Text Searching and Processing Notes

Vishnu

Wednesday 15th May, 2019

Contents

1 Alphabet and String descriptions

1.1 Foundations

An **alphabet** ($\sum$) is a finite non-empty set. Members of an alphabet are called **letters**. A sequence of letters is called a **string**.

The set of all possible strings from an alphabet $\sum$ is represented with $\sum^*$: this includes the **empty string** (a sequence of length 0), which is denoted with ε. Lastly, the length of a string s is denoted as $|s|$, which means strings can be indexed with s[i] *(zero-indexed).* **e.g.**

$$\sum^* \text{ for } \sum = 0, 1 : \varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots, 01100011, 1001010, \ldots$$

Strings of length 2 from $\left(\sum = a, b, c\right) : aa, ab, ac, ba, bb, bc, ca, cb, cc$

1.1.1 String Operators/Definitions

x, y, u and v are strings for all of the below.

Identity x=y, which denotes: $|x| = |y|$ and $x[i] = y[i]\ \forall i \in [0, |x|)$
e.g decaf=decaf, han $\neq$ hands

Concatenation xy denotes the letters of x followed by the letters of y
e.g concat(play,ground) = playground

Factor/Substring x is a factor of y if y can be written as uxv for some u,v
e.g port is a factor of sports: s + port + s

Proper Factor x is a factor of y and $x \neq y$ (i.e u and v aren't both ε)
e.g hall is a factor of hall, all is a proper factor

(Proper) Superstring If x is a (proper) factor of y, y is a (proper) superstring of x
e.g hikers is a superstring of hikers and a proper superstring of ike

Prefix x is a prefix of y if y can be written as xv for some v
e.g boy is a prefix of boyfriend

Suffix x is a suffix of y if y can be written as ux for some u
e.g ring is a suffix of earring

Occurrence if x is a factor of y, x *occurs* in y. We can mark these occurrences by the start position (first letter of x in y) or end position (last letter of x in y).
e.g ab occurs in abacbab at (0,6) for start positions and (1,7) for end positions

1.2 Powers and Primitives

Powers For a string x, x^0 is ε and x^k is x^{k-1}: essentially, x^n is x repeated n times, so is undefined for negative numbers.
Similar to real numbers, if $x^a = y^b$ and $a > 0, b > 0$ then x and y are both powers of another string z.
e.g ab^3 = ababab, aa^2 = $aaaa^1$ (both are powers of a)

Primitive A primitive string is one that isn't a power of any other string (i.e if x=a^b, then a =x and b=1). A test for this is if the string x only appears twice (as a prefix and suffix) in x^2

Root and Exponent A non-empty string x can be written as $(root)^{exponent}$, where the root is a primitive string.
e.g abc = abc^1 , ababab = ab^4

1.3 Conjugate Strings

Two non-empty strings x and y are conjugate if they can be written if x = uv and y = vu for some strings v,u (Implying that x and y are the same length). v or u can be ε, but not both.

$$\text{x and y are conjugate} \iff \text{root(x) and root(y) are conjugate}$$
$$\text{x and y are conjugate} \iff \text{there is a z such that xz=zy}$$

Proof of the first:

$\Longrightarrow$ Given: x = uv = a^n, y = vu = b^m.
This implies a has the same prefix as u and suffix as v

$\Longleftarrow$ Given root(x) = ab , root(y) = ba $\rightarrow$ x = $(ab)^n$, y=$(ba)^n$.
We can write these as x = a$(ba)^{n-1}$b, $(ba)^{n-1}$ba.
With u = a and v = $(ba)^{n-1}$b, x=uv and y = vu.

Proof of the second:

$\Longrightarrow$ Given: x = uv, y = vu. By appending u: xu = uvu, uy = uvu $\rightarrow$ z = u

$\Longleftarrow$ Given: xz = zy. Therefore we know that either x and z have a common prefix, or x is a prefix for z $\rightarrow$ x = ab, z = $x^n a$ (n=0 means common prefix, else x is prefix).

$$\begin{aligned} xz &= zy && \textit{(Expanding z)} \\ xx^n a &= x^n ay && \textit{(Removing } x^n \textit{ from prefix of both sides)} \\ xa &= ay && \textit{(Expanding x)} \\ aba &= ay && \textit{(Removing a from prefix of both sides)} \\ ba &= y \end{aligned}$$

1.4 Periods and Borders

Period p such that x[i] = x[i+p] $\forall i \in [0, x - p)$. i.e the length of a substring that repeats itself within x (note that it need not finish repeating). By definition, $|x|$ is a period of x for any string x.

per(x) The smallest period of a string.

Border a proper factor of x that is a prefix and a suffix (these can overlap)

border(x) The longest border of a string.

* per(x) + $|border(x)| = |x|$

e.g. x = abbacabba → Periods = [6,10]; Borders:[ε, a, abba]
e.g. x = aabaabaa → Periods = [3,6,7,8]; Borders:[ε, a, aabaa]

Using the above descriptions, we can say that any border for a string s is either:

- border(s)
- A border of border(s)

We can also describe a period p of a string s as:

- s is a prefix of y^k, where y is a string of length p
- s = yw = wz, where y and z are strings of length p and w is the border

2 Sequential String Matching

Checking if a string x is present in string y. This can be done naively in O($|x||y|$), by checking x against each position of y. If there is a mismatch, we increment the checking position of x by 1 and check again.

2.1 MP Algorithm

Since we know that that the prefix of s repeats on per(s), we can instead shift by the per of the portion of the string that matched. Since we know that the shifted part matches, we don't need to check it again, so we can then just start checking from the border of the string. In the algorithm, we generate the border for each substring of x and use that to shift/check.

Border Algorithm (where border[i] represents the border for substring x[0:i]):

A Border[0] = -1 *(This is where the string doesn't match at all, so we start checking from the beginning.)*

B for i in range (0, $|x|$):

1 j = border[i] *(We know that any border is either border(s) or a border for border(s), so we can use previous values to speed up the search)*

2 while j$\geq$ 0 and x[i]$\neq$x[j]:

j = border[j]

3 border[i+1]=j+1

In the algorithm, we 'shift' x to the border[$|s|$], were s was the length of the match between x and y. This effectively moves the start of x by per(s), and the next iteration checks from after the matching section.

MP(x,y):

A i = 0, j = 0 *(i tracks the position in x, j tracks the position in y)*

B while j < $|y|$:

1 while $i = |x|$ or $x[i] \neq x[j]$: *($i = |x|$ to move per(x) after finding a full match, $x[i] \neq x[j]$ for a mismatch)*

j = border[j]

2 i++, j++

3 if $i == |x|$

i. print(X occurs in Y at (j-i)))

e.g. x = abacabacab, y = ababacadab, u = matching substring
j=0, i=0 $\rightarrow$ u=a
j=1, i=1 $\rightarrow$ u=ab

j=2, i=2 → u=aba
j=3, i=3 → mismatch: i = border[3] = 1
j=4, i=2 → u = ab
⋮
j=8, i=6 → u = abacaba
j=9, i=7 → mismatch: i = border[7] = 3
j=10, i=4 → mismatch: i = border[4] = 1
⋮

2.2 KMP Algorithm

The KMP Algorithm uses strict borders and interrupted periods, but is otherwise the same.

Interrupted Period A period of the string where the repeat doesn't finish ($|s|-|strict\_border(s)|$)

Strict Border For a string x and substring u, w is a strict border for u if:

1. w is a border for u
2. wt is a prefix for x, ut is not (t is a single character)

Effectively, a strict border is one where the border doesn't continue past the end of the string. If we consider u=x, all borders are strict borders.

Strict Border Algorithm (where KMP_next[i] represents the strict border for substring x[0:i]):

A kmp_next[0] = -1, k = 0 *(This is where the string doesn't match at all, so we start checking from the beginning.)*

B for i in range (0, $|x|$):

1 if x[i] = x[k]:

i. kmp_next[i] = kmp_next[k]

2 else:

i. kmp_next[i] = k

ii. do
$k = kmp\_next[k]$
while $k \geq 0$ and $x[i] \neq x[k]$

3 k = k+1

C kmp_next[m] = k

k marks the end of the border as a prefix, **i** as a suffix: if they're the same it's not a strict border so we can re-use values, if they're different that means we've found a strict border, but have to re-calculate the distance between k and i.

2.3 String Matching Automaton

To find the string x in y, we make an automaton that accepts the sequence x, then give in the letters of y till a match is found. For space efficiency, we try to find the smallest automaton possible. Forward transitions imply a match, and backward transitions occur on a mismatch: we can make these smarter using periods and borders.

Each state of the automaton represents u, a prefix of x, and the transitions each represent a letter t. Forward arcs go from u to ut if ut is a prefix of x, and to the initial state (ε) otherwise. Backward arcs go to vt, where vt is the longest suffix of ut that is a prefix of x (effectively a strict border of u).

3 Dictionary Matching

Rather than matching a single string, this method matches multiple strings (a dictionary) from a known language. The output is the full set of occurrences of every string in the dictionary in the given string.

Trie ($\tau(X)$) A tree where each node represents a prefix of a string from X (a list of strings): it essentially matches to every string in X

Dictionary Matching Automaton (D(X)) An automaton where each state is a prefix of a string in X, terminal states represent the strings in X, and arcs are in the form: (source, transition, destination)
The destination for a transition T from a source U is the longest suffix of UT that matches prefix of a string in X. It's effectively a multiple string matching automaton.

3.1 DMA With Failure Function

In a basic DMA, a lot of links are repeated, so we can reuse them using relations between the nodes. If we get to a node U that doesn't have a forward transition for T, we can go to fail[U] and apply T there, rather than directly moving to h(UT). fail[U] is pre-computed for every U with **TargetByFail**(fail[u], a), and is done in a breadth-first manner (a is the last character of U). The generation algorithm is:

TargetByFail(U,T):

A while $u \neq Nil$ and $transition(U, T) = None$ *(U is a non-empty string which doesn't have a transition for T)*

U = fail[U]

B if $U = Nil$

return Initial State *(The algorithm is unable to find a match, return to the empty string)*

else

return transition(U,T) *(We found a match, so apply T and move on)*

Nil is a constant which means 'go to initial state', and we set fail[ε]= Nil.
This method also easily allows us to identify terminal states: if fail[X] is a terminal state, then X is a terminal state.

Rather than separately checking if there is a valid transition, the algorithm calls TargetByFail at each step since if there is a valid transition it will just apply it.

To further optimise transitions:

1. If fail[x] doesn't have the required transition, we can set fail[x] = fail[fail[x]]
2. If every transition from x and fail[x] all lead to the same place, we can set fail[x] = fail[fail[x]

4 Searching through a Set of Strings

The inverse of dictionary searching, this takes a lexicographically (alphabetically) sorted set of strings (L) and either:

- Searches for a string X, and if not present finds where it would be placed
- Finds the strings that have a string X as a prefix

The methods use LCP(a,b), which is the longest common prefix between strings a and b.

4.1 Binary Search

An simple method that uses constant space and is $O(|x| * log|L|)$ is binary search.
Binary Search(L, X)

A d=-1, f=$|L|$

B while $d + 1 < f$:

1 $i = int((d + f)/2)$, $l = lcp(x, L[i])$

2 if $l = |x|$ and $l = |L[i]|$:

return i *(The LCP of both strings = length of both strings, so they're equal)*

else if $l = |L[i]|$ OR ($l \neq |x|$ and $L[i][l] < x[l]$):

d = i *(L[i] matches but is shorter than X OR L[i] partially matches, and the next character is smaller in L[i] than X $\rightarrow$ X is lexicographically greater than L[i])*

else

f = i

C return (d,f) *(X isn't in the list of strings, but if it was would be between d and f (f=d+1))*

4.2 Binary Search Tree

To make the above easier for multiple searches, and for use in the next algorithm, we can make a binary search tree of indices. Each node is of the form(d,f).

Root: (-1, $|L|$)

Children: (int(d+f/2),f) , (d,int(d+f/2))

Leaves: (x,x+1)

4.3 LCP Search

Effectively the Binary Search with some extensions, this method reduces the overall number of iterations and comparisons by making more use of LCPs. In addition to d, f and i, the algorithm keeps track of l_d and l_f, the LCP between X and L[d]/L[f] respectively.

The algorithm uses the following three cases, all of which assume that $l_d < x < l_f$ (x is somewhere between d and f) and use $l_{if} = lcp(L[i], L[f])$:

1. $l_d \leq l_{if} < l_f \rightarrow$ x has more in common with f than i does with f $\rightarrow$ move d=i
   X is somewhere between i and f, and $l_i = l_{if}$
2. $l_d \leq l_f < l_{if} \rightarrow$ x has less in common with f than i does with f $\rightarrow$ move f=i
   X is somewhere between d and i, and $l_i = l_f$
3. $l_d \leq l_f < l_{if} \rightarrow$ f has the same prefix for x and i $\rightarrow$ check L[i] and x from index l_f

These cases can also be flipped, checking against d rather than f.

LCP Search(L, X)

A d=-1, f=$|L|$, l_d=0, l_f=0
(Since L[d] and l[f] don't exist, lcp with anything is 0 by default)

B while $d + 1 < f$:

1 $i = int((d + f)/2)$, $l_{if} = lcp(L[i], L[f]), l_{id} = lcp(L[i], L[d])$

2 if $l_d \leq l_{if} < l_f$: *(Case 1)*
d=i, l_d=$l_i f$ *(Move d up, $l_d = l_i = l_{if}$)*

else if $l_d \leq l_f < l_{if}$: *(Case 2)*
f = i *(Move f down, l_f already equals l_i)*

else if $l_f \leq l_{id} < ld$: *(Case 1 flipped for d)*
f=i, l_f=$l_i d$ *(Move f down, $l_f = l_i = l_{id}$)*

else if $l_f \leq ld < l_{id}$: *(Case 2 flipped for d)*
d = i *(Move d up, l_d already equals l_i)*

else *(Case 3 for both, so a manual check)*

i. $l = max(l_d, l_f)$

ii. $l = l+$ lcp(x[l:],L[i][l:]) *(We know the first l match, so check after that)*

iii. if $l = |x|$ and $l = |L[i]|$:
return i
else if $l = |L[i]|$ OR ($l \neq |x|$ and $L[i][l] < x[l]$):
d = i, $l_d = l$
else
f = i, $l_f = l$

C return (d,f) *(X isn't in the list of strings, but if it was would be between d and f)*

5 Suffix Trees

For a string X, a suffix tree is a Trie for the set of suffixes of X (normally represented as $X_{[0...n]}$). One application of such a tree is string matching: any substring of X has to be a prefix of a string in $X_{[0...n]}$.

Naively constructing a tree gives $O(n^2)$ nodes: the following algorithm brings that to $O(n)$, with the following assumptions/constraints:

- The alphabet's size is constant
- All the nodes that represent a suffix will be leaves
- No suffix is a prefix for another suffix : this can be achieved by adding a new character that is not part of the language (usually \$) to the end of the string

Some notation for the rest of the section:

$\boldsymbol{S_u}$ String represented by node u

child(u,c) node connected from u where the connecting edge starts with character c

parent(u) node that connects to u

depth(u) $|S_u|$

start(u) starting position of S_u in X

$\boldsymbol{S_u} =$ T[start(u) : start(u)+depth(u)-1]

edge(parent(u), u) = S_u[depth(parent(u) :]

5.1 Locus

In the context of a suffix tree, a locus is a pair (u,d) where:

$$depth(parent(u)) < d \leq depth(u)$$

Effectively, it represents the node (and string) that would be at depth d along the edge (parent(u), u) in the uncompacted tree. Every factor of X has a locus in the suffix tree.

5.2 Brute Force Construction

To make a tree, we add in suffixes one at a time, starting from the longest:

1. If the new suffix has a prefix in common with an existing suffix, we find the locus where they diverge, make a new node there, then make a leaf connected to that
2. Otherwise, or if there are no existing nodes, we make a leaf connected to the root

BruteForce($T_{[0...n]}$):

A root = empty_node(), depth(root) = 0

B u=root, d=0

C for i in range(0, n):

1 while $d = depth(u)$ and $child(u, T[i + d])\ exists$: *(Common Prefix)*

i. u = child(u,T[i+d]), d+=1

ii. while $d < depth(u)$ and T[start(u)+d] = T[i+d]: *(Traverses till it finds the difference)*

d+=1

2 if $d < depth(u)$:

u = CreateNode(u,d)

3 CreateLeaf(i,u,d)

4 u=root, d=0

In the brute-force algorithm, to add a suffix S, if it finds a node (say N) with a common prefix it traverses S_N till it finds a difference or reaches the end of the string. If it finds a difference it creates the new node then a leaf, otherwise it creates a leaf node at the end.

However, if it reaches the end of the string that means S=S_N (which is impossible, two suffixes can't be identical) or S is a prefix of S_N (also impossible due to our assumption). It only reaches the end of a string after it breaks out of the loop in 1: if it fully matches a string before that it means a locus so it just loops, and so there are no issues.

5.3 Suffix Links

For a node u, slink(u) is the node n where S_n is the longest proper suffix of S_u . For the root, this is undefined, but set as slink(root) = root.
If we take slink(u) = S_u[1:] for all u, depth(slink(u)) = depth(u)-1. Below is the proof that this exists for all u:

- Leaf Nodes: Every suffix is present, so this is trivially true
- Internal Nodes: If an internal node u exists, there's a division after a character c between two edges. This character (and split) will re-appear in any suffix of S_u that contains c: including S_u[1 :], so a node S_u[1 :] will be created

ComputeSuffixLink(u):

A d = depth(u), v = slink(parent(u))

B while $depth(v) < d - 1$:

v = child(v, X[start(u)+depth(v)+1]) *The next node in the substring u[1:]*

C if $depth(v) > d - 1$: *If the algorithm overshot and s[1:] doesn't exist yet*

Create the node s[1:], v = new node

D slink(u) = v

The algorithm effectively goes to the suffix link for the parent, then traverses nodes to find the required suffix link. If it hasn't been created yet, it finds the first node that overshoots (i.e u[s:1]+x) then inserts a new node before that.

5.4 McCreight's Algorithm

Effectively the Brute-Force algorithm with a single improvement: rather than searching from the node at each iteration, it instead goes to the suffix link of the leaf's parent (which by definition will be a prefix of the next suffix to be added to the tree) and finds the divergence from there. If the leaf's parent doesn't have a suffix link, it creates one at the same time it creates the leaf.

McCreight($T_{[0...n]}$):

A root = empty_node(), depth(root) = 0

B u=root, d=0, slink(root) = root

C for i in range(0, n):

1 while $d = depth(u)$ and $child(u, T[i + d])\ exists$: *(Common Prefix)*

i. u = child(u,T[i+d]), d+=1

ii. while $d < depth(u)$ and T[start(u)+d] = T[i+d]: *(Traverses till it finds the difference)*

d+=1

2 if $d < depth(u)$:

u = CreateNode(u,d)

3 CreateLeaf(i,u,d)

4 if slink(u) = null:

slink(u) = ComputeSuffixLink(u)

5 u=slink(u), d=max(d-1,0)

d = max(d-1,0) is the same as d = depth(slink(u)), since the suffix link is either one shorter or the root,

5.5 Generalised Suffix Tree

To create a single suffix tree for multiple strings, just use a different ending character (e.g $\$_1$,$\$_2$) for each. Both the brute-force and McCreight's algorithms can be used.

5.6 Suffix Tree Applications

5.6.1 String Matching

As mentioned before, string matching on the string becomes trivial. If x is the string to be found, simply find how many leaf nodes are descendants of x: that gives the number of occurrences of x in the string.

5.6.2 Number of Distinct Substrings

This method requires an uncompacted tree: the number of nodes is the number of distinct substrings. In a compacted tree, the number of nodes + the number of loci, since loci are effectively uncompacted tree nodes.

5.6.3 Longest non-unique substring

Since we know that all leaves are unique, and the closer to the start of the tree the shorter the string, the longest repeating substring is the deepest internal node. Any internal node implies that the string is a prefix for two (or more) substrings, so we know that it's non-unique.

5.6.4 Common Substrings

In a generalised suffix tree, each string has a unique ending character (e.g $\$_1$,$\$_2$). If an internal node is an ancestor of leaves with different ending characters, it's a common substring to the string corresponding to each character. Similary, the longest common substring is the deepest internal node with leaves with each ending character.

5.6.5 Matching Suffix Arrays

If we've pre-processed $T_{[0...N]}$, we can efficiently match $S_{[0...N]}$ against it. i.e if each suffix of S is present in T. The standard would be to just try to match each string, but since we know that $S_{n+1} = S_n[1:]$, we can use suffix links.

The algorithm follows S_n as far as possible: when it hits a mismatch at depth j (i.e S[n:j]), take the suffix link of node at j (i.e S[n+1:j]). If there is no mismatch, then all S_m where $m > n$ are present.

5.6.6 Longest Common Prefix Between Suffixes

For two suffixes of S, namely S_i and S_j, the longest common prefix is the deepest internal node that leads to both leaves. For more than two suffixes, it's the deepest internal node that lead to all the leaves.

6 Table of Prefixes

A **Prefix Table** for a string X is a table t such that:

$$t[i] = |lcp(X, X[i:])|$$

i.e the length of the lcp between X and suffix X_i. Naively, this can be constructed in $O(n^2)$, but by using previously computed values this can be $O(n)$.

6.1 Proofs for Algorithm

For $i > 1$:

$$g = max(j + t[j], 0 < j < i)$$
$$f = j (This\ is\ the\ j\ from\ g)$$

Using the above values, X[f:g-1] is a prefix of X of length t[j] (g-1-f = t[j], and we know t[j] is the length of a match). Therefore, X[f:g-1] is a border of X[0:g-1].

Using the above, if $\boldsymbol{i < g}$ we can say that:

$$t[i] = \begin{cases} pref[i-f] & , if pref[i-f] < g-i \\ g-i & , if pref[i-f] > g-i \\ g-i+|lcp(x[g-i:|x|-1], x[g:|x|-1]| & , otherwise \end{cases}$$

As a border, x[0:t[f]]=x[f:f+t[f]-1]. Since f+t[f]=g and $i < g$, i ∈ x[0:t[f]].

1. If t[i-f] < g-i, then the entire match occurs within x[0:t[f]]. The same match will occur for t[i] since it's the same string, so we just copy the value.

2. g-i is the distance between i and the mismatch. The same match occurs for t[i], so we know that the new string only matches till g as well, so the length of the match is g-i.

3. The match occurs to the end of the substring in the original, but we don't know if it ends or continues to match, so we check the remainder.

6.2 Prefix Table Algorithm

Prefixes(X):

A $pref[0] = |X|, g = 0, f = undefined$

B for i in range(1, n):

1 if $i < g$ and $pref[i - f] \neq g - i$:
pref[i] = min(pref[i-f],g-i) *(Case 1 and 2)*

2 else:

i. g = max(g,i), f=i *(if $i < g$ Case 3, otherwise regular comparison)*

ii. while $g < |x|$ and $x[g] = x[g + f]$
g++

iii. pref[i] = g-f

This algorithm runs in $O(|x|)$: ii can run at most $|x|$ times across the entire run since g only increases, and B only runs $|x|$ times, so the total is $O(2|x| - 1)$.

This algorithm can also be used for string matching: append the pattern to be found P to the string S, then find pref[i]=len(P).

6.3 Suffix Table + Algorithm

Similar to the prefix table, a **Suffix Table** is a table t for a string X such that:

$$t[i] = |lcs(X, X[0 : i])|$$

i.e the length of the longest common suffix between X and prefix X[0:i]. Using a similar method to the prefix table, this can be generated in O(n).

Suffixes(X):

A $|X| = m, \text{suff}[m - 1] = m, g = m - 1, f = undefined$

B for i in range(m-2, 0):

1 if $i > g$ and $\text{suff}[i + m - f - 1] \neq i - g$:
pref[i] = min(suff[i + m - f -1] ,i-g) *(Case 1 and 2)*

2 else:

i. g = min(g,i), f=i *(if $i > g$ Case 3, otherwise regular comparison)*

ii. while $g > 0$ and $x[g] = x[g + m - f - 1]$
g -= 1

iii. suff[i] = f-g

One thing to note about the suffix table is if that suff[i]=i+1, then x[0:i] is a border: if the longest common suffix between x and the prefix x[0:i] is the prefix itself (the length of the prefix is i+1), then that's a border.

7 Boyer-Moore Algorithm

One of the most generally efficient string matching algorithms, variations of this are often used in text editor for find/replace. Unlike previous algorithms, it checks the pattern from **right to left**. It still looks for the pattern from the beginning of the string (moves the window **left to right**), but within the window it checks from right to left.
When the algorithm finds a mismatch/matches the entire string, it uses one of two shifts to move the window to the right. For both of these, we take the original/search strings as y/x, the left-most position of the window as j (i.e y[j] is compared against x[0]), and $|x|$ as m.

7.1 Good-Suffix Shift

Assuming the mismatch happens at x[i] (y[i+j]), we can say that the substrings x[i+1:m-1] and y[j+i+1:j+m-1] are equal. This string is stored as u, and the mismatched character from y as c. The window is shifted so that the right-most occurrence of u in x with a different preceding character lines up with y[j+i+1:j+m-1]. This occurrence is normally to the left of the currently aligned u, so the window is shifted to the right to align it with u in y. The right-most is to prevent accidentally skipping over a match.
If there isn't such an occurrence, instead find the longest suffix of u that matches a prefix of x and align them: x is shifted to the right to match u in y. U is a suffix of x, and we're trying to align it with a prefix of x: this is finding a border, so we shift by per(x).

7.2 Bad-Character Shift

Again assuming the mismatch happens at x[i] (y[i+j]), we store the mismatched character from y as c. The window is shifted to align y[i+j] with the right-most occurrence of in x[0:m-2]. Again, the right-most is to prevent accidentally skipping over a match. If there is no occurrence of c in x, j is set to j+m+1: effectively shifting the entire string by its full length.

7.3 Pre-Processing

Both of these shifts can be pre-processed for the search string (x), and the algorithm checks the table for both to decide which gives the larger shift.

7.4 Good-Suffix

Since this uses suffixes of x, we denote each suffix of x as U_i. We then define d(u) = $min|z|$ such that:

1. TUZ is a suffix of X, but TU isn't a suffix of x

   OR

2. x is a suffix of uz

Z is a string matching one of these conditions, T is an single character belonging to $\sum$. Using the above, we get d(u) = per(uz).

Proof of the above:

1. Since uz and u are both suffixes of x, and $uz > u$, u is a suffix of uz $\rightarrow$ therefore also a border of uz, and z is a period for uz.
   Also, since we're searching left to right, and TU isn't a suffix of x, T can't be the character that caused the mismatch. Therefore this gives us the an occurrence of u to the left of the current position that has a different preceding character. Since we're looking for $min|z|$, this gives us the first occurrence.

2. Since u is a suffix of x and x is a suffix of uz, u is also a suffix of uz $\rightarrow$ This means u is a border of uz, so z is a period of uz.
   This gives us the a suffix of u that matches a prefix of x. Since we're looking for $min|z|$, this gives us the longest suffix.

Z effectively shifts U to the left, to align it with a new occurrence or find the longest prefix. Since this new occurrence is to the left in X, we shift X to the right to line it up with Y. Since U shifts by $|z|$, and we have to shift all of X to align U in the same way, we shift $|x|-|z|$: in rare cases this might be less than 0, so we set the shift to min(0, $|x|$-$|z|$).

To efficiently compute this, we compute the suffix table for x (see section 6.3). After that, we compute d(u) for each u: first we try to find the periods (Case 1), and then check to see if it's more efficient to find prefixes of x in its suffix (Case 2).

Compute_D(suff):

A $|X| = m, j = 0, D[i] = m \; \forall \; i$

B for i in range(m-1, 0):

1 if suff[i] = $i + 1$: *This is a border: see the bottom of 6.3*

while $j < m - i - 1$ *i+1 is a border,* $|x| - i - 1$ *is a period. This loop runs through the length of this period.*

A. D[j] = m-i-1 *Sets each member of the period string to the period (Case 1) (i.e the shift to get the same character)*

B. j ++

2 for i in range(0, m-2)

i. D[m-suff[i]-1] = m-i-1 *m-suff[i]-1 is the appearance of that character at x[m-i-1] in the suffix of the prefix x[0:1] (Case 2) (m-i-1 is the shift to match them)*

7.4.1 Bad-Character

Unlike every other table so far, this stores a value DA for each character in $\sum$. DA[c] for a character c in x is index of the right-most occurrence in $x[0:|x|-2]$, **with the indices reversed** .i.e the distance between the right-most occurrence in $x[0:|x|-2]$ and the end of $x[0:|x|-2]$. This guarantees that $DA[c] \geq 2$ If c doesn't appear in x, DA[c]=$|x|$.
To convert from DA[c] to the shift:

$$Shift = DA[c] - |x| + 1 + i \tag{1}$$

Where i is the mismatched position in x (this is the regular index, even though the checking is done from right to left), and c is the mismatched character in y.

7.5 Algorithm + Performance

With the pre-processed shifts the algorithm is relatively simple: it checks each window from right to left, then uses whichever shift gives it a larger right shift.
BM(x,y):

A $j = 0$

B while $j < |y| - |x|$:

1 i = $|x| - 1$ *(j/i are for y/x respectively)*

2 while $i \geq 0$ and $x[i] = y[i + j]$: *(Checking left to right)*

i++

3 if $i = -1$:

i. print(X is present in Y at j)

ii. j+=per(x)

4 else:

j+= max(D[i], DA[y[j+i]]-m+i+1)

At worst, this is $O(|x||y|)$: but it's usually closer to $3|y|$, and at best can be $\frac{|y|}{|m|}$. It's better than MP/KMP for natural language processing, but for periodic sequences like DNA it's worse.

8 Karp-Robin Algorithm

One weakness of most algorithms is that they have to check each character of x and y at each stage. The Karp-Robin instead hashes x, then compares it with the hash of the window in y: only if they're the same (so the strings are similar) does it compare the actual characters.
An immediate downside of this is that the window can only be shifted by 1 each time, since we can't use periods/borders/any other shortcuts on the string, However, by building a hash such that each hash can be constructed from the previous one, this algorithm can become very efficient: even O(n) on average.

8.1 Constructing the Hash

A hash is a function that converts from some range of values to a limited set of integers. Using a modulus is a popular hash function, as x%y converts any integer x to the range (0,y-1). It's not perfect though, as $x\%y = z\%y$ does not imply $x = y$, which might provide false positives.
A useful property of the modulus is:

$$((x\%q) + (y\%q))\%q = (x + y)\%q$$

Using this property, we can make a hash such that the hash of $x[i_1 : i_2]$ can be used to generate the hash of $x[i_1 + 1 : i_2 + 1]$. The hash for a string s, using user-parameters d and q, is:

$$hash(s) = (\sum_{i=0}^{|s|-1} (int)s[i] * d^{m-1-i}) \ \% \ q$$

Effectively, convert each character to a number (this could be ASCII, or more likely custom-based on the language), multiply it by the appropriate power of d (so each position has a weight . i.e $hash(ab) \neq hash(ba)$), then modulus the whole thing. The weight to each position and each character having a custom value makes the hash for each string quite unique before modulus, but without the modulus the numbers could quickly become very large.
To generate the next hash (removing s[0] and adding c, the next character in the searched string):

$$new\_hash = ((hash(s) - ((int)s[i] * d^{m-1})) * d + (int)c) \ \% \ q$$

Effectively, remove the first term (which has weight d^{m-1}), increase each character's weight in the hash by one position, then add in the new character with weight 1 (implying the last position) and modulus the whole hash. By the above modulus properties, this is the same as just generating the new hash, but is $O(1)$ rather than $O(n)$.

8.2 Algorithm + Performance

Again, with the hash the algorithm is relatively simple: shift the window by 1 at each iteration, and if the hashes are equal check the characters. **KR(x,y,d,q)**:

A $h_x = 0, h_y = 0, D = d^{|x|-1}$

B for i in range(0, $|x| - 1$): *(Generating hash(x) and initial hash(y))*

(a) $h_x = h_x * d + (int)x[i]$

(b) $h_y = h_y * d + (int)y[i]$

C for j in range(0,$|y| - |x|$): *(Since the window starts at j, we don't need windows starting at the last $|x|$ chars)*

1 if $h_x = h_y$

i. i = 0

ii. while $i < |x|$ and $x[i] = y[i + j]$: *(Checking left to right)*

i++

iii. if $i = |x|$:

A. print(X is present in Y at j)

2 if $j! = |y| - |x|$: *(y[i+j] is out of bounds)*

i. $h_y = (h_y - (y[j] * D)) * d + (int)y[i]$

Again, at worst this is $O(|x||y|)$ if the hash is terrible (e.g. hash(s) = 1). However, if $q \geq m$ and assuming that the strings are evenly distributed in the range (0,q) by **%q**, then the number of false positives is at worst n/q (all the strings with equal hashes are false matches). The algorithm is then $O(n + \frac{nm}{q})$, so overall $O(n)$.
This algorithm is also great for multiple string searching: we generate a hash for each string to be found, and they can all be compared at the same time, rather than running through and checking for each string separately.

9 Suffix Arrays

Most algorithms so far have dealt with pre-processing a single pattern, so it can be searched for in various texts. Often however, we'll have a single text that we want to search for different patterns. This problem is called **Text Indexing**.

Karp-Robin or Suffix Trees can work for this, but a more efficient way is to use *Suffix Arrays*. These are lexographically sorted lists of the suffixes of a string: any pattern/substring that is present in the string will be a prefix for one of these suffixes. Rather than directly storing the suffixes (a huge amount of space), the array instead stores an index i, representing string $x[i:]$.

Since this is a sorted array, we can just use a modified binary search or an LCP search that checks if the pattern P is a prefix for the suffixes: if we use the interval method, this gives us a list of occurrences of P in the string.
With storage and searching out of the way, the most time-consuming part is constructing the suffix array. Naively for a string s, this could take $O(|s|^2 log|s|)$, but this can be reduced to $O(|s|)$. For the algorithms, n is $|s$.

9.1 Algorithm Groundwork

9.1.1 Radix Sort

Using a Radix Sort, a sequence of n numbers with k possible numbers (e.g all the numbers are in the range $(1 \dots k)$ can be sorted in O(n+k). This also applies to pairs of numbers. Radix Sort works by considering each index of the number individually, then using Bucket/Counting sort (Make a 'bucket' for each possible value, put the numbers in the corresponding bucket, append all the buckets together) to sort them. If done from the lowest to highest index (Least Significant) this is a stable sort, but highest to lowest (most Significant) is usually faster.

9.1.2 Merging Arrays

Two sorted arrays A and B can be merged into a single sorted array in $O(|A| + |B|)$, as below: **Merge(A,B)**:

A i=0, j=0, a=0, b=0, C = []

B while $i + j < |A| + |B|$:

1 a = A[i] *(If out of bounds, a = max_value)*

2 b = B[i] *(If out of bounds, b = max_value)*

3 if $a < b$:

i. C.append(a)

ii. i++

else:

i. C.append(b)
ii. j++

Effectively, check the current position of each array, append the lower to c, then increment that array's position by 1. This can be made more efficient by stopping when one of the arrays is exhausted and simply appending the other array.

9.2 $O(n^2)$ Algorithm

This algorithm is similar to Least Significant Radix Sort, in that it sorts the population by only considering one index and using the ordering of the sorted substrings to resolve conflicts. To simplify, head(s) refers to the first character of s, and tail(s) refers to s[1:].

At each iteration, the algorithm generates a sorted list of substrings. Rather than performing a full sort each time, the algorithm generates a pair for each substring w: the first element is (int) head(w), the second is the index of tail(w) in the previous iteration's list. This list of pairs is sorted, converted back to strings, then used in the next generation. If a substring wasn't present in the previous list, it's given a default value of 0.

To convert a char to int, a custom list is made for the language (e.g. $\sum$=(a,g,t)→A:1,G:2,T:3). The algorithm uses previous orderings to sort, so only the head is considered at each generation which makes it more efficient, but it still sorts the list n (=$|s|$) times.

SuffixArray(s):

A for i in range(n-1, 0):

1 L_i = []

2 for j in range(i,n-1): *(Generate L_i)*

L_i.append((int)w[j], idx(L_{i+1}, w[j+1...$|s|-1$]))

3 Sort L_i

B Return L_0

9.3 $O(nlog(n))$ Algorithm

This is similar to the above, but rather than one letter at a time it doubles the length of the strings to sort in each generation (therefore taking it to $log_2(n)$ from n generations). To include all substrings, the length of the string is first doubled by adding $|s|$ null characters (any character not in the alphabet, such as $).
The pair is generated by halving the strings at each generation, and using the indices of each half from the previous generation. Again, if a substring wasn't present in the previous list it's given a default value of 0 (this will only happen if it started with $, which comes before all characters in $\sum$).

SuffixArray(s):

A n = $|s|$, Append $|s|$ $ to s , i = 2

B $L_1 = \sum$ *(L_1 is just sorted individual characters)*

C while $i \leq n$:

1 ss = []

2 for j in range(0,n): *(Add all unique substrings of len i that don't start with $)*

i. x = s[j:j+1]

ii. if x not in ss **and** x[0] $\neq$ $:

ss.append(x)

3 L_i = []

4 for j in range(0, len(ss)): *(Construct pairs)*

i. a = ss[j][0:i/2], b=ss[j][i/2:] *(Split each string into half)*

ii. L_i.append(idx($L_{i/2}$, a), idx($L_{i/2}$, b))

5 Sort L_i

6 i *= 2

D Return L_0

9.4 O(n) Algorithm (DC3 Algorithm)

Original paper: https://www.cs.helsinki.fi/u/tpkarkka/publications/jacm05-revised.pdf. This version slightly deviates from the original, but is still O(n).

The suffixes for s are split into 3 groups: S_0, S_1 and S_2. S_0 and $S_1 \cup S_2$ are sorted independently (so a 1/3 and 2/3 split), then merged back together. Sorting is done by creating triples in the strings, ranking each triple, and using them to rank the strings. To ensure that the triples are created properly, $$ (or another null character) is appended to S.

9.4.1 Splitting the Substrings

Substrings are added to each group in order. i.e s[0:] $\rightarrow S_0$, s[1:] $\rightarrow S_1$, s[2:] $\rightarrow S_2$, s[3:] $\rightarrow S_0 \ldots$. The two substrings starting with $ are ignored, so the groups may not be the same size, but the smallest substring has length=3. e.g

$$
\begin{aligned}
S &= aabbbab\$\$ \\
S_0 &= [aabbbab\$\$, bbab\$\$, b\$\$] \\
S_1 &= [abbbab\$\$, bab\$\$] \\
S_2 &= [bbbab\$\$, ab\$\$]
\end{aligned}
$$

9.4.2 Sorting S_0

The largest substring (index 0, which is equal to s), is split into groups of 3 characters. These triples are ranked, converting them to integers. Using these integers, each string in S_0 is converted to a string of integers. These are used to sort S_0 using MS radix sort. e.g.

$$\begin{aligned} S_0 &= [aabbbab\$\$, bbab\$\$, b\$\$] \\ aab|bba|b\$\$ & \\ 1 \mid\ 3 \mid\ 2 & \\ I_0 &= [132, 32, 2] \\ sorted(I_0) &= [132, 2, 32] \\ sorted(S_0) &= [aabbbab\$\$, b\$\$, bbab\$\$] \end{aligned}$$

9.4.3 Sorting $S_1 \cup S_2$

Same as sorting S_0, except the substring used is $S_1[0]S_2[0]$, and the integers for S_1 all have $I_2[0]$ appended to them (effectively, each string has the integer string that it's the prefix for).e.g.

$$\begin{aligned} S_1 \cup S_2 &= [abbbab\$\$, bab\$\$, bbbab\$\$, ab\$\$] \\ abb|bab|bbb|ab\$ & \\ 2 \mid\ 3 \mid\ 4 \mid\ 1 & \\ I_{12} &= [2341, 341, 41, 1] \\ sorted(I_{12}) &= [1, 2341, 341, 41] \\ sorted(S_1 \cup S_2) &= [ab\$\$, abbbab\$\$, bab\$\$, bbbab\$\$] \end{aligned}$$

9.4.4 Merging S_0 and $S_1 \cup S_2$

This is a standard merge, as in 9.1.2. Rather than comparing the strings directly (which would make the merge $O(|s|^2)$), the strings are converted into tuples using the sorted $S_1 \cup S_2$ (henceforth referred to as S_{12}), and these are compared. The tuples vary based on if the considered string belongs to S_1 or S_2:

- The considered string from S_0 is C_a, from S_{12} is C_b, idx(x,y) is the index of y in x
- C_b belongs to S_1:
  - Compare ($C_a[0]$, idx(S_{12},$C_a[1:]$)) and ($C_b[0]$, idx(S_{12},$C_b[1:]$))
- C_b belongs to S_2:
  - Compare ($C_a[0]$, $C_a[1]$, idx(S_{12},$C_a[2:]$)) and ($C_b[0]$, $C_b[1]$, idx(S_{12},$C_b[2:]$))

9.4.5 Algorithm

SuffixArray(s):

A n = $|s|$, Append $ $ to s , S = [[],[],[]]

B for i in range(0,n-1):

S[i%3].append(s[i:])

C Sort S[0]

D R = S[1].extend(S[2])

E Sort R

F i=0, j=0, a = 0, b = 0, C = []

G while $i < |S[0]| and j < |R|$:

1 if R[j] $\in$ S[1]:

i. a = (S[0][i][0], R.idx(S[0][i][1:]))

ii. b = (R[j][0], R.idx(R[j][1:]))

else :

i. a = (S[0][i][0], S[0][i][1], R.idx(S[0][i][2:]))

ii. b = (R[j][0], R[j][1], R.idx(R[j][2:]))

2 if $a < b$:

i. C.append(a)

ii. i++

else:

i. C.append(b)

ii. j++

H if $i < |S[0]|$: *(If S[0] didn't finish)*

(a) C.append(S[0][i:])

else: *(If R didn't finish)*

(a) C.append(R[j:])

I return C

9.5 Suffix Inverse Array

If the suffix array is denoted as SA, SA^{-1} is the suffix inverse array: for a string w, $SA^{-}1$[i] is the position of w[i:] in SA. Essentially, it's the rank of a substring in the suffix array.

9.6 Longest Common Prefix Array

The LCP array for a Suffix Array (SA) is an array of the length of the LCP between each adjacent suffix.

$$LCP[0] = 0$$
$$LCP[i] = lcp(SA[i], SA[i-1])$$

Since SA is lexicographically sorted, any common prefix between SA[i] and SA[j] is also common for any SA[k] where $k \in (i, j)$. This can be trivially found by finding the minimum value of LCP[k]. Using this, we can find the common prefix between any two suffixes of a string: use SA^{-1} to find the positions in SA, then find the minimum in the LCP. To generate the array in O(n):

LCPArray(s)

A $l = 0$

B for i in range(0,n-1)

1 k = SA^{-1}[i], j = SA[k-1] *(j is the start idx of the substring before s[i:] in SA)*

2 while s[i+l] =s[j+1]:

l++

3 LCP[k] = l

4 l = max(l-1,0) *(LCP[i]-1 is* $\leq$ *LCP[i+1])*

10 Regular Expressions

Regular expressions are a way of matching multiple patterns with a single string, effectively providing a template. The basic symbols that can be used in a regular expression are :

a,b,c... **:** Any single alphanumeric character

ε **:** An empty character

| **:** OR . e.g. $a|bs|cr$ represents the strings a, bs and cr

() **:** Parentheses . e.g $(a|bs)cr$ represents acr and bscr

* **:** 0 or more .e.g a* represents ε,a,aa,aaa,aaaa,...

+ **:** 1 or more .e.g ba+ represents ba,baa,baa,...
a+ can be written as aa*

? **:** 0 or 1. e.g ca?r represents cr and car
a? can be written as (a|ε)

| and * refer to the single character/parentheses block before them. Since ? and + can be re-written using | and *, for the purpose of the below algorithms they're unused.

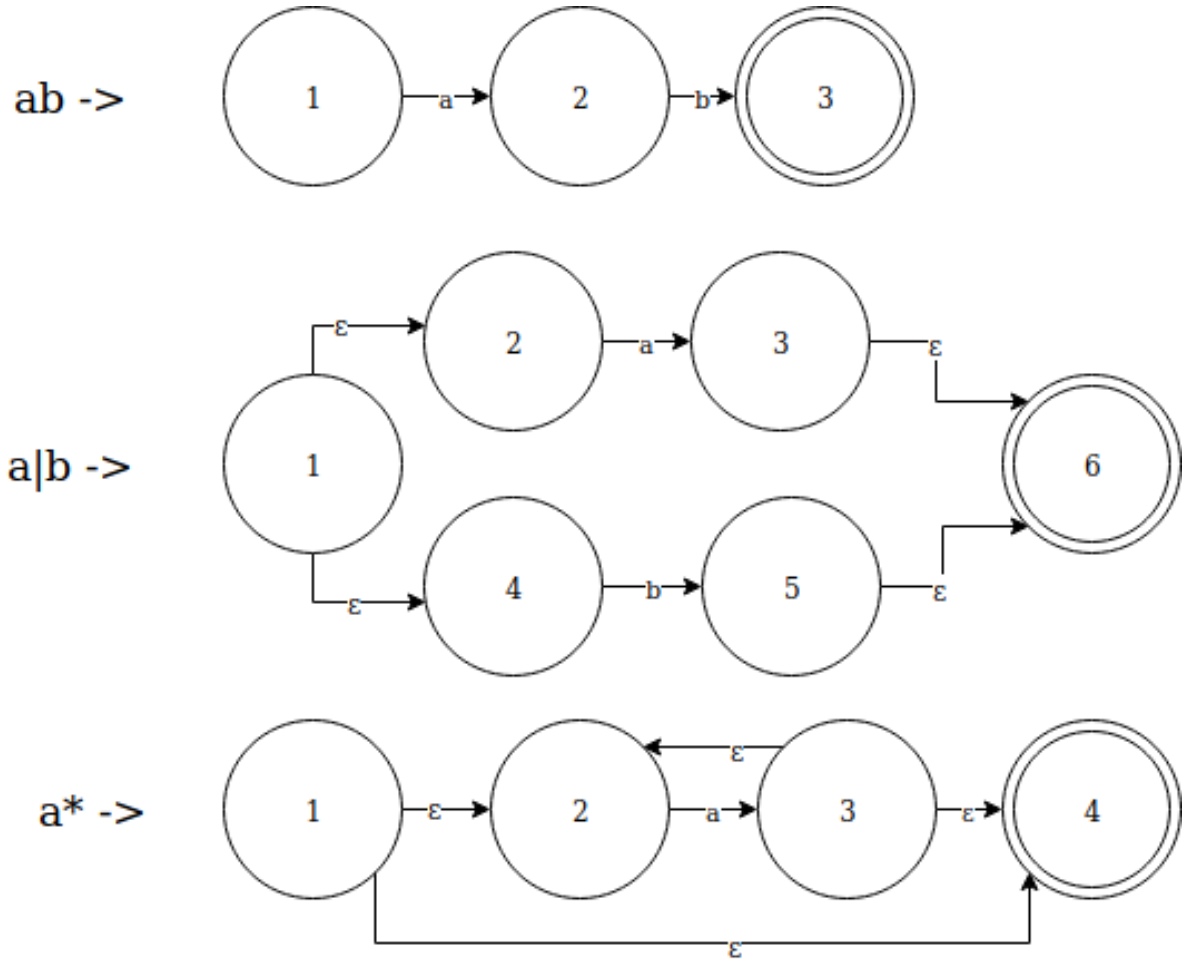
10.1 Non-Deterministic Finite Automata (NFA)

An automaton with two differences:

1. ε-transitions, so a transition that can occur without a symbol for free movement from one state to the other
2. A state doesn't need to have a transition for every symbol in the language, and can have multiple transitions for a single symbol.

Any regular expression can be converted to an NFA, and vice-versa.

10.2 Thomson's Construction



Good O($|s|$) construction, max. $2|s|$ states, max. $|s|$ labelled transitions, max. $4|s|$ ε-transitions, max. 2 incoming/outgoing edges per state

Bad Lots of ε-transitions, this causes a very large number of possible paths for checking a single text - O($|s|$*size($\sum$))

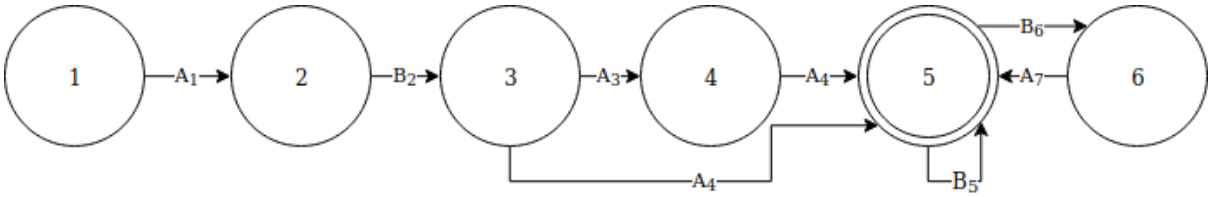
10.3 McNaughton and Yamada's Construction

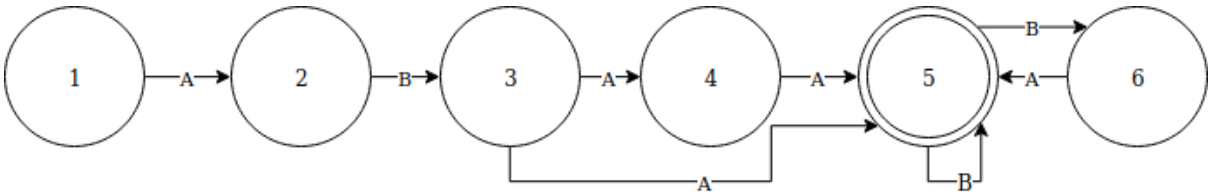
1. Add a subscript to each character
2. Make a table of potential successors for each character
3. Identify Start/End Characters for a pattern
4. Using 2 for transitions and 3 for start/initial state transitions, construct the NFA
5. Remove the subscripts

e.g.

$$s = ab(a|\varepsilon)a(b^*|ba)$$
$$s = a_1b_2(a_3|\varepsilon)a_4(b_5^*|b_6a_7)$$

Character	Successor
START	a_1
a_1	b_2
b_2	a_3, a_4
a_3	a_4
a_4	END,b_5, b_6
b_5	END, b_5
b_6	a_7
a_7	END





Good $|s| + 1$ states , no ε-transitions

Bad Making the table can be $O(|s| + |s|^2)$