Strings

Arnar Bjarni Arnarson   Atli FF

25. október 2023

School of Computer Science

Reykjavík University

Today we're going to cover

- String matching
  - Naive algorithm
  - Knuth–Morris–Pratt (KMP) algorithm
- Tries
- Aho-Corasick
- Suffix Tries
- Suffix Arrays

String problems

- Strings frequently appear in our kind of problems
  - I/O
  - Parsing
  - Identifiers/names
  - Data
- But sometimes strings play the key role
  - We want to find properties of some given strings
  - Is the string a palindrome?
- Here we're going to talk about things related to the latter type of problems
- These problems can be hard, because the length of the strings are often huge

String matching

- Given a string S of length n,
- and a string T of length m,
- find all occurrences of T in S
- Note:
    - Occurrences may overlap
    - Assume strings contain characters from some alphabet Σ

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$
- Three occurrences:

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$
- Three occurrences:
  - $\mathrm{cabc}\textcolor{red}{\mathrm{aba}}\mathrm{bacaba}$

String matching

Example:

- $S = \text{cabcabababacaba}$
- $T = \text{aba}$
- Three occurrences:
  - cabc**aba**bacaba
  - cabcab**aba**caba

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$
- Three occurrences:
  - cabc**aba**bacaba
  - cabcab**aba**caba
  - cabcababac**aba**

Naive string matching algorithm

- For each substring of length m in S,
- check if that substring is equal to T.

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `   ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `    ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

```
int string_match(const string &s, const string &t) {
    int n = s.size(),
        m = t.size();

    for (int i = 0; i + m - 1 < n; i++) {
        bool found = true;
        for (int j = 0; j < m; j++) {
            if (s[i + j] != t[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            return i;
        }
    }

    return -1;
}
```

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case
- Can we do better?

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: bacbababaabcbab
  - T: abababaca

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `  ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `   ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`
- The number of shifts depend on which characters are currently matched

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

- If, at position i, q characters match (i.e. $T[1 \ldots q] = S[i \ldots i+q-1]$), then
  - if $q = 0$, shift pattern 1 position right
  - otherwise, shift pattern $q - \pi[q]$ positions right

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `    ababaca`

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- Given π, matching only takes $O(n)$ time
- π can be computed in $O(m)$ time
- Total time complexity of KMP therefore $O(n+m)$ worst case

Knuth–Morris–Pratt algorithm

```
vi kmppi(string &p) {
  int m = p.size(), i = 0, j = -1;
  vi b(m + 1, -1);
  while(i < m) {
    while(j >= 0 && p[i] != p[j]) j = b[j];
    b[++i] = ++j;
  }
  return b;
}

vi kmp(string &s, string &p) {
  int n = s.size(), m = p.size(), i = 0, j = 0;
  vi b = kmppi(p), a = vi();
  while(i < n) {
    while(j >= 0 && s[i] != p[j]) j = b[j];
    ++i; ++j;
    if(j == m) {
      a.push_back(i - j);
      j = b[j];
    }
  }
  return a; }
```

Sets of strings

- We often have sets (or maps) of strings
- Insertions and lookups usually guarantee $O(\log n)$ comparisons

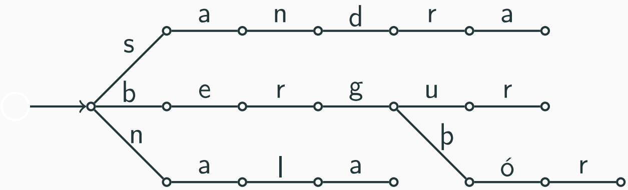
- But string comparisons are actually pretty expensive...

- There are other data structures, like tries, which do this in a more clever way

Tries

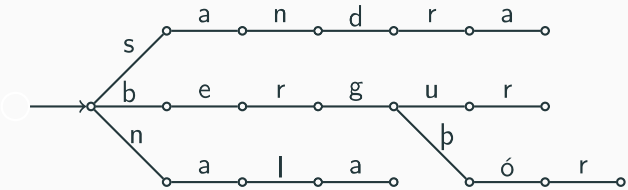
- Tries contain strings not at every node, but as paths in a tree.
- Each node only has a character and we say the trie contains the string if you can get it by walking along nodes starting at the root.
- The nodes can also carry additional data, quite a lot in fact, as we will see later.

Example



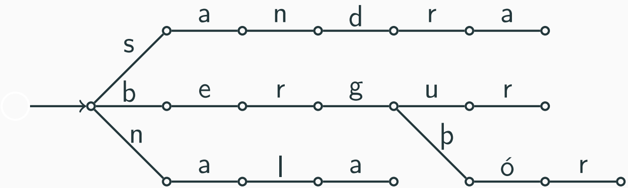
- Examples of strings in this trie include:

Example



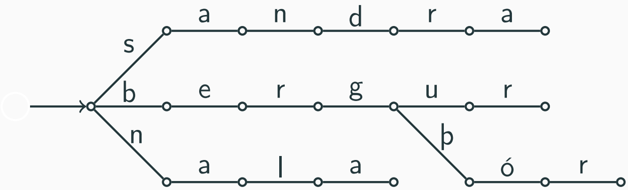
- Examples of strings in this trie include:

Example



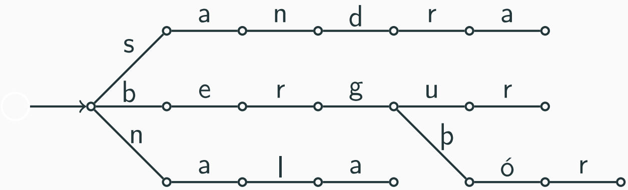
- Examples of strings in this trie include:
    - „sandra",

Example



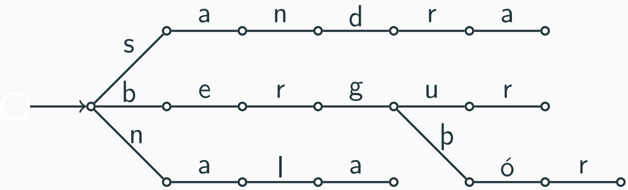
- Examples of strings in this trie include:
    - „sandra",
    - „nala",

Example



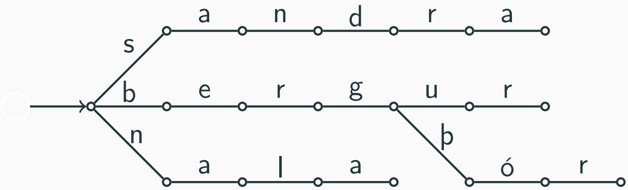
- Examples of strings in this trie include:
    - „sandra”,
    - „nala”,
    - „bergur”,

Example



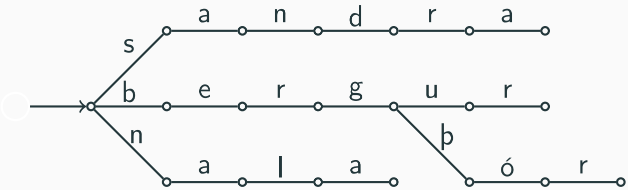
- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór",

Example



- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór",
  - „san" and

Example

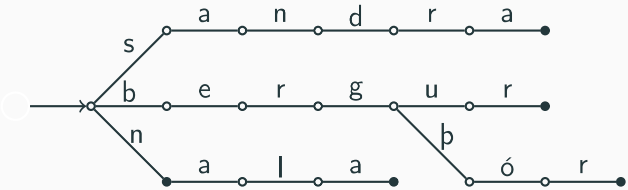


- Examples of strings in this trie include:
    - „sandra",
    - „nala",
    - „bergur",
    - „bergþór",
    - „san" and
    - „" (empty string)

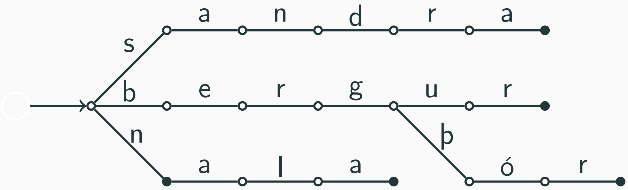
End nodes

- It is common to mark some nodes as end nodes.
- This is an example of extra data to put into nodes.
- Then we can consider a string s to be in the tree if you can walk through the tree to get the string **and** end at an end node.

Example

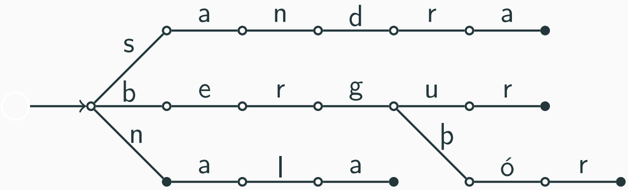


Example



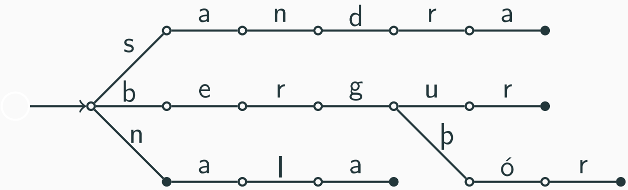
- The strings in the trie are:

Example



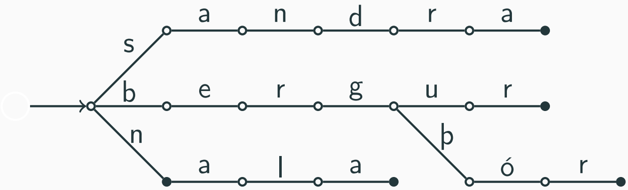
- The strings in the trie are:
    - „sandra",

Example



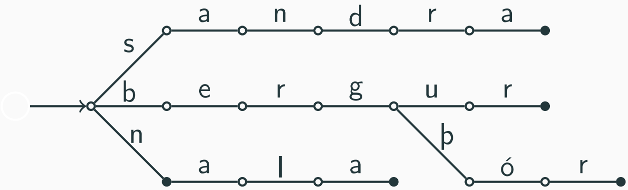
- The strings in the trie are:
  - „sandra",
  - „nala",

Example



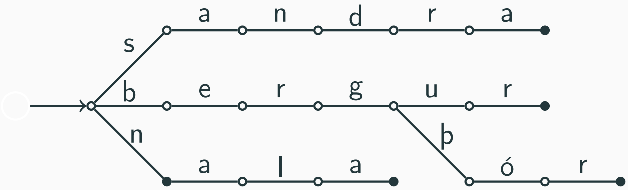
- The strings in the trie are:
    - „sandra”,
    - „nala”,
    - „bergur”,

Example



- The strings in the trie are:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór" and

Example

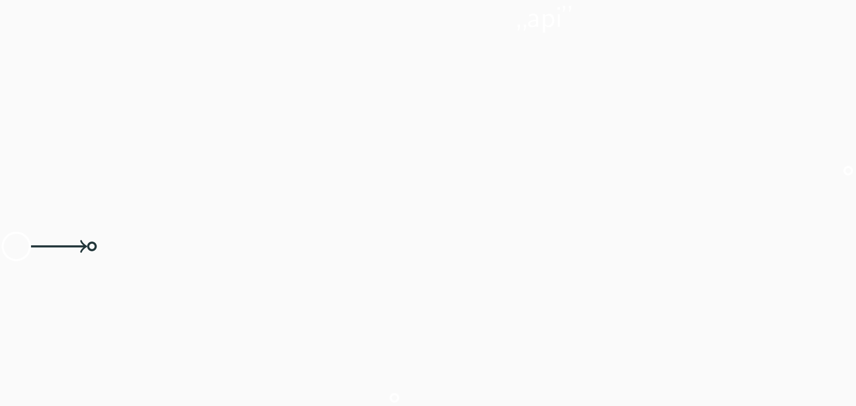


- The strings in the trie are:
    - „sandra",
    - „nala",
    - „bergur",
    - „bergþór" and
    - „n"

Adding strings

- What if we want to add a string to a trie?
- We walk through it as usual, but simply add nodes when we find ourself at a dead end with letters left to walk through.
- This increases the size of the tree by at most the size of the string.

Example



Example

„api"

Example

„api"

Example

Example

Example

Example

Example

„apar”

Example

„apar”

Example

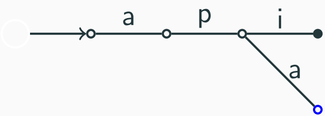
„par"

Example

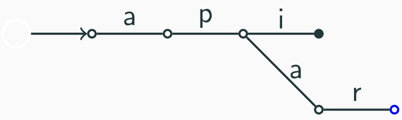
„ar"

Example

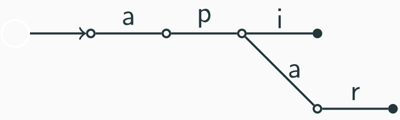
„r”



Example

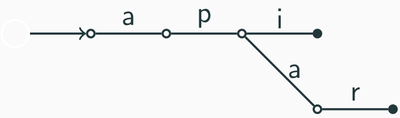


Example



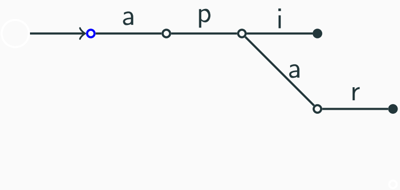
Example

„apaköttur”



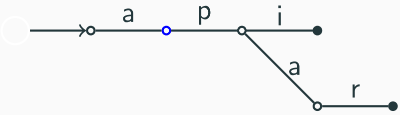
Example

„apaköttur”



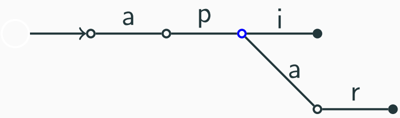
Example

„pakötttur"



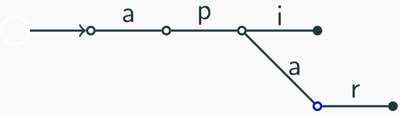
Example

„aköttur"



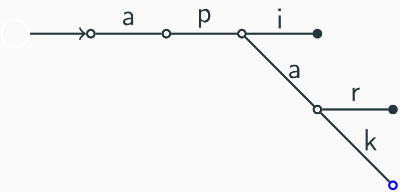
Example

„köttur"



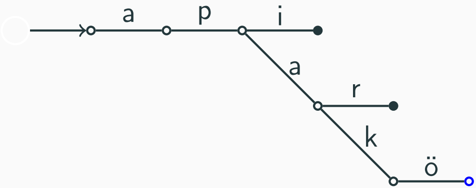
Example

„öttur"



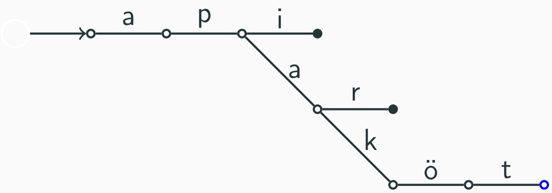
Example

„ttur"



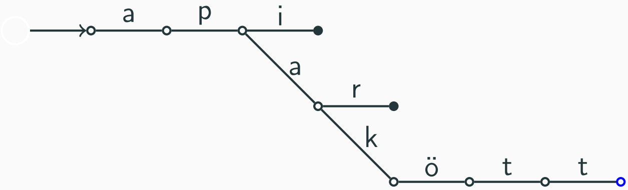
Example

„tur”



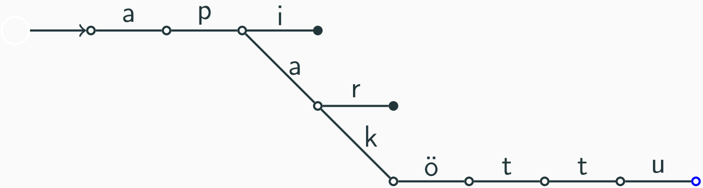
Example

,,ur''

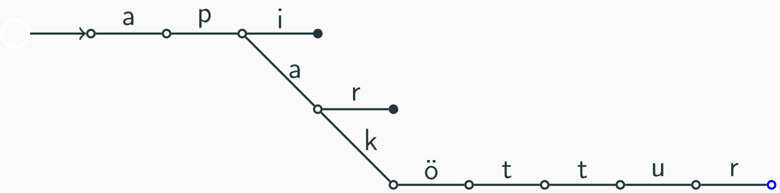


Example

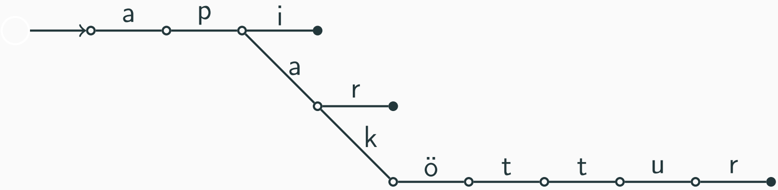
„r”



Example

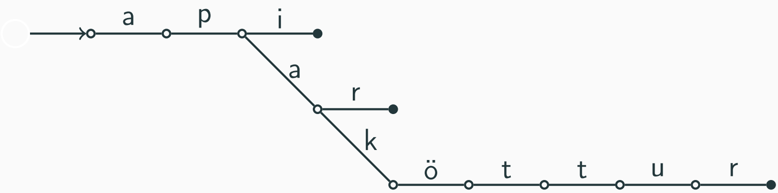


Example



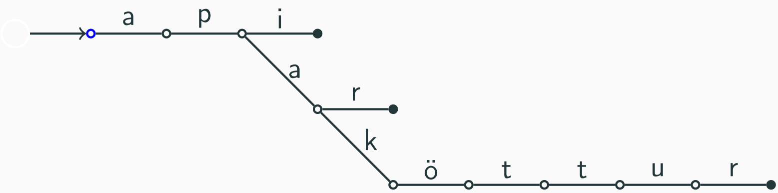
Example

„altari”



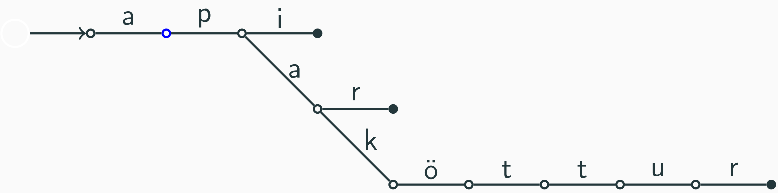
Example

„altari”



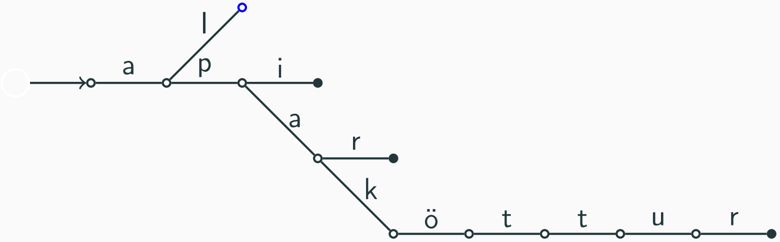
Example

„Itari”



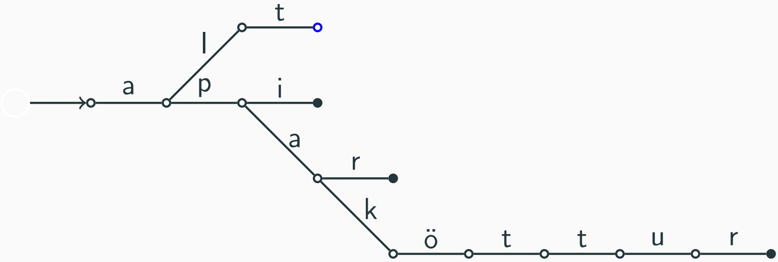
Example

,,tari''



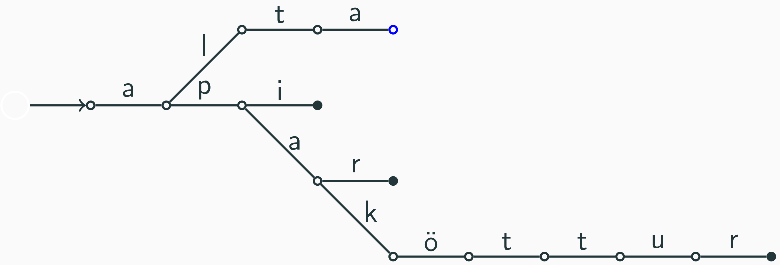
Example

„ari"



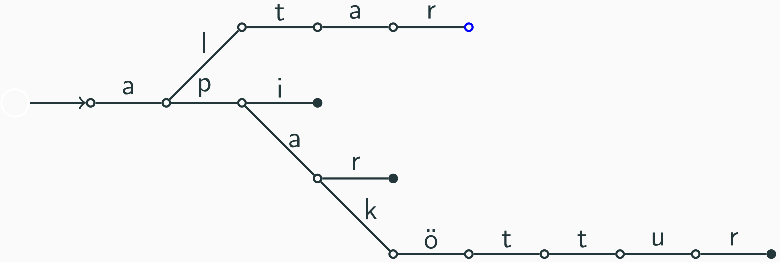
Example

„ri“

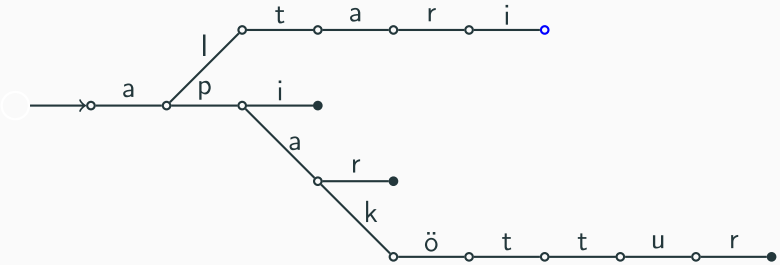


Example

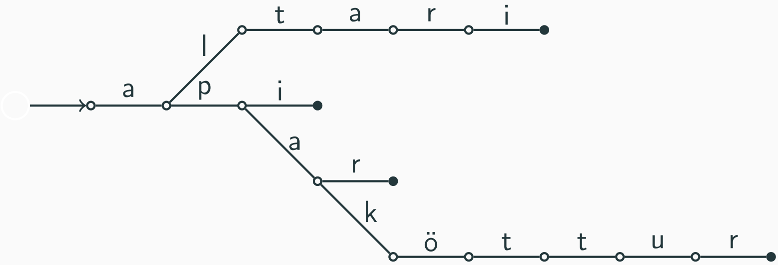
„i“



Example

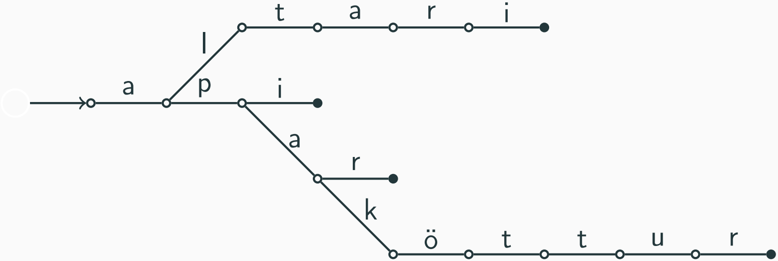


Example



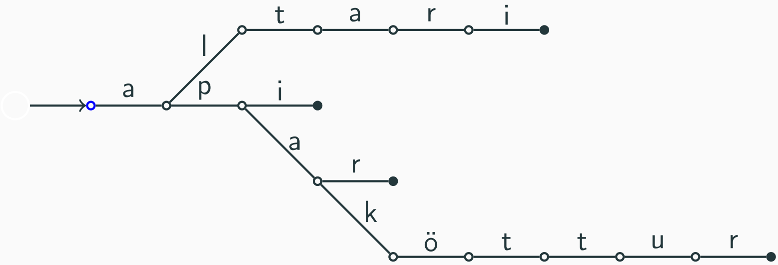
Example

„apaspil"



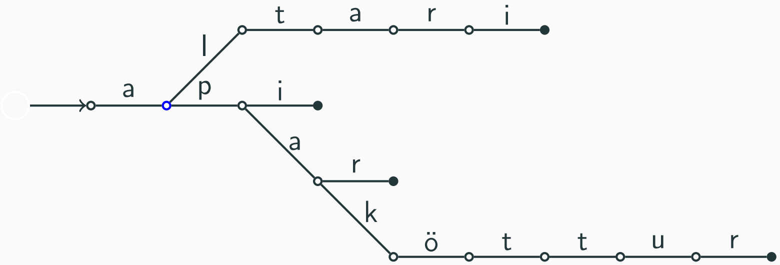
Example

„apaspil"



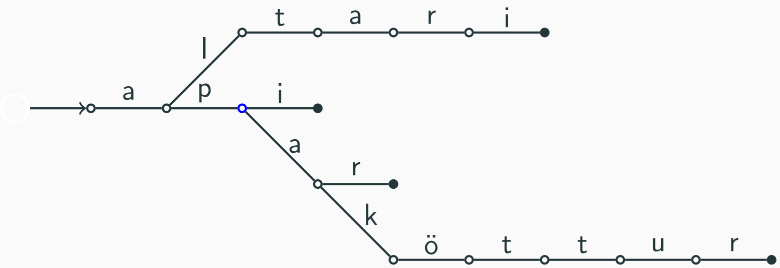
Example

„paspil''



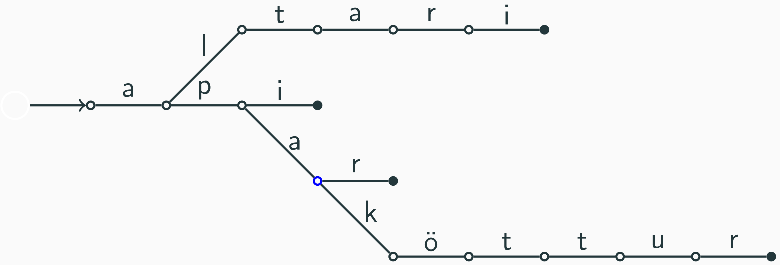
Example

„aspil"



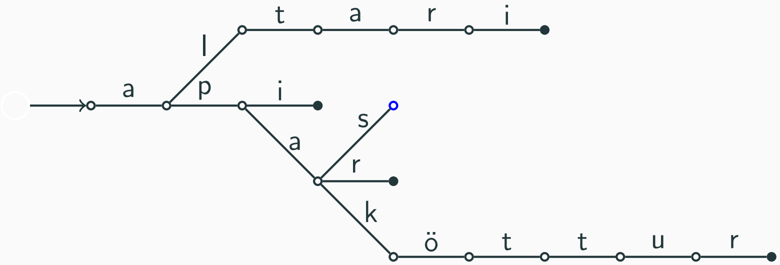
Example

„spil"



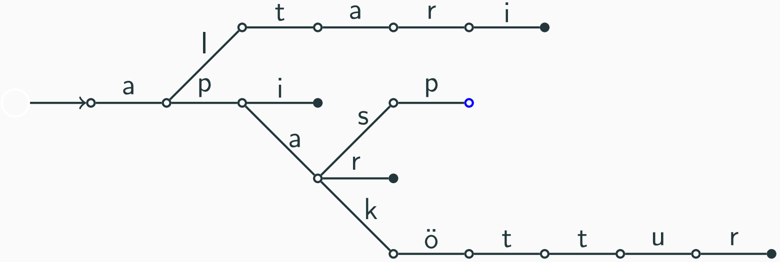
Example

„pil"

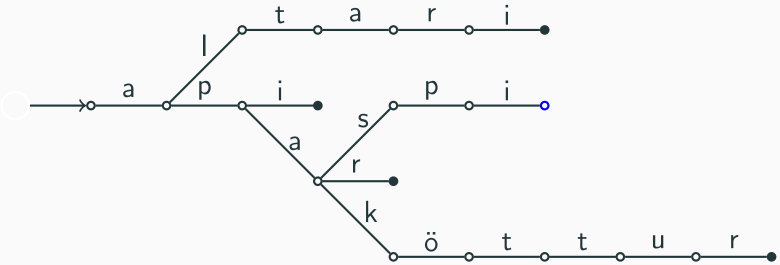


Example

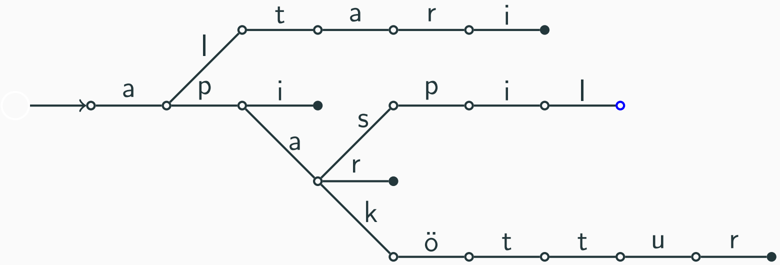
„il”



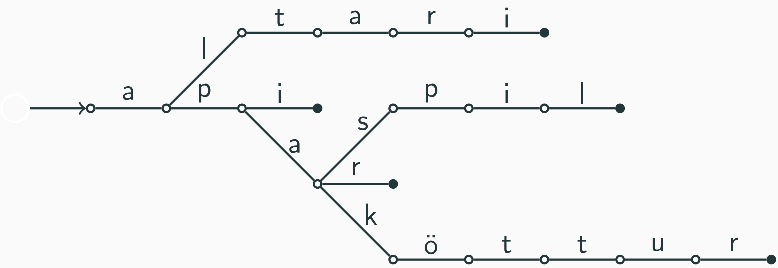
Example



Example

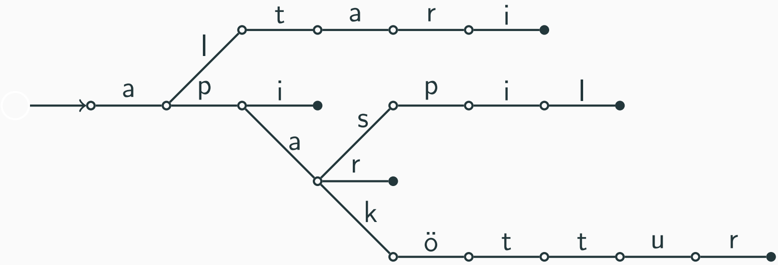


Example



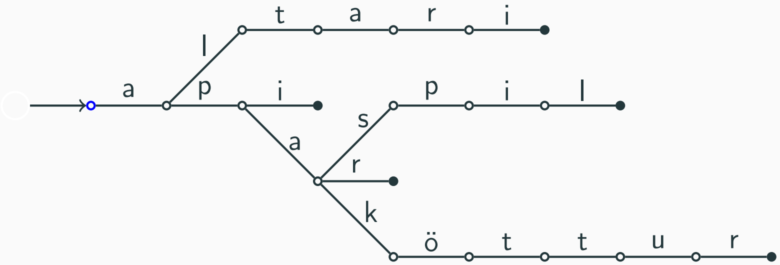
Example

„altaristafla"



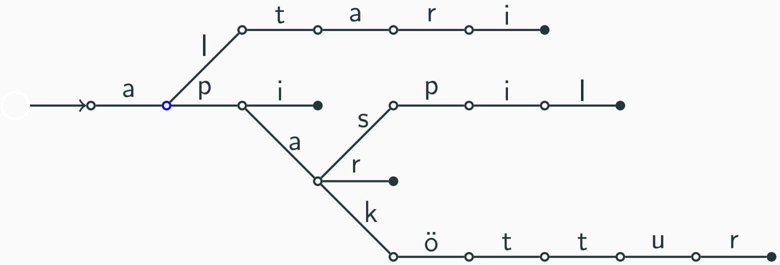
Example

„altaristafla"



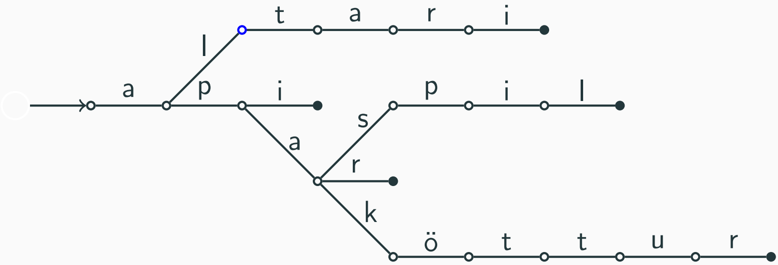
Example

„ltaristafla"



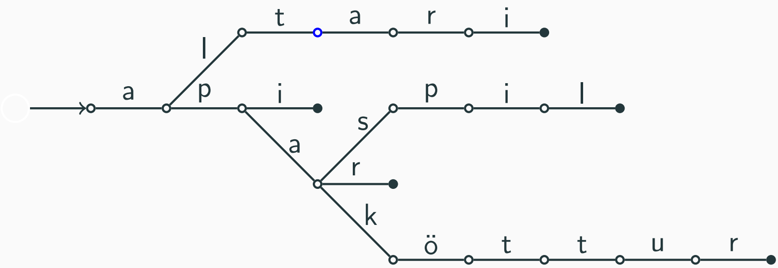
Example

„taristafla"



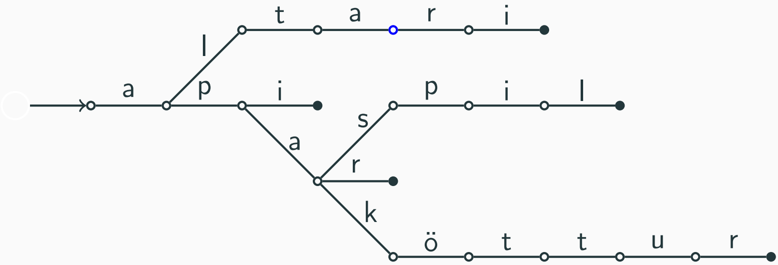
Example

„aristafla"



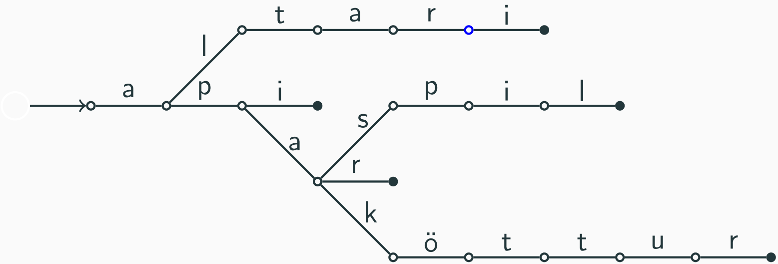
Example

„ristafla"



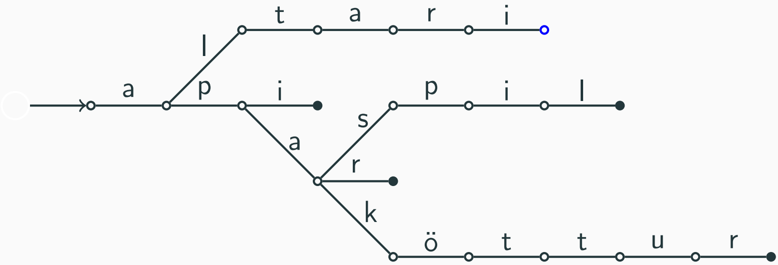
Example

„istafla”



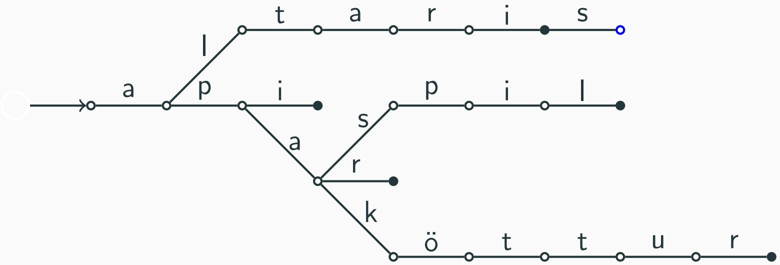
Example

„stafla"



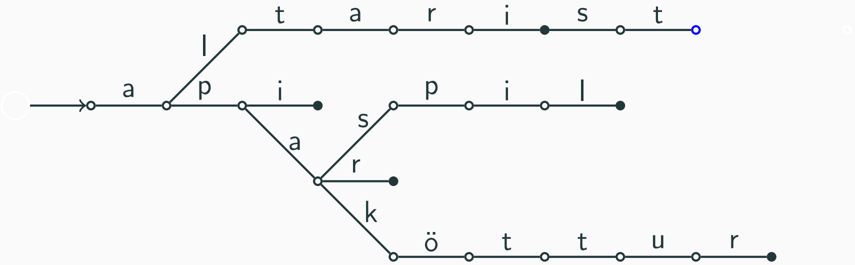
Example

„tafla”



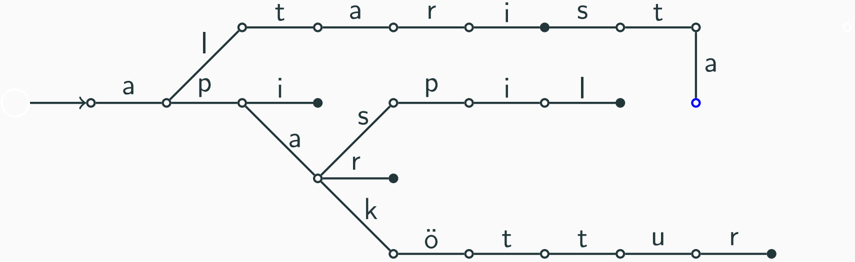
Example

„afla"

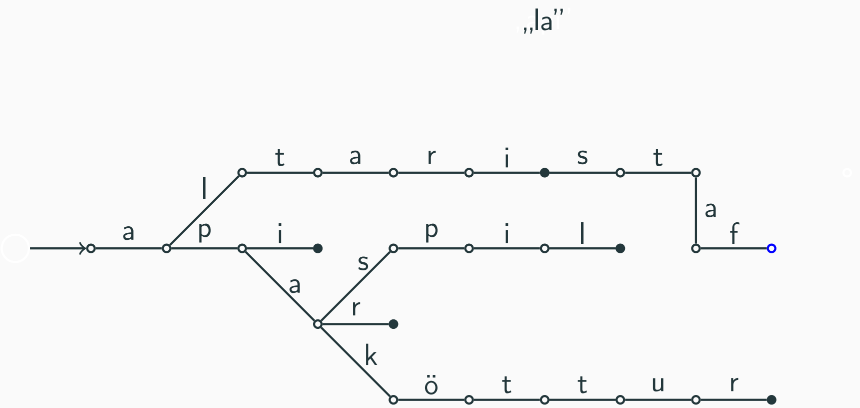


Example

„fla"

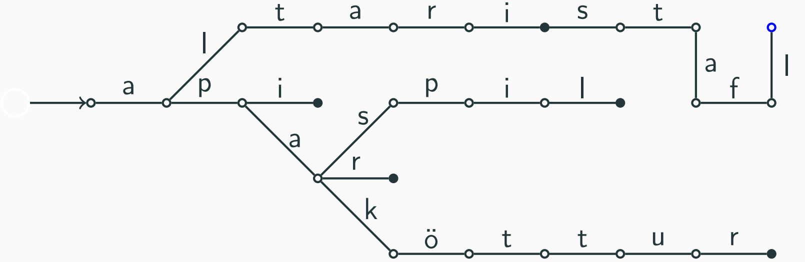


Example

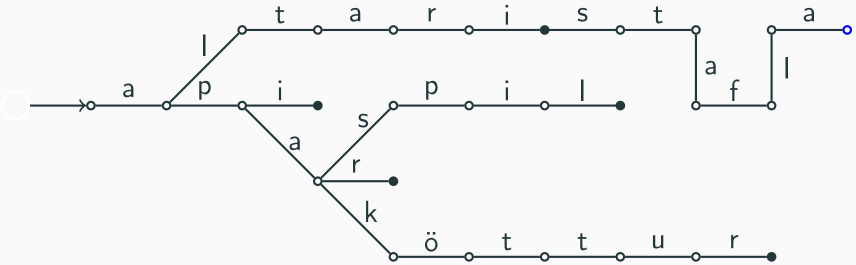


Example

„a“

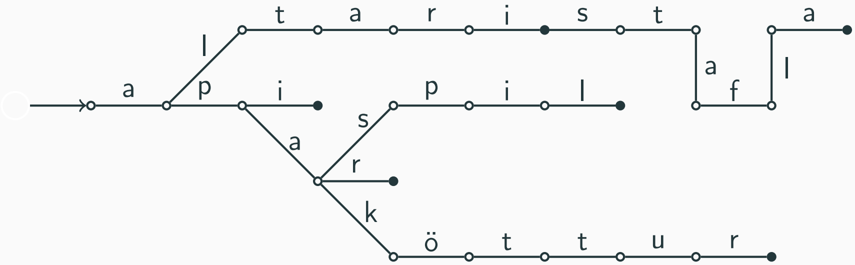


Example



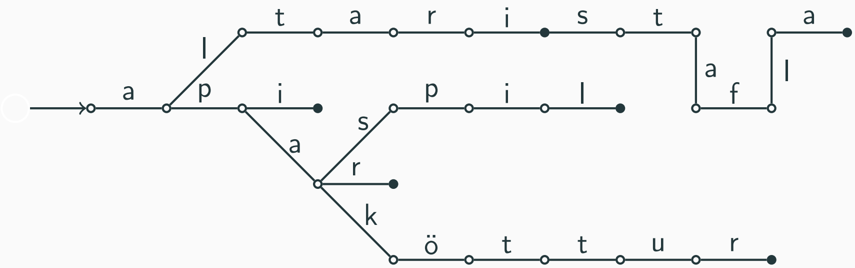
Example

„api“



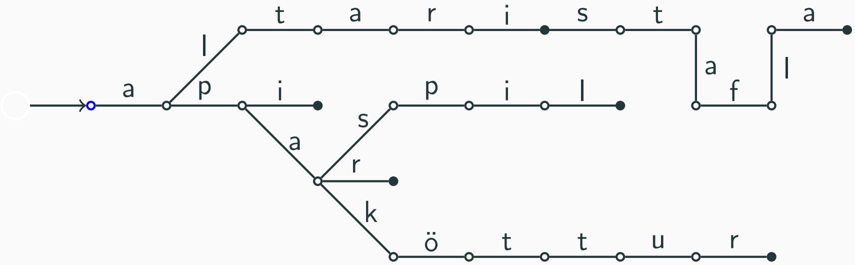
Example

„altarisganga”



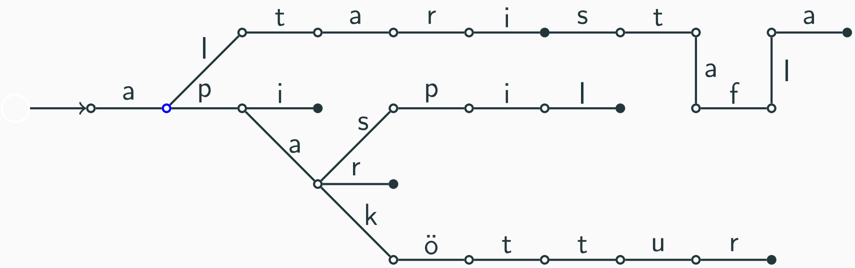
Example

„altarisganga"



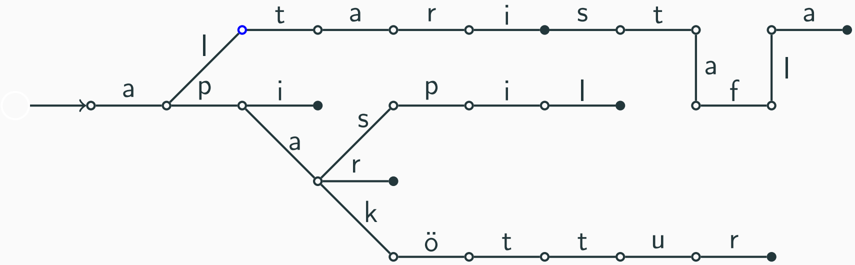
Example

„ltarisganga”



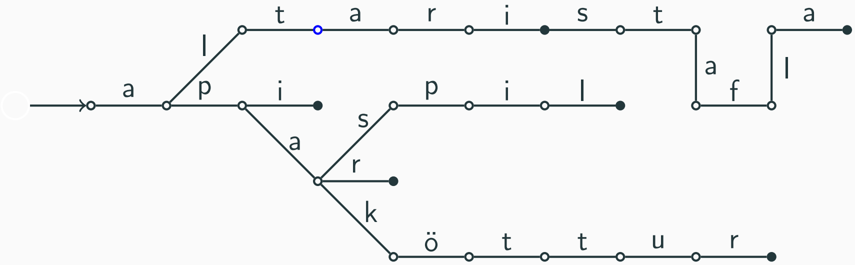
Example

„tarisganga”



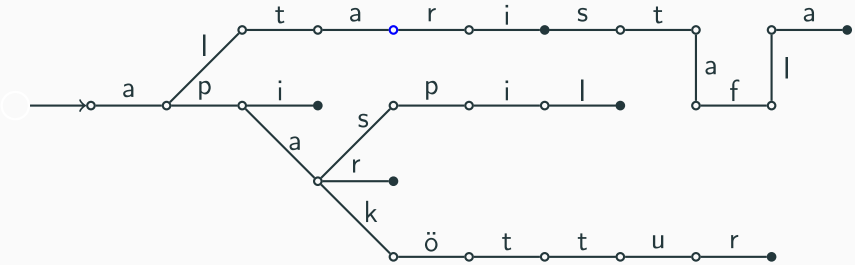
Example

„arisganga"



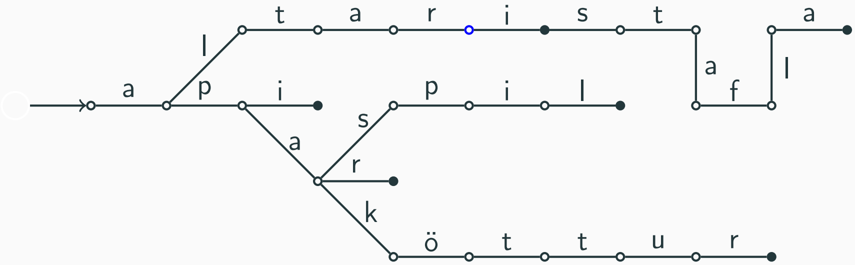
Example

„risganga"



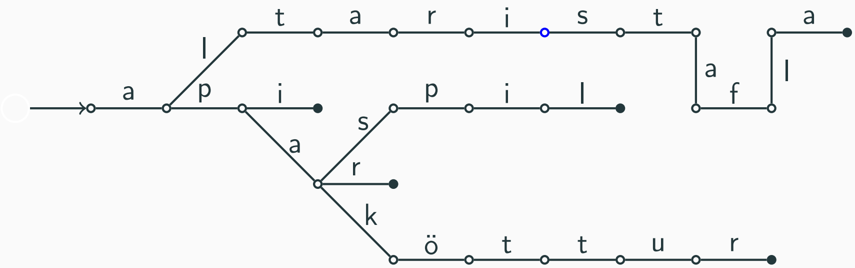
Example

„isganga"



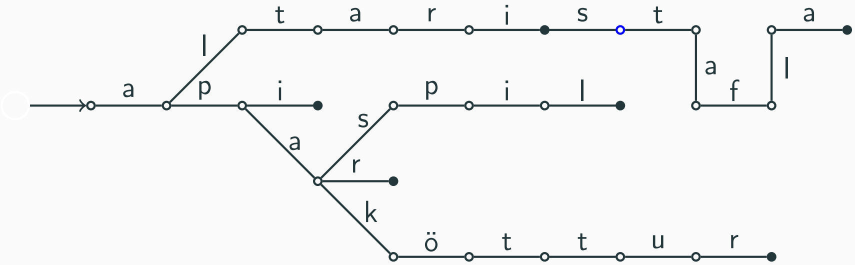
Example

„sganga"



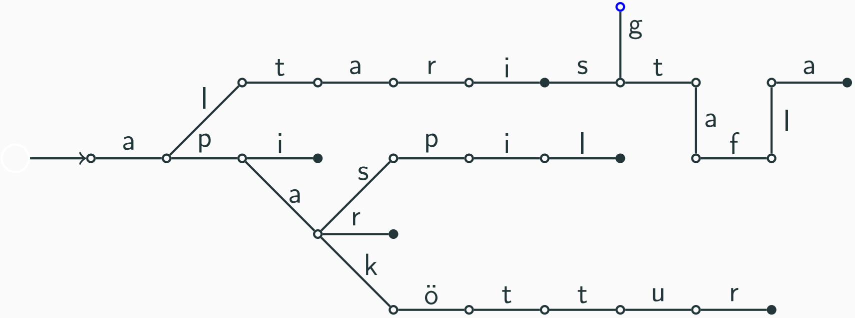
Example

„ganga"

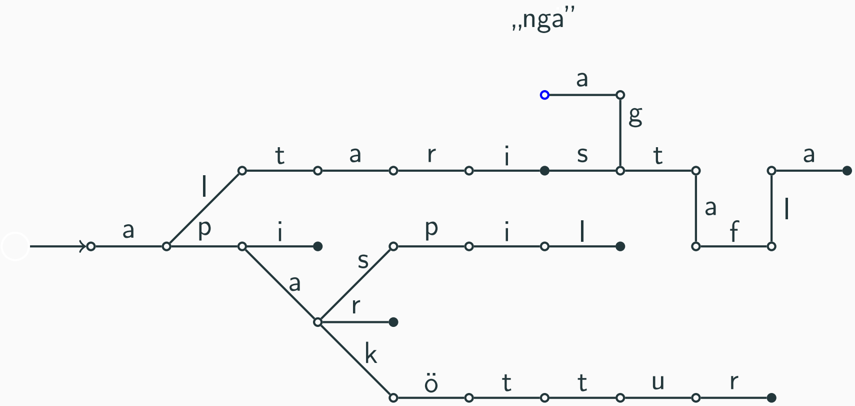


Example

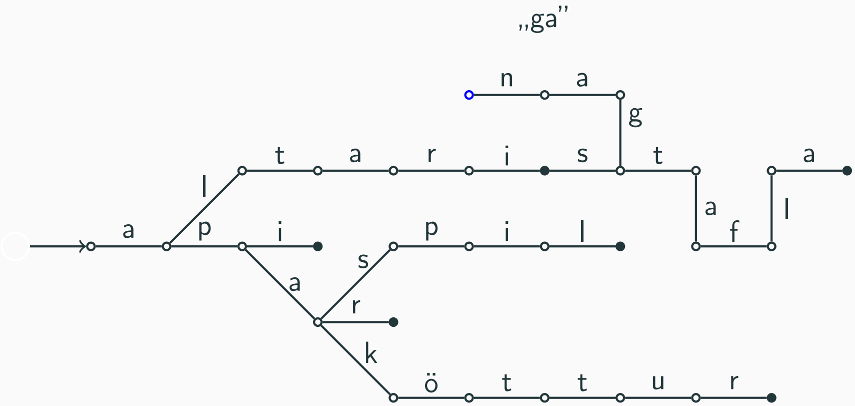
„anga"



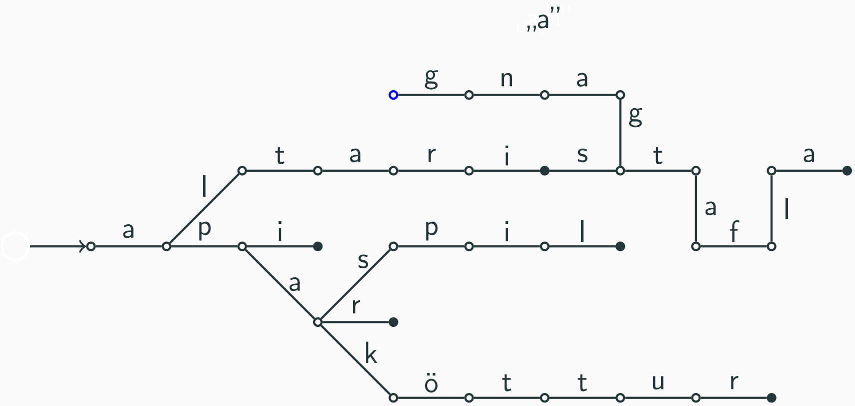
Example



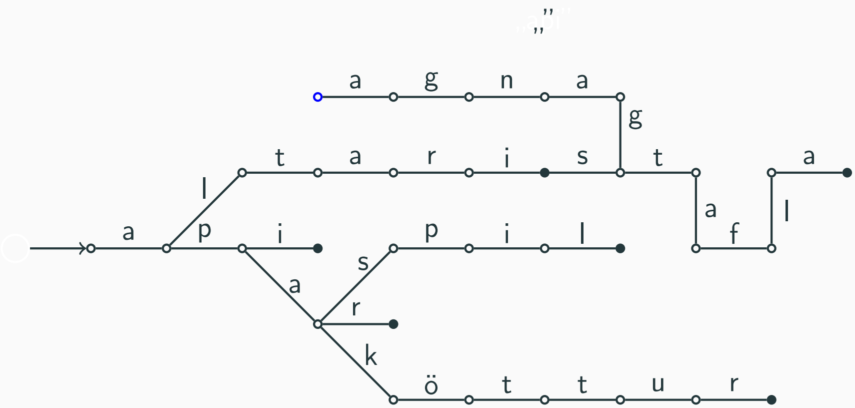
Example



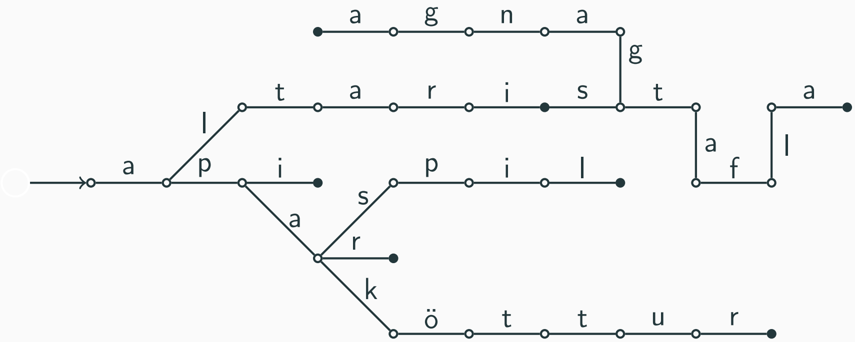
Example



Example



Example



Tries

```cpp
struct node {
    node* children[26];
    bool is_end;

    node() {
        memset(children, 0, sizeof(children));
        is_end = false;
    }
};
```

Tries

```
void insert(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            nd->children[*s - 'a'] = new node();

        insert(nd->children[*s - 'a'], s + 1);
    } else {
        nd->is_end = true;
    }
}
```

Tries

```
bool contains(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            return false;

        return contains(nd->children[*s - 'a'], s + 1);
    } else {
        return nd->is_end;
    }
}
```

Tries

```
node *trie = new node();

insert(trie, "banani");

if (contains(trie, "banani")) {
    // ...
}
```

Tries

- Time complexity?

- Let k be the length of the string we're inserting/looking for
- Lookup is $\mathcal{O}(k)$ and insertion is both $\mathcal{O}(k|\Sigma|)$
- The insertion takes this time because we might have to make k nodes, each needing $|\Sigma|$ pointers initialized

Aho-Corasick

- Let us now have some string s and a list of n strings p, where we denote the j-th string by p_j.
- Let $|s|$ be the length of s and $|p| = |p_1| + \cdots + |p_n|$.
- We want to find all substrings of s that are in the list p.
- We could run KMP n times, once for each p_j, for a time complexity of $\mathcal{O}(n \cdot |s| + |p|)$.
- The Aho-Corasick algorithm improves on this.

The algorithm

- We start by putting all strings in p into a trie T, we want to turn this into a finite state automata.
- We then want to turn T into a finite state automata.
- The nodes of the trie will be our states but the transitions from each state will correspond to a letter from Σ.

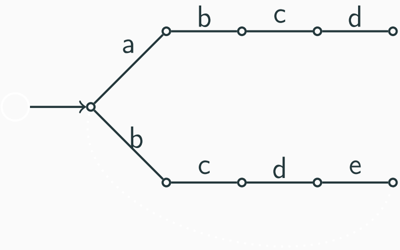
The automata

- Suppose we are in node v in T and want to transition according to the letter c in Σ.
- If there is an node corresponding to adding a c after v we can travel there.
- If not we need to travel back to some node w so the string corresponding to w is a suffix of the one corresponding to v.
- We want to drop the least amount of information, so we want w to be as long as possible.
- We call these transitions *suffix links*. Note that they are essentially independent of c.
- We let the suffix link of the root point back to itself for simplicity's sake.

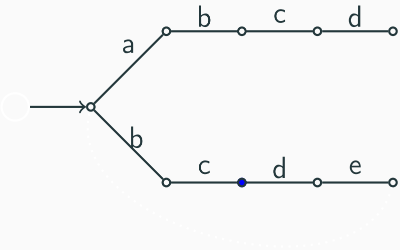
Suffix links

- How do we find the suffix links?
- Let $f(w, c)$ denote the transition from node w with the letter c and let $g(w)$ be the suffix link of w.
- Also let p be the parent of v and $f(p, a) = v$. Then $g(v) = f(g(p), a)$.
- Thus we have a recursive formula we can use.

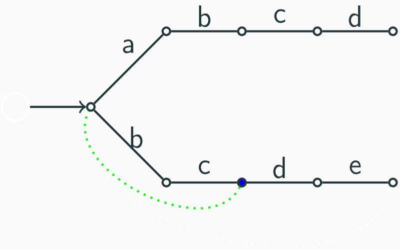
Example



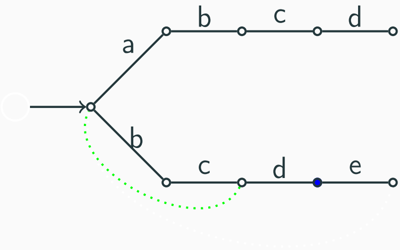
Example



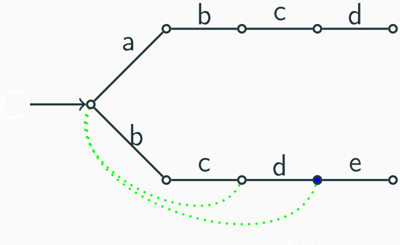
Example



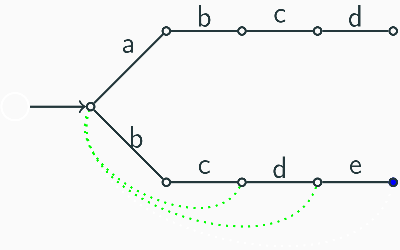
Example



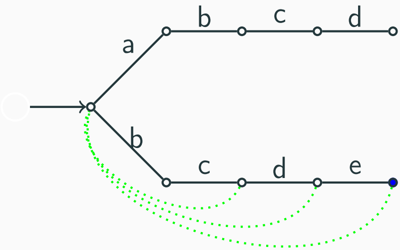
Example



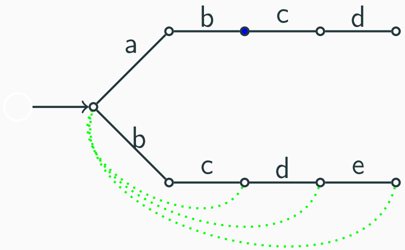
Example



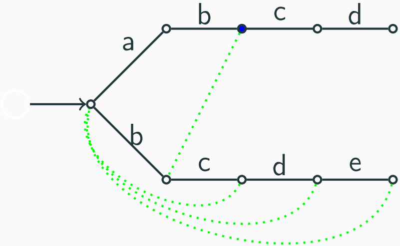
Example



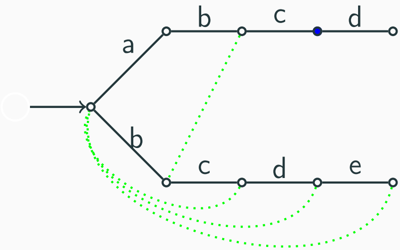
Example



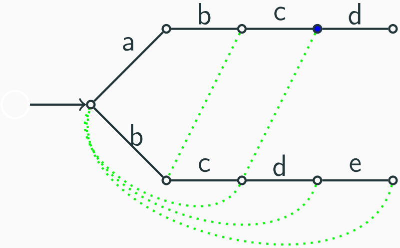
Example



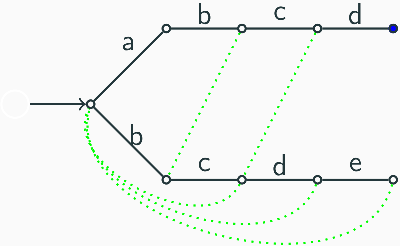
Example



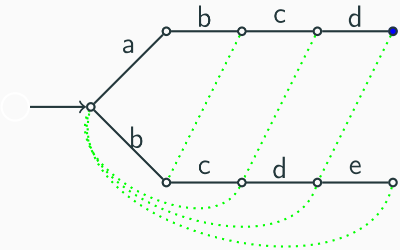
Example



Example



Example

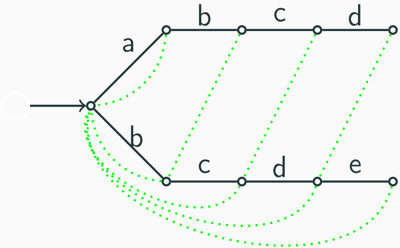


End nodes

- We also have to mark end nodes in T.
- We then walk through s and move around the state machine according to the letters encountered.

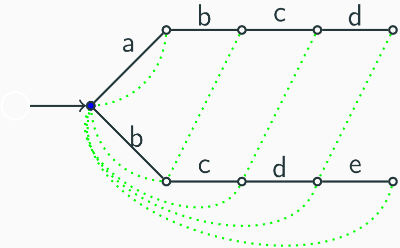
Example

„abcdcdeaaabcdeabcxab"



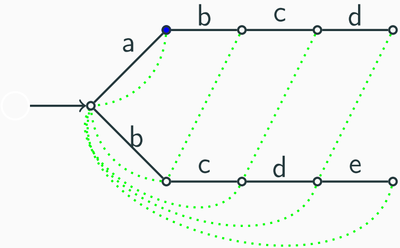
Example

„abcdcdeaaabcdeabcxab"



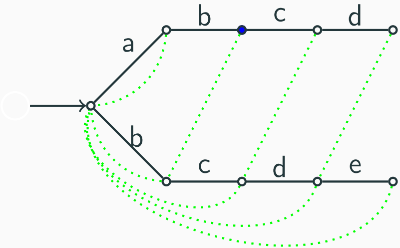
Example

„bcdcdeaaabcdeabcxab"



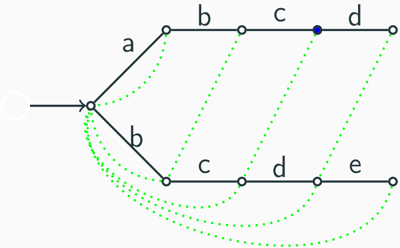
Example

„cdcdeaaabcdeabcxab"



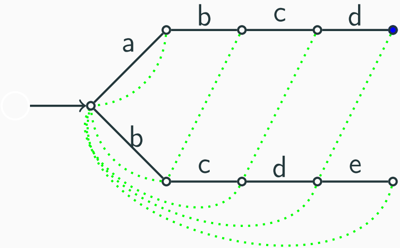
Example

„dcdeaaabcdeabcxab"



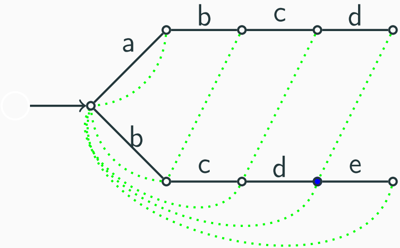
Example

„cdeaaabcdeabcxab"



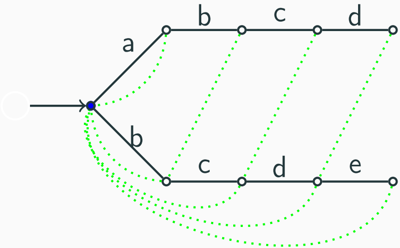
Example

„cdeaaabcdeabcxab"



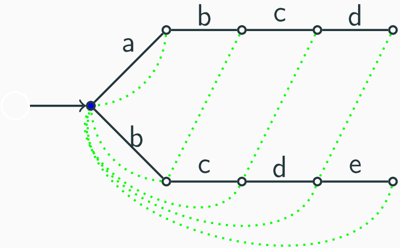
Example

„cdeaaabcdeabcxab"



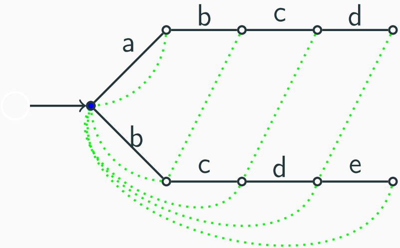
Example

„deaaabcdeabcxab"



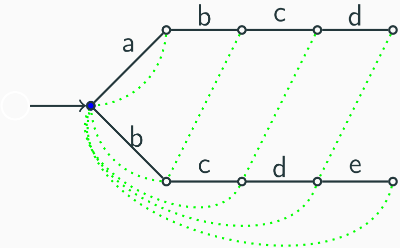
Example

„eaaabcdeabcxab"



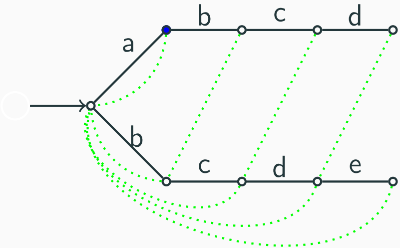
Example

„aaabcdeabcxab"



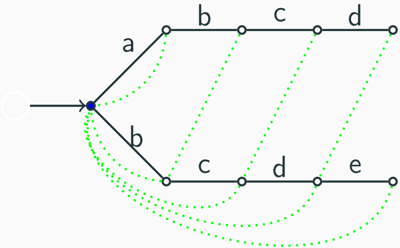
Example

„aabcdeabcxab"



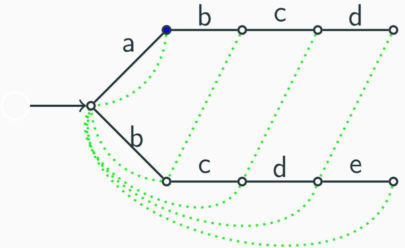
Example

„aabcdeabcxab"



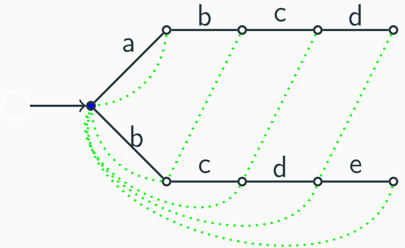
Example

„abcdeabcxab"

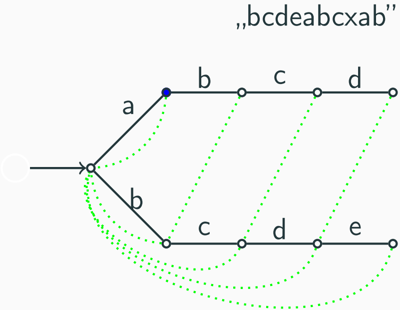


Example

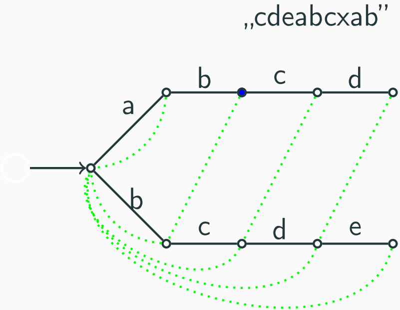
„abcdeabcxab"



Example

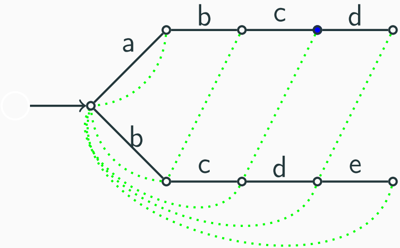


Example



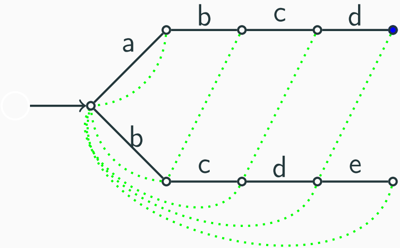
Example

„deabcxab"



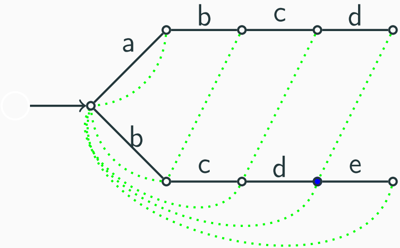
Example

„eabcxab"



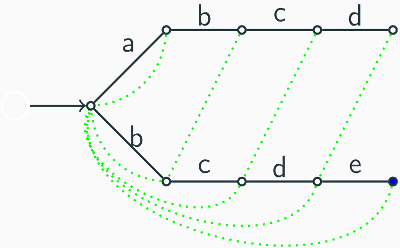
Example

„eabcxab"



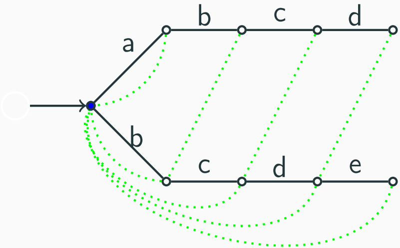
Example

„abcxab"



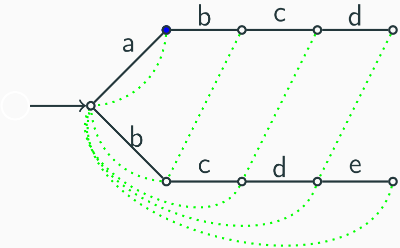
Example

„abcxab"



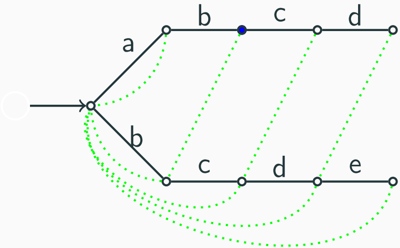
Example

„bcxab"



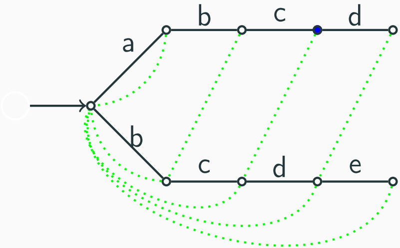
Example

„cxab"



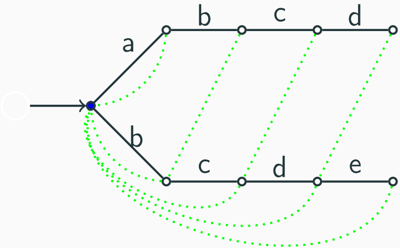
Example

„xab"



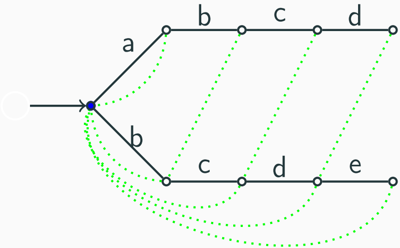
Example

„xab"



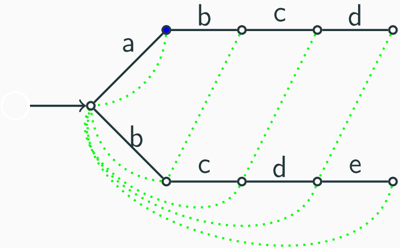
Example

„ab"

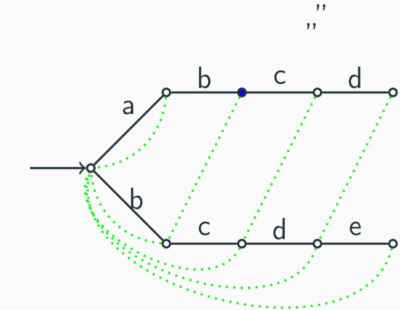


Example

„b"

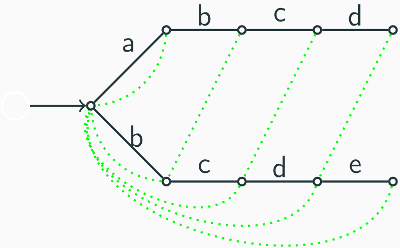


Example



Example

,,"
,,"



End nodes

End nodes

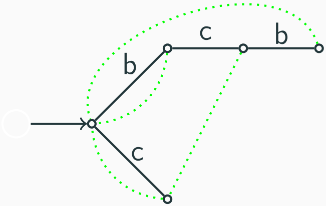
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?

End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.

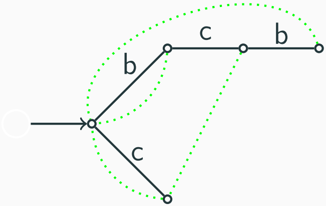
End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



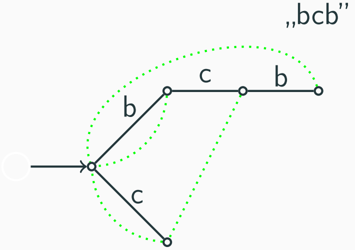
End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



End nodes

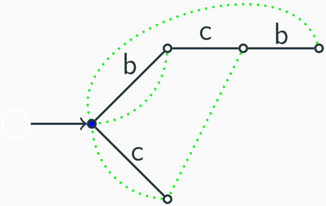
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



End nodes

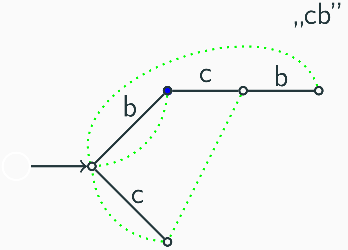
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.

„bcb"



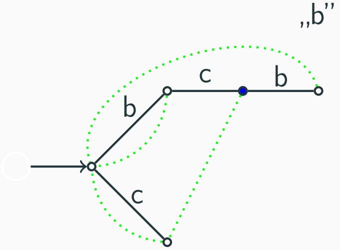
End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



End nodes

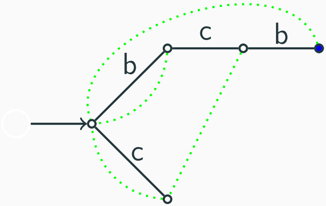
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



End nodes

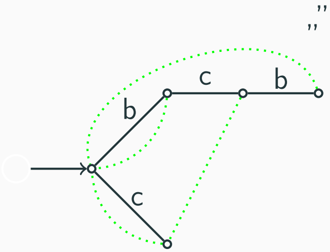
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.

,,"



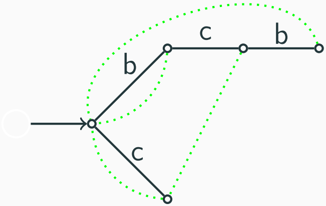
End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



End nodes

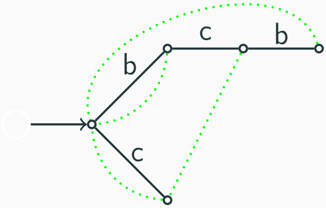
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



- To keep the complexity in check we again use dynamic programming.

End nodes

- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



- To keep the complexity in check we again use dynamic programming.
- We add the concatenated links into the tree, calling them *exit links*.

Speed

- Let us assume the strings in p appear k times in s.
- Then the time complexity is $\mathcal{O}(|s| + |\Sigma| \cdot |p| + k)$
- If we only want the number of matches, the implementation can be modified accordingly and then the complexity is $\mathcal{O}(|s| + |\Sigma| \cdot |p|)$.
- Note that for a bounded alphabet, this second complexity is linear.

Implementation explanation

- The implementation contains three helper functions.
- The first is `trie_step(...)` which is used to move around the state machine.
- The second is `trie_suffix(...)` which is used to find suffix links.
- The third is `trie_exit(...)` which is used to find exit links.
- All these functions are recursive and memoized.

Aho nodes

```
#define ALPHABET 128
// Helper function to get index of letter
int val(char c) { return c; }
struct listnode {
    // n is index of next node, v is value of this node
    int v, n;
    listnode(int _v, int _n) : v(_v), n(_n) { }
};
struct trienode {
    // l is the index of the pattern that ends here or -1 if none
    // e is the exit link index, d is the suffix link index
    // p is the parent index
    // c is the character of the incoming edge
    // t is the transition table of the trie node
    int t[ALPHABET], l, e, p, c, d;
    trienode(int _p, int _c) :
        l(-1), e(-1), p(_p), c(_c), d(-1) {
        memset(t, -1, sizeof(t));
    }
};
```

Aho trie

```
struct trie {
    // r is the index of the root
    int r;
    vector<trienode> m;
    vector<listnode> w;

    trie() {
        m = vector<trienode>();
        w = vector<listnode>();
        r = trie_node(-1, -1);
    }

    int list_node(int v, int n) {
        w.push_back(listnode(v, n));
        return w.size() - 1;
    }
    int trie_node(int p, int c) {
        m.push_back(trienode(p, c));
        return m.size() - 1;
    }

    void trie_insert(string &s, int x) {
        int h, i = 0;
        for(h = r; i < s.size(); h = m[h].t[val(s[i])], i++)
            if(m[h].t[val(s[i])] == -1)
                m[h].t[val(s[i])] = trie_node(h, val(s[i]));
        m[h].l = list_node(x, m[h].l);
    }

    int trie_suffix(int h) {
        if(m[h].d != -1) return m[h].d;
        if(h == r || m[h].p == r) return m[h].d = r;
        return m[h].d =
            trie_step(trie_suffix(m[h].p), m[h].c);
    }

    int trie_step(int h, int c) {
        if(m[h].t[c] != -1) return m[h].t[c];
        return m[h].t[c] = h == r ? r :
            trie_step(trie_suffix(h), c);
    }

    int trie_exit(int h) {
        if(m[h].e != -1) return m[h].e;
        if(h == 0 || m[h].l != -1) return m[h].e = h;
        return m[h].e = trie_exit(trie_suffix(h));
    }
};
```

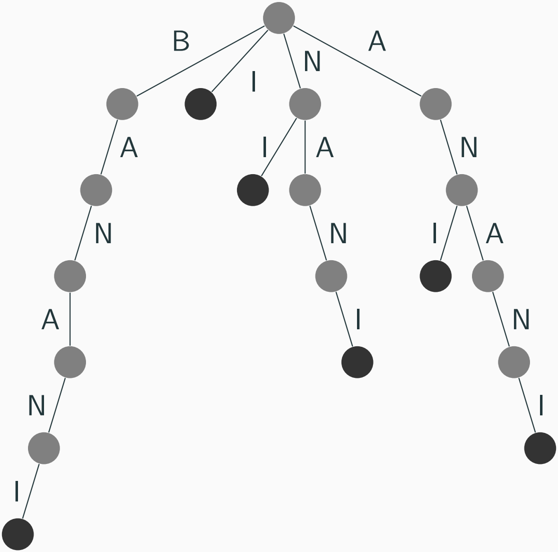
Aho implementation

```
int aho_corasick(string &s, vector<string> &p) {
    trie t; int h, i, j, k, w, m = p.size(), l[m];
    for(i = 0; i < m; i++) l[i] = p[i].size();
    for(i = 0; i < m; i++) t.trie_insert(p[i], i);
    s.push_back('\0');
    for(i = 0, j = 0, h = t.r; j < s.size(); j++) {
        k = t.trie_exit(h);
        while(t.m[k].l != -1) {
            for(w = t.m[k].l; w != -1; w = t.w[w].n) {
                cout << p[t.w[w].v] << " found at index " <<
                    j - l[t.w[w].v] << '\n';
            }
            k = t.trie_exit(t.trie_suffix(k));
        }
        h = t.trie_step(h, val(s[j]));
    }
    return i;
}
```

Suffix tries

- Say we're dealing with some string S of length n
- Let's insert all suffixes of S into a trie
- $S = \texttt{banani}$
  - `insert(trie, "banani");`
  - `insert(trie, "anani");`
  - `insert(trie, "nani");`
  - `insert(trie, "ani");`
  - `insert(trie, "ni");`
  - `insert(trie, "i");`

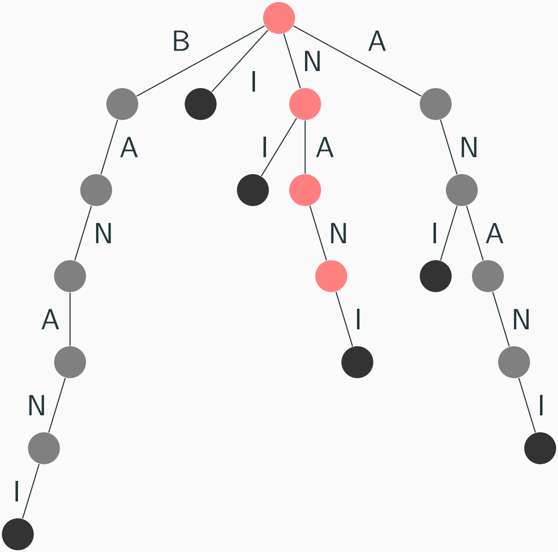
Suffix tries



Suffix tries

- There are a lot of cool things we can do with suffix tries
- Example: String matching
- If a string T is a substring in S, then (obviously) it has to start at some suffix of S
- So we can simply look for T in the suffix trie of S, ignoring whether the last node is an end node or not
- This is just $O(m)$...

Suffix tries



Suffix tries

- String matching is fast if we have the suffix trie for S

- But what is the time complexity of suffix trie construction?
- There are n suffixes, and it takes $O(n)$ to insert each of them
- So $O(n^2)$, which is pretty slow

- Can we do better?
- There can be up to n^2 nodes in the graph, so this is actually optimal...

Suffix trees

- There exists a compressed version of a suffix trie, called a suffix tree

- It can be constructed in $O(n)$, and has all the features that suffix tries have
- But the $O(n)$ construction algorithm is pretty complex, a big disadvantage for us

Suffix arrays

- A variation of the previous structures
- Can do everything the other structures can do, with a small overhead
- Can be constructed pretty quickly with relatively simple code

Suffix arrays

- Take all the suffixes of S

```
banani
anani
nani
ani
ni
i
```

- and sort them

```
anani
ani
banani
i
nani
ni
```

Suffix arrays

- We can use this array to do everything that suffix tries can do
- Like string matching

Suffix arrays

- Let's look for `nan`

```
anani
ani
banani
i
nani
ni
```

Suffix arrays

- Let's look for `nan`
- The first letter in the string has to be `n`, so we can binary search for the range of strings starting with `n`

```
anani
ani
banani
i
nani
ni
```

- Let's look for `nan`
- The first letter in the string has to be `n`, so we can binary search for the range of strings starting with `n`

```
nani
ni
```

Suffix arrays

- Let's look for `nan`
- The second letter in the string has to be `a`, so we can binary search for the range of strings that have `a` as the second letter

```
nani
ni
```

Suffix arrays

- Let's look for `nan`
- The second letter in the string has to be `a`, so we can binary search for the range of strings that have `a` as the second letter

```
nani
```

Suffix arrays

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

```
nani
```

Suffix arrays

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

`nani`

Suffix arrays

- Let's look for `nan`
- The third letter in the string has to be `n`, so we can binary search for the range of strings that have `n` as the third letter

```
nani
```

- If there is at least one string left, we have a match

Suffix arrays

- Time complexity?
- For each letter in T, we do two binary searches on the n suffixes to find the new range
- Time complexity is $O(m \times \log n)$
- A bit slower than doing it with a suffix trie, but still not bad

Suffix arrays

- But how do we construct a suffix array for a string?

- A simple `sort(suffixes)` is $O(n^2 \log(n))$, because comparing two suffixes is $O(n)$
- And we still have the same problem as with suffix tries, there are almost n^2 characters if we store all suffixes

Suffix arrays

- The second problem is easy to fix
- Just store the indices of the suffixes

```
anani
ani
banani
i
nani
ni
```

- becomes

```
1: anani
3: ani
0: banani
5: i
2: nani
4: ni
```

Suffix arrays

- What about the construction?
- In short, we
  - sort all suffixes by only looking at the first letter
  - sort all suffixes by only looking at the first 2 letters
  - sort all suffixes by only looking at the first 4 letters
  - sort all suffixes by only looking at the first 8 letters
  - ...
  - sort all suffixes by only looking at the first 2^i letters
  - ...
- If we use an $O(n \log n)$ sorting algorithm, this is $O(n \log^2 n)$
- We can also use an $O(n)$ sorting algorithm, since all sorted values are between 0 and n, bringing it down to $O(n \log n)$

Suffix arrays

```
struct suffix_array {
    struct entry {
        pair<int, int> nr;
        int p;

        bool operator <(const entry &other) {
            return nr < other.nr;
        }
    };


    string s;
    int n;
    vector<vector<int> > P;
    vector<entry> L;
    vi idx;

    // constructor
};
```

Suffix arrays

```
suffix_array(string _s) : s(_s), n(s.size()) {
    L = vector<entry>(n);
    P.push_back(vi(n));
    idx = vi(n);

    for (int i = 0; i < n; i++) {
        P[0][i] = s[i];
    }

    for (int stp = 1, cnt = 1; (cnt >> 1) < n; stp++, cnt <<= 1) {
        P.push_back(vi(n));
        for (int i = 0; i < n; i++) {
            L[i].p = i;
            L[i].nr = make_pair(P[stp - 1][i], i + cnt < n ? P[stp - 1][i + cnt] : -1);
        }

        sort(L.begin(), L.end());
        for (int i = 0; i < n; i++) {
            if (i > 0 && L[i].nr == L[i - 1].nr) {
                P[stp][L[i].p] = P[stp][L[i - 1].p];
            } else {
                P[stp][L[i].p] = i;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        idx[P[P.size() - 1][i]] = i;
    }
}
```

Suffix arrays

- There is also one other useful operation on suffix arrays
- Finding the longest common prefix (lcp) of two suffixes of S

```
1: anani
3: ani
0: banani
5: i
2: nani
4: ni
```

- `lcp(1,3) = 2`
- `lcp(2,1) = 0`

- This function can be implemented in $O(\log n)$ by using intermediate results from the suffix array construction

Suffix arrays

```
int lcp(int x, int y) {
    int res = 0;
    if (x == y) return n - x;
    for (int k = P.size() - 1; k >= 0 && x < n && y < n; k--) {
        if (P[k][x] == P[k][y]) {
            x += 1 << k;
            y += 1 << k;
            res += 1 << k;
        }
    }
    return res;
}
```

Longest common substring

- Given two strings S and T, find their longest common substring
- `S = banani`
- `T = kanina`
- Their longest common substring is `ani`
- *see example*