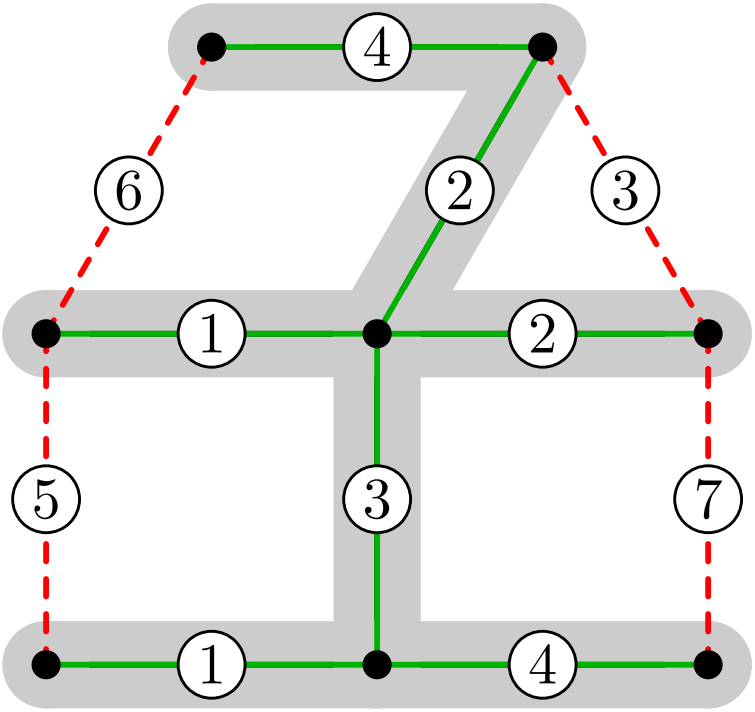
# Kruskal: Minimaler Spannbaum



# Kruskals Algorithmus – Implementierung

## Algorithmus

**function** *Kruskal*$(G, w)$ :
$A := \emptyset$;
**for** *each vertex* $v \in V[G]$ **do**
  *Make_Set*$(v)$
**od**;
*sort the edges of* $E$ *into nondecreasing order by weight*;
**for** *each edge* $\{u, v\} \in E$, *nondecreasingly* **do**
  **if** *Find_Set*$(u) \neq$ *Find_Set*$(v)$ **then**
    $A := A \cup \{\{u, v\}\}$;
    *Union*$(u, v)$
  **fi**
**od**;
**return** $A$

# Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.
Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
$n$ Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

# Union-Find: naiv

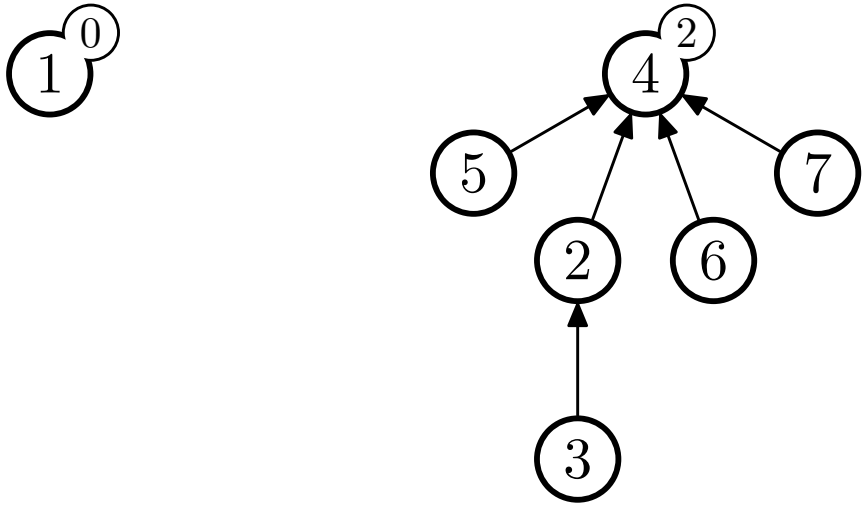
*union*$(a, b)$: Hänge *find*$(a)$ bei *find*$(b)$ ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

# Union-Find: union by rank

*union*$(a, b)$: Verwende die ranghöhere Wurzel

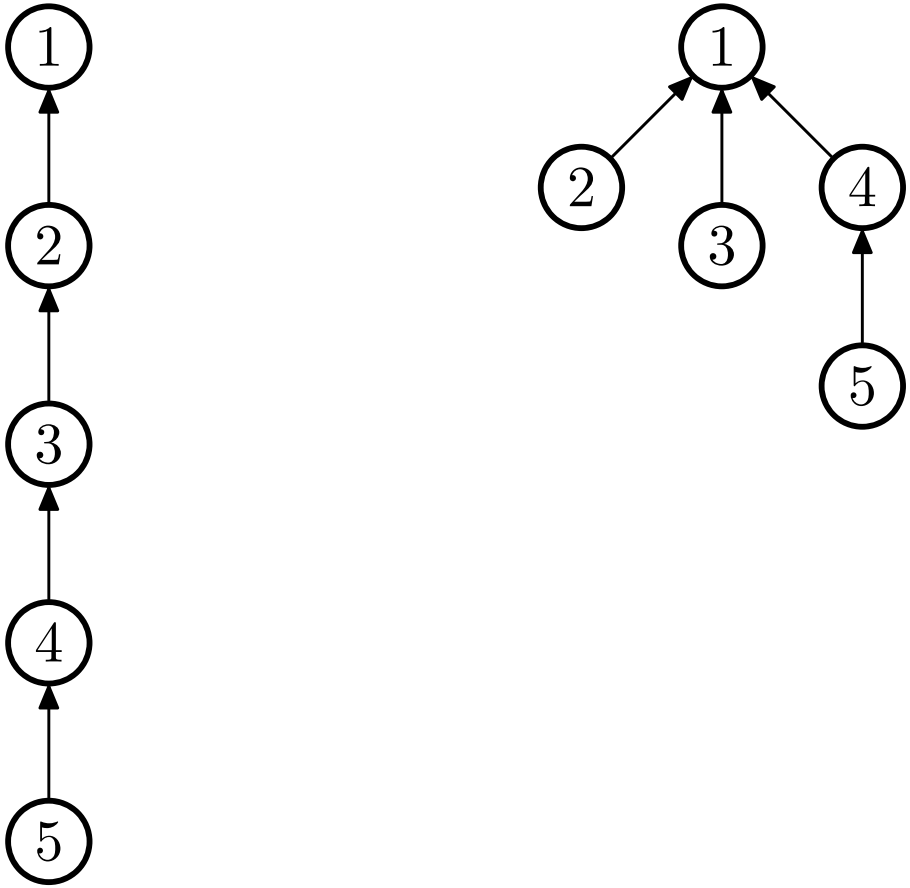
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



# Union-Find: Pfadkompression

*find*(*a*): Komprimiere durchlaufene Pfade

Beispiel: *find*(4)



# Union-Find

## Algorithmus

**procedure** *Make_Set*$(x)$ :
$p[x] := x$;
$rank[x] := 0$

- Wir betrachten jeweils eine Menge $\{0, \ldots, n-1\}$
- Ein Array für die Eltern eines für den Rang
- Ein Baum pro Menge repräsentiert durch die Wurzel
- Wurzel hier kurzgeschlossen

# Union-Find

## Algorithmus

**function** $Find\_Set(x)$ :
**if** $x \neq p[x]$ **then** $p[x] := Find - Set(p[x])$ **fi**;
**return** $p[x]$;

## Algorithmus

**procedure** $Union(x, y)$ :
$x := Find\_Set(x)$;
$y := Find\_Set(y)$;
**if** $rank[x] > rank[y]$ **then** $p[y] := x$
  **else** $p[x] := y$;
    **if** $rank[x] = rank[y]$ **then** $rank[y]{+}{+}$**fi**;
**fi**;

## Java

```
public class Partition {
  int[ ] s;
  public Partition(int n) {
    s = new int[n];
    for(int i = 0; i < n; i++) s[i] = i;
  }
  public int find(int i) {
    int p = i, t;
    while(s[p] ≠ p) p = s[p];
    while(i ≠ p) {t = s[i]; s[i] = p; i = t; }
    return p;
  }
  public void union(int i, int j) {s[find(i)] = find(j); }
}
```

# Union-Find – Analyse

Zunächst keine Pfadkompression.

**Lemma**

*Falls eine Union-Find-Datenstruktur einen Baum mit $m$ Elementen enthält, dann ist seine Höhe höchstens* $\log m + 1$.

**Beweis.**

Wird ein Element neu hinzugefügt, dann ist die Höhe des Baums $1 = \log(1) + 1$.
Werden zwei Bäume zu einem kombiniert, dann ist die Höhe anschließend unverändert, außer die Höhen beider Bäume waren exakt gleich $h$.
Falls die Bäume vorher $k$ und $m$ Elemente enthielten, galt $h \leq \log(k) + 1 \leq \log(m) + 1$.
Daher $h + 1 \leq \log(2m) + 1 \leq \log(k + m) + 1$. □

# Union-Find – Analyse

**Theorem**

*In einer anfangs leeren Union-Find–Datenstruktur mit Rangheuristik werden $m$ Operationen in $O(m \log m)$ Zeit ausgeführt.*

**Beweis.**

- Es gibt stets höchstens $m$ Elemente
- Die Höhe aller Bäume ist durch $\log(m) + 1$ beschränkt
- Union und Find benötigt also nur $O(\log m)$ Zeit

□

# Rang und Pfadkompression

Mittels amortisierter Analyse (Tarjan 1975): $m$ Operationen in $O(m\alpha(m))$ mit
$\alpha(m)$ funktionale Inverse der Ackermannfunktion

Tarjan 1979, Fredman, Saks 1989: Das ist optimal!

Beweis recht kompliziert...

# Algorithmische Geometrie

Probleme der Algorithmischen Geometrie haben üblicherweise diese Eigenschaften:

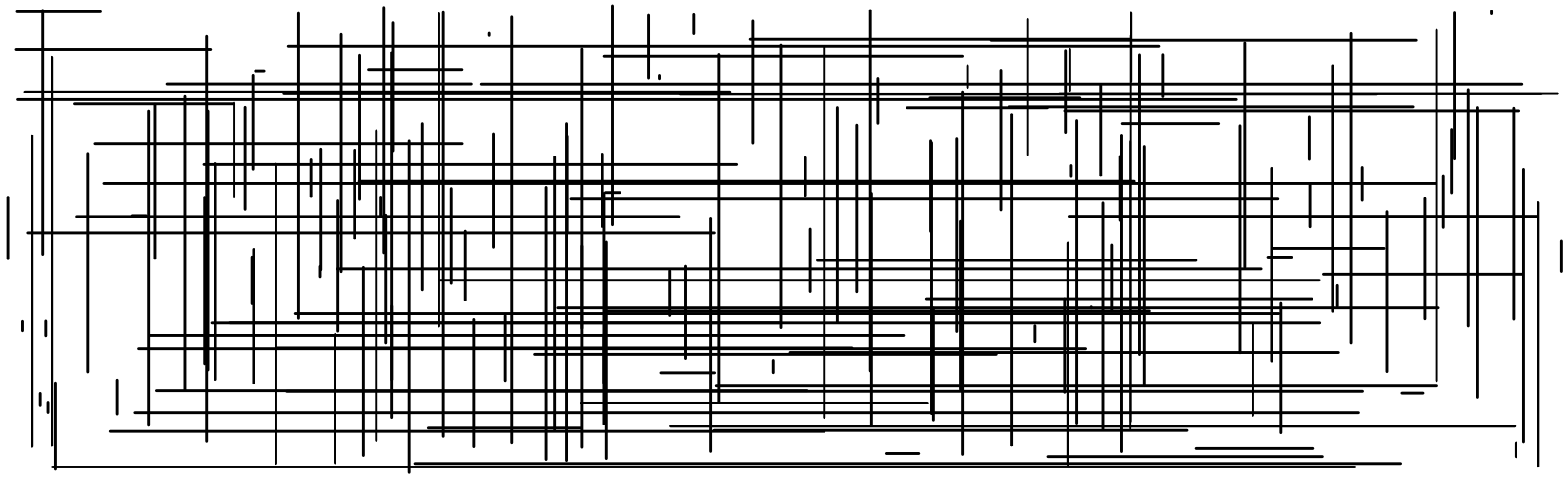
- Die Eingabe besteht aus Punkten, Segmenten, Kreisbögen usw. in der euklidischen Ebene.
- Die Fragestellung ist relativ einfach.
- Sehr große Eingaben müssen bewältigt werden.

Anwendungen beispielsweise im VLSI-Design.

# Schnitte von Segmenten

Eingabe: Horizontale und vertikale Segmente

Ausgabe: Paare von Segmenten, die sich schneiden



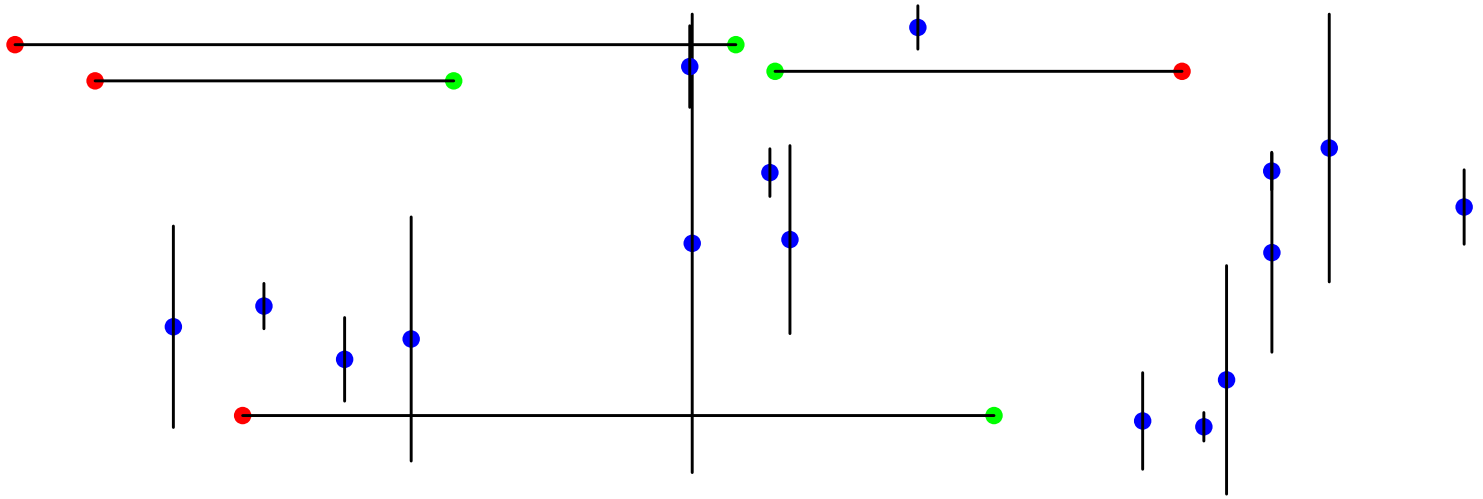
Naiver Algorithmus:

Teste alle Paare, ob sie sich schneiden.

Laufzeit: $\Theta(n^2)$

Variante: Finde heraus, ob es einen Schnitt gibt.

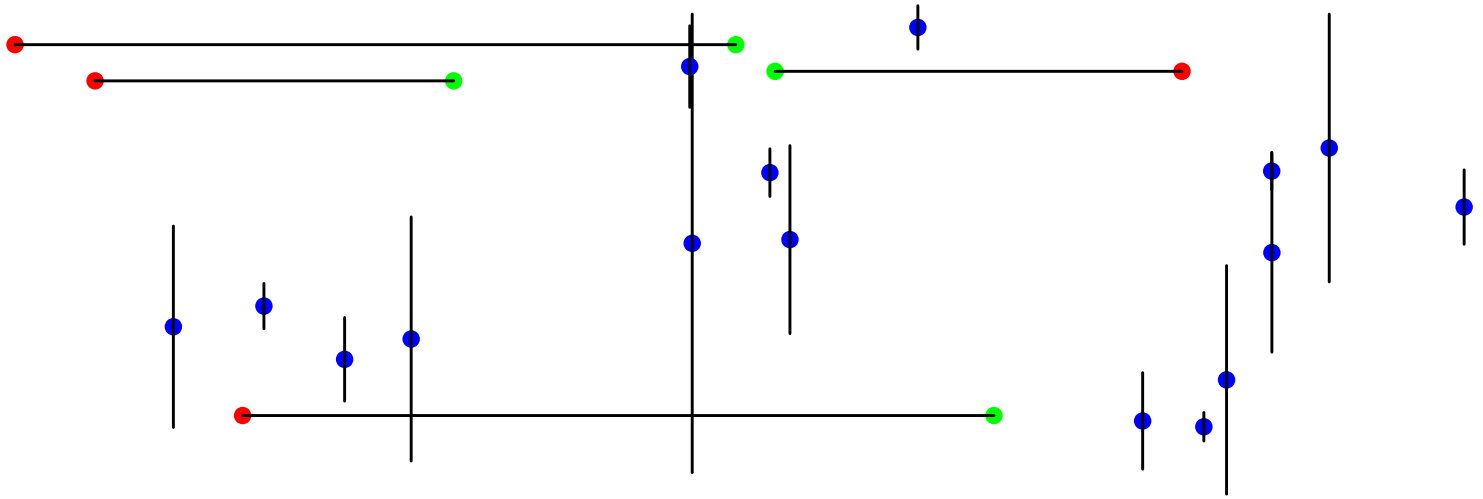
# Sweepline-Algorithmen



- Interessante Punkte: Nach $x$-Koordinate sortieren
- Es gibt ein aktive Menge von Segmenten
- Eine imaginäre vertikale Linie bewegt sich von links nach rechts
- Roter Punkt: Segment in aktive Menge aufnehmen
- Grüner Punkt: Segment aus aktiver Menge entfernen
- Blauer Punkt: Segment mit aktiver Menge vergleichen

Suche nur Schnitte zwischen horizontalem und vertikalem Segment.

# Sweepline-Algorithmen



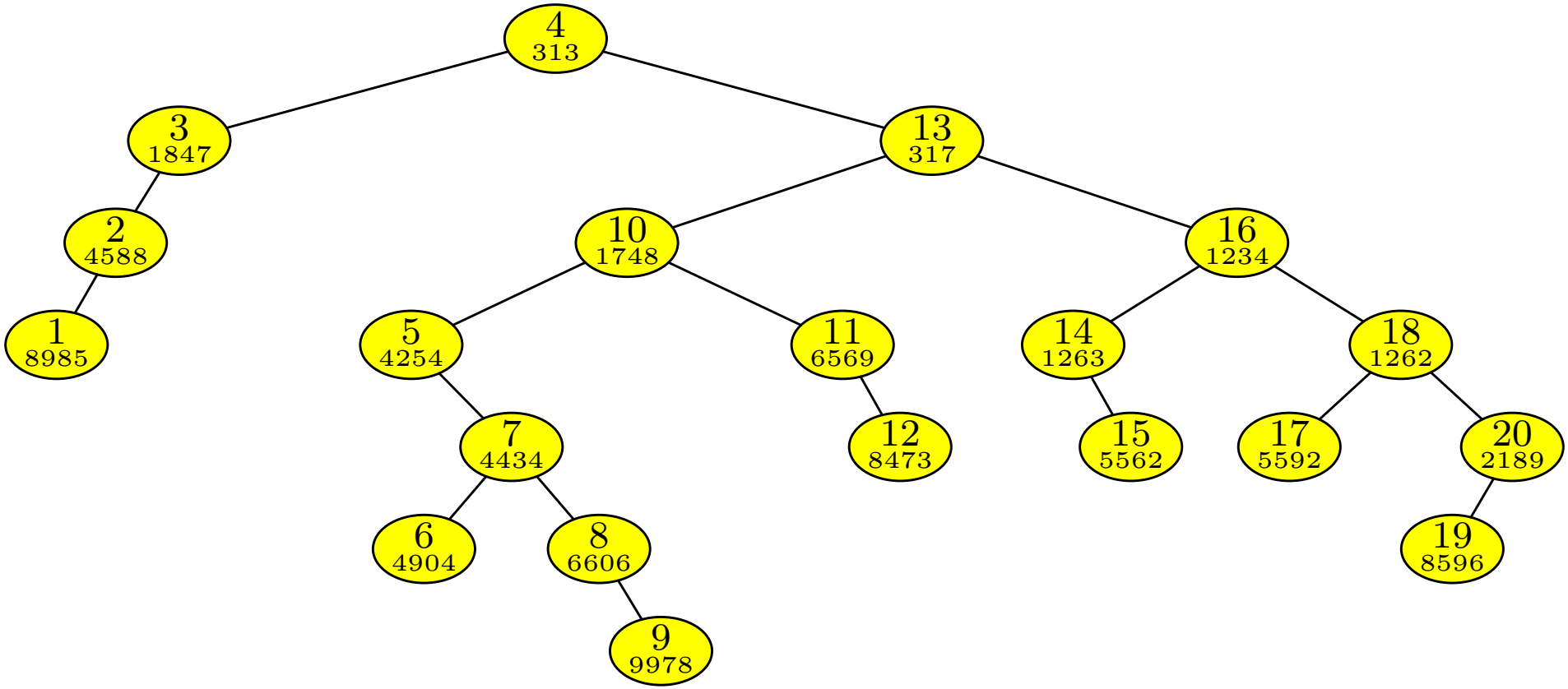
- Wie finden wir bei einem vertikalen Segment die geschnittenen horizontalen Segmente?
- Welche Datenstruktur für die aktive Menge?

Lösung: Speicher $y$-Koordinaten $Y$ der aktiven Menge in balanziertem Suchbaum.

Gibt es einen Schnitt? $\rightarrow O(\log |Y|)$ Schritte

Alle Schnitte $S$ ausgeben: $O(\log |Y| + |S|)$ Schritte

In aktive Menge einfügen oder löschen: $O(\log |Y|)$ Schritte




Aufgabe:
Finde alle $y$-Koordinaten zwischen $a$ und $b$.

Laufzeit: $O(\log |Y| + |S|)$

1. Finde größtes Element, daß kleiner oder gleich $a$ ist.
2. Gehe von dort aus Element aufsteigend durch
3. Beende, wenn $b$ überschritten.