Zweite Runde Theorie

Aufgaben

Swiss Olympiad in Informatics

3. März 2018

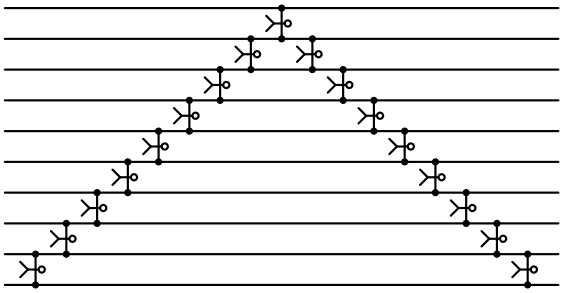
Sushi Belts

Teilaufgabe 1: Serviere alle Mahlzeiten an alle Gäste (10 Punkte)

Die Zeichnung im nächsten Abschnitt ist eine der möglichen Lösungen für diese Teilaufgabe.

Teilaufgabe 2: Minimale Anzahl Angestellte (30 Punkte)

Wir benötigen mindestens $2b - 3$ Angestellte um jedes Gericht zu jedem Tisch im Restaurant zu servieren. Für $b = 10$ ist ein mögliche Platzierung von Angestellten unten dargestellt.



Die Idee hinter dieser Anordnung ist die folgende:

- Die ersten $b - 2$ Angestellten können Gericht von jedem beliebigen Fliessband zu Fliessband 1 befördern.
- Der Mitarbeiter in der Mitte kann ein Gericht von Band 1 zu Band 0 bringen.

Nach den ersten $b-1$ Mitarbeitern kann jedes Gericht jedes Fliessband mit einer kleineren Nummer erreichen.
Bemerke auserdem, dass:

- Der Mitarbeiter in der Mitte kann ein Gericht von Fliessband 0 zu Fliessband 1 bringen.
- Die hinteren $b-2$ Angestellten kann ein Gericht von Fliessband 1 zu jedem anderen Fliessband befördern.

Nach den letzten $b - 1$ Angestellten kann demzufolge jedes Gericht ein Fliessband mit einer höheren Nummer erreichen.

Wir wissen nun, dass es immer möglich ist, mit $2b - 3$ Mitarbeitenden alle Gäste zu bedienen. Es bleibt zu zeigen, dass dies die minimale Anzahl ist, also dass wir bei $b > 2$ Fliessbändern genau $2b - 3$ Mitarbeitende benötigen.

Dazu führen wir einen Widerspruchsbeweis und nehme an, es geht mit maximal $2b-4$ Angestellten. Dies möchten wir zu einem Widerspruch führen.

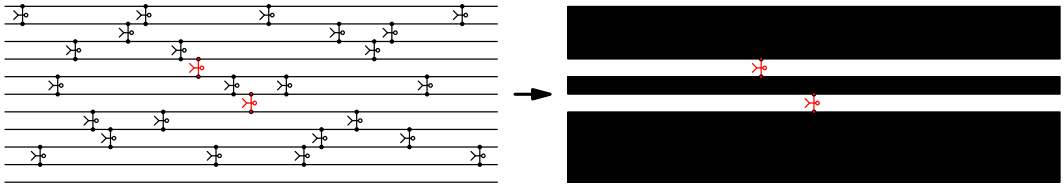
Da es b Fliessbänder gibt, hat es $b-1$ Reihen, in denen ein Mitarbeiter platziert werden kann. In einer Zeile kann offensich nur jemanden haben. Da wir aber höchstens $2b-4$ Mitarbeiter zur Verfügung haben, gibt es mindestens zwei Reihen, in denen sich nur ein Mitarbeiter befindet. Wir nennen diese Mitarbeiter "Solo-Mitarbeiter".

Definiere i und j so, dass sich der einer dieser Solo-Mitarbeitern zwischen Fliessbändern i und $i+1$ an Koordinate x_i befindet und ein anderer Solo-Mitarbeiter zwischen Fliessbändern j und $j+1$ an Koordinate x_j. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass $i < j$ und $x_i < x_j$.

Die Solo-Mitarbeiter teilen die Fliessbänder in mindestens drei nichtleere Gruppen auf:

1. Fliessbänder nördlicher der beiden Solo-Mitarbeitern (0 bis i)
2. Fliessbänder zwischen den beiden Solo-Mitarbeitern ($i+1$ bis j)
3. Fliessbänder südlich der beiden Solo-Mitarbeitern ($j+1$ bis $b-1$)

Dies ist im folgenden Bild dargestellt:



Wir stellen fest, dass das Gericht von Fliessband $b-1$ nicht an Tisch 0 gelangen kann. Die einzige Möglichkeit um die zweite Gruppe von Fliessbändern zu erreichen ist über den zweiten Solo-Mitarbeiter zu gehen. Aber dann sind wir schon zu weit östlich und sind schon hinter dem ersten Solo-Mitarbeiter der als einziger fähig wäre, ein Gericht von der zweiten Gruppe in die erste Gruppe zu bringen.

Damit haben wir einen Widerspruch gefunden, weshalb es nicht möglich sein kann, eine gültige Anordnung mit maximal $2b-4$ Mitarbeitern zu finden.

Teilaufgabe 3: Suboptimale Angestellte (60 Punkte)

Zuerst machen wir folgende Beobachtung: Auf jeder Position auf jedem Fliessband ist die Liste der erreichbaren Gerichte ein Intervall. Der Beweis dazu ist einfach: Wenn ein Gericht, welches auf Fliessband i startet nach Fliessband i gelangen kann, dann kann auch jedes Gericht, welches zwischen i und j startet nach j gelangen, in dem es zunächst wartet und dann den Weg kopiert, der von Gericht i genommen wurde.

Anstatt dass wir eine Menge von Gerichten für jedes Fliessband merken müssen (was bis zu n Zahlen pro Fliessband wären), reichen uns zwei Werte: die kleinste und die grösste Nummer der möglichen Gerichte, die das Ende erreichen können.

Wir machen einen Scanline-Ansatz, d.h. wir gehen die Mitarbeitenden in aufsteigender x-Koordinate durch (von West nach Ost). Ohne Beschränkung der Allgemeinheit nehmen wir an, dass die Mitarbeitenden bereits sortiert sind: Aus $i < j$ folgt $x_i < x_j$. Während der Scanline merken wir uns für Fliessband i folgende zwei Zahlen:

- Die kleinste Nummer $L_{i,j}$, von der ein Gericht über die ersten j Mitarbeitenden nach Fliessband i gelangen kann.
- Die grösste Nummer $H_{i,j}$, von der ein Gericht über die ersten j Mitarbeitenden nach Fliessband i gelangen kann.

Zu Beginn kann nur Gericht i nach Fliessband i gelangen, deshalb gilt $L_{i,0} := i$ und $H_{i,0} := i$.

Nun zum Scanline-Schritt. Angenommen, wir haben bereits die ersten j Mitarbeitenden bearbeitet (Nummern 0 bis $j-1$) und wir kennen die Werte $L_{i,j}$ und $H_{i,j}$ für dieses j j und alle i. Als nächstes bearbeiten wir Mitarbeiter Nummer j. Fast alle Werte $L_{*,j+1}$ und $H_{*,j+1}$ sind die gleichen wie die entsprechenden Werte in $L_{*,j}$ und $H_{*,j}$, es ändern sich nur zwei. Angenommen, der Mitarbeiter bewegt Gerichte zwischen Fliessbändern y und $y+1$. Die einzigen Intervalle von Gerichten, die sich ändern können, sind von diesen beiden Fliessbändern. Alle anderen Intervalle bleiben gleich. Ausserdem werden sich nur zwei der vier Werte dieser beiden Fliessbänder ändern: Es können etwas mehr südliche Gerichte nach Fliessband y und etwas mehr nördliche Gerichte nach Fliessband $y+1$ gelangen. Die neuen Werte von L und H berechnen sich deshalb wiefolgt:

$$\begin{aligned} L_{y+1,j+1} &:= L_{y,j} \\ L_{i,j+1} &:= L_{i,j} \qquad \forall i \neq y+1 \\ H_{y,j+1} &:= H_{y+1,j} \\ H_{i,j+1} &:= H_{i,j} \qquad \forall i \neq y \end{aligned}$$

Nachdem alle Mitarbeitenden bearbeitet sind, also die Scanline fertig ist, können wir die Ausgabe leicht berechnen: $m_i := H_{i,w} - L_{i,w} + 1$.

Um die Lösung abzuschliessen, bemerken wir, dass wir die Werte L und H in-place updaten können – d.h. anstatt explizit die Werte von $L_{*,y}$ nach $L_{*,y+1}$ zu sortieren reicht es, ein einziges Array L zu haben und für jeden Mitarbeitenden ein einzelnes Update in konstanter Zeit durchzuführen. (Wir benötigen die alten Werte nicht mehr, also können wir sie einfach überschreiben.)

EIne mögliche Lösung, welche volle Punktzahl erhalten würde, sieht so aus:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int B, W;
  cin >> B >> W;    // counts of Belts and Waiters

  // read waiters from the input
  vector<pair<int,int> > waiters(W);
  for(int i=0; i<W; ++i) {
```

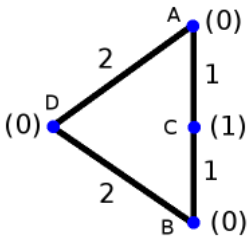
```
14      cin >> waiters[i].first >> waiters[i].second; // X and Y coordinates
15      --waiters[i].second;  // The belts are numbered from 0.
16    }
17    sort(waiters.begin(), waiters.end());
18
19    // initialization
20    vector<int> highest, lowest;
21    for(int i=0; i<B; ++i) {
22      highest.push_back(i);   // highest[i] = max meal that can reach belt i
23      lowest.push_back(i);    // lowest[i]  = min meal that can reach belt i
24    }
25
26    for(int i=0; i<W; ++i) {
27      int y = waiters[i].second;  // Y coordinate
28      highest[y]  = highest[y+1];
29      lowest[y+1] = lowest[y];
30    }
31
32    for(int i=0; i<B; ++i) {
33      cout << highest[i] - lowest[i] + 1;
34      cout << (i==B-1 ? "\n" : " ");
35    }
36  }
```

Dieser Code sortiert die Mitarbeidenden, der Rest der Lösung ist offensichtlich in linearer Zeit möglich. Die Laufzeit ist deshalb $O(w \log w + b)$. Wir speichern die Positionen der Mitarbeidenden und die Listen l und H von Länge b, die Speichernutzung ist deshalb $O(b + w)$.

Osaka Tube

Teilaufgabe 1: Ein Gegenbeispiel finden (10 Punkte)

Das Netz ist auf dem folgenden Bild dargestellt. Es gibt zwei Zonen ($z = 2$), die jeweils in Klammern am Knoten vermerkt sind. Die Fahrtdauern sind an den Kanten markiert. Die beiden Zonen kosten $p_0 = p_1 = 1$ Yen. Nun führt die einzige kürzeste Route vom Knoten A zum Knoten B über den Knoten C und kostet $p_0 + p_1 = 2$ Yen. Die Route über den Knoten D ist strikt länger, aber billiger, da nur die Zone 0 gelöst werden muss.



Teilaufgabe 2: Ringnetz (30 Punkte)

Nehmen wir an, das gegebene Netz ist nicht konsistent. Dann gibt es zwei Stationen A und B, sodass die (einzige) kürzeste Route und die (einzige) billigste Route intern disjunkt sind. Die Station B lässt sich nun entlang der kürzesten Route in Richtung ab A verschieben, solange diese die einzige kürzeste Route bleibt, womit die andere Route (deren Zonenmenge eine Teilmenge der ursprünglichen Zonenmenge ist) die einzige billigste Route bleibt. Da es zwei mögliche Richtungen für die kürzeste Route aus A gibt, gibt es höchstens zwei Kandidatenknoten B für einen bestimmten Knoten A, die überprüft werden müssen.

Wir müssen also für einen Knoten A und die zwei Kandidatenknoten B jeweils die Länge und den Preis der beiden Routen von A nach B (effizient) bestimmen. Die Länge einer Route zu einem Knoten B lässt sich z.B. mit einem Array der Präfixsummen in $O(1)$ bestimmen. Die Preise der beiden Routen von A nach B lassen sich in $O(z)$ bestimmen, indem wir in zwei Arrays jeweils der Länge z die Anzahl Stationen auf den jeweiligen Routen in den jeweiligen Zonen berechnen und dann die Summe der Preise von den Zonen mit einer positiven Anzahl bestimmen.

Die beiden Arrays und die zugehörigen Preise lassen sich nun in $O(1)$ aktualisieren, wenn wir von einem Knoten B zu einem Nachbarknoten B' übergehen. Somit lassen sich in $O(n + z)$ die beiden Kandidatenknoten B für einen fixen Knoten A bestimmen und überprüfen.

Wenn wir schliesslich A auf einen Nachbarknoten verschieben, müssen wir möglicherweise die beiden Kandidatenknoten B auch wieder verschieben (in $O(1)$ pro Verschiebung). Insgesamt (für alle betrachteten Knoten A) machen die beiden Kandidatenknoten B höchstens einen vollen Umlauf (d.h. $O(n)$ Verschiebungen). Die Gesamtlaufzeit bleibt

also in $O(n + z)$ und die Speicherkomplexität kann nicht höher sein.

Bewertungsschema:

- Erklärung wie man die Länge einer Route auf dem Kreis bestimmt: 5 Punkte.
- Erklärung wie man den Preis einer Route auf dem Kreis bestimmt: 5 Punkte.
- Eine Beobachtung oder Bedingung wann ein Ringnetz konsistent ist: 5 Punkte.
- Die korrekte Laufzeit: 5 Punkte. ($z \in O(n)$ darf angenommen werden.)
- Die optimale Laufzeit: 10 Punkte.

Teilaufgabe 3: ÖV-Netz von Osaka (60 Punkte)

Zunächst berechnen wir die Längen der kürzesten Routen zwischen allen Paaren von Stationen. Mit dem Algorithmus von Floyd-Warshall lässt sich dieser Schritt in $O(n^3)$ implementieren.

Nun müssen wir die Längen der kürzesten billigsten Routen zwischen allen Paaren von Stationen bestimmen. Wir betrachten dazu alle möglichen Teilmengen der Zonen und für jede solche Teilmenge erstellen wir den Teilgraph aller Stationen, die zu einer dieser Zonen gehören. Dann bestimmen wir die Längen der kürzesten Routen zwischen allen Paaren von Stationen in diesem Teilgraph. Schliesslich schauen wir für alle Paare von Stationen, bei denen die eine von der anderen in dem Teilgraph erreichbar ist, ob die jeweilige Teilmenge der Zonen eine billigere Route als bisher liefert bzw. ob es beim gleichen (günstigsten) Preis eine kürzere Route gibt. Somit bestimmen wir die Längen der kürzesten billigsten Routen für alle Paare von Stationen. Mit dem Algorithmus von Floyd-Warshall lässt sich dieser Schritt in $O(n^3)$ pro Teilmenge implementieren, d.h. in $O(2^z n^3)$ insgesamt.

Am Schluss muss für jedes Paar von Stationen überprüft werden, ob die Länge der kürzesten Route gleich der Länge der kürzesten billigsten Route ist.

Die Gesamtlaufzeit ist somit in $O(2^z n^3)$ und die Speicherkomplexität bei $O(n^2 + z)$.

Bemerkung. Die Gesamtlaufzeit lässt sich um den Faktor z verbessern, indem die Teilmengen der Zonen der Kardinalität nach bearbeitet werden und für eine Teilmenge Z der Zonen eine Zone $i \in Z$ ausgesucht wird, sodass höchstens $\frac{n}{z}$ Stationen dieser Zone angehören (wenn es keine solche Zone $i \in Z$ gäbe, dann gäbe es mehr als n Stationen — Widerspruch). Dann lässt sich die bereits berechnete Floyd-Warshall Tabelle der Zonenmenge $Z \setminus \{i\}$ in die Floyd-Warshall Tabelle der Zonenmenge Z umwandeln, durch Hinzufügen der höchstens $\frac{n}{z}$ Stationen der Zone i, mit Zeitaufwand in $O(n^2)$ pro Knoten (die inneren zwei Schleifen von Floyd-Warshall). Somit wird die Gesamtlaufzeit auf $O(2^z n^2 \frac{n}{z}) = O(2^z n^3/z)$ reduziert. Die Speicherkomplexität steigt jedoch auf $O(2^z n^2)$. Der Speicherverbrauch lässt sich auf $O(2^z + z n^2)$ reduzieren, indem die Abhängigkeiten der verschiedenen Floyd-Warshall Tabellen explizit vorberechnet und mit einer Tiefensuche abgearbeitet werden. (Der Baum der Abhängigkeiten hat Tiefe $\leq z$.)

Bemerkung. Wenn statt dem Algorithmus von Floyd-Warshall der Algorithmus von Dijkstra mit Fibonacci-Heap aus allen Knoten gestartet wird, um die kürzesten Routen

zwischen allen Paaren von Stationen zu bestimmen, lässt sich die Gesamtlaufzeit auf $O(2^z(mn + n^2 \log\ n))$ reduzieren, bei gleicher Speicherkomplexität.

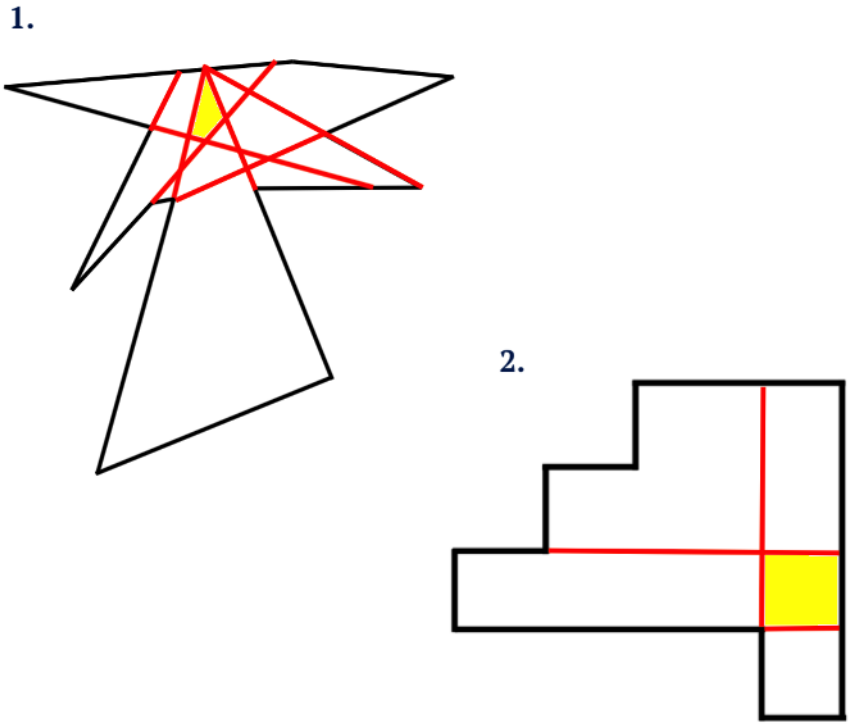
Bewertungsschema:

- Erklärung wie man die Längen der kürzesten Routen zwischen allen Paaren von Stationen bestimmt: 10 Punkte.
- Erklärung wie man die Längen der kürzesten billigsten Routen zwischen allen Paaren von Stationen bestimmt: 40 Punkte.
- Die korrekte Laufzeit einer korrekten Lösung: 10 Punkte.

Dandyshow

Teilaufgabe 1: (10 Punkte)

Markiere alle Stellen, wo Dandy seine Discoparty veranstalten kann.



(Wie man dazu kommt, wird in den nächsten Aufgaben erläutert).

Teilaufgabe 2: Lösen von achsenparallelen Universen (30 Punkte)

Wir haben die Kanten des Polygons im Uhrzeigersinn gegeben. Wenn wir entlang dieses Polygons im Uhrzeigersinn wandern unterscheiden wir jetzt zwischen 4 Typen von Kanten mit Richtung (nach oben, nach unten, nach links, nach rechts). Realisiere, dass man jede dieser Strahle zu einer Linie verlängern kann und dass auf einer Seite der Linie Maus Stoffl die Party nicht schmeissen kann und auf der Anderen schon (aus Sicht dieser Kante). Wenn man also für jede Kante eine Hälfte abdeckt, bekommt man genau die Fläche, auf der Stofl seine Party schmeissen kann (da für jede Kante diese Bedingung notwendig und hinreichend ist). Dabei muss man nur für die $\downarrow$ Strahlen die Rechteste nehmen, für $\uparrow$ die Linkeste, für $\leftarrow$ die Unterste und für $\rightarrow$ die Oberste. Wenn nun diese rechteste $\downarrow$ rechts von der linkesten $\uparrow$ ist überdecken sich die Abdeckungsflächen

und Stoffl kann seine Party nicht machen. Analog gilt, wenn die unterste ← unterhalb des obersten → ist, dass Stoffl seine Party nicht machen kann. Falls aber diese beiden notwendigen Bedingungen erfüllt sind, kann Dandy seine Show irgendwo in dem übrigbleibendem Rechteck machen. Also kann man zum Beispiel den Mittelpunkt ausgeben:

$$\left(\frac{(\text{linkeste } \downarrow) + (\text{rechteste } \uparrow)}{2}, \frac{(\text{unterste } \leftarrow) + (\text{oberste } \rightarrow)}{2}\right)$$

(Je nach Definition kann man die Fälle wo nur ein Punkt oder eine Linie übrig ist seperat behandeln, dass wurde aber bei der Korrektur ignoriert).

Es gab 30 Punkte für eine korrekte Idee in linearer Laufzeit mit Pseudocode und Laufzeitanalyse und jeweils Abzug falls gewisses ausgelassen wurde.

Langsamere Lösung

Manche Teilnehmer haben angenommen, dass die Koordinaten ganze Zahlen ist. Dann haben sie einfach flood-fill jede Kante extended bis sie sich mit einer anderen Kante berührt. Mit der Annahme, dass Koordinaten ganze Zahlen sind, gibt es 10 Punkte, wenn man alles korrekt hingeschrieben hat, Laufzeit richtig berechnet hat $O(n \cdot X \cdot Y)$ und einen Pseudocode hingeschrieben hat.

Man kann die gleiche Lösung auch machen, in dem man einfach statt ganze Zahlen, die Zahlen entlang der X und Y Koordinate sortiert (coordinate-compression).
Im schlimmsten Fall haben alle Kanten dann unterschiedliche Koordinaten und es kann $O(n^2)$ Laufzeit haben. Diese Lösung gab 20 Punkte.

Teilaufgabe 3: Lösen vom allgemeinen Fall (60 Punkte)

Im allgemeinen Fall, gilt das Prinzip mit dem Abdecken immer noch. Nur ist hier die übrigbleibende Fläche kein Rechteck mehr, sondern eine konvexes Polygon (es muss konvex sein, da diese Halbflächen (also Ungleichungen der Form $a \cdot x + b \cdot y \leq c$) eine konvexe Menge ist und die Schneidung von konvexen Mengen immer noch konvex ist). Wie bekommt man nun aber diese Intersektion dieser Strahlen effizient hin? Der Trick funktioniert sehr ähnlich wie die Idee zur konvexen Hülle. Wir sortieren zuerst die Strahlen anhand ihrer Richtung, also anhand ihres Winkel zur x-Achse. Diesen Winkel können wir entweder durch eine Fallunterscheidung mit sin und cos hinbekommen, oder etwas einfacher mithilfe des Skalarprodukts.

$$a \cdot b = |a||b| \cos(\measuredangle) \Rightarrow \measuredangle = \arccos\left(\frac{a \cdot b}{|a||b|}\right)$$

Da acos nur definiert ist auf $[0, \pi)$, muss man trotzdem noch eine kleine Fallunterscheidung machen. Für einen beliebigen Strahl erhalten wir also den Winkel wie folgt:

$$\measuredangle(s) = \begin{cases} \arccos\left(\frac{s \cdot t}{|s||t|}\right), \text{falls } y(s) \geq 0 \\ 2\pi - \arccos\left(\frac{s \cdot t}{|s||t|}\right), \text{falls } y(s) < 0 \end{cases}$$

wobei t der Vektor der x-Achse ist, also: $t = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

Die genaue Implementation des Winkels war nicht notwendig um die volle Punktzahl zu erreichen, es gab aber Bonuspunkte, wenn man es richtig geschrieben hat.

Das sortieren der Kanten anhand ihres Winkels geht bekanntlich in $O(n \log n)$ (Bei uns ist Knotenanzahl = $n = m$ = Kantenanzahl).

Falls es Strahlen gibt mit dem gleichen Winkel, können wir wie bei Teilaufgabe 2 nur diejenigen betrachten, die am meisten überdecken. In den graphisch gelösten Aufgaben, sind die Strahlen die so aussortiert werden jeweils mit einem ' markiert. (Diesen übrigbleibenden Strahl kann man z.B. herausfinden, indem man einen Strahl nimmt der 90 Grad im Uhrzeigersinn gedreht ist und die Schnittpunkte von den gleichwinkligen Strahlen berechnet. Der Strahl, der den Schnittpunkt am weitesten in Richtung des gedrehten Strahls hat, ist dieser übrigbleibende Strahl).

Nun betrachten wir die beiden Kanten mit den kleinsten Winkel. Realisiere, dass beide weniger als 180° auseinander sind, da zwei aufeinanderfolgende Kanten in einem einfachen Polygon weniger als 180° Zwischenwinkel haben. Somit existiert für jede Kante eine weitere, die weniger als 180° auseinander ist. Von diesen beiden Strahlen berechnen wir den Schnittpunkt und setzen diesen auf eine leere Schnittpunkt-Deque. Die beiden Strahle fügen wir in eine leere Strahlen-Deque (zuerst der mit dem kleinsten Winkel und dann der mit dem zweit-kleinsten Winkel). Ein Deque ist einfach ein Array, welches $O(1)$ Zugriff und amortisiert $O(1)$ vorne und hinten hinzufügen kann. (Siehe auch $std :: deque$).

Am Ende wir der Schnittpunkt-Deque alle Eckpunkte des konvexen Polygon besitzen, welches der Bereich markiert, auf dem Dandy seine Show machen kann (auf dem Strahlen-Deque wird die Richtung der Strahlen gespeichert).

Um den nächsten Schritt zu verstehen, schaut man sich am besten die vorgelösten Beispiele an.

Wir iterieren jetzt über die restlichen Strahlen (nach Winkel, beginnend mit dem Strahl mit dem drittkleinsten Winkel). Wir müssen nun jede dieser Strahle I schneiden mit der bis jetzt berechneten konvexen Menge, auf der Dandy seine Show machen kann. Der hinterste Strahl des Strahlen-Deques nennen ich jetzt der einfachheitshalber A und der hinterste Schnittpunkt des Schnittpunkt-Deques a. Nun berechnen wir den Schnittpunkt von I und A (der existiert immer, ausser beim Randfall wo A und I genau aufeinander zeigen, dass wird später gehandhabt). Falls der neue Schnittpunkt i in Richtung der Kante A **nach dem** Schnittpunkt a kommt, können wir den neuen Schnittpunkt einfach zum Schnittpunkt-Deque hinten hinzufügen und I zum Strahlen-Deque hinten hinzufügen. Dieser Fall geschieht zum Beispiel bei Schritt (3) auf (4) in Beispiel 1. Ich habe dort die

Werte i, I, a, A der einfachheitshalber noch hingezeichnet. Falls der neue Schnittpunkt i in Richtung der Kante A **vor dem** Schnittpunkt a kommt, entfernen wir A und a von ihren jeweiligen Deques und versuchen erneut I hinzuzufügen, bis es klappt. Falls sich I mit allen Kanten **vor dem** jeweiligen hintersten Schnittpunkt des Schnittpunkt-Deques schneidet, existiert keine Lösung (siehe Beispiel 3). Das ist im Prinzip die ganze Idee, aber das Schneiden ist noch etwas komplizierter sobald man Strahle betrachtet, die mehr als 180° Winkel zum ersten Strahl haben. Diese Strahlen schneiden die konvexe Menge eben an zwei Orten (einmal vorne und einmal hinten). In den jeweiligen Beispielen sind die pupuren/pinken Punkte jeweils die Punkte, die sich mit dem Strahl hinten vom Strahlen-Deque schneiden und die blauen Punkte sind jeweils die Punkte, die sich mit dem Strahl vorne vom Strahlen-Deque schneiden. Man macht also hier einfach das gleiche nur einmal von vorne und einmal von hinten. Bei Beispiel 2 sieht man beide dieser Schneidungen sehr schön von Schritt (6) auf (7) oder bei Beispiel 1 von Schritt (7) auf (8).

Jetzt muss man nur noch den Randfall betrachten, wo Dandy seine Show auf einer Linie (oder einen Punkt) machen könnte. Dies wurde aber nicht verlangt, du konntest also annehmen das Dandy's Party ein gewisses Volumen besitzen musste. Im Prinzip funktioniert das ganze immer noch, nur kann es jetzt vorkommen, dass purpure/pinke und blaue Punkte am gleichen Ort sein können. Die konvexe Mengen beschrieben durch die Ungleichungen, bzw. durch die Schnittpunkte und Strahlen des Deques Formen aber immer noch die Ränder dieser konvexen Menge. Beispiel 4 sollte das besser erläutern.

Laufzeitanalyse: Das sortieren geht in $O(n \cdot log(n))$. Hätte man die Strahlen schon sortiert, würde der Rest des Algorithmus in $O(n)$ laufen, da jede Kante höchstens zweimal zum Stack hinzugefügt, bzw. zweimal vom Stack entfernt wird (zweimal weil jede Kante sowohl einen purpuren Schnittpunkt, wie einen blauen Schnittpunkt haben kann). Dieses Argument über die Laufzeit nachzudenken heisst Amortisation. Am besten schaust du dir die Laufzeit von Grahams konvexer Hülle Algorithmus an, da es exakt die gleiche Amortisation verwendet. Falls dir Amortisation neu ist, kannst du dir mal überlegen, wie $std :: vector$ eine amortisierte Laufzeit von $O(1)$ für $push_back$ benötigt.

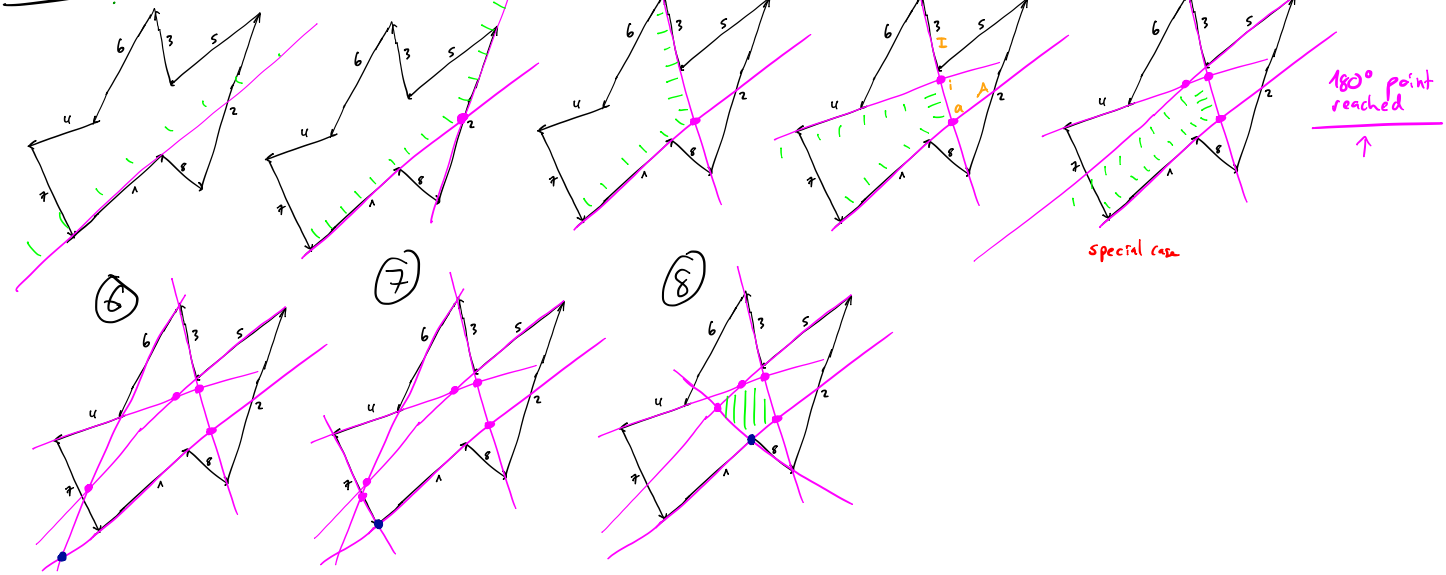
Es gab 20 Punkte für eine korrekte Lösung. Zusätzlich gab es 10 Punkte für eine Lösung in $O(n^2)$ Zusätzlich gab es 30 Punkte für eine Lösung in $O(n \cdot \log n)$

Pseudocode:

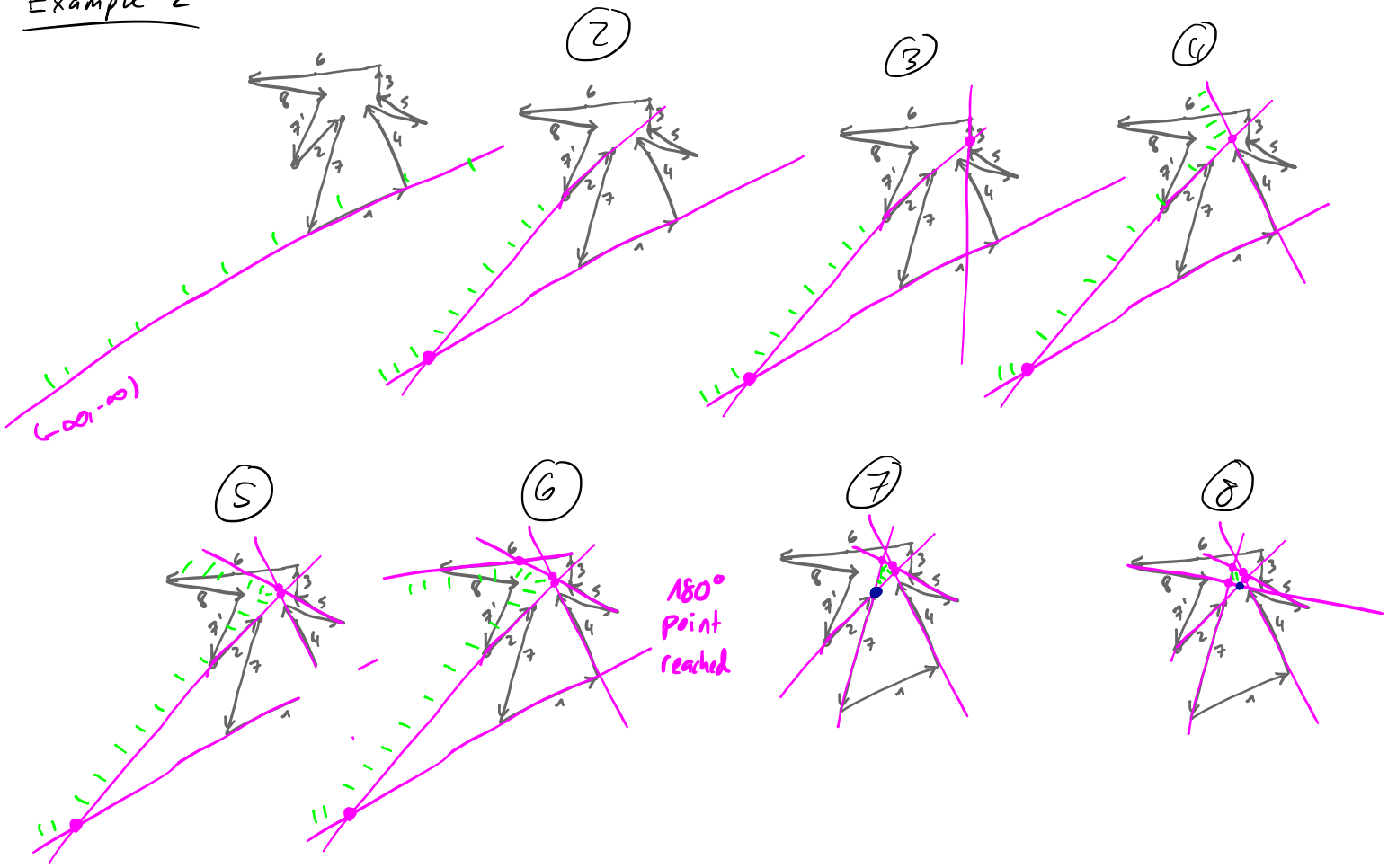
```
1  input: tuple<int,int,int,int>[] rays
2  algorithm:
3  sortedrays := sort_by_angle_to_x_axis (rays)
4  deque<tuple<int,int,int,int> > raystack
5  deque<pair<int,int> > intersectionstack
6
7  raystack = sortedrays[0];
8  firstangle = angle(sortedrays[0], x-axis)
9
10 foreach iray : sortedrays [2..N-1] {
11   while(true) {
12     if (raystack.size < 2) exit('too bad')
13     intersectpoint= intersect (iray, raydeque.back)
14
15     if intersectpoint comes after the intersectionstack.back along raydeque.back
```

```
16          intersectionstack.push_back(intersectpoint)
17          raystack.push_back(iray)
18          break;
19        raystack.pop_back
20        intersectionstack.pop_back
21      }
22      if(angle (iray,x-axis) - 180 >= firstangle) {
23        while(true) {
24          if (raystack.size < 2) exit('too bad')
25          intersectpoint= intersect (iray, raydeque.front)
26
27          if intersectpoint comes before the intersectionstack.front along raydeque.front
28            intersectionstack.push_front(intersectpoint)
29            raystack.push_front(iray)
30            break;
31          raystack.pop_front
32          intersectionstack.pop_front
33        }
34      }
35    }
```

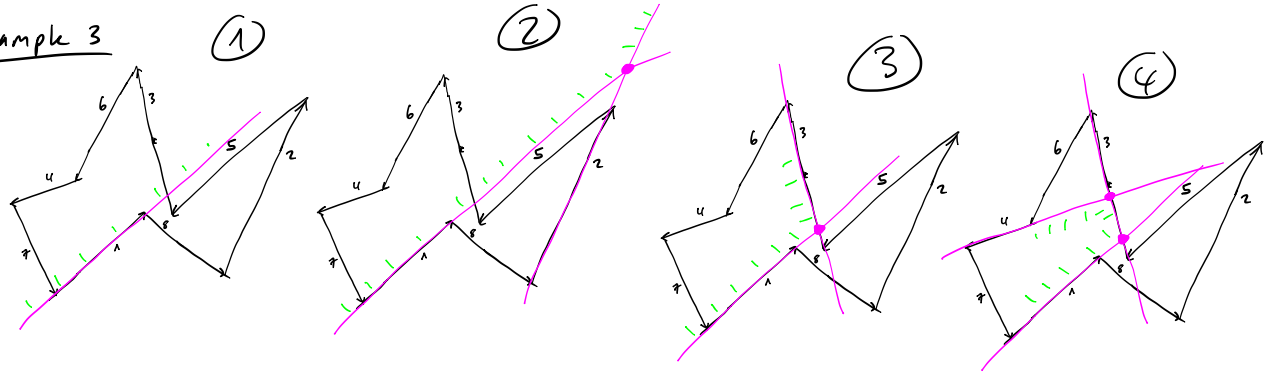
Example 1

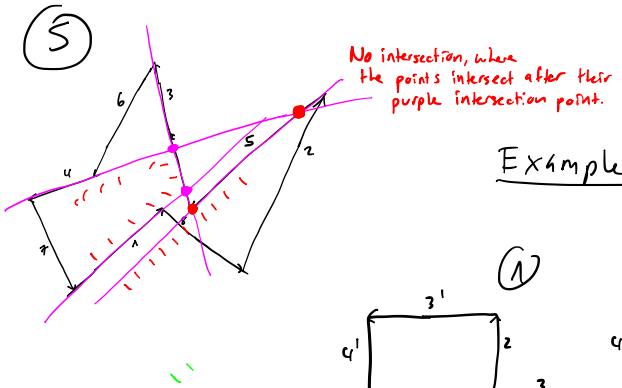


Example 2

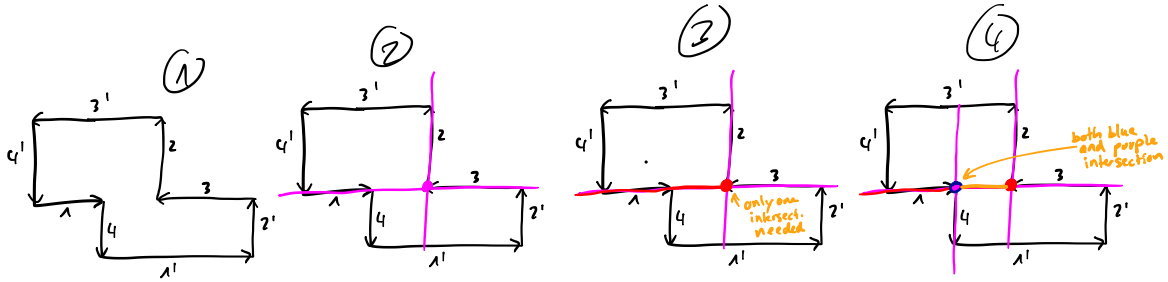


Example 3



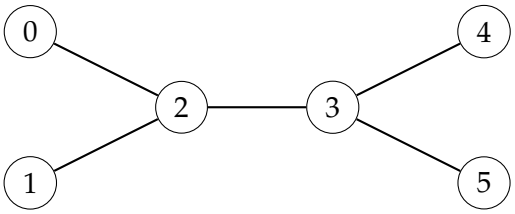


Example 4 Special case 180° →←



Erdgasversorgung

Teilaufgabe 1: (10 Punkte)

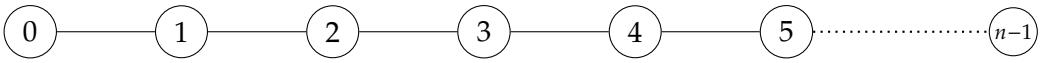


Das Leck kann mit 3 Anfragen gefunden werden. Zuerst schauen wir, in welche Richtung das Gas zwischen den Knotenpunkten 2 und 3 fliesst. Angenommen es fliesst nach Links, das heisst das Leck ist entweder in Knotenpunkt 0, 2 oder 1. Um die genau Position zu bestimmen prüfen wir auch 0-2 und 1-2. Nun gibt es 3 Möglichkeiten:

- Beide fliessen nach rechts: Das Leck ist in Knotenpunkt 2.
- In Leitung (0,2) fliesst es nach links: Das Leck ist in Knotenpunkt 0.
- In (1,2) fliesst es nach links: Das Leck ist in Knotenpunkt 1.

Falls das Gas in (2,3) nach Rechts fliesst, gehen wir analog vor.

Teilaufgabe 2: (30 Punkte)



Wir können das Leck mit Binärer Suche mit $\log n$ Anfragen finden. Wenn wir eine Anfrage für die Leitung die sich in der Mitte zwischen Knotenpunkt $m_0 = \left\lfloor \frac{n-1}{2} \right\rfloor$ und $m_0 = \left\lfloor \frac{n-1}{2} \right\rfloor + 1$ befindet stellen, dann halbiert sich die Anzahl Knotenpunkten, in denen sich das Leck befinden kann. So können wir mit jeder Anfrage die Anzahl Kontenpunkten, die das Leck enthalten können halbieren, bis wir schliesslich nach $\log n$ Anfragen nur noch einen Knotenpunkt, den mit dem Leck, übrig haben.

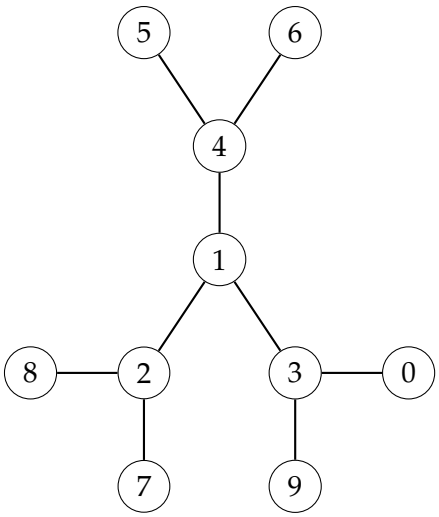
Der Algorithmus muss vor jeder Anfrage die beiden mittleren Knoten berechnen und nach der Anfrage das Intervall, auf dem sich das Leck befinden kann, aktualisieren. Die Laufzeit dafür ist konstant und da $\log n$ Anfragen nötig sind, ist die Laufzeit des Algorithmus ebenfalls $O(\log n)$. Beachte wie wir in der Implementierung der Binären Suche halboffene Intervalle benutzen. Das heisst wenn $l = 1$ und $r = 3$, dann sind die Elemente die im Intervall sind $1, 2$.

$l \leftarrow 0$
$r \leftarrow n - 1$
while $r - l > 0$ **do**

```
    m_0 ← l + (r − l)/2
    m_1 ← l + (r − l)/2 + 1
    answer ← Query(m_0, m_1)          ▷ Here we actually go check a pipe
    if answer = m_0 then
        r ← m_1
    else
        l ← m_1
    end if
end while
print "The Leak is in node $l"
```

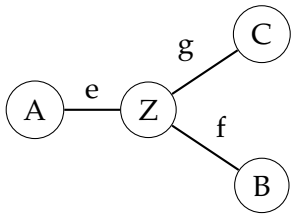
Teilaufgabe 3: (60 Punkte)

Wir würden gerne etwas ähnliches machen wie für den vorherigen Subtask, wo wir mit jeder Anfrage die Hälfte, der noch übrigen Knotenpunkten ausschlissen konnten. Es ist aber auch klar, dass der Ansatz nicht mehr funktioniert. Betrachten wir folgendes Beispiel:



Egal welche Leitung wir anschauen, ist es nicht möglich die Hälfte der Knotenpunkten auszuschliessen. Es ist aber möglich ein Drittel der Knotenpunkte auszuschliessen, wenn wir eine Leitung betrachten die mit Knotenpunkt 1 verbunden ist. Wir können uns daher fragen, ob es immer möglich ist ein Drittel der Knoten auszuschliessen? Und die Antwort ist in der Tat Ja und wir können es wie folgt beweisen:

Nehmen wir an, es wäre nicht möglich eine Leitung zu finden, die mindestens ein Drittel der Knotenpunkten ausschliesst. Dann sei e die Leitung bei der die Seite, auf der sich weniger Knotenpunkte befinden, möglichst viele Knotenpunkte sind. Etwas anschaulicher:



Wir nennen A die Seite die weniger Knotenpunkte hat. Wir wissen das $|A| < \frac{1}{3} \cdot n$, da wir ja angenommen haben, dass es keine Leitung gibt, bei der man mindestens ein Drittel der Knotenpunkte ausschliessen könnte.

Beachte, dass Z nicht zwingend ein Knotenpunkt mit Grad 3 ist, $|C|$ oder $|B|$ kann auch Null sein. Nun wissen wir das $|A| + |B| + |C| + 1 = n$ ist (das +1 ist der Z Knoten) und dass $|A| < \frac{1}{3} \cdot n$. Also gilt $|B| + |C| + 1 > \frac{2}{3}n$. Nun ist mindestens eine der 2 Möglichkeiten erfüllt:

- $|B| > \frac{1}{3}n$: In dem Fall hätten wir die Leitung f anstatt die Leitung e ausgewählt, da es auf der Seite mit weniger als $\frac{1}{3}n$ Knotenpunkten bei f mindestens einen Knotenpunkt mehr hat, nämlich Z. Wir haben e aber so gewählt das $|A|$ maximal ist, dass ist ein Widerspruch.
- $|C| > \frac{1}{3}n$: Das selbe Argument, aber mit der Leitung g statt f.

Es führt also in beiden Fällen zu ein Widerspruch, woraus wir schliessen, dass es unmöglich ist, dass es keine Leitung gibt mit der wir nicht mindestens ein Drittel der Knotenpunkte ausschliessen können.

Wir können also immer eine Leitung finden, mit der wir mindestens $\frac{1}{3} \cdot n$ Knotenpunkte ausschliessen können. Wenn wir mit jeder Anfrage $\frac{1}{3} \cdot n$ ausschliessen, dann müssen wir maximal $\lceil log_{3/2}(n) \rceil = \left\lceil \frac{log(n)}{log(3/2)} \right\rceil$ Anfragen stellen. Das Aufrunden können wir auch weglassen, denn es fügt höchstens 1 hinzu, ein konstanter Term, an dem wir nicht interessiert sind.

Nun müssen wir noch einen Algorithmus entwerfen der eine solche Leitung findet. Dazu können wir eine Tiefensuche(DFS) starten, die für jede Leitung berechnet, wie viele Knotenpunkte sich auf den beiden Seiten befinden. Die Leitung bei der die Differenz zwischen den beiden minimal ist, ist die optimal. Die Laufzeit von DFS ist $O(n)$. Nach jeder Anfrage müssen wir erneut eine Tiefensuche auf den maximal $\frac{2}{3}n$ verbleibenden Knotenpunkten machen. Die Laufzeit ist also $O(n + \frac{3}{2}n + (\frac{3}{2})^2 \cdot n + \dots) = O(n)$.

Es gibt übrigens noch eine bessere Lösung die eine Technik namens "Centriod Decomposition"[1] verwendet. Dadurch lässt sich der dominante Faktor von $\frac{1}{log(3/2)}$ auf $\frac{1}{\log \frac{1}{2}(1+\sqrt{5})}$ verkleinern. Diese Lösung wurde jedoch nicht verlangt und soll nur als Anmerkung für Interessierte dienen.

[1] `https://threads-iiith.quora.com/Centroid-Decomposition-of-a-Tree`