

TSP: Text Searching and Processing

Important Notes About These Notes

These notes were written by me, Mark Ormesher, during my revision for Computer Science exams at King's College London. I am publishing them here to help my classmates and other students in the years below me. To the best of my knowledge they are correct.

These notes are **not endorsed** by King's College London, the module lecturers, or any member of College staff. These notes are **not checked** by any qualified individual, and I offer no guarantee as to how accurate or complete these notes are. They are offered for free, as-is, and I accept no responsibility for how you choose to use them.

These notes were relevant for my year (2016/17) but **the content for your course may be different**. Check your lecture slides, syllabi, tutorial guides, etc.

These notes were produced for my own personal use (it's part of how I study and revise). That means that some annotations may be irrelevant to you, and **some topics might be skipped** entirely.

Feel free to **share** these notes, however please only share a link to the repo (see the link below), not individual files.

Notes are originally from **<https://github.com/markormesher/CS-Notes>**. All original work is and shall remain the copyright-protected work of Mark Ormesher. Any excerpts of other works, if present, are considered to be protected under a policy of fair use.

Contents

1	Important Notes About These Notes	1
2	Terms and Definitions	4
2.0.1	Alphabets	4
2.0.2	String Basics	4
2.0.3	Identity Between Strings	4
2.0.4	Concatenation/Product of Strings	4
2.0.5	Factor/Substring of a String	4
2.0.6	Superstring of a String	4
2.0.7	Suffix and Prefix of a String	4
2.0.8	Occurrence of a String	5
2.0.9	Power of a String	5
2.0.10	Primitivity of Strings	5
2.0.11	String Roots and Exponents	5
2.0.12	Conjugate Strings	5
2.0.13	Periods of a String	6
2.0.14	Borders of a String	6
2.0.15	Relationship Between Periodicity and Borders	6
3	String Matching	7
3.1	Notation	7
3.1.1	Online and Offline Algorithms	7
3.2	Näive Matching	7
3.3	Morris-Pratt (MP) Matching	7
3.3.1	MP Pre-processing	9
3.4	Knuth-Morris-Pratt (KMP) Matching	9
3.4.1	KMP Pre-processing	10
3.5	Runtime of MP and KMP Matching	11
4	Matching with Automata	12
4.1	Designing Automata to Match Periodic Strings	12
4.2	Designing Substring-Matching Automata (SMAs)	12
4.2.1	Matching Using SMAs	13
4.2.2	Systematic Construction	13
4.2.3	Systematic Construction Example	14
4.3	Matching n^{th} Character from the Left and Right	15
5	Dictionary Matching	16
5.1	Tries of Dictionaries	16
5.2	Dictionary Matching Automata (DMAs)	16
5.2.1	Matching Using DMAs	17
5.3	DMAs With Failure Links	17
5.3.1	Matching Using DMAs with Failure Links	18
5.3.2	Computing Failure Links	19
5.3.3	Optimising Failure Links	20
5.3.4	Delay	21
6	Searching in a List of Strings	22

6.1	Simple Searching Algorithm	22
6.2	Improving Search with LCPs	23
6.2.1	Side Note: Binary Search Tree	23
6.2.2	Notation	24
6.2.3	Case 1	24
6.2.4	Case 2	25
6.2.5	Case 3	25
6.2.6	Improved Search Algorithm	26
6.2.7	Improved Interval Algorithm	27
6.2.8	Preprocessing the List	27
7	Regular Expressions	28
7.1	Regular Expressions and Automata	28
7.1.1	Union of Two Automata	28
7.1.2	Concatenation of Two Automata	29
7.1.3	Kleene Star of an Automaton	29
7.1.4	Compliment of an Automata	29
7.1.5	Intersection of Two Automata	30
7.2	Example Automata Questions	30
7.2.1	Given a finite automaton M and a string w , check $w \in L(M)$	30
7.2.2	Given a finite automaton M , check $L(M) = \emptyset$	30
7.2.3	Given a finite automaton M , check $L(M) = \sigma^*$	30
7.2.4	Given two finite automata M_1 and M_2 , check $L(M_1) \subseteq L(M_2)$	30
7.2.5	Given two finite automata M_1 and M_2 , check $L(M_1) = L(M_2)$	30
7.3	Checking if a Language is Regular	30
7.3.1	The Pumping Lemma	31
7.3.2	Example 1: Proof by Contradiction	31
7.3.3	Example 2: Proof by Contradiction	31
7.3.4	Example 3: Proof by Restriction and Contradiction	32
7.3.5	Example 4: Proof by Restriction and Contradiction	32
7.3.6	Example 5: Proof by Restriction and Contradiction	33
7.3.7	Example 6: Proof by Restriction and Contradiction	33
7.3.8	Example 7: Proof by Restriction and Contradiction	34
7.4	Converting Regular Expressions to Automata	34
7.5	Automata to Regular Expressions	35

Terms and Definitions

Alphabets

An alphabet Σ is a finite, **non-empty** set of elements called **letters**.

String Basics

A string on an alphabet Σ is a finite, **possible empty** sequence of letters from Σ .

The **empty string** is denoted by ϵ .

The set of **all possible strings** on an alphabet Σ is denoted by Σ^* .

The **length** of a string x is the length of its sequence of letters, denoted by $|x|$.

The letter at the **i -th position** in the string x is denoted by $x[i]$, for all $0 \leq i < |x| - 1$.

Identity Between Strings

Two strings x and y are **identical** if and only if $|x| = |y|$ and $x[i] = y[i]$ for all $0 \leq i < |x| - 1$.

Concatenation/Product of Strings

The concatenation or product of two strings x and y is the string formed from all letters of x followed by all letters of y . It is denoted by xy .

Factor/Substring of a String

A string x is a factor or substring of a string y if two strings u and v exist such that $y = uxv$. Note that u and/or v may be the empty string ϵ .

A factor or substring x of a string y is **proper** if $x \neq y$.

Superstring of a String

A string x is a superstring of a string y if two strings u and v exist such that $x = u y v$. Note that u and/or v may be the empty string ϵ .

Suffix and Prefix of a String

Let the strings x, y, u and v exist such that $y = uxv\dots$

- If $u = \epsilon$ then x is a prefix of y .
- If $v = \epsilon$ then x is a suffix of y .

Occurrence of a String

If x and y are two strings and $x \neq \epsilon$, we can say that x **occurs in** y if x is a factor of y .

Every occurrence of x can be identified by a position within y . We say that x occurs in y starting at position i if $y[i \dots i + |x| - 1] = x$.

It is sometimes more useful to refer to the end position of the occurrence, $i + |x| - 1$.

Example, in the string `abracadabra`, the string `abr` occurs starting from positions 0 and 7.

Power of a String

For a string x and a natural number n , the n -th power of x , denoted by x^n , is defined as follows:

- $x^0 = \epsilon$
- $x^n = x^{n-1}x$

Example: if $a = \text{ha}$ then $a^4 = \text{hahahaha}$.

If two strings x and y and two natural numbers n and m exist such that $x^m = y^n$ then x and y are powers of some string z .

Primivity of Strings

A string is **primitive** if it is not the power of any other string.

A string x is **primitive** if and only if it exists as a factor of x^2 as a prefix and a suffix.

Example: given $x = \text{abaab}$ and $x^2 = \text{abaababaab}$, the string x is primitive because x only appears at the very start and very end of x^2 .

String Roots and Exponents

If $x \neq \epsilon$, a **primitive** string z and natural number n exist such that $x = z^n$. z is the **root** of x and n is the **exponent** of x . If x is primitive then $z = x$ and $n = 1$.

Example: $x = \text{abab}$, $z = \text{ab}$, $n = 2$.

Example: $x = \text{abcd}$, $z = \text{abcd}$, $n = 1$.

Conjugate Strings

Two non-empty strings x and y are conjugate if two strings u and v exist such that $x = uv$ and $y = vu$.

Example: `goldfish` and `fishgold` are conjugate.

Two non-empty strings are conjugate if and only if their **roots are also conjugate**.

Two non-empty strings are conjugate if and only if a string z exists such that $xz = zy$ (for the above example, $z = \text{gold}$).

Periods of a String

For a non-empty string x , an integer p such that $0 < p \leq |x|$ is a period of x if $x[i] = x[i + p]$ for all $0 \leq i < |x| - p$.

A string may have multiple periods, and every string has at least one period (the length of the string itself). We define *the* period of a string as its **smallest** period, denoted by $\text{per}(x)$.

Example: the string $x = \text{aababaaa}$ has periods of 3, 6, 7 and 8, and $\text{per}(x) = 3$.

Borders of a String

For a non-empty string x , a border is simultaneously a **proper factor**, **prefix** and **suffix** of x .

We define *the* border of a string as its **longest** border. $\text{border}(x)$ denotes the **length** of the longest border of x .

Example: the string $x = \text{aababaaa}$ has borders of ϵ , a , aa and aabaa , so $\text{border}(x) = 5$.

Relationship Between Periodicity and Borders

For any non-empty string x it holds that $\text{per}(x) + \text{border}(x) = |x|$.

String Matching

Objective: given two strings, *pattern* and *text*, find the **starting positions of all occurrences** of *pattern* within *text*.

Example: within the text `abracadabra`, the pattern `abr` starts at positions 0 and 7.

Notation

- The **pattern** is denoted as x with length m .
- The **text** is denoted as y with length n .

Online and Offline Algorithms

- In an online algorithm, only the **pattern** is known and can be pre-processed.
- In an offline algorithm, only the **text** is known and can be pre-processed.

Näive Matching

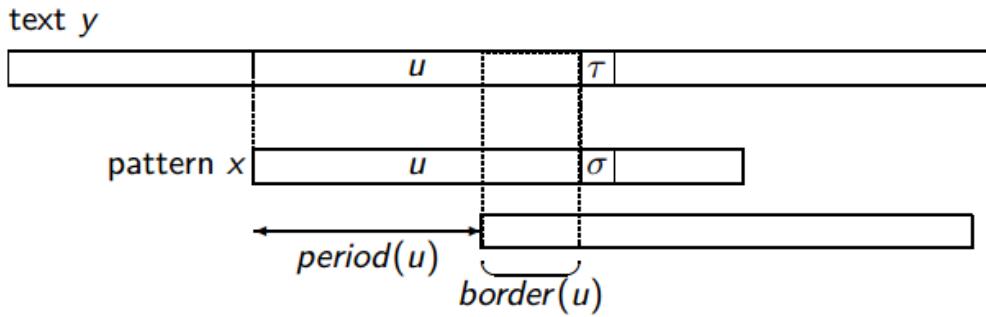
Matches are found by comparing the pattern to the text string repeatedly, incrementing the starting character each time. In the example below, the text is `abracadabra`, the pattern is `abr`, and matching portions are shown in red.

Index i :	0	1	2	3	4	5	6	7	8	9	10
Text y :	a	b	r	a	c	a	d	a	b	r	a
Successful try from 0:	a	b	r								
Failed try from 1:	a	b	r								
Failed try from 2:		a	b	r							
Failed try from 3:			a	b	r						
Failed try from 4:				a	b	r					
Failed try from 5:					a	b	r				
Failed try from 6:						a	b	r			
Successful try from 7:						a	b	r			
Failed try from 8:							a	b	r		

The runtime of this algorithm is $O(nm)$ in the worst case; this is **too slow**.

Morris-Pratt (MP) Matching

This algorithm improves the runtime by ‘shifting’ the pattern along the text by **more than one position** when a mismatch occurs.



In the diagram above, u shows the **prefix** of x that has been successfully matched within the text y ; the mismatch is caused by the non-equal characters τ and σ .

When the mismatch occurs, it is safe to move x along so that it lines up with the **end border** of u and resume searching from the position of τ (because we already know that the portion within the border already matches). The size of this movement is $period(u)$, because $|u| - border(u) = period(u)$.

The general structure of the algorithm is as follows:

- Memorise the borders of all prefixes of the pattern.
- Perform straightforward online search.
- Shift by $period(u)$ when a mismatch occurs.
- Resume scanning from $border(u)$.

```

1  fun MP_MATCHING(string y, string x, int n, int m):
2
3      mpNext = COMPUTE_MP_NEXT(x, m)
4
5      i = 0 // index of current comparison, relative to pattern x
6      j = 0 // start of window in text y
7
8      while (j < n) do:
9          while (i >= 0 and x[i] != y[i]) do:
10             i = mpNext[i]
11             i++
12             j++
13             if (i >= m):
14                 output(j - i)
15                 i = mpNext[i]
16
17     return matches

```

In the above code, $mpNext[i]$ is the length of the longest border of the first i characters of the pattern x .

MP Pre-processing

It is possible to efficiently compute the border size of pattern prefixes using one key property: **a border of a border of u is also a border of u** . Using this, the algorithm will compute the array border (a.k.a mpNext) such that $\text{border}[i]$ is the length of the longest border of the first i characters of the pattern x .

After defining $\text{border}[0] = -1$ for the empty string, for each x -prefix of length $1 \leq i \leq m$ the algorithm starts by assuming the length of the previous border. If the i^{th} character matches then the while loop is skipped and the border length is increased by 1. When a mismatch occurs, the algorithm tries successively shorter ‘borders of borders’ until a match is found.

```

1  fun COMPUTE_MP_NEXT(string x, int m) :
2      border[0] = -1
3
4      for (i from 1 to m) do:
5          j = border[i - 1]
6          while (j > 0 and x[i - 1] != x[j]) do:
7              j = border[j]
8          border[i] = j + 1
9
10     return border

```

Note: $\text{border} = \text{mpNext}$.

In the outer loop j is only incremented exactly m times. In the inner loop j can only decrease but $j \geq 0$ at all times. This means that the maximum amount of ‘travel’ for the value of j is $2m$ and therefore the algorithm’s runtime is $O(m)$.

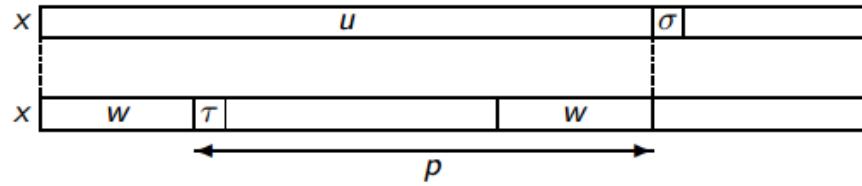
Knuth-Morris-Pratt (KMP) Matching

This algorithm improves slightly on the runtime of the MP matching algorithm ([see more: Morris-Pratt \(MP\) Matching, page 7](#)) by changing only the pre-processing stage to give slightly longer shifts.

The improvement relies on **strict borders** and their corresponding **interrupted periods**.

w is a strict border of the prefix u if:

- w is a border of u , and
- $w\tau$ is a prefix of x but $u\tau$ is not.



In the diagram above, w is a strict border of the prefix u because $\tau \neq \sigma$. Therefore p is the interrupted period of u .

KMP Pre-processing

This algorithm assumes that $mpNext$ is already available. Assuming $k = mpNext[i]$, it is defined as follows:

$$kmpNext[i] = \begin{cases} k & x[i] \neq x[k] \text{ or } i = m \\ kmpNext[k] & x[i] = x[k] \end{cases}$$

This states that $kmpNext[i]$ is equal to $mpNext[i]$ when there is a mismatch between the next character after the prefix and the next character after the border (i.e. a strict border). If the letters do match then the strict border of the border is taken.

```

1 | fun COMPUTE_KMP_NEXT(string x, int m):
2 |
3 |     kmpNext[0] = -1
4 |     k = 0 // k will be mpNext
5 |
6 |     for (i from 1 to m - 1) do:
7 |
8 |         if (x[i] == x[k]):
9 |             kmpNext[i] = kmpNext[k]
10 |         else:
11 |             kmpNext[i] = k
12 |             do:
13 |                 k = kmpNext[k]
14 |                 while (k >= 0 and x[i] != x[k])
15 |
16 |             k++
17 |
18 |     kmpNext[m] = k

```

As before, in the outer loop k is only incremented exactly m times. In the inner loop k can only decrease but $k \geq 0$ at all times. This means that the maximum amount of 'travel' for the value of k is $2m$ and therefore the algorithm's runtime is $O(m)$.

Runtime of MP and KMP Matching

On a text of length n , the MP and KMP algorithms have runtimes of $O(n)$. In particular, they make no more than $2n$ character comparisons. This is because positive comparisons will increase the value of j (which ranges from 0 to n and does not decrease), and negative comparisons will increase the value of the starting position $j - i$ (which also ranges from 0 to n and does not decrease).

Including the pre-processing, the runtime of both algorithms is $O(n + m)$.

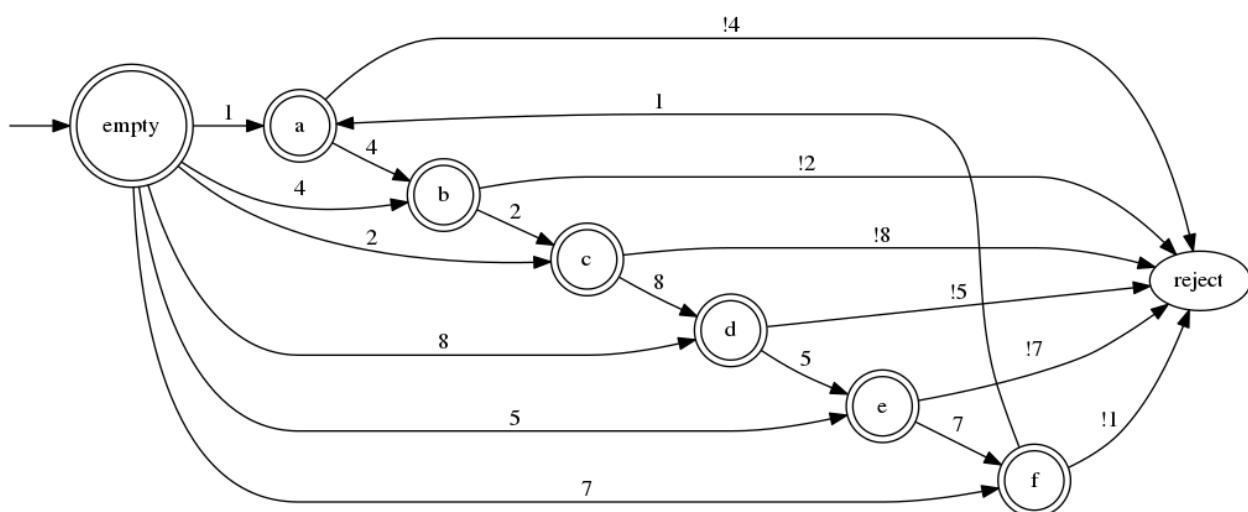
Matching with Automata

Designing Automata to Match Periodic Strings

A **non-empty string** is periodic if it has the form $uu\dots uu'$, such that u' is a prefix of u . For example, the string $abbabba$ b is periodic, where $u = abb$ and $u' = ab$ (see more: *Periods of a String, page 6*).

Automata can be easily designed to **accept any substring of a repeating string**, including the empty string ϵ .

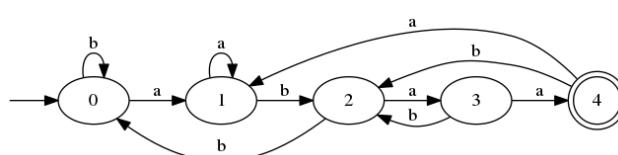
For example, the following automaton operates over the alphabet $\Sigma = \{1, 2, 4, 5, 7, 8\}$ and accepts any substring of the decimal expansion of $\frac{1}{7} = 0.142857142857\dots$ (ignoring the 0. prefix).



Designing Substring-Matching Automata (SMAs)

String-matching automata (SMAs) should reach an accepting state every time a given pattern is encountered as a sub-string of the input. For example, with the pattern abb and the input $cabbabbb$, the automaton should reach an accepting state after processing characters 3 and 6 (note: zero-indexed).

For example, the automaton below accepts $u = abaa$ in the alphabet $\Sigma = \{a, b\}$.



Text y	b	a	b	b	a	a	b	a	a	b	b	b	a	
State	0	0	1	2	0	1	1	2	3	4	2	3	4	2

Matching Using SMAs

A simple online parsing of the text y with $SMA(u)$ is sufficient:

```

1  fun SMA_MATCHING(string y, sma):
2      q = sma.initial_state
3
4      if (q is terminal):
5          report occurrence of pattern
6
7      while (y has remaining characters) do:
8          a = next character of y
9          q = q.successors[a]
10         if (q is terminal):
11             report occurrence of pattern

```

Systematic Construction

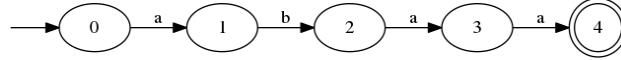
An SMA that matches u (where $|u| = m$) will contain $m + 1$ states numbered $0..m$. In general, if the automaton is in state k then the k most recently read characters match the first k characters of the pattern. The final state (numbered m) is the only accepting state.

The SMA can be constructed with the following process:

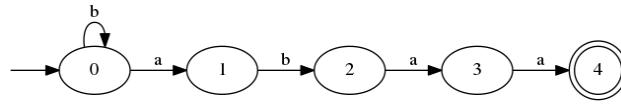
1. Create the initial $m + 1$ states, numbered $0..m$.
 - Declare state 0 as the initial state.
 - Declare state m as the accepting state.
2. For each $0 \leq k < m$, create a forwards arc from state k to state $k + 1$ labelled by $u[k]$.
 - This is the successful path that simply spells u .
3. At each state, create a backwards arc for each character in Σ that breaks the pattern u .
 - The backwards arc should lead to state k , where the k most recently read characters match the first k characters of the pattern.
 - Another way of stating this is to find the longest suffix of processed input that matches a prefix of u . The length of this suffix/prefix is k .
 - k can be found by shifting the pattern along the read characters to the right, until a match occurs. An example follows.

Systematic Construction Example

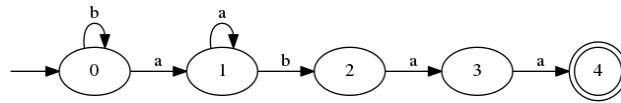
Consider the alphabet $\Sigma = \{a, b\}$ and the pattern $u = \text{abaa}$. After steps 1 and 2, the following SMA will be created:



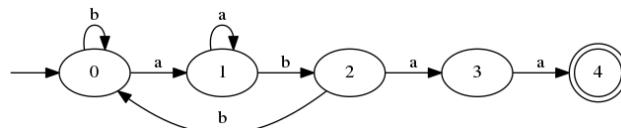
Applying step 3 to state 0, a break in the pattern occurs after reading b . This shares no characters with the start of u , so the arc should lead to state 0:



Applying step 3 to state 1, a break in the pattern occurs after reading aa . The longest suffix of aa that matches a prefix of abaa is a , so the backwards arc leads to state 1:



Applying step 3 to state 2, a break in the pattern occurs after reading abb . No suffix of abb matches a prefix of abaa , so the backwards arc leads to state 0:

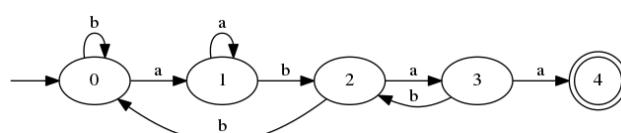


Applying step 3 to state 3, a break in the pattern occurs after reading $abab$. The longest suffix of $abab$ that matches a prefix of abaa is ab , so the backwards arc leads to state 2. This can be seen by shifting the pattern to the right along the input:

1 | Input: abab
2 | Pattern: abaa

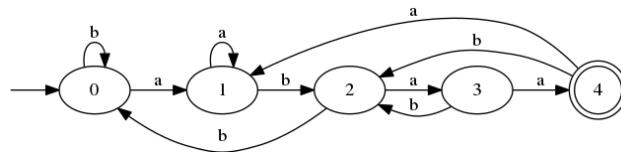
1 | Input: abab
2 | Pattern: abaa

1 | Input: abab
2 | Pattern: abaa
3 | ^ Match!



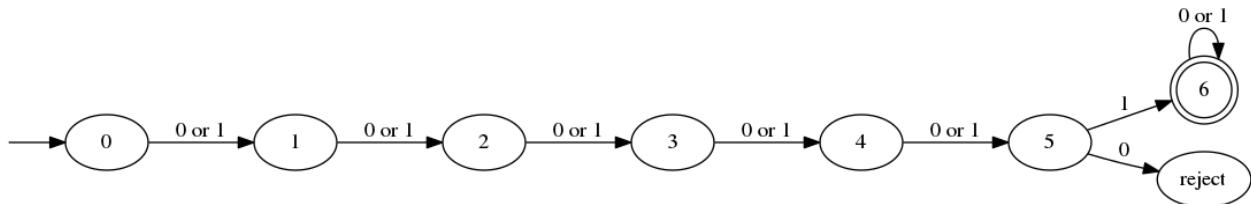
Finally, at state 4, backwards arcs for a and b need to be created.

The longest suffix of abaaa matching a prefix of abaa is a, so a backwards arc for a leads to state 1. The longest suffix of abaab matching a prefix of abaa is ab, so a backwards arc for b leads to state 2:

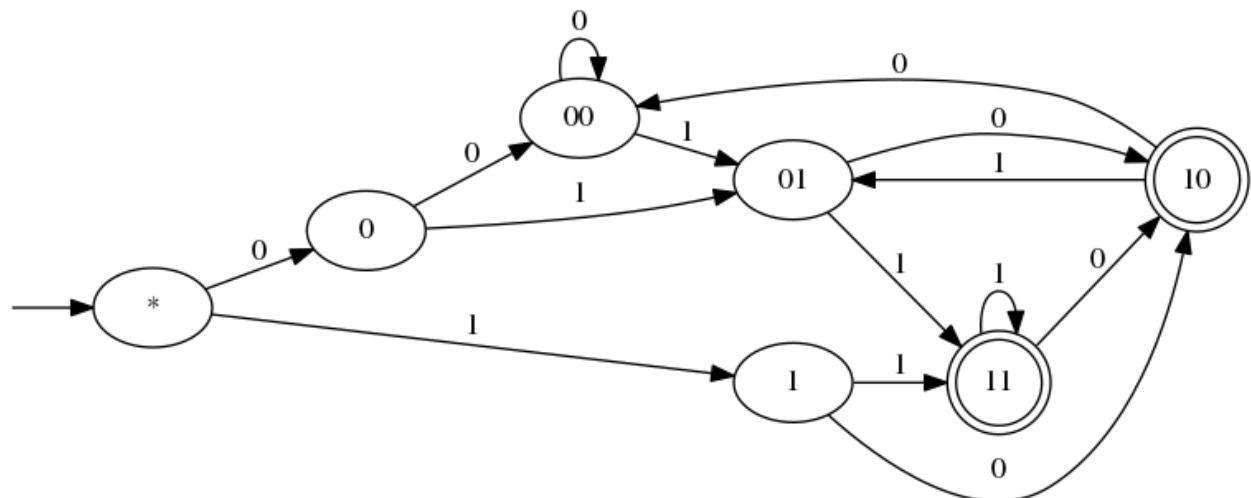


Matching n^{th} Character from the Left and Right

Matching **from the left** is trivial: the first $n - 1$ character are skipped, and the n^{th} character must match. For example, the following automaton accepts any string from $\Sigma = \{0, 1\}$ where the 6th character from the left is 1:



Matching **from the right** is more complex: the automaton must be ‘tricked’ into keeping a memory. This can be achieved by making a state for every combination of the n right-most characters and accepting states that begin with the required character. For example, the following automaton accepts any string from $\Sigma = \{0, 1\}$ where the 2nd character from the left is 1:



Dictionary Matching

Formally, a **dictionary** is a set of strings $X = \{x_0, x_1, \dots\}$ such that $\epsilon \notin X$.

Objective: given a **dictionary** of strings X and a text y , find all occurrences of strings from the dictionary within y . The output should be the **list of positions** in y that are **end positions** of some string in X .

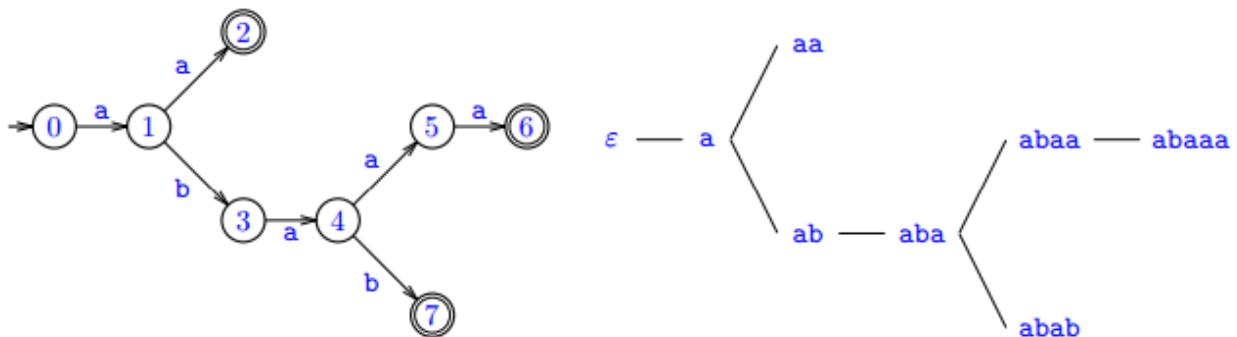
Example: if $X = \{\text{abc}, \text{cba}\}$ and $y = \text{aabcbabc}$, then the output should be $[3, 5, 7]$.

The standard approach for this is to use a **dictionary matching automaton (DMA)**.

Tries of Dictionaries

A trie $T(X)$ is a tree whose leaves are labelled by strings of X . Internal nodes are prefixes of strings in X . As an automaton, the trie $T(X)$ accepts all strings in X .

For example, $T(\{\text{aa}, \text{abaaa}, \text{abab}\})$:



Dictionary Matching Automata (DMAs)

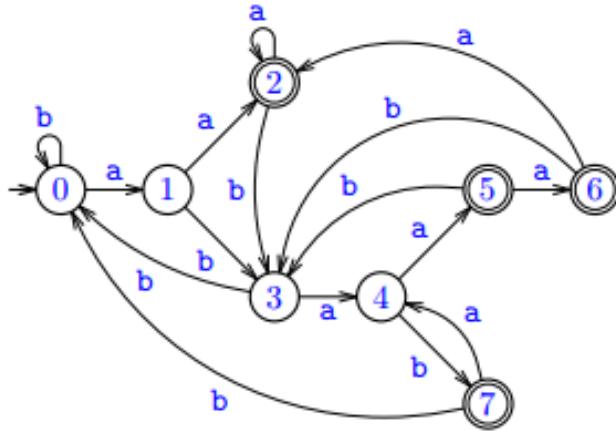
DMAs reach an accepting state every time a word from the dictionary is read in full (very similar to the SMAs from earlier). Formally, a DMA for the dictionary X (denoted $D(X)$) operates over the alphabet A and accepts any word in the form A^*X (i.e. any string from the language, suffixed with a string from X).

A dictionary trie forms the basis of a DMA. DMAs are constructed as follows:

- The set of states is $\text{pref}(X)$.
 - i.e. all prefixes of strings in the dictionary X .
- The initial state is the empty string ϵ .
- The set of terminal/accepting states is $\text{pref}(X) \cap A^*X$.
 - i.e. all prefixes that are suffixed with strings from X .

- Edges are in the form $\langle u, a, h(ua) \rangle$, where u is the text read so far, a is the character labelling the edge, and $h(ua)$ is the longest suffix of ua that belongs to $\text{pref}(X)$.

For example, $D(\{\text{aa}, \text{abaaa}, \text{abab}\})$:



$0 = \epsilon$, $1 = a$, $2 = aa$, $3 = ab$ etc.

Note: state 5 = abaa is an accepting state because abaa is suffixed by aa, which is in the dictionary.

The size of DMAs is $O(|A| \cdot \Sigma(|x| : x \in X))$ (i.e. one arc for each letter in the alphabet, for each of the possible prefix states).

Matching Using DMAs

Almost identical to the SMA searching algorithm ([see more: Matching Using SMAs, page 13](#)).

```

1  fun DMA_MATCHING(string y, dma):
2      q = dma.initial_state
3
4      for each character a in y, do:
5          q = q.successors[a]
6          if (q is terminal):
7              report occurrence of dictionary string

```

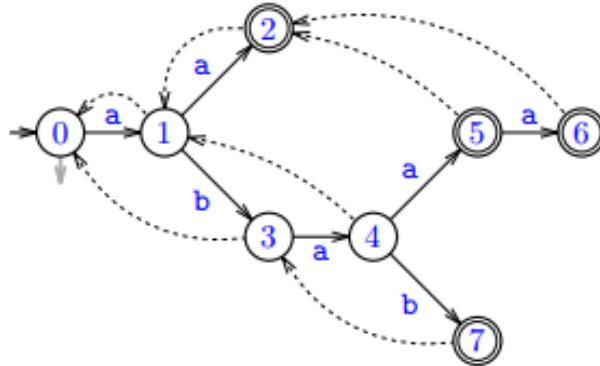
The running time of this matching process on a text y is $\Theta(|y|)$.

DMAs With Failure Links

This adaptation of DMAs reduces the size to $O(\Sigma(|x| : x \in X))$, independent of the alphabet size. It works by keeping the ‘forward’ arcs between prefixes and replacing all ‘backward’ arcs with one **failure link** per prefix node.

Failure links are determined by the **failure function** f , such that $f(u) = \text{the longest proper suffix of } u \text{ that is in } \text{pref}(X)$. The failure link for a node labelled by u leads to $f(u)$.

For example, $D(\{\text{aa, abaaa, abab}\})$:



This can be represented with a compact successor table, as below. Note that the 'label' column is not needed and is only for clarity.

State	Label	Successors	Failure
0	ϵ	($a, 1$)	<i>nil</i>
1	a	($a, 2$), ($b, 3$)	0
2	aa		1
3	ab	($a, 4$)	0
4	aba	($a, 5$), ($b, 7$)	1
5	abaa	($a, 6$)	2
6	abaaa		2
7	abab		3

Matching Using DMAs with Failure Links

This matching process works as before when successors (forward arcs) exist, but when no successors exist the algorithm will **follow failure links** until it reaches a state containing an appropriate successor (or the initial state).

```

1  fun DMA2_MATCHING(string y, dma):
2      q = dma.initial_state
3
4      for each character a in y, do:
5          q = TARGET_BY_FAILURE(q, a, dma)
6          if (q is terminal):
7              report occurrence of dictionary string
  
```

```

1 | fun TARGET_BY_FAILURE(p, a, dma):
2 |     while (p != nil and p.successors[a] = nil) do:
3 |         p = p.failure
4 |
5 |     if (p == nil):
6 |         return dma.initial_state
7 |     else:
8 |         return p.successors[a]

```

For example, using the DMA with failure links above and the text abbabababbabb:

Text y	a	b	b	a	b	a	a	b	a	b	a	b	b	
State	0	1	3	0, 0	1	3	4	5	2, 1, 3	4	7	3, 4	7	3, 0, 0

Computing Failure Links

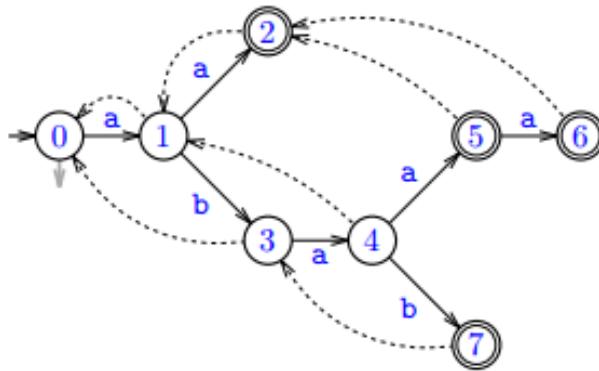
Failure links can be efficiently computed via a **breadth-first traversal** of the trie, as described below. Breadth-first traversal is efficient because failure links for a given state can be computed using the failure links for 'earlier' states, which will have already been computed.

```

1 | fun BUILD_DMA_WITH_FAILURE_LINKS(X):
2 |     M = BUILD_TRIE(X)
3 |     fail[M.initial_node] = nil
4 |
5 |     Q = empty queue
6 |     Q.enqueue(M.initial_state)
7 |     while (Q is not empty) do:
8 |         t = Q.dequeue()
9 |
10 |         for each pair (a, p) in t.successors, do:
11 |
12 |             // successor failure is built on the parent's failure
13 |             r = TARGET_BY_FAILURE(fail[t], a)
14 |             p.failure = r
15 |
16 |             // if the failure goes to a terminal state,
17 |             // the success becomes terminal
18 |             if (r.terminal):
19 |                 p.terminal = true
20 |
21 |             Q.enqueue(p)
22 |
23 |     return M

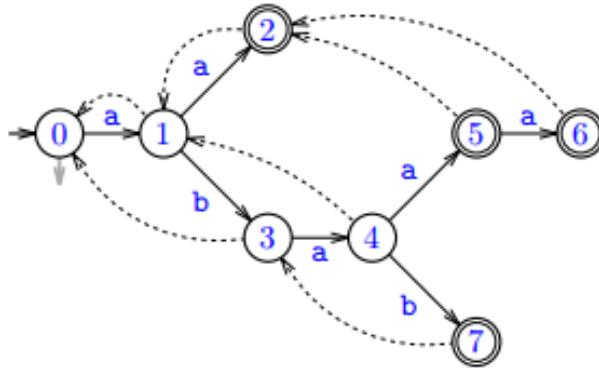
```

For example, using $D(\{aa, abaaa, abab\})$:



- State 5 is a child of state 4 along an arc labelled a.
 - $fail[4] = 1$.
 - $TARGET_BY_FAILURE(1, a) = 2$, so $fail[5] = 2$.
 - State 2 is terminal, so state 5 is marked terminal.
- State 6 is a child of state 5 along an arc labelled a.
 - $fail[5] = 2$.
 - $TARGET_BY_FAILURE(2, a) = 2$, so $fail[6] = 2$.

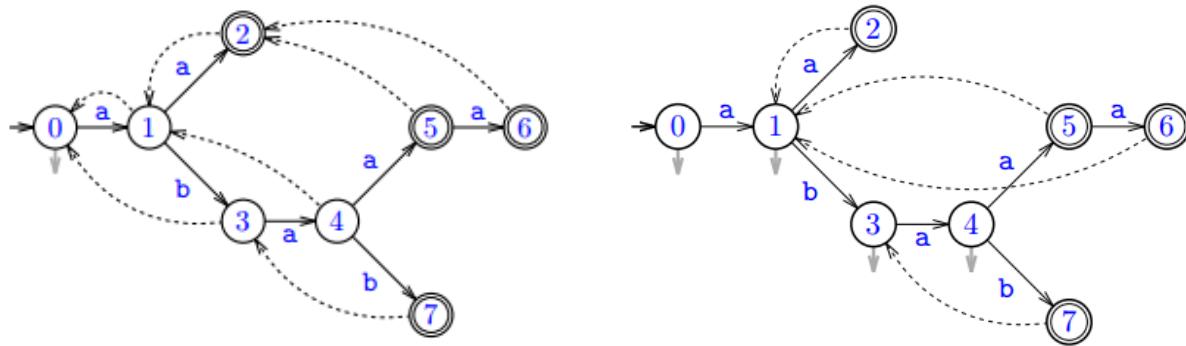
Optimising Failure Links



In the example above, $fail[4] = 1$. However, the outgoing edges from state 1 are a subset of the outgoing edges from state 4, so any input that failed on state 4 is guaranteed to also fail on state 1.

Therefore, the initial set of failure links can be optimised: at a state k , follow failure links until a state is reached with outgoing edges that are not a subset of k 's outgoing edges (i.e. a state with 'new' outgoing options).

This is shown in the following example, where the optimised version is on the right:



Delay

The delay is the **maximum time spent on any letter** of the input y . Even with optimised failure links, it is $\max\{|x| : x \in X\}$.

Searching in a List of Strings

- Input:
 - A list L of n strings of Σ^* , sorted in lexicographic order.
 - A string $x \in \Sigma^*$ of length m .
- Simple searching output:
 - $x \in L$: a position i such that $0 \leq i < n$, where $L_i = x$.
 - $x \notin L$: positions d, f such that $d+1 = f$ and $-1 \leq d < f \leq n$, where $L_d < x < L_f$.
 - i.e. i is the position of x , or d and f are indexes of the strings either side of where x would be if it was in the list.
- Interval output:
 - Positions d, f such that $-1 \leq d < f \leq n$, where x is a prefix of L_i for all $d < i < f$.

For example:

	L_0	L_1	L_2	L_3	L_4	L_5	Simple Search:	Interval:
	aaabaa							
		aaabb						
$L =$			aabbba				$x = aaabb \rightarrow 1$	$x = aa \rightarrow (-1, 3)$
				ab				
					baaa		$x = aaba \rightarrow (1, 2)$	
						bb		

Simple Searching Algorithm

The simple search algorithm works in a similar fashion to **binary search** over a sorted array, using a 'mid-point' i and choosing to search in the first or second half of the remaining list. At each i the length of the **longest common prefix** (LCP) of x and L_i is recorded as l , then one of a few things can happen:

- If $|lcp(x, L_i)| = |L_i| = |x|$, the search term has been found so i is returned.
- If $|lcp(x, L_i)| = |L_i| \neq |x|$, the search term must exist in the second half of the list.
- If $|lcp(x, L_i)| \neq |x|$ and the next character after the LCP is lower in L_i than it is in x , the search term must exist in the second half of the list.
- Otherwise, the search term must exist in the first half of the list.

```

1  fun SIMPLE_LIST_SEARCH(L, n, x, m):
2      d = -1
3      f = n
4
5      while (d + 1 < f) do:
6          i = floor((d + f) / 2)
7          lcp = LONGEST_COMMON_PREFIX(x, L_i).length
8
9          if (lcp == L_i.length and lcp == m):
10             return i
11
12         else if (lcp == L_i.length):
13             d = i
14
15         else if (lcp != m and L_i[lcp] < x[lcp]):
16             d = i
17
18         else:
19             f = i
20
21     return (d, f)

```

The binary search takes $O(\log_2(n))$ and computing the LCP at each step takes $O(m)$, so this has a total running time of $O(m \cdot \log_2(n))$.

The worst case running time can be created with $L = \{a^{m-1}b, a^{m-1}c, a^{m-1}d, \dots\}$ and $x = a^m$.

Improving Search with LCPs

Objective: reduce running time to $O(m + \log_2(n))$.

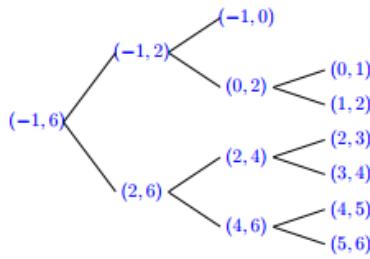
This improvement relies on $\text{lcp}(L_d, L_f)$ being known for any for (d, f) considered in the binary search. This requires $O(n)$ extra space to store the $2n + 1$ LCPs associated with the nodes in the binary search tree (see below).

The algorithm is based on properties that arise in three cases (and their symmetric cases).

Side Note: Binary Search Tree

- Nodes:
 - $n + 1$ external nodes, $(-1, 0), (0, 1), (1, 2), \dots, (n - 1, n)$.
 - n internal nodes in the form (d, f) with children $(d, \lfloor (d+f)/2 \rfloor)$ and $(\lfloor (d+f)/2 \rfloor, f)$.
 - Root of $(-1, n)$.
- Size: $2n + 1$ for a list of n strings.

Example for $n = 6$:



Notation

Throughout the algorithm...

- $ld = |lcp(x, L_d)|$
- $lf = |lcp(x, L_f)|$
- $i = \lfloor (d + f)/2 \rfloor$

Case 1

Hypothesis:

- $L_d < x < L_f$
- $ld \leq |lcp(L_i, L_f)| < lf.$

Example:

L_d	aaaca	$x = \text{aa}bbb\text{aa}$
	aaacba	
L_i	<u>aabba</u> ba	
	aabbabb	
L_f	<u>aabbb</u> ab	$x = \text{aabbb}\text{a}\text{a}$

Conclusion:

- $L_i < x < L_f$
 - x is in the **second half** of the list.
- $|lcp(x, L_i)| = |lcp(L_i, L_f)|$
 - See underlined portions above.

Case 2

Hypothesis:

- $L_d < x < L_f$
- $ld \leq lf < |lcp(L_i, L_f)|$.

Example:

L_d	<u>aaaca</u>	$x = \text{aabacb}$
	aaacba	
L_i	<u>aabbaba</u>	
	aabbabb	
L_f	<u>aabbbab</u>	$x = \text{aabacb}$

Conclusion:

- $L_d < x < L_i$
 - x is in the **first half** of the list.
- $|lcp(x, L_i)| = |lcp(x, L_f)|$
 - See underlined portions above.

Case 3

Hypothesis:

- $L_d < x < L_f$
- $ld \leq lf = |lcp(L_i, L_f)|$.

Example:

L_d	<u>aaaca</u>	$x = \text{aabbab}$
	aaacba	
L_i	<u>aabbaba</u>	
	aabbabb	
L_f	<u>aabbbab</u>	$x = \text{aabbab}$

Conclusion:

- Compare x and L_i from position lf .

Improved Search Algorithm

```

1  fun SEARCH(L, n, x, m, lcp):
2      d = -1, ld = 0
3      f = n, lf = 0
4
5      while (d + 1 < f) do:
6          i = floor((d + f) / 2)
7
8          if (ld <= lcp(i, f)) < lf):           // case 1
9              d = i
10             ld = lcp(i, d)
11
12         else if (ld <= lf < lcp(i, f)):    // case 2
13             f = i
14
15         else if (lf <= lcp(d, i) < ld):    // case 1 (symmetry)
16             f = i
17             lc = lcp(d, i)
18
19         else if (lf < ld < lcp(d, i)):    // case 2 (symmetry)
20             d = i
21
22         else:                                // case 3
23             li = max(ld, lf)
24             li = li + LCP(x[li...], L_i[li...]).length
25
26             if (li == L_i.length and li == m):
27                 return i
28
29             else if (li == L_i.length):
30                 d = i
31                 ld = li
32
33             else if (li != m and L_i[li] < x[li]):
34                 d = i
35                 ld = li
36
37             else:
38                 f = i
39                 lf = li
40
41     return (d, f)

```

The complexity of this algorithm is $O(m + \log_2(n))$, as explained:

- Each positive comparison increases l , which can happen no more than m times.

- Each negative comparison halves the value of $f - d$, giving no more than $\lceil \log_2(n + 2) \rceil$ comparisons.
- After preprocessing, LCP can run in constant time.

Improved Interval Algorithm

TODO: Improved Interval Algorithm

Preprocessing the List

Notation: $\|L\| = \sum_{i=0}^{n-1} |L_i|$ (i.e. the total length of all strings in the list).

- The list can be sorted in time $O(\|L\|)$ using repetitive bucket sorting.
- Computing LCPs for L_i and L_{i-1} for $0 \leq i \leq n$ can be done in time $O(\|L\|)$.
- Computing LCPs for other nodes inside the tree can be done in constant time:
 - Let $L_0 \leq L_1 \leq L_2 \leq \dots \leq L_n$.
 - Let $-1 < d < i < f < n$.
 - $|lcp(L_d, L_f)| = \min\{|lcp(L_d, L_i)|, |lcp(L_i, L_d)|\}$.

Therefore, the entire preprocessing stage can be done in time $O(\|L\|)$.

Regular Expressions

Regular expressions are used for **matching strings** or **substrings**. A regular expression is simply a string that is augmented with special characters that have specific meanings:

- (and) - used for grouping and/or nesting expressions.
- * (Kleene star) - indicates zero or more repetitions of the preceding expression.
- \cup (union) - indicates a choice.
- Two expressions side-by-side implies the concatenation of those expressions.

For example, the expression $1^*011^*(0 \cup 1)^*111$ will match and string with the pattern 'any number of 1s, then 01, then any number of 1s, then any number of 1s and 0, then 111'.

Regular Expressions and Automata

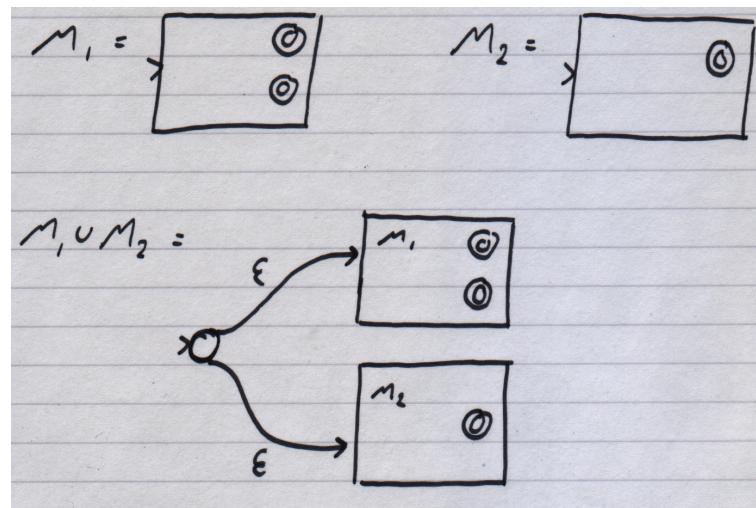
Regular expressions and automata are **equally powerful** and any one can be converted into the other (although not always easily). This is proved in the following sections, that show that every function of regular expressions can be implemented with automata.

Union of Two Automata

If M is the union of two automata, then M should accept any string that would be accepted by either of the two original automata.

Example: an automaton that accepts x when x has a substring ab or has a substring bbb .

- Create a new initial state.
- Create arcs from the new initial state to the previous initial states, labelled with ϵ .

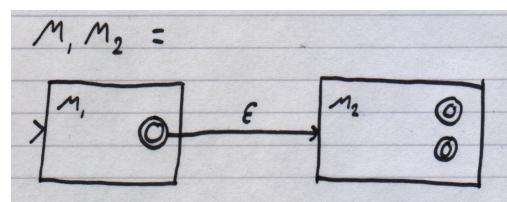


Concatenation of Two Automata

If M is the concatenation of two automata, then M should accept any string that can be broken into two strings, each of which satisfies the first and second original automata respectively.

Example: an automaton that accepts x when x is made up of a string that has ab as a substring, followed by another string that has bbb as a substring.

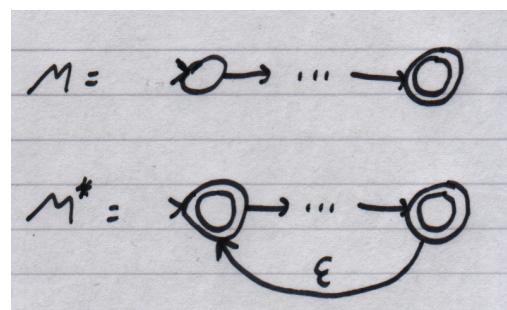
- Join the final state(s) of the first automaton to the initial state of the second automaton with an arc labelled by ϵ .



Kleene Star of an Automaton

If M is the Kleene star of an automaton, then M should accept any number of repetitions of a string that would be accepted by the original automaton.

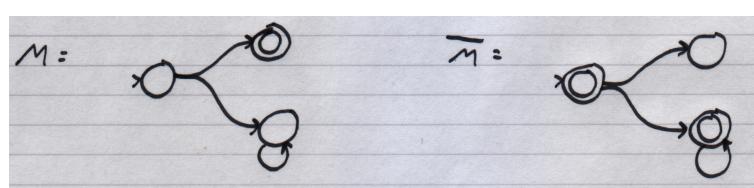
- Join the final state(s) of the automaton back to the initial state with arcs labelled by ϵ .
- Convert the initial state into a final/accepting state (to allow for zero repetitions).



Compliment of an Automata

If M is the compliment an automaton, then M should accept anything that would be rejected by the original automaton.

- Swap the type of all states, so that final states become non-final and vice versa.



Intersection of Two Automata

If M is the intersection an automaton, then M should accept anything that would be accepted by both of the original automata.

This can be derived from previous cases:

- Let M_1 and M_2 be the two original automata.
- Let $L(n)$ be the language accepted by some automaton, n .
- $L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}$

Example Automata Questions

Given a finite automaton M and a string w , check $w \in L(M)$.

Solution: trace execution of the automaton, letter by letter.

Given a finite automaton M , check $L(M) = \emptyset$.

Solution: check for a path from the initial state to any final state. If such a path exists, a string exists in the language that matches it; if there is no path, $L(M)$ is empty.

Given a finite automaton M , check $L(M) = \sigma^*$.

i.e. does it accept every possible string in the language?

Solution: yes, if $\overline{L(M)} = \emptyset$.

Given two finite automata M_1 and M_2 , check $L(M_1) \subseteq L(M_2)$.

Solution: yes, if $L(M_1) \cap \overline{L(M_2)} = \emptyset$.

Given two finite automata M_1 and M_2 , check $L(M_1) = L(M_2)$.

Solution: yes, if $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$.

Checking if a Language is Regular

A language regular if there exists an **automaton/regular expression that accepts it**.

The Pumping Lemma

If L is an **infinite**, regular language, then there exists strings x , y and z such that $y \neq \epsilon$ and $xy^n z \in L$ for any n .

If this lemma is violated then L cannot be an infinite regular language. Note that this only works one way: satisfying this lemma does not guarantee that L is infinite and regular.

Example 1: Proof by Contradiction

$$L = \{0^i 1^i \mid i > 0\}$$

We assume towards contradiction that the language L is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in L$.

- y could be a number of 0s.
 - This won't work, because xyz and xy^2z will have different numbers of 0s.
- y could be a number of 1s.
 - This won't work, because xyz and xy^2z will have different numbers of 1s.
- y could be a mixture of 0s and 1s.
 - This won't work, because any $n > 1$ is guaranteed to break the pattern of the language.

The pumping lemma cannot be satisfied, so therefore the assumption cannot hold.

Example 2: Proof by Contradiction

$$L = \{a^i aba^i \mid i > 0\}$$

We assume towards contradiction that the language L is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in L$.

- y could be a^n .
 - This won't work, because different values of n will create different, non-equal quantities of a on either side.
- y could be b .
 - This won't work, because there must be only one b .
- y could be a mixture of as and bs .
 - This won't work, because there must be only one b .

The pumping lemma cannot be satisfied, so therefore the assumption cannot hold.

Example 3: Proof by Restriction and Contradiction

$$L = \{ww \mid w \in \{a,b\}^*\}$$

The pumping lemma could be satisfied now: $x = a$, $y = aa$, $z = a$.

We can still prove that L is not an infinite regular language by using a known regular language to **create a restriction**, then using proof by contradiction on that. This works because the intersection of two languages will only be regular if both of the original languages were.

$$P = L \cap \{a^i ba^k b \mid i > 0\} = \{a^m ba^m b \mid m > 0\}$$

We assume towards contradiction that the language P is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in P$.

- y could be a number of a s.
 - This won't work, because varying values of n will change the number of a s on one side.
- y could be a number of b s.
 - This won't work, because there must be only one b on each side.
- y could be a mixture of a s and b s.
 - This won't work, because there must be only one b on each side.

The pumping lemma cannot be satisfied for P , so therefore the assumption cannot hold (and therefore both P and L are not infinite regular languages).

Example 4: Proof by Restriction and Contradiction

$$L = \{w\bar{w} \mid w \in \{a,b\}^*\}$$

$$P = L \cap \{a^k bb^l a \mid i > 0\} = \{a^m bb^m a \mid m > 0\}$$

We assume towards contradiction that the language P is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in P$.

- y could be a number of a s.
 - This won't work, because varying values of n will change the number of a s on one side.
- y could be a number of b s.
 - This won't work, because varying values of n will change the number of b s on one side.
- y could be a mixture of a s and b s.
 - This won't work, because only a few carefully chosen values of n will preserve the correct format.

The pumping lemma cannot be satisfied for P , so therefore the assumption cannot hold (and therefore both P and L are not infinite regular languages).

Example 5: Proof by Restriction and Contradiction

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

Note: w^R is the reverse of w .

$$P = L \cap \{a^k bba^l \mid i > 0\} = \{a^m bba^m \mid m > 0\}$$

We assume towards contradiction that the language P is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in P$.

- y could be a number of a s.
 - This won't work, because varying values of n will change the number of a s on one side.
- y could be b .
 - This won't work, because there must be only one b in each w .
- y could be a mixture of a s and b s.
 - This won't work, because there must be only one b in each w .

The pumping lemma cannot be satisfied for P , so therefore the assumption cannot hold (and therefore both P and L are not infinite regular languages).

Example 6: Proof by Restriction and Contradiction

$$L = \{w\bar{w}w \mid w \in \{a, b\}^*\}$$

$$P = L \cap \{a^j bba^k a^l b \mid i > 0\} = \{a^m bba^m a^m b \mid m > 0\}$$

We assume towards contradiction that the language P is an infinite regular language, and therefore strings x , y and z should exist such that $y \neq \epsilon$ and $xy^n z \in P$.

- y could be a number of a s.
 - This won't work, because varying values of n will change the number of a s in one section.
- y could be b .
 - This won't work, because there must be only one b in each w .
- y could be a mixture of a s and b s.
 - This won't work, because there must be only one b in each w .

The pumping lemma cannot be satisfied for P , so therefore the assumption cannot hold (and therefore both P and L are not infinite regular languages).

Example 7: Proof by Restriction and Contradiction

$L = \{x \in \{0, 1\}^* \mid \text{equal number of } 1\text{s and } 0\text{s}\}$

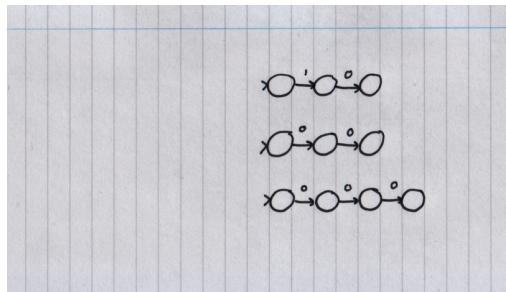
$P = L \cap \{0^j 1^k\} = \{0^m 1^m \mid m > 0\}$ which was already proved to be non-regular.

Converting Regular Expressions to Automata

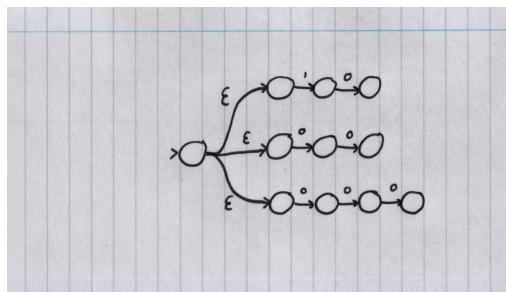
Regular expressions can be converted to automata by drawing the trivial automata for the most deeply-nested concatenated strings, then working outwards to combine automata as described earlier ([see more: Regular Expressions and Automata, page 28](#)).

For example, consider the construction of the automata for $101(10 \cup 00 \cup 000) * 111$.

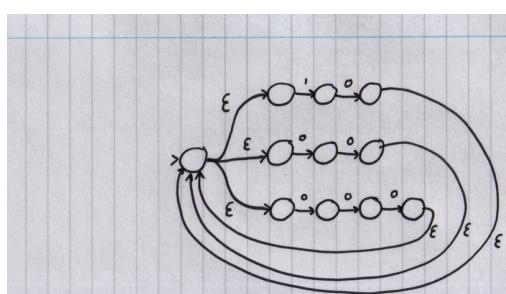
The three primitive automata for 10, 00 and 000 are drawn:



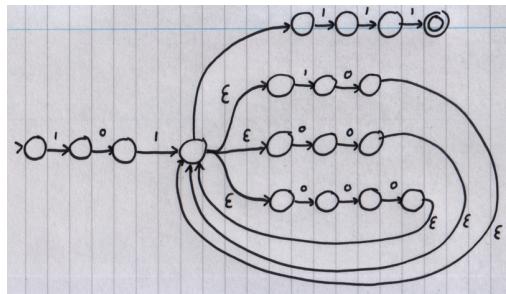
They are combined to create $10 \cup 00 \cup 000$:



They are wrapped to create $(10 \cup 00 \cup 000)^*$:



101 and 111 are added as a prefix and suffix respectively:



During this process, it is valid to **compress** multiple empty ϵ transitions.

Automata to Regular Expressions

To convert automata to regular expressions, we must find **all strings that drive the automaton from the initial state to accepting state(s)**.

We use $R(i, j, k)$ to denote the set of all strings that can drive the automaton in question from the state q_i to the state q_j without **passing through** any state q_x where $x \geq k$.

For example, $R(2, 4, 2)$ is the set of all strings that can drive the automaton from q_2 to q_4 , only passing through q_1 .

The **base step** of computation must be done manually, to determine $R(i, j, 1)$ for all pairs of states, q_i and q_j . The equation for $k = 1$ is as follows:

$$R(i, j, 1) = \begin{cases} \{\sigma \in \Sigma : \delta(q_i, \sigma) = q_j\} & i \neq j \\ \{\epsilon\} \cup \{\sigma \in \Sigma : \delta(q_i, \sigma) = q_j\} & i = j \end{cases}$$

The first term states that $R(i, j, 1)$ is the set of characters that will drive the automaton from q_i to q_j **directly** when $i \neq j$. The second term is similar but adds the empty string when $i = j$ (i.e. you can apply no characters and stay in the state).

The **inductive step** then allows the final output for $R(q_{initial}, q_{final}, n + 1)$ (where n is the number of states) to be built with the following rule:

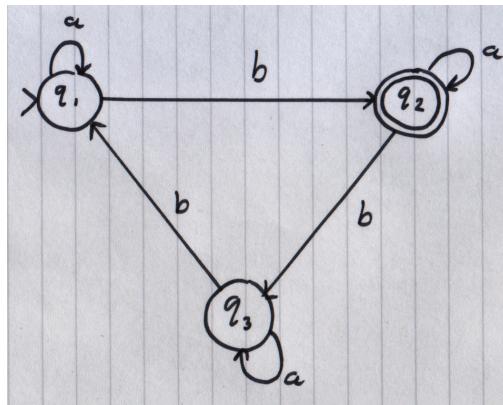
$$R(i, j, k + 1) = R(i, j, k) \cup R(i, k, k)R(k, k, k)^*R(k, j, k)$$

This states that to drive the automaton from q_i to q_j without passing through a state q_x where $x \geq k + 1$, there are two options:

- Go from q_i to q_j without passing through any state q_x where $x \geq k$.

- Go...
 - ...from q_i to q_k ,
 - then from q_k to q_k zero or more times,
 - then from q_k to q_j ,
 - all without passing through any state any state q_x where $x \geq k$.

For example:



$$R(1, 1, 1) = a \cup \epsilon$$

$$R(1, 2, 1) = b$$

$$R(1, 3, 1) = \emptyset$$

$$R(2, 1, 1) = \emptyset$$

$$R(2, 2, 1) = a \cup \epsilon$$

$$R(2, 3, 1) = b$$

$$R(3, 1, 1) = b$$

$$R(3, 2, 1) = \emptyset$$

$$R(3, 3, 1) = a \cup \epsilon$$

We want to get from q_1 to q_2 , so...

$$\begin{aligned} R(1, 2, 2) &= R(1, 2, 1) \cup R(1, 1, 1)R(1, 1, 1)^*R(1, 2, 1) \\ &= b \cup (a \cup \epsilon)(a \cup \epsilon)^*b \end{aligned}$$

The final result, $R(1, 2, 4)$, would take several more rounds of computation.