow \ldots q_k$$

This means a DFA accepting a string of length K looks something like this:



Theorem 3 (The Pumping Lemma) *Let L be a regular language. Then there exists $n \geq 1$ such that any string $w \in L$ with $|w| \geq n$ can be written as $w = xyz$ such that*

1. $y \neq e$
2. $|xy| \leq n$
3. $xy^i z \in L$ *for all* $i \geq 0$

Intuitively, this means that any DFA which accepts a sufficiently long string will contain a cycle. Note that the Pumping Lemma does not tell us where the cycle is; just that there must be one somewhere. We often use this to show that a language is **not** regular; but not the other way round.

Be careful about exactly what is stated here;

for each regular language L,
there exists an $n \geq 1$ such that,
for each string $w \in L$ longer than n **there exist** strings x, y and z such that $w = xyz$, $y \neq e$, $|xy| \leq n$ and
for all $i \geq 0$ $xy^i z \in L$.

Think of this as a game:

- you make *any* choice for the **for each** parts;
- your opponent has a procedure for coming up with the **there exists** parts for any choice you make.

In order to use the Pumping Lemma show a language is not regular, we need to apply *proof by contradiction*. The general procedure is to assume L is regular, and use the Pumping Lemma to get a contradiction. More precisely:

1. Assume language L is regular.
2. Apply Pumping Lemma.
3. Choose string w.
4. Use $|xy| \leq n$ to get information about y.
5. Choose i such that $xy^iz \notin L$.
6. Contradiction; hence L not regular.

All such proofs are the same except for steps 3 and 5.

Proposition 4 *$L = \{a^ib^i \mid i \geq 0\}$ is not regular.*

Proof: Assume that L is regular. Then the Pumping Lemma applies, i.e. that $\exists n$ such that for all $w \in L$ with $|w| \geq n$, $w = xyz$, $y \neq e$, $|xy| \leq n$ and $xy^iz \in L$ for all $i \geq 0$.

Let $w = a^nb^n$. The $w \in L$ and $|w| > n$, and so $w = xyz$. As $|xy| \leq n$, y is of the form a^j for some $1 \leq j \leq n$.

Hence $xyyz \in L$, i.e. $a^{n+j}b^n \in L$, which is a contradiction.

Hence L is not regular.

Proposition 5 *$L = \{a^i \mid i$ is prime $\}$ is not regular.*

Proof: Assume that L is regular. Then the Pumping Lemma applies as above.

Let $w = a^m$ where m is any prime larger than n. Then we have $w = a^m = xyz$, $y \neq e$ and $xy^iz \in L$ for all $i \geq 0$.

Let $i = m + 1$. Then

$$\begin{aligned} |xy^iz| &= |xy^{m+1}z| \\ &= |xyz| + |y^m| \\ &= m + m|y| \\ &= m(1 + |y|). \end{aligned}$$

Hence $|xy^{m+1}z|$ is not prime, and so $|xy^{m+1}z| \notin L$ which is a contradiction.

Hence L is not regular.

The Pumping Lemma can also be used "positively". Let M be a DFA with n states.

- $L(M)$ is not empty iff M accepts a string of length $< n$;
- $L(M)$ is infinite iff M accepts a string w with $n \leq |w| < 2n$.

Note that if M accepts w with $|w| \geq n$, by the Pumping Lemma it also accepts xz, where $w = xyz$. Both of the above observations follow from this.

Algorithms for FSMs

We have seen algorithms for:

1. Converting an NFA into a DFA.
2. Converting a regular expression into an NFA.
3. Converting an NFA into a regular expression.

Algorithm 1 is exponential (there can be $2^{|K|}$ states in the DFA compared to $|K|$ in the NFA). Algorithm 2 is polynomial. Algorithm 3 is exponential (!!); the resulting expression can be exponential in size. There is also a polynomial time algorithm for converting a DFA into an equivalent *minimal* DFA, i.e. a DFA with a minimum number of states (see pp.48-53 of the book). Thus we have also algorithms for:

4. Converting a DFA into a minimal DFA (polynomial).
5. Determining whether 2 DFAs are equivalent (polynomial).
6. Determining whether 2 NFAs are equivalent (exponential).

Algorithm 5 uses Algorithm 4 by minimizing each DFA. Algorithm 6 uses Algorithm 1 to convert each NFA to a DFA, and then uses Algorithm 5.

Is there a polynomial algorithm for:

- equivalence of NFAs?
- minimizing an NFA?
- finding a minimal, equivalent DFA for an NFA?

Well, does $\mathcal{P} = \mathcal{NP}$? Basically, all of the above questions are equivalent to this one ...

FSMs as Algorithms

A DFA M is an efficient algorithm for testing membership of $L(M)$. There is a linear algorithm based on the DFA for this (such ideas are used in string matching algorithms). An NFA can also be "executed", by constructing the possible set of states "on the fly" rather than in advance as in the above conversion algorithm. Hence, (simple) computations can be modelled by FSMs.

Extensions of FSMs

As general computational models, FSMs are not powerful enough. As we have seen, some simple languages cannot be represented. We need to add some kind of *memory*. The key questions are then how we access this memory and indicate results from it. Later we will see more powerful machines (pushdown automata, Turing machines). The basic idea of FSMs as a computational description has been extended in many ways. There are a number of variations of FSMs, such as *Buchi automata*, which use more sophisticated execution models, but the same basic idea.

Section 11: Chomsky Hierarchy

Turing machines are one type of *automata*. As we have seen previously, automata usually have a corressponding grammar. For example,

- NFA $\equiv$ DFA $\equiv$ regular grammar
- PDA $\equiv$ context-free grammar
- TM $\equiv$???

Consider grammar rules of the form $A \to v$. A grammar is regular if:

- A is a nonterminal (ie $A \in V - \Sigma$);
- $v \in \Sigma$ or $v \in \Sigma(V - \Sigma)$.

It is context-free if:

- A is a nonterminal (ie $A \in V - \Sigma$);
- $v \in V^*$.

Each of the previous classes places restrictions on rules. An *unrestricted grammar* is one in which rules are of the form $u \to v$ where $u, v \in V^*$, except that u must contain a nonterminal somewhere. Note the significance of the name *context-free*; such rules can replace a nonterminal in any context whatsoever. Unrestricted grammars can make use of context.

An *unrestricted grammar* G is a quadruple (V, Σ, R, S) where,

- V is an alphabet;
- $\Sigma \subseteq V$ is the set of terminal symbols;
- $S \in V - \Sigma$ is the start symbol;
- **(NEW)** R is a set of rules and it is a subset of $(V^*(V - \Sigma)V^*) \times V^*$

Note that the rules in context-sensitive or unrestricted grammars require that some nonterminal be present on the left hand side. $uAv \to uwv$ means "replace A with w in the context of u and v". We say $u \Rightarrow_G v$ if $\exists w_1, w_2 \in V^*$ and $u' \to v' \in R$ such that $u = w_1 u' w_2$ and $v = w_1 v' w_2$.

$G = (V, \Sigma, R, S)$, where
$V = \{S, a, b, c, A, B, C, T_a, T_b, T_c\}$, $\Sigma = \{a, b, c\}$, and the rules are:
$S \to ABCS \mid T_c$

$CA \to AC$
$BA \to AB$
$CB \to BC$
$CT_c \to T_c c \mid T_b c$
$BT_b \to T_b b \mid T_a b$
$AT_a \to T_a a$
$T_a \to e$

A derivation using the grammar above is as follows:

$S \Rightarrow^* (ABC)^n T_c$
$\Rightarrow^* A^n B^n C^n T_c$
$\Rightarrow^* A^n B^n T_b c^n$
$\Rightarrow^* A^n T_a b^n c^n$
$\Rightarrow a^n b^n c^n$

Unrestricted grammars (sometimes called *rewriting systems*) are equivalent to Turing machines. In order to convert grammar $\Rightarrow$ Turing machine: use a 2-tape machine.

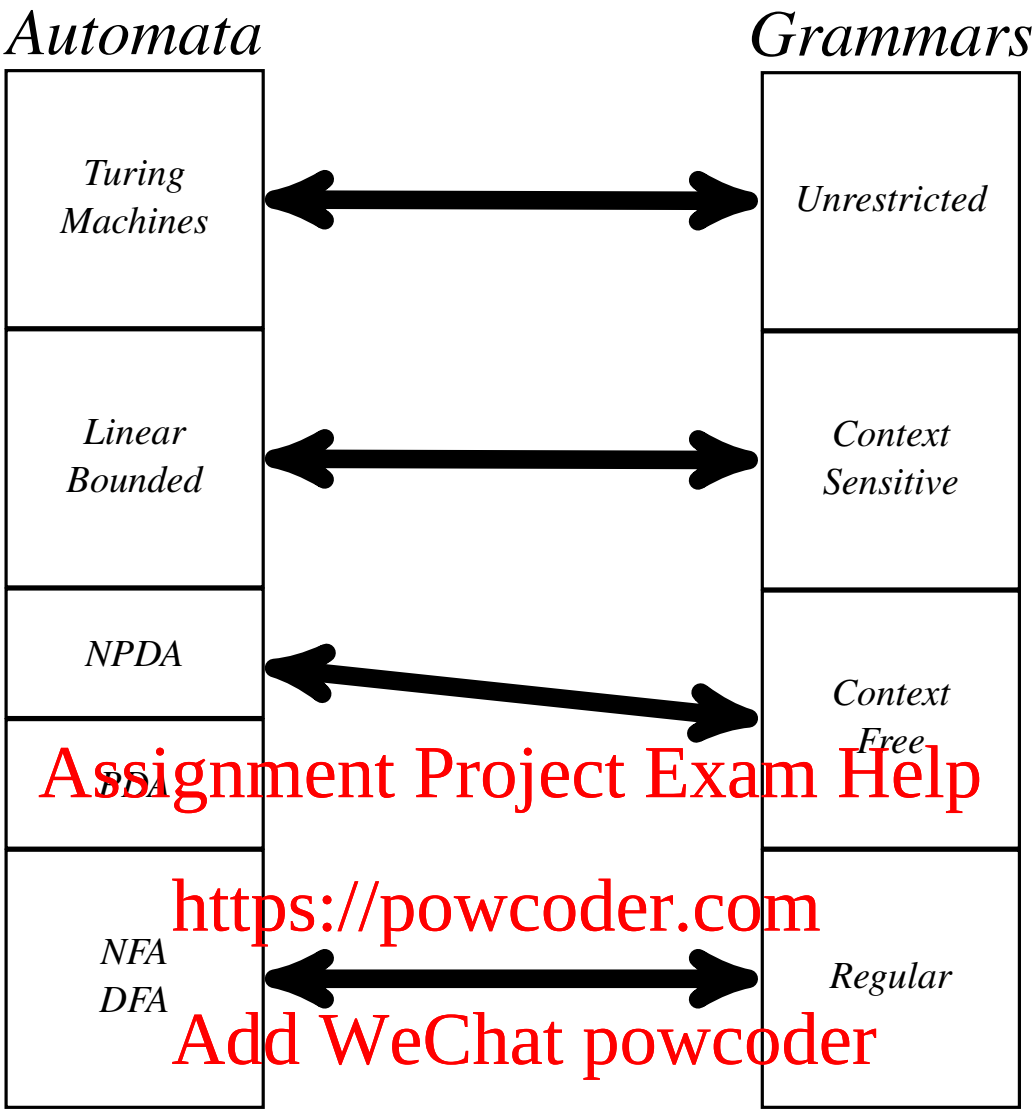
- Tape 1 contains the input w and is never changed;
- Initially write S on tape 2;
- Nondeterministically update tape 2 with symbols according to a rule in R;
- Check whether generated string is w.

In order to convert Turing machine $M \Rightarrow$ grammar G: let V be $\Sigma \cup K \cup \{S, \triangleright\}$

G simulates *backwards* computations of M, with S corresponding to a halt state. Configuration $(q, \triangleright\ u\underline{a}w$ becomes the string $\triangleright\ uaqw \triangleleft$. ($\triangleleft$ for end of string). For each $q \in K, a \in \Sigma$, G has a rule

1. If $\delta(q, a) = (p, b), b \in \Sigma$, $bp \rightarrow aq$
2. If $\delta(q, a) = (p, \rightarrow)$, $abp \rightarrow aqb \forall b \in \Sigma$, $a \sqcup p \triangleleft \rightarrow aq \triangleleft$
3. If $\delta(q, a) = (p, \leftarrow), a \neq \sqcup$, $pa \rightarrow aq$
4. If $\delta(q, \sqcup) = (p, \leftarrow)$, $pab \rightarrow aqb \forall b \in \Sigma$, $p \triangleright \rightarrow \sqcup q \triangleleft$

We also add the rules $S \rightarrow \triangleright \sqcup h \triangleleft$, $\triangleright \sqcup S \rightarrow e$ and $\triangleleft \rightarrow e$. $S \Rightarrow \triangleright \sqcup h \triangleleft \Rightarrow^* \triangleright \sqcup sw \triangleleft \Rightarrow^* w \triangleleft \Rightarrow w$. *Context-sensitive grammars* (CSG) are similar to unrestricted grammars, except that a rule $u \rightarrow v$ must have $|v| \geq |u|$. Hence, context-sensitive grammars *do not allow derived strings to get shorter. Such grammars correspond to* linear-bounded automata *(LBA). An LBA is a Turing machine which only uses* $|w| + 2$ *tape squares.*

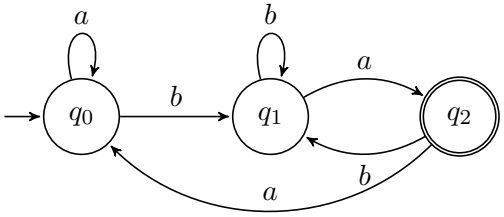


Mapping DFAs to Grammars

Any regular language can be represented as a DFA. Any DFA can be transformed to a context-free grammar. Therefore all regular languages are context-free languages. To form a grammar from a DFA:

- *Let DFA states be grammar non-terminals.*
- *For each transition $q \to p$ on input a, we get a rule $q \to ap$.*

For e.g. $S \to aS$



Grammar rules for above diagram:

$$
\begin{array}{ll}
S \to aS & S \to bA \\
A \to aB & A \to bA \\
B \to aS & B \to bA \\
B \to e &
\end{array}
$$

Pumping Lemma for Context-Free Languages

Similar in principle to the Pumping Lemma for Regular Languages. If G is a context-free grammar, then any string, w, in G that is longer than the fanout of G, can be written as w = uvxyz, such that either u or v is non-empty and uv^nxy^nz is in L(G) for every n ≥ 0. *(fanout of G is largest number of symbols on RHS of a rule in G). This lemma can be used to* prove by contradiction *that a language is not context-free. (see the book pp. 97-8).*

Algorithmic Properties

There are algorithms which convert one form of automata to corressponding languages and grammars, or checking for equivalence. Here are their properties:

- polynomial algorithm *for Regular Expression to NFA.*
- exponential algorithm *for NFA to Regular Expression.*
- polynomial algorithm *for minimizing DFA.*
- no algorithm *for minimizing PDA's.*
- polynomial algorithm *for deciding equivalence of DFA's.*
- exponential algorithm *for deciding equivalence of NFA's.*
- no algorithm *for deciding equivalence of PDA's.*
- polynomial algorithm *for CFG to PDA.*
- polynomial algorithm *for PDA to CFG.*
- polynomial algorithm, *which given a context-free grammar G and a string w, decides whether $w \in L(G)$.*

Deterministic PDAs

Context-free grammars are used extensively for describing syntax of real programming languages. Compilers thus need to include a parser (to build parse tree to produce assembly language). Although CFG in Chomsky normal form allows straightforward polynomial parsing, it's still too slow for large programs (cubic in the size of input string). Many successful parsing optimisations (for particular languages) developed by compiler builders are based on PDAs.

PDAs are nondeterministic. For parsing purposes, nondeterminism is a problem, as deterministic machines are much more practical. However, nondeterministic PDAs cannot *be transformed into deterministic ones. Simply put, nondeterministic ones can do more sophisticated counting, as the precise state of the stack may not be predictable in advance. Deterministic ones cannot make such subtle choices. The details are on pp.105-9 of the book.*

Deterministic Context Free Languages

Arbitrary PDAs are not useful for algorithms as they are non-deterministic. Unfortunately, we can't always get a deterministic PDA (recall non-deterministic PDAs are more powerful than deterministic ones). There are heuristics which can guide development of deterministic PDAs from suitable context-free languages. These are called Deterministic Context Free Languages.

Deterministic context-free languages are basically those accepted by a deterministic PDA with the additional ability to sense the end of the string. (pp. 159 Lewis).

Section 12: Extension topics

Turing Machines

Turing machines are simple and intuitive model of computation and the most commonly used model. They are robust (no know variations increase power) and powerful enough to cover all computations. Yet, there are some things that cannot be computed by even using a Turing machine. For instance, termincation cannot be guaranteed under certain circumstances.

However, an important observation is the Church-Turing thesis; anything computable can be computed on a Turing machine.

Computability

What kinds of problems cannot be computed? Main problem is termination: *program X solves problem Y, but may not terminate. For example,*

- *Halting problem;*
- *Decision problems for machines;*
- *Decision problems for grammars;*
- *Tiling problem.*

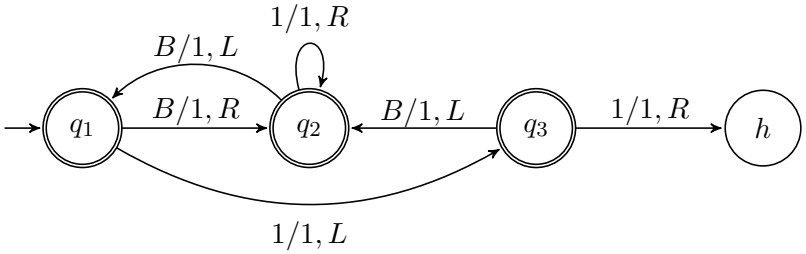
Busy Beaver Problem

Basic idea: *Turing machine of given size which prints out maximum number of 1's and then halts. This is a conceptually simple example of* Busy Beaver *problem. There are many intricacies though. This problem was first studied in 1960's. Even though the problem defines a simple function, this function is not computable. One of the key requirements is that it should halt. There are many cases which are still unsolved.*

A busy beaver *machine is a Turing machine with the following properties:*

- *deterministic;*
- *two-way tape, entirely blank on input;*
- *output is a sequence of 1's;*
- $\Sigma(n)$*: maximum number of 1's output for machine of size* n*;*
- *size* $n \Rightarrow n+1$ *states ("halt" state);*
- $\mathcal{S}(n)$*: maximum number of state transitions for machine of size* n*;*
- **always** *terminates;*
- *tape head need not be "wound back".*

Problem: *What is the value of* $\Sigma(n)$*?* $\mathcal{S}(n)$*? Following is a busy beaver machine for* $n = 3$*:*



$(q_1, \sqcup^*) \Rightarrow (q_2, 1^*\sqcup) \Rightarrow (q_1, \underline{1}^*1) \Rightarrow (q_3, \sqcup 1^*1) \Rightarrow (q_2, \sqcup 11^*1) \Rightarrow (q_1, \sqcup 111^*1) \Rightarrow (q_2, 1\underline{1}111^*1) \Rightarrow (q_2, 11\underline{1}1^*1) \Rightarrow (q_2, 111\underline{1}*1) \Rightarrow (q_2, 1111^*\underline{1}) \Rightarrow (q_2, 1111^*1\sqcup) \Rightarrow (q_1, 1111^*\underline{1}*1) \Rightarrow (q_3, 111\underline{1}^*11) \Rightarrow (q_1, 1111^*\underline{1}1)$

This shows that $\Sigma(3) \geq 6$, $\mathcal{S}(3) \geq 13$. However, how do we find the maximum? Enumerate all such possible Turing machines! It may be finite, but such number could be exponential. Generally we cannot tell whether the machine halts.

n	$\Sigma(n)$	$\mathcal{S}(n)$
1	*1*	*1*
2	*4*	*6*
3	*6*	*21*
4	*13*	*107*
5	*$\geq$ 4,098*	*$\geq$ 47,176,870*
6	*$\geq$ 95,524,079*	*$\geq$ 8,690,333,381,690,951*
7	*????????????*	*????????????????????*

What is the complexity of $\Sigma(n)$? $\mathcal{S}(n)$? In fact, neither $\Sigma(n)$ nor $\mathcal{S}(n)$ is computable! Both grow faster than any *computable function (including $2^{2^{2^n}}$, $2^{2^{n!}}$, ...). To see this, let f be any computable function. As f is computable, so is*

$$F(x) = \Sigma_{0 \leq i \leq x}(f(i) + i^2)$$

Hence there is a TM M_F for F, i.e. M_F starts with x 1's on the tape, and finishes with $F(x)$ 1's on the tape. Let M_F have k states.

Let X be the Turing machine which prints x 1's on an initially blank tape. Note that X of size x (x states plus a halting state). We construct M as XM_FM_F. M behaves as follows:

- *M first writes x 1's and then a blank;*
- *M mimics M_F, writing $F(x)$ 1's on the tape, and a then blank;*
- *M mimics M_F again, writing $F(F(x))$ 1's on the tape.*

Note that M has $x + 2k$ states — x from X, and $2k$ for the two copies of M_F. As the busy beaver machine for $x + 2k$ outputs the maximum number of 1's we have that

$$\begin{aligned}\Sigma(x + 2k) &\geq \text{number of 1's output by } M \\ &= x + F(x) + F(F(x))\end{aligned}$$

Now we know that $F(x) \geq x^2$ and $\exists n$ such that $x^2 > x + 2k$ for $x \geq n$. Therefore we have $F(x) > x + 2k$ for $x \geq n$. Also, as $F(x) > F(y)$ if $x > y$, we have

$$F(F(x)) > F(x + 2k) > f(x + 2k)$$

Hence we have,

$$\begin{aligned}\Sigma(x + 2k) &\geq x + F(x) + F(F(x)) \\ &> F(F(x) \\ &> F(F(x)) \\ &\geq F(x + 2k) \\ &> f(x + 2k)\end{aligned}$$

Hence $\exists n$ such that $\forall x \geq n \Sigma(x) \geq f(x)$. Hence, as f was arbitrary, Σ is not computable. Note that this is quite a strong result; in fact, we have that for any *computable function f,*

$$f(\Sigma(n)) < \Sigma(n + 1)$$

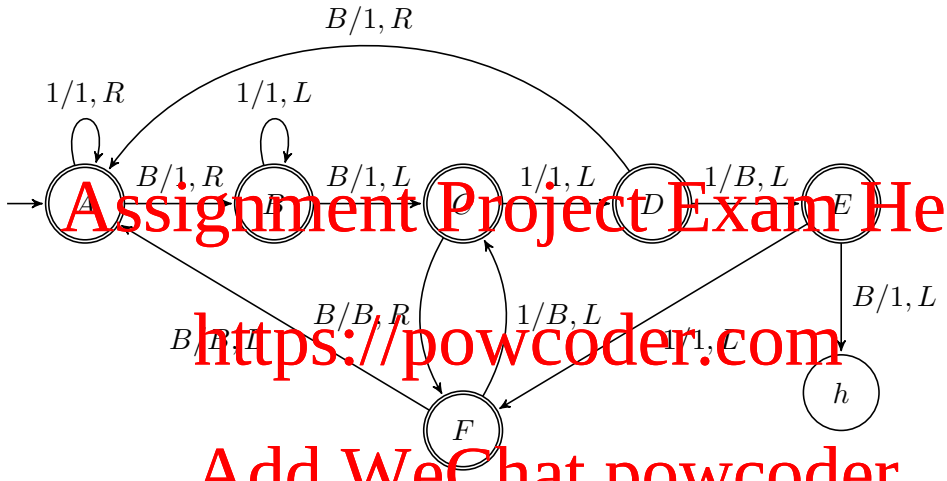
for an infinite number of values of n. This is quite weird:
$\Sigma(n + 1) > \Sigma(n)^2$
$\Sigma(n + 1) > 2^{\Sigma(n)}$

$\Sigma(n+1) > \Sigma(n)!$
$\Sigma(n+1) > \Sigma(n)^{\Sigma(n)}$
$\Sigma(n+1) > \Sigma(n)^{\Sigma(n)^{\cdots^{\Sigma(n)}}}$

Hence, even if $f(n) = n^{n^{n^{\cdots^{n}}}}$, *we still have that* $\Sigma(n+1) > f(\Sigma(n)) = \Sigma(n)^{\Sigma(n)^{\cdots^{\Sigma(n)}}}$. $\mathcal{S}(n)$ *is even worse:* $\mathcal{S}(n) \geq \Sigma(n)$ *(!!)*

To see this, consider the machine X. *There is a version of this machine which prints* $\Sigma(n)$ *1's and has size* $\Sigma(n)$, *and hence there are* $\Sigma(n)$ *transitions in the execution of this machine. Hence, as* $\mathcal{S}(n)$ *is the maximum number of state transitions that can be made for a Turing machine of size* n, $\mathcal{S}(n) \geq \Sigma(n)$. $\mathcal{S}(n)$ *is generally much larger than* $\Sigma(n)$ *(!!)*

n	$\Sigma(n)$	$\mathcal{S}(n)$
3	*6*	*21*
4	*13*	*107*
5	*$\geq$ 4,098*	*$\geq$ 47,176,870*
6	*$\geq$ 95,524,079*	*$\geq$ 8,690,333,381,690,951*



This prints out 95,524,079 1's on the tape. (!!) It takes 8,690,333,381,690,951 transitions. (!!!) Don't try to execute this by hand! There is a JFLAP version, but JFLAP does not go further than 1600 steps ...

3n+1 sequence

Termination is not just tricky . . . but sometimes even undecidable, rather cannot be proved. Consider the following operation on a positive integer n:

- *if* n *is even, divide* n *by 2;*
- *if* n *is odd, replace* n *by* $3n + 1$.

This is known as $3n + 1$ *problem, Collatz problem, Ulam sequences, Hailstone numbers, ... If we repeat this operation on a given number* x, *do we eventually halt with* 1 *or not?*

$9 \to 28 \to 14 \to 7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$

Number	*Maximum*
21	*64*
23	*160*
25	*88*
27	*9,232*
29	*88*
31	*9,232*
33	*100*
35	*160*
37	*112*
39	*304*
41	*9,232*
43	*196*
45	*136*
47	*9,232*
49	*148*

All known cases terminate. How could we prove this? We need to show that each number eventually "hits" a power of 2. We cannot (yet) put accurate bounds on the sequence. There are lots of interesting statistics, though.

Further Reading

There are many sources of information on the busy beaver problem.

- *T. Rado, On non-computable functions,* Bell System Technical Journal **41** *(1962), 877-884.*
- *M.W. Green, A lower bound on Rado's Sigma function for binary Turing machines,* Proceedings of the 5th Symposium on Switching Circuit Theory and Logical Design, *(1964), 91-94.*
- *S. Lin and T. Rado, Computer studies of Turing machine problems,* Journal of the ACM **12** *(1965), 196-212.*
- *A.H. Brady, The determination of the value of Rado's noncomputable function* $\Sigma(k)$ *for four-state Turing machines,* Math. Comp. **40** *(1983), 647-665.*
- *H. Marxen and J. Buntrock, Attacking the busy beaver 5,* Bulletin of the EATCS **40** *(1990), 247-251.*

Tiles Revisited

Tiling Problem 1:

Given a set of tile designs T, is it possible to cover any *rectangular area using only designs from T? Rotations are* **not** *allowed, and the patterns on each of the four sides of a tile* **must** *match the neighbouring tiles.*

Informally, can we tile any *bathroom this way? (including some enormous examples as the bathrooms at Versailles, or the Taj Mahal, or ...)*

Answer: *This problem is* undecidable *(see Lewis & Papadimitrou pp.263-7).*

Tiling Problem 2:

Given a set of tile designs T and a bathroom wall of size $s \times t$, is it possible to cover the wall using only designs from T? Rotations are **not** *allowed, and the patterns on each of the four sides of a tile* **must** *match the neighbouring tiles.*

Informally, can we tile my *bathroom this way? This is clearly a more specific question that the earlier version.*

Answer: *This problem is* NP-complete *(see Lewis & P pp.310-3).*

Tiling Problem 3:

Given a set of tile designs T and a bathroom wall of size $s \times t$, is it possible to cover the wall using only designs from T? Rotations **are** *allowed, but the patterns on each of the four sides of a tile must still match the neighbouring tiles.*

Informally, can we tile my bathroom by turning the tiles around, if necessary?

Answer: *This problem is* tractable.

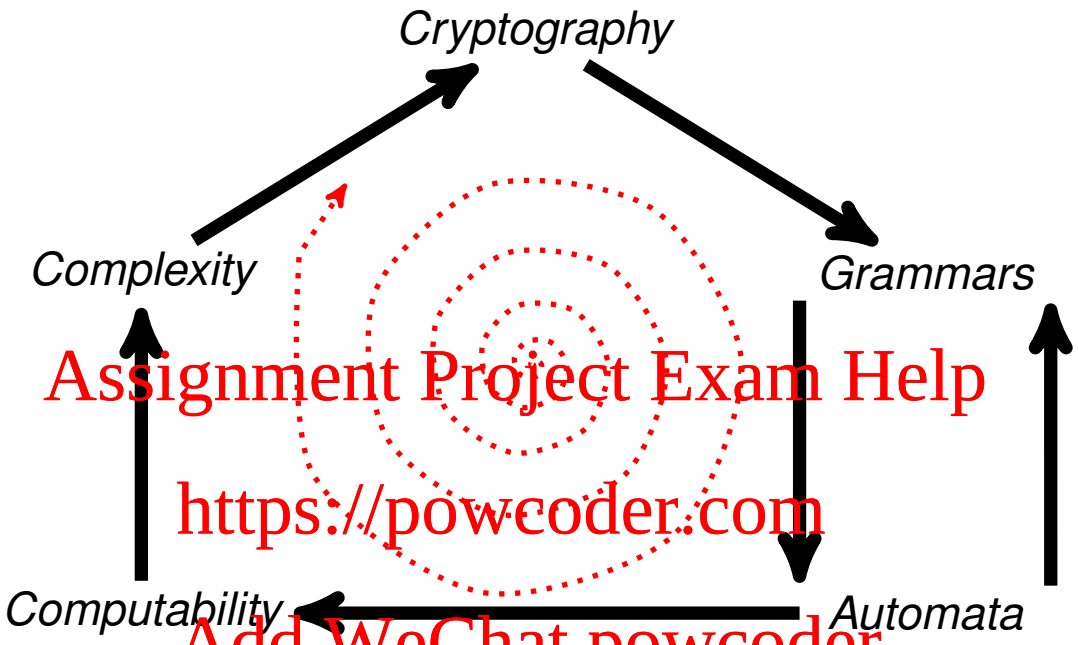
And Finally ...

- *Complexity hierarchy — co-NP, PSPACE, NC, ...*
- *Complexity analyses — time and space*
- *$\mathcal{P} = \mathcal{NP}$?*
- *Theories of concurrency and parallel computing*
- *Semantics of programming languages*
- *Correctness properties of programs*

- *Logic, functional and object-oriented programming paradigms*
- *Artificial intelligence*
- *Automated theorem proving*
- *Database formalisms*
- ...

'There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy' — Hamlet *by William Shakespeare*

The book on computing theory is far from closed, and this is only the introductory chapter. When you are bored: can you write a program whose output is its own source code? (this will **not** *be an exam question!)*



THE END