University of Portsmouth
BSc (Hons) Computer Science
Third Year

Theoretical Computer Science (THEOC)

M21276

September 2025 - January 2026

20 Credits

Thomas Boxall
thomas.boxall1@myport.ac.uk

Contents

Page 1

Lecture - A1: Introduction to Languages

2025-09-29 14:00 Janka

There are two useful decks of slides on Moodle: Introduction to THEOCs and Overview of THEOCs. There is also a *Worksheet 0* which recaps the key concepts from year 1's Architecture & Operating Systems (Maths) and 2nd year's Discrete Maths and Functional Programming.

1.1 Introduction

Languages are a system of communication. The languages we commonly use are built for communicating and passing along instructions to other humans or computers. Depending on the context in which a language is used, will vary the precision which must exist within the language. For example, a language to convey "pub tonight?" to a friend can be as simple as that, where the human can add context clues to fill in the blanks; however to convey `print('hello, world')` to a computer - the language must be precise as it is not designed to interpret sloppy writing.

Languages are defined in terms of the set of symbols (called it's *alphabet*), which get combined into acceptable *strings*, which happens based on rules of sensible combination called *grammar*.

We can take this definition and see it in practice for the English Language:

Alphabet The alphabet for the English Language is Latin: $A = \{a, b, c, d, e, \ldots, x, y, z\}$

Strings (words) Strings are formed from A, for example 'fun', 'mathematics'. The English vocabulary defines which are really strings (for example which appear in the Oxford English Dictionary)

Grammar From the collection of words, we can build sentences using the English rules of *grammar*

Language The set of possible sentences that make up the English Language

Whilst this is an example around a tangible, understandable example - the elements of a formal language are exactly the same however they must be defined without any ambiguity. For example, programming languages have to be defined with a precise description of the syntax used.

1.2 Formalising Language Definitions

Definitions

Alphabet A finite, nonempty set of symbols. For example: $\Sigma = \{a, b, c\}$

String A finite sequence of symbols from the alphabet (placed next to each other in juxtaposition). For example: abc, aaa, bb are examples of strings on Σ

Empty String A string which has no symbols (therefore zero length), denoted Λ.

Language Where Σ is an alphabet, then a language over Σ is a set of strings (including empty string Λ) whose symbols come from Σ

For example, if $\Sigma = \{a, b\}$, then $L = \{ab, aaab, abbb, a\}$ is an example of a language over Σ.

Languages are not finite and they may or may not contain an empty string.

If Σ is an alphabet, then Σ^* denotes the infinite set of all strings made up from Σ - including an empty string. For example, if $\Sigma = \{a, b\}$ then $\Sigma^* = \{\Lambda, a, b, ab, aab, aaab, bba, \ldots\}$. We can therefore say that when looking at Σ^*, a language over Σ is any subset of Σ^*.

Example: Languages

For a given alphabet, Σ, it is possible to have multiple languages. For example:

- $\emptyset$ - an empty language
- $\{\Lambda\}$ - a language containing only an empty string (silly language)
- Σ - the alphabet itself
- Σ^* - the infinite set of all strings made up from the alphabet

Alternatively, we can make this slightly more tangible:

Where $\Sigma = \{a\}$:

- $\emptyset$
- $\{\Lambda\}$
- $\{a\}$
- $\{\Lambda, a, aa, aaa, aaaa, \ldots\}$

1.3 Combining Languages

It is possible to combine languages together to create a new language.

1.3.1 Union and Intersection

As languages are just sets of strings, we can use the standard set operations for Union and Intersection to combine the languages together.

Example: Union and Intersection

Where $L = \{aa, bb, ab\}$ and $M = \{ab, aabb\}$

Intersection (common elements between the two sets): $L \cap M = \{ab\}$

Union (all elements from each set): $L \cup M = \{aa, bb, ab, aabb\}$

1.3.2 Product

The product of two languages is based around concatenation of strings…

The operation of *concatenation of strings* places two strings in juxtaposition. For example, the concatenation of the two strings *aab* and *ba* is the string *aabba*. We use the name *cat* to denote this operation: cat(aab, ba) = aabba. We can combine two languages L and M by forming the set of all concatenations of strings in L with strings in M, which is called the product of two languages.

Definitions

Product of two languages If L and M are languages, then the new language called the product of L and M is defined as $L \cdot M$ (or just LM). This can be seen in set notation below:

$$L \cdot M = \{cat(s,t) : s \in L \text{ and } t \in M\}$$

The product of a language, L, with the language containing only an empty string returns L:

$$L \cdot \{\Lambda\} = \{\Lambda\} \cdot L = L$$

The product of a language, L, with an empty set returns an empty set:

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

The operation of concatenation is not commutative - meaning the order of the two languages matters. For two languages, it's usually true that:

$$L \cdot M \neq M \cdot L$$

Example: Commutativity Laws of Concatenation

For example, if we take two languages: $L = \{ab, ac\}$ and $M = \{a, bc, abc\}$

$$\begin{aligned} L \cdot M &= \{aba, abbc, ababc, aca, acbc, acabc\} \\ M \cdot L &= \{aab, aac, bcab, bcac, abcab, abcac\} \end{aligned}$$

They have no strings in common!

The operation of concatenation is associative. Which means that if L, M, and N are languages:

$$L \cdot (M \cdot N) = (L \cdot M) \cdot N$$

Example: Associativity Laws of Concatenation

For example, if $L = \{a, b\}$, $M = \{a, aa\}$ and $N = \{c, cd\}$ then:

$$\begin{aligned} L \cdot (M \cdot N) &= L \cdot \{ac, acd, aac, aacd\} \\ &= \{aac, aacd, aaac, aaacd, bac, bacd, baac, baacd\} \\ \text{which is the same as} \\ (L \cdot M) \cdot N &= \{aa, aaa, ba, baa\} \cdot N \\ &= \{aac, aacd, aaac, aaacd, bac, bacd, baac, baacd\} \end{aligned}$$

1.3.3 Powers of a Language

For a language, L, the product $L \cdot L$ is denoted by L^2.

The language product L^n for ever $n \in \{0, 1, 2, \ldots\}$ is defined as follows:

$$\begin{aligned} L^0 &= \{\Lambda\} \\ L^n &= L \cdot L^{n-1}, \text{if } n > 0 \end{aligned}$$

Example: Powers of Languages

If we take $L = \{a, bb\}$:

$$
\begin{aligned}
&L^0 = \{\Lambda\} \\
&L^1 = L = \{a, bb\} \\
&L^2 = L \cdot L = \{aa, abb, bba, bbbb\} \\
&L^3 = L \cdot L^2 = \{aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb\}
\end{aligned}
$$

1.4 Closure of a Language

The *closure* of a language is an operation which is applied to a language.

If L is a language over Σ (for example $L \subset \Sigma^*$) then the closure of L is the language denoted by L^* and is defined as follows:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$$

The *Positive Closure* of L is the language denoted by L^+ and is defined as follows:

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$$

So from this we can derive that $L^* = L^+ \cup \{\Lambda\}$. However - it's not necessarily true that $L^+ = L^* - \{\Lambda\}$.

For example, if we take our alphabet as $\Sigma = \{a\}$ and our language to be $L = \{\Lambda, a\}$ then $L^+ = L^*$.

Based on what we now know, there's some interesting properties of closure we can derive. Let L and M be languages over the alphabet Σ:

- $\{\Lambda\}^* = \emptyset^* = \{\Lambda\}$
- $L^* = L^* \cdot L^* = (L^*)^*$
- $\Lambda \in L$ if and only if $L^+ = L^*$
- $(L^* \cdot M^*)^* = (L^* \cup M^*)^* = (L \cup M)^*$
- $L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L$

These will be explored more in the coming Tutorials

1.5 Closure of an Alphabet

As we saw earlier, Σ^* is the infinite set of all strings made up from Σ. The closure of Σ coincides with our definition of Σ^* as the set of all strings over Σ. In other words, it is a nice representation of Σ^* as follows:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

From this, we can see that Σ^k represents the set of strings of length k, for each their symbols are in Σ.

Page 2

Lecture - A2: Grammars

📅 2025-09-29 🕒 15:00 Janka

As we saw in the previous lecture, languages can be defined through giving a set of strings or combining from the existing languages using operations such as productions, unions, etc. Alternatively, we can use a grammar to define a language.

To describe a grammar for a language - two collections of alphabets (symbols) are necessary.

Definitions

Terminal Symbols from which all strings in the language are made. They are symbols of a ‘given’ alphabet for generated language. Usually represented using lower case letters

Non-Terminal Temporary Symbols (different to terminals) used to define the grammar replacement rules within the production rules. They must be replaced by terminals before the production can successfully make a valid string of the language. Usually represented using upper case letters.

Now we know what terminals & non-terminals are - we need to know how to produce terminals from non-terminals. This is where the *Production Rules* come into play. Productions take the form:

$$\alpha \to \beta$$

where α and β are strings of symbols taken from the set of terminals and non-terminals.

A grammar rule can be read in any of several ways:

- “replace α by β”
- “α produces β”
- “α rewrites to β”
- “α reduces to β”

We can now define the grammar.

Definitions

Grammar A set of rules used to define a language - the structure of the strings in the language. There are four key components of a grammar:

1. An alphabet T of symbols called *terminals* which are identical to the alphabet of the resulting language
2. An alphabet N of grammar symbols called *non-terminals* which are used in the production rules
3. A specific non-terminal called the *start symbol* which is usually S
4. A finite set of *productions* of the form $\alpha \to \beta$ where α and β are strings over the alphabet $N \cup T$

2.1 Using a Grammar to Generate a Language

Every grammar has a special non-terminal symbol called a *start symbol* and there must be at least one production with left-side consisting of only the start symbol. Starting from the production rules with the start symbol, we can step-by-step generate all strings belonging to the language described by a given grammar.

As we begin converting from Non-Terminal to Terminal containing strings, we introduce the *Sentential Form*. As we continue to generate strings, we introduce *derivation*.

Definitions

Sentential Form A string made up of terminal and non-terminal symbols.

Derivation Where x and y are sentential forms and $\alpha \rightarrow \beta$ is a production, then the replacement of α with β in $x\alpha y$ is called a derivation. We denote this by writing:

$$x\alpha y \Rightarrow x\beta y$$

During our derivations, there are three symbols we may come across:

- $\Rightarrow$ derives in one step
- $\Rightarrow^+$ derives in one or more steps
- $\Rightarrow^*$ derives in zero or more steps

Finally, we can define $L(G)$: the Language defined by the given Grammar.

Definitions

$L(G)$ If G is a grammar with start symbol S and set of terminals T, then the language generated by G is the following set:

$$L(G) = \{s | s \in T^* and S \Rightarrow^+ s\}$$

Great - now we've seen the theory, lets put it into an example.

Example: Using a Grammar to Derive a Language

Let a grammar, G, be defined by:

- the set of terminals $T = \{a, b\}$
- the only non-terminal start symbol S
- the set of production rules: $S \rightarrow \Lambda,\quad S \rightarrow aSb$
  or in shorthand: $S \rightarrow \Lambda \mid aSb$

Now, beginning the derivations. We have to start with the start symbol S, and we can either derive Λ or aSb. Obviously deriving Λ would end the production and deriving aSb would allow us to keep re-using the production rules:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow \dots$$

Using a combination of the two production rules, we can build up a picture of what strings we can derive from the start symbol:

$$S \Rightarrow \Lambda, S \Rightarrow aSb \Rightarrow ab$$

The second string above we can turn into the following shorthand:

$$S \Rightarrow^* ab$$

Or alternatively, we can use shorthand to jump forward and yet continue the derivation:

$$S \Rightarrow^* aaaSbbb$$

This brings us to the end of the example as we can now define $L(G)$:

$$L(G) = \{\Lambda, ab, aabb, aaabbb, \ldots\}$$

Example: Longer Derivation of a String

Let $\Sigma = \{a, b, c\}$ be the set of terminal symbols and S be the only non-terminal symbol. We have four production rules:

- $S \to \Lambda$
- $S \to aS$
- $S \to bS$
- $S \to cS$

Which can alternatively be represented in shorthand: $S \to \Lambda \mid aS \mid bS \mid cS$

To derive the string *aacb* we would undergo the following derivation:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\Lambda = aacb$$

Which can be shortened to $S \Rightarrow^* aacb$

Note how we started on the left and worked left-to-right. This makes this derivation a *leftmost* derivation because we produced the leftmost characters first.

2.2 Infinite Languages

As in the previous example, note how there is no bound on the length of the strings in an infinite language. Therefore there is no bound on the number of derivation steps used to derive the strings. If the grammar has n productions, then any derivation consisting of $n+1$ steps must use some production twice.

Where a language is infinite - some of the productions or sequence of productions must be used repeatedly to construct the derivations.

Example: Infinite language

Take the infinite language $\{a^n b | n \geq 0\}$ which can be described by the grammar $S \to b \mid aS$.

To derive the string $a^n b$, the production $S \to aS$ is used repeatedly, n times and then the derivation is stopped by using the production $S \to b$.

The production $S \to aS$ allows us to say "If S derives w, then it also derives aw".

2.3 Recursion / Indirect Recursion

A production is called recursive if its left side occurs on it's right side. For example the production $S \to aS$ is recursive.

A production $A \to \dots$ is indirectly recursive if A derives a sentential form that contains A in two or more steps.

Example: Indirect Recursion

If the grammar contains the rules $S \to b \mid aA,\ A \to c \mid bS$ then both productions $S \to aA$ and $A \to bS$ are indirectly recursive:

$$S \Rightarrow aA \Rightarrow abS$$
$$A \Rightarrow bS \Rightarrow baA$$

A grammar can also be considered recursive where it contains either a recursive production or an indirectly recursive production. We can deduce from this that a grammar for an infinite language must be directly or indirectly recursive.

2.4 Constructing Grammars

Up to now, we've looked at deriving a language from a given grammar. Now we will take the inverse - be given a language and construct a grammar which derives the specified language.

Sometimes it is difficult or even impossible to write down a grammar for a given language. Unsurprisingly, a language may have more than one grammar which is correct and valid.

2.4.1 Finite Languages

If the number of strings in a language is finite, then a grammar can consist of all productions of the form $S \to w$ for each string w in the language.

Example: Finite Language

The finite language $\{a, ba\}$ can be described by the grammar:

$$S \to a \mid ba$$

2.4.2 Infinite Languages

To find the grammar for a language where the number of strings is infinite is a considerably bigger challenge. There is no universal method for finding a grammar for an infinite language, however the method of *combining grammars* can prove useful.

Example: Infinite Language

To find a grammar for the following simple language:

$$\{\Lambda, a, aa, \dots, a^n, \dots\} = \{a^n : n \in \mathbb{N}\}$$

We can use the following solution:

- We know the set of terminals: $T = \{a\}$
- We know the only non-terminal start symbol: S
- So therefore we can generate the production rules: $S \to \Lambda,\ S \to aS$

2.5 Combining Grammars

If we take L and M to be languages which we are able to find the grammars; then there exist simple rules for creating grammars which produce the languages $L \cup M$, $L \cdot M$, and L^*. This therefore means we can describe L and M with grammars having disjoint sets (where neither set has common elements) of non-terminals.

The combination process is started by assigning start symbols for the grammars of L and M to be A and B respectively:

$$L : A \to \dots, \quad M : B \to \dots$$

2.5.1 Union Rule

The union of two languages, $L \cup M$ starts with the two productions:

$$S \to A \mid B$$

which is followed by: the grammar rules of L (start symbol A) and then the grammar rules of M (start symbol B).

Example: Combining Grammars Using Union Rule

If we take the following language:

$$K = \{\Lambda, a, b, aa, bb, aaa, bbb, \dots, a^n, b^n, \dots\}$$

Now to find the grammar for it.

Firstly we look at it and see quite clearly there is a pattern, K is a union of the two languages:

$$L = \{a^n | n \in \mathbb{N}\} \text{ and } M = \{b^n | n \in \mathbb{N}\}$$

Therefore we can write a grammar for K as follows:

- $A \to \Lambda \mid aA$ (grammar for L)
- $B \to \Lambda \mid bB$ (grammar for M)
- $S \to A \mid B$ (union rule)

2.5.2 Product Rule

Much the same as the Union Rule, the product of two languages, $L \cdot M$ starts with the production:

$$S \to AB$$

Which is then followed by: the grammar rules of L (start symbol A) and then the grammar rules of M (start symbol B).

Example: Combining Grammars Using Product Rule

If we take the following language:

$$\begin{aligned} K &= \{a^m b^n | m, n \in \mathbb{N}\} \\ &= \{\Lambda, a, b, aa, ab, aaa, bb\} \end{aligned}$$

We can first find out that K is the product of two languages:

$$L = \{a^n | n \in \mathbb{N}\} \text{ and } M = \{b^n | n \in \mathbb{N}\}$$

Therefore we can write a grammar for K as follows:

- $A \to \Lambda \mid aA$ (grammar for L)
- $B \to \Lambda \mid bB$ (grammar for M)
- $S \to AB$ (product rule)

2.5.3 Closure Rule

The grammar for the closure of a language, L^*, starts with the production:

$$S \to AS|\Lambda$$

Which is followed by: the grammar rules of L (start symbol A).

Example: Grammar Closure Rule

If we take the problem that we want to construct a language, L, of all possible strings made up from zero or more occurrences of aa or bb:

$$L = \{aa, bb\}^* = M^*$$

Where $M = aa, bb$

Therefore:

$$L = \{\Lambda, aa, bb, aaaa, aabb, bbbb, bbaa, \ldots\}$$

Therefore, we can write a grammar for L as follows:

- $S \to AS|\Lambda$ (closure rule)
- $A \to aa|bb$ (grammar for $\{aa, bb\}$)

2.6 Equivalent Grammars

Grammars are not unique; a given language can have many grammars which could produce it. Grammars can be simplified down to their simplest form.

Example: Simplifying Grammars

If we take the grammar from the previous example:

$$S \to AS|\Lambda, \quad A \to aa|bb$$

We can simplify this:

- Replace the occurrence of A in $S \to AS$ by the right side of $A \to aa$ to obtain the production $S \to aaS$
- Replace A in $S \to AS$ by the right side of $A \to bb$ to obtain the production $S \to bbS$

We can therefore write the grammar in simplified form as:

$$S \to aaS \mid bbS \mid \Lambda$$

Page 3

Lecture - A3: Regular Languages

2025-10-06 14:00 Janka

3.1 What Are We Trying To Solve Here?

The problem we are trying to solve with Regular Languages is that of precision and absolute certainty - a mathematicians favourite situation.

If we take a statement: "What do we mean by a decimal number?"

We can solve this in a number of ways. One might assume *"Some digits followed maybe by a point and some more digits"* is a good description. However they would be wrong - this is imprecise and inaccurate (mathematicians nightmare).

So we can make it more precise *"Optional minus sign, any sequence of digits, followed by optional point and if so then optional sequence of digit"*. This is obviously better, and now more precise & accurate as we're acknowledging negative numbers are a thing. Although it still isn't great, there's too many words for a mathematician to approve.

So that brings us to the best option: a regular expression:

$$(-+\Lambda)DD^*(\Lambda+.D^*), D \text{ stands for a digit}$$

3.2 Regular Languages

Definitions

Regular Language A formal language that can be described by a regular expression or recognised by a finite automaton

Regular Languages are extremely useful, they are easy to recognise and describe. They provide a simple tool to solve some problems. We see regular expressions in many places within Computing - for example in pattern matching in the `grep` filter in UNIX systems or in lexical-analyser generators in breaking down the source program into logical units such as keywords, identifiers, etc.

There are four different ways we can define a regular language:

1. Languages that are *inductively* formed from combining very simple languages
2. Languages described by a *regular expression*
3. Languages produced by a grammar with a special, very restricted form
4. Languages that are accepted by some finite automaton (covered in subsequent lectures)

3.3 Defining a Regular Language with Induction

Definitions

Induction This is a process which works through the problem, situation, etc in a step-by-step way. We can inference from one step to another, for example if we know 1 is a number then 1+1 will be a number.

Defining a regular language by induction starts with the basis of a very simple language which then gets combined together in particular ways. For example, if we take L and M to be regular languages then the following languages are also regular:

$$L \cup M, L \cdot M, L^*$$

So to generalise this, for a given alphabet Σ: all regular languages over Σ can be built from combining these four in various ways by recursively using the union, product and closure operation.

Example: Regular Language Definitions

For this example, we take $\Sigma = \{a, b\}$.

This gives us four regular languages:

$$\emptyset, \{\Lambda\}, \{a\}, \{b\}$$

Ex. 1: Is the language $\{\Lambda, b\}$ regular?

Yes, it can be written as the union of two regular languages $\{\Lambda\}$ and $\{b\}$:

$$\{\Lambda\} \cup \{b\} = \{\Lambda, b\}$$

Ex. 2: Is the language $\{a, ab\}$ regular?

Yes, it can be written as the product of the two regular languages $\{a\}$ and $\{\Lambda, b\}$:

$$\{a, ab\} = \{a\} \cdot \{\Lambda, b\} = \{a\} \cdot (\{\Lambda\} \cup \{b\})$$

Ex. 3: Is the language $\{\Lambda, b, bb, \ldots, b^n, \ldots\}$ regular?

Yes, it is the closure of the regular language $\{b^*\}$:

$$\{b\}^* = \{\Lambda, b, bb, \ldots, b^n, \ldots\}$$

Ex. 4: Is the language $\{a, ab, abb, \ldots, ab^n, \ldots\}$ regular?

Yes, we can construct it:

$$\{a, ab, abb, \ldots, ab^n, \ldots\} = \{a\} \cdot \{\Lambda, b, bb, \ldots, b^n, \ldots\} = \{a\} \cdot \{b\}^*$$

Ex. 5: Is the language $\{b, aba, aabbaa, \ldots, a^n ba^n, \ldots\}$ regular?

No. This cannot be regular because we have now way to ensure that the two sets of a are both repeated n times.

There are additional examples in the Lecture A3 slides on Moodle.

So what we've learnt from the above is that regular languages can be finite or infinite, and that they cannot have the same symbol repeated in two different places the same number of repetitions.

3.4 Defining a Regular Language with Regular Expressions

Definitions

Regular Expression A sequence of characters that define a specific search pattern for matching text

In our use case, a *Regular Expression* is a shorthand way of showing how a regular language is built from the bases set of regular languages. It uses symbols which are nearly identical to those used to construct the language. Any given regular expression has a language closely associated with it.

For each regular expression E, there is a regular language $L(E)$.

Much like the languages they represent, a regular expression can be inductively manipulated to form new regular expressions. For example if we take R and E as regular expressions then the following are also regular:

$$(R), R + E, R \cdot E, R^*$$

Example: Regular Expresssions

If we take the alphabet to be $\Sigma = \{a, b\}$ then listed below are some of the infinitely many regular expressions:

$$\Lambda, \emptyset, a, b$$

$$\Lambda + b, b^*, a + (b \cdot a), (a + b) \cdot a, a \cdot b^*, a^* + b^*$$

Much like maths, we have an order of operations to help us understand how to interpret a given regular expression. This goes, evaluated first to last: ( ), $*$, $\cdot$, $+$

It's worth noting that the $\cdot$ symbol is often dropped so instead of writing $a + b \cdot a^*$, you would write $a + ba^*$; in it's bracketed form - this would be $(a + (b \cdot (a^*)))$.

The symbols of the regular expressions are distinct from those of the languages, as can be seen in the following table. The language will always be either an empty set, or a set.

Regular Expression	Language
$\emptyset$	$L(\emptyset) = \emptyset$
Λ	$L(\Lambda) = \{\Lambda\}$
a	$L(a) = \{a\}$

Table 3.1: Comparison of Regular Expression syntax and Language syntax

There are two binary operations on regular expressions ($+$ and $\cdot$) and one unary operator ($*$). These are closely associated with the union ($+$), product ($\cdot$) and closure ($*$) operations on the corresponding languages.

Example: Regular Language Operations

The regular expression $a + bc^*$ is effectively shorthand for the regular language:

$$\{a\} \cup (\{b\} \cdot (\{c\}^*))$$

Example: Translating a Regular Expression into a Language

If we take the regular expression $a + bc^*$, we can find it's language:

$$\begin{aligned} L(a + bc^*) &= L(a) \cup L(bc^*) \\ &= L(a) \cup (L(b) \cdot L(c^*)) \\ &= L(a) \cup (L(b) \cdot L(c)^*) \\ &= \{a\} \cup (\{b\} \cdot \{c\}^*) \\ &= \{a\} \cup (\{b\} \cdot \{\Lambda, c, c^2, \ldots, c^n, \ldots\}) \\ &= \{a\} \cup \{b, bc, bc^2, \ldots, bc^n, \ldots\} \\ &= \{a, b, bc, bc^2, \ldots, bc^n, \ldots\} \end{aligned}$$

Example: Translating from Language to Regular Expression

If we take the regular language:

$$\{\Lambda, a, b, ab, abb, abbb, \ldots, ab^n, \ldots\}$$

We can represent with a regular expression:

$$\Lambda + b + ab^*$$

We've used *union* not *product* because the language doesn't include a leading b; there are three options for the strings structure:

Λ the empty string

b a singular b on it's own

ab^* a single a followed by zero or more b

Regular Expressions may not be unique, in that two or more different regular expressions may represent the same languages. For example the regular expressions $a + b$ and $b + a$ are different, but they both represent the same language:

$$L(a + b) = L(b + a) = \{a, b\}$$

We can say that regular expressions R and E are equal if their languages are the same (i.e. $L(R) = L(E)$), and we denote this equality in the familiar way $R = E$.

There are many general equalities for regular expressions. All the properties hold for any regular expressions R, E, F and can be verified by using properties of languages and sets.

Additive (+) properties

$$\begin{aligned} R + E &= E + R \\ R + \emptyset &= \emptyset + R = R \\ R + R &= R \\ (R + E) + F &= R + (E + F) \end{aligned}$$

Product ($\cdot$) properties

$$\begin{aligned} R\emptyset &= \emptyset R = \emptyset \\ R\Lambda &= \Lambda R = R \\ (RE)F &= R(EF) \end{aligned}$$

Closure Properties

$$\begin{aligned}
\emptyset^* &= \Lambda^* = \Lambda \\
R^* &= R^*R^* = (R^*)^* = R + R^* \\
R^* &= \Lambda + RR^* = (\Lambda + R)R^* \\
RR^* &= R^*R \\
R(ER)^* &= (RE)^*R \\
(R+E)^* &= (R^*E^*)^* = (R^* + E^*)^* = R^*(ER^*)^*
\end{aligned}$$

Distributive Properties

$$\begin{aligned}
R(E+F) &= RE + RF \\
(R+E)F &= RF + EF
\end{aligned}$$

We can use a combination of these properties to simplify regular expressions and prove equivalences.

Example: Regular Expression Equivalence

Show that: $\Lambda + ab + abab(ab)^* = (ab)^*$ Using the above properties.

$$\begin{aligned}
\Lambda + ab + abab(ab)^* = (ab)^* &= \Lambda + ab(\Lambda + ab(ab)^*) \\
&= \Lambda + ab((ab)^*) \quad (\text{Using } R^* = \Lambda + RR^*) \\
&= \Lambda + ab(ab)^* \\
&= (ab)^* \quad (\text{Using } R^* = \Lambda + RR^* \text{ again})
\end{aligned}$$

3.5 Defining a Regular Language using Regular Grammars

Definitions

Regular Grammar A grammar where each production takes one of the following restricted forms:

$$\begin{aligned}
&B \to \Lambda,\ B \to w, \\
&B \to A, \\
&B \to wA
\end{aligned}$$

Where A, B are non-terminals and w is a non-empty string of terminals.

There are two regulations all regular grammars must adhere to:

- Only one non-terminal can appear on the right hand side of a production
- Non-terminals must appear on the right end of the right hand side

Therefore, $A \to aBc$ and $S \to TU$ are not part of a regular grammar. The production $A \to abcA$ is, however. Obviously things like $A \to aB \mid cC$ are allowed because they are two separate productions.

For any given regular language, we can find a regular grammar which will produce it. However there may also be other non-regular grammars which also produce it.

Example: Regular and Irregular Grammars

If we take the regular language a^*b^*, we can see it has a regular grammar and an irregular grammar.

Irregular Grammar

$$\begin{aligned} S &\to AB \\ A &\to \Lambda \mid aA \\ B &\to \Lambda \mid Bb \end{aligned}$$

Regular Grammar

$$\begin{aligned} S &\to \Lambda \mid aS \mid A \\ A &\to \Lambda \mid bA \end{aligned}$$

Page 4

Lecture - A4: Finite Automata

📅 2025-10-06 🕒 15:00 Janka

This topic will continue into lecture A5 (next Monday).

4.1 Models of Computation

In this module we'll study different models of computation. These are theoretical ways of representing the computation which is going on within the computer for a given scenario. Examples include *finite automata*, *push-down automata* and *Turing machines*.

All these models have an *input tape*. This is a continuous input string which is divided up into single string segments. The models can either accept or reject the input strings based on their rules. The set of all accepted strings over the alphabet is the language recognised by the model.

They have different types of memory - some may have finite and others infinite. Some models may have additional features.

4.2 Finite Automata: An Introduction

The most basic model of a computer is the *Finite Automata* (FA). These have three components:

- an *input tape* which contains an input string over Σ
- a *head* which reads the input string one symbol at a time
- some *memory* which is a finite set of Q states. The FA is always only in one state, called a *current state* of the automaton

The *program* of the FA defines how the symbols that are read change the current program.

Finite Automata are commonly represented as a transition graph (directed graph, cue flashbacks to DMAFP) because they are simpler to interpret than the formal definitions, which we will cover later.

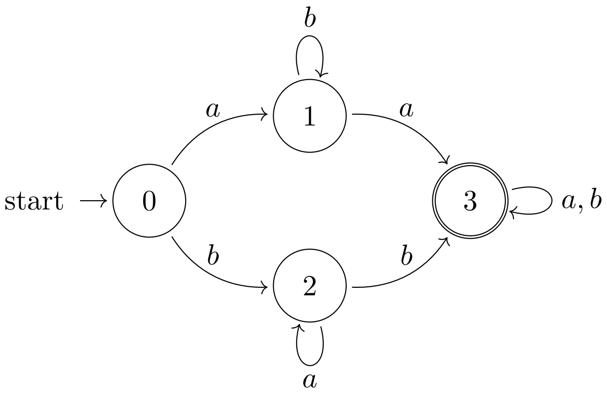


Figure 4.1: Example of Finite Automata

All *finite automata* will have one *initial* state and at least one *final state* (denoted by a double circle).

FAs work by starting in the initial state (0) and as we read off symbols in the string we move from state-to-state (vertex-to-vertex). If after reading the entire input string, the automaton is in the final state - the input string is accepted; if the automaton is not in the final state - the input string is rejected.

To define the function of a FA in mathematical terms - they read a finite-length input string over Σ, one symbol at a time. A FA is always in a *state*, from the set of states Q. They begin in a designated initial (start) state, then on reading a symbol - the state changes which is called a transition. The new state depends on the current state and the symbol read in. There is no option to re-read the input symbols or to write them anywhere. At the end of the string, the machine either accepts the string if and only if its state is one of the final state, otherwise it gets rejected. The language of the automaton is the set of strings it accepts.

Example: Finite Automata Processing

Looking at the Automata in Figure 4.1, we can take the example input *abbbba*.

This would start in state 0, and travel to state 1, accepting the initial a. The automata then loops around from state 1 to state 1 accepting the b, which is repeated 4 times in total. The automata then takes the final a and transitions from state 1 to 3. As we have processed all the input string and we are in the final state - the input string is accepted.

Below is a representation of the transitions taken by the automata to process the input string:

$$0 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{a} 3$$

4.2.1 State Transition Functions

As much as pretty pictures of Finite Automata are all well and good - there is a second way to represent the transitions: using *transition functions*.

Transition functions take the form:

$$T : Q \times \Sigma \to Q$$

Where Q is the set of states and Σ is the alphabet. We can see below an abstracted FA showing two states i and j, with a single symbol a:

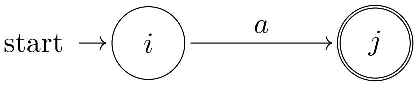


Figure 4.2: Abstracted FA with two states

In the above Finite Automata (Figure 4.2) we can see how the transition function behaves. It is represented by $T(i, a) = j$, where $i, j \in Q$ and $a \in \Sigma$.

Example: State Transition Functions

If we take a more complex Finite Automata:

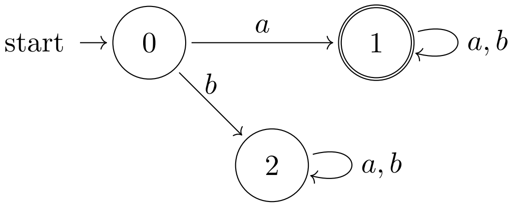


Figure 4.3: Example Finite Automata for Transition Functions

We know it has a set of states: $Q = \{0, 1, 2\}$; a start state: 0; and some final state(s): 1.

We can take the transition function and see the possible transitions over $\Sigma = \{a, b\}$:

$$T(0, a) = 1,\ T(0, b) = 2,$$
$$T(1, a) = T(1, b) = 1$$
$$T(2, a) = T(2, b) = 2$$

4.3 Deterministic Finite Automata

Definitions

Deterministic Finite Automata a DFA over a finite alphabet Σ is a finite directed graph with the property that each node emits one labelled edge for each distinct element of Σ

Except, hang on - isn't that what we've just seen so far. Yes, all the examples explored in this lecture so far have been DFAs; as there is exactly one option of transition for every state and every symbol, with every node in the graph having exactly one edge coming out for each possible input symbol.

We can define DFA more formally in that a DFA *accepts* a string w over Σ^* if there is a path from the start state to a final state such that w is the concatenation of the edges of the path; otherwise the DFA *rejects* w.

We also need to know that the set of all strings over Σ accepted by a DFA M is called the language of M and is denoted as $L(M)$.

For any regular language, a DFA can be found which recognises it. This will be proved in the next lecture.

Example: Constructing a DFA for a given Regular Expression

If we take the following regular expression:

$$(a + b)^* abb \text{ over the alphabet } \Sigma = \{a, b\}$$

We can make an observation: the language is the set of strings that begin with anything but must end with the string *abb* therefore we're looking for strings which have a particular patten to them. This method could be extended if we had a bigger alphabet, for example if we were looking for all strings ending in .tex, or .pdf.

The challenge with this regular expression is that we won't know when the string will end. For example, the string could be *abb*, or *abababababb*. So to get around this, we will keep track of the last three symbols we've seen:

- If in state 1: the last character was a
- If in state 2: the last two symbols were ab
- If in state 3: the last three wee abb

With this in mind, we can now construct the DFA.

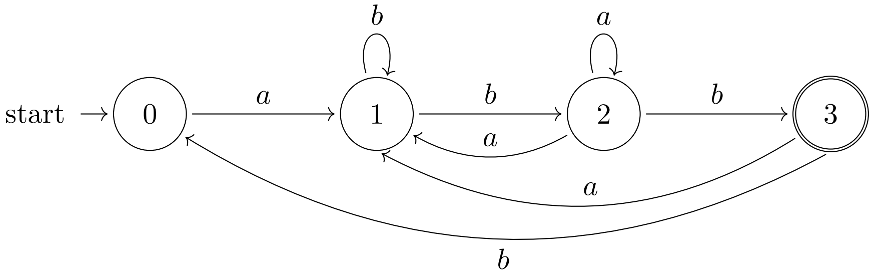


Figure 4.4: DFA constructed from Regular Expression

There are additional examples of DFA construction for a given regular expression in the slides available on Moodle.

4.4 Non-Deterministic Finite Automata

As we know with *Deterministic* Finite Automata - we know exactly which state it is in and the path it took to get there for any given input string. To take the inverse of this - *Non-Deterministic* Finite Automata (NFA) may have more than one option we can follow with the same input character, or there may be no option for a given input character. A NFA accepts and rejects strings in the same way as a DFA: accepting any string which ends up in its final state and rejecting everything else.

A NFA over an alphabet Σ is a finite transition graph with each node having zero or more edges. Each edge is labelled with either a letter from Σ or Λ. Multiple edges may go from the same node with the same label, and some letters may not have an edge associated with them - strings following such paths are rejected.

If an edge is labelled with the empty string Λ, then we can move to the next state (along the edge) without consuming an input letter - effectively we could be in either state and so the possible paths could branch. If there are two edges with the same label from one node, we can move along any of them.

Example: Construct a NFA for a given Regular Expression

If we take the regular expression $ab + a^*a$. We can draw a NFA to recognise the language of it.

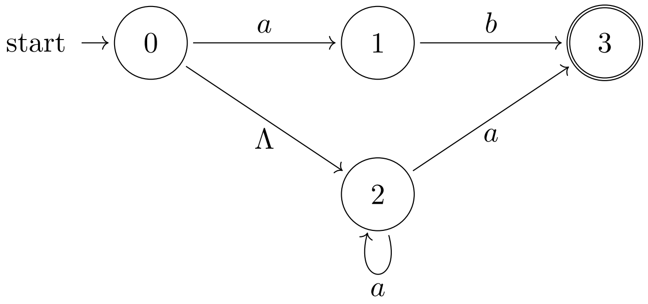


Figure 4.5: Example Non-Deterministic Finite Automata

In this NFA, we see that the “upper” path corresponds to ab and the “lower” path to a^*a. We know this is a NFA because it has a Λ edge and two $a-$edge from state 2.

Due to the non-deterministic nature of a NFA, the output of the transition functions are sets of states, $T : Q \times \Sigma \to P(Q)$.

Example: NFA Transition Functions

For example, if there are no edges from state k labelled with a, we'll write:

$$T(k, a) = \emptyset$$

If there are three edges from state k all labelled with a going to states i, j, and k, we'll write:

$$T(k, a) = \{i, j, k\}$$

Looking back at Figure 4.5 from the previous example, we can see there are four states 0, 1, 2, 3; where 0 is the starting state and 3 is the final state. From here we can see the transition functions:

$$\begin{aligned} T(0, a) &= \{1\} \\ T(0, \Lambda) &= \{2\} \\ T(1, b) &= \{3\} \\ T(3, a) &= \{2, 3\} \end{aligned}$$

4.5 DFA vs. NFA

All digital computers are deterministic. The usual mechanism for deterministic computers is to try one particular path and then to backtrack to the last decision point if that path proves to be poor. Parallel computers make non-determinism almost realisable; for example, we can let each process make a random choice at each branch point thereby exploring many possible trees.

Generally speaking, NFAs are easier to construct and tend to be simpler with fewer states, for a given regular expression to recognise. However, DFAs are easier to operate as the path followed is always unique. Given that they recognise the same language, one is always able to find a DFA which recognises the language of a given NFA. DFAs are a subset of NFAs, so we only need to show that we can map any NFA into a DFA.

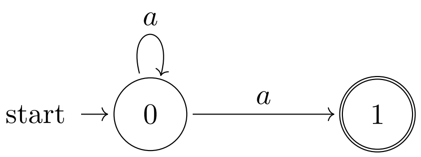


Figure 4.6: Example NFA for a^*a

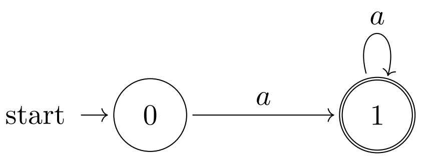


Figure 4.7: Example DFA for a^*a

4.6 Finding an Equivalent DFA for a given NFA

We can prove the equivalence of NFAs and DFAs by showing how for any NFA by constructing a DFA which recognising the same language. Generally the DFA will have more possible states than the NFA; if the NFA has n states then the DFA could have as many as 2^n states.

Example: Converting a NFA to a DFA

If we take the following NFA which recognises the language $(a+b)^*ab$ over the alphabet $\{a, b\}$

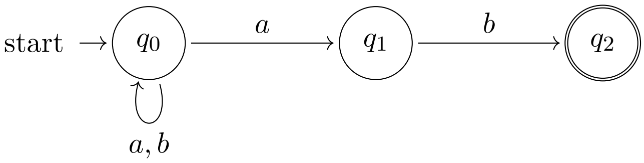


Figure 4.8: NFA To Be Converted

Step 1

Begin in the NFA stat state; if it is connected to any others by Λ, the DFA start state could be a set of states.

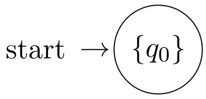


Figure 4.9: Start symbol of DFA

Step 2

For each symbol - determine the set of possible NFA states you could be in after reading it. This set is a label for a new DFA state and is connected to the start by that symbol. In our example - the start state is q_0, but following an a you could be in q_0 or q_1; following a b you could only be in state q_0.

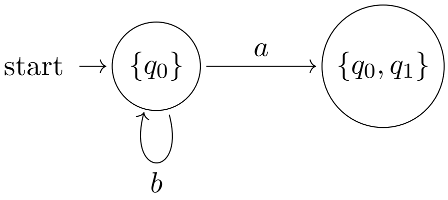


Figure 4.10: First step of converting NFA to DFA

Step 3

Repeat step 2 for each new DFA state, exploring the possible results for each symbol until the system is closed.

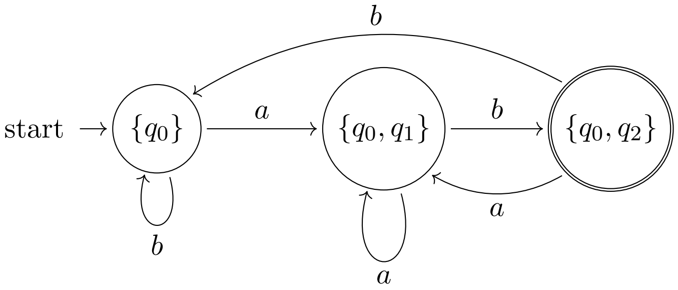


Figure 4.11: DFA showing all valid states

The final state of the DFA are those that include an NFA final state in the set.

If there is no transition for a state / a symbol in the NFA (non-acceptance of the string), create a new state in the DFA labelled $\emptyset$ and add loops for all symbols (a non-final trap state).

4.6.1 Trap States

Definitions

Trap State A state in which the machine cannot reach any final or accepting state.

Trap States are needed in DFAs because to satisfy the requirement to be a DFA - every state must have an outgoing transition for every symbol in the alphabet. This is not an issue in NFAs because the Automaton assumes that if it can't find a suitable transition to use - the input string is invalid.

Example: Trap States in Action

If we take the regular expression from our previous example, $(a+b)^*ab$ and expand the alphabet used to be $\{a, b, c\}$. This presents a problem as the input could contain c, but there's no suitable transitions the DFA can take for such a letter. We use a *trap state* to catch this pesky c.

We can add a trap state to the DFA we created in the previous example.

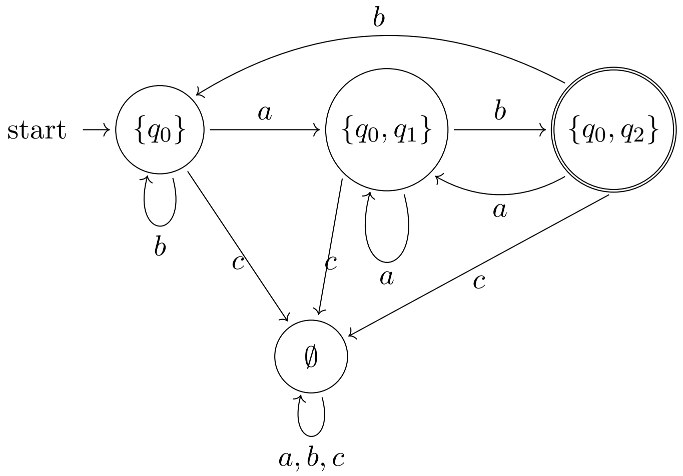


Figure 4.12: DFA showing a Trap State

Page 5

Lecture - A5: Finite Automata and Regular Languages

2025-10-13 14:00 Janka

5.1 Introduction

This lecture will look at the idea that we can construct a NFA from any regular expression and vice versa.

Looking at the production *Regular Expressions ⇒ Finite Automata*, we can show that for any regular expressions it is possible to find a NFA which recognises it. Therefore this proves:

$$L(\text{Regular expressions}) \subseteq L(\text{NFA})$$

Looking at the production *Finite Automata ⇒ Regular Expressions*, we can show that for a given NFA it is possible to find a Regular Expression which defines the same language. Therefore this proves:

$$L(\text{NFA}) \subseteq L(\text{Regular Expression})$$

This means, if we combine both of the previous results:

$$L(\text{NFA}) = L(\text{Regular Expression})$$

Let's prove it...

5.2 Regular Expression ⇒ Finite Automata

Given a regular expression, we will construct a finite automaton (NFA or DFA) which recognises its language. We can do this because the operations within a regular expression (union, product, and closure) can be translated into the directed graph style of a FA using a set of rules.

Rule 0 Start the algorithm with a draft of a machine that has: a start state; a single final state; and an edge labelled with the given regular expression.

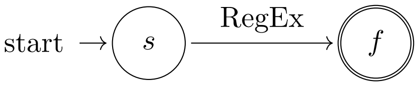


Figure 5.1: Rule 0 of Regular Expression ⇒ Finite Automata

Rule 1 If any edge is labelled with ∅, then erase the edge

Rule 2 Transform any edge of the form:

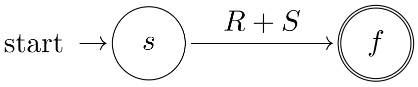


Figure 5.2: Rule 2 (input) of Regular Expression ⇒ Finite Automata

into the edge:

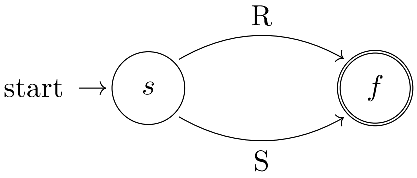


Figure 5.3: Rule 2 (output) of Regular Expression ⇒ Finite Automata

Rule 3 Transform any edge of the form:

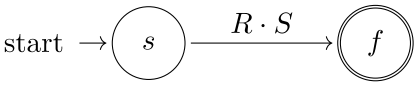


Figure 5.4: Rule 3 (input) of Regular Expression ⇒ Finite Automata

into the edge:

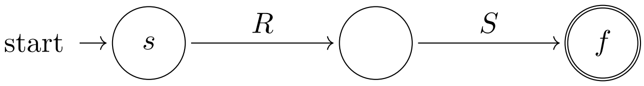


Figure 5.5: Rule 3 (output) of Regular Expression ⇒ Finite Automata

Rule 4 Transform any part of the diagram:

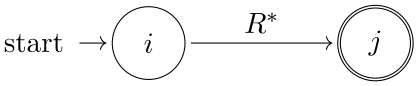


Figure 5.6: Rule 4 (input) of Regular Expression ⇒ Finite Automata

into the diagram:

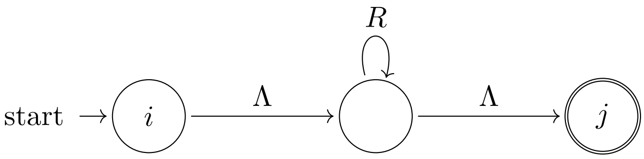


Figure 5.7: Rule 4 (output) of Regular Expression ⇒ Finite Automata

Continue these operations until no labels can be broken up any further.

Now we know the theory - lets see it in practice.

Example: Construct a NFA for a given Regular Expression

If we take the regular expression $a^* + ab$.

We start with **rule 0**:

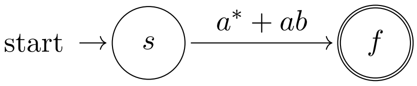


Figure 5.8: Applied rule 0

Now we see that the "last" operation applied to the regular expression is the union, so this is the first we undo by applying **rule 2**:

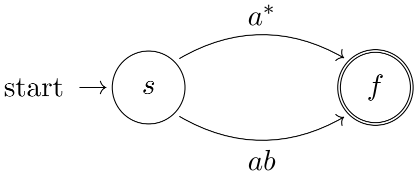


Figure 5.9: Applied rule 1

We have two options of which rule to apply next, either 3 or 4. We will apply **rule 4**:

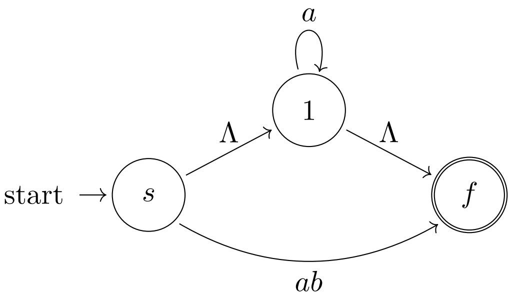


Figure 5.10: Applied rule 4

We can then apply **rule 3**:

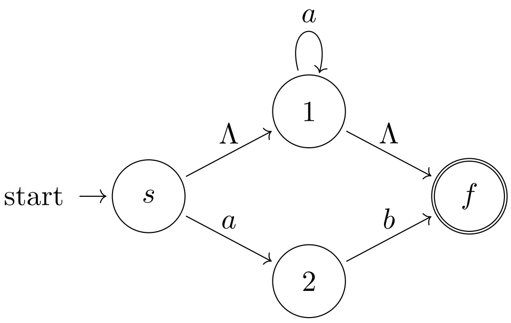


Figure 5.11: Applied rule 3

This generic formula can be applied to any regular expression.

5.3 Finite Automata ⇒ Regular Expression

Rather than adding states, as we have just seen, we are looking to eliminate states and compose the transitions into more complex expressions until we reach just the start and final state exist, which are connected by the final regular expression. There is an algorithm which can be used to work through this...

Step 1 Create a new start state s, and draw a new edge labelled with Λ from s to the original start state. This transforms the FA from:

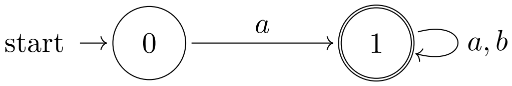


Figure 5.12: Step 1 (input) of Finite Automata ⇒ Regular Expression

into the FA:

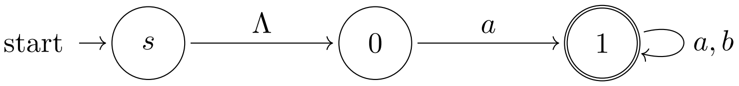


Figure 5.13: Step 1 (output) of Finite Automata ⇒ Regular Expression

Step 2 Create a new final state f, and draw a new edge labelled with Λ from the original final state to f. This transforms the FA from:

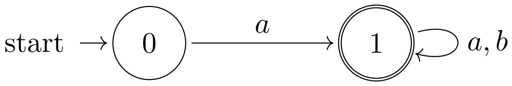


Figure 5.14: Step 2 (input) of Finite Automata ⇒ Regular Expression

into the FA:

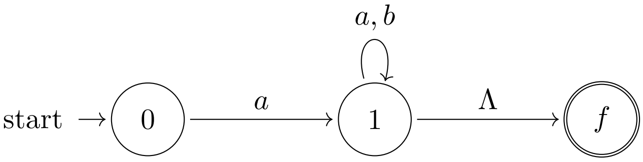


Figure 5.15: Step 2 (output) of Finite Automata ⇒ Regular Expression

Step 3 Merge Edges: For each pair of states, i and j, with more than one edge from i to j - replace all the edges from i to j by a single edge with the regular expression formed by the sum of the labels on each of the edges from i to j. For example:

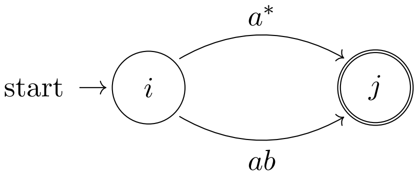


Figure 5.16: Step 3 (input) of Finite Automata ⇒ Regular Expression

gets converted to:

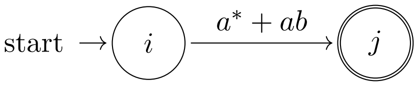


Figure 5.17: Step 3 (output) of Finite Automata ⇒ Regular Expression

Step 4 Eliminate States: Step-by-step eliminate states (one at a time) and change their corresponding labels until the only states remaining are s and f. When we delete a state, we must replace any possible transitions that went through it with a regular expression which carries the information that was removed. For example, if we take the FA:

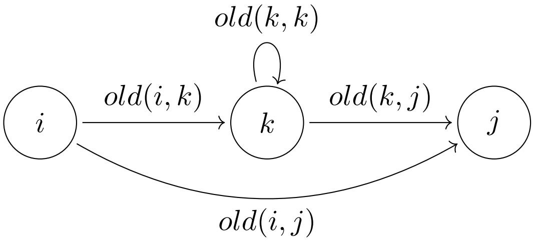


Figure 5.18: Step 4 (input) of Finite Automata ⇒ Regular Expression

We can see that $old(i, j)$ denotes the label on the edge between i and j before elimination; similarly for $old(k, j)$, $old(i, k)$, and $old(k, k)$.

We can now think about how best to represent this with a single edge (consolidating the four edges into one edge). We start building our regular expression which we will label our new single edge with. Looking at the FA - we can see there are two possible paths, the 'top' and the 'bottom' therefore we know our Regular Expression will take the form $new = bottom + top$. The 'bottom' path will be $old(i, j)$, so we can substitute that into the regular expression: $new = old(i, j) + top$. We can calculate that the top must be $old(i, k)old(k, k) * old(k, j)$, so we can substitute that in:

$$new = old(i, j) + old(i, k)old(k, k) * old(k, j)$$

Now we can substitute a more sensible name for new:

$$new(i, j) = old(i, j) + old(i, k)old(k, k) * old(k, j)$$

This will leave our FA looking something like the following:

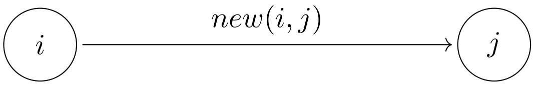


Figure 5.19: Step 4 (output) of Finite Automata ⇒ Regular Expression

If no edge exists, we label it $\emptyset$, for example for a loop.

Step 4 is repeated until all states except s and f are eliminated. We end up with a two-state machine with a single edge between s and f which is labelled with the desired regular expression.

Example: Converting DFA to Regular Expression

If we take the DFA:

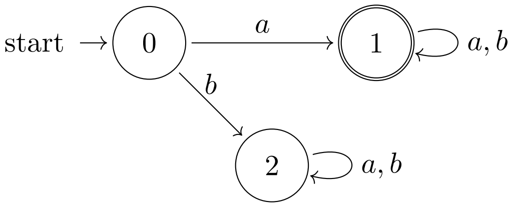


Figure 5.20: Initial DFA for conversion

We can start by applying **step 1** and **step 2** to add start and final states

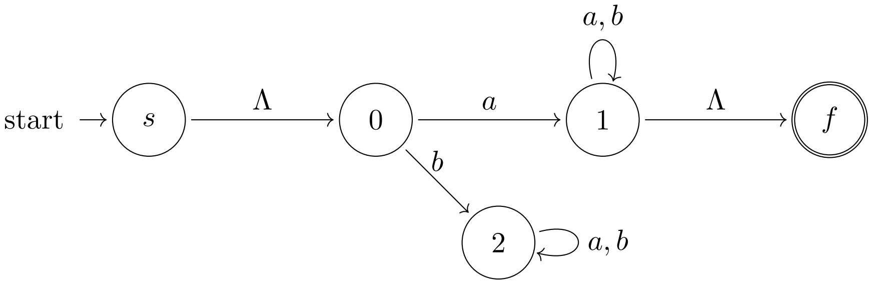


Figure 5.21: Converting DFA to Regular Expression step 1

We then apply **step 3** - which has no effect on the FA.

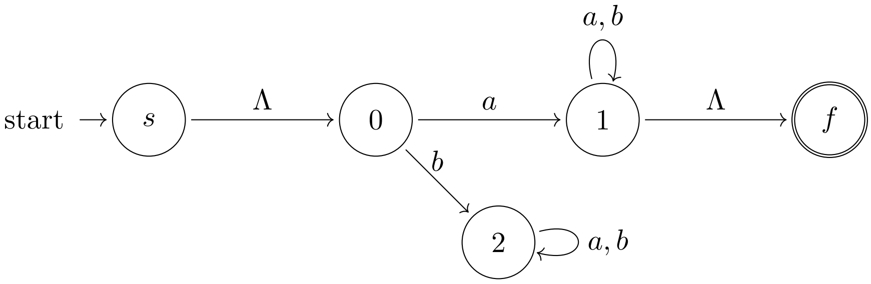


Figure 5.22: Converting DFA to Regular Expression step 2

Now we begin working on **step 4**, which starts by eliminating state 2. This has no change to the other edges, as there are no paths passing through state 2.

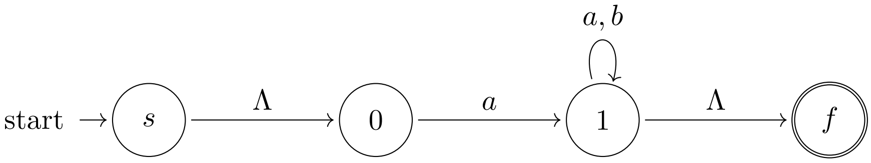


Figure 5.23: Converting DFA to Regular Expression step 3

Continuing with **step 4**, we can eliminate state 0. This creates the label Λa which simplifies to a.

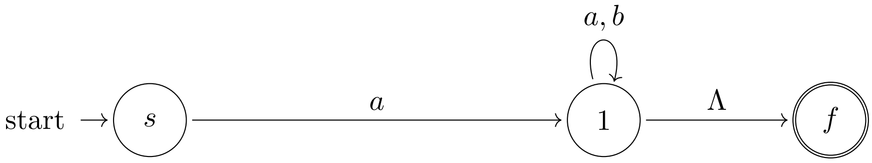


Figure 5.24: Converting DFA to Regular Expression step 4

Finally with **step 4**, we can eliminate state 1.

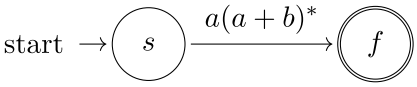


Figure 5.25: Converting DFA to Regular Expression step 5

This gives us the final regular expression: $a(a+b)^*$

5.4 Finding Minimum State DFA

So far we have proven:

$$\text{Regular Expression} \Leftrightarrow \text{NFA} \Leftrightarrow \text{DFA}$$

In some cases our constructed DFAs can be complicated and have more states than are necessary. We can transform a given DFA into a unique DFA with the minimum number of states that recognise the same language.

The *Myhill-Nerode Theorem* states that every regular expression has a unique (up to a simple renaming of the sates) minimum DFA.

There are two parts to finding the minimum state of a given DFA:

Part 1 Find all pairs of equivalent (indistinguishable) states

Part 2 Combine equivalent states into a single state, modifying the transition functions appropriately

5.4.1 Equivalent States

We define two states: s and t to be equivalent (indistinguishable) if for all possible strings w left to consume (including Λ), the DFA after consuming w will finish in the same type of state (final / non-final). This means, that once you arrive in an indistinguishable state (s or t), they always lead to the same result “accept”/“reject” for any given input string.

Two states s and t are not equivalent if $\exists$ a string w such that “following” w from s and t will finish in the final state for one state (s), and in the nonfinal state for the second state (t).

Example: Basic Equivalent States

If we take the following DFA:

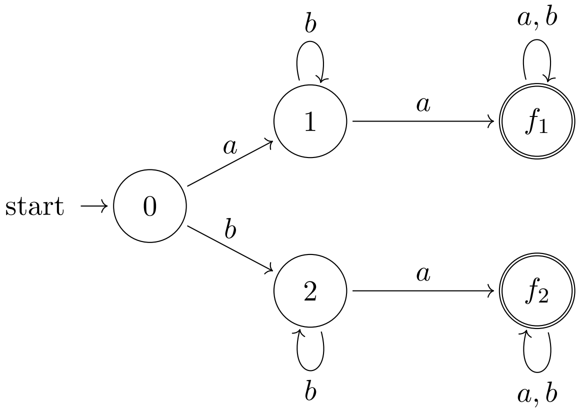


Figure 5.26: Example DFA

We can see that states 0 and 1 are not equivalent. This is because if we take $w = a$, state 0 will transition to state 1 (nonfinal) and state 1 will transition to state f_1 (final).

We can see that states 1 and 2 are equivalent. This is because if we take $w = b^*$, both states will transition to themselves unendingly and never reach a final state; while we if we take $w = a$, both states will transition to their respective final states and accept the string.

Now we've seen this in action - we need to define it in some way useful to us:

1. Begin with clearly distinguishable pairs of states (including final and nonfinal states)

   $$E_0 = \{\{s,t\}|s \text{ and } t \text{ are distinct and either both states are final or both states are nonfinal}\}$$

   For example $E_0 = \{\{1,2\},\{0,1\},\{0,3\},\ldots\}$ but $\{3,4\} \notin E_0$

2. Next eliminate all pairs, which on the same input symbol, lead to a distinguishable pair of states, construct E_1

   $$E_1 = \{\{s,t\}|\{s,t\} \in E_0 \text{ and for every } x \in \Sigma \text{ either } T(s,x) = T(t,x) \text{ or } \{T(s,x),T(t,x)\} \in E_0\}$$

   For example, from the set E_0, we can eliminate $\{0,1\}$ because $T(1,a) = 4$ and $T(0,a) = 4$, and $\{3,4\}$ is not in E_0.

3. We repeat this process until there are some changes: calculating the sequence of sets of pairs $E_0, \supseteq E_1, \subseteq E_2, \subseteq \ldots$ as follows:

   $$_{i+1} = \{\{s,t\}|\{s,t\} \in E_1 \text{ and for every } x \in \Sigma \text{ either } T(s,x) = T(t,x) \text{ or } \{T(s,x),T(t,x)\} \in E_i\}$$

   Stop when $E_{k+1} = E_k$ for some k, the remaining pairs are indistinguishable.

Example: Finding equivalent pairs in DFA

Given a DFA, find the equivalent pairs:

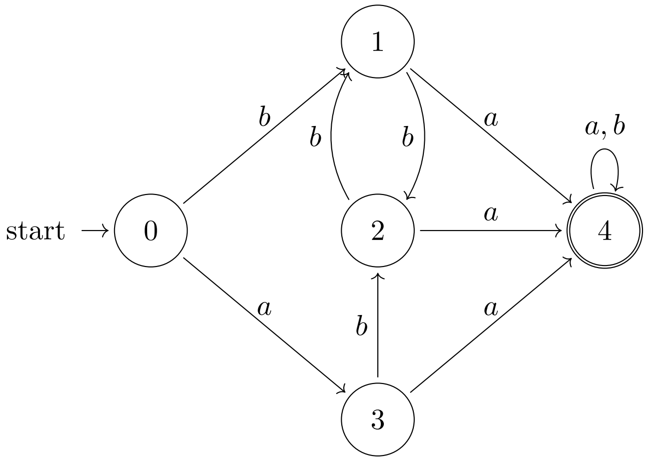


Figure 5.27: DFA to find pairs in

We start by eliminating the simple options: it can't be any pair containing 4 as this is the only finite state. This means $\{0,4\}$, $\{1,4\}$, $\{2,4\}$ and $\{3,4\}$ are distinguishable, and therefore are eliminated.

We can then explore all the pairs containing 0. Working through $\{0,1\}$, $\{0,2\}$ and $\{0,3\}$. We can see that they are all distinguishable when provided the input a; with $\{0,1\}$ ending up in states 3 & 4 respectively (one final and one nonfinal therefore not equivalent), similarly for $\{0,2\}$ ending up in states 3 & 4 respectively and for $\{0,3\}$ ending up in states 3 & 4 respectively.

We can then explore where the resultant states are known to be distinguishable therefore the input states are indistinguishable: $\{1,2\}$, $\{2,3\}$, and $\{1,3\}$.

We can see this in the formal notation below:

$$E_0 = \{\{0,1\},\{0,2\},\{0,3\},\{1,2\},\{0,3\},\{2,3\}\}$$
$$E_1 = \{\{1,2\},\{1,3\},\{2,3\}\}$$
$$E_2 = E_1$$

The relation "be equivalent" is the equivalence relation:

$$[1] = [2] = [3] = \{1,2,3\}; \quad [4] = \{4\}$$

States 1, 2 and 3 are all indistinguishable, therefore the minimal DFA will have three states: $\{0\}$, $\{1,2,3\}$ and $\{4\}$.

5.4.2 Modifying DFA

Now that we have established how to get the minimum number of required states for our DFA to represent the same thing - we can modify the DFA such that it is drawn with the minimum number of states. Again, similar to the Equivalent States method, there is an algorithm to follow to do this:

1. Construct a new DFA where any set of indistinguishable states for a single state in the new DFA
2. The start state will be the state containing the original start state, the final states will be those which contain original final states
3. The transitions will be the full set of transitions from the original states - these should all be

consistent, $T_{min}([s], a) = [T(s, a)]$, where $[s]$ denotes the equivalence class containing s and a is any letter.

Example: Minifying DFA

This is a continuation of the previous example, and will minify the DFA shown in Figure 5.27.

We can construct the minified DFA as follows

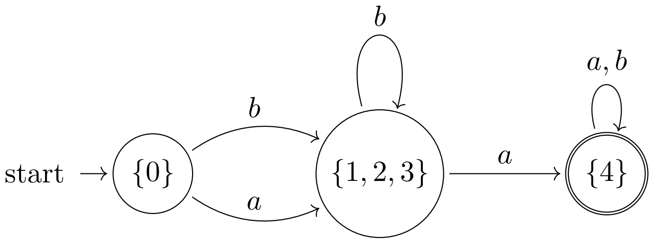


Figure 5.28: Minified DFA

We can now explore the minified transitions. Originally, we had $T(0, a) = 3$ and $T(0, b) = 1$ and now the new transitions are:

$$T_{min}(\{0\}, a) = T_{min}([0], a) = [T(0, a)] = [1] = \{1, 2, 3\}$$
$$T_{min}(\{1, 2, 3\}, a) = 4 \text{ and } T_{min}(\{1, 2, 3\}, b) = \{1, 2, 3\}$$
$$T_{min}(\{4\}, a) = 4 \text{ and } T_{min}(\{4\}, b) = 4$$

5.5 The Complete Cycle

We have now seen the full circle, from Regular Expression through FA, and back to Regular Expression. This means we can now explore:

1. Start with a regular expression exp_0
2. Construct an NFA which recognises the given expression exp_0
3. Transform the constructed NFA to the equivalent DFA
4. Simplify the DFA to the one with the minimum number of states
5. Convert the simplified DFA back to a regular expression, exp_1

In this display of painful TikZ diagrams, we will see this full circle…

Example: Complete Cycle from RegEx to RegEx

Step 1. Start with a Regular Expression

We start with the regular expression $a^* + ba$

Step 2. Construct a NFA which Recognises the given Regular Expression

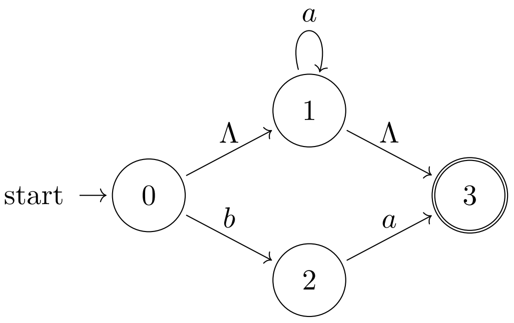


Figure 5.29: NFA which recognises $a^* + ba$

Step 3. Transform the Constructed NFA to the Equivalent DFA

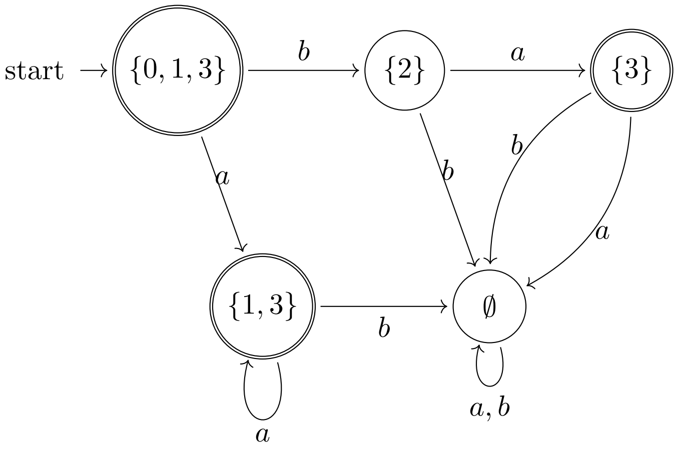


Figure 5.30: DFA which recognises $a^* + ba$

Step 4. Simplify the DFA to the one with the Minimum Number of States

Examining the DFA in Figure 5.30, we can see there are four possibly indistinguishable states: $B, \emptyset$; A, C; C, D; and A, D. However none of them are indistinguishable therefore no states can be removed from this DFA.

Step 5. Convert the Simplified DFA back to a Regular Expression

Starting with the DFA in Figure 5.30, we can add the start and final state, and eliminate any states not leading to f (i.e. the trapped state).

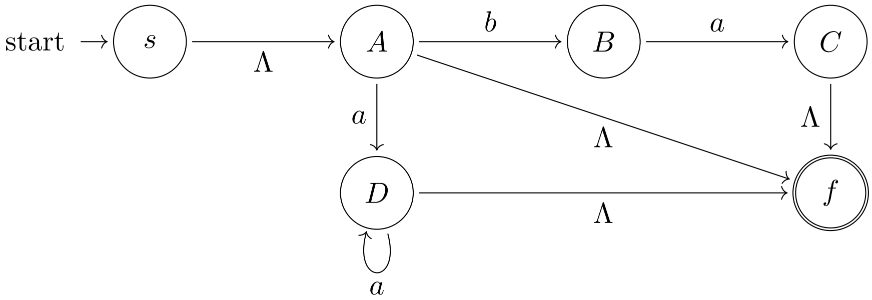


Figure 5.31: NFA having removed the trap state and added s and f

We can now eliminate states B and C.

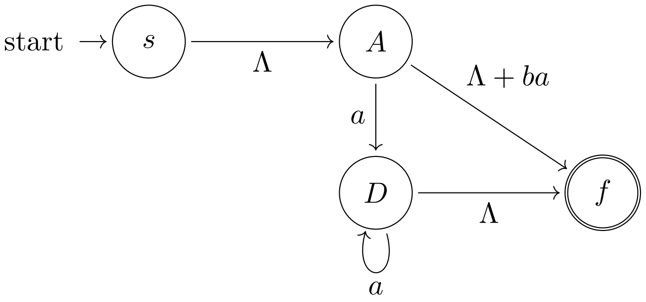


Figure 5.32: NFA having eliminated state B and state C

We can eliminate our final state now, D.

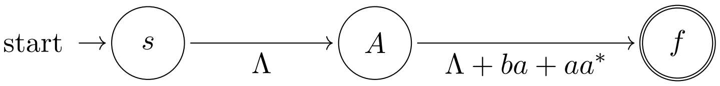


Figure 5.33: NFA having eliminated state D

This leaves us with the regular expression $\Lambda + aa^* + ba$ which is equivalent to our original regular expression.

Page 6

Lecture - A6: What Is Beyond Regular Languages

2025-10-13 15:00 Janka

6.1 NFAs to Regular Grammars

Every NFA can be simply converted into a corresponding regular grammar, and vice versa (remember these from lecture A3). Each state (node) of the NFA is associated with a non-terminal symbol of the grammar; the initial state is associated with the start symbol. Every transition is associated with a grammar production. Every final state has an additional production.

Example: Converting NFA to Regular Grmmars

If we take the subsequent NFA:

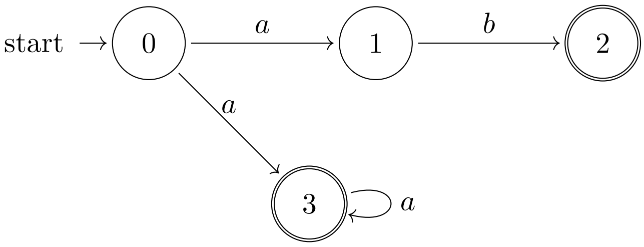


Figure 6.1: Example NFA

Then we can utilise the rules stated above to get it's grammar set:

$$\begin{aligned} S &\to aA \mid aC \\ A &\to bB \\ B &\to \Lambda \\ C &\to aC \mid \Lambda \end{aligned}$$

Obviously in the above we've assigned state 0 the start symbol, S; state 1 the non-terminal A; state 2 the non-terminal B; and state 3 the non-terminal C.

Yes, this isn't the simplest grammar we could produce - but it's clear and shows the point here: all NFAs can be converted into a Regular Grammar

6.2 (Dis)Proving a Language's Regularity

As we saw in lecture A3, a regular language is one such that it can be recognised by regular expression or finite automaton. Naturally, all languages are not regular, for example:

$$\{a^n b^n | n > 0\}$$

This isn't regular because we do not have a way of defining n with it's repeated use. This means that because FAs work without memory (possibly beyond the last state in certain circumstances) - we cannot guarantee that the number of a is equal to the number of b.

We now need a way to prove that this language isn't regular as the hand-wavey explanation above isn't enough, because maths. This is where the *Pumping Lemma* is introduced - which applies for infinite languages (remember all finite languages are regular).

6.3 The Pumping Lemma

The underlying principle explored here is defined with *The Pigeonhole Principle*, which states that if we put n pigeons into m pigeonholes (where $n > m$), then at least one pigeonhole must have more than one pigeon.

Returning to computer science, not feathery beasties, we can see that if the input string is long enough (i.e. greater than the number of states of the minimum state DFA), then there must be at least one state Q which is visited more than once. Therefore there must be at least one closed loop, which begins and ends at state Q and a particular string, y, which corresponds to this loop. A schematic representation of this can be seen in the following figure.

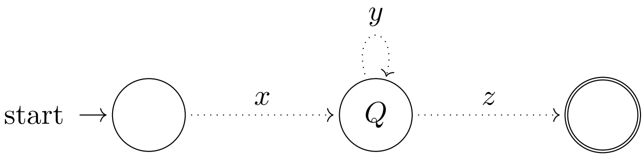


Figure 6.2: Schematic Representation of the Pigeonhole Principle

Each dotted arrow represents a path that may contain other states of the DFA:

- x is a string of letters which the automaton reads from the start to state Q
- y is the string of letters around the closed loop
- z is a string of letters from Q to a final state

We know the string xyz is accepted. But this means that the DFA must also accept xz, $xyyz$, $xyyyz$, ..., $x\underbrace{y\dots y}_{k}z$, ... We say that the middle string, k is "pumped".

We can formalise our theorem of the *Pumping Lemma* as: Let L be an infinite regular language accepted by a DFA with m states. Then any string w in L with at least m symbols can be decomposed as $w = xyz$ with $|xy| \leq m$, and $|y| \geq 1$ such that

$$w_i = x\underbrace{y\dots y}_{i}z$$

is also in L for all $i = 0, 1, 2, \dots$.

The pumping lemma can be used to prove that a language is not regular. However, the pumping lemma alone cannot be used to prove that a language is regular. This is in line with the "Necessary, but not sufficient" condition covered in Discrete Maths at Level 5.

Example: Necessary, but not Sufficient

If L is an infinite regular language (A) then all strings of L must satisfy the pumping lemma (have a pre-described structure) (B)

Case 1

$$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$$

If an infinite language fails to satisfy the pumping lemma (i.e. there exists a string without pre-described structure) then it cannot be regular.

Case 2

$$A \Rightarrow B \not\equiv B \Rightarrow A$$

But if all strings satisfy the pumping lemma (have pre-described structure), this alone does not prove the language is regular.

Example: Pumping Lemma in Action

This example will prove by contradiction using the pumping lemma the theorem that $L = \{a^n b^n | n \geq 0\}$ is not regular.

If we assume that L is regular, then L is accepted by a unique DFA with m states. Therefore any string from L of length at least m can be decomposed as

$$xyz \text{ with } |xy| \leq m \text{ and } |y| \geq 1$$

such that $x \underbrace{y \dots y}_{k} z$ is also in L for all $i = 0, 1, 2, \dots$.

If we take $n > m$ and the string $a^n b^n$ from L then the substring y (of length k) must consist entirely of a's.

Due to the pumping lemma, the string $a^{n-k} b^n$ must be from L which is not true.

Therefore the assumption that $L = \{a^n b^n | n \geq 0\}$ is regular must be false.

Alternatively to the Pumping Lemma, we can consider the grammar for an Infinite Regular Language. The grammar must contain a production that is recursive or indirectly recursive.

If we take the following grammars:

$$\begin{aligned} S &\to xN \\ N &\to yN \mid z \end{aligned}$$

We can then generate the following production sequence

$$S \Rightarrow xN \Rightarrow xyN \Rightarrow xyyN \Rightarrow xyyyN \Rightarrow \dots$$

Therefore, this grammar accepts all strings of the form $x \underbrace{y \dots y}_{k} z$ for all $k \geq 0$.

6.4 Context Free Language

The grammar of the regular language is too strict and doesn't allow the description of many simple languages, for example $L = \{a^n b^n | n > 0\}$.

To work around this, we will work step-by-step adding more freedom to the grammar production to define other families of the languages.

Definitions

Context Free Grammar A grammar, G where all of it's productions take the form $N \to \alpha$ where N is a non-terminal and α is any string over the alphabet of terminals and non-terminals.

All regular languages are context-free, but not all context-free languages are regular.

From the above examples, we can see that the term "context-free" has come from the requirement that all productions contain a single non-terminal on the left. When this is the case, any production (ie $N \to \alpha$) can be used in a derivation without regard to the "context" in which the grammatical symbol N appears. From this we can derive:

$$aNb \Rightarrow a\alpha b$$

Which we can see the "context" is in reference to whatever surrounds the N.

Example: Context Free Grammars

Ex. 1 The grammar over the alphabet $\{a, b\}$ with productions $S \to aSb \mid \Lambda$ is context-free. This generates the language $L = \{a^n b^n | n \geq 0\}$

Ex. 2 The grammar over the alphabet $\{a, b\}$ with productions $S \to aSa \mid bSb \mid \Lambda$ is context-free. This generates the language $L = \{ww^R : w \in \{a, b\}^*\}$

6.4.1 Non Context-Free Grammars

A grammar that is not context-free must contain a production whose left hand side is a string of two or more symbols.

For example, the production $Nc \to \alpha$ is not part of any context-free grammar; because a derivation that uses this production can replace the non-terminal N *only in a "context"* that has c on the right. For example $aNc \Rightarrow a\alpha$.

6.4.2 Chomsky Normal Form

The grammar of every context-free language can be expressed in a more suitable way: *Chomsky Normal Form*

Definitions

Chomsky Normal Form A context-free grammar is in Chomsky normal form if all productions are of the form $A \to BC$ and $A \to a$ where a is any terminal, and A, B, C are non-terminals (with B and C not being start symbols). If Λ is needed, it is produced via $S \to \Lambda$.

Any context-free grammar has an equivalent grammar in Chomsky normal form.

6.4.3 Context-Free and Programming Languages

The text of a program is easy to understand by humans, but the computer must convert it into a form which it understands. This process is called "parsing" and consists of two parts:

1. The *tokenizer* (or *lexer* or *scanner*), which takes the source text and breaks it into the reserved words, constants, identifiers and symbols that are defined in the language (using a DFA)
2. These tokens are subsequently passed to the actual *parser* which analyzes the series of tokens and determines when one of the language's syntax rules is complete.

Following the language's grammar, a "tree" representing the program is created. Once this form is reached, the program is ready to be interpreted or compiled by the application.

6.5 Context Sensitive Languages

A context-sensitive grammar allows for even more complex transitions.

Definitions

Context-Sensitive Grammar A grammar whose productions are of the form

$$\alpha A\beta \to \alpha\gamma\beta$$

where $\alpha, \beta \in (N \cup T)^*$, $A \in N$; $\gamma \in (N \cup T)^+$ and a rule of the form $S \to \lambda$ is allowed if the start symbol S does not appear on the right hand side of any rule.

The language generated by such a grammar is called a context-sensitive language.

Every context-free grammar is also context-sensitive, therefore the context-free languages are a subset of the context-sensitive languages (see Chomsky Normal Form). However, not every context-sensitive language is context free.

Example: Context Sensitive Languages

If we take the language $L = \{a^n b^n c^n, n \geq 1\}$, which is context-sensitive but not context-free.

It has the following production rules.

$S \to aSBC|aBC$, $CB \to HB$, $HB \to HC$, $HC \to BC$, $aB \to ab$, $bB \to bb$, $bC \to bc$, $cC \to cc$

We can then review a derivation of the string *aabbcc* using this grammar.

$$\begin{aligned}
S &\Rightarrow a\underline{S}bC \\
&\Rightarrow a\underline{aB}CBC && \text{(using } S \to aBC\text{)} \\
&\Rightarrow aab\underline{CB}C && \text{(using } aB \to ab\text{)} \\
&\Rightarrow aab\underline{HB}C && \text{(using } CB \to HB\text{)} \\
&\Rightarrow aab\underline{HC}C && \text{(using } HB \to HC\text{)} \\
&\Rightarrow aa\underline{bB}CC && \text{(using } HC \to BC\text{)} \\
&\Rightarrow aab\underline{bC}C && \text{(using } bB \to bb\text{)} \\
&\Rightarrow aabb\underline{cC} && \text{(using } bC \to bc\text{)} \\
&\Rightarrow aabbcc && \text{(using } cC \to cc\text{)}
\end{aligned}$$

The Context-sensitive languages can also be generated by a *monotonic grammar* where any production is allowed permitting there are no rules for making strings shorter (such as $S \to \Lambda$).

6.6 Phrase Structure Grammars

The most general grammars which we can define are *Phrase Structure Grammars* or *Unrestricted Grammars.*

Definitions

Phrase Structure Grammar A grammar whose productions are of the form $\alpha \to \beta$ where $\alpha \in (N \cup T)^+$ and $\beta \in (N \cup T)^*$

The above definition means that α and β can be any sequence of non-terminals and terminals, but β could also be Λ.

The phrase structure grammars generate the most general class of languages, called *recursively enumerable.*

6.7 Chomsky Hierarchy

We can form a hierarchy of languages (called the Chomsky Hierarchy), where each language includes the ones below it. As the grammar rules become less restrictive, the language classes grow, but they include the simpler languages as subsets.

Type 0 Phrase Sensitive

Type 1 Context-Sensitive

Type 2 Context-Free

Type 3 Regular

There are also infinite languages which cannot be generated by a finite set of recursive productions which are known as *non-grammatical* languages.

Page 7

Lecture - A7: Pushdown Automata

📅 2025-10-20 🕑 14:00 Janka

We have already seen how *Deterministic Finite Automata* and *Non-Deterministic Finite Automata* can be used to identify regular languages.

Today we will explore the wonderful world of *(Non-Deterministic) Pushdown Automata* (PDA) and how these can be used to identify context-free languages. When we talk about a PDA, we are talking about a Non-Deterministic Pushdown Automata (NPDA) by default.

7.1 Non-Deterministic Pushdown Automata

NPDAs are like finite automata, in that they read a single input character from the input tape at a time and may or may not perform a transition based on this. However, NPDAs also have a stack memory where they can store an arbitrary amount of information.

7.1.1 The Stack Memory

As we have seen in modules of years gone past, the stack operates in *Last In First Out* where only the top element can be operated on in a single operation.

There are three operations we can do with the stack:

pop reads the top symbol and removes it from the stack

push writes a designed symbol onto the top of the stack; for example $push(X)$ means put X on the top of the stack

nop does nothing to the stack

The symbols put onto the stack are different from the language's alphabet which is used on the input tape.

At the start of processing a fresh input on the input tape - the stack starts with only the initial stack symbol ($) on the stack. The automaton starts in it's initial state, as we'd expect.

7.1.2 Transitions

For each step, there are three inputs used to determine the transition:

- The Current State,
- The Input Element,
- The Top Symbol of The Stack

One transition step includes:

- changing the state (as with FAs)
- (optional) reading a symbol from the input tape and moving to the next right symbol (as with FA)

- change the stack - push a symbol onto the stack, pop a symbol off the stack, no change to the stack

Transition steps are formally defined by transition functions, although often represented in the form of transition instructions.

For each transition function, there are three inputs: the state, the input character (which can be Λ), and the character which must be at the top of the stack for the input condition to hold. There are two outputs of the transition functions: the "new" state, and a stack operation.

There are three different ways we can represent the transition.

Firstly, we can see the transition on a transition diagram. The label on the edge is vital here - as it gives us the input character, stack character, and output operation on the stack. The two states show the initial state, and the output state, as with a FA.

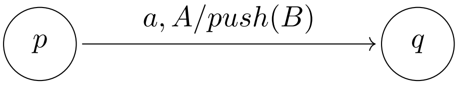


Figure 7.1: Transition Diagram representation of a transition

Alternatively, we can write this as a transition function:

$$T(p, a, A) = (push(B), q)$$

Where here we can see the left hand side are the three inputs: the input state, the input character, and the stack input character; and the right hand side shows us the output operation on the stack and the state to transition to.

Finally, we can re-write this as a transition instruction:

$$(p, a, A, push(B), q)$$

Here we can see that there is a single set of brackets containing, in order: the initial state; the input character; the stack input character; the stack output function; and the state to transition to.

7.1.3 Formal Definition

We can formally describe NPDA as:

- a finite set Q of states (and the start states, and the set of accepting / final states)
- a finite set Σ which is called the input alphabet
- a finite set Γ which is called the stack alphabet (and the initial stack symbol \$)
- a finite set of transition instructions, or a transition functions where

  $$T : Q \times \Sigma \cup \{\Lambda\} \times \Gamma \to \Gamma^* \times Q$$

  or represented by a 'transition' diagram.

An input string is accepted by an NPDA if there is some path (i.e. a sequence of instructions) from the start state to a final state that consumes all letters of the string; otherwise the string is rejected. The language of an NPDA is the set of all strings that it accepts.

There are a few reasons that an NPDA may reject a string:

- If reading an input string finishes without reaching a final state
- If for a current state / symbol on the stack / input symbol there is no transition

- If it attempts to pop the empty stack

Example: $\{a^n b^n | n \geq 0\}$

Yes, $\{a^n b^n | n \geq 0\}$ is back again.

To tackle this problem we first come up with a plan on how to solve it:

1. Begin reading the string, and for each a read push a Y onto the stack
2. On the first b change states, and begin removing one Y from the stack
3. If you reach the end of the input and have just cleared the stack, accept the string
4. Otherwise reject (e.g. if the stack runs out before the input - more b's than a's)

Now we have figured out how we need our NPDA to behave - we can design it.

It will have 3 states: 0 (start), 1, 2 (final); the input alphabet will contain two elements: $\{a, b\}$ and the stack alphabet will also contain 2 elements: $\{Y, \$\}$.

Note here that Λ is not in the input alphabet as we can safely assume it always will be; and that the stack alphabet contains the initial stack symbol \$.

From here we can draw our NPDA.

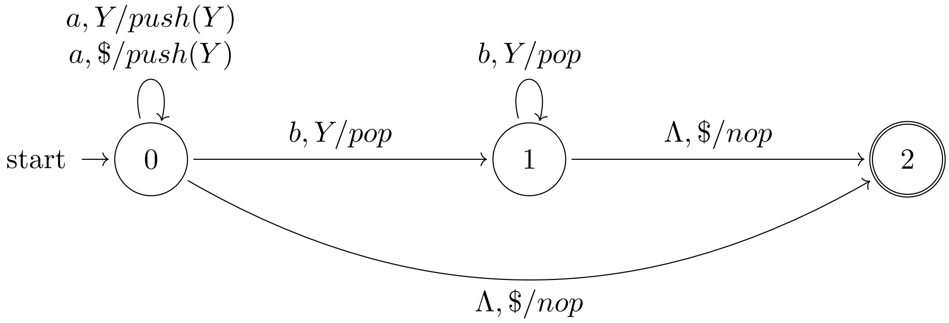


Figure 7.2: NPDA Diagram

From this NPDA - we can learn there are 6 allowed instructions:

$$\begin{aligned}
T_1 &:(0, a, \$, push(Y), 0) \\
T_2 &:(0, a, Y, push(Y), 0) \\
T_3 &:(0, \Lambda, \$, nop, 2) \\
T_4 &:(0, b, Y, pop, 1) \\
T_5 &:(1, b, Y, pop, 1) \\
T_6 &:(1, \Lambda, \$, nop, 2)
\end{aligned}$$

7.1.4 Instantaneous Description

We are able to use *Instantaneous Descriptions* to describe the process the NPDA is going through at any instant. Three are three things we need to keep track of:

- The current state
- What input characters are left
- What is on the stack

The instantaneous descriptions take the form:

(current state, unconsumed input, stack contents)

We can see this in action if we take an NPDA which has the instantaneous description:

$$(0, abba, YZ\$)$$

and the NPDA includes an instruction of the following form:

$$(0, a, Y, pop, 1)$$

After the transition, the instantaneous description is changed to:

$$1, bba, Z\$$$

From this we can see that the transition function has changed the state from 0 to 1, used up the letter a from the input tape and popped Y from the top of the stack.

Example: Instantaneous Descriptions

If we take the string $aabb$, we can represent it's journey through the NPDA with instantaneous descriptions.

$$\begin{aligned}
Start &\rightarrow (0, aabb, \$) \\
T_1 &\rightarrow (0, abb, Y\$) \\
T_2 &\rightarrow (0, bb, YY\$) \\
T_4 &\rightarrow (1, b, Y\$) \\
T_5 &\rightarrow (1, \Lambda, \$) \\
T_6 &\rightarrow (2, \Lambda, \$)
\end{aligned}$$

We have reached the final state, 2, therefore we accept the input string.

Note the transition marker T_n denotes which transition function was used for a given instantaneous description.

7.1.5 NPDAs and Context-Free Languages

As we have seen in previous lectures, the class of context-free languages are generated by context-free grammars which have all their production rules of the form $N \rightarrow \alpha$ where N is a non-terminal α is any string over the alphabet of terminals and non-terminals.

This looks familiar - as we have seen that this $N \rightarrow \alpha$ is the form that the productions take for NPDAs. Therefore we can theorise that the Context-Free languages are exactly the languages that that are accepted by non-deterministic pushdown automata.

This theorem can be proven in two steps (similar to NFA $\Leftrightarrow$ regular languages):

1. If we take a NPDA: we can find a context-free grammar which generates the language accepted by the given NPDA.
2. If we take a context-free language: we can find an NPDA that accepts the given context-free language.

Example: Finding a NPDA for a Context-Free Language

If we take a language containing all strings over a and b with exactly the same number of a's as b's. We can show that this is context-free.

We can plan for this as follows:

- Keep track of the difference between the number of a's and b's we've read by changing the symbol in the stack.
- Use one symbol, X, if we've seen more as and another, Y, if we've seen more bs.

For our NPDA, we use two states: 0 (start) and 1 (final). We use the input alphabet $\{a, b\}$ and our stack alphabet as $\{X, Y, \$\}$.

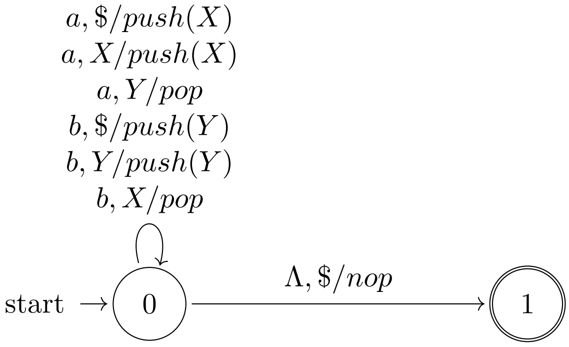


Figure 7.3: NPDA Solution

Form here we can define the 7 instructions for this NPDA in instruction form:

$$
\begin{aligned}
T_1 :& (0, a, \$, push(X), 0) \\
T_2 :& (0, a, X, push(X), 0) \\
T_3 :& (0, a, Y, pop, 0) \\
T_4 :& (0, b, \$, push(Y), 0) \\
T_5 :& (0, b, Y, push(Y), 0) \\
T_6 :& (0, b, X, pop, 0) \\
T_7 :& (0, \Lambda, \$, nop, 1)
\end{aligned}
$$

Alternatively, this can be represented using the Instantaneous Descriptions method:

$$
\begin{aligned}
Start \rightarrow & (0, abbbaa, \$) \\
T_1 \rightarrow & (0, bbbaa, X\$) \\
T_6 \rightarrow & (0, bbaa, \$) \\
T_4 \rightarrow & (0, baa, Y\$) \\
T_5 \rightarrow & (0, aa, YY\$) \\
T_5 \rightarrow & (0, a, Y\$) \\
T_5 \rightarrow & (0, \Lambda, \$) \\
T_6 \rightarrow & (1, \Lambda, \$)
\end{aligned}
$$

7.2 Determinism vs Non-Determinism

In a similar way to that of the finite automata, push-down automata can either be *deterministic* or *non-deterministic*.

A deterministic PDA never has a choice of the next step. It has, at most, one possible output for every combination of state, input and character. This is in a similar way to that of the DFA.

For every combination of state & stack character, only one of the transactions is allowed. This is either for the empty symbol, Λ, or for an input symbol, or there can be no transaction at all.

Example: NPDA vs DPDA

The following instructions cannot be contained by a deterministic push-down automata while they can be contained by a non-deterministic push-down automata.

$$(0, a, \$, push(Y), 0); \quad (0, a, \$, pop, 1)$$

In a different way to that of the DFA and NFA (where both recognise the same languages) - NPDA and DPDA accept different languages. Deterministic push-down finite automata cannot recognise the whole family of context-free languages.

Example: NPDA works where DPDA fails

If we take the language $L = \{ww^R | w \in \{a, b\}^+\}$ which recognises even length palindromes, we can design a PDA which recognises this.

We start with our plan:

- Read in each string and save it to the stack
- At each step, consider the possibility that we might have reached the middle
- Once reaching the midpoint - start working backwards, removing things from the stack if they match what was saved

Except this leaves us with a problem; how do we know we've reached the midpoint? A Non-deterministic PDA can help.

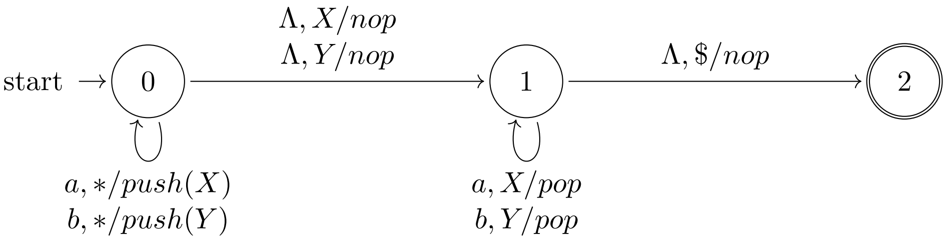


Figure 7.4: NPDA Solution

In the above figure, $*$ stands for $X, Y, \$$.

We can then see the instantaneous description derivation for an example palindrome: *aabbaa*.

$$\begin{aligned}
\text{Start} &\rightarrow (0, aabbaa, \$) \\
\text{Load stack} &\rightarrow (0, abbaa, X\$) \\
\text{Load stack} &\rightarrow (0, bbaa, XX\$) \\
\text{Load stack} &\rightarrow (0, baa, YXX\$) \\
\text{Try: is this the middle?} &\rightarrow (1, baa, YXX\$) \\
\text{Pop stack} &\rightarrow (1, aa, XX\$) \\
\text{Pop stack} &\rightarrow (1, a, X\$) \\
\text{Pop stack} &\rightarrow (1, \Lambda, \$) \\
\text{Done} &\rightarrow (2, \Lambda, \$)
\end{aligned}$$

The above example was an example of a non-deterministic PDA. This is because, from state 0, it branches either loading another letter on or trying to take letters off. This could only be done non-deterministically. A deterministic PDA would need to know when to start removing letters from the stack. Therefore a NPDA can recognise the language of the even palindromes, but a DPDA cannot.

Deterministic push-down automata recognise regular languages and also some which are not regular, but not all of the context free languages.

Page 8

Lecture - A8: Application of context-free grammars

📅 2025-10-20 🕒 15:00 Janka

The idea of a context-free language was first proposed in the mid 1950s by Chomsky. The idea was to describe the grammar of English in terms of their block structure, and recursively built up from smaller phrases. An essential property of these block structures is that the logical units never overlap.

An example, as seen below, shows how a sentence can be comprised of three blocks:

"((The boy touches) (the other boy (with the flower.)))"

Whilst context-free grammars are simple and mathematically precise, the derivations can add some ambiguity. For example, taking the above sentence - there are at least two possible interpretations of it: one boy uses the flower to touch the other boy; or one boy touches the other boy who is holding the flowers. The ambiguity is introduced because the same string can be derived in multiple ways; so to understand the true meaning of the string - we must understand the way in which it was derived.

8.1 Derivations and Parse Trees

Definitions

Parse Tree is a diagrammatic representation of the parsed structure of a sentence or string.

Parse trees can be used to describe a derivation, starting with an initial symbol and working down towards the string.

The start symbol of a set of production rules is the tree's root. The tree is built by adding children to the nodes in the tree; for a production $X \to Y_1 \dots Y_n$ children $Y_1, \dots, Y_n$ are added to the tree. Terminals are added at the leaves, non-terminals at the interior nodes.

Example: Parse Trees

If we consider a grammar fragment for simple arithmetic expressions where E is the only non-terminal:

$$E \to E - E \mid 0 \mid 1 \mid 2 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We can derive strings such as $4 - 8$, $9 - 1$, $5 - 6 - 9 - 2$, etc. We can also assign a meaning (or value, in this case) to the strings: $4 - 8 = -4$, $9 - 1 = 8$, etc.

To understand the meaning of the string, we have to know how it was derived.

If we consider the string $2 - 4 - 6$, we can construct a parse tree

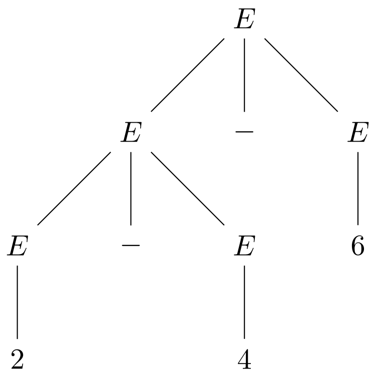


Figure 8.1: Parse Tree 1 for derivation of $2 - 4 - 6$

The above parse tree returns the meaning -8.

Except it is also possible to find a different parse tree which has a different meaning.

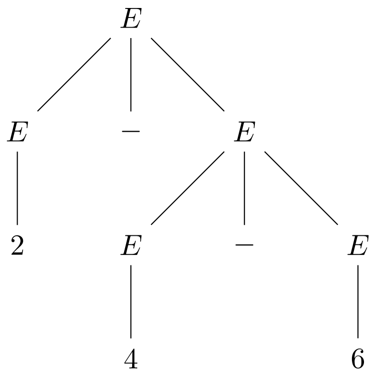


Figure 8.2: Parse Tree 2 for derivation of $2 - 4 - 6$

The above parse tree returns the value $+4$.

Both values are correct.

8.2 Ambiguous Grammars

Definitions

Unambiguous Grammar Where each string has only one parse tree (or equivalently : there is only one left-most (or right-most) derivation for each string). Otherwise the grammar is ambiguous

This can be seen in the grammar below:

$$E \to E - E \mid 0 \mid 1 \mid 2 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Where $\exists$ a string from that language with two different parse trees.

There is no general technique for handling ambiguity; and it is impossible to automatically convert an ambiguous grammar to an unambiguous one. However, in some cases, the grammar can be modified to generate the same language and to remove ambiguity. There are various techniques for this such as using parenthesis.

8.3 Parsing

Parsing is one of the important components of a compiler which is a process that involves checking the input string for whether the input string has the correct syntax; and then constructing a parse tree which captures the internal structure of the string, recording how the input can be derived from the start symbol.

Parsing gives more than the Yes / No answer we can obtain from a NPDA that accepts a given context-free language.

There are two ways in which parsing can be done, both to be explored subsequently.

8.3.1 Top-Down Parsing

Definitions

Top Down Parsing Constructs a derivation by starting with the grammar's start symbol and working towards the string.

Top-Down parsers start at the root of the parse tree and grow towards the leaves. They pick a production and try to match the input. However, if they select a production badly - they may need to backtrack to find a production which works. Some grammars are backtrack free.

If we take the grammar

$$E \to E - E \mid 0 \mid 1 \mid 2 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

and the input string $2 - 4 - 6$. We can see the derivation as follows:

$$\begin{aligned} E \Rightarrow& \underline{E} - E \\ \Rightarrow& \underline{E} - E - E \\ \Rightarrow& 2 - \underline{E} - E \\ \Rightarrow& 2 - 4 - \underline{E} \\ \Rightarrow& 2 - 4 - 6 \end{aligned}$$

We can see the parse tree this produces in Figure 8.2 from an earlier example. However, if after following every logical lead we can't generate the string then the string cannot be parsed; sometimes this can be difficult to decide.

8.3.2 Bottom-Up Parsing

Definitions

Bottom-Up Parsing Constructs a derivation by starting with the string and working backward to the start symbol

Bottom-Up parsers start at the leaves and grow towards the root. As the input is consumed, the possibilities are encoded in an internal state. It starts in a valid state for first legal tokens. Bottom-Up parsers handle a large class of grammars.

If we take the grammar:

$$E \to E - E \mid 0 \mid 1 \mid 2 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

and the input string $2 - 4 - 6$. We can see the derivation as follows:

$$\begin{aligned} 2 - 4 - 6 &\Leftarrow E - 4 - 6 \\ &\Leftarrow E - E - 6 \\ &\Leftarrow E - 6 \\ &\Leftarrow E - E \\ &\Leftarrow E \end{aligned}$$

If all the combinations fail, then the string cannot be parsed.

8.4 CFGs to Describe Programming Languages

The advantage of using context-free grammars for the description of the programming language is that it leads to an easy construction of a parser which can represent the structure of the source program by means of the parse tree. This was one of the first theoretical results of Computer Science to be used in practice.

We want to have efficient parsers for computer languages and to avoid the problems with ambiguous languages. If certain restrictions are placed on the grammar defining a language - efficient stack-based parsing algorithms can be designed.

The best candidates to be used to describe programming languages are languages based on LL(k) grammars or LR(k) for any $k \geq 0$.

LL(k) grammars are based on LL(k) parsers for top-down parsing. LR(k) grammars are based on LR(k) parsers for bottom-up parsing.

8.4.1 LL(k) Grammars

LL(k) means the following:

First L An input string is parsed from left-to-right

Second L Only the left-most derivations of the input string are considered

k is the number of look ahead symbols needed to decide parsing (it is not necessary to know all symbols of the input string before making a decision about derivation)

This means that a LL(1) grammar looks at the current symbol on the input tape to decide which production rule to follow, while a LL(2) would look at the current and next, or a LL(3) would look at the current and 2 next symbols.

Example: LL(1) Derivation

The grammar $S \to aSc \mid b$ with the initial non-terminal A for the language $LL(1) = \{a^n b c^n, n \geq 0\}$ is an example of an LL(1) grammar.

If we consider the input string $aabcc$. We can look at the first input symbol and we know which leftmost derivation has to be used.

Step 1 Starting at the beginning of the string $\underline{a}abcc$, and looking at the current symbol - we know that the production $S \to aSc$ has to be used first.

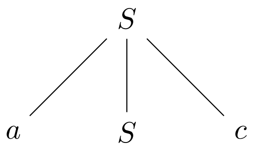


Figure 8.3: LL(1) Derivation Step 1

Step 2 Moving onto the next symbol $a\underline{a}bcc$, we know that we have to use the production $S \to aSc$ for a second time.

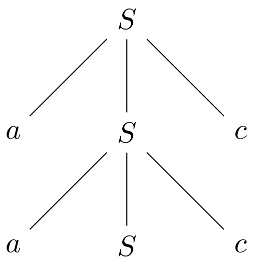


Figure 8.4: LL(1) Derivation Step 2

Step 3 Moving onto the next symbol $aa\underline{b}bcc$, we can see that the only production rule which produces a b is $S \to aSc$ therefore we have to use that one.

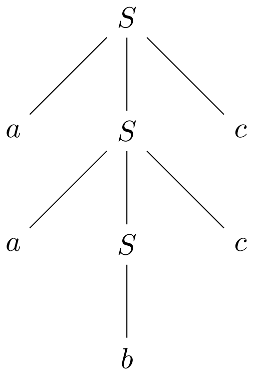


Figure 8.5: LL(1) Derivation Step 3

This is the completed parse tree, and as we've seen - we could select the production to use by looking only at the current symbol.

Example: LL(2) Derivation

We can take the grammar $S \to AB$, $A \to aA \mid a$, $B \to bB \mid c$ with the initial non-terminal S for the language $\{a^m b^n c, m \geq 1, n \geq 0\}$ as an example of a LL(2) grammar.

This is a LL(2) grammar because two productions begin with a and we can't determine which one to use without looking ahead to the next symbol. Looking ahead to the next symbol is enough so this must be a LL(2) grammar.

If we take the string $aabbc$, we can see how the LL(2) derivation works.

Step 1 The derivation must begin with $S \rightarrow AB$.

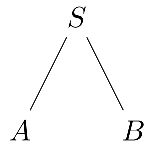


Figure 8.6: LL(2) Derivation Step 1

Step 2 We now take the input string $\underline{a}abbc$ and have a decision to make: use the production $A \rightarrow aA$ or $A \rightarrow a$. To answer this dilemma, we look at the second character of the input string, and as that's an a - we know to use the production which allows for multiple as.

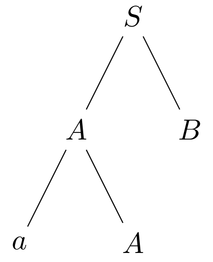


Figure 8.7: LL(2) Derivation Step 2

Step 3 We now take the second input character $a\underline{a}bbc$ and as that is followed by a b - we know it must be produced with $A \rightarrow a$.

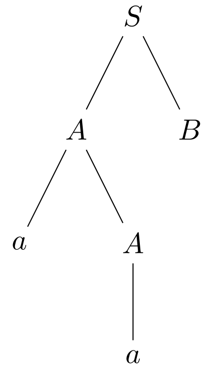


Figure 8.8: LL(2) Derivation Step 3

Step 4 We take the next input character $aa\underline{b}bc$ and as this is followed by another b, we know it must be produced with $B \rightarrow bB$.

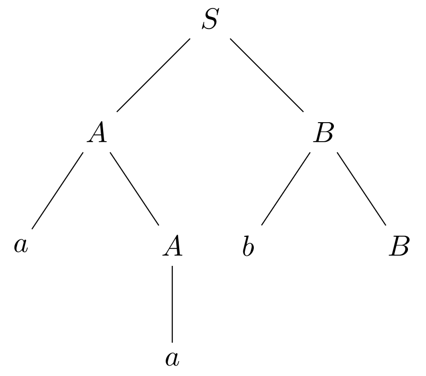


Figure 8.9: LL(2) Derivation Step 4

Step 5 We take the next input character $aab\underline{b}c$ and as this is followed by a c, which we know we can produce from a B, so we know the character must be produced with $B \to bB$.

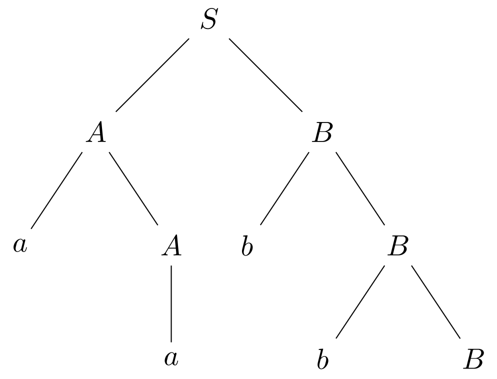


Figure 8.10: LL(2) Derivation Step 5

Step 6 We take the final input character $aabb\underline{c}$. This isn't followed by anything so we find the production rule which produces this $B \to c$.

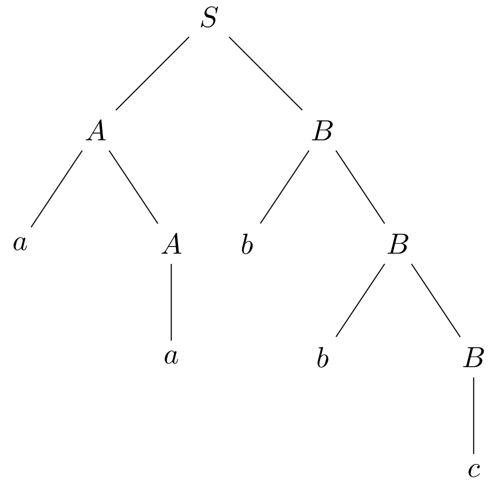


Figure 8.11: LL(2) Derivation Step 6

There are, of course, grammars which don't conform to the LL(k) structure. For example the language

$L = \{ba^n, n \geq 0\}$ where the production rules are of the form $S \rightarrow Sa \mid b$ is an example of a non-LL(k) grammar. We can show this with the example string $baaaa$. We cannot know precisely how b was derived without knowing the length of the full string; it is impossible to do a partial derivation knowing only how the string starts.

8.4.2 LR(k) Grammars

The languages based on LR(k) grammars can be parsed bottom up.

The L in LR(k) means that we scan the string from left to right, however the R means we produce a right-most derivation (R) in reverse. This involves reversing the productions and more more difficult to see through.

LR(1) languages are exactly LR(k) languages which are exactly deterministic context-free languages (DCFL).

Deterministic context-free grammars (DCFG) are a proper subset of the context-free grammars - they can be recognises by the DPDA. DCFGs are always unambiguous.DCFGs are of great practical interest - they can be parsed in linear time and in fact a parser can be automatically generated from the grammar by a parser generator.

Example: LR(0) Grammars

If we consider the language $L = \{ab^{2n+1}c | n \geq 0\}$ with the grammar $S \rightarrow aAc$ and $A \rightarrow Abb \mid b$. We can take the input string $abbbbbc$

To parse this, we we would start at the bottom and work up. We look for the right hand side of the a production in a the string. The first one we find is b, which we replace with A:

$$a\underline{b}bbbbc \Leftarrow a\underline{A}bbbbc$$

We repeat the process with this new string, and the first one we find is Abb:

$$a\underline{Abb}bbc \Leftarrow a\underline{A}bbc$$

We repeat the process on the new string, we again find Abb:

$$a\underline{Abb}c \Leftarrow a\underline{A}c$$

Then finally we see aAc which we can produce S from:

$$a\underline{A}c \Leftarrow \underline{S}$$

We can then invert this to find the full derivation from Start Symbol to string of terminals

$$S \Rightarrow aSc \Rightarrow aAbbc \Rightarrow aAbbbbc \Rightarrow abbbbbc$$

Page 9

Lecture - A9: Turing Machines

2025-11-03 14:00 Janka

There are a few slides onn the history of Alan Turing available in the slides on Moodle.

Turing Machines are are more powerful than both finite automata or pushdown automata - they are as powerful as any computer as we have ever built, except modern computers are just faster.

The main improvement of a Turing Machine over a pushdown automata is that the TM has infinite accessible memory which can be written to and read from; and that the read/write head can move to the left and to the right on the input tape, or not change position at all.

The TM works on a tape divided into cells which is infinite in both directions from the read/write head. Each of the cells contain either a symbol from an alphabet or the blank symbol (□). There are only a finite number of non-blank symbols written on the tape.

The Turing Machine is always in one of a finite number of states. The read/write head reads the symbol in the current cell. Depending on the symbol on the tape and the current state, the Turing Machine will do one of a number of things:

- change the state
- move the head (see below)
- re-write the current symbol, or leave it unchanged

By default, we are talking about deterministic Turing Machines.

As we see above - the head of the Turing Machine moves. There are three possible movements that the head can undertake, note the bracketed identifier for each option:

- Moves to the left by once cell from the current cell (L)
- stay at the current cell (S)
- Moves to the right by once cell (R)

9.1 Formalising TM Instructions

We can see that a Turing Machine instruction takes a familiar yet unfamiliar form of

$$T : Q \times \Gamma \to \Gamma \times Q \times \{L, R, S\}$$

which is similar to that of the Transition Function seen in our pushdown automata of lectures gone past, but different in that there are more elements!

Every Turing Machine contains five parts:

- Input: The current machines state (from Q)
- Input: The tape symbol read from the current tape cell, which can be blank symbol (from Γ)
- Output: A tape symbol to write to the current tape cell, which may be the blank symbol or other symbols (from Γ)

- Output: The next machine state (from Q)
- A direction for the tape head to move in (from $\{L, R, S\}$)

It's important to note that Γ (the tape alphabet) contains Σ (the alphabet of the language), $\square$ (the empty tape symbol), and also other symbols as needed.

So from this we can see that the instructions have two inputs and three outputs:

$$T(i, a) = (b, j, L)$$

Or alternatively represented as

$$(i, a, b, L, j)$$

Where the current stat of the machine is i, and the symbol in the current tape is a. Then we can write b into the current tape cell, move left by one cell and go to state j.

Finally we can see this in the below transition diagram

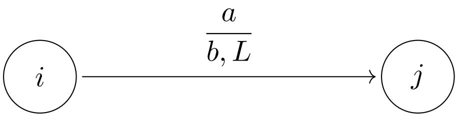


Figure 9.1: Graphical form of a TM Instruction

9.2 The Input Tape

The input string is represented on the tape by placing the letters of the string (which have come from Σ) into adjacent cells. All other cells of the tape contain the blank symbol. The head is usually positioned over the leftmost cell of the input string, which is the leftmost non-blank tape symbol.

It's important when we're reading from and writing to the input tape that we don't write the blank symbol into the middle of the string, as this gets confusing.

9.3 The Halt State

Much like our the automaton we've become friendly with so far in this module - Turing Machines have to start in a the one start state, they also have to stop somewhere. Unlike FAs or PDAs, Turing Machines must have only one final state - which is called the *halt state*.

A Turing Machine halts when it either: enters the halt state; or when it enters a state for which there is no valid move.

9.4 Turing Machines Recognise Languages

A Turing Machine, T, recognises a string x (over Σ) if and only if when:

1. T starts in the initial position and x is written on the tape
2. T halts in a final state

T is said to recognise a language A if x is recognised by T if and only if x belongs to A. Except, while running - a TM can also read/write other symbols from/onto the tape which are not necessarily from the alphabet A.

A TM doesn't recognise a string if it doesn't halt ever, or halts in a non-final state.

9.5 Instantaneous Descriptions

To describe a Turing Machine at any given time, we need to know three things:

- What is on the tape?
- Where is the tape head?
- What state is the control in?

We can then represent this information as follows:

$$\text{State } i : \square\ a\ a\ \underline{b}\ a\ b\ \square$$

Where the underlined symbol represents the current position of the tape head.

As will be seen in the following examples, when doing questions relating to Turing Machines - show the plan for the operation of the TM in plain pseudocode (almost). This is so if you botch the syntax of the algorithm then Janka can see some insight into the logic (or lack thereof) and possibly award some marks for workings.

Example: TM to recognise a^*

If we take $\Sigma = \{a, b\}$ we can design a Turing Machine that accepts the language denoted by the regular expression a^*.

The plan:

- Starting at the left end of the input, we read each symbol and check that it is an a
- If it is - we continue moving right
- If we reach a blank symbol without encountering anything but a, we terminate and accept the string
- If the input contains a b anywhere (meaning the string is not in $L(a^*)$), we halt in a non-final state

This is nice and easy, there are two transitions:

$$T(0, a) = (0, a, R),$$
$$T(0, \square) = (1, \square, R)$$

Which we can represent with a pretty picture. Finally we can see this in the below transition diagram

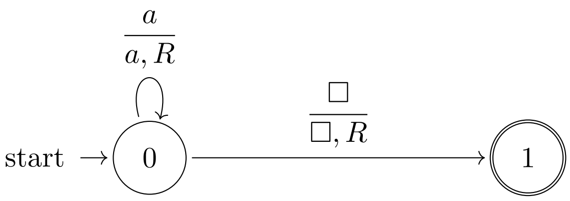


Figure 9.2: TM designed to accept a^*

Example: TM for $\{a^n b^n c^n, n \geq 0\}$

Now for a slightly more complicated example.

We can see that the language $\{a^n b^n c^n, n \geq 0\}$ is not context-free as it cannot be recognised by a PDA, however it can be recognised by a Turing Machine.

The plan:

- If the current cell is empty, then accept and halt
- If the current cell contains a
  - then replace it by X and scan right, looking for the corresponding b to the right of any as, replace this by Y
  - then continue scanning to the right, looking for a corresponding c to the right of any bs, replace this by Z
- Now scan left to the X and see whether there is an a to its right:
  - if so - start the process again
  - if there are no As, then scan right to make sure there are no bs and cs.

We can now design the algorithm

If $\square$ is found then halt.

$(0, \square, \square, S, Halt)$	Success

If a is found, then write X and scan right

$(0, a, X, R, 1)$	Replace a by X and scan right

Scan right, looking for b. If found replace by Y

$(1, a, a, R, 1)$	Scan right
$(1, Y, Y, R, 1)$	Scan right
$(1, b, Y, R, 2)$	Replace b by Y and scan right

Scan right, looking for c. If found replace by Z

$(2, b, b, R, 2)$	Scan right
$(2, Z, Z, R, 2)$	Scan right
$(2, c, Z, R, 3)$	Replace c by Z and scan left

Scan left, looking for X. Then move right and repeat the process

$(3, a, a, L, 3)$	Scan left
$(3, b, b, L, 3)$	Scan left
$(3, Y, Y, L, 3)$	Scan left
$(3, Z, Z, L, 3)$	Scan left
$(3, X, X, R, 0)$	Found X. Move right one cell

Now being back in the state 0, we have to scan over Ys and Zs to find the right end of the string

$(0, Y, Y, R, 4)$	Scan right

Scan right over Ys and Zs looking for □

$(4, Y, Y, R, 4)$	Scan right
$(4, Z, Z, R, 4)$	Scan right
$(4, \square, \square, S, Halt)$	Success

As we can see from the above, constructing Turing Machines to compute relatively simple tasks can be extremely lengthy.

9.6 Computing Problems with Turing Machines

Turing Machines can recognise a set of strings (language) similarly to the finite automata and push-down automata.

Turing Machines can also do more - they can read an input, perform transformations of the string on the tape and write down a result on the tape. When performing in this mode, they are known as *transducers*.

The next lecture (over the page) will show how Turing Machines can solve mathematical problems like sums, products, etc.

Page 10

Lecture - A10: Computing with TMs and Alt. Definitions

2025-11-03 15:00 Janka

Caffeine break later, we're back at them friendly lil' Turing Machines...

10.1 Computing Functions

For a given input string, x, we can construct a TM which will do some computation following its instructions. This gives the output of a string y on the tape of the TM when it halts.

From this we can see that we can define a *partial* function $T(x) = y$ for all strings x for which the TM halts. This means that the TM can compute values of functions on strings.

This is great for strings, but what about integers? We represent non-negative integers in other ways, for example n by a string of $n + 1$ (or n) 1s or in binary format.

Example: TM for adding 2

If we take a natural number n to be our input, we can compute $n + 2$ as our output.

For the purposes of this example, we'll represent natural numbers in unary form (for example 3=111, or 5=11111) and 0 will be represented by the empty symbol (□).

The plan:

- Move tape head to the left of the first 1 (if it exists)
- Change that empty cell to a 1
- Move left and repeat
- Halt

We'll need 3 states: 0 (initial), 1 and Halt as well as three instructions to make this happen.

$(1) : (0, 1, 1, L, 0)$	move left to blank cell
$(2) : (0, \square, 1, L, 1)$	Write 1 into cell and move left
$(3) : (1, \square, 1, S, Halt)$	Write 1 into cell and halt

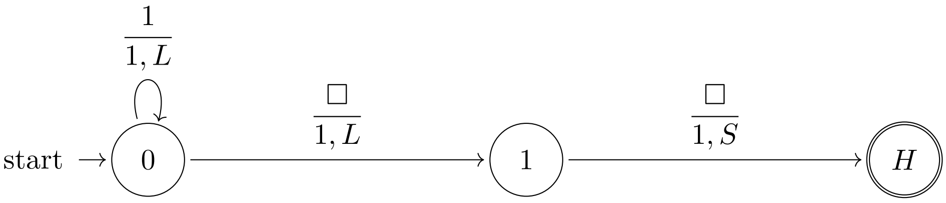


Figure 10.1: TM designed for adding 2

We can now see the instantaneous description for adding 2 to a given input string:

State 0:	$\square\ \underline{1}\ 1\ 1\ \square$	begin in state 0
State 0:	$\underline{\square}\ 1\ 1\ 1\ \square$	(1)
State 1:	$\underline{\square}\ 1\ 1\ 1\ 1\ \square$	(2)
Halt:	$\square\ \underline{1}\ 1\ 1\ 1\ 1\ \square$	(3)

There is a second example of a Turing Machine for Computing Functions, this time for adding 1 in binary available in the Lecture A10 slides on Moodle.

10.2 Non-deterministic Turing Machines

In a similar to PDA (an sort of similar to that of NFA), a TM's non-determinism presents through having multiple possible outputs for one input configuration. The non-deterministic TM is like the TM but with finite number of choices of moves and may have more than 1 move with the same input state & symbol. The non-deterministic TM accepts the input w if there is at least one computation that halts normally for the input w.

Non-determinism is more powerful than determinism for pushdown automata. However it makes no different for finite automata. In a similar way, deterministic and non-deterministic Turing Machines posses the same power. We can therefore derive the theorem: if a non-deterministic Turing Machine accepts a language, L, then there is a deterministic Turing Machine that also accepts L.

Both deterministic and non-deterministic Turing Machines accept ths same family of languages: recursive enumerable languages generated by the unrestricted grammars.

10.3 Alternative Definitions

There are a number of other similar machines which might look more or less powerful than a Turing Machine but can be seen to be equivalent in power to the simple Turing Machine. This can be achieved by adding more tapes, more control units, etc.

We show the equivalence of these Turing Machine'nts and Turing Machines by describing how a simple one-tape Turing Machine can be used to emulate them and vice-versa.

10.3.1 Two Stack PDA

This variation has two stacks in lieu of an input tape, both still connected to the Control Node. The right half of the tape is kept on one stack, while the left half kept on the other. As we move along the tape we pop characters off one stack and push them onto the other.

10.3.2 Semi-Infinite Tape

In this variation, we are presented with an infinite tape which is only infinite in one direction. We can emulate the standard TM by splitting the cells into two groups, alternating as we go down the tape. One group represents the left half of the infinite tape and the other group represents the right half.

10.3.3 Multi-Track

A multi-track Turing Machine has k tracks on the same tape which are read and written at the same time. A multi-track TM can simulate a standard TM when all bar the first track are ignored. A standard TM (with Σ') can simulate a multi-track TM (with Σ) if we map every order pair $[x_1, x_2, \ldots, x_k]$ of symbols from Σ to a unique symbol in Σ' where $|\Sigma'| = |\Sigma|^k$.

10.3.4 Multi-Tape

In the multi-tape Turing machine, there are k tapes with k tape heads each moving independently. This is a generalisation of the multi-track Turing Machine. Each TM has its own read-write head but the state is common for all.

In each step, transition, the TM reads the symbols scanned by all heads. Then depending on the read symbols and the current state - each head writes, moves R or L and the control unit enters a new state. Actions of the heads are independent of each other.

Example: Multiplying two numbers with Multi-Tape Turing Machine

If we take the example that we want to multiply two numbers, each of which represented as a unary string of ones, to get a third number. This would be difficult to do with a simple Turing machine but is fairly straight forward with a three-tape machine.

We can use a three-tape Turing Machine, each with a specific data item to represent:

- Tape 1: the first number in the multiplication (i.e. 3)
- Tape 2: the second number in the multiplication (i.e 4)
- Tape 3: the output

We start by checking whether either number is zero

$(0, (\square, \square, \square), (\square, \square, \square), (S, S, S), Halt)$	Both are zero
$(0, (\square, 1, \square), (\square, 1, \square), (S, S, S), Halt)$	First is zero
$(0, (1, \square, \square), (1, \square, \square), (S, S, S), Halt)$	Second is zero
$(0, (1, 1, \square), (1, 1, \square), (S, S, S), 1)$	Both are non-zero

Add the number on the second tape to the third tape

$(1, (1, 1, \square), (1, 1, 1), (S, R, R), 1)$	Copy
$(1, (1, \square, \square), (1, \square, \square), (S, L, S), 2)$	Done Copying

Move the tape head of the second tape back to the left end of the number; move the tape head of the first number one cell to the right

$(2, (1, 1, \square), (1, 1, \square), (S, L, S), 2)$	Move to the left end
$(2, (1, \square, \square), (1, \square, \square), (R, R, S), 3)$	Both types to the right one cell

Check the first tape head to see if all the additions have been performed

$(3, (\square, 1, \square), (\square, 1, \square), (S, S, L), Halt)$	Done
$(3, (1, 1, \square), (1, 1, \square), (S, S, S), 1)$	Do another add

Every Multi-Tape Turing Machine has an equivalent single tape TM. If M has k tapes, M' simulates the effect of k tapes by storing the information on its single tape. It uses a new symbol # as a delimiter to separate the contents of the different tapes (marks the left and the right portions of the tape). M' also must keep track of the locations of the heads on each tape. It writes a tape symbol with dot above it to mark where the head on that tape would be. Dotted symbols are simply new symbols added to the tape alphabet.

If the movement of one T's tape head causes M's tape head to bump into either □ or # then that side of the tape must be moved to make room for a new cell.

10.3.5 Multi-Head

The Multi-Head Turing Machine has one tape with many tape heads moving independently. Two heads are usually better than one, but not in the case of the TM as only one can be active at a given time. A particular head is associated with each state - which is part of the instructions.

10.3.6 Off-Line Turing Machine

This is a Turing Machine with two tapes, where one tape is a read-only version of the input.

10.3.7 Multi-Dimensional Tape

This is a Turing Machine which has one tape where the tape may extend in two more more dimensions.

10.4 Linear Bounded Automata

The final piece of the puzzle...

The Linear Bounded Automaton (LBA) is a Turing machine which can only use a tape which is the size of the initial input, ie it cannot use any more tape than the size of the initial input. This type of machine recognises exactly the family of context-free languages. We won't cover these in any more detail.

So excluding the languages which can't be described by a grammar - we've done it. We've identified a type of machine which can recognise each of our types of language as prescribed in the Chomsky Hierarchy.

Language Family	Grammars	Recognition Machine
Regular Languages	Regular Grammar	Deterministic Finite Automata (DFA) & Non-deterministic Finite Automata (NFA)
Context-Free Languages	Context Free Grammars	Non-deterministic Pushdown Automata (NPDA)
Context Sensitive Languages	Context Sensitive Grammars	Linear Bounded Automata (LBA)
Recursive Enumerable Languages	Unrestricted Grammars	Turing Machines (TM)

Table 10.1: Chomsky Languages and Recognition Machines

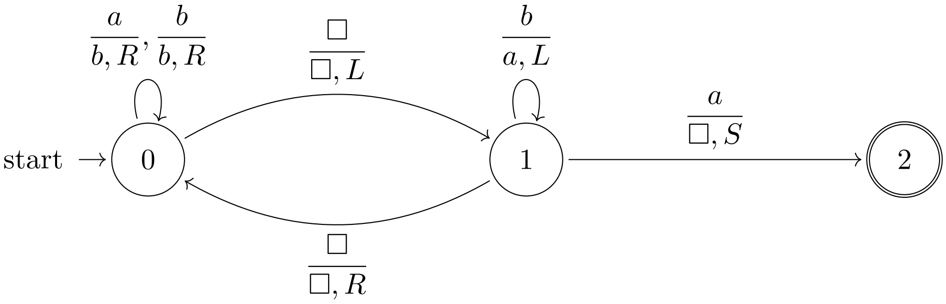
Page 11

Lecture - A11: More about Turing Machines

2025-11-10 14:00 Janka

11.1 Some Turing Machines Don't Halt

It is possible to design a Turing Machine which doesn't halt. For example the following TM takes an input from $\Sigma = \{a, b\}$ and scans on the input tape turning a to b and leaving b as b. Then when it reaches the end of the input, it scans left over the input turning b into a. Then it returns to state 0 and starts again. It will never hit the transition function $T(1, a) = (\square, 2, S)$ because when in state 1, the input tape will never be in the condition where a is on the input tape. What a silly Turing Machine...



$\frac{a}{b,R}, \frac{b}{b,R}$
$\frac{\square}{\square,L}$
$\frac{b}{a,L}$
$\frac{a}{\square,S}$
start
0
1
2
$\frac{\square}{\square,R}$

Figure 11.1: Silly Turing Machine which never halts

If we let L be the language accepted by a TM, then:

- If the input string is in the language L, then the machine must halt in a finite number of steps
- For a string that is not in the language L the TM can finish in a non-halt state or get stuck in a loop / go on forever

The key difference here is that it is possible for a TM to finish in a non-halt state, or not halt at all. Both of these conditions would mean that the TM rejects the input string.

11.1.1 Recursive and Recursive Enumerable Languages

Definitions

Recursive Language A language, L, is recursive (decidable) if L is the set of strings accepted by some TM that halts on every input

Recursively Enumerable A language, L, is recursively enumerable if L is the set of strings accepted by some TM

If L is a recursive language then:

- if $w \in L$ then a TM halts in a final state
- if $w \notin L$ then TM halts in a non-final state

If L is a recursive enumerable language then:

- if $w \in L$ then a TM halts in a final state
- if $w \notin L$ then a TM halts in a non-final state or loops forever

So from this we can see that every recursive language is also recursive enumerable, but not the other way around.

In other words:

- Recursive Language - a language in which the TM halts, either in an accepting state or non-accepting state for any given input
- Recursive Enumerable Language - a language in which the TM does not halt for the given input, either in an accepting or non-accepting state.

11.2 Models of Computing: The Final Version

As can be seen in the following list - the most restrictive languages are *regular languages* with the least restrictive being *languages without grammar.* This final type are uncomputable, meaning there is not a way for them to be programmatically recognised (ie through FA, PDA, TM, etc).

List of models of computing from least restrictive to most restrictive including what machine can recognise them:

- Languages without grammar (uncomputable)
- Recursive Enumerable language (Nondeterministic Turing Machine (NTM) / Turing Machine (TM))
- Recursive Language (Nondeterministic Turing Machine (NTM) / Turing Machine (TM) that halts for every input)
- Context-Sensitive Languages (Linear Bound Automaton(LBA))
- Non-deterministic context-free languages (Non-deterministic Push-down Automata (NPDA))
- Deterministic context-free language (Deterministic Push-down Automata (DPDA))
- Regular Language (Non-deterministic Finite Automata (NFA) / Deterministic Finite Automata (DFA))

11.3 Universal Turing Machines

We can construct a Turing Machine which can do the job of any other Turing Machine. So not truly universal, but as close as we can get...

We take an arbitrary Turing machine, M, and an input w, the Universal Turing Machine, U will simulate the operations of M on w. It returns the output from M for w. U must halt for a given input if and only if M halts. If M accepts, then U must accept. If M rejects, then U must reject. This is known as the *Universal Turing Machine.*

We can imagine the UTM as a TM with three tapes:

Tape 1 will correspond to M's tape

Tape 2 will contain M's program which is the program that U is running

Tape 3 will contain the encoding of the state that M is in at any point during the simulation

A UTM acts as a completely general purpose computer, effectively storing a program and data on the tape (1 and 2 respectively) and then executing the program.

11.4 Busy Beavers

If we consider a deterministic Turing Machine that:

- can write either $\square$ or 1 on a tape cell and
- must shift left or right after each move

The Busy Beaver problem is about finding a Turing Machine which will run and find the longest finite sequence of 1s with a given number of TM states. This number of states is given without the halt state included. For example, if we take a 1 state Turing Machine, it can write 1 1 on the tape before reaching the halt state, either using $(0, \square, 1, R, 0)$ or $(0, \square, 1, L, 0)$.

We can formally define the problem as: Let $b(n)$ denote the number of 1s that can be written by a busy beaver with n states, not including the halt state. The critical component of the busy beaver is that it must not loop infinitely, rather it must always reach the halt state.

This is a deceptively simple problem. Rado in 1962 discovered that $b(2) = 4$, meaning for 2 states - the longest sequence of 1s able to be written to the tape is 4. Rado then went on to discover that b cannot be computed by any TM, so the busy beaver problem is now described as a *noncomputable function.*

Subsequently, it has been calculated that $b(3) = 6$ and $b(4) = 13$. Marxen and Buntrock in 1989 found $b(5) \geq 4098$ using 47,176,870 steps; and Kropitz in 2010 found that $b(6) \geq 10^{18267}$ which used over 10^{36534} steps.

And that's it. Part A of this module has reached it's halt state (ha). Part B will commence next Monday, the Part A exam is next Wednesday.

Page 12

Lecture - B1: Computability and Equivalent Models Pt. 1

2025-11-17 14:00 Janka

We are now halfway through this module, exam on Wednesday to celebrate this. Happy international day of students, although no time to celebrate until after the exam on Wednesday - must spend all time revising for Wednesday, Turing Machines dancing in sleep and all...

Today's lecture will begin the "B" part of the module - this will be examined in the exam in the January Exam period, and is weighted 50% (same as the exam on Wednesday for Part A topics).

> In the slides on Moodle - there is some history about what will be covered in this half of the module, including a reference to a video on Bob National.

12.1 Computability

Something is *computable* if there is some computation that computes it, or if it can be described by an algorithm. We can take this to mean that a computation is an execution of an algorithm.

The "computable" property has something to do with a formal process (execution) and a formal description (algorithm). For example:

- the derivation process associated with grammars
- the evaluation process associated with functions
- the state transition process associated with machines
- the execution process associated with programs and programming languages

A *model* is a formalization of an idea. This means we have several ways to model the idea of computability.

We know that one computational model is more powerful than other computational models. For example, using what we've learnt in Part A, the Turing Machine is more powerful than a pushdown automata. However there are also models with the same power, for example non-deterministic and deterministic finite automata.

There is a most powerful model. There are many models equivalent to that. Therefore, as proven by the Church-Turing Thesis, anything that is intuitively computable can be computed by a Turing machine.

This is called a 'thesis' rather than a 'theorem' as we only have an informal idea of what being computable is, as mathematicians have not yet been able to define this. However, we have been able to define the Turing Machine and this is a formal and well-understood concept.

As yet, there hasn't been a computational model invented which is more powerful than the Turing Machine! There are, however, several alternative formalisations for the notion of computability which are equivalent to the Turing Machine. These will be discussed in this lecture and the subsequent lecture.

12.2 Equivalence of Computational Models

In the early 20th century, there were a number of attempts to formalise the notion of computability. The American mathematician Alonzo Church created a method for defining functions called the λ-calculus. Later, British mathematician Alan Turing created a theoretical model for a machine that could carry out calculations from inputs. Church, along with mathematician Stephen Kleene and a logician J.B. Rosser created a formal definition of a class of functions whose values could be calculated by recursion (these are known as the *partial recursive functions*, we'll see more of these later).

Even through computational models may process different kinds of data, they can still be compared with respect to how they process natural numbers (represented as $\mathbb{N}$).

While exploring the computation models in the subsequent sections, we'll make the assumption that there is an unlimited amount of memory available. This means that we can represent any natural number or any finite string.

12.3 Model 1: λ-calculus

The λ-calculus was introduced by Church in the 1930s as part of an investigation into the foundations of mathematics.

Church's original system was shown to be logically inconsistent - so he later introduced two weaker systems:

- untyped lmbda calculus
- lambda calculus

Both types of calculus played an important role in the theoretical side of development of programming languages, with untyped lambda calculus being the original inspiration for functional programming, especially Lisp.

12.4 Model 2: Simple Programming Language

Stepherdson and Sturgis introduced a *simple programming language* in 1963. It has the same power as a Turing Machine which means:

- any program that can be solved by a Turing Machine can be solved with a simple program
- any problem that can be solved by a simple program can be solved by a Turing machine

The Simple Programming Language is, as the name suggests, simple. All the variables within the Simple Programming Language are values from the set $\mathbb{N}$ of natural numbers. There exists a while statement of the form:

$$\text{while X} \neq 0 \text{ do } \textit{statement} \text{ od}$$

There is an assignment statement taking one of the three forms:

$$X := 0, \quad X := succ(Y), \quad X := pred(Y)$$

From this we can see that a statement is always one of the following:

- a while statement
- an assignment statement
- a sequence of two or more statements separated by semicolons

A simple program is a statement.

As we saw earlier, all values used within the simple programming language are form ℕ. This means that it's not possible to go below zero in our operations, therefore $pred(0) = 0$.

All variables in a simple programming language program have been given initial values and the output consists of the collection of values at program termination.

Example: Simple Programming Language Snippets

Ex. 1: Code for the macro statement $X := Y$

$$X := succ(Y);\ X := pred(X)$$

Ex. 2: Code for the macro statement $X := 3$

$$X := 0;\ X := succ(X);\ X := succ(X);\ X := succ(X)$$

More examples of the Simple Programming language available in the Tutorial.

12.5 Model 3: Markov Algorithms

In 1954, Markov designed an approach to computation which is equivalent in power to Turing Machines.

A Markov algorithm over an alphabet Σ is a finite ordered sequence of productions $x \rightarrow y$, where $x, y \in \Sigma^*$. Some productions may be labelled with the word "halt" although this is not a requirement. If there is a production $x \rightarrow y$ such that x occurs as a substring of w, then the leftmost occurrence of x in w is replaced by y. A Markov algorithm transforms an input string into an output string, we can see it computes a function from Σ^* to Σ^*.

The Markov algorithm works as follows, when given an input string:

1. Check the productions in order from top to bottom to se whether any of the patterns can be found in the input string
2. If none are found - the algorithm stops
3. If one (or more) is found, use the first (topmost) of them to replace the leftmost matching text in the input string with it's replacement
4. If the applied production was a terminating one - the algorithm stops
5. Return to step 1 and carry on

When processing string with the Markov Algorithm - we assume $w = \Lambda w$. Which in non-mathematical speak translates to every string (w) beginning with the empty string character (Λ). We can then see that a production of the form $\Lambda \rightarrow y$ would transform w to yw.

Example: Markov Algorithm Execution

If we take M as the Markov algorithm over $\{a, b\}$ consisting ovf the following sequence of three productions:

1. $aba \rightarrow b$
2. $ba \rightarrow b$

3. $b \to \Lambda$

We can trace the execution of M for the string $w = aabaaa$

$$\begin{aligned}
w = aabaaa &\to abaa && (1)\\
&\to ba && (1)\\
&\to b && (2)\\
&\to \Lambda && (3)
\end{aligned}$$

From this we can see that this algorithm returns Λ for all strings of the form a^iba^j where $i \leq j$; and a^{j-i} for all strings of the form a^iba^j where $i > j$.

12.6 Model 4: Post Algorithms

Emil *Post* developed the *Post* algorithm in 1943 which is another string processing model with equivalent power to the Turing machines.

A post algorithm over an alphabet, Σ, is the set of productions that are used to transform strings. The productions have the form $s \to t$, where s and t are strings made up of symbols form Σ and possibly some variables. If a variable X occurs in t then X occurs in s. Some productions may be labelled with the word “halt”, although this is not required.

Given an input string, $w \in \Sigma^*$, the Post algorithm works as follows:

1. Find a production $x \to y$ such that w matches x.
2. If so, use the match and y to construct a new string. Otherwise - halt.
3. If the $x \to y$ is labelled with “halt”, then halt.
4. Otherwise, return to step 1 and carry on.

Post algorithms can be deterministic or non-deterministic and variables may also match Λ.

Example: Post Algorithm Execution

If we consider the following single production over the alphabet $\{a, b\}$:

$$aXb \to X$$

If we take the start string aab, we can see how the string is processed. First the string aab is matched with aXb, meaning $X = a$. Therefore the string aab is transformed into the string a. Now that a doesn’t match the left hand side of the production - the computation halts.

Note that X can match the empty string too and in that way, ab can be transformed to Λ.

This Post algorithm does many things, for example it transforms the string of form a^ib to a^{i-1} for $i > 0$.

Page 13

Lecture - B2: Computability and Equivalent Models P2. 2

2025-11-17 15:00 Janka

10 minute doom-scroll-of-cat-photos break later, we're back at it for Part 2 of this 2 parter.

This lecture picks up directly from the last - starting with...

13.1 Model 5: Recursive Functions

As we have seen in the second year Discrete Maths module - the concept of a *function* is fundamental to much of mathematics.

In this topic, we consider functions whose arguments and values are natural numbers. The basic characteristic of a *computable function* is that there must be a finite procedure (known as an algorithm) telling the 'computer' how to compute the function. Expressions can be one way to define this, such as $f(n) = n^3 + 1$ as a clear recipe is given; however for more vague functions (such as the busy beaver), there isn't a definitive recipe for them.

Church, Kleene & Rosser created a formal definition of a class of functions whose values could be calculated by recursion, which are known as the *partial recursive functions.*

We believe that functions of the following form are computable:

$$f(x) = 0, \quad g(x) = x + 1, \quad H(x, y, z) = x$$

In some fun news - all we need to construct all possible computational functions are basic functions like those above, and some simple combining rules! Unlike for Turing Machines, the description of recursive functions is *inductive.*

13.1.1 The Idea...

The idea of recursive functions is simple - we break down the target into a series of steps which the computer is able to compute.

If we consider the function $exp(x, y)$ which finds the y exponent of x (x^y), we know that

$$\begin{aligned} x^0 &= 1 \\ x^1 &= x \\ x^2 &= x \cdot x \\ x^3 &= x \cdot x \cdot x \end{aligned}$$

We can see a sort of generalisation forming here:

$$\begin{aligned} x^y &= x \cdot x \cdots x && (y \text{ occurrences of } x) \\ x^{y+1} &= x \cdot x \cdots x && (y + 1 \text{ occurrences of } x) \\ &= x \cdot x^y \end{aligned}$$

From this we can see two “rewriting rules” emerge which are enough to define the *exp* function:

$$x^0 = 1$$
$$x^{y+1} x \cdot x^y$$

Well this is most exciting - we can see that the rules for *exp* reduce exponentiation to multiplication.

If we now consider multiplication:

$$x \cdot 0 = 0$$
$$x \cdot (y + 1) = x + x \cdot y$$

We can see that rules for multiplication reduce the multiplication operation to additions. I spy a pattern here...

We can finally take a look at the rules for addition:

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

Here we find the simplest, most primitive operation we know about: *succ*, which adds 1.

From this exercise of inductive decomposition, we’ve found that primitive recursion is in the spirit of “computation by rewriting” definitions of *exp*, $\cdot$ and $+$. If we want to use “recursion”, it consists of one rule for $y = 0$ and one rule for $y > 0$ where y acts as a countdown for the number of remaining steps in the computation.

There are five building blocks for primitive recursive functions:

- Three basic functions: *successor*, *zero* and *projections*
- Two ways of building new primitive functions from old ones: *composition* and *primitive recursion*

13.1.2 Recursive Function Building Blocks

The *successor* function we met in the last lecture as part of the simple programming language. Successor takes a value and returns the successor number to it: $succ(x) = x + 1$.

The *zero* function is a simple function. It takes a natural number and returns 0: $zero(x) = 0$.

The *projection* function takes a natural number as input, i and returns the i-th element of the provided tuple of natural numbers: $project_i : \mathbb{N}^k \to \mathbb{N}$, where

$$project_i(x_1, \ldots, x_k) = x_i, i \in \{1, \ldots, k\}$$

Composition replaces the arguments of a function with other functions. If we take g_1, g_2, ..., g_m as functions $\mathbb{N}^k \to \mathbb{N}$, and f is a function $\mathbb{N}^m \to \mathbb{N}$, then the function $h : \mathbb{N}^k \to \mathbb{N}$ is given by:

$$h(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$$

Then the function h is said to arise by composition from f, g_1, ..., g_m.

Example: Composition

If we consider the function:

$$h(x) = succ(succ(zero(x)))$$

We can see that this function will always return 2 and is made of three functions: *succ* twice, and *zero* once.

Primitive Recursion is where a new function is defined in terms of existing functions as follows:

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= h(x_1, x_2, \ldots, x_n) \\ f(x_1, x_2, \ldots, x_n, succ(y)) &= g(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)) \end{aligned}$$

The first function above uses $y = 0$ and as we saw above in our inductive experiments - this is defined in a different way as this is often directly computable. The second function then increases y to the point needed for the equation - the right hand side using y as is and the left hand side using $succ(y)$ to increase it's value by 1. (We'll see an example of this in action shortly).

A function is *primitive recursive* if it can be built up using the base functions and the operations of composition and primitive recursion.

Example: Addition as a Primitive Recursive Function

If we take the function $add(x, y) = x + y$ we can see it's primitive recursiveness.

Firstly, we define addition recursively:

$$\begin{aligned} add(x, 0) &= x \\ add(x, succ(y)) &= succ(add(x, y)) \end{aligned}$$

Where the first row is the zero-condition which the computer will always be able to compute, and the second row is the recursive statement which we can use to sum two numbers.

$$\begin{aligned} add(2, 2) &= succ(add(2, 1)) \\ &= succ(succ(add(2, 0))) \\ &= succ(succ(2)) \\ &= succ(3) \\ &= 4 \end{aligned}$$

We can see that rewriting the right hand side of the recursive addition definition to use primitive recursive functions:

$$\begin{aligned} f(x, 0) &= h(x) \\ f(x, succ(y)) &= g(x, y, f(x, y)) \end{aligned}$$

where:

$$\begin{aligned} h(x) &= x = project_1(x) \\ g(x, y, u) &= succ(u) = succ(project_3(x, y, u)) \end{aligned}$$

Most of the functions normally studied in number theory are primitive recursion. Addition, division, factorial, exponential and the n-th prime are all primitive recursion. From this we see that the primitive recursive functions are defined for all non-negative natural numbers (total functions).

From the definition it follows a primitive recursive function is computable by a Turing Machine. However, there are known functions which are not primitive recursive.

13.1.3 Ackermann Function

The Ackermann Function is a total function which is very fast growing. It grows faster than any primitive recursive function does. Ackermann's function is defined by:

$$\begin{aligned} A(0,y) &= y+1 \\ A(x,0) &= A(x-1,1) \\ A(x,y+1) &= A(x-1,A(x,y)) \end{aligned}$$

This definition is given by formula, which is not a primitive recursive formula. However this is not a proof, although this doesn't mean that the formula cannot be massaged into the primitive recursive function.

Ackermann's function is not primitive recursive.

13.1.4 Minimisation

Minimisation defines a new function f in terms of a total function g as follows (where x represents any number of arguments). The value $f(x)$ is determined by searching the following sequence for the smallest y such that $g(x,y) = 0$.

$$g(x,0),\ g(x,1),\ g(x,2),\ \ldots$$

If such a y exists then define $f(x) = y$. Otherwise, $f(x)$ is undefined.

$$f(x) = min(y, g(x,y) = 0)$$

The algorithm used by $f(x)$ is shown below:

```
y = 0;
while (not (g(x,y) = 0)){
    y = y + 1;
}
return y;
```

13.1.5 Recursive Functions: A Summary

A function is considered to be *partial recursive* if it can be built up using the base functions and the operations of composition, primitive recursion, and minimisation. A function can be computed by a Turing machine if and only if it is partial recursive, which can also be denoted as μ-recursive.

Example: Recursive Functions

Ex. 1 $f(x) = min(y, xy = 0)$ defines $f(x) = 0$.

Ex. 2 $f(x) = min(y, x + y = 0)$ defines $f(x) =$ 'if $x = 0$ then 0, else undefined'.

13.2 Model 6: Cellular Automata - The Game of Life (Interest Only)

As we saw towards the end of Part A - one variation of the Turing Machine is to have many tape heads where many cells can be modified in each step. If we take this one step further such that we can modify every cell in each step, all in parallel - we find the basis of *cellular automata*.

We will focus on one particular cellular automata called the *Game of Life* which was devised by Conway, a British mathematician in 1970. Rendell in 2000 and 2010 proved that the Game of Life is equivalent in power to the Universal Turing Machine.

The Game of Life is an infinite two-dimensional orthogonal grid of square cells. Each cell is in one of two states: live or dead. Every cell interacts with it's neighbours which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time - there are three possible transitions which can occur depending on the number of live neighbours:

Death if the count is less than 2 or greater than 3, the current cell is switched off (dies)

Survival if the count is exactly 2 or the count is exactly 3 and the current cell lives, the current cell is left unchanged (it lives on to the next generation)

Birth if the current cell is off (dead) and the count is exactly 3, the current cell is switched on (becomes live)

The initial pattern in the GoL is called the *seed* of the system. The first generation is created by applying the rules above simultaneously to every cell in the seed - births and deaths occur simultaneously in each generation, depending on the neighbours of the previous generation. The rules continue to be applied repeatedly to create further generations.

There are links in the slides on Moodle, as well as on Moodle itself, to simulators for the Game of Life.

Page 14

Lecture - B3: Diagonalisation and the Halting Problem

2025-11-24 14:00 Janka

We all like problem solving - this is a core component of Computer Science. However, what about problems which can't be solved? Does this mean that they're unsolvable, or does this mean that the solution is tricky to find and we'll eventually find it.

We have already seen a number of different computational models which have the same power as the Turing Machine. However, there are some limits of computation and we know that not all problems are algorithmically solvable (meaning they're not able to be solved by a TM or any computational model).

This lecture will introduce some classic problems which cannot be solved by any computer - they're known as the *Undecidable Problems*.

14.1 Introductory Concepts

There are a couple of concepts which we need to be familiar with to be able to understand this lecture.

14.1.1 Tool 1: Self Reference

If we take the following scenario:

- There is only a small town
- Every man in the own keeps himself clean-shaved; some by shaving themselves, some by visiting the barber
- the barber obeys the following rule (the definition of the barber): "He shaves all and only those men in town who do not shave themselves."

This presents a question: does the barber shave himself?

This feels like an obvious answer - Yes, as we know all men in the town keep themselves clean shaven. However for the definition of the barber to be considered true:

- If the barber does not shave himself, he must abide by the rule and shave himself
- If he does shave himself, according to the rule, he will not shave himself

We now get endlessly confused and decide there is no solution - this is a paradox.

14.1.2 Countable Sets

If we take two sets, S and T - how do we know if they're the same size or not?

If they're finite then we can pair-off the elements (seeing 1-1 correspondence), taking an element form S and an element from T until either both sets are empty (same size) or one set has elements remaining & one is empty (not same size).

However if the sets are infinite - ie a set of all the natural numbers $\mathbb{N}$, then we have no easy way of telling how big the set is so therefore we can't pair off in the same way.

We can use $\mathbb{N}$ to our advantage when referring to infinite sets; as we known $\mathbb{N}$ itself is infinite. Therefore we can define a set as *countably infinite* if it is the same size as $\mathbb{N}$ (i.e. we can assign a member form $\mathbb{N}$ in a 1-1 relationship to an element from the set we are trying to determine if it is countable or not). We may find that using a function (i.e. $f(x) = \frac{x-1}{2} + 1$) is useful in matching an element in our set to the set to an element in the set of natural numbers.

From this, we can see that a set is *countable* if it is finite or countably infinite - meaning we can construct a numbered list of all its elements.

Example: Countable Sets

The set of integers is countable, the set of odd numbers is countable, the set of rational numbers is countable.

14.1.3 Tool 2: Diagonalisation

This brings us onto diagonalisation, which was invented by Cantor (around 1880). Diagonalisation is the idea that there are some sets in which some sense contain different elements from another set, which therefore shows that your original set does not contain all the possibilities of the infinite elements which therefore proves that the original set is not countable.

Example: Diagonalisaiton

If we take a list of words of the same length:

```
Q U I E T
S T O N E
O F F E R
C L E A R
P H L O X
```

We can very easily see that there are words of the same length which are not in the list, for example `SNACK`. But how can we prove this? We need to find a way to definitively define what is going on here this is Level 6 - so we're beyond the point of plucking words out of the air and hoping for the best.

What about if we take the diagonal string within the above matrix and advance each string by one place. We take `QTFAX` and produce `RUGBY`.

This new word cannot be in the list because:

- It is different to the first word in the first letter
- It is different to the second word in the second letter
- etc...

14.1.4 Uncountable Sets

In opposite to the *countable sets* we have seen in the Diagonalisation section - such a thing exists called an *uncountable set* where it is not possible to count the elements of the set.

Cantor theorised that the set $P(\mathbb{N})$ is not countable.

If we assume that it is countable then we can write down a list of all the subsets of $\mathbb{N}$. It could look like this:

$$\{2,3\},\ \{4,7\},\ \{2,4,6,8\},\ \ldots$$

From this we can see that we have a function $f : \mathbb{N} \to P(\mathbb{N})$ that maps numbers to sets such that every set appears in the list.

If we define a set T as follows: add i to the set T exactly when $i \notin f(i)$, for example:

- $1 \in T$ because $1 \notin f(1)$
- $3 \in T$ because $3 \notin f(2)$
- etc...

However the set T is not on the list! T is different from the first set, the second set, etc. Therefore T is no ton the list which contains all subsets of $\mathbb{N}$ despite the fact that T is a subset of $\mathbb{N}$.

We have proven, by contradiction, that such a list does not exist and therefore $P(\mathbb{N})$ is not countable. This is the general rule - a power set of an infinitely countable set is uncountable.

14.2 Decision Problems

This brings us onto the next type of problem to consider - *decision problems.*

Definitions

Decision Problem is a problem that asks a question that has a *yes* or *no* answer.

Some examples of decision problems can be seen below:

- Decide whether in a given finite array of integers the maximum is larger than 10
- For a given language, L, decide whether a string s is from L or not
- Decide whether the following polynomial has a solution in the natural numbers

$$f(x) = 2x^{10} - 3x^8 + 4x^6 + 32x + 10$$

We can take a decision problem and view it as a formal language where the members of the language are instances whose answer is yes and the non-members are those instances whose output is no.

Definitions

Decidable means that for a decision problem, there is an algorithm that for every input instance of the program halts with a correct answer outputting ‘YES’ or ‘NO’.

Undecidable means that for a decision problem, there is not an algorithm that outputs ‘YES’ or ‘NO’ for every input instance of of the program.

Example: Decidable

To decide whether a particular string w is in a given regular language is a decidable problem.

Decidable problems correspond to recursive languages. They can be recognised by a Turing Machine that halts for every input.

14.2.1 Partially Decidable Problems / Undecidable Problems

Definitions

Partially Decidable problem is an undecidable problem if there is an algorithm that halts with the answer yes for those instances of the problem that have 'YES' answers, but may run forever for those instances of the problem whose answers are 'NO', and the opposite.

Partial decidable problems correspond to the recursive enumerate languages (unrestricted grammar). They can be recognised with Turing Machines.

Example: Partial Decisions

Ex. 1 Deciding whether a particular string w is in a language generated by an unrestricted grammar is a partially decidable problem.

Ex. 2 Deciding whether a polynomial has a solution in the natural numbers is a partially decidable problem.

Now we've seen what the undecidable problem is - we need to define how we prove tht a problem is undecidable.

To prove that a problem is undecidable - it is enough to compare the number of decidable problems and the number of decision problems, or to simply find one and prove that it is undecidable.

Example: Undecidable Problems

Ex. 1: How many TMs exist?

If we let S be a countable set of symbols, then any TM can be coded as a finite string of symbols over S (all transition functions one after the other). There are a countable number of finite strings over S.

There exists an assignment which assigns a unique number $< M >$ to each TM M.

This means that the set of TMs is countable.

Ex. 2: How Many Languages Can We Have?

A language is a subset of a countable set of strings. We have shown that the size of the set of all subsets of $\mathbb{N}$ is not countable. This therefore means that the set of languages is uncountable.

14.3 The Halting Problem

The *Halting Problem* is famous because it was one of the first problems to be proven to be algorithmically undecidable (by Turing in 1936).

The Halting problem asks "is there an algorithm that can decide whether the execution of an arbitrary program halts on an arbitrary input?"

Page 15

Lecture - B4: Undecidable Problems

📅 2025-11-24 🕒 15:00 🧑‍🏫 Janka

15.1 The Halting Problem

As we saw in the last lecture - the halting problem is set around deciding if there is an algorithm that can decide whether the execution of an arbitrary program halts on an arbitrary input.

We cna see this in action if we consider the following JavaScript programs:

```
for(quarts = 1 ; quarts < 10 ; quarts++){
    liters = quarts/1.05671;
    alert( quarts+" "+liters);
}
```

We can clearly see that it will terminate after outputting 10 lines of output.

```
limit = prompt("Max Value","");
    for(quarts = 1 ; quarts < limit ; quarts++){
    liters = quarts/1.05671;
    alert( quarts+" "+liters);
}
```

The above program will alert as may times as indicated by the input.

```
green = ON; red = amber = OFF;
while(true){
    amber = ON; green = OFF; wait 10 seconds;
    red = ON; amber = OFF;
    wait 40 seconds; green = ON; red = OFF;
}
```

The above program will run forever, never halting.

From these three algorithms, we learn a few things:

- Algorithms may contain loops which may be infinite or finite in length
- The amount of work done in algorithms usually depends on the data in put
- Algorithms may consist of various numbers of loops, nested or in sequence.

15.1.1 Solving the Halting Problem

Using what we now know - we want to try and solve the *Halting problem.* We first try running the program with the given inputs; if the program stops we know the program halts; but if the program doesn't stop in a reasonable amount of time - we cannot conclude that it won't stop as we don't know if we waited long enough or not. What is a reasonable amount of time anyway?

The Halting problem is partially decidable. This means that there is no algorithm (and no TM) which could solve the Halting problem - meaning return the correct 'YES' or 'NO' answer for any input in the finite number of steps. There is only an algorithm for a finite 'YES' decision.

We can prove the Halting problem is partially decidable using a direct diagonalisation argument.

If we define the set:

$$A = \{< M, w >: M \text{ is a TM that accepts } w\}$$

where $< M, w >$ is a unique coding. So we consider all possible TMs (which are countable) and all possible strings (again, countable) and if M accepts w, it belongs to A.

Another formulation of the Halting problem can be defined as: Is there a TM which will recognise the set A. No - there is not! However, for some simpler computing models, such as FA or PDA, we can always find a TM which recognises A.

15.1.2 Proving the Halting Problem

We can prove the Halting problem using a Proof by Contradiction: *Suppose there was a machine SOLVER that on every input w and for every TM M would tell us if M accepted / rejected w.*

If we now build a new TM, *OPPOSER*, that does the following:

- *OPPOSER* takes the input w and determines the TM $< w >$ that w encodes (If the input is not the encoding of a TM then *OPPOSER* rejects the input)
- Ask *SOLVER* for the answer: "Does the TM $< w >$ accept w?"
- If *SOLVER* accepts, then *OPPOSER* rejects;
  if *SOLVER* rejects, then *OPPOSER* accepts

OPPOSER is a perfectly valid TM because *SOLVER* always halts.

Let's play with this a bit now. What does *OPPOSER* do if the input is the encoding of *OPPOSER*. Let <*OPPOSER*> = $\tilde{w}$:

- *OPPOSER* asks *SOLVER* for an answer on "Does the TM $< \tilde{w} >$ accept $\tilde{w}$?"
- If *SOLVER* claims that *OPPOSER* accepts $\tilde{w}$, then *OPPOSER* rejects $\tilde{w}$.
- If *SOLVER* claims that *OPPOSER* rejects $\tilde{w}$, then *OPPOSER* accepts $\tilde{w}$.

This is now what was to be expected - and is the opposite to that of the above. The thing which went wrong this time around was assuming that *SOLVER* exists. Meaning that *SOLVER* does not exist.

The Halting problem shows that computers can't help mathematicians to solve old hard problems.

Example: Golbach's Conjecture

If we consider a TM that tries to find a counterexample to Golbach's conjectture (from the 18th century):

Every even number ≥ 4 is the sum of two primes.

The TM tries every even value of n in increasing in order. For each n, it checks if there is a value of i such that i, $n - i$ are primes. If not, it stops; otherwise it continues forever.

If the halting problem returns 'YES' - it has found a counterexample of Goldbach's conjecture. However if the Halting problem returns 'NO' - Goldbach's Conjecture is valid.

Unfortunately - the halting problem is undecidable, so computers can't help mathematicians in this way.

15.1.3 Proving a Problem is Undecidable

To prove that a problem is undecidable - we have two options.

Firstly, we can use *contradiction*: we start by assuming it is solvable and see if it leads to an internal contradiction, or conflicts with something else we believe is true.

Or alternatively, we can use *reduction*: see if another unsolvable problem can be reduced to solving it. This means if we assume it is solvable, then can we show that another problem we believe is unsolvable (like the Halting problem) be solved.

The rest of this lecture will look at examples of undecidable problems.

15.2 Post Correspondence Problem

The *Post Correspondence Problem* was introduced by *Post* in 1946. It is a decision problem that looks at if there is a sequence of pairs of strings from a given finite set such that the concatenation of the first in pairs equals the concatenation of the second in pairs.

We can see this formalised if we take an alphabet, Σ, and a finite sequence of pairs of strings over Σ:

$$(s_1, t_1), \dots, (s_n, t_n)$$

We then look for a sequence of indexes $i_1, \dots, i_k$ with repetition allowed such that:

$$s_{i_1}, \dots, s_{i_k} = t_{i_1}, \dots, t_{i_k}$$

Example: Post Correspondence Problems

Ex. 1: Valid

If we consider an instance of the problem consisting of the following sequence of pairs over $\{a, b\}$:

$$1.(ab, a),\ 2.(aba, bb),\ 3.(aa, b),\ 4.(b, aab)$$

After some fiddling - we find out solution:

$$\underbrace{ab}_{1}\underbrace{aa}_{3}\underbrace{b}_{4} = \underbrace{a}_{1}\underbrace{b}_{3}\underbrace{aab}_{4}$$

We see here that (1, 3, 4) is a solution.

Ex. 2: Invalid

We can see how an instance over $\{a, b\}$:

$$1.(ab, a),\ 2.(b, ab)$$

may not have a solution.

Ex. 3: Repetition

We can take another instance of the problem, again over $\{a, b\}$:

$$1.(a, baa),\ 2.(ab, aa),\ 3.(bba, bb)$$

And we can find that:

$$\underbrace{bba}_{3}\underbrace{ab}_{2}\underbrace{bba}_{3}\underbrace{a}_{1} = \underbrace{bb}_{3}\underbrace{aa}_{2}\underbrace{bb}_{3}\underbrace{baa}_{1}$$

Therefore we can see that (3,2,3,1) is a valid solution.

All the problems we've looked at in the above example have been reasonably small; small enough that we can just brute force the answer out of it. However this will begin to be an issue as we use larger and larger input strings.

If there is no solution - the algorithm will just keep churning, and churning, and churning - never stopping. This means we can see that the Post correspondence problem is undecidable, but partially decidable.

15.3 The Tiling Problem

⚠ Add when I'm feeling brave with TikZ

15.4 Hilbert's Tenth Problem

Hilbert's 10th problem states that: Does a polynomial equation $p(x_1, \ldots, x_n) = 0$ with integer coefficients have a solution consisting of integers.

Example: Specific Instance of Hilbert's Tenth Problem

We can generally solve specific instances of the problem - such as integer solutions to the equation:

$$2x + 3y + 1 = 0$$

In 1970, Matiyasevich proved that Hilbert's tenth problem is undecidable. Meaning that there is no algorithm to decide whether an arbitrary polynomial equation with integer coefficients has a solution of integers. However this problem is partially decidable - as we can write an algorithm (or a TM) which returns a solution if the solution exists.

15.5 The Equivalence Problem

The Equivalence problem asks if there exists an algorithm that can decide whether two arbitrary computable functions produce the same output.

We can tell that two functions, f and g are equal where $f(x) = x + x$ and $g(x) = 2x$.

So is the Equivalence Problem a decidable problem? No, of course it isn't - the equivalence problem is undecidable.

15.6 The Total Problem

The Total Problem asks if there is an algorithm to tell whether an arbitrary computer function is total.

As we saw in DMAFP - a Total Function is one who for every input has a defined output.

Example: A Total Problem(atic Function)

If we take a function $f : \mathbb{N} \to \mathbb{N}$ as defined by $f(x) = x + 1$ we can see that it's total.

The Total Problem is an undecidable problem as there is not an algorithm to answer the problem for all computable functions.

Page 16

Lecture - B5: Introduction to Computational Complexity

2025-12-01 14:00 Janka

Up until today's lecture - we've taken a bit of a "timey-wimey" approach to algorithmic complexity; in that we haven't cared how long it takes, rather looking at if a solution will be found *eventually* or not. This all changes today...

16.1 Why is Time Efficiency Important?

Within Computer Science, a major goal is to understand how to solve problems using computers. We can see a few example problems we may try to solve below:

Array Sum Problem Given an array of n integers, return their sum

Searching for a key Given an ordered list L of n integers, find a given key in it

Sorting Problem Given an array L of integers, arrange L in ascending order (i.e. for any $1 \leq i < j \leq n, L[i] \leq L[j]$)

Travelling Salesperson Problem Find the shortest path for a salesman to visit all the cities on his route

Some of these problems we have seen before and some are new to us today; it's not the problem that's important - rather we're thinking about *how* we solve this problem, and how efficient the algorithm is.

When we come to solve a problem - we'll commonly go through a few steps:

1. Designing an algorithm or step-by-step procedure for solving the problem
2. Analysing the correctness and efficiency of the algorithm
3. Implementing the procedure in some programming language
4. Testing the implementation

We'll focus around step 2 today, looking at the time efficiency of an algorithm.

Informally, we may often refer to a program as "fast" or "slow," but in reality - this is a naff definition and we need to be more precise. The absolute execution time of an algorithm can depend on may factors:

- the algorithm used to solve the problem
- the programming language used to interpret the algorithm (where interpreted languages such as Python are typically slower than compiled languages such as C++)
- the quality of the actual implementation (where good code can be much faster than poor, sloppy code)
- the machine on which the code is run (a supercomputer is faster than a laptop)

- the size of the input (meaning searching through a list of length 1000 takes longer than searching through a list of length 10)

16.2 Time Complexity

The *time complexity function*, $T(n)$ expresses the number of primitive operations (usually the addition of two numbers or their comparison) which the algorithm needs to execute on an input of size n.

In analysing the efficiency of an algorithm - we focus on the *speed* of the algorithm (the complexity of the algorithm) as a function of the size of the input on which it is run; where we think of *speed* in terms of the number of basic steps / operations in the program.

16.2.1 Worst Case Time Complexity

Generally when discussing time complexity, we'll be considering the *worst-case* time complexity. The algorithm has to finish in time $T(n)$ for all instances of size n, but for some instances it may finish quicker.

Example: Array Sum Problem

The Array Sum Problem adds all the n integers in the array S. It's inputs are the positive integer n and the array of numbers S indexed from 1 to n; and its output is the sum of the integers in S.

An algorithm can be seen below:

```
function sum (n: integer; S : array[1, 2, . . . , n] of integers)
    var
        i : index;
    begin
        sum := 0;
        for i := 1 to n do
            sum:= sum + S[i]
        print sum;
    end;
```

We can see this quite simply loops over all the array elements, adding each in turn to `sum` which gets outputted at the end of the function.

The basic operation here is the addition of an item in the array to `sum`. This is the most expensive (in time) of operations in the function. The size of the input is proportional to n which is the number of items in the array. Therefore, the time complexity function can be seen as $T(n) = A \cdot n$. (Where A is a suitable constant - ie how many primitive operations correspond to our operation with the array)

The above example shows us an important detail of the way in which time complexity works - only *some* of the statements in a function are weighty by way of time complexity. What this means is that an if statement or a print statement are often considered to be negligible - as they are constant. However, a loop is considered to be expensive by way of time so this is a factor we consider.

Example: Searching for a Key

We will explore two different searching algorithms and the way in which they approach the problem of searching for a key in an ordered list of numbers.

Ex. 1: Sequential Search The Sequential Search works by looping through all the elements in the array and checking for a match to the key. An algorithmic implementation can be seen below:

```
procedure seqsearch (n: integer; S : array [1, 2, . . . , n] of integers;
                    x: integer);
    var
        location : index;
    begin
        location := 1;
        while location <= n and S[location] != x do
            location := location + 1;
        if location > n then
            location := 0
    end;
```

This algorithm also works with unsorted arrays and and has the same complexity: $T(n) = A \cdot n$.

Ex. 2: Binary Search Algorithm The Binary Search Algorithm also finds if the key x is in the *sorted* array S of n integers. The key component here is that the array must already be sorted. The idea of the algorithm is to, at each step, cut the number of possible positions the search key could be in by half. An algorithmic implementation can be seen below:

```
procedure binsearch (n: integer; S : array [1, 2, . . . , n] of integer;
                    x: key);
    var location : index; low, high, mid : index;
    begin
        low := 1; high := n; location := 0;
        while low <= high and location = 0 do
            mid := (low + high) div 2
            if x = S[mid] then location := mid
                // Checks the middle entry in the array!
                else if x < S[mid] then high := mid - 1
                    else low:= mid + 1
        od;
    end;
```

We can see here that the time complexity of a Binary Search is lower than that of a sequential search, as it will do considerably less operations given that the binary search halves the list every loop until nothing is left. The binary search has a worst case time complexity of: $T(n) = B \cdot \log_2 n$.

From these examples - we can see the name "worst-case" in action - as for both algorithms the key may be found in the first step, which would be excellent; but it also may never be found meaning all the possible options must be checked according to each algorithm's way of working.

Example: Travelling Salesman Problem

The Travelling Salesman problem is a problem which has been studied for as long as it's existed. The problem seeks out the shortest path between all nodes on a weighted graph, with the least accumulated weight for the whole journey, representing the journey a salesman takes travelling between cities in the country selling his goods.

We're left trying to find the optimal route for our salesman, as he's on a tight budget and doesn't want to spend lots of money on train travel - so we have to find him a route. If we take that there are n cities to visit, and he's currently in the first one - then there are:

- $n - 1$ ways to choose the next city
- $n - 2$ ways to choose the city after that
- etc, etc, until he only has 1 city left to visit

We can formalise this as follows:

$$\frac{(n-1) \times (n-2) \times \ldots \times 1}{2} = \frac{(n-1)!}{2}$$

This defines the factorial function, which grows very quickly with n:

$$15! = 1,307,674,368,000$$

This is the start of our brute-force algorithm:

```
begin
    mindist := 'total distance of any one route';
    for each of the possible different routes R
        begin
            compute total distance D of R;
            if D <= mindist then mindist := D
        end
end;
```

So how's this looking from a Time Complexity perspective? There are $(n-1)!$ possible routes to the travelling salesman problem so to compute the total distance per route requires n additions. Therefore we get a time complexity of $T(n) = C \cdot n!$.

In tangible time - how long does the algorithm take? If $n = 30$ then we would have to do 10^{20} computations. The fastest processors can do is about 10^{12} additions per second. So this would take 10^{8} seconds (a couple of weeks) to run. If we increase n to 100, the time needed to execute increases to exceed the age of the universe.

There have been various branch-and-bound algorithms used to process TSPs containing 85900 cities. Progress improvement algorithms which use techniques of linear programming works well for up to 200 cities. An exact solution for 15,112 German towns was found in 2001 base don linear programming. The computations were performed on a network of 110 processors, the total computation time was equivalent to 22.6 years on a single 50MHz Alpha processor.

Page 17

Lecture - B6: Asymptotic Growth

2025-12-01 15:00 Janka

Usually, there are several algorithms which exist to solve the same problem. We want to find and use the most efficient, in terms of resources like time and storage, to solve the problem. Efficient algorithms lead to better programs; efficient programs sell better; efficient programs make sell better and make better use of hardware; and programmers who write efficient programs are more marketable than those who don't.

If we return to an example we saw in the previous lecture - searching for a key in an array. We saw two different methods: the Sequential Search and the Binary Search; and we saw that the binary search was more efficient than the sequential search.

There aren't, however, just two types of algorithmic efficiency steps - there are many.

A graph showing the different functions and their growth can be found in the slides on Moodle.

The below table shows the growth of these functions in a numerical capacity.

n	$n \log_2 n$	n^2	n^3	2^n	$n!$	n^n
10	33	100	1000	1024	7 digits	11 digits
50	282	2500	125000	16 digits	65 digits	85 digits
100	665	10000	7 digits	31 digits	161 digits	201 digits
300	2469	90000	8 digits	91 digits	623 digits	744 digits
1000	9966	7 digits	10 digits	302 digits	:o	:o

Table 17.1: Comparison of growth functions

For reference - the number of protons in the known universe has 79 digits; and the number of microseconds since the Big Bang has 24 digits - so some of these numbers are flipping humongous!

17.1 Asymptotic Analysis of the Algorithm

Asymptotic refers to something that approaches a specific value or curve arbitrarily close as it gets closer to a limit, most often infinity.

We are usually interested in how fast the time a program takes to run changes as the size of the input becomes larger and larger. As well as the fact that the counting of basic steps in the time complexity function doesn't given an accurate picture - the details depends on the programming language, compiler, etc; the distance is at most a constant factor.

Comparing the time complexity of two algorithms is the same as comparing the asymptotic growth of the time complexity functions.

Example: Asymptotic Analysis

Ex. 1 If the algorithm A has a time complexity $T_A(n) = n + 10$ and the algorithm B has time complexity $T_B(n) = n$ then the asymptotic growth of both functions is the same n, so $n \sim n + 10$.

Ex. 2 If $T_A(n) = 4n^2 + 3n + 10$ and $T_B(n) = 2n^2$ then asymptotic growth of both functions is the same n^2, so $4n^2 + 3n + 10 \sim 2n^2 \sim n^2$.

Ex. 3 If $T_A(n) = 4n^2 + 3n + 10$ and $T_B(n) = 2^n$, the asymptotic growth of both functions is not the same.

From the above we can see a generalisation of what we're looking for: we look at the running time of an algorithm when the input size n is large enough so that constants and lower-order terms do not matter.

These ideas can be formalised to show the growth rate of a function with the dominant term in respect to n and ignoring any constants in front of it:

$$k_1 n + k2 \sim n$$
$$k_1 n \log n \sim n \log n$$
$$k_1 n^2 + k_2 n \sim n^2$$

We also want to formalise that one type of algorithm is better than another (for example $n \log n$ is better than n^2). There are three different notational forms we see to formalise this: Big-O, Ω-notation, and Θ-notation

17.2 Big-O Notation

Big O notation represents the asymptotic upper bound of a function. When f and g are two non-negative functions then $f(n)$ is $O(g(n))$ if f grows at most as fast as g.

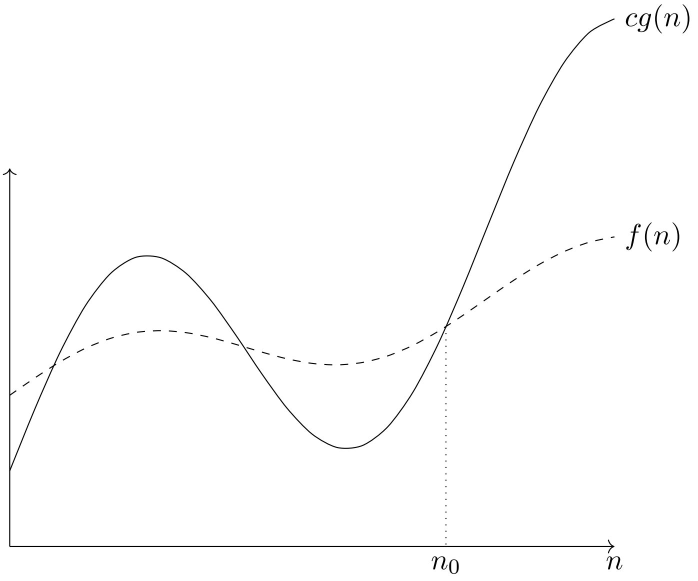


Figure 17.1: Big-O notation functions

This can be formally defined as $f(n) = O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq n_0, f(n) \leq c \cdot g(n)$.

We express our Big O notation as:

$$f(n) = O(g(n)) \quad \text{or} \quad f(n) \in O(g(n))$$

and read this as "$f(n)$ is big O of $g(n)$"

Example: Big O

Ex. 1 $2n^2 + 10 = O(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that:

$$c \cdot g(n) \geq 2n^2 + 10 \text{ for all } n \geq n_0$$

After a think, we an decide that $2n^2 + 10 = O(n^2)$ because

$$3n^2 \geq 2n^2 + 10 \text{ for all } n \geq 4$$

Therefore we can see that $c = 3$ and $n_0 = 4$. However it would also work for many other combinations such as $100n^2 \geq 2n^2 + 10$ for all $n \geq 1$.

Ex. 2 $\frac{1}{3}n^2 - 3n = O(n^2)$ because:

$$\frac{1}{3}n^2 - 3n \leq cn^2 \text{ if } c \geq \frac{1}{3} - \frac{3}{n}$$

which is true if $c = \frac{1}{3}$ and $n \geq 1$

Ex. 3 Generalising a bit more, we can see how $O()$ gives an upper bound on f, but not always a tight one:

$$n = O(n),\ 3n = O(n^2),\ 7n = O(n^3),\ n = O(n^{100})$$

17.3 Big-Ω Notation

Big Omega (Big-Ω) is an asymptotic lower bound of a function. Informally described as when f and g are two functions then $f(n)$ is $\Omega(g(n))$ if f grows at least as fast as g.

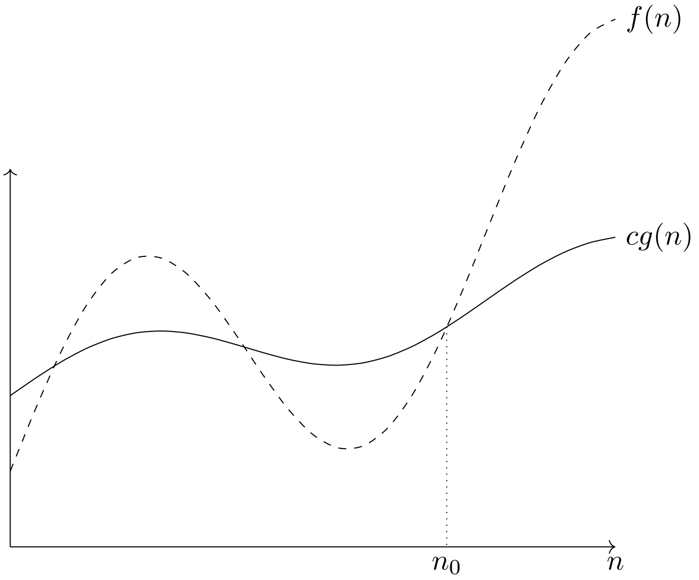


Figure 17.2: Big-Ω notation functions

This can be formally defined as $f(n) = \Omega(g(n))$ if there exists $c, n_0 \in \mathbb{R}^+$ such that for all $n \geq$

$n_0, f(n) \geq c \cdot g(n)$.

We express our Big-Ω notation as:

$$f(n) = \Omega(g(n)) \quad \text{or} \quad f(n) \in \Omega(g(n))$$

and read this as "$f(n)$ is big omega of $g(n)$"

Example: Big-Ω Notation

Ex. 1 $4n^2 - 10 = \Omega(n^2)$ because $f(n) = 4n^2 - 10$ and $n \leq 4n^2 - 10$ for all $n \geq 2$, hence $c = 1$ and $n_0 = 2$

Ex. 2 $\frac{1}{3}n^2 - 3n = \Omega(n^2)$ because $\frac{1}{3}n^2 - 3n \geq cn^2$ if $c \leq \frac{1}{3} - \frac{3}{n}$ which is true if $c = \frac{1}{6}$ and $n \geq 18$.

Ex. 3 $\Omega()$ gives a lower bound on f but not necessarily a tight one:

$$n = \Omega(n),\ n^2 = \Omega(n),\ n^3 = \Omega(n),\ n^{100} = \Omega(n)$$

17.4 Big-Θ

Big Theta (Big-Θ) notation is used to denote the asymptotic tight bound of a function. Informally, this is when f and g are two functions then $f(n)$ is $\Theta(g(n))$ if f is essentially the same as g to within a constant multiple.

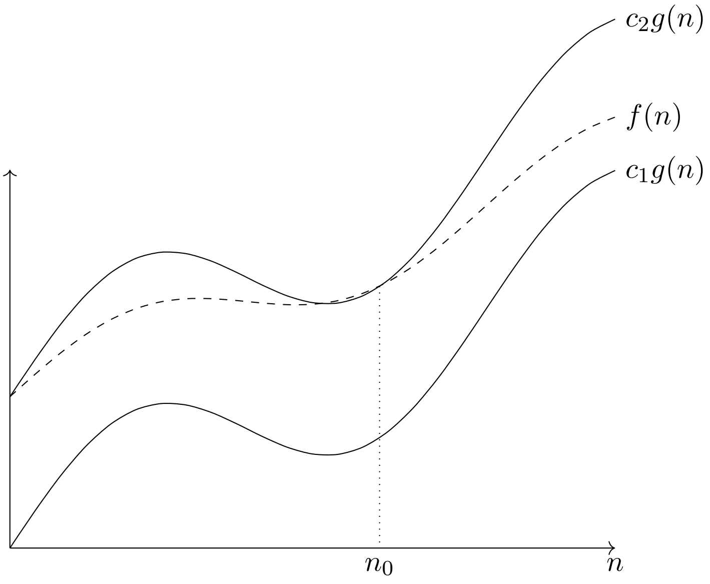


Figure 17.3: Big-Θ notation functions

Formally, $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ meaning there exists $n_0, c_1, c_2 \in \mathbb{R}^+$ such that for all $n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

We express our Big-Θ notation as:

$$f(n) = \Theta(g(n)) \quad \text{or} \quad f(n) \in \Theta(g(n))$$

and read this as "$f(n)$ is big theta of $g(n)$".

Example: Big-Θ Notation

Ex. 1 $4n^2 - 10 = \Omega(n^2)$ and $4n^2 - 10 = O(n^2)$ which means $4n^2 - 10 = \Theta(n^2)$.

Ex. 2 $\frac{1}{3}n^2 - 3n = \Omega(n^2)$ and $\frac{1}{3}n^2 - 3n = O(n^2)$ which means $\frac{1}{3}n^2 - 3n = \Theta(n^2)$

Ex. 3 However, Big-Θ is considerably more restrictive than either of Big O or Big-Ω alone...

$$n \neq \Theta(n^2),\ n^2 + 3n \neq \Theta(n^3),\ \log n \neq \Theta(n)$$

17.5 Comparing $T(n)$

Naturally, as with all things in life, we want to know which is best. In this case - we are looking for the lowest time complexity of an algorithm as this will naturally be the quickest to run. We can compare two functions with $f(n) \prec g(n)$ if and only iff:

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) \neq \Theta(g(n))$$

Below is a hierarchy of some familiar functions from this lecture, according to their growth rates:

$$1 \prec \log n \prec n \prec n \log n \prec n^2 \prec n^3 \prec 2^n \prec 3^n \prec n! \prec n^n$$

We can even name these functions:

Constant Functions $\Theta(1)$

Logarithmic Functions $\Theta(\log n)$

Quadratic Functions $\Theta(n^2)$

Cubic Functions $\Theta(n^3)$

Polynomial Functions $\Theta(n^a)$

Exponential Functions $\Theta(2^n), \Theta(a^n), \Theta(n!), \Theta(n^n), \ldots$

Calculating the time complexity, $T(n)$ for an algorithm is often very difficult - there are lots of different techniques which could be used. When analysing an algorithm, you might need to estimate the complexity of sums; so it's worth knowing their complexity:

$$\sum_{i=1}^{n} i = \Theta(n^2)$$
$$\sum_{i=1}^{n} i^2 = \Theta(n^3)$$
$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$
$$\log n! = \Theta(n \log n)$$

Example: Asymptotic Behaviour

If we take the function $f(n) = log_{10}(n^3 - 4n^2 + 6n + 5)$.

We can see pretty clearly that when $n > 20$ that $n^2 < n^3 - 4n^2 + 6n + 5 < n^4$.

Therefore $f(n) > 2\log_{10} n, n > 20$, thus $f(n) = \Omega(\log_{10} n)$.

We can also then see that $f(n) < 4\log_{10} n, n > 20$, therefore $f(n) = O(\log_{10} n)$.

Putting this together - we can see that $f(n) = \Theta(\log_{10} n)$

Exponential time is bad, polynomial time is better. For each problem, it is always best to find an algorithm with a polynomial time complexity and then to make the leading exponent of that polynomial as small as possible.

17.6 Tractable and Intractable

A Tractable Problem, one that's easy to control or deal with, is a problem that is solvable by a polynomial-time algorithm. There are a number of examples of *tractable* problems below:

- Searching an unordered list
- Searching an ordered list
- Sorting a list of two numbers
- Multiplication of two matrices
- Finding a minimum spanning tree in a weighted graph
- Finding a Eulerian circuit in a graph
- Finding the shortest path between any two vertices in a graph

As you'd expect - an Intractable problem, one that which is hard to control or deal with, is a problem where there is not a known polynomial time algorithm. For some of these problems - there exists a proof that a polynomial algorithm doesn't exist, and for others - no one has been able to prove it so far. There are a number of intractable problems below:

- Travelling salesman problem
- Tower of Hanoi
- Finding a Hamiltonian cycle in a graph
- Finding the longest path between any two vertices in a graph
- More to be covered later in this module...

Page 18

Lecture - B7: Analysis of Algorithms

2025-12-08 14:00 Janka

As we have seen over the last few lectures, the design of the algorithm is very important when solving a problem. We often strive to find a fast algorithm that doesn't use lots of memory and time (both of which are 'bounded' resources). We saw, last week, some fundamental techniques and tools used to analyse algorithms for their time and space that they require to execute, and learnt that our main focus is the time complexity function of a given algorithm. The time complexity for a given algorithm is a dependency of the time it takes to solve a problem as a function of the problem dimension (or size). The worst-case analysis gives an upper bound as to how much time will be needed to solve any instance of the problem.

18.1 Sorting Algorithms

We're very familiar with the concept of a sorting algorithm. These are the handy algorithms that when fed an array of integers, S, which is indexed from 1 to n and a positive integer, n, they return the array S containing the integers sorted into ascending order.

There are a number of sorting algorithms which exist, but we want to find out which of them are the best asymptotically, i.e. when $n \to \infty$.

> For all the below examples, there are links in the slides on Moodle which contain graphical animations of the sorting algorithms in practice.

18.1.1 Bubble Sort Algorithm

The bubble sort works by going through the list of numbers comparing each pair of adjacent items and swapping them if they are in the wrong order. The algorithm repeats passing through the list unitl there are now swaps needed which indicates that the list is sorted.

The algorithm for the bubble sort is shown below, including it's time complexities per line

```
procedure bubblesort (n: integer; var S: array [1...n] of integers)
    var i, j: integer;
    begin
1.      for i := 1 to n - 1 do               ($\sum_{i=1}^{n-1}(n-1) \times Const$)
2.          for j := 1 to n - i do           ($(n-i) \times Const$)
3.              if S[j] > S[j+1] then        ($1$ comparison : $Const$)
4.                  swap S[j] and S[j+1]     ($Const$)
    end
```

From the above we can see a few things. Line 3 uses one comparison, which is always in constant time. The swap of two elements which take place in line 4 happens in constant time. The for loop running on lines 2-4 runs in $(n-i)\times$ constant time, while the for loop on lines 1 to 4 runs in $\sum_{i=1}^{n-1}(n-i)\times$ constant time.

We can express simplify this as follows:

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

Therefore we can see that $T(n) = \Theta(n^2)$

18.1.2 Exchange Sort Algorithm

The exchange sort works differently to the bubble sort. It works by comparing the first unsorted element with each following element of the array, making any necessary swaps to sort the first then the second, etc elements into the right place. This is repeated unitl no swaps are needed which indicates that the list is sorted.

The algorithm including the relevant lines complexities can be seen below.

```
procedure exchangesort (n: integer; var S: array [1...n] of integers)
    var i, j: integer;
    begin
1.      for i := 1 to n - 1 do          $(\sum_{i=1}^{n-1}(n-i) \times Const)$
2.          for j := i + 1 to n do      $((n-i) \times Const)$
3.              if S[j] < S[i] then     $(Const)$
4.                  swap S[i] and S[j]  $(Const)$
    end
```

The Exchange Sort algorithm's complexity tells a similar story to that of the bubble sort. Line 3 uses one comparison, therefore constant time and the swap taking place in line 4 is always constant time. The for loop on lines 2 to 4 runs in $(n-i)\times$ constant time and the for loop on lines 1 to 4 uses $\sum_{i=1}^{n-1}(n-i)\times$ constant time to execute.

This can be simplified as follows:

$$\sum_{i=1}^{n-1}(n-i) = n(n-1) - \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Therefore, as with the bubble sort, the worst-case time complexity for the Exchange Sort is $T(n) = \Theta(n^2)$.

18.1.3 Insertion Sort

The insertion sort works by maintaining a sorted front section of the array; and then at each step inserting the current element to sort into the appropriate point in the sorted section, then moving onto the next unsorted element.

An algorithm to do this task can be seen below.

```
procedure insertsort (n: integer; var S: array [1...n] of integers)
    var i, j, key: integer;
    begin
        for j:= 2 to n do
            key := S[j]; i:= j - 1;
            while i > 0 and S[i] > key
                S[i+1] := S[i]; i := i - 1;
        S[i+1] := key
    end
```

Analysing the time complexity of the insertion sort gives us the same outcome as the bubble and exchange sort, that being a worst case complexity of $T(n) = \Theta(n^2)$.

18.1.4 Merge Sort

From the three sorting algorithms we've seen so far - it would seem that the fastest an array of n integers can be sorted is in $T(n) = \Theta(n^2)$ time. However, this is wrong. The merge sort is much faster.

The merge sort works by taking the initial array and then dividing it into two sub-arrays with approximately half the items in each sub-array. Each sub-array is then sorted by recursively calling the same algorithm. Then the solutions ot the sub-arrays are merged into a single sorted array.

The merge sort algorithm is an example of a *divide-and-conquer* algorithm.

An example algorithm for the implementation of merge sort can be seen below

```
procedure mergesort (n: integer; var S: array [1...n] of integers)
    const h = n/2;
          m = n/2 + 1;
    var U: array[1...h] of integer;
        V: array[1...m] of integer;
    begin
        if n > 1 then
            copy S[1] through S[h] to U;
            copy S[m] through S[n] to V;
            mergesort(h, U);
            mergesort(m, V);
            merge(h, m, U, V, S);.
        end
    end;
```

From the above algorithm we can see that where $n = 1$, the mergesort is complete, therefore this uses constant time. However where $n \neq 1$ - we have to recursively sort $S[1...\lceil n/2\rceil]$ and $S[\lceil n/2\rceil + 1...n]$ which gives a time complexity of $2T(n/2)$.

To be pedantic - where $n \neq 1$ the time complexity *should* be $T(\lceil n/1\rceil) + T(\lfloor n/2\rfloor)$ instead of $2T(n/2)$ but the difference here does matter asymptotically.

Now that we've split our array into the smallest sub-array we can find and sorted them, we need to merge them. This calls for the *merge* algorithm.

```
procedure merge (h, m: integer; U: array [1...h] of integer
                 V: array [1...m] of integer;
                 S: array [1...(h+m)] of integer;)
    var i, j, k: index;
    begin
        i := 1; j := 1; k := 1;
        while i <= h and j <= m do
            if U[i] < V[j] then S[k] := U[i]; i := i + 1
                else S[k] := V[j]; j := j + 1
            k := k + 1;
        od;
        if i>h then copy V[j] through V[m] to S[k] through S[h+m]
            else copy U[i] through U[h] to S[k] through S[h+m]
        // Once one list is empty, copy the rest
    end;
```

The merge part of the algorithm always has a constant time: *const* × $(h + m)$.

We can see a recurrence for the merge algorithm's time complexity:

$$T(n) = \begin{cases} C_0, & \text{if } n = 1; \\ 2T(n/2) + C \times n, & \text{if } n > 1. \end{cases}$$

In the above recurrence, C_0 and C are suitable constants; in further discussions we omit the constant C_0.

We can use a recursion tree to solve the time complexity of a merge sort.

⚠ add diagram of tree and explanation here including about last line

The worst case time complexity for a merge sort can be seen as $\Theta(n \log_2 n)$. From this we can learn that the merge sort asymptotically beats the insertion, exchange and bubble sort in the worst case as $n \log_2 n$ grows more slowly than n^2. This is as fast as we can get with a sorting algorithm.

18.2 Recurrences

We have seen an example of a *recurrent problem* with the merge sort this lecture. Recurrent problems come up again and again in the analysis of recursive algorithms. A recurrent problem is one which is solved with recursion, i.e. calling itself from inside itself with different input parameters.

A common example of a problem solved using a recurrence is *The Tower of Hanoi*. This is a problem where a tower of n disks are initially stacked in increasing size on one of three pegs. The objective is to transfer the entire tower of one of the other pegs, however we can only move one disk at a time and we must never place a large disk on top of a smaller one. The question we seek to solve is finding how many steps do we need for n disks?

We can start trying to solve this by investigating a small number of disks, where $T(n)$ is the minimum number of steps to solve for n disks:

$$T(0) = 0, \quad T(1) = 1, \quad T(2) \leq 3, \quad T(3) \leq ??$$

From this we can start to see a pattern - the Towers of Hanoi problem can be solved recursively, as we know the base case (where $n = 1$). This means the problem always has a solution. The generalised formula for this can be seen as follows:

$$T(n) \leq 2 \times T(n-1) + 1$$

Or, with a specific example, trying to solve for $n = 3$:

$$T(3) \leq 2 \times T(2) + 1 = 7$$

Extrapolating this idea is how we can find the lower bound. We know that to solve the problem we must move the nth disk from the first post to a different post, which then means the $n-1$ smaller disks must all be stacked out of the way on the only remaining post. To arrange the $n-1$ smaller disks this way requires at least T_{n-1} moves. This gives us our base case and recurrence case:

$$\begin{aligned} &T(1) = 1, \\ &T(n) = 2 \times T(n-1) + 1 \end{aligned}$$

To solve a recurrence for a given n, we first construct what we know, using the recurrence case. We then substitute what we don't know with the next instance of the recurrence case. We repeat this until we reach something that we do know, the base case.

$$\begin{aligned}
T(n) &= 1 + 2T(n-1) \\
&= 1 + 2(1 + 2T(n-1)) \\
&= 1 + 2 + 4T(n-2) \\
&= 1 + 2 + 4 + 8T(n-3) \\
&= \ldots \\
&= 1 + 2 + 4 + \ldots + 2^{n-2} + 2^{n-1}T(1) \\
&= 1 + 2 + 4 + \ldots + 2^{n-2} + 2^{n-1} = \sum_{i=0}^{n-1} 2^i \\
&= 2^n - 1
\end{aligned}$$

Page 19

Lecture - B8: Problem Complexity

2025-12-08 15:00 Janka

19.1 Introduction to Problems

No, this lecture is not talking about *that* pesky straight boy.

So far we have been studying *algorithmic complexity* (i.e. the complexity of merge sort, exchange sort, binary search, etc). More often, we need to study *problem complexity* (i.e. the complexity of sorting or searching in general).

For lots of problems, we do not know the problem complexity, however we do often have bounds on the complexity. We can often see that the complexity of a given problem falls between an *upper bound* and a *lower bound*.

19.1.1 Upper Bounds

The discovery of a better algorithm to solve a given problem brings the upper bound on the worst-case time complexity down. If we take a problem, Q, and someone comes along with algorithm A of complexity $O(n^3)$. Therefore we know that the problem complexity cannot be higher than $O(n^3)$.

Later on in time, someone might discover a better algorithm whose complexity is $O(n^2)$. This now means that the problem complexity cannot be higher than $O(n^2)$.

Each better algorithm brings the upper bounds downwards.

19.1.2 Lower Bounds

Computer Scientists prove statements about the lower bound. If we take that the problem Q cannot be solved in less than linear time, $\Omega(n)$. They prove that no algorithm could possibly exist that would solve this problem in less than $\Omega(n)$ time. Therefore the problem complexity cannot be lower than $\Omega(n)$.

However later someone finds that the problem cannot be solved in less than $\Omega(n \times \log n)$ time. Therefore the problem complexity cannot be lower than $\Omega(n \log n)$. Each better proof brings the lower bound upwards.

19.1.3 Unsolved Complexity

The complexity for many problems is still an open discussion.

In some situations, the lower bound (discovered by proof) and the upper bound (exhibited by an algorithm) will coincide. This shows us the complexity of the problem Q and tha the problem is close d as far as big-Theta time estimates are concerned. Regardless of this, we may continue to search for algorithms whose constant factors are smaller.

More often, however, there is a gap between the lower and upper bound. For example we may find a proof that no algorithm can solve Q in less than $\Omega(n \log n)$ time and that the best algorithm we know takes $O(n^2)$.

Where we find a gap like this later example, we should seek both better proofs and algorithms to close the gap and find the complexity for a problem.

Example: Problem Complexities

Ex. 1: Searching an unordered list of n items

- Upper bound: $O(n)$ comparisons (from linear search)
- Lower bound: $\Omega(n)$ comparisons

As there is no gap, we know the problem complexity is $\Theta(n)$.

Ex. 2: Searching an ordered list of n items

- Upper bound: $O(\log n)$ comparisons (from binary search)
- Lower bound: $\Omega(\log n)$ comparisons (from decision trees)

As there is no gap, we know the problem complexity is $\Theta(\log n)$.

Ex. 3: Sorting n arbitrary elements

- Upper bound: $O(n \times \log n)$ comparisons (from merge sort)
- Lower bound: $O(n \times \log n)$ comparisons

As there is no gap, we know the problem complexity is $\Theta(n \times \log n)$.

Ex. 4: Towers of Hanoi

- Upper bound: $O(2^n)$ moves
- Lower bound: $\Omega(2^n)$ moves

As there is no gap, we know that the problem complexity is $\Theta(2^n)$.

19.1.4 Proving Lower Bounds

Proofs of lower bounds are hard to come up with. This is one of the main reasons why there are gaps for so many problems.

A useful method for proving a lower bound as $\log n$ for searching for an item x in an ordered array S of n items is based on decision trees. This method can also be used for solving other problems and providing a lower bound $n \log n$ for sorting.

The binary search algorithm has a complexity of $O(\log n)$, so the complexity of the problem "searching an ordered list" is at most $O(\log n)$. Using the decision trees it is already a lower bound for the problem.

19.2 Decision Trees

A *decision tree* can be used to represent the process that takes place in an algorithm. They are trees, as we became familiar with in DMAFP in second year. The internal nodes represent decision point sin the algorithm (a query) and the leaves represent possible outcomes (an output).

Decision trees can be useful in trying to construct an algorithm or trying to find properties of algorithm, for example lower bounds may equate to the depth of a decision tree. If every query only has two possible outcomes - the decision tree is a *binary decision tree*.

For a given leaf on a decision tree, the number of decisions is the number of internal nodes from the root to the leaf. The worst case scenario corresponds to the height of the tree, so the worst case running time is just the height of the tree. We may sometimes want to consider decision trees with hight a higher degree where every query has (at most) k different answers, for a given constant k.

> There are some examples of decision trees in the slides available on Moodle.

The number of decisions in a decision tree is a lower bound on teh actual running time. This is good enough to prove a lower bound on the complexity of a given problem. If a problem has n different outputs, then any decision tree must have at least n leaves (it's possible for several leaves to specify the same output).

Harking back to DMAFP, the number of leaves in a binary tree of height d is at most 2^d. Therefore if every query has at most possible two answers, then $2^d \geq n$, meaning that the height of the decision tree must be at least $\lceil \log_2 n \rceil = \Omega(\log n)$.

If we see a tree with 15 different outputs, the height of the decision tree must be at least $\lceil \log_2 15 \rceil = 4$. The best performance we can hope for on this problem comes from algorithms that carry out these comparisons, therefore the lower bound is $\Omega(\log n)$ and no algorithm could be better than this. We can also consider the binary search algorithm with the complexity $O(\log n)$. Finally, knowing all this - we can see that we have an algorithm whose performance is as good as the lower bound therefore the complexity of the problem is $\Theta(\log n)$.

19.3 Problem Classification

So far, we have focused on two large families of problems: decidable and undecidable.

19.3.1 Undecidable Problems

We know that undecidable problems, also known as decision problems, are problems where it has been proven that no algorithm can ever exist to return 'yes'/'no' answers. These include:

- the Halting problem;
- the Post Correspondence problem;
- Hilbert's Tenth problem;
- the (unbounded) Tiling problem;
- the total problem.

Some of these problems are partially decidable - meaning that there is an algorithm which returns 'yes' but might never return 'no'.

19.3.2 Decidable Problems: Overview

Decidable problems, those who always return 'yes'/'no' for any given input can be divided into three categories:

- Proven intractable - solvable, but impractical
- Apparently intractable - those which appear to be intractable but which could possibly be tractable, we're not sure...
- Tractable - practically solvable

19.3.3 Decidable Problems: Proven Intractable

A proven intractable problem is a problem where it has been proved that no polynomial algorithm for such a problem exists.

We can subdivide this category of decidable problems further.

19.3.3.1 Decidable Problems: Proven Intractable: Type 1

Type 1 Proven Intractable problems are those that require a non-polynomial amount of output. these problems ordinarily pose no difficulty. We are asking for more information than we could possibly use, the problem is not defined realistically.

Example: Type 1 Proven Intractable Problems

Ex. 1 Find all routes for the Travelling Salesman problem with lengths less than B (a given constant).

To solve this problem, we have to go through $(n-1)!$ such routes (cycles) which means that our request is not reasonable.

Ex. 2 Finding the number of steps for n disks in the Tower of Hanoi.

A simple recursive solution is known with $2^n - 1$ steps which is also a lower bound for the problem. This means that it is decidable (solvable) but intractable.

19.3.3.2 Decidable Problem: Proven Intractable: Type 2

Type 2 Proven Intractable problems are those who do not require a non-polynomial amount of output, but we can prove that the problem cannot be solved in polynomial time.

There are a few such problems of this type are known, which includes all undecidable problems.

In the 1950/60s the decidable problems were “artificially” constructed to have such properties. In the early 1970s, some natural decidable decision problems were proven to be intractable, for example Presburger Arithmetic.

Example: Type 2 Proven Intractable Problems

We can see this type of problem through Draughts. For a $n \times n$ draughts board with an arrangement of pieces, we are asked to determine whether there is a winning strategy (a sequence of moves) for White so that no matter what Black does, White is guaranteed to win.

This has been proven that any algorithm that solves this problem must have a worst-case running time that is at least 2^n.

19.3.4 Decidable Problems: Apparently Intractable

An apparently intractable problem is problem for which a polynomial time algorithm has never been found, but yet no one ever proved that such an algorithm is not possible.

Examples of these problems include:

- the Hamiltonian Cycle problem
- the Travelling Salesman problem
- bounded tiling problem
- the Minimum Colouring problem

Example: Hamiltonian Cycle Problem

In a graph, a Hamiltonian cycle is a path that passes through all the vertices of the graph, passing through each vertex exactly once and the path ends at the same vertex that the path starts at. It takes an input of a graph, G, and produces an output of 'YES' where G has a Hamiltonian cycle or 'NO' in all other cases.

The best known algorithm for this problem is *exponential* and nobody knows whether there exists a polynomial algorithm.

19.3.5 Decidable Problems: Tractable

A tractable problem is a problem for which we have found a polynomial time algorithm has been found. For example:

- sorting or searching for a key in an array
- matrix multiplication
- the Minimum Spanning Tree problem
- deciding whether a graph is Eulerian (has an Eulerian circuit)
- the shortest path between two vertices in a graph
- the Maximum Flow in a network

Example: Eulerian Circuit Problem

In a graph, an Eulerian circuit is a walk that passes through all the edges of the graph, traversing each edge exactly once and the walk edges at the same vertex where it started. It takes an input of a graph, G and produces an output of 'YES' where G is Eulerian and 'NO' otherwise.

Contrary to what one might think when approaching this problem, we don't need to brute force and check through all possible cycles (there would be about $(n!)^2$ of them). We can follow Euler's advice and see that a connected graph contains and Eulerian circuit if and only if the degree of each vertex is even (known as *Property X*).

There exists a simple polynomial algorithm for the decision problem - checking only the degree of each vertex *Property X* and similarly for the problem of finding the Eulerian cycle.

19.4 An Aside on Decision Problems

The previous section includes discussion on not only decision problems, but also the problem of sorting, or to find the maximum values. Which are obviously not decision problems...

Going forth, we will focus on *decision problems*, the problems whose outputs are just 'YES' or 'NO'.

Most problems can be re-written in decision form. Take the standard Travelling Salesman Problem:

"*Given n cities and the pairwise distances between them, find the shortest possible rount trip (visiting each vertex exactly once).*"

Which we can rewrite as a decision problem:

"*Given n cities, the pairwise distance between them and a constant $d > 0$. Is it possible to find round trip with the total length less than d?*"

See that it's now not asking us to solve the problem of TSP, rather it's asking us if we *can* solve the problem.

19.5 Complexity Classes

Decision problems can be separated into various complexity classes, where we only restrict on time complexity.

The basic complexity classes are P, NP, and $NP-$complete.

19.5.1 Class P

P is the set of all decision problems that can be solved by polynomial-time algorithms.

This means that, for any given problem who is a member of P, we know of an algorithm that solves any instance of size n in $O(n^k)$ time. From this we can deduce that P is just the set of tractable decision problems: the decision problems for which we have polynomial-time algorithms.

There are a number of different kinds of problem in P:

- Decide whether the maximum of n integers is larger than a given constant
- Decide whether a graph is Eulerian
- Determine if a key appears three or more times in a given list of integers
- Decision problems corresponding to optimisation problems for which we have found a polynomial time algorithm, for example the maximum flow problem

There are obviously some problems which are not in P. One type being the proven intractable or undecidable problems, as these are the only ones for which we know that there is no polynomial algorithm, for example the Halting problem.

19.5.2 Class NP

The class NP is the set of all problems that can be solved by non-deterministic algorithms in polynomial time. NP stands for *non-deterministically polynomial*.

A non-deterministic algorithm can allow at every possible step multiple continuation. For example - a man walking down a path and every time he steps further, he must pick which fork in the road he wishes to take; the path to his destination isn't clear.

A non-deterministic algorithm accepts an input, x, if there exists a sequence of choices for the algorithm that returns a 'YES'. Note that the formal definition of NP says nothing about the running time of producing 'NO' answers.

We can consider a two-stage solution:

Guessing Stage (non-deterministic part) Make a guess at a solution (choices of continuations), either by plucking from thin air or using our special THEOC magic wands

Verification Stage A deterministic algorithm checks if the 'certificate' produced by the above stage is a solution in polynomial time, definitely halting in every true case

Example: Decision Version of TSP

The decision version of the Travelling Salesman Problem is NP.

We can see it's inputs: a graph with the pairwise distances between n vertices and a value d.

We can see it's outputs: Is it possible to find a complete round trip (visit each vertex exactly once) in the total length less than d?

If someone claimed that they have a solution to a given instance, can we verify it in polynomial time? In other words, and a given number d, can we test if a particular tour has the length less or equal to d?

Yes, we can. An algorithm for checking is shown below:

```
function verify(G: graph; d: number; S: claimed-tour): boolean;
begin
    if S is a tour and length-of-tour(S) <= d then
        verify:= true
    else
        verify:= false
end;
```