Velkomin á Ísland

Ein Ort unendlich$_1$ weit weg vom Rest der Welt.

[1] fast

Zwei Probleme

Island hat keine Süßigkeiten.

Der Rest der Welt hat kein Eis.

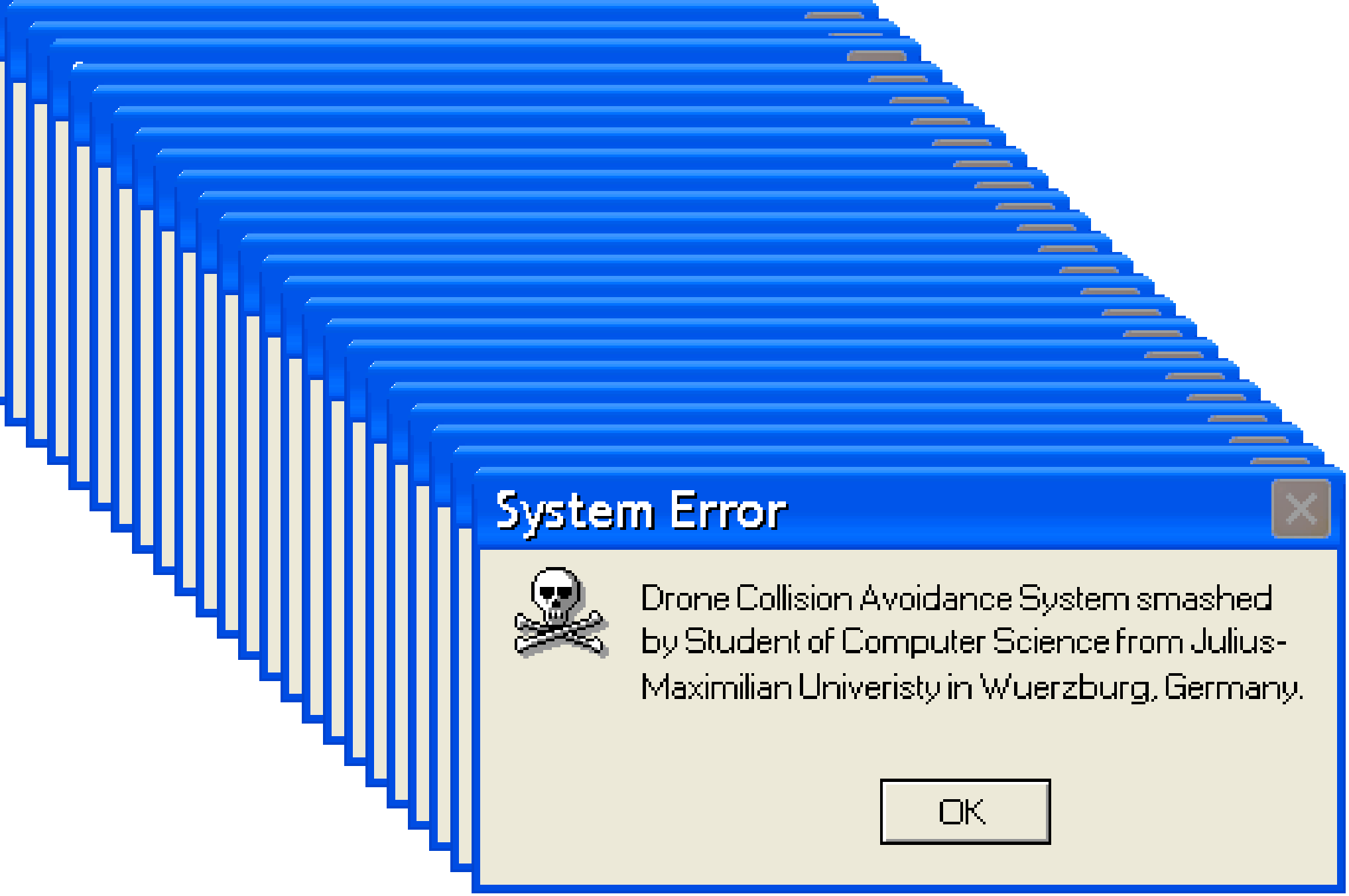
Eine Lösung

Es gibt nur einen Weg zwischen Island und dem Rest der Welt

→ Der Bifrost$_1$

- Drohnen transportieren auf diesem Süßigkeiten nach Island und im Gegenzug Eis zurück in den Rest der Welt.
- Sich entgegenkommende Drohnen weichen einander geschickt aus.

[1] hier schwierig erkennbar: Eine unendliche lange Linie


System Error
Drone Collision Avoidance System smashed by Student of Computer Science from Julius-Maximilian Univeristy in Wuerzburg, Germany.
OK

Die Zeit gefriert$_1$

Es liegt nun an Dir junge*r Informatik-Student*in, das Land zu retten und herauszufinden, welche Süßigkeiten ihr Ziel$_2$ erreichen werden. Ruhm und Eis werden Dich erwarten, solltest du erfolgreich nach Hause zurückkehren$_3$.

[1] zumindest in Island
[2] unendlich weit in der Zukunft
[3] Die Alternativstrategie Zuhausebleiben zählt nicht!

Das Setting

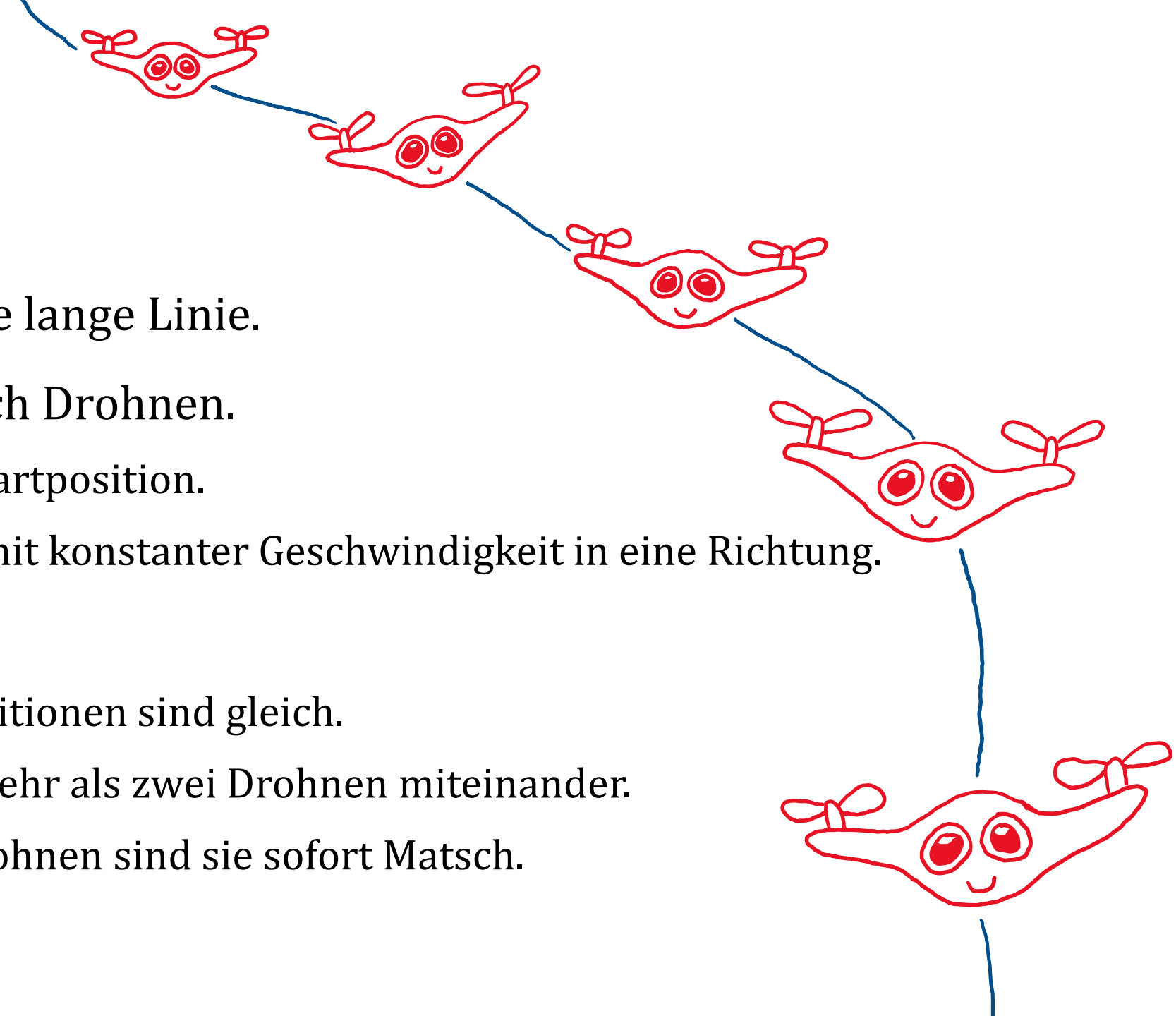
Es gibt eine unendliche lange Linie.

Auf dieser befinden sich Drohnen.

- Diese haben eine Startposition.
- Und bewegen sich mit konstanter Geschwindigkeit in eine Richtung.

Einschränkungen:

- Keine zwei Startpositionen sind gleich.
- Es kollidieren nie mehr als zwei Drohnen miteinander.
- Kollidieren zwei Drohnen sind sie sofort Matsch.



Die Aufgabe

Kommen sich zwei Drohnen entgegen kollidieren sie.
Das gleiche gilt, wenn eine Drohne eine andere einholt.

→ Finde heraus, welche Drohnen überleben!

Input

Ein Zeile mit der Anzahl der Drohnen n

- $1 \leq n \leq 10^5$

n Zeilen mit je zwei Zahlen[1] x_i und v_i

- $-10^9 \leq x_i, v_i \leq 10^9$
- x_i ist die Startposition der Drohne auf der Linie.
- v_i ist die Geschwindigkeit der Drohne. (das Vorzeichen die Richtung)

Die Drohnen sind bereits in aufsteigender Startposition sortiert.

[1] getrennt durch Leerzeichen

Output

Eine Zeile mit der Anzahl der Drohnen, die niemals crashen.

Eine Zeile mit allen Nummern[1] der Drohnen, die niemals crashen.

- Die Drohnen-Nummer ist ihr Index *i* aus der Eingabe

[1] getrennt durch Leerzeichen

```
4
10 -30
30 20
50 0
90 -10
```

```
2
1 4
```

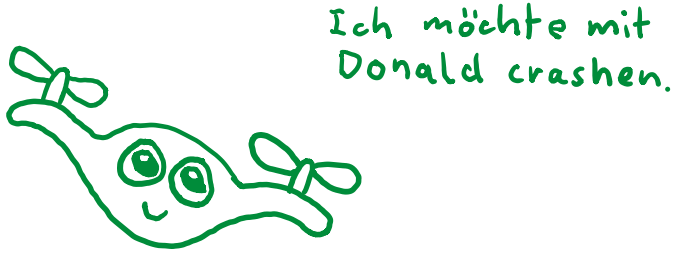
Wie viel Zeit haben wir?

Es gibt maximal 10^5 Drohnen.

→ Das riecht nach $O(n \log n)$

→ Greedy für jedes Paar von Drohnen einen Kollisionszeitpunkt zu berechnen, braucht also zu lange.

Wie geht's schneller?



Kollidieren alle Drohnen, gibt es $\frac{n}{2}$ Crashs.

Welche Drohnen können crashen?

→ Nur Nachbarn!

Nachbarschaften:

- Anfangs gibt es $n - 1$
- Nach jedem Crash gibt es eine neue[1]

[1] und drei alte fallen weg

Und was geht daran schneller?

Angenommen alle Drohnen crashen

→ Dann gibt es insgesamt (maximal)

$n - 1$ vom Anfang

$+ \frac{n}{2}$ maximale Anzahl an Crashs.

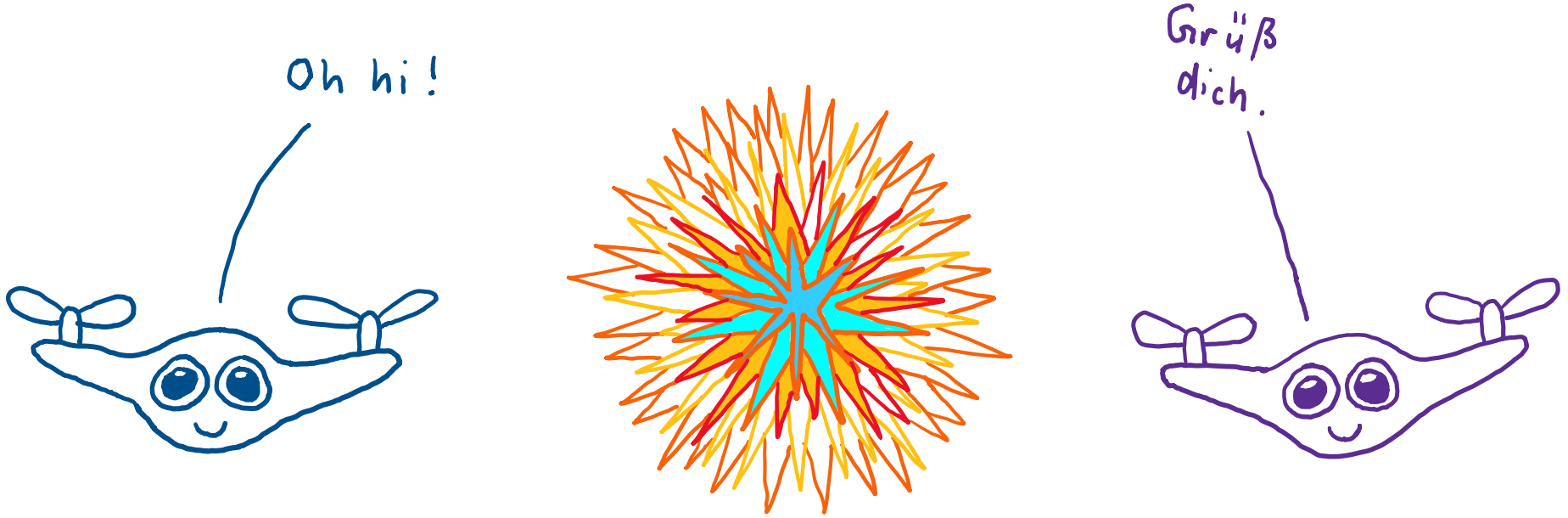
Nachbarschaften.

→ Eine lineare Anzahl an Nachbarschaften, spitze!

... und am Anfang sind die Drohnen bereits sortiert, noch spitzer!

Welche Drohnen sind Nachbarn?

1. Alle Drohnen, die am Anfang Nachbarn sind.
2. Alle Drohnen zwischen denen irgendwann genau zwei Drohnen liegen, die kollidieren.



Welche Drohnen werden Nachbarn sein?

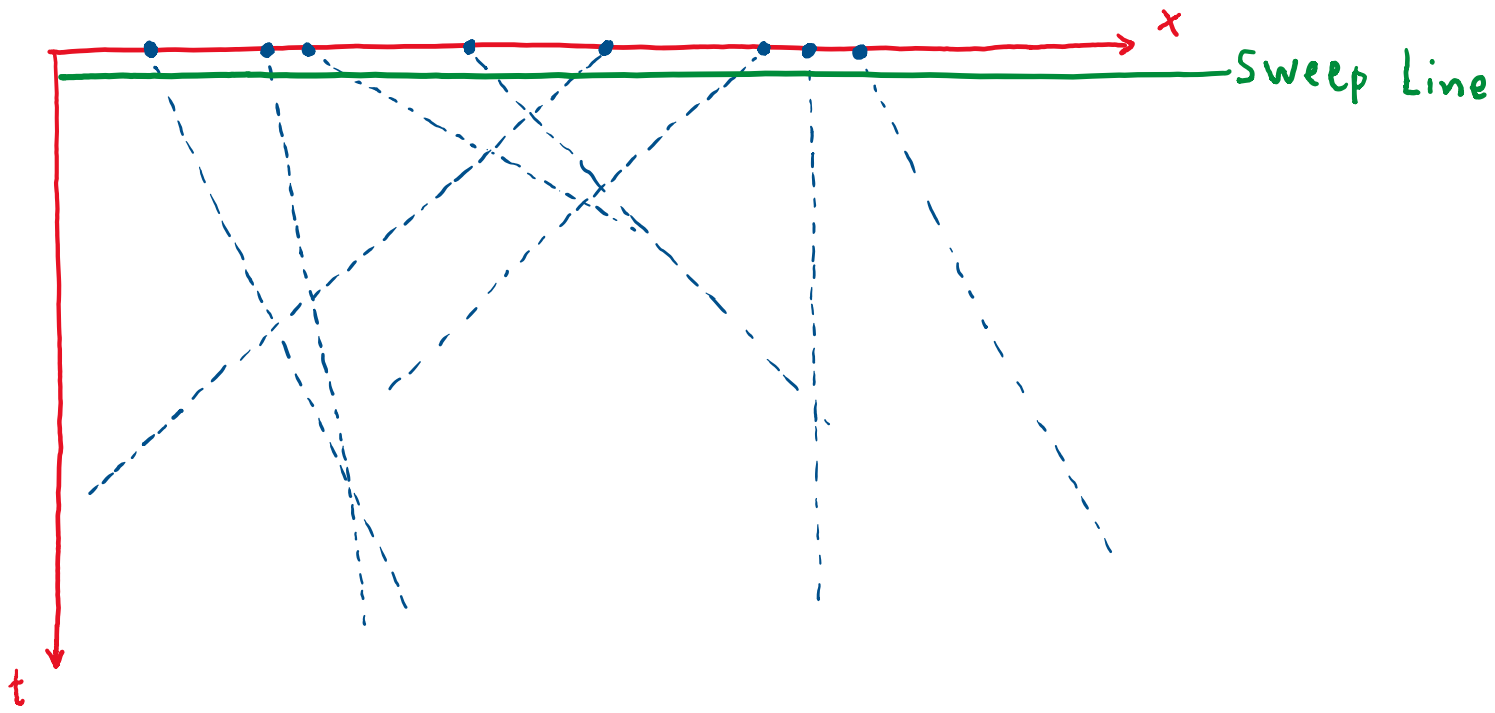
Am Anfang sind also nicht alle Nachbarschaften bekannt.

Aber: Zu jedem Zeitpunkt kennen wir eine Nachbarschaft, die auf jeden Fall zu einer Kollision führen wird.

→ Aus dieser entsteht eine neue Nachbarschaft,
also eine neue potentielle Kollision.

Sweep Line

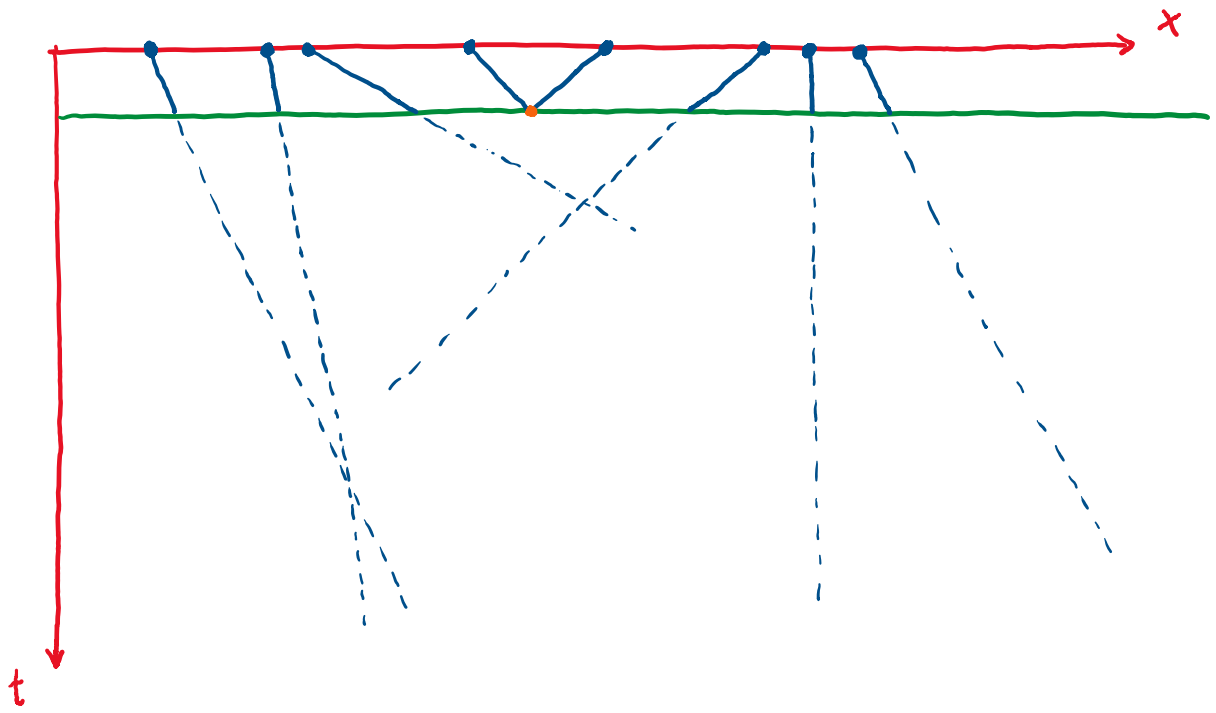
Wir brauchen einen Sweep Line Algorithmus[1].



[1] mehr dazu auf Wikipedia

Sweep Line

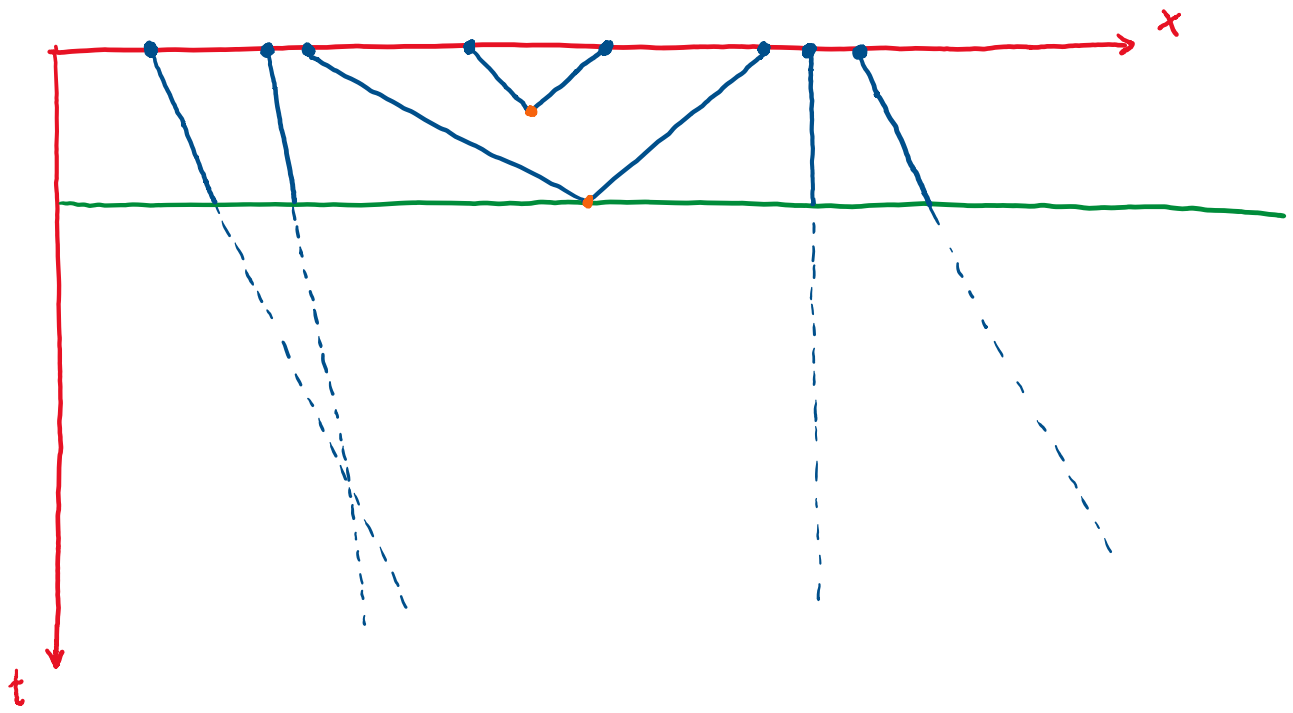
Wir brauchen einen Sweep Line Algorithmus[1].



[1] mehr dazu auf Wikipedia

Sweep Line

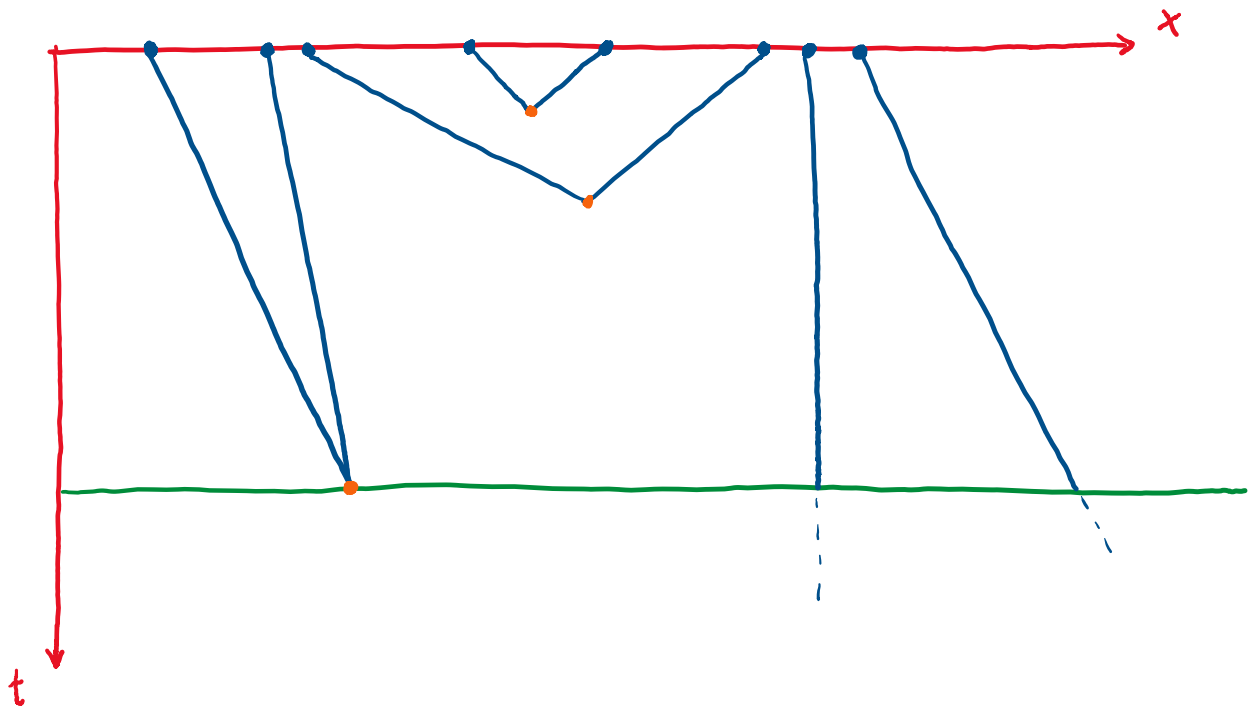
Wir brauchen einen Sweep Line Algorithmus[1].



[1] mehr dazu auf Wikipedia

Sweep Line

Wir brauchen einen Sweep Line Algorithmus[1].



[1] mehr dazu auf Wikipedia

Wann kollidieren zwei Nachbarn?

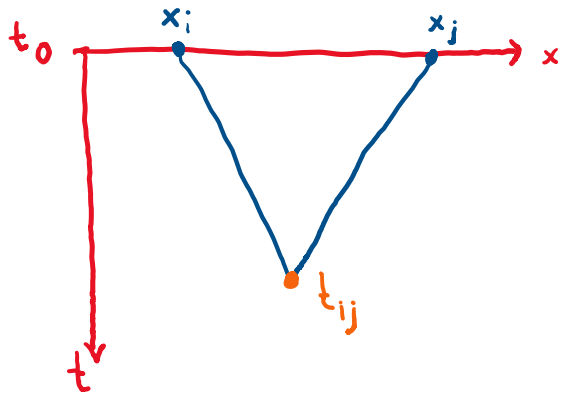
Betrachte die Position einer Drohne zum Zeitpunkt t als Funktion:

$$f_i(t) = x_i + t \cdot v_i$$

Der Kollisionszeitpunkt zweier Drohnen ist der Schnittpunkt ihrer Funktionen. Da die Funktionen linear sind, ist dieser eindeutig[1].

$$x_i + t \cdot v_i = x_j + t \cdot v_j \quad | - x_j$$
$$x_i - x_j + t \cdot v_i = t \cdot v_j \quad | - t \cdot v_i$$
$$x_i - x_j = t \cdot v_j - t \cdot v_i$$
$$x_i - x_j = t \cdot (v_j - v_i) \quad | : (v_j - v_i)$$

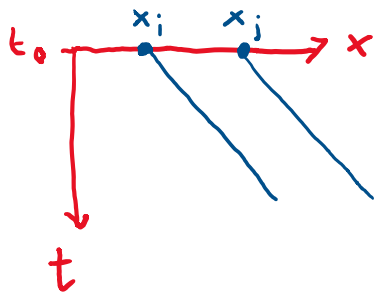
$$t_{ij} = \frac{x_i - x_j}{v_j - v_i}$$



[1] fast

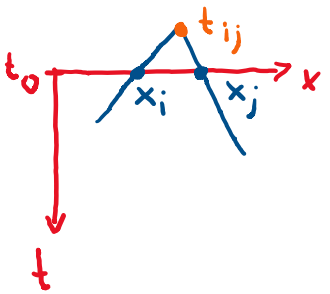
$$t_{ij} = \frac{x_i - x_j}{v_j - v_i}$$

Und welche Nachbarn kollidieren?



Vorsicht! $(v_j - v_i)$ darf nicht 0 sein!

→ Klar, zwei gleich schnelle Drohnen kollidieren nicht.



Ein negativer Zeitpunkt liegt in der Vergangenheit.

→ Drohnen mit negativem Kollisionszeitpunkt kollidieren nicht.

→ Die nächste Kollision ist also die mit dem kleinsten positiven Zeitpunkt zweier nicht gleich schneller Drohnen.

Effizient die nächste Kollision ermitteln

Folgende Methoden müssen wir häufig ausführen: häufig heißt hier: „O(n) mal"

- Einfügen neuer Kollisionen
- Extrahieren der nächsten Kollision (kleinster positiver Zeitpunkt)
- Optional: Löschen von *überholten* Kollisionen

→ Das stinkt nach Min-Heap!

Im Sweep-Line-Slang wird dieser als Event-Point-Queue bezeichnet.

Nochmal kurz und knackig

1. Für alle unterschiedlich schnellen Nachbarn
   - Kollisionszeitpunkt t berechnen
   - Nachbarschaften mit positivem t in den Min-Heap einfügen
2. Solange der Min-Heap nicht leer ist
   - Die nächste potentielle Kollision herausziehen.
   - Falls beide Drohnen noch leben → Drohnen killen! 
     - Die Nachbarn der Kollision verknüpfen und ihren Kollisionszeitpunkt t berechnen.
     - Falls t positiv ist, die neue Nachbarschaft in den Min-Heap einfügen
3. Alle überlebenden Drohnen ausgeben

Laufzeit

Zur Initialisierung müssen wir $n - 1$ Nachbarschaften einfügen.

Am teuersten wird es, wenn alle Drohnen kollidieren[1].

- Dann gibt es $\frac{n}{2}$ Kollisionen
- Für jede müssen wir eine neue Nachbarschaft einfügen.

→ Macht $n - 1 + \frac{n}{2} < \frac{3}{2}n$ Einfüge-Operationen von je $O(\log n)$

→ Also eine Laufzeit von $O(n \log n)$

[1] eine ungerade Anzahl an Drohnen können wir für die Laufzeit ignorieren

$$t_{ij} = \frac{x_i - x_j}{v_j - v_i}$$

Kollisionszeitpunkt

Der Kollisionszeitpunkt ist eine rationale Zahl[1].

→ Je nach Datentyp können Ungenauigkeiten beim Runden zu einem fehlerhaften Programm führen.

Die sind fies und testen das!

[1] bzw. kann unendlich viele Nachkommastellen haben

$$t_{ij} = \frac{x_i - x_j}{v_j - v_i}$$

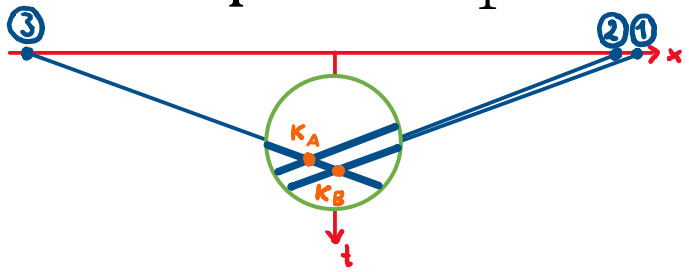
Lösung 1 – Präzision

Auf welche Nachkommastelle dürfen wir runden?

- x und v sind beschränkt durch -10^9 und 10^9

→ Eine untere Schranke für den Unterschied von Kollisionszeitpunkten[1]

$$\Delta t_{min} = \frac{2 \cdot 10^9 - 1}{2 \cdot 10^9} - \frac{2 \cdot 10^9 - 2}{2 \cdot 10^9 - 1} > \frac{2{,}5}{10^{19}}$$



→ Mit einem Datentyp, der 19 dezimale Nachkommastellen drauf hat, bist du also auf der sicheren Seite.

[1] – die nicht gleichzeitig sind –

$$t_{ij} = \frac{x_i - x_j}{v_j - v_i}$$

Lösung 2 – Brüche

Wir haben in Java eine eigene Bruch-Implementierung[1] verwendet:

- Zwei Attribute speichern Zähler und Nenner.
- Die Klasse implementiert Comparable.
- In der compareTo Methode muss dann nur multipliziert werden.

$$t_1 = \frac{\Delta x_1}{\Delta v_1}$$

$$t_2 = \frac{\Delta x_2}{\Delta v_2}$$

$$t_1 - t_2 = \frac{\Delta x_1}{\Delta v_1} - \frac{\Delta x_2}{\Delta v_2} = \frac{\Delta x_1 \cdot \Delta v_2 - \Delta x_2 \cdot \Delta v_1}{\cancel{\Delta v_1 \cdot \Delta v_2}}$$

Vorsicht mit dem Vorzeichen!

→ Vorsicht mit Vorzeichen, sowie dem Wertebereich der Datentypen!

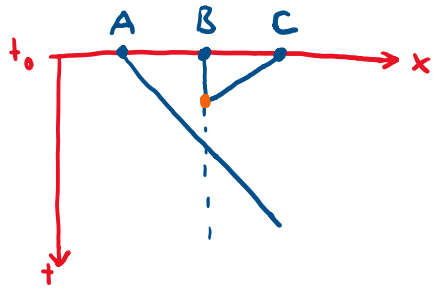
[1] viel smarter!

Min-Heap

Wir haben für den Min-Heap eine Java PriorityQueue genutzt.

- increase- und decreaseKey werden eh nicht benötigt

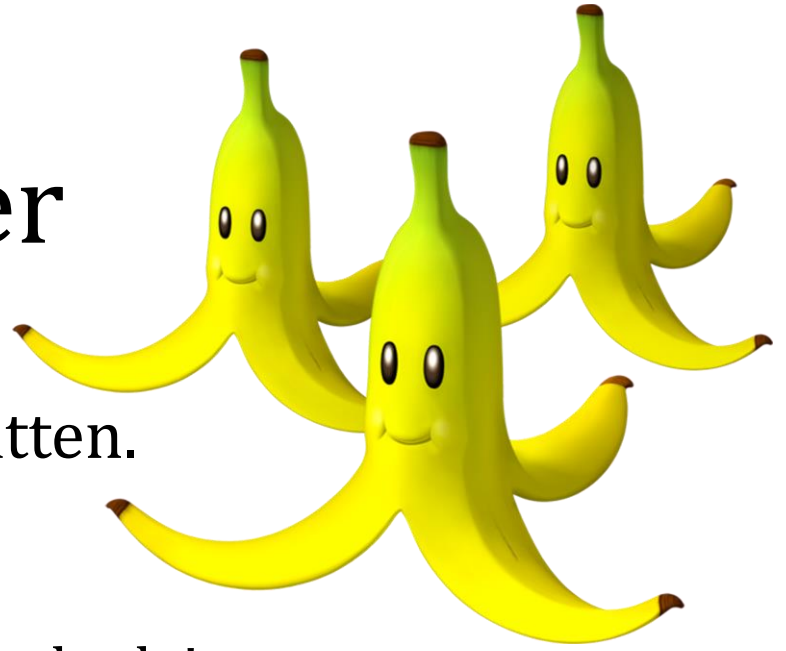
Zuvor zum Heap hinzugefügte Kollisionen können ihre Gültigkeit verlieren, wenn eine der Drohnen zuvor mit einer anderen gecrasht ist.



Kollisionen aktiv zu entfernen ist eher teuer.

→ Geschickter: Beim Herausnehmen überprüfen, gegebenenfalls ignorieren

Macht lieber eure eigenen Fehler



Auf DomJudge seht ihr, dass wir zwei Fehlversuche hatten.

1. Upsi, Max- statt Min-Heap gebastelt.
   - Mit Links-Rechts-Schwäche: Vorsicht bei compareTo-Methode!
2. Bei folgenden Sonderfällen hatten wir den Output vercheckt$_1$:
   - Wenn keine Drohne kollidiert.
   - Wenn die letzte Kollision im Heap nicht mehr gültig ist.

[1] reine Eitelkeit

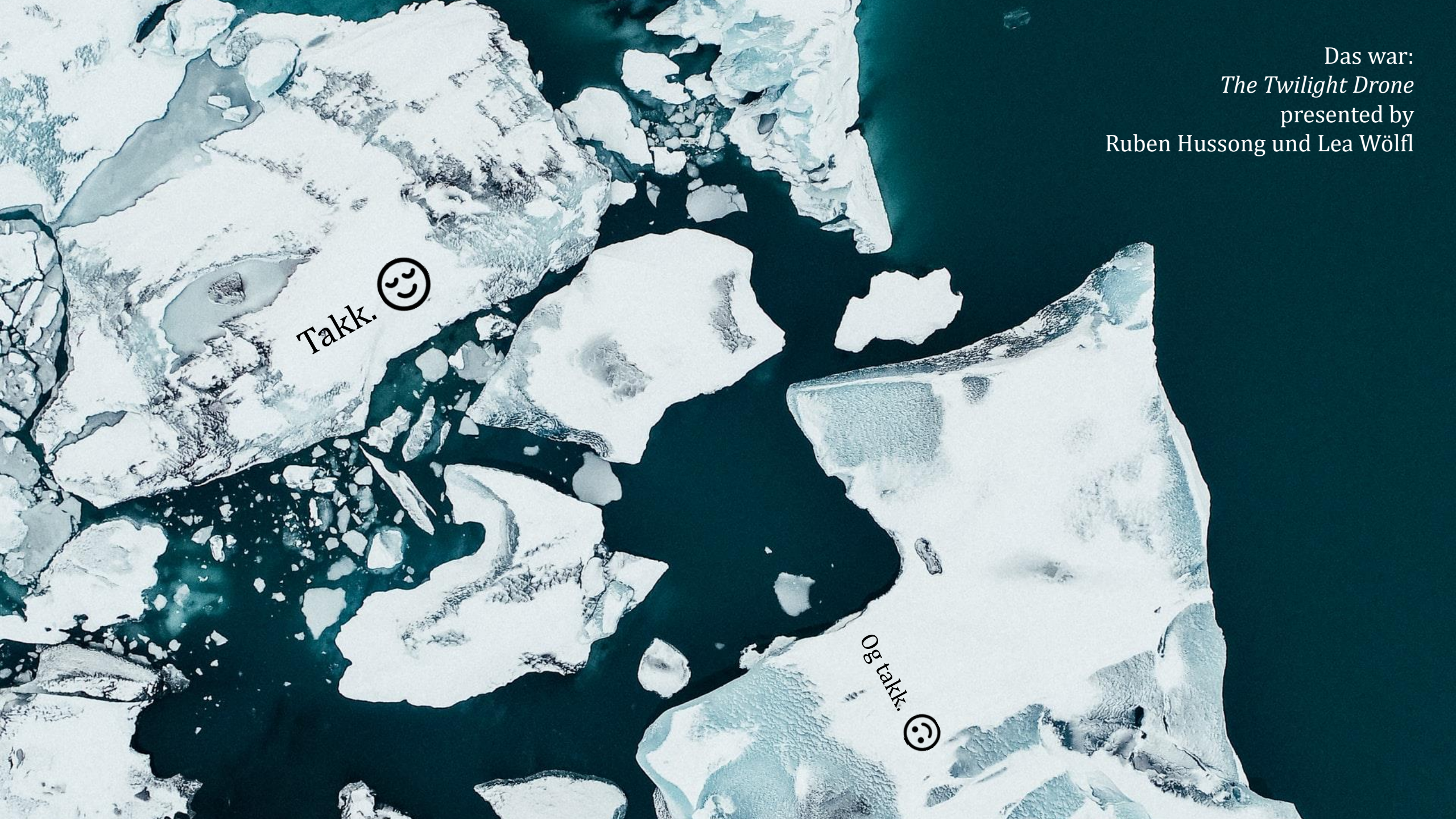
Tests

Auf WueCampus findet ihr noch ein paar Schnelltests.

Auf folgende Extremfälle zielen die ab:

1. Max-Heap Min-Heap Verwechsler
2. Laufzeit (große Eingabe, viele Kollisionen)
3. Keine Kollisionen
4. Präzisions-Test machen

Wenn ihr fertig seid: Kauft euch ein Eis.



Das war:
The Twilight Drone
presented by
Ruben Hussong und Lea Wölfl