Strings & Geometry

Atli FF

November 4, 2024

School of Computer Science
Reykjavík University

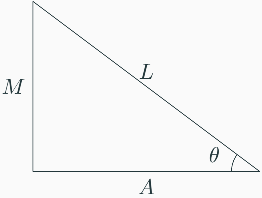
Today we're going to cover

- Trigonometry
- Geometry
- Computational geometry
- String matching (naive, KMP)
- Tries
- Aho-Corasick

Trigonometry

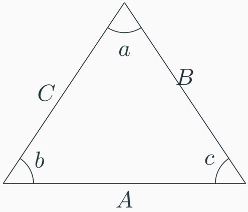
Trigonometry

- Before we even dive into the geometry and how to do it on a computer, let's jog your memories.
- You should all hopefully be familiar with the trigonometric functions.
- We consider a triangle to be right-angled if it has a corner that's 90°.
- For such triangles we have:
  - $\frac{A}{L} = \cos\theta$.
  - $\frac{M}{L} = \sin\theta$.
  - $\frac{M}{A} = \frac{M}{L}\frac{L}{A} = \frac{\sin\theta}{\cos\theta} = \tan\theta$.
- We also have the pythagorean theorem $L^2 = A^2 + M^2$.



More trig

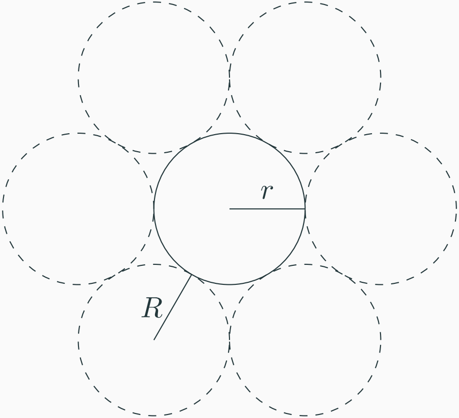
- More generally we have:
  - $\frac{\sin a}{A} = \frac{\sin b}{B} = \frac{\sin c}{C}$ (sine law).
  - $A^2 = B^2 + C^2 - 2BC \cos a$ (cosine law)
- **Exercise:** Prove the pythagorean theorem using the cosine law.



Example: NN and the Optical Illusion

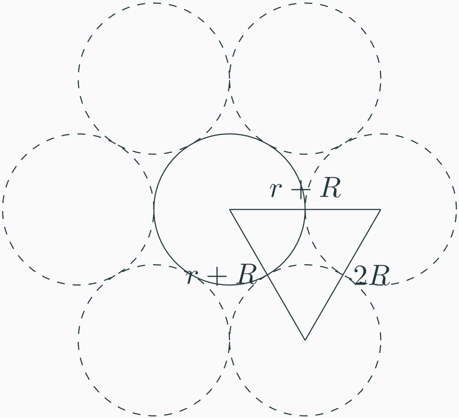
- You are given an integer n and a real number r.
- You then draw a circle of radius r.
- You then want to draw n circles of the same size tangent to the outside of this circle and such that they are tangent to their neighbours.
- What radius will the outer circles have?

$N = 6$ image



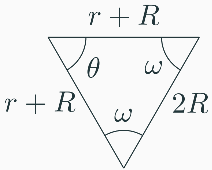
Towards a solution

We see that the distance from the center of the circle in the middle to the center of an outer circle is $r + R$. We thus get an isosceles triangle.



Closer and closer

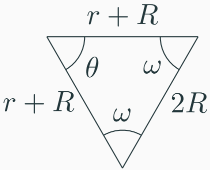
- Now we have $\theta = \dfrac{360^\circ}{n}$ and $\omega = \dfrac{180^\circ - \theta}{2}$.



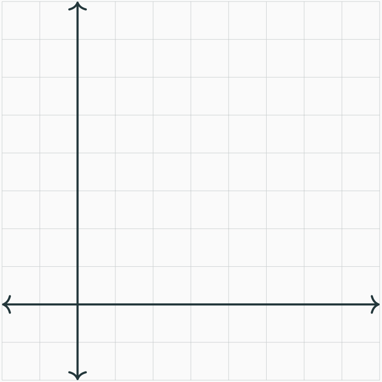
Solution

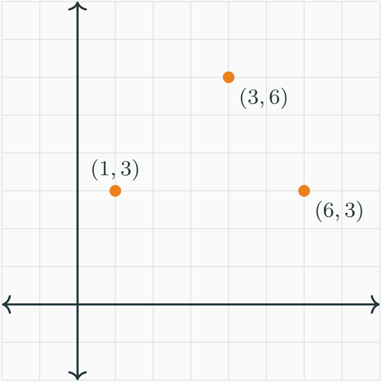
- Finally the law of sines gives us

$$\frac{2R}{\sin\theta} = \frac{r+R}{\sin\omega} \Rightarrow 2R\sin\omega = r\sin\theta + R\sin\theta$$
$$\Rightarrow 2R\sin\omega - R\sin\theta = r\sin\theta$$
$$\Rightarrow R = \frac{r\sin\theta}{2\sin\omega - \sin\theta}.$$

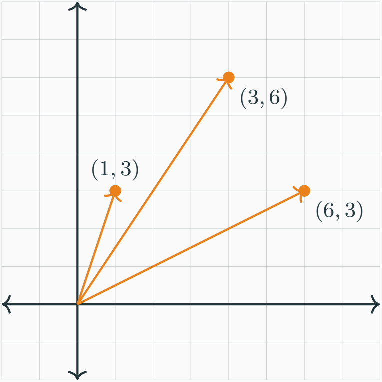


Computer representation

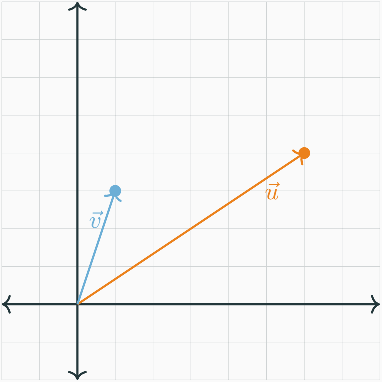


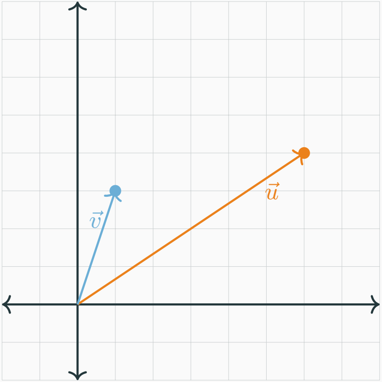


- Points are represented by a pair of numbers, (x, y).



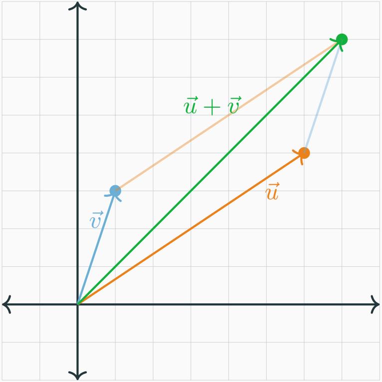
- Points are represented by a pair of numbers, (x, y).
- Vectors are represented in the same way.
- Thinking of points as vectors allows us to do many things.


$\vec{v}$
$\vec{u}$



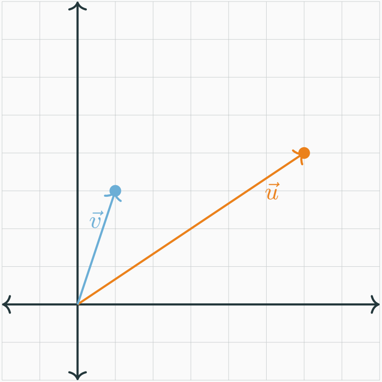
- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$



- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

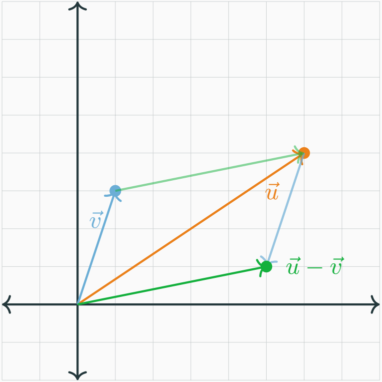


- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

- Subtraction is defined in the same manner

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 - x_1 \\ y_0 - y_1 \end{pmatrix}$$



- Simplest operation, addition is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 + x_1 \\ y_0 + y_1 \end{pmatrix}$$

- Subtraction is defined in the same manner

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 - x_1 \\ y_0 - y_1 \end{pmatrix}$$

```cpp
struct point {
    double x, y;
    point(double _x, double _y) {
        x = _x, y = _y;
    }

    point operator+(const point &oth){
        return point(x + oth.x, y + oth.y);
    }

    point operator-(const point &oth){
        return point(x - oth.x, y - oth.y);
    }
};
```

. . . or we could use the `complex<double>` class.

```
using point = complex<double>;
```

…or we could use the `complex<double>` class.

```
using point = complex<double>;
```

The `complex` class in `C++` and `Java` has methods defined for

- Addition
- Subtraction
- Multiplication by a scalar
- Length
- Trigonometric functions
- And much more!

Complex numbers

- We define $\mathbb{C} := \mathbb{R} \times \mathbb{R}$.
- Then we define addition on $\mathbb{C}$ such that for $(a, b), (c, d) \in \mathbb{C}$ we get
$$(a, b) + (c, d) = (a + c, b + d).$$
- We also define multiplication on $\mathbb{C}$ such that for $(a, b), (c, d) \in \mathbb{C}$ we get
$$(a, b) \cdot (c, d) = (ac - bd, ad + bc).$$
- We usually denote $(0, 1) \in \mathbb{C}$ by i and $(x, y) \in \mathbb{C}$ by $x + yi$.
- Note that $(x, y) = (x, 0) + i \cdot (y, 0)$ here.
- We call these numbers in $\mathbb{C}$ *complex numbers*.

Complex numbers ctd.

- If $z = x + yi \in \mathbb{C}$ then
  - We call x the *real part* of z and y the *imaginary part* of z.
  - We define the *magnitude* of z by $|z| = \sqrt{x^2 + y^2}$.
  - We call $x - yi$ the *conjugate* of z, denoted by $\overline{z}$.
  - We call the angle (x, y) makes with the positive x-axis the *argument* of z and denote it by $\mathrm{Arg}(z)$.

Operations

- Let $w, z \in \mathbb{C}$.
- Then $w + z$ will be z translated by w, as if we were adding vectors.
- If $|w| = 1$ then $z \cdot w$ will be z rotated around 0 by $\mathrm{Arg}(w)$ radians.
- If $|z| = r$ and $\mathrm{Arg}(z) = \theta$ we can write $z = re^{i\theta}$.
- If $z = r_1 e^{i\theta_1}$ and $w = r_2 e^{i\theta_2}$ then $z \cdot w = r_1 r_2 e^{i(\theta_1 + \theta_2)}$.

Using complex in C++

- Usually we do `using point = complex<double>;`
- Then we can initialize a point with `point z(x, y)`
    - `real(z)` returns the x-coordinate
    - `imag(z)` returns the y-coordinate
    - `abs(z)` returns the magnitude $|z|$
    - `abs(z - w)` returns the distance from z to w
    - `arg(z)` returns the argument of z
    - `conj(z)` returns the conjugate $\overline{z}$

Example

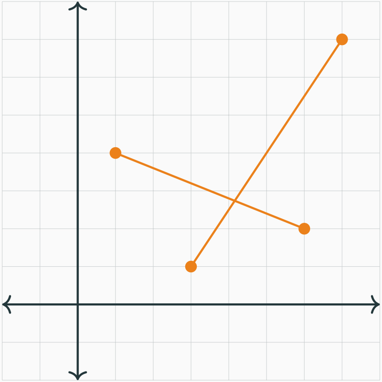
- Let us consider a problem.
- You start at $(0, 0)$ and get a sequence of commands.
- All the commands consist of a single letter and a number. The commands are:
    - ...`f x` you move forward x meters..
    - ...`b x` you move backwards x meters.
    - ...`r x` you rotate x radians to the right.
    - ...`l x` you rotate x radians to the left.
- How far from $(0, 0)$ do you end up after following the commands?

Solution

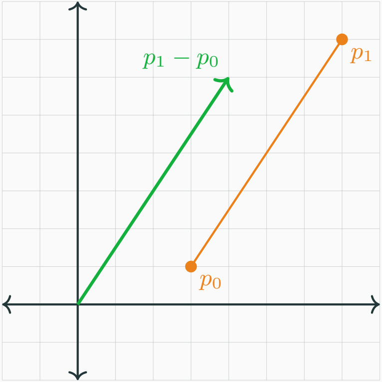
- If we are stood at $p \in \mathbb{C}$ and want to take a step of r meters in the direction θ we simply add $re^{i\theta}$ to p.
- What direction we are facing at the start makes no difference since it gives the same distance at the end.

Code

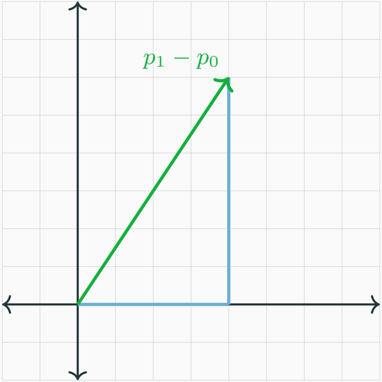
```cpp
#include <bits/stdc++.h>
using namespace std;
using point = complex<double>;
int main() {
    int n; cin >> n;
    double x, r = 0.0;
    point p(0, 0);
    while (n--) {
        char c; cin >> c >> x;
        if (c == 'f')      p += x*exp(1i*r);
        else if (c == 'b') p -= x*exp(1i*r);
        else if (c == 'l') r += x;
        else if (c == 'r') r -= x;
        else assert(0);
        }
    cout << setprecision(15) << abs(p) << endl;
}
```



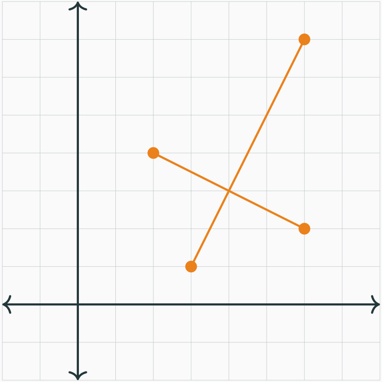
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



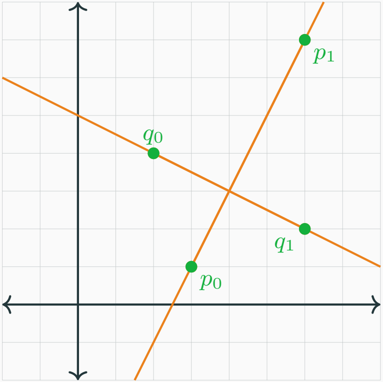
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



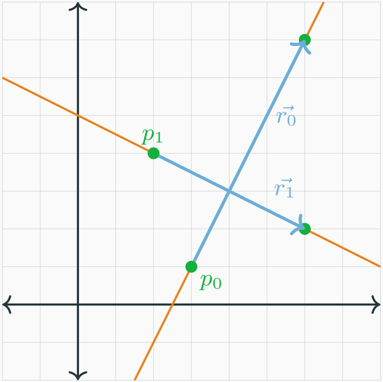
- Line segments are represented by a pair of points, $((x_0, y_0), (x_1, y_1))$.



- Line representation same as line segments.

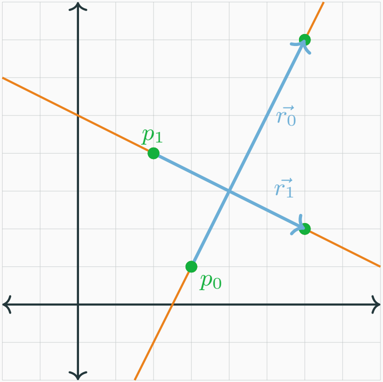


- Line representation same as line segments.
- Treat them as lines passing through the two points.



- Line representation same as line segments.
- Treat them as lines passing through the two points.
- Or as a point and a direction vector.

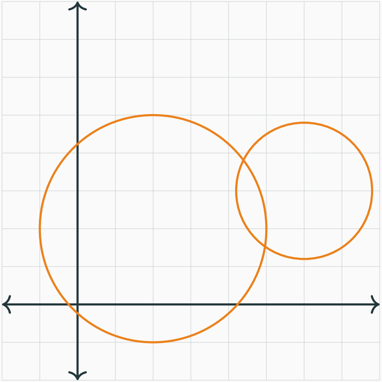
$$p + t \cdot \vec{r}$$



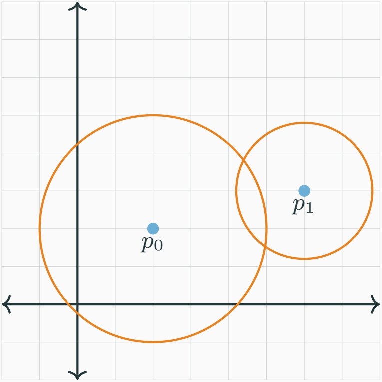
- Line representation same as line segments.
- Treat them as lines passing through the two points.
- Or as a point and a direction vector.

$$p + t \cdot \vec{r}$$

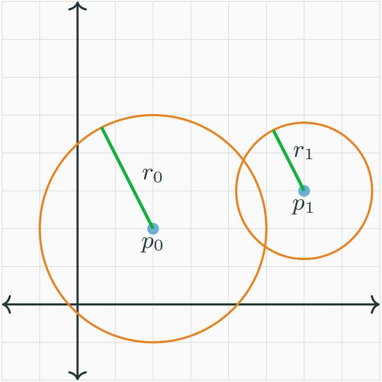
- Either way `pair<point,point>`



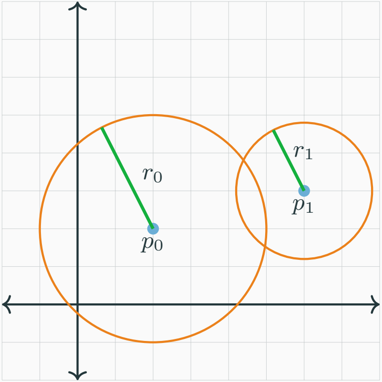
- Circles are very easy to represent.



- Circles are very easy to represent.
- Center point $p = (x, y)$.



- Circles are very easy to represent.
- Center point $p = (x, y)$.
- And the radius r.



- Circles are very easy to represent.
- Center point $p = (x, y)$.
- And the radius r.

```
pair<point,double>
```

Computational geometry

Given two vectors

$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the dot product of $\vec{u}$ and $\vec{v}$ is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = x_0 \cdot x_1 + y_0 \cdot y_1$$

Given two vectors

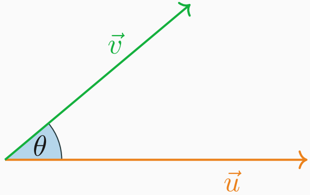
$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the dot product of $\vec{u}$ and $\vec{v}$ is defined as

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = x_0 \cdot x_1 + y_0 \cdot y_1$$

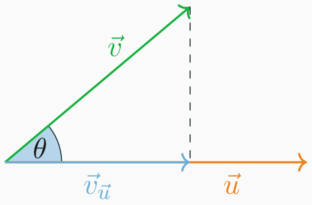
Which in geometric terms is

$$\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}| \cos\theta$$



- Allows us to calculate the angle between $\vec{u}$ and $\vec{v}$.

$$\theta = \arccos\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}\right)$$

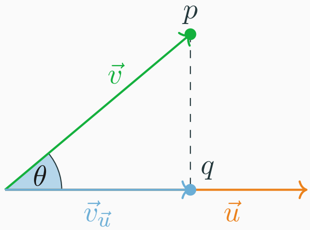


- Allows us to calculate the angle between $\vec{u}$ and $\vec{v}$.

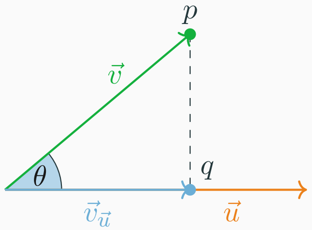
$$\theta = \arccos\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|}\right)$$

- And the projection of $\vec{v}$ onto $\vec{u}$.

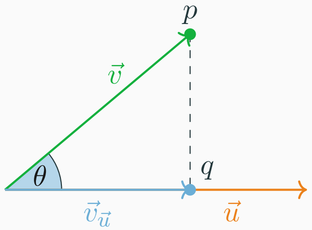
$$\vec{v}_{\vec{u}} = \left(\frac{\vec{u}\cdot\vec{v}}{|u|^2}\right)\vec{u}$$



- The closest point on $\vec{u}$ to p is q.



- The closest point on $\vec{u}$ to p is q.
- The distance from p to $\vec{u}$ is the distance from p to q.



- The closest point on $\vec{u}$ to p is q.
- The distance from p to $\vec{u}$ is the distance from p to q.
- Unless q is outside $\vec{u}$, then the closest point is either of the endpoints.

Rest of the code will use the `complex` class.

```
#define P(p) const point &p
#define L(p0, p1) P(p0), P(p1)
double dot(P(a), P(b)) {
    return real(a) * real(b) + imag(a) * imag(b);
}
double angle(P(a), P(b), P(c)) {
    return acos(dot(b - a, c - b) / abs(b - a) / abs(c - b));
}
point closest_point(L(a, b), P(c), bool segment = false) {
    if (segment) {
        if (dot(b - a, c - b) > 0) return b;
        if (dot(a - b, c - a) > 0) return a;
    }
    double t = dot(c - a, b - a) / norm(b - a);
    return a + t * (b - a);
}
```

Given two vectors

$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the cross product of $\vec{u}$ and $\vec{v}$ is defined as

$$\left| \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \times \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right| = x_0 \cdot y_1 - y_0 \cdot x_1$$

Given two vectors

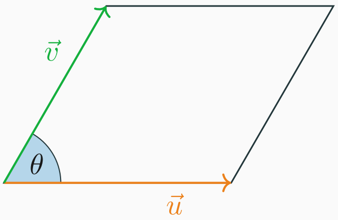
$$\vec{u} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \qquad \vec{v} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

the cross product of $\vec{u}$ and $\vec{v}$ is defined as

$$\left| \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \times \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right| = x_0 \cdot y_1 - y_0 \cdot x_1$$

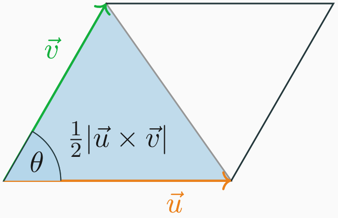
Which in geometric terms is

$$|\vec{u} \times \vec{v}| = |\vec{u}||\vec{v}| \sin \theta$$



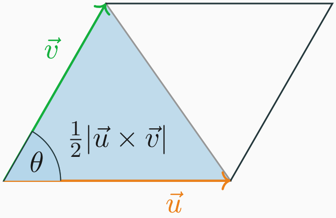
- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$



- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$

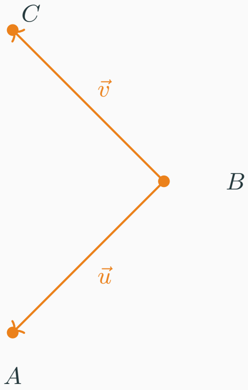


- Allows us to calculate the area of the triangle formed by $\vec{u}$ and $\vec{v}$.

$$\frac{|\vec{u} \times \vec{v}|}{2}$$

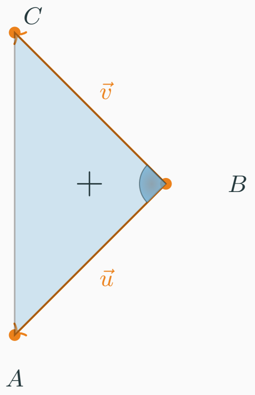
- And can tell us if the angle between $\vec{u}$ and $\vec{v}$ is positive or negative.

$$\begin{aligned} |\vec{u} \times \vec{v}| < 0 &\quad \text{iff} \quad \theta < \pi \\ |\vec{u} \times \vec{v}| = 0 &\quad \text{iff} \quad \theta = \pi \\ |\vec{u} \times \vec{v}| > 0 &\quad \text{iff} \quad \theta > \pi \end{aligned}$$



- Given three points A, B and C, we want to know if they form a counter-clockwise angle in that order.

$$A \to B \to C$$

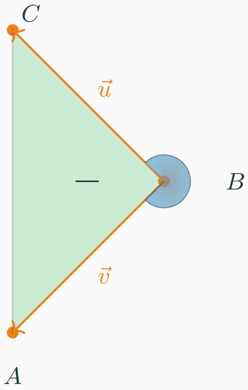


- Given three points A, B and C, we want to know if they form a counter-clockwise angle in that order.

$$A \to B \to C$$

- We can examine the cross product of and the area of the triangle formed by

$$\vec{u} = B - C \quad \vec{v} = B - A$$
$$\vec{u} \times \vec{v} > 0$$



- The points in the reverse order do not form a counter clockwise angle.

$$C \to B \to A$$

- In the reverse order the vectors swap places

$$\vec{u} = B - A \quad \vec{v} = B - C$$
$$\vec{u} \times \vec{v} < 0$$

- The points in the reverse order do not form a counter clockwise angle.

$$C \to B \to A$$

- In the reverse order the vectors swap places

$$\vec{u} = B - A \quad \vec{v} = B - C$$
$$\vec{u} \times \vec{v} < 0$$

- If the points A, B and C are on the same line, then the area will be 0.

```
double cross(P(a), P(b)) {
    return real(a)*imag(b) - imag(a)*real(b);
}
double ccw(P(a), P(b), P(c)) {
    return cross(b - a, c - b);
}
bool collinear(P(a), P(b), P(c)) {
    return abs(ccw(a, b, c)) < EPS;
}
```

Very common task is to find the intersection of two lines or line segments.

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.
- We can do so by setting

$$A = y_1 - y_0$$

$$B = x_0 - x_1$$

$$C = A \cdot x_0 + B \cdot y_1$$

Very common task is to find the intersection of two lines or line segments.

- Given a pair of points (x_0, y_0), (x_1, y_1), representing a line we want to start by obtaining the form $Ax + By = C$.
- We can do so by setting

$$A = y_1 - y_0$$
$$B = x_0 - x_1$$
$$C = A \cdot x_0 + B \cdot y_1$$

- If we have two lines given by such equations, we simply need to solve for the two unknowns, x and y.

For two lines

$$A_0 x + B_0 y = C_0$$
$$A_1 x + B_1 y = C_1$$

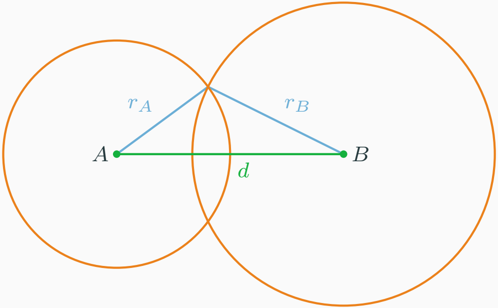
The intersection point is

$$x = \frac{(B_1 \cdot C_0 - B_0 \cdot C_1)}{D}$$
$$y = \frac{(A_0 \cdot C_1 - A_1 \cdot C_0)}{D}$$

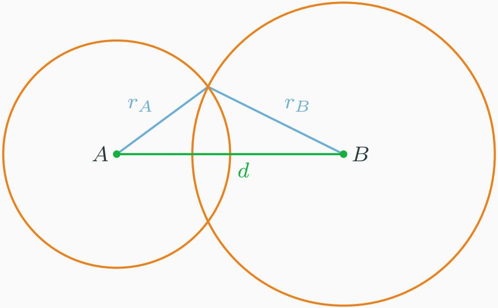
Where

$$D = A_0 \cdot B_1 - A_1 \cdot B_0$$

Quite similar problem is to find the intersections of two circles.

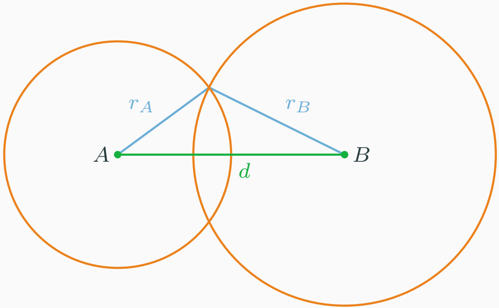


Quite similar problem is to find the intersections of two circles.



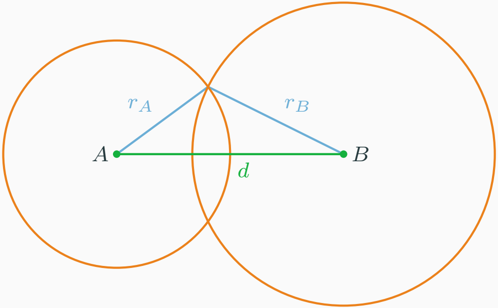
- If $d > r_0 + r_1$ the circles do not intersect.

Quite similar problem is to find the intersections of two circles.



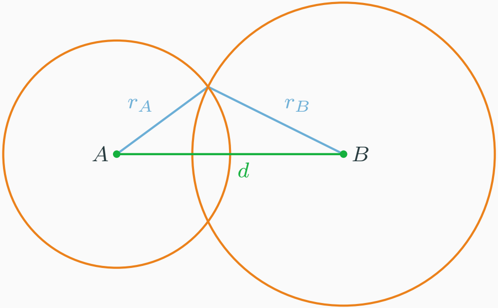
- If $d > r_0 + r_1$ the circles do not intersect.
- If $d < |r_0 - r_1|$, one circles is contained within the other.

Quite similar problem is to find the intersections of two circles.



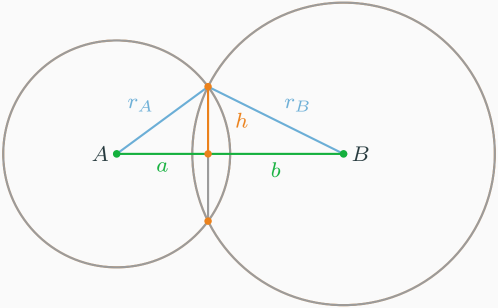
- If $d > r_0 + r_1$ the circles do not intersect.
- If $d < |r_0 - r_1|$, one circles is contained within the other.
- If $d = 0$ and $r_0 = r_1$, the circles are the same.

Quite similar problem is to find the intersections of two circles.



- If $d > r_0 + r_1$ the circles do not intersect.
- If $d < |r_0 - r_1|$, one circles is contained within the other.
- If $d = 0$ and $r_0 = r_1$, the circles are the same.
- Let's look at the last case.

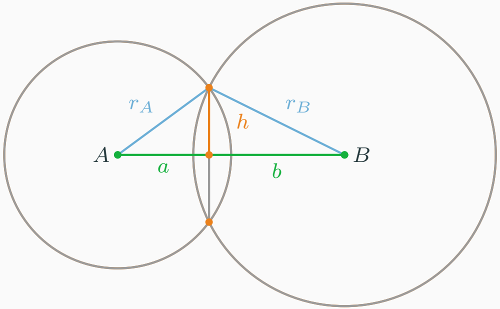
Quite similar problem is to find the intersections of two circles.



- We can solve for the vectors a and h from the equations

$$a^2 + h^2 = r_0^2 \quad b^2 + h^2 = r_1^2$$

Quite similar problem is to find the intersections of two circles.



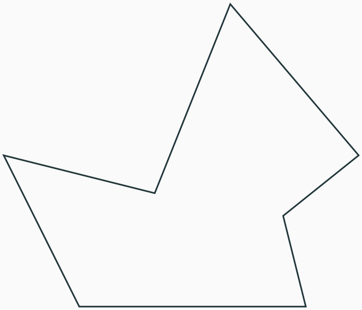
- We can solve for the vectors a and h from the equations

$$a^2 + h^2 = r_0^2 \quad b^2 + h^2 = r_1^2$$

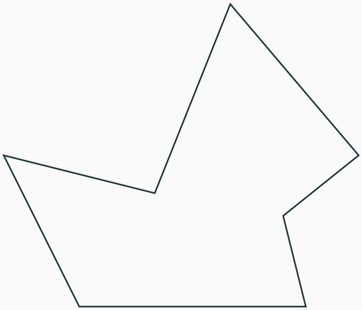
- We get

$$a = \frac{r_A^2 - r_B^2 + d^2)}{2 \cdot d}$$

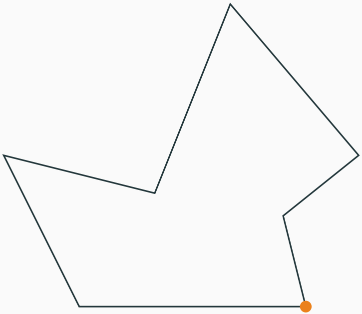
$$h^2 = r_A^2 - a^2$$



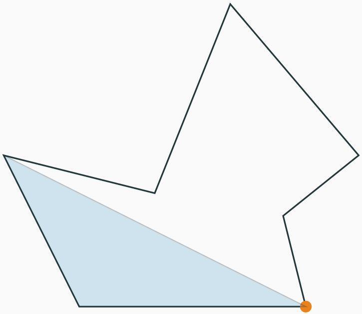
- Polygons are represented by a list of points in the order representing the edges.



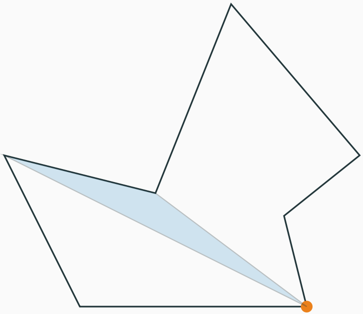
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area



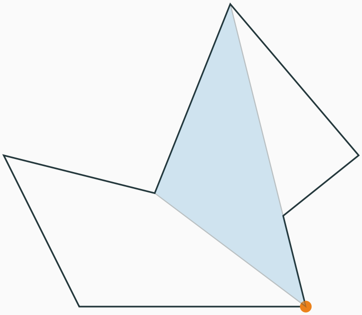
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.



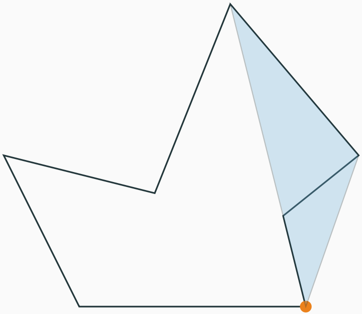
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
    - We pick one starting point.
    - Go through all the other adjacent pair of points and sum the area of the triangulation.



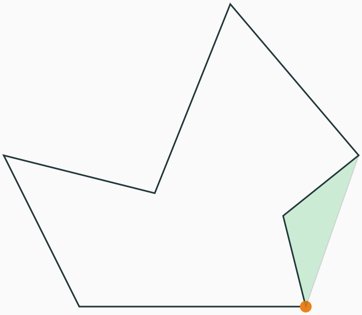
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



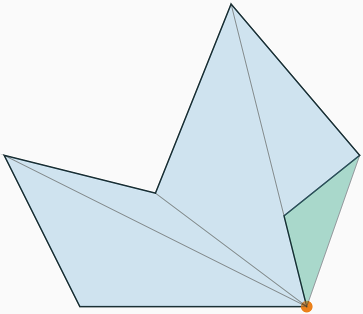
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.



- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.
  - Even if we sum up area outside the polygon, due to the cross product, it is subtracted later.



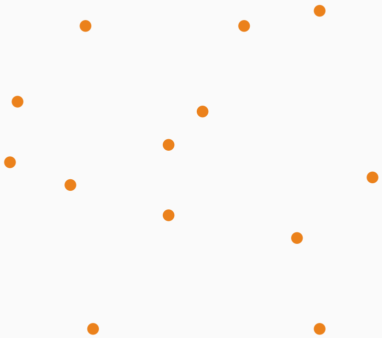
- Polygons are represented by a list of points in the order representing the edges.
- To calculate the area
  - We pick one starting point.
  - Go through all the other adjacent pair of points and sum the area of the triangulation.
  - Even if we sum up area outside the polygon, due to the cross product, it is subtracted later.

```
double polygon_area_signed(const vector<point> &p) {
    double area = 0;
    int cnt = size(p);
    for (int i = 1; i + 1 < cnt; i++){
        area += cross(p[i] - p[0], p[i + 1] - p[0])/2;
    }
    return area;
}
double polygon_area(vector<point> &p) {
    return abs(polygon_area_signed(p));
}
```

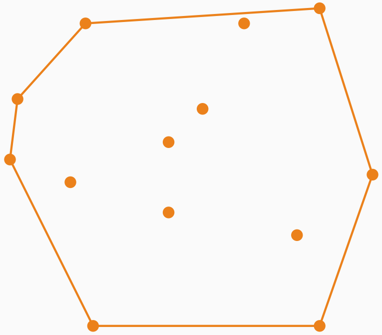
- Given a set of points, we want to find the convex hull of the points.

- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.

- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.



- Given a set of points, we want to find the convex hull of the points.
- The convex hull of points can be visualized as the shape formed by a rubber band around the set of points.



Graham scan:

Graham scan:

- Pick the point p_0 with the lowest y coordinate.

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points

Graham scan:

- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.

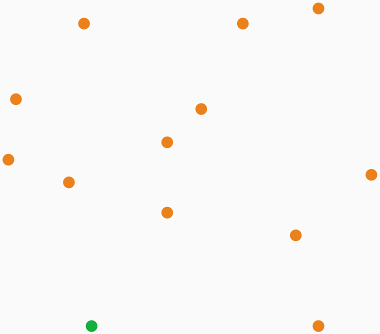
Graham scan:

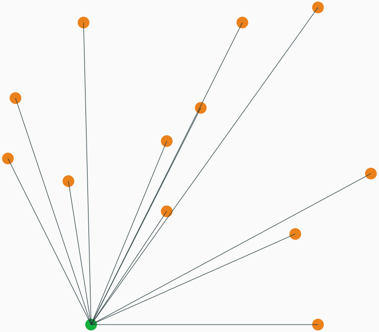
- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.
- Otherwise, add the current point to the convex set.

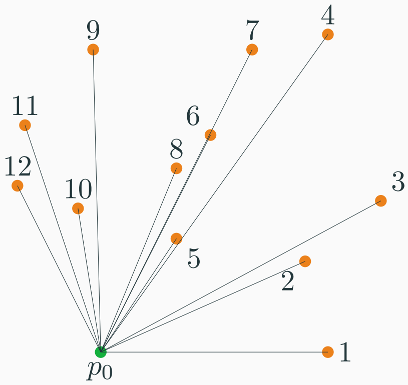
Graham scan:

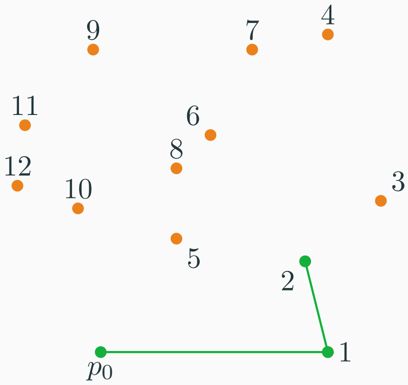
- Pick the point p_0 with the lowest y coordinate.
- Sort all the points by polar angle with p_0.
- Iterate through all the points
- If the current point forms a clockwise angle with the last two points, remove last point from the convex set.
- Otherwise, add the current point to the convex set.

Time complexity $O(N \log N)$.

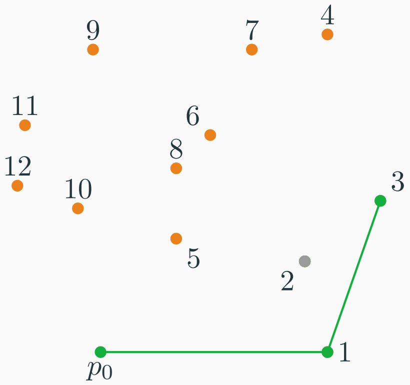


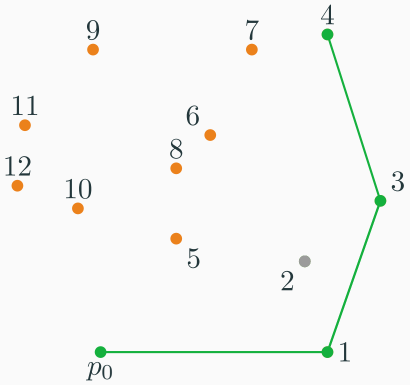



9
7
4
11
6
8
12
10
3
5
2
1
p_0

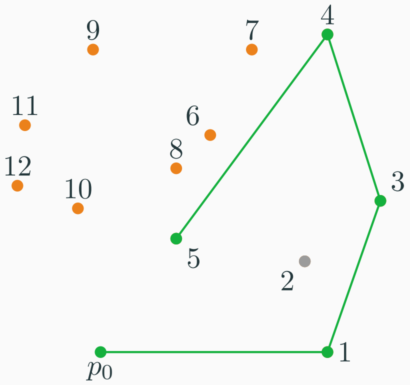


9
7
4
11
6
8
12
10
3
5
2
1
p_0

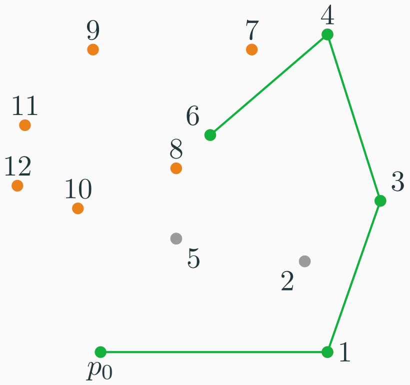

9
7
4
11
6
8
12
10
3
5
2
p_0
1



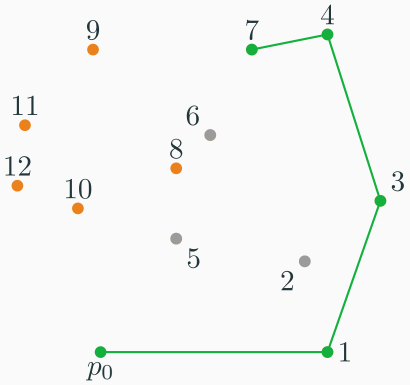
4
9
7
11
6
8
3
12
10
5
2
1
p_0

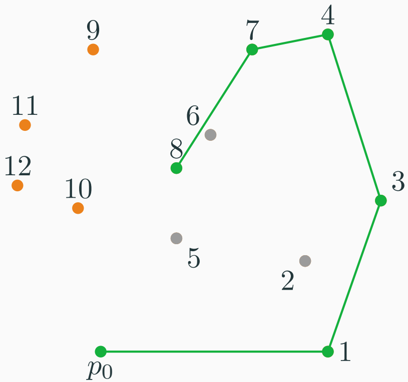


9
7
4
11
6
8
12
10
3
5
2
1
p_0

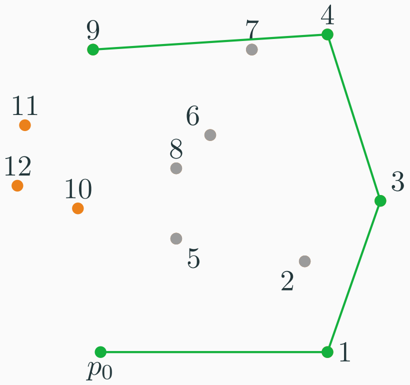


9
7
4
11
6
8
3
12
10
5
2
1
p_0

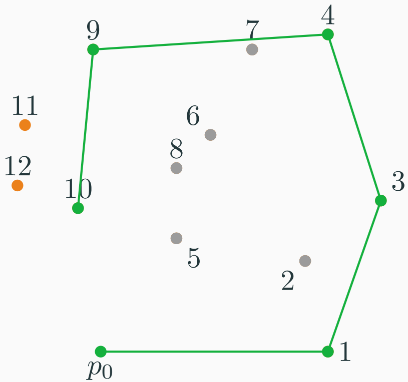

9
7
4
11
6
8
12
3
10
5
2
p_0
1



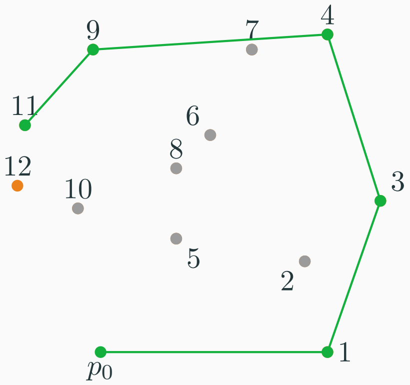
9
7
4
11
6
8
12
10
3
5
2
1
p_0

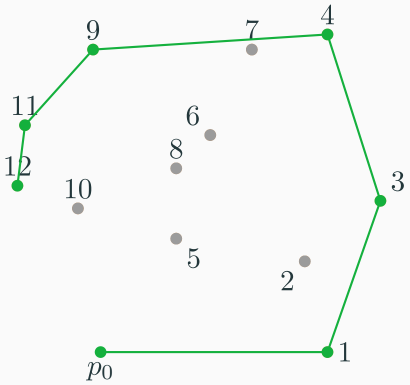


9
7
4
11
6
8
12
10
3
5
2
1
p_0

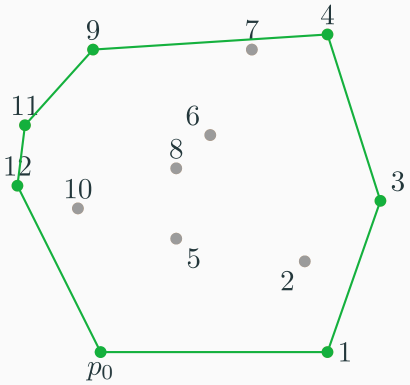


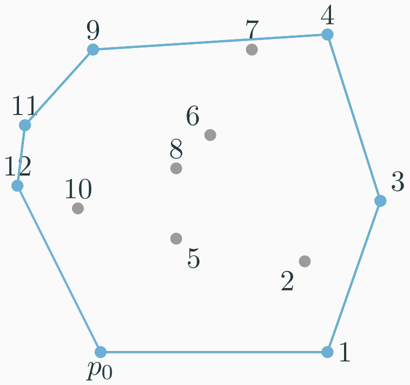
9
7
4
11
6
8
12
10
3
5
2
1
p_0


9
7
4
11
6
8
12
10
3
5
2
1
p_0



9
7
4
11
6
8
12
3
10
5
2
1
p_0


9
7
4
11
6
8
12
10
3
5
2
1
p_0



9
7
4
11
6
8
12
10
3
5
2
1
p_0

```
point hull[MAXN];
bool cmp(const point &a, const point &b) {
  return abs(real(a) - real(b)) > EPS ?
    real(a) < real(b) : imag(a) < imag(b); }
int convex_hull(vector<point> p) {
    int n = size(p), l = 0;
    sort(p.begin(), p.end(), cmp);
    for (int i = 0; i < n; i++) {
        if (i > 0 && p[i] == p[i - 1])
            continue;
        while (l >= 2 && ccw(hull[l - 2], hull[l - 1], p[i]) >= 0)
            l--;
        hull[l++] = p[i]; }
    int r = l;
    for (int i = n - 2; i >= 0; i--) {
        if (p[i] == p[i + 1])
            continue;
        while (r - l >= 1 && ccw(hull[r - 2], hull[r - 1], p[i]) >= 0)
            r--;
        hull[r++] = p[i]; }
    return l == 1 ? 1 : r - 1; }
```

Many other algorithms exist

Many other algorithms exist

- Gift wrapping aka Jarvis march.

Many other algorithms exist

- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.

Many other algorithms exist

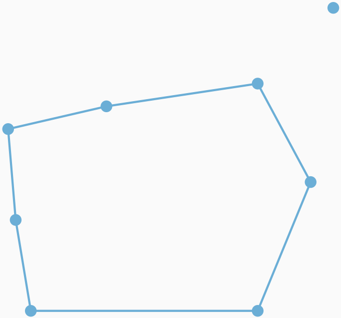
- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.
- Divide and conquer.

Many other algorithms exist

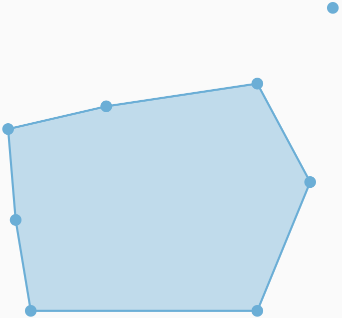
- Gift wrapping aka Jarvis march.
- Quick hull, similar idea to quicksort.
- Divide and conquer.

Some can be extended to three dimensions, or higher.

Simple algorithm to check if a point is in a convex polygon.

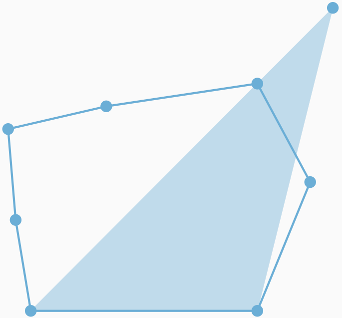


Simple algorithm to check if a point is in a convex polygon.



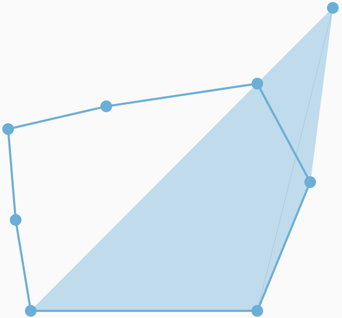
- We start by calculating the area of the polygon.

Simple algorithm to check if a point is in a convex polygon.



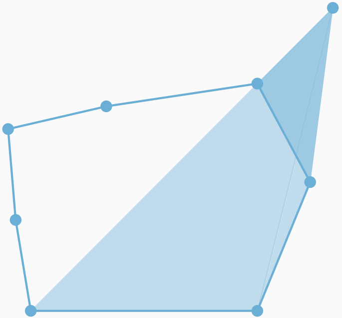
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



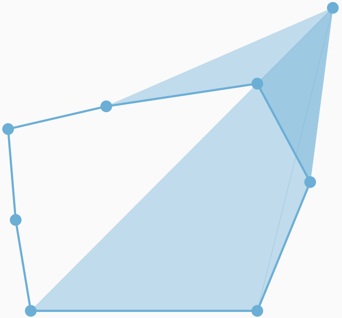
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



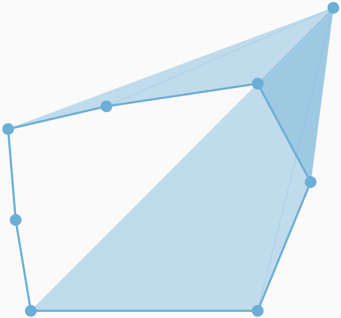
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



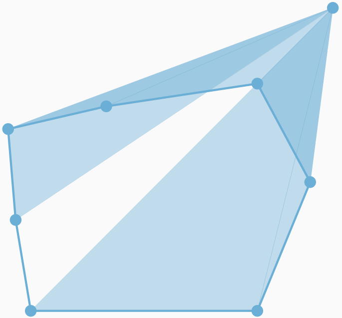
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



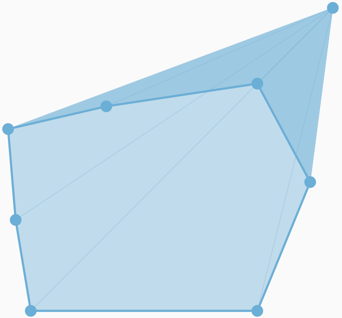
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



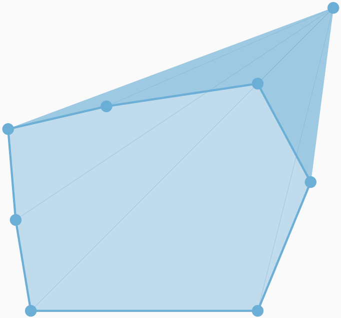
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



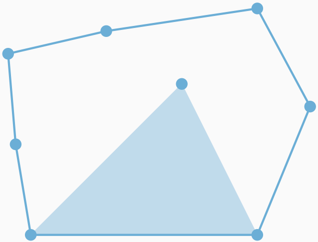
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.

Simple algorithm to check if a point is in a convex polygon.



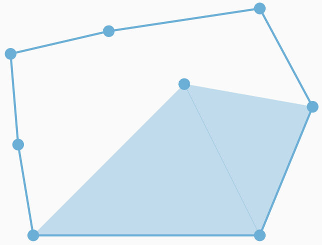
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



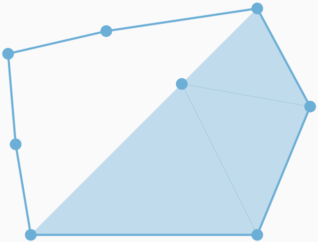
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



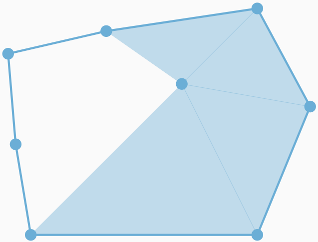
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



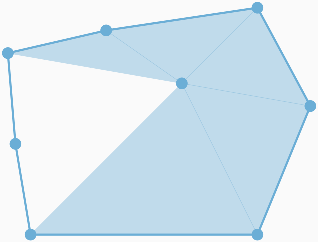
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



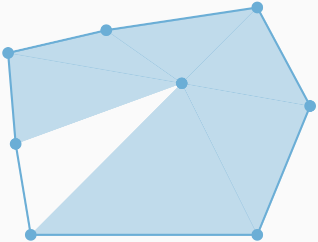
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



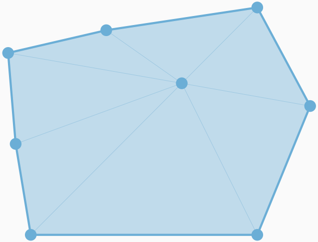
- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

Simple algorithm to check if a point is in a convex polygon.



- We start by calculating the area of the polygon.
- To check if our point is contained in the polygon we sum up the area of the triangles formed the point and every two adjacent points.
- The total area of the triangles is equal to the area of the polygon iff the point is inside the polygon.

How about non convex polygon?

How about non convex polygon?

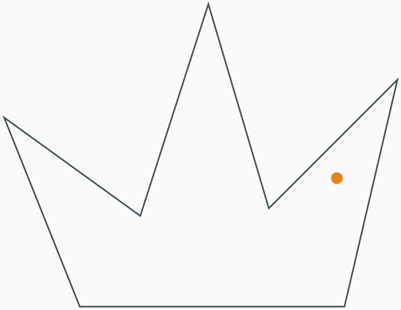
- The *even-odd rule* algorithm.

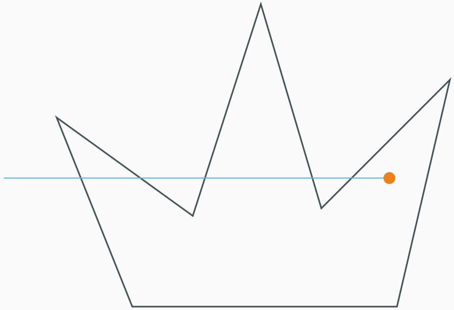
How about non convex polygon?

- The *even-odd rule* algorithm.
- We examine a ray passing through the polygon to the point.

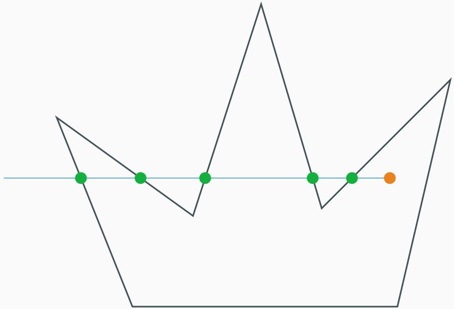
How about non convex polygon?

- The *even-odd rule* algorithm.
- We examine a ray passing through the polygon to the point.
- If the ray crosses the boundary of the polygon, then it alternately goes from outside to inside, and outside to inside.

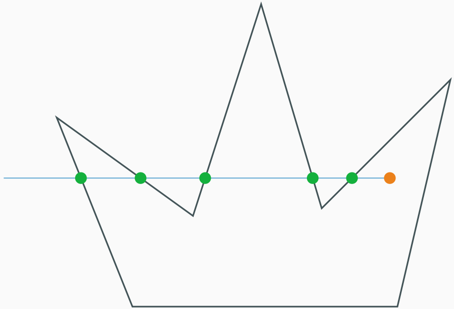




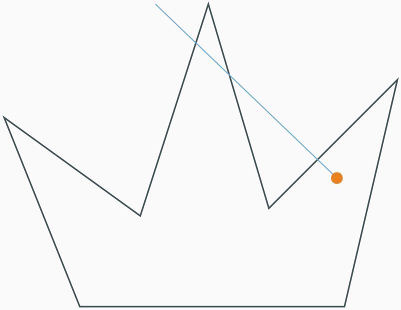
- Ray from the outside of the polygon to the point.



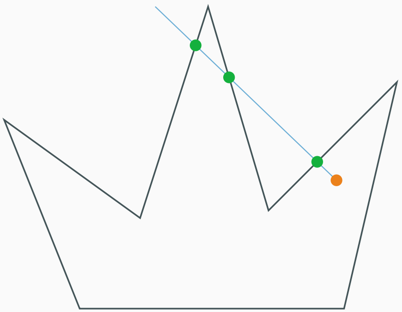
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.



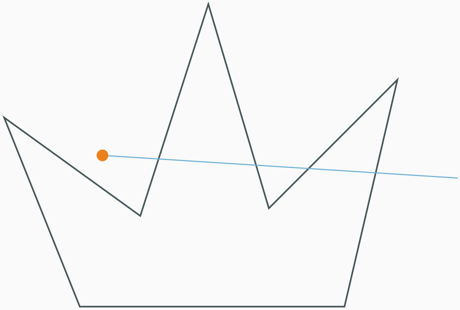
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.



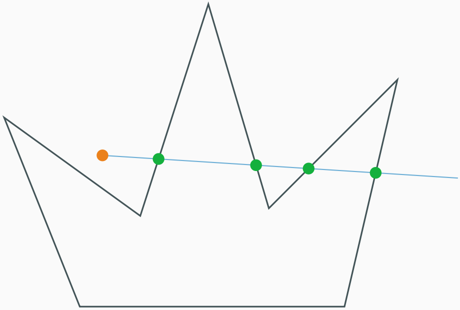
- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.



- Ray from the outside of the polygon to the point.
- Count the number of intersection points.
- If odd, then the point is inside the polygon.
- If even, then the point is outside the polygon.
- Does not matter which ray we pick.

An algorithm

- Computational geometry has a lot of impressive and technical algorithms.
- The most famous one is probably Delaunay triangulation.
- But that one is a bit too hard for this course, so we will instead look at the classical closest point algorithm.
- We are given n points in the plan, find the pair of points that are closest to one another.
- We can clearly solve this in $\mathcal{O}(n^2)$ time, but can we do better?

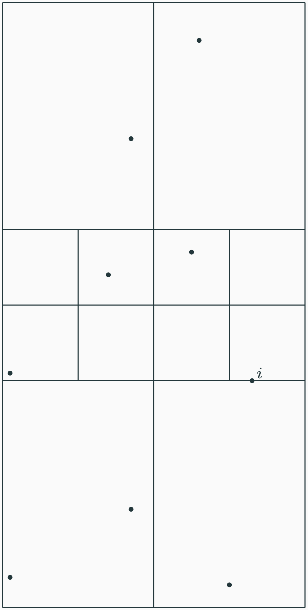
Divide and conquer

- We sort the points by x-coordinate and split the list in half.
- Let x_0 be such that it's between the coordinates of the left and right halves.
- Start by solving each half recursively.
- We now have to find if there's some pair with one point in each half that does better.
- We can't simply try all pairs, that's too slow. Suppose the smallest distance we found recursively was d.
- Then we can ignore all points with x-coordinte outside $[x_0 - d, x_0 + d]$.
- Sort the points inside of this interval by their y-coordinate.
- The big trick is now that we only need to consider a few neighbours for each point.

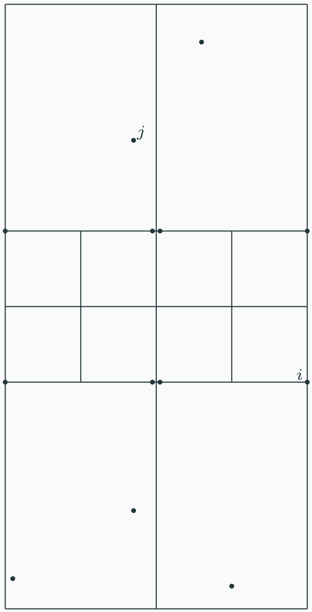
Neighbours

- Divide the area above x_i into 8 squares, each with side length $d/2$.
- If he distance between all points in each half is at least d, then we can have at most each point per square.
- All points outside these squares are at a distance of at least d from x_i, so we can ignore them.
- Thus we only need to look at the distance from x_i to x_j when $j - i \leq 7$.

Diagram



Diagram



Complexity

- Each recursive call is $\mathcal{O}(n\log(n))$.
- Thus by the master theorem the total complexity is $\mathcal{O}(n\log^2(n))$.
- If we sort the y values as we go using mergesort, we can actually do each call in $\mathcal{O}(n)$.
- This way the complexity is actually $\mathcal{O}(n\log(n))$.

Strings

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

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:

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

String matching

Example:

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

String matching

Example:

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

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: `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: `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`
- 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 $\pi[q] = \max\{k : k < q \text{ and } T[1 \dots k] \text{ is a suffix of } T[1 \dots q]\}$
- 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; }
```

Tries

Sets of strings

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

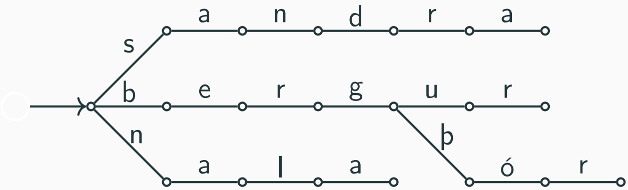
- But string comparisons are actually pretty expensive...

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

Tries

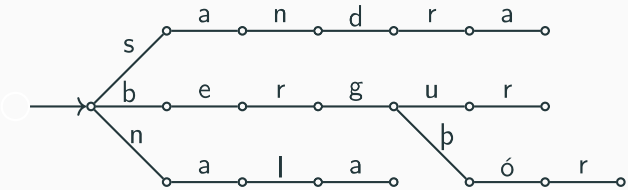
- Tries contain strings not at every node, but as paths in a tree.
- Each node only has a character and we say the trie contains the string if you can get it by walking along nodes starting at the root.
- The nodes can also carry additional data, quite a lot in fact, as we will see later.

Example



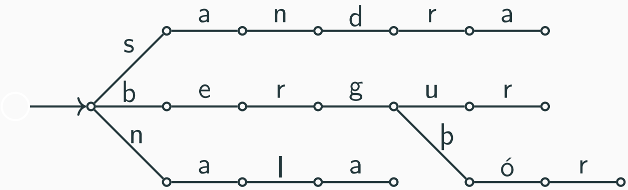
- Examples of strings in this trie include:

Example



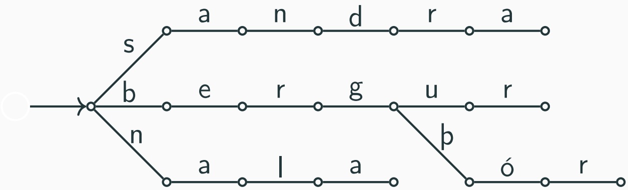
- Examples of strings in this trie include:

Example



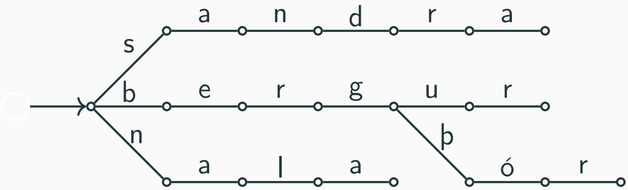
- Examples of strings in this trie include:
    - „sandra",

Example



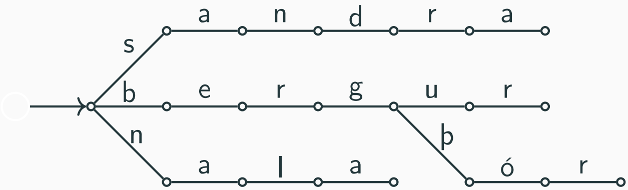
- Examples of strings in this trie include:
  - „sandra",
  - „nala",

Example



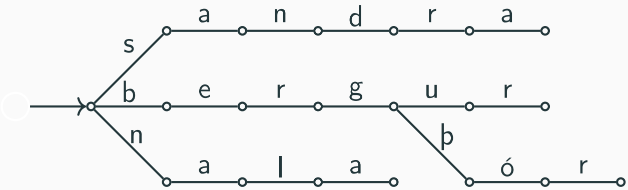
- Examples of strings in this trie include:
    - „sandra",
    - „nala",
    - „bergur",

Example



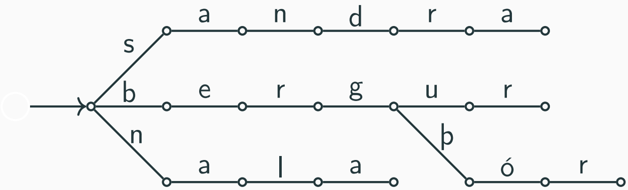
- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór",

Example



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

Example

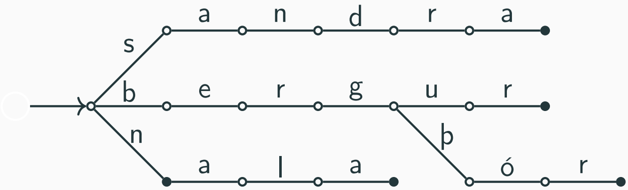


- Examples of strings in this trie include:
    - „sandra",
    - „nala",
    - „bergur",
    - „bergþór",
    - „san" and
    - „" (empty string)

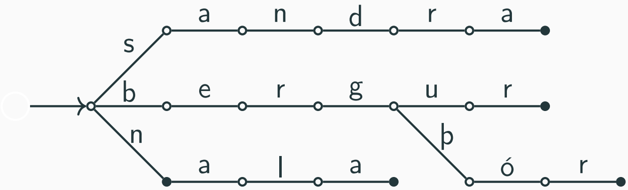
End nodes

- 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

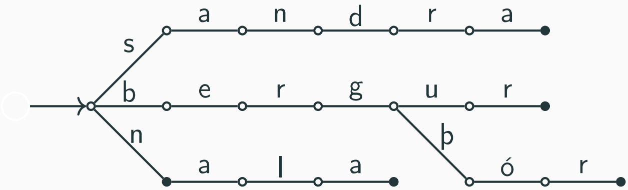


Example



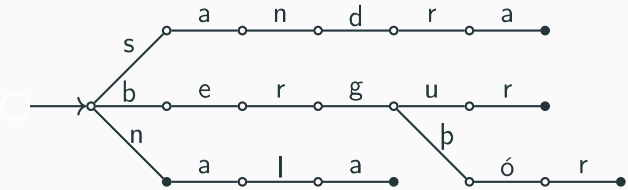
- The strings in the trie are:

Example



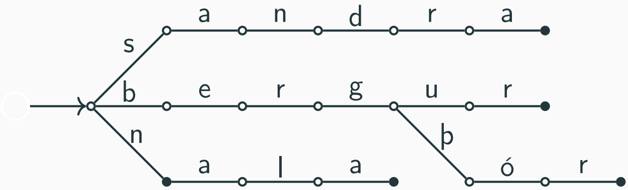
- The strings in the trie are:
    - „sandra",

Example



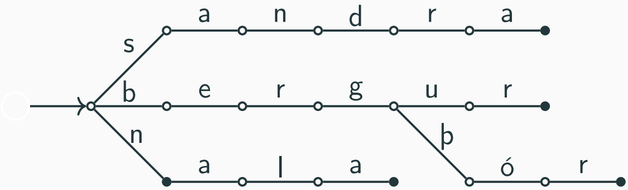
- The strings in the trie are:
  - „sandra",
  - „nala",

Example



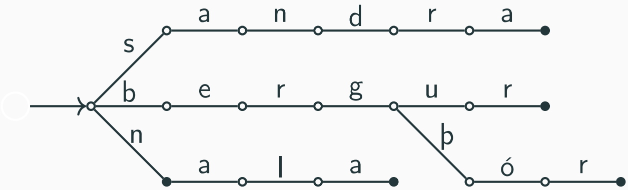
- The strings in the trie are:
  - „sandra",
  - „nala",
  - „bergur",

Example



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

Example

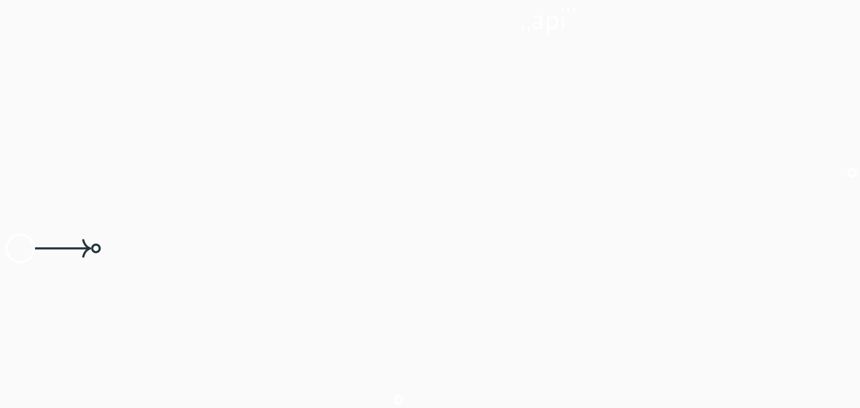


- The strings in the trie are:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór" and
  - „n"

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

„pi”

Example

Example

Example

„api“

Example

„apar"

Example

„apar"

Example

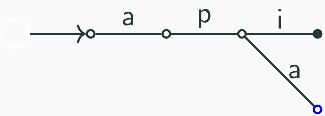
„par''

Example

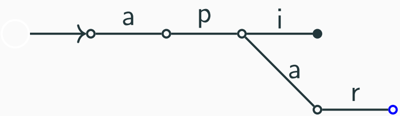
„ar"

Example

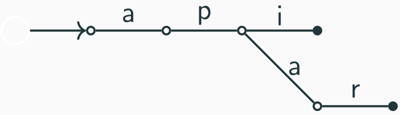
„r"



Example

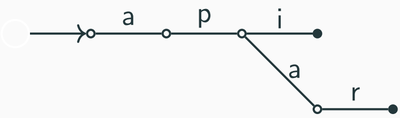


Example



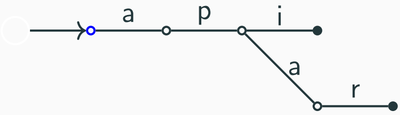
Example

„apaköttur"



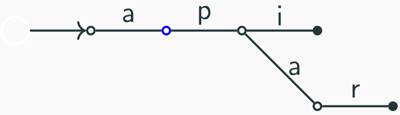
Example

„apaköttur"



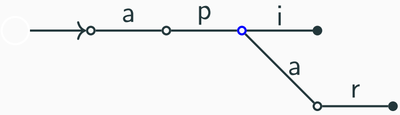
Example

„pakөttur"



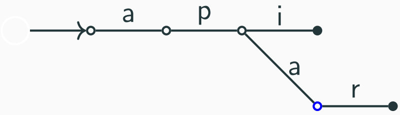
Example

„aköttur"



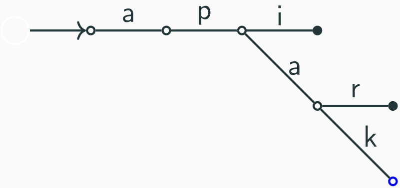
Example

„köttur"



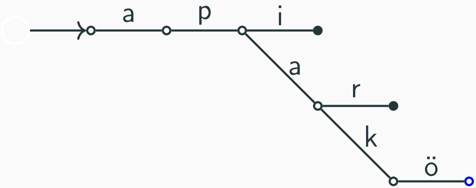
Example

„öttur"



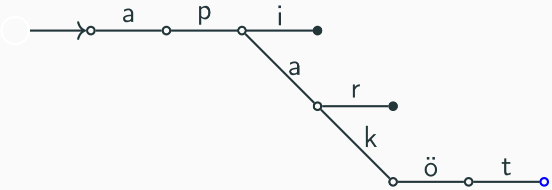
Example

„ttur"



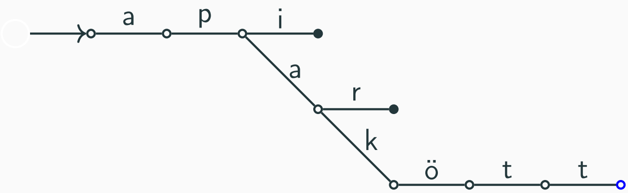
Example

„tur"



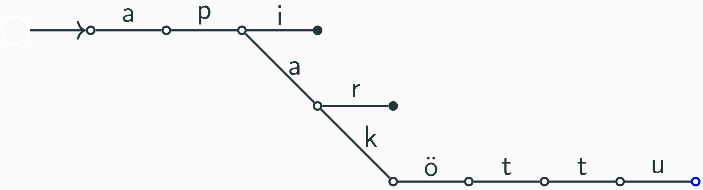
Example

„ur"

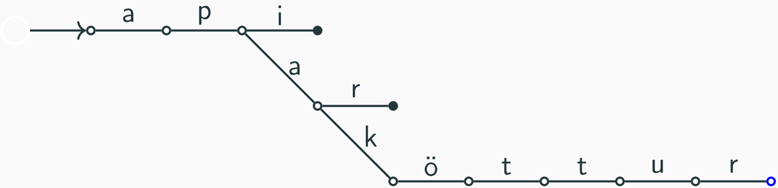


Example

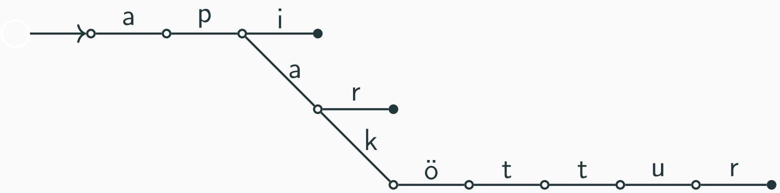
„r”



Example

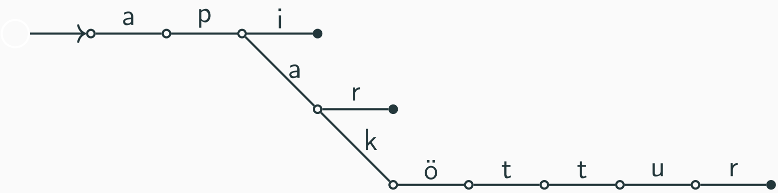


Example



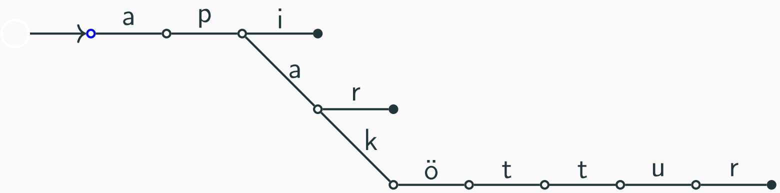
Example

„altari”



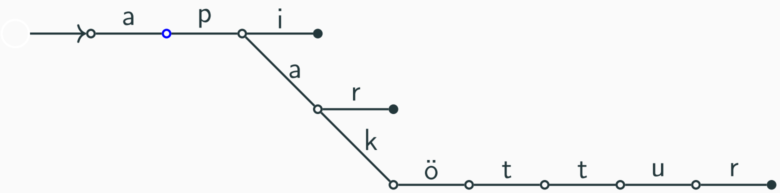
Example

„altari"



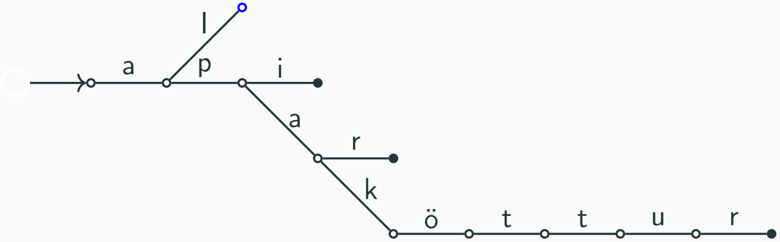
Example

„Itari"



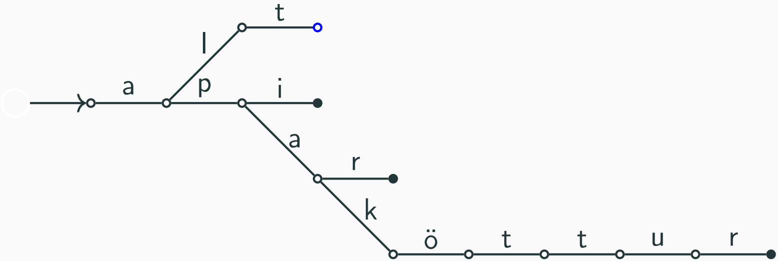
Example

„tari"



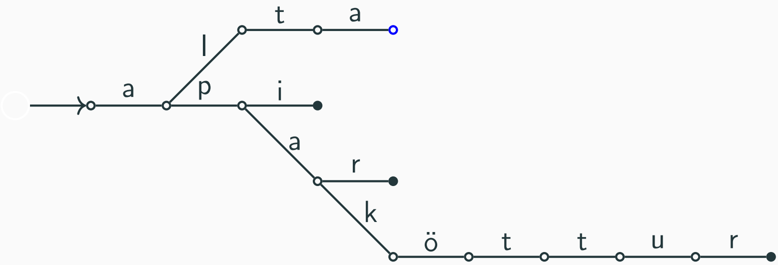
Example

„ari"



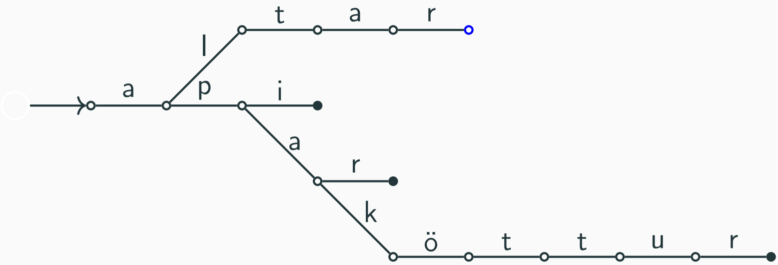
Example

„ri“

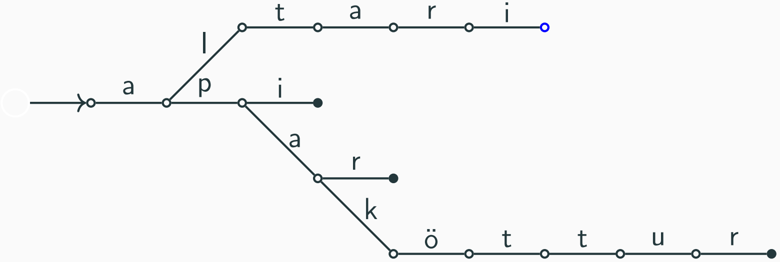


Example

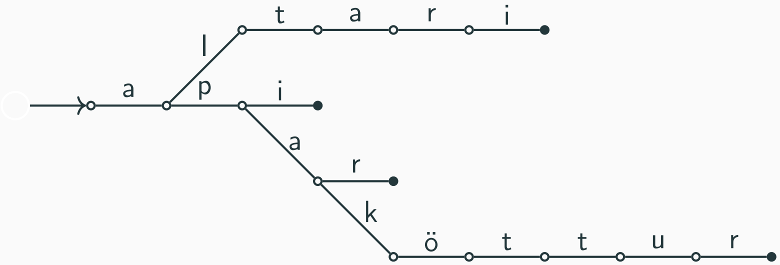
„i”



Example

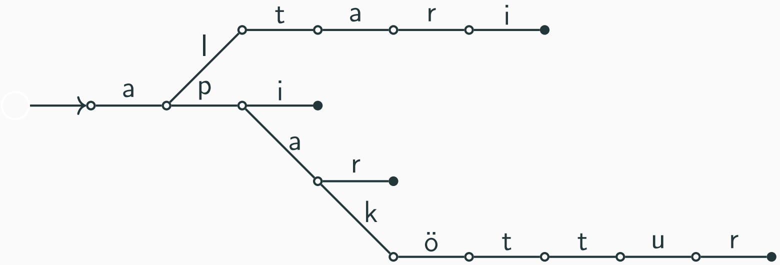


Example



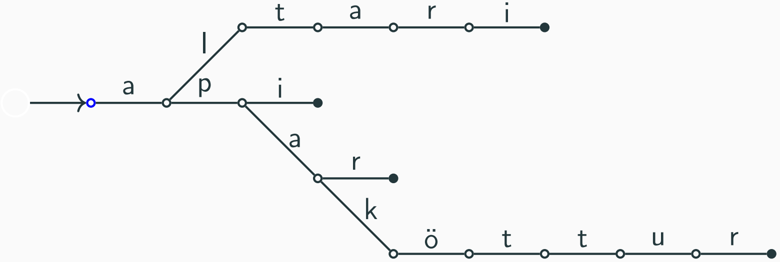
Example

„apaspil"



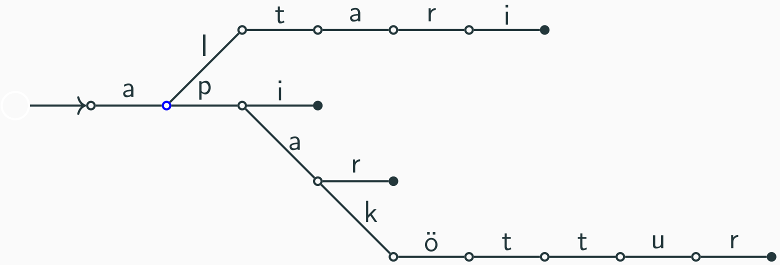
Example

„apaspil"



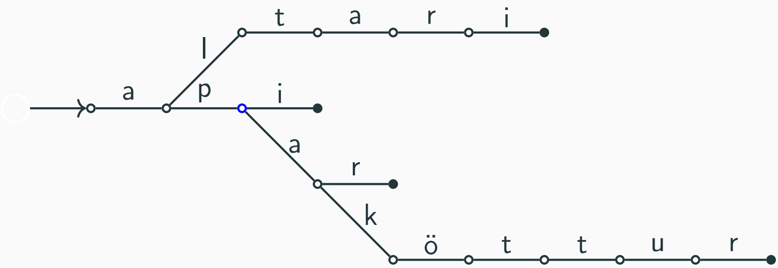
Example

„paspil”



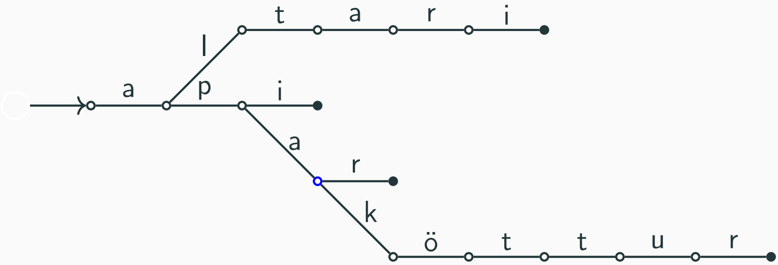
Example

„aspil"



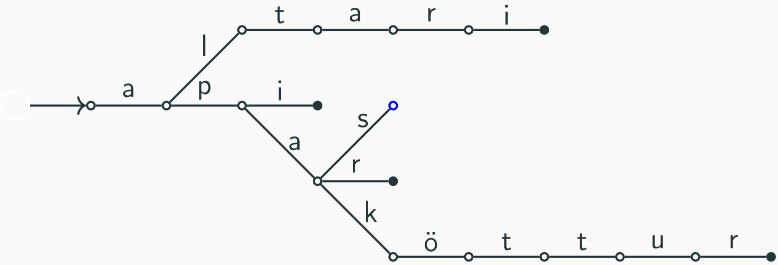
Example

„spil''



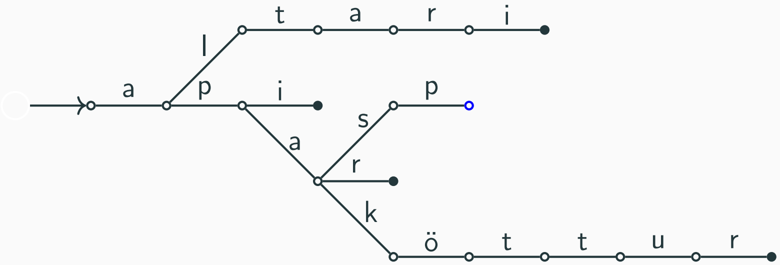
Example

„pil"



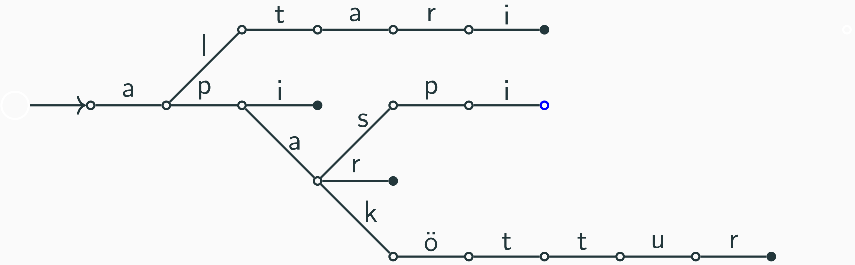
Example

„il"

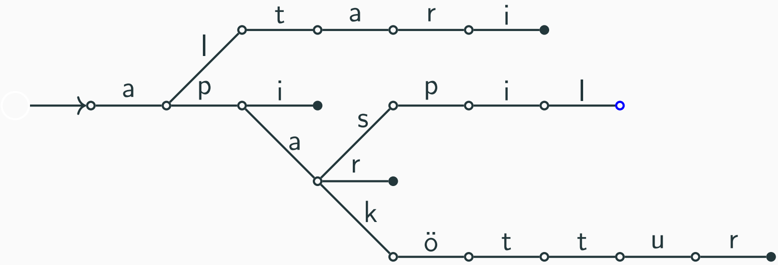


Example

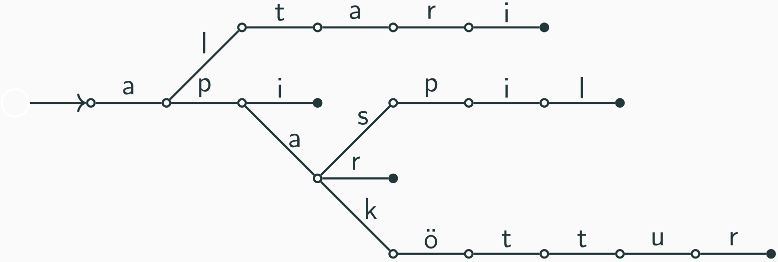
„l”



Example

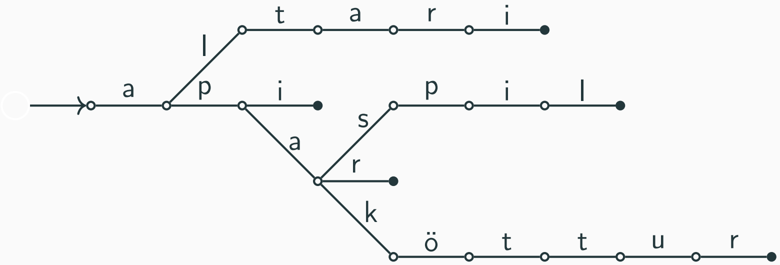


Example



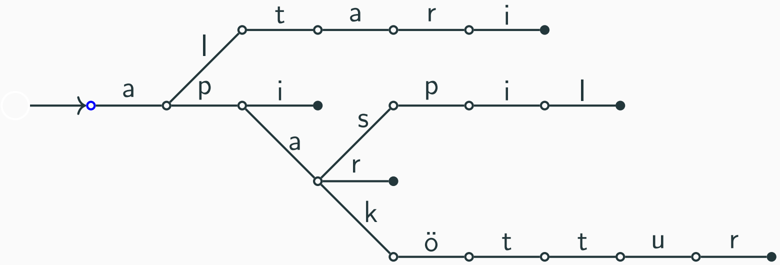
Example

„altaristafla"



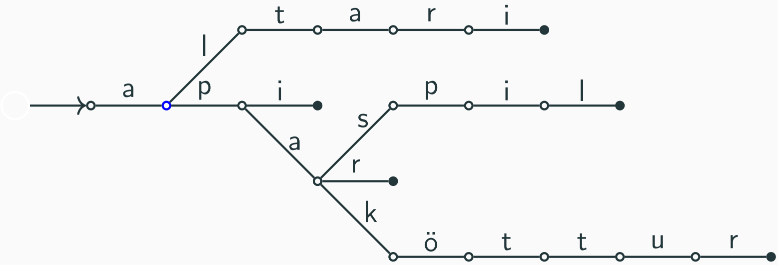
Example

„altaristafla"



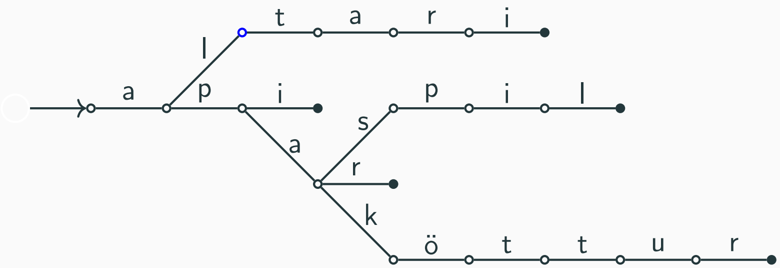
Example

„ltaristafla"



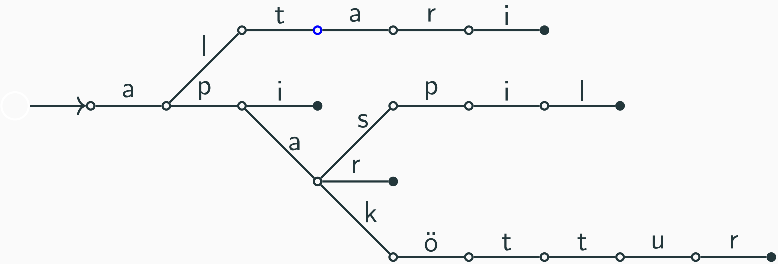
Example

„taristafla"



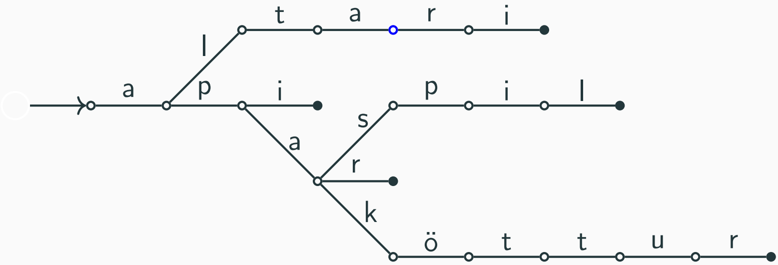
Example

„aristafla"



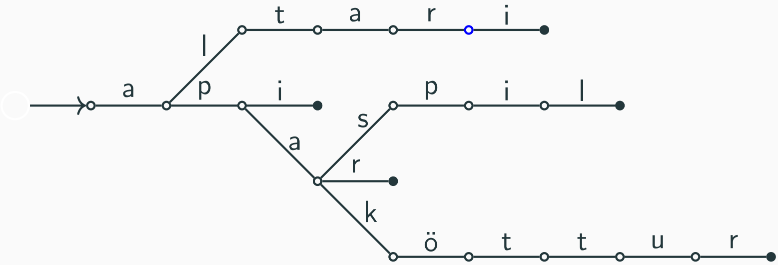
Example

„ristafla''



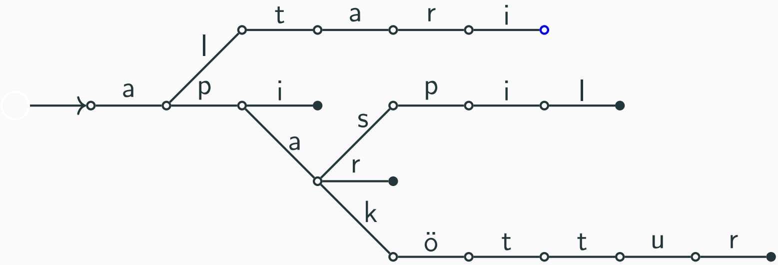
Example

„istafla”



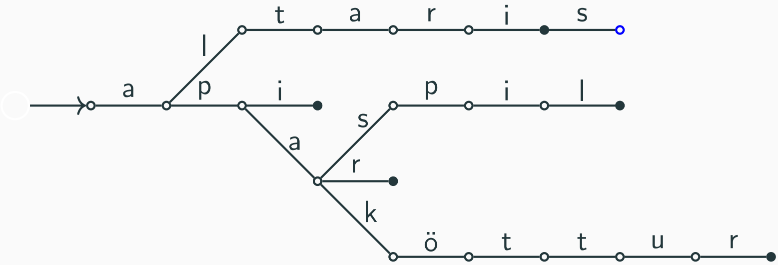
Example

„stafla"



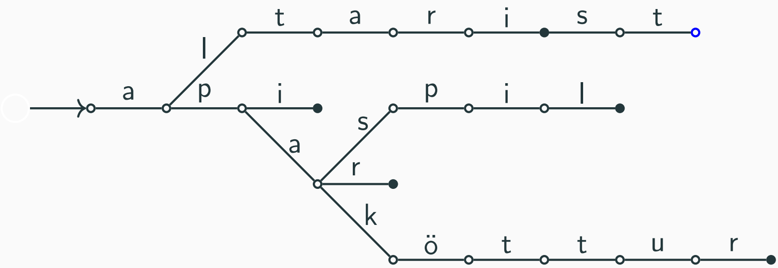
Example

„tafla"



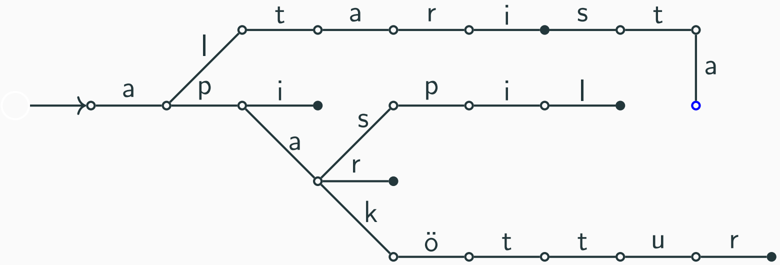
Example

„afla"



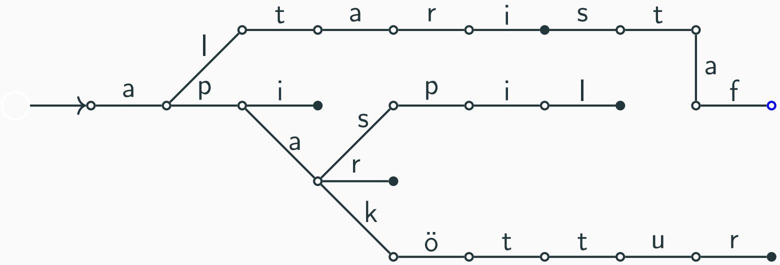
Example

„fla"



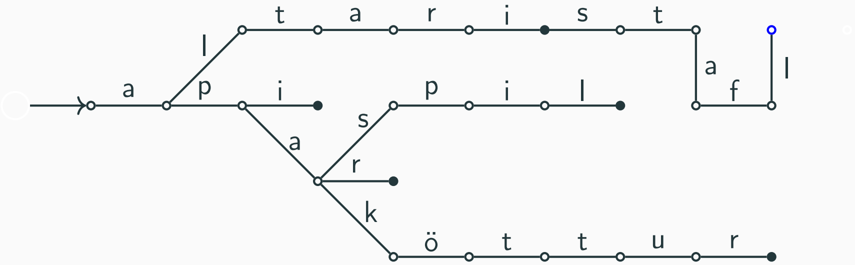
Example

„la"



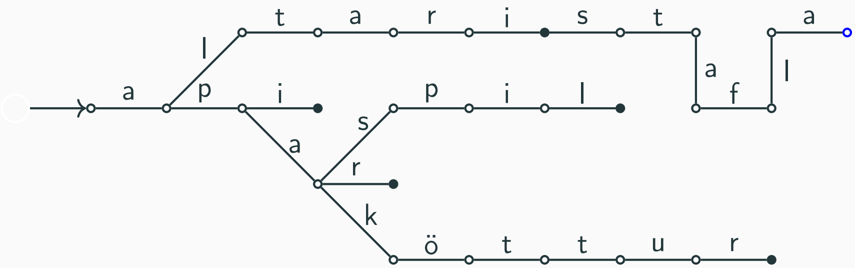
Example

„a"

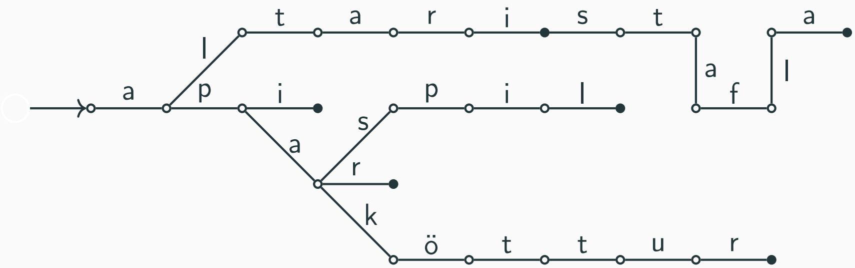


Example

„abi"

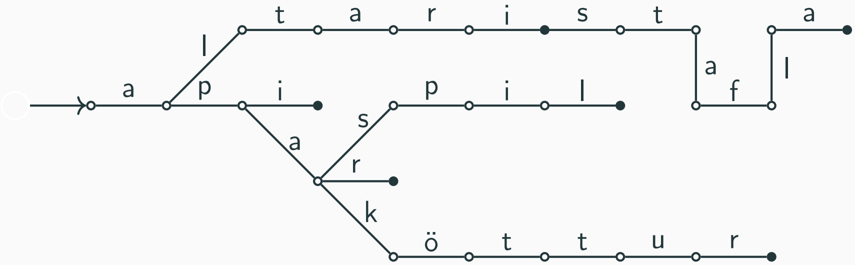


Example



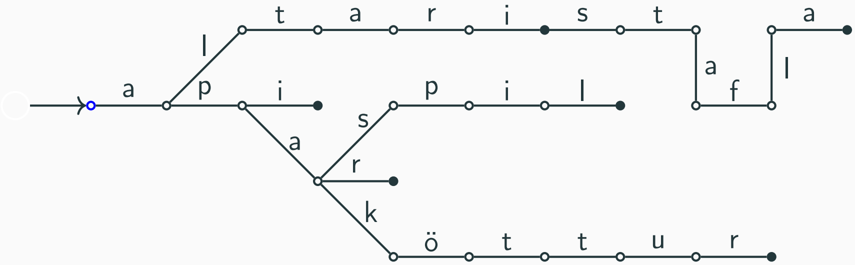
Example

„altarisganga”



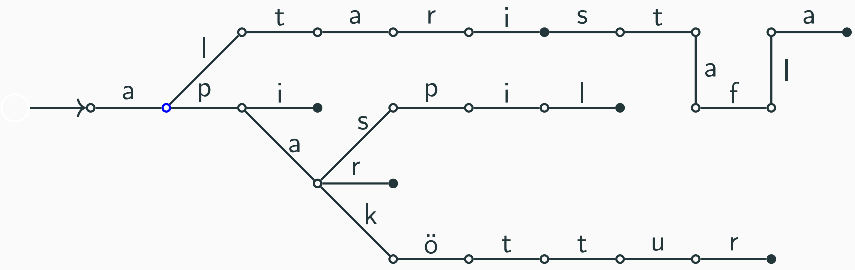
Example

„altarisganga"



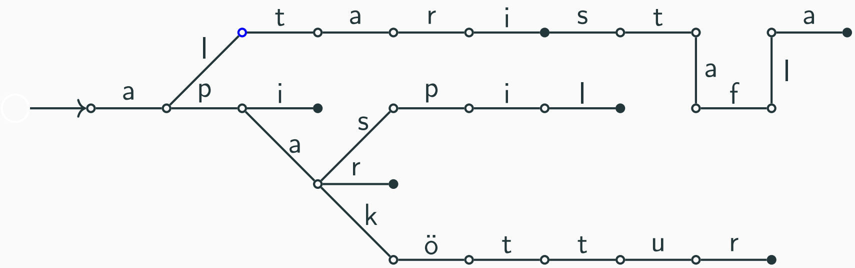
Example

„ltarisgänga”



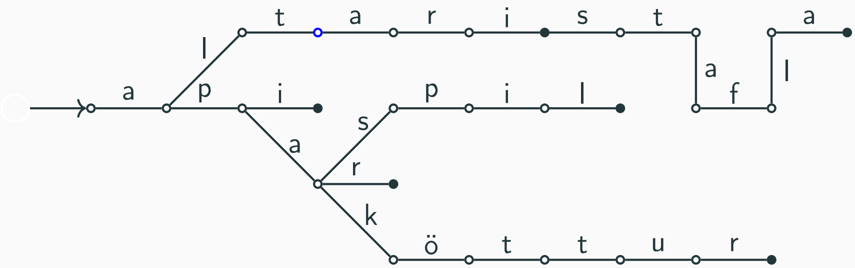
Example

„tarisgänga"



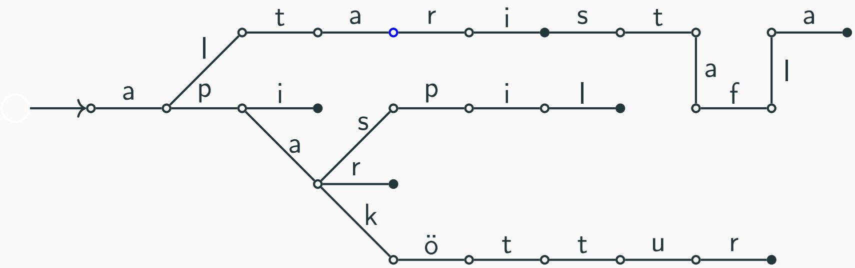
Example

„arisganga"



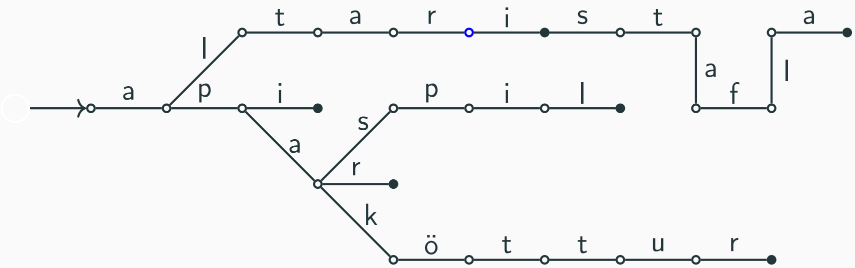
Example

„risganga"



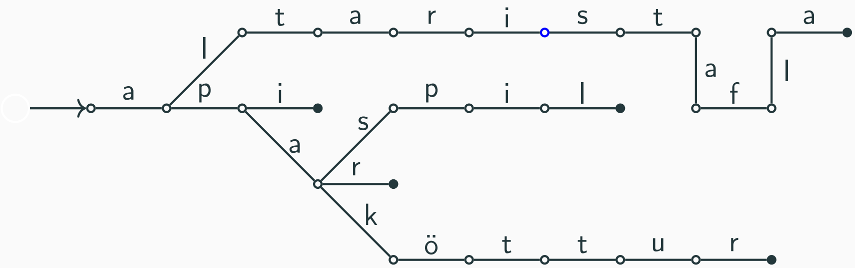
Example

„isganga"



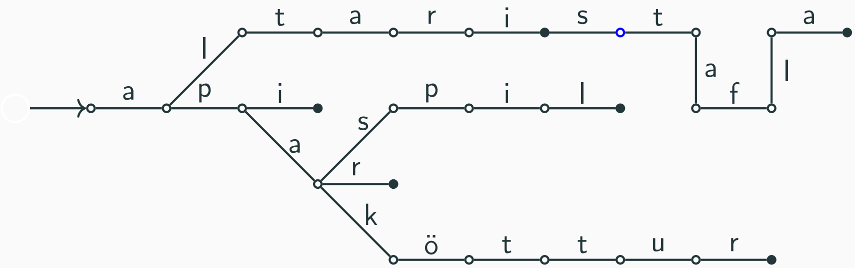
Example

„sganga''

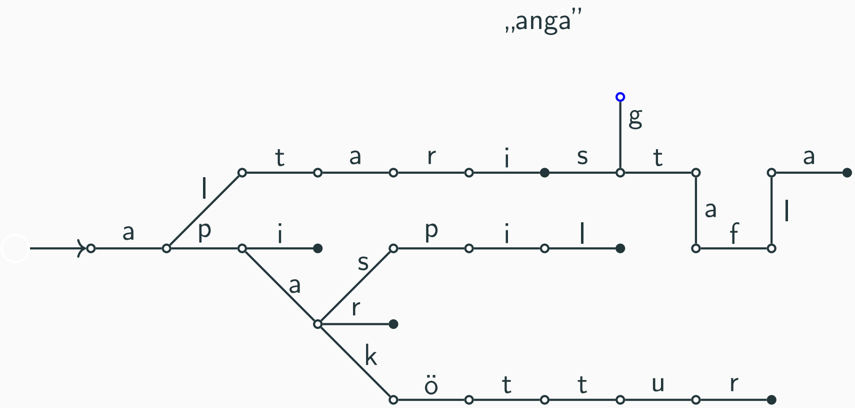


Example

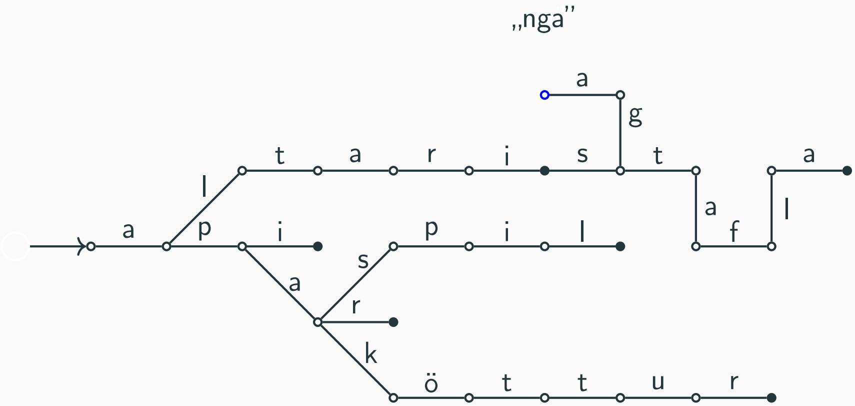
„ganga"



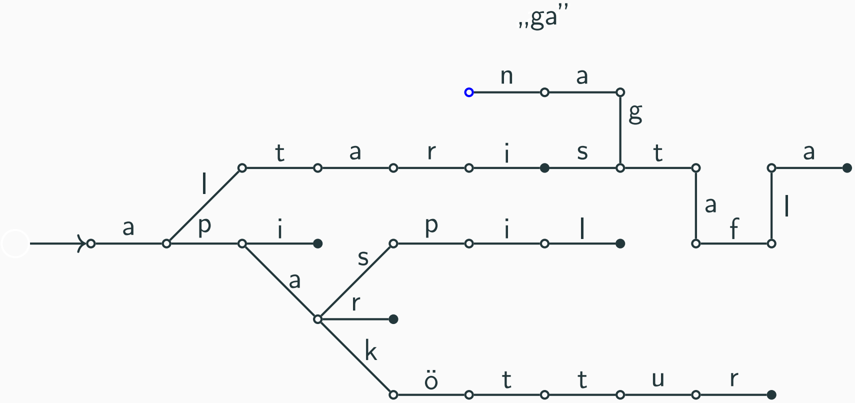
Example



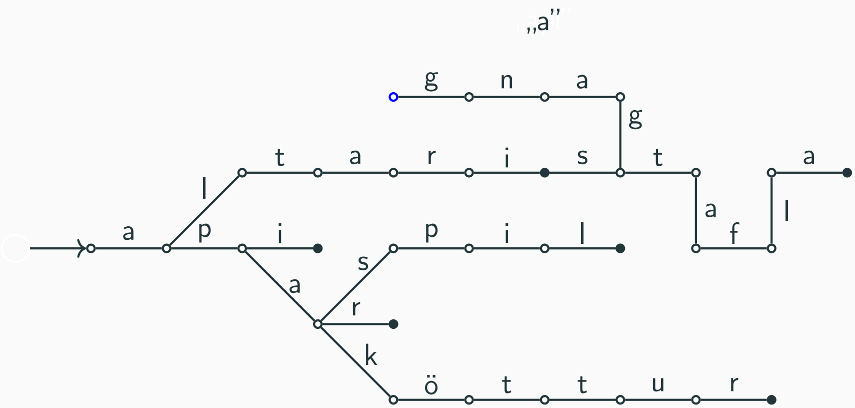
Example



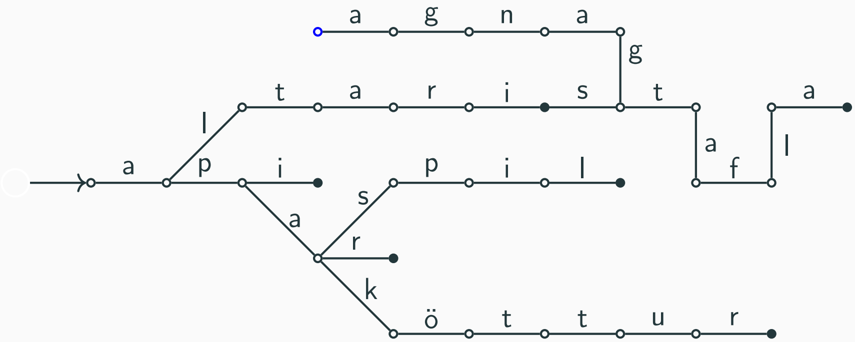
Example



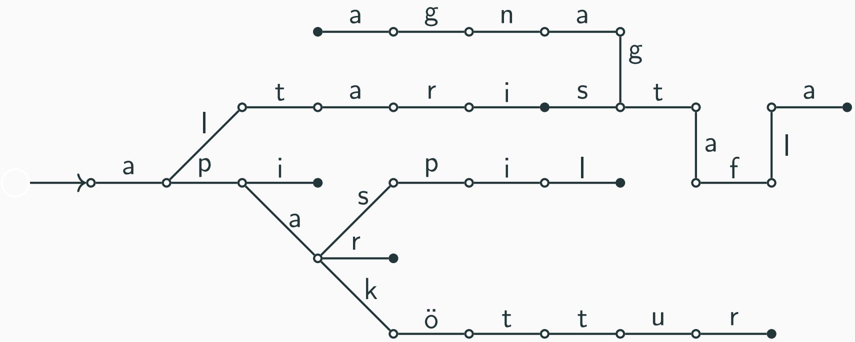
Example



Example



Example



Tries

```
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
- This can be improved by using a map/dict for children instead, but that does make lookup slower, tradeoffs as usual

String multimatching

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 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.

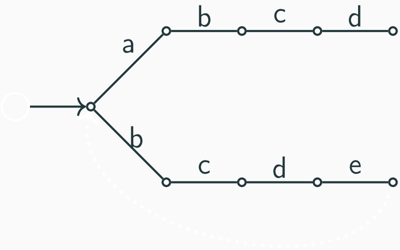
The automata

- 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.

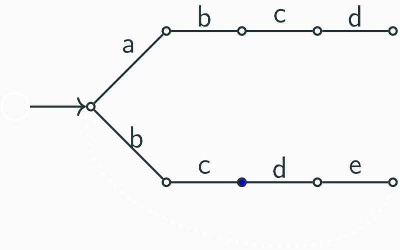
Suffix links

- How do we find the suffix links?
- We will cheat and borrow a method from a past week, dynamic programming.
- 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 p and $f(p, a) = v$. Then $g(v) = f(g(p), a)$.
- Thus we have a recursive formula we can use.

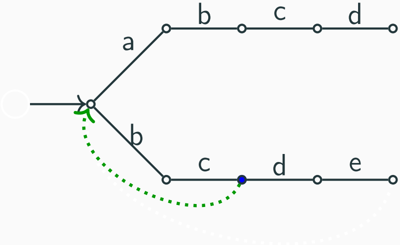
Example



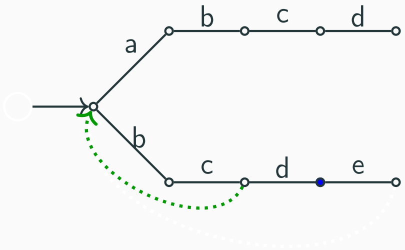
Example



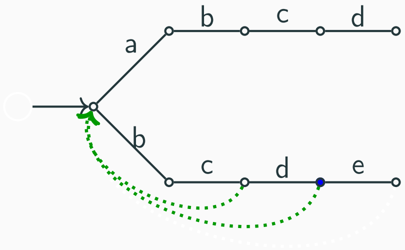
Example



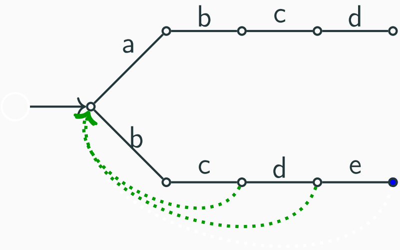
Example



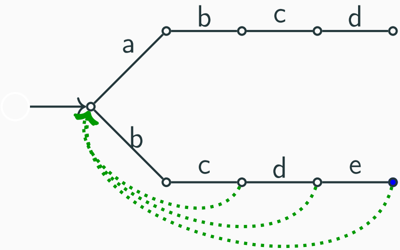
Example



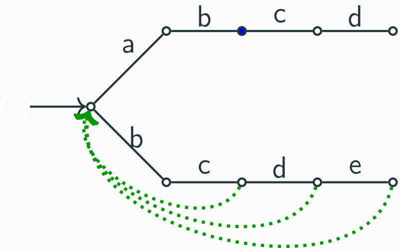
Example



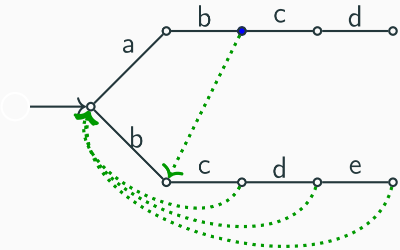
Example



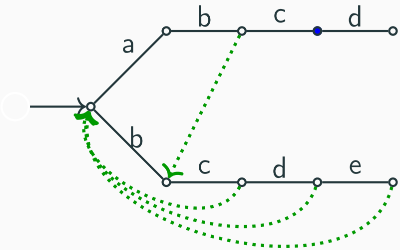
Example



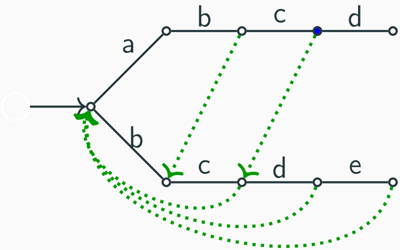
Example



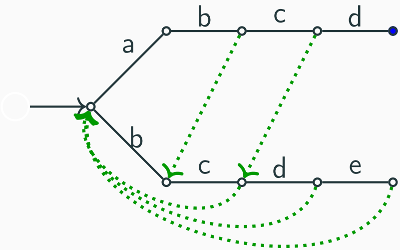
Example



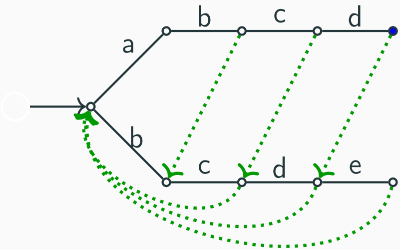
Example



Example



Example

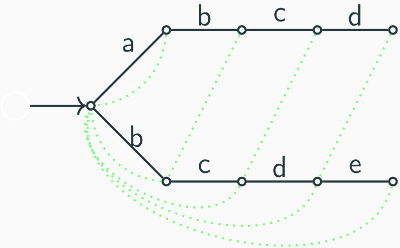


End nodes

- 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.

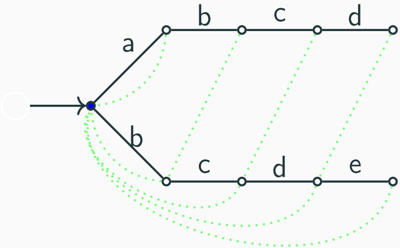
Example

„abcdcdeaaabcdeabcxab"



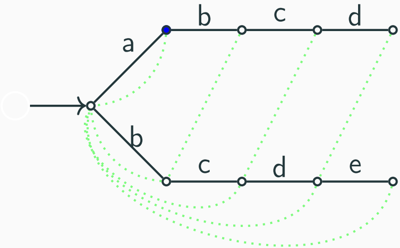
Example

„abcdcdeaaabcdeabcxab"



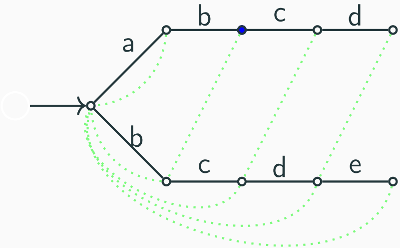
Example

„bcdcdeaaabcdeabcxab"



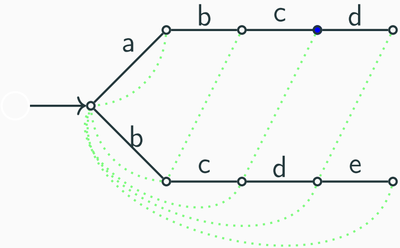
Example

„cdcdeaaabcdeabcxab"



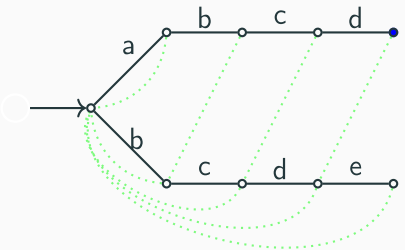
Example

„dcdeaaabcdeabcxab"



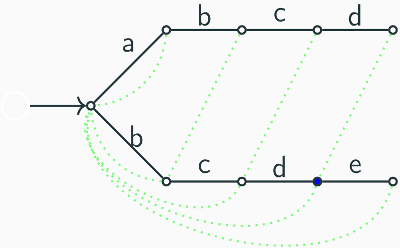
Example

„cdeaaabcdeabcxab"



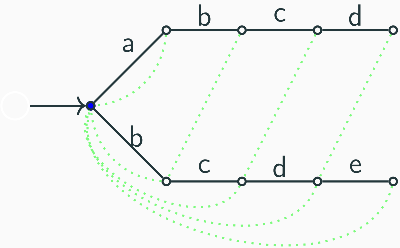
Example

„cdeaaabcdeabcxab"



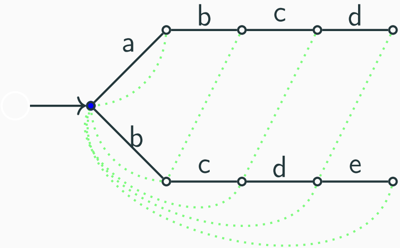
Example

„cdeaaabcdeabcxab"



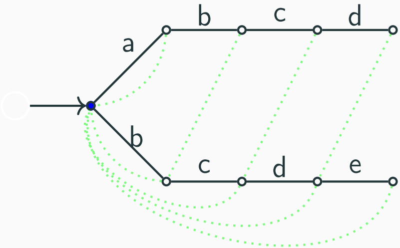
Example

„deaaabcdeabcxab"



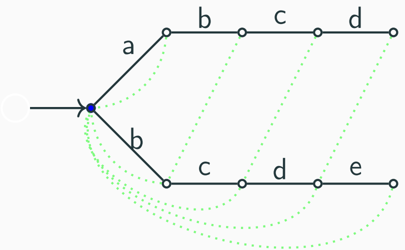
Example

„eaaabcdeabcxab"



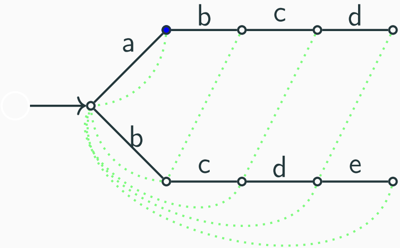
Example

„aaabcdeabcxab"



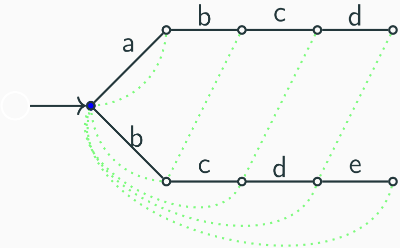
Example

„aabcdeabcxab"



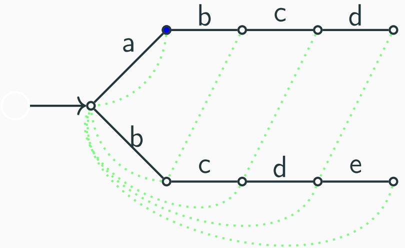
Example

„aabcdeabcxab"

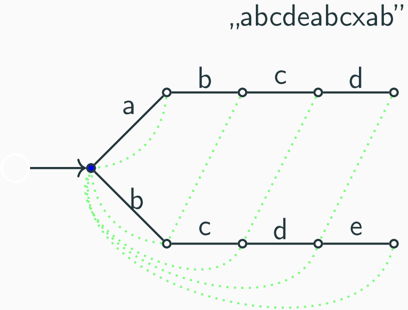


Example

„abcdeabcxab"

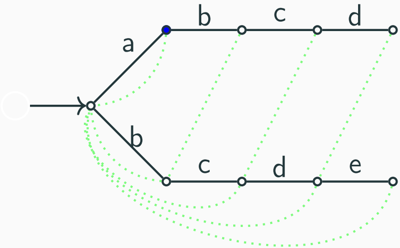


Example



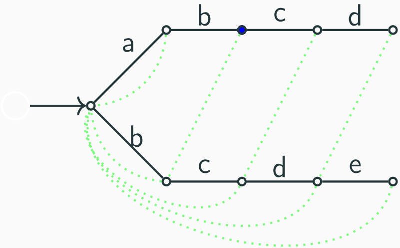
Example

„bcdeabcxab"

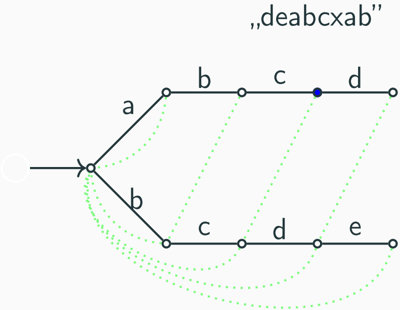


Example

„cdeabcxab"

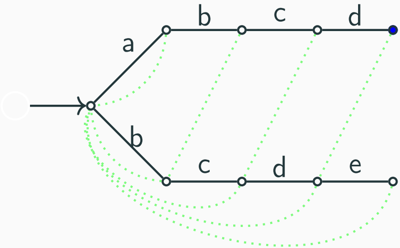


Example



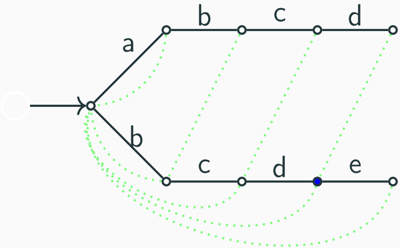
Example

„eabcxab"



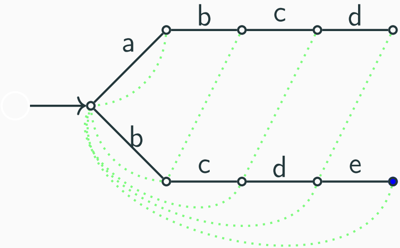
Example

„eabcxab"



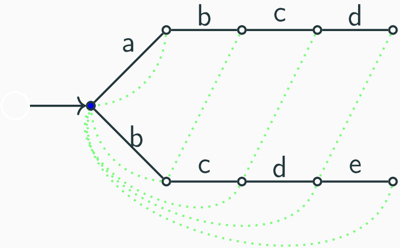
Example

„abcxab"



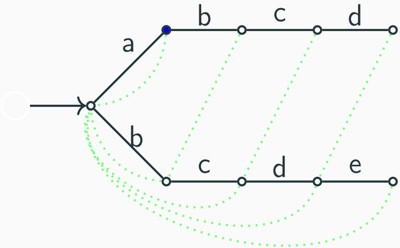
Example

„abcxab"



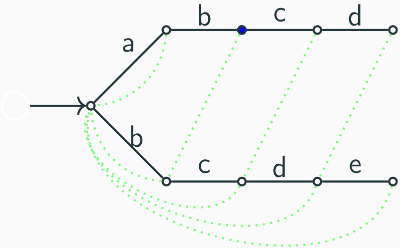
Example

„bcxab"



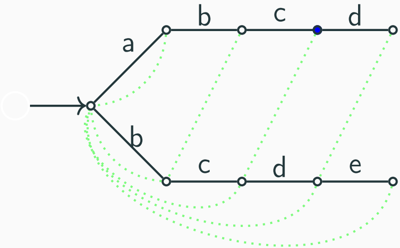
Example

„cxab"



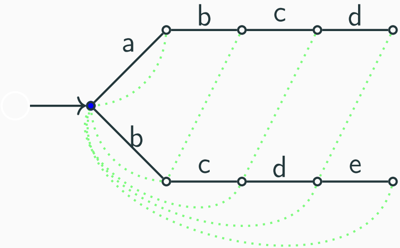
Example

„xab"



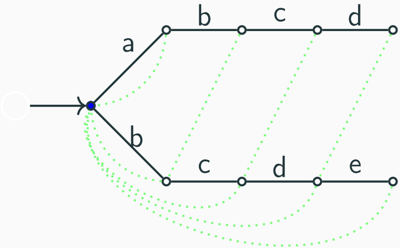
Example

„xab"

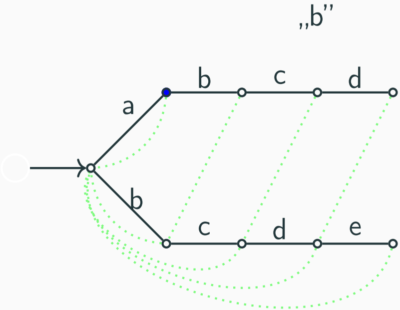


Example

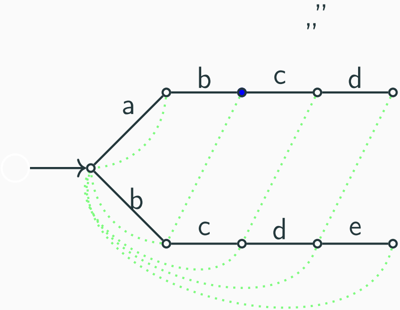
„ab"



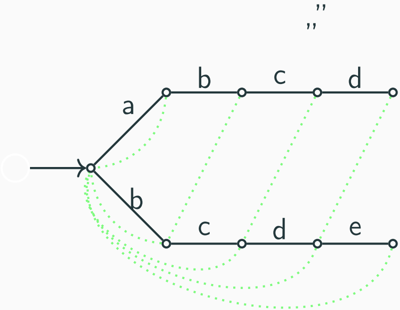
Example



Example



Example



End nodes

End nodes

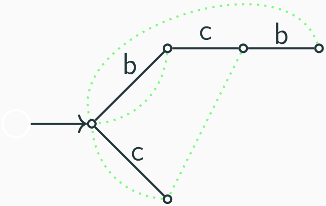
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?

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.

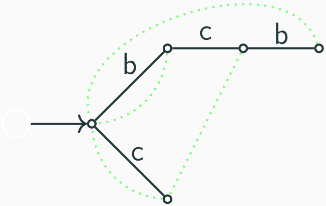
End nodes

- 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

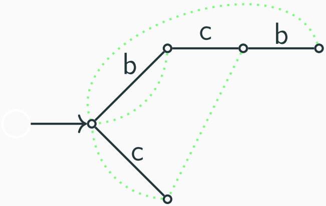
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



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.

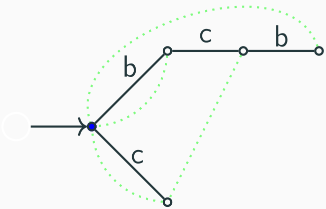
„bcb"



End nodes

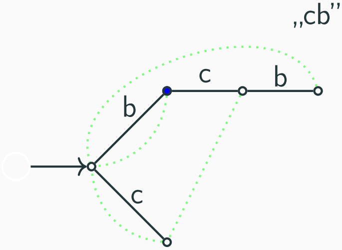
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.

„bcb"



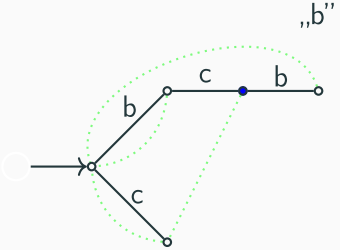
End nodes

- 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.



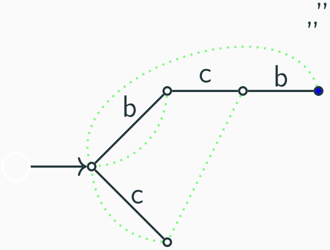
End nodes

- 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.



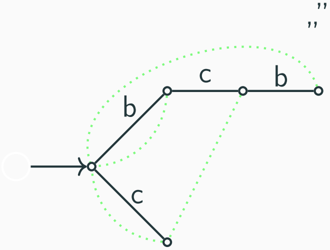
End nodes

- 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.



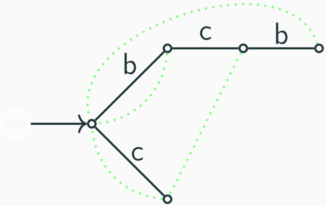
End nodes

- 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

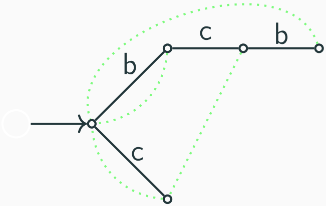
- Thus every time we are at an end node we have a substring in s that is in p. Are these the only ones?
- No, we also need to consider if we can get to end nodes by traveling along suffix links.



- 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

```
constexpr int ALPHABET{ 128 };
// Helper function to get index of letter
constexpr 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;
}
```