CHAPTER 2

STRING AND TEXT ALGORITHMS

2.1 Suffix Tries and Trees

In this section, we will look at suffix tries and trees, and how they can be used to efficiently perform different string operations. We begin by defining some terminologies:

- an *alphabet* consists of all letters that a string can be composed of, e.g. ASCII. It is denoted by Σ.
- a *string* is a word on the alphabet.
- the *length* of a string S is the number of letters it has.
- for a string S, we use $S[i]$ to denote the i-th character in the string.
- for a string S, we use $S[i:j]$ to denote the substring of S from position i to j (both included).
- for strings S and T, their *concatenation* is denoted by ST.
- the *empty string* is denoted by ϵ. It has length 0.
- for a string S, another string U is a *substring* of T if $S = TUV$, for strings T and V.
- for two strings S and T, we say that U is a *common substring* if it is a substring of both S and T.
- for a string S, a *subsequence* of S is composed of characters in S (in the given order), but it need not be a continuous block like a substring. We can delete characters from S to get a subsequence.
- for strings S and T, we say that U is a *common subsequence* if it is a subsequence of both S and T.
- for a string S, a *prefix* is a substring starting from the start.
- for a string S, a *suffix* is a substring ending at the end.
- for an alphabet Σ, the set Σ^* is composed of all strings on Σ, including the empty string ϵ.
- a *leaf node* in a tree has no children.
- a *branch node* in a tree has at least one child.
- a *unary node* in a tree has precisely one child.
- a *binary node* in a tree has precisely two children.

We have previously seen suffix tries used to search hundreds of substrings in a single long text. We can do this using KMP or BM, but these are linear in terms of the long text during each computation. We can use suffix tries to make the algorithm to process the long text just once, and linear in terms of the number of strings we have to search. This is because the trie pre-processes the text and creates an index for it. Because the text is typically much longer than the substrings, this is much faster in practice.

Longest repeated and common substring

Given a string S, we want to compute the longest repeated substring in S. A naive solution for this will loop through the starting position of the first repeat and the second repeat, and check whether there is a match of some length- this takes $O(n^3)$ time. We can use a dynamic programming algorithm to make the algorithm more efficient.

In the algorithm, we store the characters of the string as both rows and columns. If there is a match at a cell, we make the entry 1; otherwise, the entry is 0. The longest common substring corresponds to the greatest sum we can produce when we traverse the long diagonals. The following example illustrates this algorithm to find the longest repeated substring of `ababa`:

	a	b	a	b	a
a	-	0	1	0	1
b	-	-	0	1	0
a	-	-	-	0	1
b	-	-	-	-	0
a	-	-	-	-	-

We only compute the entries on the right side of the diagonal as the values are symmetric. There is always a match at the diagonal- we are not interested in that. In this case, the longest repeated substring is `aba`. Note that the value `a` is shared between the two substrings- this is allowed because the index of the matching `a` is still different.

This algorithm has complexity $O(n^2)$ in both time and space.

We can easily modify this dynamic programming algorithm to compute the longest common substring of two strings. The following example illustrates this.

	a	b	c	a	b
b	0	1	0	0	1
b	0	1	0	0	1
c	0	0	1	0	0
a	1	0	0	1	0
a	1	0	0	1	0

In this case, we need to compute both sides of the diagonal, and the diagonal, since the two strings are assumed to be different. Here, the longest common substring is `bca`.

This algorithm has complexity $O(mn)$ in both time and space, where m and n are the lengths of the two strings.

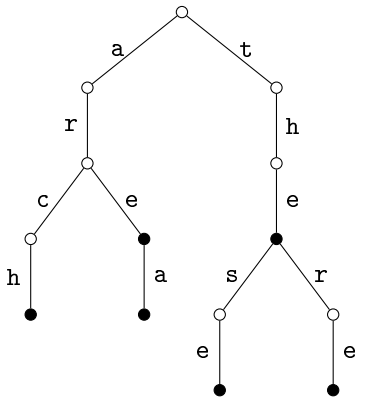
Suffix Tries

We will now improve the time (and space) complexity of these algorithms using suffix tries. For a string S, the suffix tree of S stores all the suffixes in S including the entire string. Using this representation, we can compute the longest repeated substring in $O(n)$ time and longest common substring in $O(m+n)$ time. Moreover, we can search multiple strings (whose sum of length is r) in $O(n+r)$ time.

A trie is a branching tree that is used to store strings in an alphabet. It has the following properties:

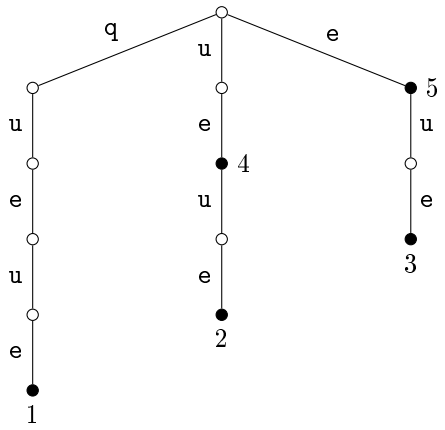
- a trie has a root vertex;
- every edge in the trie corresponds to a character from the alphabet;
- no two children of a branch node have the same edge label;
- every node corresponds to a string in the alphabet, which is given by traversing the tree from the root to the given node;
- every node is marked in a way to recognise whether it corresponds to one of the given words.

An example of the trie on the words- `arch`, `are`, `area`, `the`, `there` and `these` is given below:

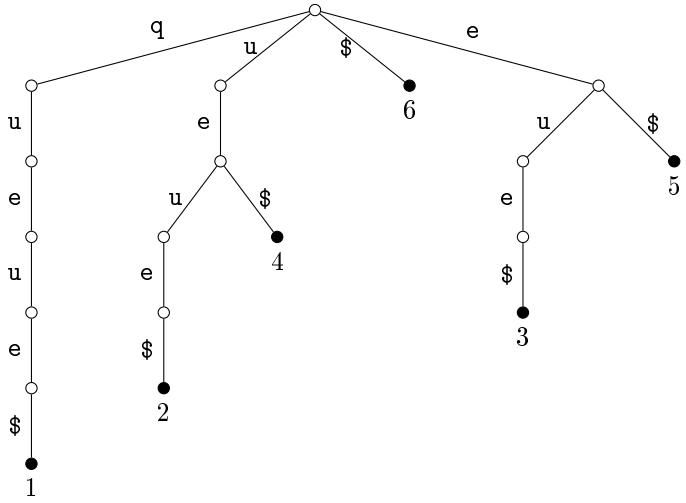


We mark by black any node that corresponds to a string.

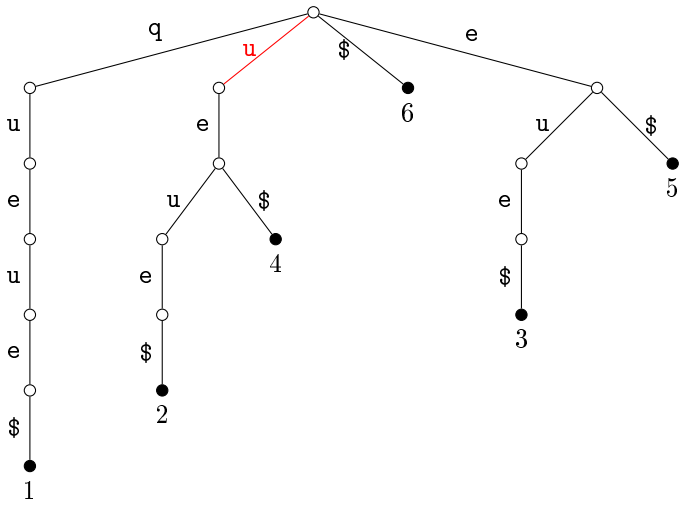
A suffix trie is a trie that stores all the suffixes of a given string. For example, the suffix trie of `queue` is given below.



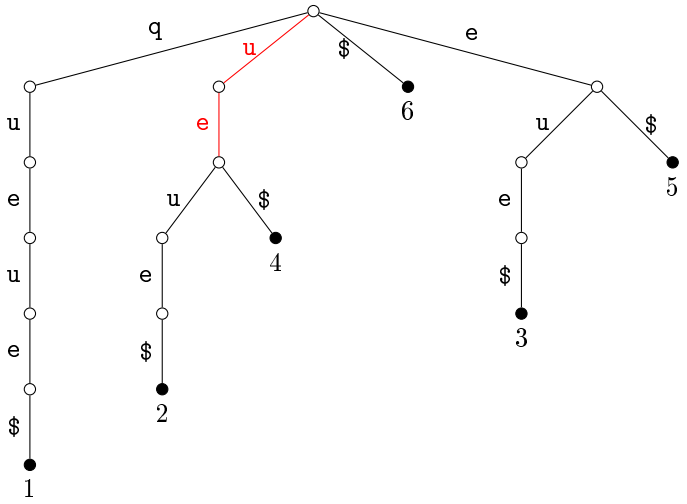
For a suffix trie, we also want each suffix to end at a leaf node. This is not the case in the trie above- the suffixes `ue` and `e` end at branch nodes. This is because there is a suffix (`ue`) that is a prefix of another one (`ueue`). To fix this issue, we add a unique termination symbol `$` that is not present in the alphabet. This ensures that no suffix is a prefix of another suffix. In this case, the suffix trie for `queue` is the following:



We will now use the suffix trie above to find `ue`. We use the suffix trie to descend from the root to the branch corresponding to the character `u`.



We then descend from this branch to the one corresponding to the character `e`.



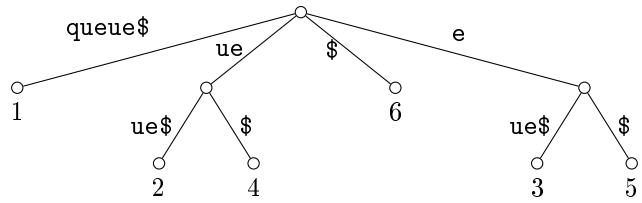
We have now completely matched the substring. This means that the substring occurs at indices 2 and 4. Using the suffix trie, we were able to find all the occurrences in linear time with respect to the substring (assuming that the alphabet has constant size). If we had tried to match `ux` instead, we would not have been able to find a branch for the second letter. That implies that there is no match in the string.

We can use the suffix trie to also find the longest repeated substring. This corresponds to finding the branch node with the greatest string depth. A branch node is a prefix of at least 2 suffixes, and so represents a repeated substring. For the string `queue`, the repeated substrings, as seen in the trie, are `e` and `ue`. Hence, the longest repeated substring is `ue`, with length 2, with starting indices 2 and 4.

To build a suffix trie, we repeated insert all the suffixes to the trie. To insert the suffix i, it takes $O(n-i)$ time. So, overall, it takes $O(n^2)$ time and space. This is not tractable for long texts. To improve this, we make use of suffix trees.

Suffix Trees

Suffix trees are very similar to suffix tries, but we have compressed unary branch nodes. So, the suffix tree for `queue` is the following:



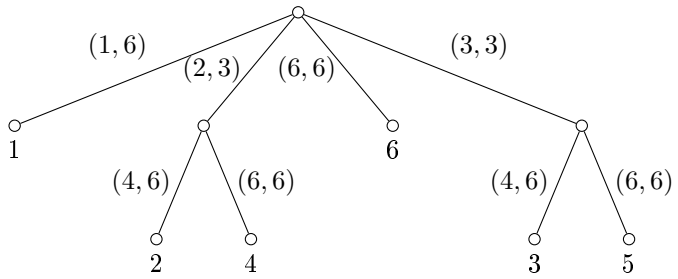
This is a much more efficient representation since we do not need to store a node for each letter. In this case, the number of nodes is $O(n)$, whereas for the suffix trie, it was $O(n^2)$.

Now, we define the suffix tree for some string S. We first append \$ to the string and construct all the suffixes that satisfy the following properties:

- each edge has a string label;
- all branch nodes have at most 2 children;
- no two children of a branch node have two edges that started with the same character;
- there is a bijection between the leaf nodes and the suffixes, i.e. we can descend from the root to a leaf to recover the suffix. The entire label (from root to a node) is called its *path label*.

In a suffix tree, there are $n+1$ suffixes, so $n+1$ leaf nodes. Also, the number of branch nodes is at most n. To see this, assume that we have x edges, n leaf nodes and b branches. Then, $x = b + n - 1$- we have an edge going to each branch node and leaf node, except for the root. Moreover, since every branch node has at least 2 children, $x \geq 2b$. Hence, $b \leq n - 1$. Hence, there are $O(n)$ nodes in total. This is an improvement from suffix tries, which have $O(n^2)$ nodes.

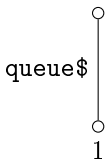
However, the suffix tree still has $O(n^2)$ space complexity- we need linear space to store the entire suffix. To mitigate this, we just store the index of the start and the end of the substring in the text. So, the suffix tree for `queue` is the following:



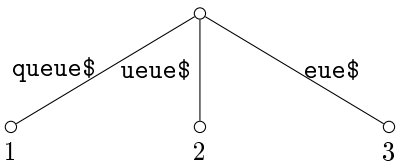
Each label now takes $O(1)$ space, so the total complexity is $O(n)$ space. We call this representation the *suffix tree with short edge labels*, and the one above as the *suffix tree with full edge labels*.

One way to construct the suffix tree with short edge labels is to create the suffix trie, use it to construct the suffix tree with full edge labels by collapsing unary branches, and then replace the full edge labels with short edge labels. This is not the most efficient approach since we do need $O(n^2)$ space during the algorithm.

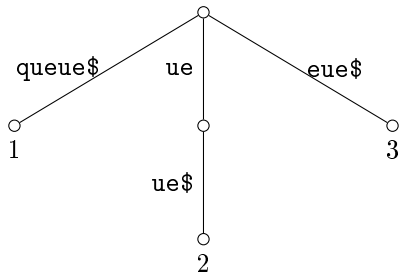
A more efficient approach would involve directly constructing the suffix tree with short edge labels. We illustrate this with an example, by constructing the suffix tree with full edge labels for `queue`. We will use full edge labels for illustration purposes; the algorithm would instead make use of the short edge labels. We start by inserting the suffix `queue$`.



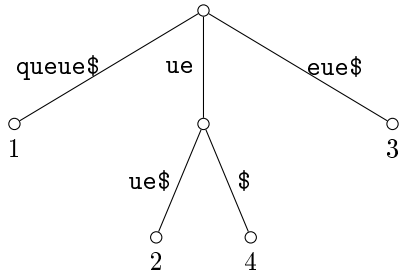
We insert the suffix by branching from the root. We can do the same thing when inserting the next two suffixes- `ueue$` and `eue$`.



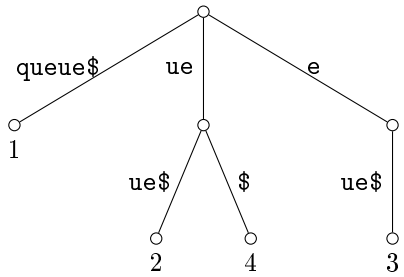
At this point, the next suffix is `ue$`. We already have a node starting with `ue`, but this is not a complete match (i.e. the branch has a longer label). So, we perform an *edge-split* first to ensure we can branch at the right position.



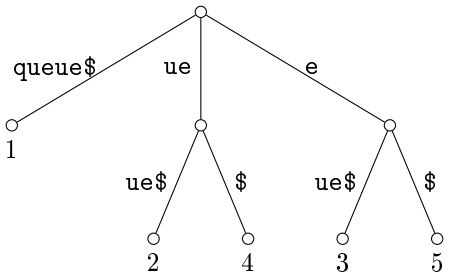
We can now insert `ue$` as a child of the branch `ue`.



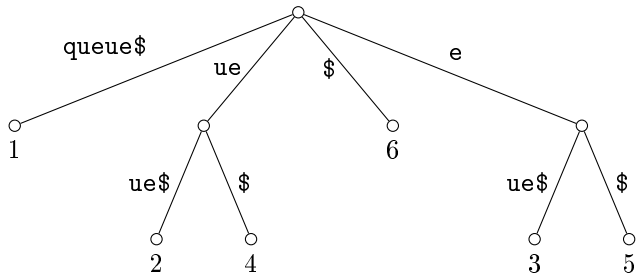
We have to do the same thing when inserting `e$`- we edge-split `eue$`.



We can now insert `e$` to the tree.



Next, we insert `$`.



The suffix tree is now complete.

This algorithm takes $O(n)$ space complexity, but has worst-case complexity $O(n^2)$, e.g. if we have to edge-split every time. The average case is $O(n \log n)$. There are more efficient algorithms that require $O(n)$ time complexity.

We will now use the suffix tree for different string algorithms. If we have a string S and a substring T to search in the string, we construct a tree on S, and then match each letter in T with the corresponding node. If we cannot match T completely, then the string is not present. If it is present, we can return the index of the match by using the branch node we are at. This algorithm is very similar to the one for suffix tries we saw above. The time taken for this algorithm is linear with respect to the substring. We can also find all occurrences using this approach.

Now, we try to find the longest repeated substring in a text S. We can construct the suffix tree for S in $O(n)$ time. We then traverse the tree to find the branch node with the greatest string depth. A branch node corresponds to a common substring, and the string depth keeps track of the length of the common substring. We can traverse in $O(n)$ time, so the entire algorithm is $O(n)$.

Next, we can find a longest common substring of texts S and T. To do so, we construct the suffix tree for $S\#T\$$, where $\#$ and $\$$ are symbols not in Σ. This is called the *generalised suffix tree*. We then traverse the tree in a similar way as the longest repeated substring, but we are looking for a common branch node with the greatest string depth. A common branch is one that has leaf nodes corresponding to suffix starting in both S and T.

Now, we will look at a method to identify common branch nodes. For each branch node v, we keep track of two values- whether it has a leaf node with suffix value corresponding to a string in S ($b_1(v)$), and whether it has a leaf node with suffix value corresponding to a string in T ($b_2(v)$). Then, v is a branch node if and only if $b_1(v) \wedge b_2(v)$. We can compute the $b_i(v)$ values as follows:

- if v is a leaf node, then we can check the value of the suffix to compute $b_i(v)$- $b_1(v)$ is `true` if $1 \leq j \leq m$ and $b_2(v)$ is `true` if $m+2 \leq j \leq m+n+1$, where m and n are the lengths of S and T respectively;
- instead, if v is a branch node, then $b_i(v) = b_i(w_1) \wedge b_i(w_2) \wedge \cdots \wedge b_i(w_n)$, where $w_1, w_2, \ldots, w_n$ are the descendant leaf nodes of v.

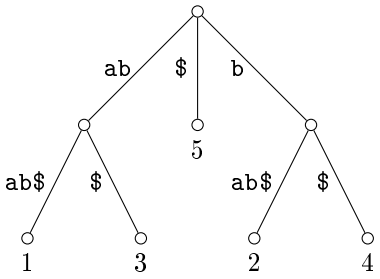
So, the algorithm for longest common substring is the following:

- build the generalised suffix tree;

- calculate the values $b_i(v)$ for every node v;
- find the common branch node with maximum string depth.

The last 2 steps can be combined. Each step takes $O(m+n)$ time, so the algorithm overall takes $O(m+n)$ time.

We will now illustrate that the character # is required in order to correctly output the result. So, assume that the two strings are `a` and `bab`. Then, the suffix tree for $ST\$$ is the following:



Here, the common branch node with the greatest depth is `ab`. However, it is not a common substring of `a` and `bab`.

2.2 Matching Regular Expressions

In this section, we will look at searching a regular expressions in some text. There are many algorithms to search a string in some text, including brute-force, KMP, BM and suffix trees. We will generalise the brute-force algorithm to match regular expressions.

A *regular expression* represents a set of strings on some alphabet Σ, called a *language*. For example, if the regular expression is `a|b`, then the language is: $\{\texttt{a}, \texttt{b}\}$. We say that a string *matches* a regular expression if it is an element of the language.

We will now define regular expressions over some alphabet Σ.

- the empty string ϵ is a regular expression, with language $\{\epsilon\}$;
- for a character $\sigma \in \Sigma$, σ is a regular expression, with language $\{\sigma\}$;
- for regular expressions `R` and `S` with languages $L_{\texttt{R}}$ and $L_{\texttt{S}}$, their concatenation `RS` is a regular expression with language

$$L_{\texttt{RS}} = \{XY \mid X \in L_{\texttt{R}}, Y \in L_{\texttt{S}}\}.$$

- for regular expressions `R` and `S` with languages $L_{\texttt{R}}$ and $L_{\texttt{S}}$, their choice `R|S` is a regular expression, with language $L_{\texttt{R|S}} = L_{\texttt{R}} \cup L_{\texttt{S}}$.
- for a regular expression `R` with language $L_{\texttt{R}}$, its closure $\texttt{R}^*$ is a regular expression, with language

$$L_{\texttt{R}^*} = \{\epsilon\} \cup \{XY \mid X \in L_{\texttt{R}}, Y \in L_{\texttt{R}^*}\}.$$

- for a regular expression `R` with language $L_{\texttt{R}}$, its bracketed form `(R)` is a regular expression with language $L_{\texttt{(R)}} = L_{\texttt{R}}$.

When we unpack a regular expression, the following is the order of precedence:

- the bracket has the highest precedence (and can be used to override any of the precedence rule below);
- the closure operation;
- the concatenation operation; and
- the choice operation.

Hence, the regular expression $\texttt{ab}^*\texttt{d}$ has the language

$$\{\texttt{ad}, \texttt{abd}, \texttt{abbd}, \dots\},$$

while $(\texttt{ab})^*\texttt{d}$ has the language

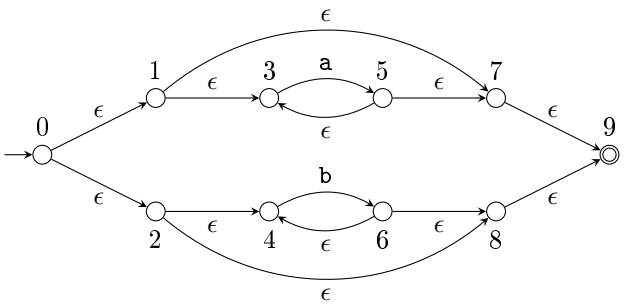
$$\{\texttt{d}, \texttt{abd}, \texttt{ababd}, \dots\}.$$

There are further regular expressions such as complement `-x` which refers to all characters in Σ except `x`, and `?` which corresponds to any character in Σ.

We will now look at the problem- given a text `T` and a regular expression `R`, check whether `T` matches `R`. The algorithm we will look at considers each position in text, and checks whether we have had a match with a substring ending at that position. To do so, we need to keep track of all prefixes of `R` that are matched at any given position. We will use an non-deterministic finite automaton (NDFA) to do so.

An NDFA is a directed graph with vertices, called states. There are always two states- a starting state and a halting state. The edges are called transitions, and have an label- a character in the alphabet Σ, including the empty character ϵ. We say that an NDFA *accepts* a string if there exists a path from the starting state to the halting state that matches all the characters in the string using the edge labels.

Every regular expression can be represented by an NDFA, and vice versa (though neither representation is unique). For instance, an NDFA for the regular expression $\texttt{a}^*|\texttt{b}^*$ is:

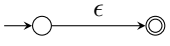


The starting state is 0, shown by an arrow coming to it. The halting state is 9, which is shown by double circle. Some of the strings it accepts are given below:

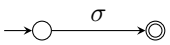
- the empty string, which can be matched by following the path: 0−1−7−9;
- the string `b`, which can be matched by following the path: $0-2-4-6-8-9$; and
- the string `aa`, which can be matched by following the path: $0-1-3-5-3-5-7-9$.

We will now systematically construct an NDFA given a regular expression. This representation will have different properties that will allow us to conclude the efficiency of the algorithm. The following is the way we construct an NDFA:

- The following is the NDFA for the empty string:

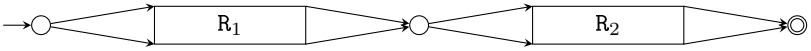


- The following is the NDFA for a character σ:



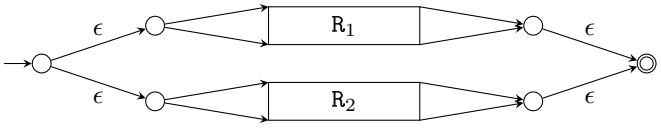
- Let the NDFAs for $\texttt{R}_1$ and $\texttt{R}_2$ be the following:

Then, the NDFA for $\texttt{R}_1\texttt{R}_2$ is the following:

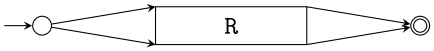


- Let the NDFAs for $\texttt{R}_1$ and $\texttt{R}_2$ be the following:

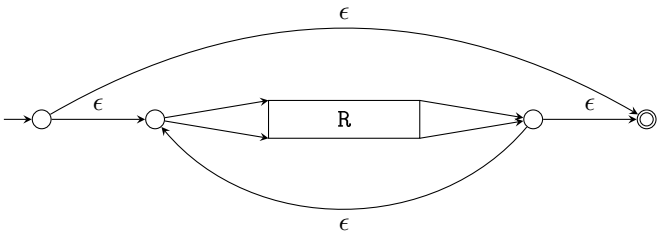
Then, the NDFA for $\texttt{R}_1|\texttt{R}_2$ is the following:



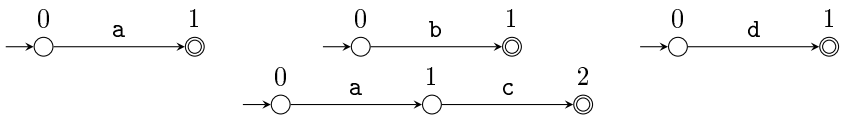
- The NDFA for `(R)` is the same as `R`.
- Let the NDFA for `R` be the following:



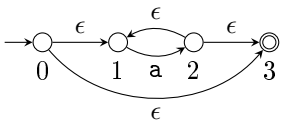
Then, the NDFA for $\texttt{R}^*$ is the following:



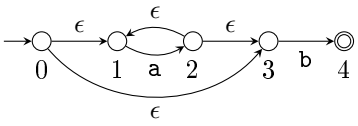
We illustrate the construction of the NDFA for the regular expression $(\texttt{a}^*\texttt{b}|\texttt{ac})\texttt{d}$. The NDFA for `a`, `b`, `ac` and `d` are the following:



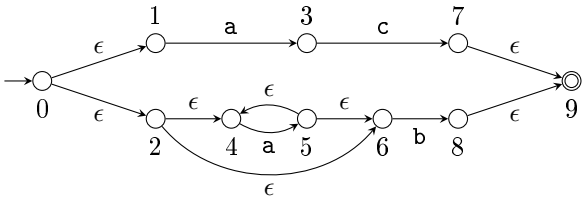
So, the regular expression for $\texttt{a}^*$ is the following:



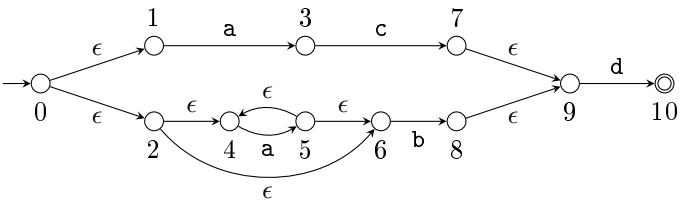
Now, the regular expression for `a*b` is the following:



Next, we construct the regular expression for `a*b|ac`.



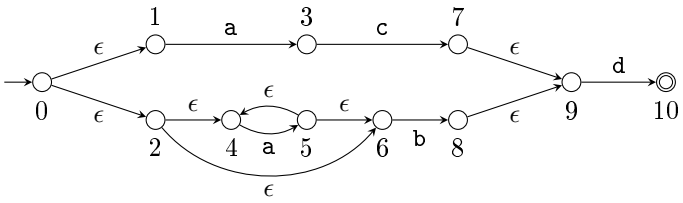
Finally, the regular expression for `(a*b|ac)d` is the following:



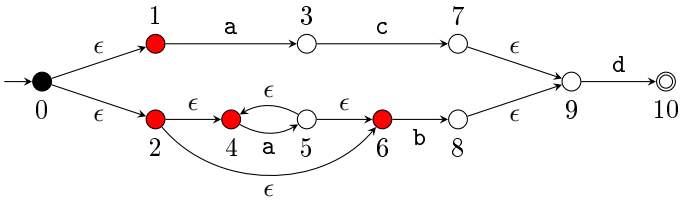
By constructing the NDFA this way, we can guarantee that there is precisely one start and halt state. This follows from construction. Also, if the regular expression has length r, then there are at most $2r$ states in the NDFA. This is because every node has either one edge labelled by a character, or at most 2 edges labelled by ϵ.

We will now consider how the algorithm works. Given a string s and a regular expression `R`, we first construct the NDFA for `R` as given by the algorithm above. Then, we run the algorithm by iterating through the characters in s. We keep track of all the states we can get to by following s up to some index (and taking any ϵ-edges). If we can reach the halting state at any point, we will have matched the string. We will make use of two functions to do so- `extendByEpsilon`, which extends the list of states by the ones that can be reached from this list and any ϵ-edges; and `extendByChar`, which tells us what the next state would be if we took the character given from any of the states present.

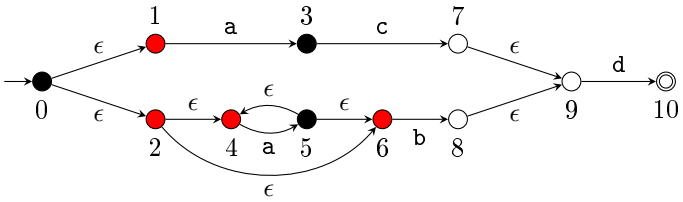
We illustrate the algorithm with an example- we have the regular expression `(a*b|ac)d` and the text `cabaabd`. The NDFA for the regular expression is the following:



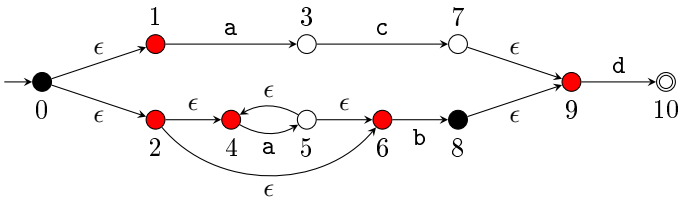
At the start, we can only be at the starting state. Nonetheless, there are some ϵ-edges that we can take from the start. This is shown below.



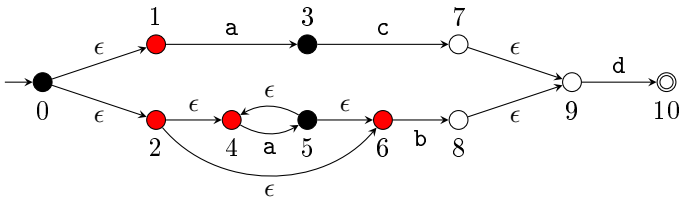
The reached states are given in black and the extended states are given in red. If the state 10 was already red (or black), the algorithm would be over. However, this is not the case, so we enter the main loop of the algorithm. We start by matching `c`. There is no `c` that can be reached from any of the black or the red states. At any point, we can restart and reach the initial state. Hence, the state of the NDFA remains as given above. Next, we move to `a`. In this case, we can match an `a` and get further in the NDFA:



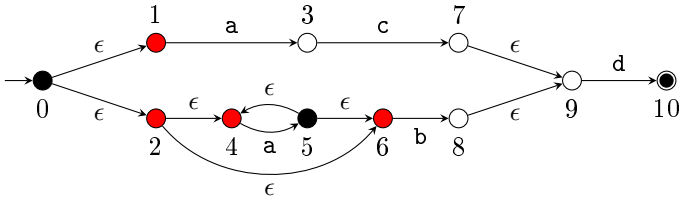
We still haven't reached the final state, so the algorithm has not yet terminated. We next match `b` and continue along.



We have now reached state 9, but still not 10. So, we continue on with the algorithm. The next letter is `a`. We cannot go further from 8 or 9 using `a`, so we are back at the previous state:



We are at the same state when we match the `a` again. When we match `b`, we get the same state as above. Finally, we match `d`. In that case, the state of the NDFA is the following:



We have now reached the final state, meaning that we have found a match of the regular expression in the text ending at position. In this case, it is `abd`.

The pseudocode for the algorithm is given below:

```
bool match(String string, NDFA ndfa) {
    Set<NDFAState> y = {ndfa.startState};
    Set<NDFAState> x = extendByEpsilon(y);

    // if the ndfa accepts the empty string, there os a match
    if (x.contains(ndfa.haltState)) {
        return true;
    }

    for (int i=0; i<string.length; i++) {
        y = extendByChar(x);
        y.add(ndfa.startState);

        x = extendByEpsilon(y);
        if (x.contains(ndfa.haltState)) {
            return true;
        }
    }

    return false;
}
```

We will now consider the complexity of the algorithm. Let r be the length of the regular expression and t the length of the text. We loop at most t times. Within each iteration, we use `extendByEpsilon` and `extendByChar`. The function `extendByChar` has complexity $O(n)$, where n is the number of states in the NDFA- we compute `extendByChar` for each state in $O(1)$ time and combine the result. The function `extendByEpsilon` uses a depth first search approach, and has complexity $O(n+m)$, where m is the number of edges. We know that an NDFA constructed as above has at most $2r$ states, and each state has at most 2 edges. Hence, the algorithm is $O(rt)$.

2.3 Longest Common Subsequence

In this section, we will consider computing the longest common subsequence in different ways. A subsequence of a string can be constructed by removing some characters from it, while maintaining the same order. We want to compute the longest common subsequence for two strings- its length and the actual subsequence. Typically, the longest common subsequence is not unique. Longest common substring is a measure of global similarity of two strings.

Our first approach for the computation is dynamic programming (DP). Here, we build the solution by computing smaller solutions. We will first look at iterative DP. When computing the LCS of two strings `s1` and `s2`, we have a table with the row label the characters in `s1` and the column label the characters in `s2`, including the empty character. We fill the first row and column with 0s, and recursively define

$$\texttt{table}[i,j] = \begin{cases} 1 + \texttt{table}[i-1, j-1] & \texttt{s1}[i-1] == \texttt{s2}[j-1] \\ \max(\texttt{table}[i-1, j], \texttt{table}[i, j-1]) & \text{otherwise.} \end{cases}$$

The length of the LCS is given at the bottom right position. We illustrate this with an example- the table below is the DP table where we compute the LCS of `acbacadb` and `abadcda`:

	ϵ	a	c	b	a	c	a	d	b
ϵ	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1
b	0	1	1	2	2	2	2	2	2
a	0	1	1	2	3	3	3	3	3
d	0	1	1	2	3	3	3	4	4
c	0	1	2	2	3	4	4	4	4
d	0	1	2	2	3	4	4	5	5
a	0	1	2	2	3	4	5	5	5

In the DP table, we have computed the LCS of different substrings of the 2 strings. For example, the LCS of any string with the empty string is 0- this is given in the first row and the first column. Moreover, the LCS of `acba` and `aba` is 3. In particular, the LCS of the two strings is 5- this is the bottom right entry.

The algorithm for this version of LCS, to compute the length, is given below:

```
int lcs(String s1, String s2) {
    // initialise the 2D table with 0s
    List<List<int>> table = List(s1.length+1,List(s2.length+1,0));

    // use the formula to fill the table
    for (int i=1; i<s1.length+1; i++) {
        for (int j=1; j<s2.length+1; j++) {
            if (s1[i-1] == s2[j-1]) {
                table[i][j] = 1 + table[i-1][j-1];
            } else {
                table[i][j] = max(table[i-1][j], table[i][j-1]);
            }
        }
    }

```

```
    return table[s1.length][s2.length];
}
```

We will now consider the complexity of the algorithm. It takes $O(1)$ time to fill in each entry of the table. So, the algorithm takes $O(mn)$ time and space, where m and n are the lengths of `s1` and `s2` respectively.

We will now look at constructing the LCS using the DP table. We do this by backtracking from the bottom right entry- what choice did we make to get to that value. Every time we go up diagonally, we know that there was a match, so this character is part of the common subsequence. We illustrate this with the LCS for `acbacadb` and `abadcda`:

	ϵ	a	c	b	a	c	a	d	b
ϵ	0	0	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1	1	1
b	0	1	1	2	2	2	2	2	2
a	0	1	1	2	3	3	3	3	3
d	0	1	1	2	3	3	3	4	4
c	0	1	2	2	3	4	4	4	4
d	0	1	2	2	3	4	4	5	5
a	0	1	2	2	3	4	5	5	5

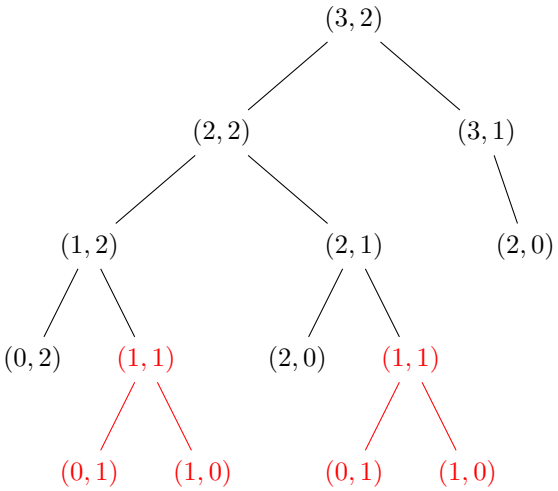
The values in red spell out how we got to the bottom right entry. The blue characters are the ones part of the common subsequence. In this case, it is `abacd`. This is not a unique traceback. For example, instead of going up from the bottom right position, we could have also gone to the left. This can lead to different common subsequences, but they will all have the maximum length possible.

Instead of storing the entire table, we just need to store the previous row (and the diagonal value). This version of the algorithm is given below: This algorithm still has $O(mn)$ time complexity, but the space complexity is $O(n)$. This approach does not allow us to construct the LCS. Nonetheless, there exists an algorithm that takes $O(mn)$ time complexity and $O(n)$ space and computes the actual LCS.

When we construct the DP table iteratively, we compute all the entries. However, this might not be necessary. We can use a *lazy* approach that computes all values that we need in order to find the bottom right entry on the table. This approach uses recursion. A naive approach to do so is the following:

```
// compute the lcs of s1[0..i] and s2[0..j]
int lcs(String s1, String s2, int i, int j) {
    if (i == 0 || j == 0) {
        return 0;
    } else if (s1[i-1] == s2[j-1]) {
        return 1 + lcs(s1, s2, i-1, j-1);
    } else {
        return max(lcs(s1, s2, i-1, j), lcs(s1, s2, i, j-1));
    }
}
```

In this case, we do not necessarily compute all the entries in the DP table. However, this algorithm is very inefficient- it is exponential. This is because many computations are being repeated. For instance, the recursion trace of the values (i, j) when we compute LCS for `caa` and `ab` is shown below:



Even when we are comparing such simple strings, we have 3 common nodes in the recursion trace. Nonetheless, we can see that some combinations of (i, j) are never computed, e.g. $(3, 0)$.

To improve the complexity of the recursive algorithm, we make use of *memoisation*. Here, we use some space to keep track of which values we have already computed, i.e. we maintain an instance of the DP table. This version of the algorithm is given below:

```
// compute the lcs of s1[0..i] and s2[0..j] given the table of
// values (initially -1)
int lcs(String s1, String s2, int i, int j,
    List<List<int>> table) {
    // compute the value if needed
    if (table[i][j] == -1) {
        if (i == 0 || j == 0)  {
            table[i][j] = 0;
        } else if (s1[i-1] == s2[j-1]) {
            table[i][j] = 1 + lcs(s1, s2, i, j, table);
        } else {
            table[i][j] = max(
                lcs(s1, s2, i-1, j, table),
                lcs(s1, s2, i, j-1, table)
            );
        }
    }

    return table[i][j];
}
```

This algorithm has time and space complexity $O(mn)$ due to memoisation, and it only computes the necessary values. We can see this below, where we compute the LCS for `acbacadb` and `abadcda`:

	ϵ	a	c	b	a	c	a	d	b
ϵ	0	-1	0	0	-1	0	-1	0	-1
a	-1	1	1	-1	1	1	1	1	-1
b	0	1	1	2	2	2	2	2	2
a	-1	1	1	2	3	3	3	3	3
d	0	1	1	2	3	3	3	4	4
c	0	1	2	2	3	4	4	4	4
d	0	1	2	2	3	4	-1	5	5
a	-1	-1	-1	-1	-1	-1	5	5	5

Clearly, not all the values have been computed, and we could still recover the LCS. However, we initialise the table with value `-1`- this means that we have already made use of the entire space.

To avoid this, we can declare the list and its size, and not initialise it with any values, and use pointers. In this case, we store the following values:

- the DP table, which is not initialised (and will have garbage values at the start);
- the pointers table, which is also not initialised, that has the same dimension as the DP table;
- a 1D list of pointers, with length equal to the product of the dimension of the DP table; and
- the number of values we have computed in the table.

The pointers table stores at which call the value at table was computed. The pointer list points back to the index in the DP table/pointer (we can convert the 2D index (i, j) to the 1D index $i * m + j$, where m is the length of `s1`). This way, we can avoid initialising the DP table. The algorithm now computes whether a value is correct as follows:

- we first check the pointers table at the given index. If it is less than 0 or bigger than the number of values we have computed, then it is not a valid value, i.e. we have not yet computed the value.
- otherwise, we check whether the 1D list of pointers. We know at this point that the value at this list of pointer is correct (it is one of the values that has been computed). So, if this pointer value points back to the same index in the table of pointers/values, we know that the value in the table is correct.

In this case, the algorithm is the following:

```
1 int lcs(String s1, String s2, int i, int j,
2     List<List<int>> table, List<List<int>> tablePointers,
3     List<int> pointers1D) {
4     bool isValid = false;
5     int val = tablePointers[i][j];
6     // to be valid, val must be between 0 and len = all inserted
      pointers in pointers1D
7     if (val >= 0 && val < len) {
8         // check whether the 1D pointer matches as well
9         isValid = pointers1D[val] == i*s1.length+j;
```

```
10      }
11
12      if (!isValid) {
13          if (i == 0 || j == 0)  {
14              table[i][j] = 0;
15          } else if (s1[i-1] == s2[j-1]) {
16              table[i][j] = 1 + lcs(s1, s2, i, j, table);
17              len++;
18          } else {
19              table[i][j] = max(
20                  lcs(s1, s2, i-1, j, table),
21                  lcs(s1, s2, i, j-1, table)
22              );
23          }
24
25          // change pointer values
26          tablePointers[i][j] = len;
27          pointers1D[len] = i*s1.length+j;
28          len++;
29      }
30
31      return table[i][j];
32  }
```

Although this algorithm takes more space, it avoids initialising the table of values- it is truly lazy. This technique is called *virtual initialisation* (inserting values at unpredictable positions and deciding whether an entry is garbage).