Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Daniel Seemaier, Tobias Heuer

6. Übungsblatt zu Algorithmen II im WS 2020/2021

http://algo2.iti.kit.edu/AlgorithmenII_WS20.php
{sanders, daniel.seemaier, tobias.heuer}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: ADAC Mitgliedschaft*)

Der "**A**utomobil **D**urch **A**lgorithmiker **C**lub" (ADAC) leistet auf Autobahnen Pannenhilfe. Ein Autofahrer hat in seiner Zeit als Verkehrsteilnehmer n Pannen, $n \in \mathbb{N}_{\geq 0}$, für die er die Hilfe des ADAC in Anspruch nehmen muss. Für jede geleistete Pannenhilfe verlangt der Club eine Aufwandsentschädigung abhängig von der Schwere der Panne. Mitglieder beim ADAC müssen lediglich ein Viertel dieser Kosten bezahlen. Eine lebenslange Mitgliedschaft kann man sich durch eine Einmalzahlung in Höhe von 1000 DM (**D**ijkstra **M**ark) sichern.
Da nicht schon mit Erwerb des Führerscheins klar ist, wie viele Pannen man in seinem Leben haben wird und wie schwerwiegend diese sein werden, stellt sich die Frage, ab wann es sich lohnt eine Mitgliedschaft beim ADAC zu erwerben.

a) Geben Sie eine Strategie an, die einen kompetitiven Faktor (competitive ratio) von ∞ erreicht. Begründen Sie kurz.

b) Wie gut ist die Strategie, sich nie eine Mitgliedschaft beim ADAC zu sichern? Begründen Sie.

c) Zeigen Sie, dass folgende Strategie einen kompetitiven Faktor $c = 3$ hat. Die Strategie ist, sich beim Pannenhelfer eine Mitgliedschaft zu kaufen, wenn die momentan von ihm bearbeitete Panne die Gesamtausgaben für Pannen (ohne Mitgliedschaft) auf über 500 DM anheben würde.

Hinweis: Verwenden Sie die summierten Gesamtkosten K über alle Pannen (ohne Mitgliederrabatt).

Musterlösung:
Ein Algorithmus ALG wird als *streng c-kompetitiv* bezeichnet, wenn für alle Eingaben I gilt

$$c = \sup_I \frac{ALG(I)}{OPT(I)}$$

Dieser Wert wird als *kompetitiver Faktor* (*competitive ratio*) bezeichnet.

Der Übersichtlichkeit halber wird im Folgenden ohne Einheiten gerechnet:

a) Wenn man sofort mit Erhalt der Fahrerlaubnis eine Mitgliedschaft beim ADAC erwirbt, gibt man im schlimmsten Fall 1000 DM aus, nimmt aber die Hilfe des ADAC nie in Anspruch. Dies ergibt einen kompetitiven Faktor von $c = \frac{1000}{0} = \infty$.

b) Wenn man sich nie eine Mitgliedschaft kauft, gibt man offensichtlich höchstens 4 mal soviel für den ADAC aus wie ein Mitglied. Die summierten Gesamtkosten für alle Pannen seien mit K bezeichnet. Für $K \to \infty$ konvergiert der kompetitive Faktor gegen $c = \frac{K}{1000+K/4} \to 4$.

c) Die summierten Gesamtkosten für alle Pannen seien mit K und die summierten Kosten vor Beitritt zum ADAC mit $K_1 \leq 500$ bezeichnet. Die eigene Strategie liefert

$$ALG = \begin{cases} K_1 + 1000 + (K - K_1)/4 & K > 500, \\ K & \text{sonst} \end{cases}$$

in Abhängigkeit davon, ob man jemals über 500 DM Kosten für Pannen hat oder nicht. Die optimale Strategie ist durch

$$OPT = \begin{cases} 1000 + K/4 & K \geq 1333\frac{1}{3}, \\ K & \text{sonst} \end{cases}$$

gegeben. Entweder kauft man sich sofort eine Mitgliedschaft oder nie. Die Grenzkosten ergeben sich durch Lösen von $1000 + K/4 \stackrel{!}{=} K$. Der kompetitive Fakor ist der maximale Quotient von ALG und OPT. Allgemein gilt

$$\frac{ALG(K)}{OPT(K)} = \begin{cases} \frac{K}{K} & K \leq 500, \\ \frac{1375+K/4}{1000+K/4} & K \geq 1333\frac{1}{3}, \\ \frac{1375+K/4}{K} & \text{sonst} \end{cases}$$

(mit $K_1 = 500$ gesetzt, da wir nur am maximalen Wert interessiert sind). Das Supremum dieses Quotienten ist 3 für $K \to 500, K > 500$. Damit ist der kompetitive Faktor dieser Strategie

$$c = \sup_K \frac{ALG(K)}{OPT(K)} = 3.$$

Aufgabe 2 (*Analyse: Online-`gaming`-Algorithmen*)

Angestellte in einem Rechenzentrum haben einen recht eintönigen Job. Ihnen stehen zwar die größten Rechner zur Verfügung, Sie dürfen diese aber nicht selbst verwenden. Stattdessen heisst es nur, die Maschinen möglichst gut auszulasten und Rechnungen zu schreiben. Ein ziemlich langweiliger Job möchte man meinen – zum Glück gibt es noch Computerspiele.
Ein Mitarbeiter hat zu Weihnachten ein neues Computerspiel erhalten, dass er liebsten andauernd spielen würde. Diesem Wunsch steht leider seine Arbeit im Wege. Eine neue Dienstanweisung besagt, dass die teuren Großrechner zu mindestens 50% ausgelastet sein müssen. Da das Scheduling in diesem Rechenzentren noch von Hand durchgeführt wird, muss der spielfreudige Mitarbeiter die laufenden Jobs überwachen und schnell neue Jobs auf leerlaufende Maschinen verteilen. So bleibt leider nur wenig Zeit zum Spielen am Arbeitsplatz.
Jeder Job hat eine Mindestlaufzeit von 5 Minuten. Die Zeit zum Verteilen der Jobs kann für die Bestimmung der Auslastung vernachlässigt werden. Der Mitarbeiter kann in dieser Zeit aber nicht spielen. Ein Job hat während seiner Ausführung die Maschine exklusiv. Es gibt keine automatischen Benachrichtigungen über das Ende eines Jobs. Der Mitarbeiter muss dies selbst periodisch überprüfen.

a) Entwerfen Sie einen *online scheduling* Algorithmus, der die freie Zeit des Mitarbeiters unter Einhaltung der Nebenbedingungen maximiert, wenn er einen Großrechner zu betreuen hat.

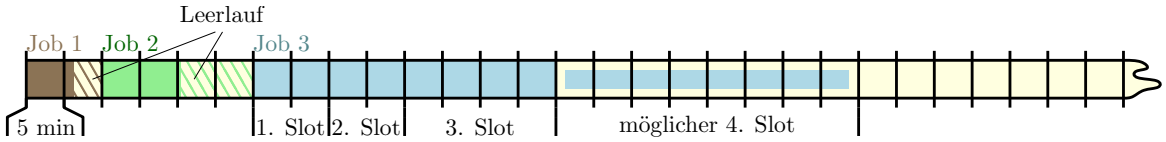
b) Zeigen Sie, dass Ihr Algorithmus optimal bzgl. der Freizeit des Mitarbeiters ist.

Musterlösung:

a) Wir definieren den Algorithmus induktiv. Durch die Mindestlaufzeit von 5 Minuten kann man jedem Job zunächst ein Zeitslot von 10 Minuten zuweisen. Nach Ablauf dieses Zeitslots muss der Mitarbeiter nachschauen, ob der Job bereits abgeschlossen ist, in der Zwischenzeit kann er spielen. Falls der Job in dieser Zeit beendet wurde, stand die Maschine höchstens 50% der Zeit still bis der Mitarbeiter dies erkannt und einen neuen Job gestartet hat. Andernfalls muss ein neuer Zeitslot für den noch laufenden Job zugewiesen werden.

Die Länge des neuen Zeitslots wird gleich der bisherigen Gesamtlaufzeit des Jobs (10 Minuten) gewählt. Dies verdoppelt die mögliche Gesamtlaufzeit des Jobs bis zur nächsten Überprüfung durch den Mitarbeiter auf 20 Minuten. Dadurch wird sichergestellt, dass der Rechner maximal die Hälfte der Zeit leerläuft. Wenn der Job ε Zeiteinheiten nach Start des neuen Zeitslots endet, ist er insgesamt $10+\varepsilon$ Minuten gelaufen und der Rechner damit zu $\frac{10+\varepsilon}{20} > 50\%$ ausgelastet gewesen. Sollte der neue Zeitslot auch nicht genügen, wird dieses Vorgehen wiederholt, bis der Job endet (neuer Zeitslot von $20, 40, \ldots$ Minuten, maximale Gesamtlaufzeit von $40, 80, \ldots$ Minuten).

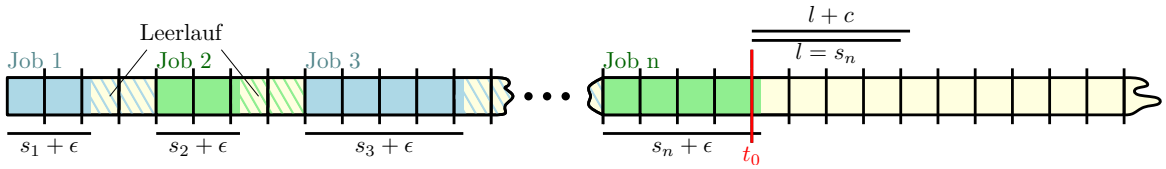
Folgende Abbildung verdeutlicht den Ablauf des Algorithmus:



Kein Job erhält mehr als das Doppelte seiner Laufzeit an Zeitslots zugeteilt. Somit ist der Großrechner im Durchschnitt zu mindestens 50% ausgelastet – wie gefordert.

Auch wenn die Aufgabenstellung einen eher scherzhaften Hintergund hat, so ist die vorgestellte Technik der Verdopplung (*doubling*) ein nützliches Hilfsmittel beim Entwurf von *online* Algorithmen. Insbesondere Algorithmen die Suchschritte benötigen, können durch derartige Techniken oft schon recht gute Approximationsgarantien (bzw. kompetitive Faktoren) liefern.

b) Angenommen, es existiere ein Algorithmus, der längere Zeitslots zwischen zwei Überprüfungen erlaubt als unser Algorithmus. Dann existiert für eine Folge an Jobs ein Zeitpunkt t_0, an dem die zugeteilten Zeitslots beider Algorithmen erstmals voneinander abweichen. Unser Algorithmus weise einen Zeitslot der Länge l zu, der potentiell bessere Algorithmus einen Zeitslot der Länge $l' = l + c$. Betrachte konkret eine Folge an Jobs, für die unser Algorithmus genau die minimale Auslastungsgrenze von 50% einhält (siehe Grafik).



Job i benötigt Zeit $s_i + \epsilon$ mit s_i gleich einer Dauer, nach der eine Überprüfung stattfindet – also nach $10, 20, 40, 80, \ldots$ Minuten. Zum Zeitpunkt t_0 weist unser Algorithmus einen Zeitslot von $l = s_n$ als Verlängerung zu, der andere Algorithmus einen Zeitslot von $l' = l + c = s_n + c$. Wähle $\epsilon < \frac{c}{2n}$, so ergibt sich nach Abarbeitung dieses Zeitslots eine Auslastung von

$$\frac{\sum_{i=1}^{n} s_i + \epsilon}{c + \sum_{i=1}^{n} 2s_i} = \frac{n \cdot \epsilon + \sum_{i=1}^{n} s_i}{c + 2 \cdot \sum_{i=1}^{n} s_i} < \frac{\frac{c}{2} + \sum_{i=1}^{n} s_i}{2 \cdot \left(\frac{c}{2} + \cdot \sum_{i=1}^{n} s_i\right)} = \frac{1}{2}$$

für den alternativen Algorithmus. Beachte: Startet zu dem Zeitpunkt t_0 ein neuer Job, so ist das Ergebnis analog zu erreichen durch eine Wahl von $s_n = 10$, der doppelten Mindestlaufzeit des Algorithmus. Damit hält er nicht – wie gefordert – eine minimale Auslastung von 50% ein. Unser Algorithmus verwendet also bereits die maximal möglichen Zeitslots zwischen zwei Überprüfungen und ermöglicht dem Mitarbeiter die meiste Freizeit.

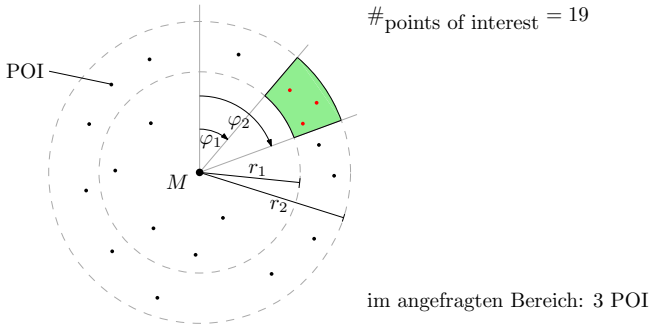
Aufgabe 3 (*Kleinaufgaben: Geometrie-Entwurf*)

Entwerfen Sie einen Algorithmus, der ...

a) in Zeit $O(n \log n)$ ein geschlossenes, kreuzungsfreies Polygon aus n Punkten $P \in \mathbb{R}^2$ konstruiert.

b) in Zeit $O(n)$ für eine Menge von n Filialen einen Standort für ein Zentrallager berechnet, so dass der maximale Abstand (in Luftlinie) zwischen Zentrallager und allen Filialen minimiert wird.

c) (*) in Zeit $O(\log n)$ die Anzahl an *points of interest* (POI) um einen fixen Mittelpunkt M in einem Winkelbereich $[\varphi_1, \varphi_2]$ und einem Entfernungsbereich $[r_1, r_2]$ bestimmt.



Bei n POI ist eine Vorverabeitungszeit von $O(n \log n)$ und ein Platzverbrauch von $O(n)$ erlaubt.

Musterlösung:

a) Bestimme den Mittelpunkt M aller Punkte aus P in $O(n)$. Sortiere die Punkte in $O(n \log n)$ im Uhrzeigersinn um M. Bei gleichem Winkel hat der Punkt, der näher am Mittelpunkt ist Vorrang. Füge die Punkte in dieser Reihenfolge zu einem Polygon zusammen.

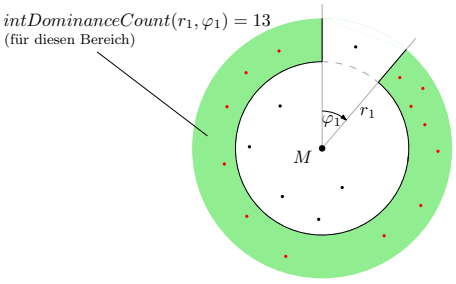
Das erzeugte Polygon ist offensichtlich geschlossen, wenn auch der letzte Punkt mit dem ersten Punkt verbunden wird. Das Polygon ist außerdem kreuzungsfrei, da es monoton in einer Richtung aufgebaut wird (im Uhrzeigersinn und von innen nach nach außen).

b) Das Problem kann mit dem Algorithmus zur Bestimmung der kleinsten einschließenden Kugel gelöst werden. Der Mittelpunkt der berechneten Kugel (bzw. des Kreises in 2D) minimiert den maximalen Abstand zu allen Filialen und ist der gesuchte Standort für das Zentrallager.

c) Die Anfrage kann durch *Wavelet Trees* mit den geforderten Eigenschaften gelöst werden. An Stelle von karthesischen Koordinaten (x, y) werden Kreiskoordinaten (r, φ) verwendet, um den *Wavelet Tree* aufzubauen. In diesem Fall gibt eine Anfrage $intDominanceCount(r_1, \varphi_1)$ die Anzahl an POI im Bereich $[r_1, \infty)$ und $[\phi_1, 2\pi)$ an (siehe Bild). Damit kann die gewünschte Bereichsanfrage konstruiert werden:

$$\begin{aligned} intRangeCount(r_1, r_2, \varphi_1, \varphi_2) &= intDominanceCount(r_2, \varphi_1) - intDominanceCount(r_2, \varphi_2) \\ &\quad - intDominanceCount(r_1, \varphi_1) + intDominanceCount(r_1, \varphi_2) \end{aligned}$$

Sollte der Winkelbereich die 0° überstreichen, teilt man die Anfrage in zwei getrennte Anfragen mit den Winkelbereichen $[\varphi_1, 2\pi)$ bzw. $[0, \varphi_2)$, deren Ergebnisse man addiert.



Aufgabe 4 (*Analyse+Entwurf: Überdeckungsproblem*)

Zur Gebietsüberwachung wurde in einem weitläufigen Gelände ein Sensornetz aus mehreren Millionen Knoten ausgelegt. Die Positionsdaten der Knoten wurden per Funkübertragung an einer zentralen Stelle gesammelt. Jeder Sensorknoten überwacht ein kreisförmiges Gebiet mit Radius r. Durch Fehler in der Ausbringung können dabei starke Überlappungen der von den Knoten überwachten Gebiete entstanden sein.
Um diese Information zu einem späteren Zeitpunkt ggf. nutzen zu können, haben die Betreiber beschlossen, dass alle Knotenpaare bestimmt werden sollen, deren Gebiete sich teilweise überlappen.

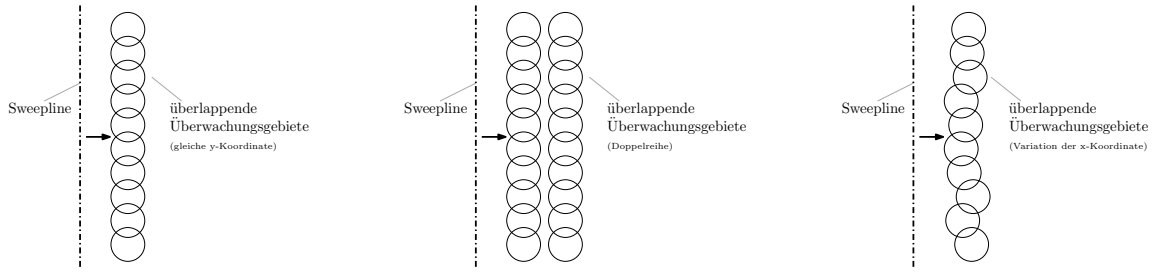
a) Zeigen Sie, dass ein *Sweepline*-Algorithmus, der einfach alle (im Rahmen der Algorithmenausführung) aktiven Sensorknoten auf Überlappung prüft, in quadratischer Laufzeit resultieren kann, auch wenn die Ausgabekomplexität (Anzahl überlappender Knotenpaare) linear ist.

b) Überlegen Sie, wie Sie dennoch einen *Sweepline*-Algorithmus verwenden können, um das Problem in Linearzeit zu lösen, falls die Ausgabekomplexität linear ist.

Musterlösung:

a) Viele *Sweepline*-Algorithmen sind für eine effiziente Ausführung darauf angewissen, dass die Zahl aktiver Elemente gering ist im Vergleich zur Anzahl vorhandener Elemente. Dies kann bei dem gestellten Problem allerdings nicht garantiert werden.

Betrachte eine Sortierung der Knoten nach x-Koordinate und anschließend nach y-Koordinate. Der *Sweepline*-Algorithmus durchlaufe die Knoten in dieser Sortierung. Sollten durch eine ungünstige Verteilung alle Sensorknoten auf der selben y-Koordinate liegen, so sind zu einem Zeitpunkt alle Knoten gleichzeitig aktiv und man muss jeden Knoten mit jedem vergleichen. Dieser Fall resultiert somit in quadratischer Laufzeit, obwohl nur linear viele Schnitte auftreten. Er könnte dennoch effizient gelöst werden, da die Knoten auch in der y-Koordinate geordnet sind und deshalb in sortierter Reihenfolge aktiviert werden. Allerdings kann diese Ordnung durch Aufteilung der Elemente in zwei gleich lange Gruppen oder Variation der x-Koordinate in Schritten von r/n zerstört werden. Die folgenden Abbildungen illustrieren die drei Fälle:

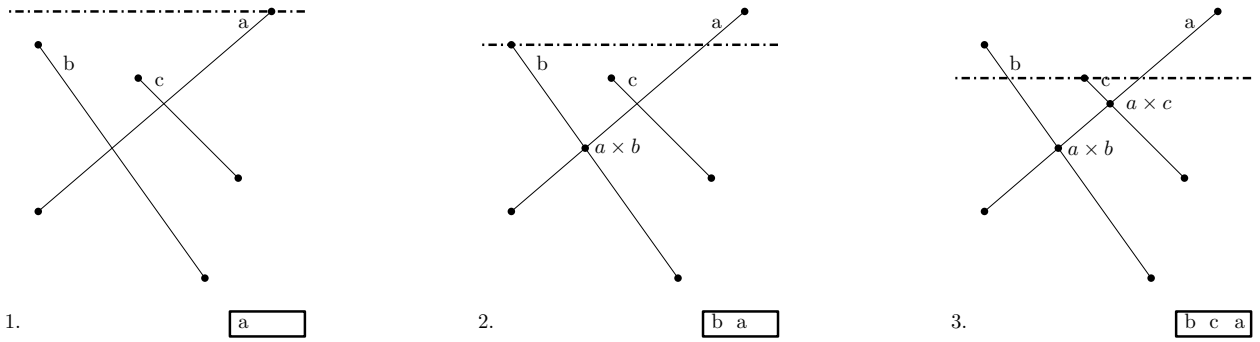


b) Wie in der vorherigen Teilaufgabe gezeigt, besteht das Problem darin, dass gleichzeitig viele Knoten aktiv sein können. Ein Trick zur Umgehung dieses Problems ist die Verwendung von Buckets. Verwaltet man die Menge aller aktiven Elemente in einer sortieren Liste an Buckets (augeteilt anhand der y-Koordinate, Bucketgröße $> r$), so sind nur Vergleiche mit den Elementen aus dem eigenen und den zwei benachbarten Buckets nötig. Da nur reine Überlagerungstests durchgeführt werden, sind bei geeigneter Wahl der Bucketgröße sogar nur Vergleiche mit den benachbarten Buckets nötig, da sich die Kreise eines Buckets garantiert überlagern (gilt für Bucketgrößen $< 2r$).

Sollten sich alle Elemente auf benachbarte Buckets verteilen, lässt sich auch mit diesem Trick eine quadratische Menge an Tests nicht vermeiden. Durch geschickte Wahl der Bucketgrößen lässt sich die Wahrscheinlichkeit eines solchen Falles allerdings meist so weit reduzieren, dass quadratische Laufzeit nur auftritt, wenn die Ausgabekomplexität quadratisch in der Eingabegröße ist. Eine solche Komplexität lässt sich dann allerdings nicht vermeiden.

Aufgabe 5 (*Entwurf: Platzeffizienter Linienschnitt*)

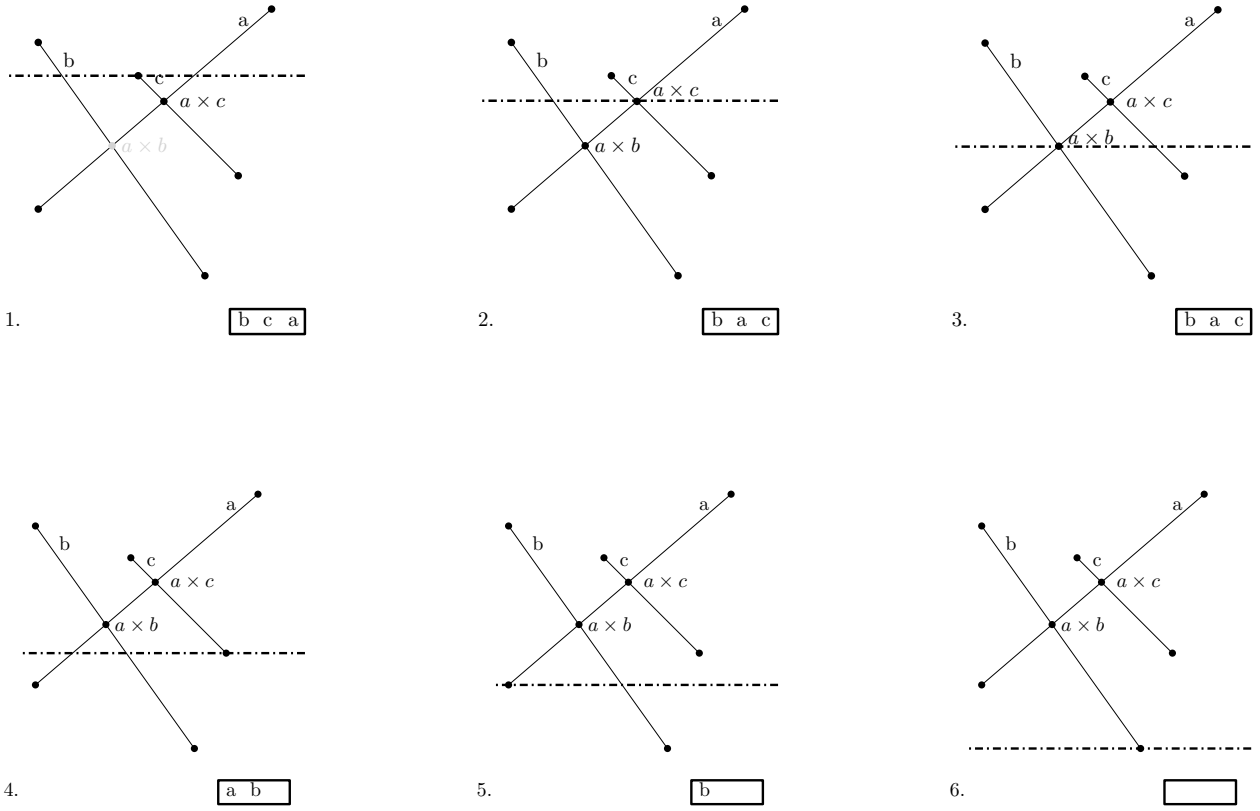
In der Vorlesung wurde ein *Sweepline*-Algorithmus zur Bestimmung der Schnitte zwischen n Liniensegmenten behandelt. Der Algorithmus arbeitet eine Liste an Ereignispunkten (Schnittpunkte, Linienanfänge und Linienenden), in Form einer nach der y-Position sortierten *Queue*, ab. Gleichzeitig führt er eine Liste aktiver Kanten in sortierter Reihenfolge.
Das betrachtete Problem lässt sich in seiner Laufzeitkomplexität nie besser lösen als durch die Anzahl Schnittpunkte vorgegeben. Liegen z.B. $O(n^2)$ Schnittpunkte vor, so kann bestenfalls eine quadratische Laufzeitkomplexität erreicht werden.
Für den Algorithmus aus der Vorlesung gilt die gleiche Einschränkung auch für den Platzbedarf. Die *Queue* muss bis zu $O(n + k)$ Ereignispunkte gleichzeitig halten, bei insgesamt k Linienschnitten. Dies liegt unter anderem daran, dass einmal erkannte Schnittpunkte auch zwischen Liniensegmenten existieren können, die in der sortierten Liste nicht (mehr) benachbart sind. Dies sei durch folgendes Beispiel illustriert:



Im Beispiel wird zuerst der Schnittpunkt $a \times b$ zwischen den Linien a und b bestimmt. Nach der Aktivierung von Linie c ist der Schnittpunkt weiterhin in der *Queue*, die Linien a und b sind allerdings nicht mehr benachbart.

Modifizieren Sie den Algorithmus so, dass er nur noch $O(n)$ Platz für die Ereignispunkte benötigt.

Musterlösung:
Die Lösung beruht auf der Tatsache, dass die an einem Schnittpunkt beteiligten Linien unmittelbar vor der Bearbeitung des Schnittpunktes in der Liste aktiver Kanten benachbart sein müssen. Damit können Schnittpunkte zeitweise verworfen werden, die zu momentan nicht mehr benachbarten Linien gehören. Dies sei durch folgendes Beispiel illustriert:



Folgt man dem Beispiel, so kann bei Aktivierung von Line c Schnittpunkt $a \times b$ verworfen werden. Erst bei Abarbeitung von Schnittpunkt $a \times c$ wird er wieder eingefügt.
Die benötigte Überprüfung ist im Allgemeinen sowieso nötig und hat somit keinen Einfluss auf die Laufzeit des Algorithmus. Durch diese Änderung kann sich zwischen jeweils zwei aktiven Linien nur jeweils ein aktiver Schnittpunkt in der *Queue* befinden. Da es maximal $n - 1$ benachbarte aktive Linienpaare geben kann, enthält die *Queue* insgesamt nur die geforderten $O(n)$ Ereignispunkte.

Aufgabe 6 (*Analyse+Entwurf:* Graham's Scan)

Betrachten Sie den *Graham's Scan* Algorithmus zur Bestimmung der konvexen Hülle einer Punktemenge $P \in \mathbb{R}^2$ mit $|P| = n$. In der Vorlesung wurde eine lexikographische Sortierung der Punkte vorgeschlagen, mit der Folge dass die obere und untere Hülle getrennt berechnet werden mussten.

a) Geben Sie eine geeignetere Sortierung der Punkte an, so dass die gesamte konvexe Hülle in einem Durchlauf berechnet werden kann.

b) Zeigen Sie, dass der *Graham's Scan* Algorithmus nicht für jede beliebige Sortierung der Punkte eine korrekte konvexe Hülle liefert (Randfälle ausgenommen).

c) Zeigen Sie anhand eines Beispiels, dass es möglich ist, dass der *Graham's Scan* Algorithmus p schon zur Hülle hinzugefügte Punkte hintereinander verwirft für beliebig großes p.

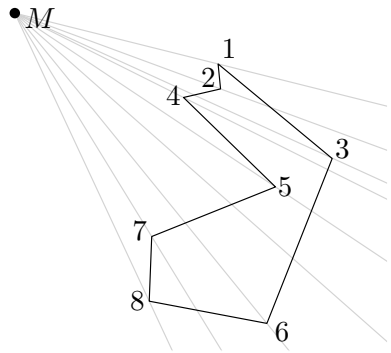
d) Eine Schneidemaschine bringt Stoffe anhand eines Schnittmusters in die gewünschte Form. Ein Schnittmuster ist dabei durch ein Polygon aus n Ecken definiert. Die Schneidemaschine kann beliebige konvexe Stoffe bearbeiten.

   Entwerfen Sie einen Algorithmus, der den minimal möglichen Verschnitt für ein gegebenes Stoffmuster in Linearzeit berechnet. Sie können davon ausgehen, dass die Ecken des Polygons als geschlossener Kantenzug vorliegen.

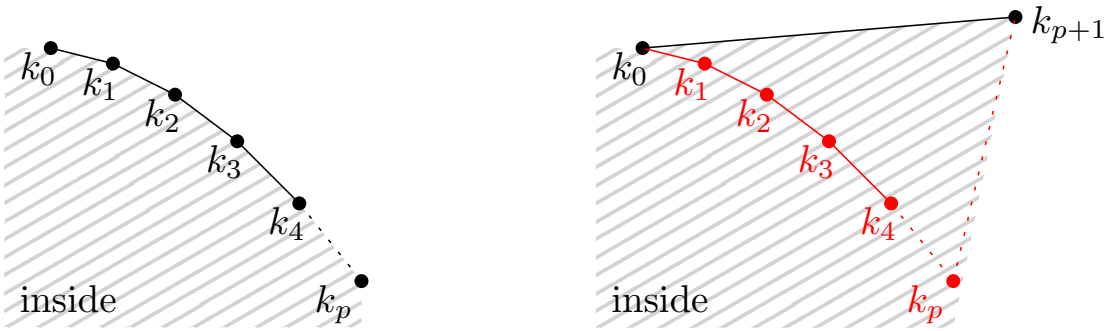
Musterlösung:

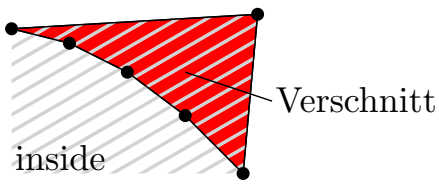
a) Eine zirkuläre Sortierung der Punkte um einen Mittelpunkt innerhalb der konvexen Hülle erfüllt diese Anforderung. Als Startpunkt für *Graham's Scan* wählt man einen Punkt der sicher auf der Hülle liegt, z.B. den Punkt mit kleinster x Koordinate. Zu berücksichtigende Sonderfälle treten auf, wenn sich Punkte in der gleichen Richtung vom Mittelpunkt aus gesehen befinden. Hier müssen die inneren vor den äußeren Punkten abgearbeitet werden.

b) Betrachte eine zirkuläre Sortierung der Punkte um einen Mittelpunkt außerhalb der konvexen Hülle. Wie im folgenden Bild zu sehen, springt die Reihenfolge der Punkte wild umher, wenn sie zirkulär um M sortiert werden.



c) Das geforderte Verhalten tritt auf, wenn beim Scan für p Punkte hintereinander eine 'Rechtsdrehung' stattfindet und anschließend eine 'Linksdrehung', so dass der $p+1$-te Punkt über dem ersten der Reihe liegt. Folgendes Bild veranschaulicht dies.



Links ist der Zustand nach Scan des p-ten Punktes zu sehen, rechts der Zustand nach Scan des $p+1$-ten Punktes. Die Punkte k_1 bis k_p wurden aus der vorläufigen konvexen Hülle entfernt.

d) Der geforderte Algorithmus ist ein modifizierter *Graham's Scan*. Die Ecken des Schnittmusters bilden die Eingabepunkte für den Algorithmus. Da sie schon sortiert vorliegen, entfällt der teuerste Schritt von *Graham' Scan*. Der Rest arbeitet in Linearzeit. Der Verschnitt wird bestimmt, indem jedes Mal, wenn eine 'Linksdrehung' beim Scan auftritt, die zusätzliche Fläche (im Bild rot) berechnet und über den gesamten Lauf des Algorithmus aufsummiert wird.