Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Laupichler
Hans-Peter Lehmann

5. Übungsblatt zu Algorithmen II im WS 2022/2023

http://algo2.iti.kit.edu/AlgorithmenII_WS22.php
{sanders, moritz.laupichler, hans-peter.lehmann}@kit.edu

Musterlösungen

Aufgabe 1 (*Kleinaufgaben: Parallele Algorithmen*)

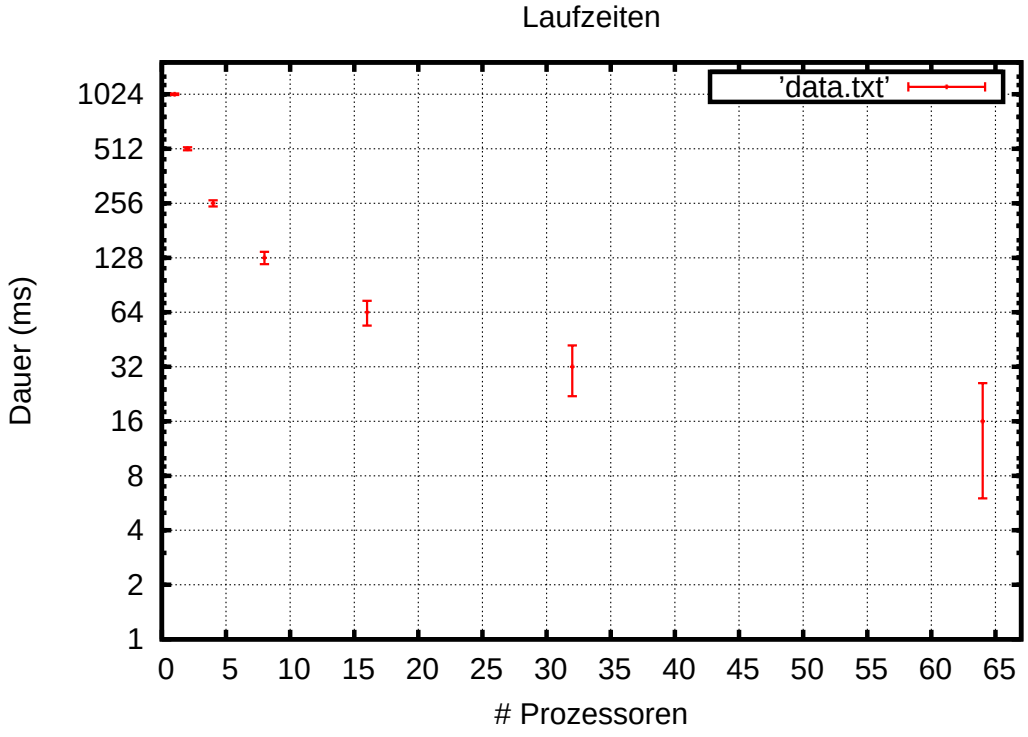
a) Gegeben sei ein paralleler vergleichsbasierter Sortieralgorithmus zum Sortieren von n komplexen Objekten auf p Prozessoren mit einer Laufzeit von

$$T(p) := \Theta(\frac{n^2 \log^2 n}{p^2}).$$

Geben Sie den absoluten *speed-up* und die *efficiency* an.

b) Wie muss in der vorherigen Teilaufgabe die Prozessorzahl p mit der Eingabegröße n wachsen, damit der absolute *speed-up* konstant bleibt?

c) Sie haben für einen parallelen Algorithmus folgende Laufzeiten bei unterschiedlicher Prozessorzahl gemessen. Was können Sie über die Skalierung dieses Algorithmus aussagen?



Musterlösung:

a) Die besten sequentiellen vergleichsbasierten Sortierverfahren benötigen $T_{seq} = \Theta(n \log n)$ Laufzeit. Damit ergibt sich für den absoluten *speed-up*

$$S(p) = \frac{T_{seq}}{T(p)} = \frac{\Theta\left(n \log n\right)}{\Theta\left(\frac{n^2 \cdot \log^2 n}{p^2}\right)} = \Theta\left(\frac{p^2}{n \log n}\right).$$

Mit der Definition der *efficiency* ergibt sich

$$E(p) = \frac{S(p)}{p} = \frac{\Theta\left(\frac{p^2}{n \log n}\right)}{p} = \Theta\left(\frac{p}{n \log n}\right).$$

b) Für einen konstanten *speed-up* unabhängig von der Prozessoranzahl muss gelten

$$S(p) = \Theta(\frac{p^2}{n \log n}) \stackrel{!}{=} \Theta(1)$$

und damit muss die Anzahl Prozessoren mit der Eingabegröße nach $p = \Theta(\sqrt{n \log n})$ wachsen.

c) Ignoriert man die Fehlerbalken, so weisst der Algorithmus eine Halbierung der Laufzeit bei Verdopplung der Prozessoren auf. Der relative *speed-up* ist also

$$S_{rel}(p) = \frac{T(1)}{T(p)} = p.$$

Der absolute *speed-up* skaliert auch linear mit p, über den genauen Faktor und über die Skalierung mit n kann man keine Aussage treffen. Die *efficiency* ist unabhängig von p.

Aufgabe 2 (*Entwurf+Analyse: CRCW und CREW Modelle*)

Gegeben sei ein Array $a[\cdot]$ im gemeinsamen Speicher, der n Zahlen hält.

a) Beschreiben Sie einen möglichst schnellen parallelen Algorithmus, der überprüft, ob mindestens eine der Zahlen durch 7 teilbar ist. Gehen Sie von $p = n$ Prozessoren und dem CRCW *common* Modell aus. Außerdem sei die Teilbarkeit in $O(1)$ prüfbar.

b) Wie ändert sich die Laufzeit, wenn Sie nur noch $p < n$ Prozessoren zur Verfügung haben?

c) Wieviel Zeit würde Ihr Algorithmus im CREW Modell benötigen (wieder $p = n$ Prozessoren)?

d) Geben Sie den absoluten *speed-up* und die *efficiency* für die vorherigen Teilaufgaben an.

e) Im Folgenden soll berechnet werden, wieviele der Zahlen in $a[\cdot]$ durch eine andere Zahl in $a[\cdot]$ (nicht durch sich selbst!) teilbar sind. Sie haben das CRCW Modell und $p = n^2$ Prozessoren zur Verfügung.

Musterlösung:

a) Die Variable zum Speichern des Resultats wird mit `result = false` initialisiert. Jeder Prozessor p_i überprüft für eine Zahl $a[i]$, ob diese durch 7 teilbar ist. Ist dies der Fall, setzt der Prozessor `result = true`. Ansonsten macht er nichts. Der Algorithmus läuft in $T(p) = O(1)$.

b) Jeder Prozessor muss sich seriell um n/p Speicherstellen kümmern. Ansonsten läuft der Algorithmus gleich ab und benötigt daher $T(p) = O(n/p)$ Zeit.

c) Im CREW Modell kann nicht mehr parallel auf eine Speicherstelle geschrieben werden. In einer einfachen Lösung würde für alle Prozessoren je ein Schreibzugriff auf `result` nacheinander ausgeführt werden. Dies würde $O(n)$ Zeit benötigen.

Geschickter ist es, das Ergebnis für jede Zahl $a[i]$ in einem Array $b[\cdot]$ zu speichern - 0 für `false` und 1 für `true`. Dies geschieht in $O(1)$. Dann wird die Summe $\sum_{i=1}^{n} b[i]$ per Reduktion in $O(\log n)$ berechnet. Ist die Summe ungleich 0, so wird `result=true` gesetzt, sonst auf `false`. Die gesamte Laufzeit ist $T(p) = O(\log n) = O(\log p)$.

d) Der optimale sequentielle Algorithmus muss im schlimmsten Fall jede Zahl prüfen, benötigt also $T_{seq} = O(n)$. Damit ergibt sich für den absoluten *speed-up*

$$S(p) = \frac{T_{seq}}{T(p)} = \frac{O(n)}{T(p)} = \begin{cases} O(n) = O(p) & \text{für (a)} \\ O(p) & \text{für (b)} \\ O(n/\log n) = O(p/\log p) & \text{für (c)} \end{cases}$$

Mit der Definition von *efficiency* ergibt sich

$$E(p) = \frac{S(p)}{p} = \begin{cases} O(1) & \text{für (a)} \\ O(1) & \text{für (b)} \\ O(1/\log n) = O(1/\log p) & \text{für (c)} \end{cases}$$

e) Berechne Hilfsarray $b[\cdot]$. Es wird $b[j]$ = `true` gesetzt, wenn $a[j]$ von einer Zahl in $a[i]_{i \neq j}$ geteilt wird. Ansonsten wird $b[j]$ = `false` gesetzt. Analog zur ersten Teilaufgabe, können je n Prozessoren für jedes i die Prüfungen auf Teilbarkeit durch $a[i]$ übernehmen. Dabei setzt der j-te Prozessor im Falle der Teilbarkeit $b[j]$ = `true`. Dies benötigt Zeit $O(1)$. Mit n^2 Prozessoren und n Einträgen in $b[\cdot]$ können alle Einträge in $O(1)$ berechnet werden. Anschließend wird über die Elemente in $b[\cdot]$ summiert, mit n Prozessoren und Reduktionsschema ist dies in $O(\log n)$ möglich. Die gesamte Laufzeit ist also $T(p) = O(\log n) = O(\log \sqrt{p})$.

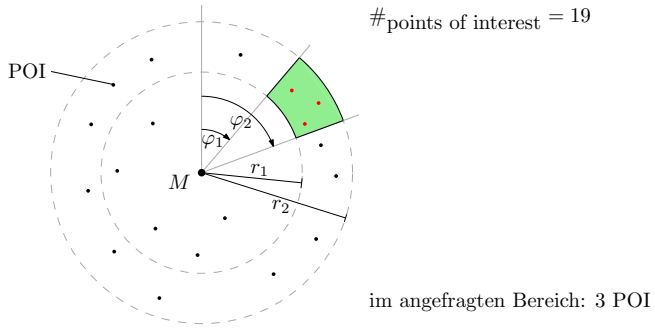
Aufgabe 3 (*Kleinaufgaben: Geometrie-Entwurf*)

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.

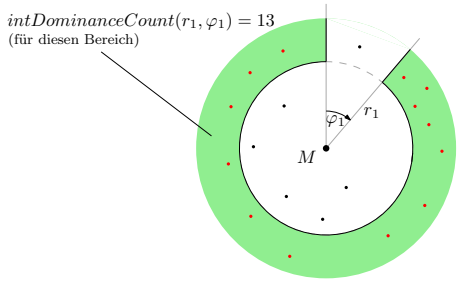
Das erzeugte Polygon ist offensichtlich geschlossen, wenn auch der letzte Punkt mit dem ersten Punkt verbunden wird. Das Polygon ist außerdem kreuzungsfrei, da es monoton in einer Richtung aufgebaut wird (im Uhrzeigersinn und von innen nach nach außen).

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:

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

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.

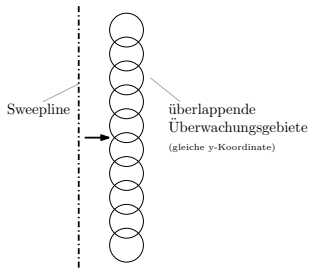
a) Zeigen Sie, dass ein *Sweepline*-Algorithmus, der einfach alle (im Rahmen der Algorithmenausführung) aktiven Sensorknoten auf Überlappung prüft, in quadratischer Laufzeit resultieren kann, auch wenn die Ausgabekomplexität (Anzahl überlappender Knotenpaare) linear ist.

b) Überlegen Sie, wie Sie dennoch einen *Sweepline*-Algorithmus verwenden können, um das Problem in Linearzeit zu lösen (exklusive das Sortieren am Anfang), 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. Nach Annahme in der Aufgabenstellung vergleicht man alle aktiven Knoten. 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.

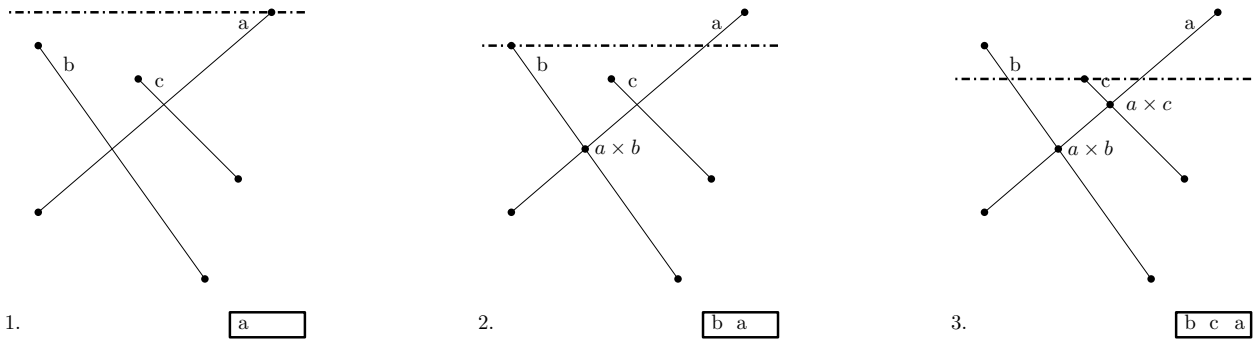


b) Wie in der vorherigen Teilaufgabe gezeigt, besteht das Problem darin, dass gleichzeitig viele Knoten aktiv sein können. Ein Trick zur Umgehung dieses Problems ist die Verwendung von Buckets. Verwaltet man die Menge aller aktiven Elemente in einer sortieren Liste an Buckets (augeteilt anhand der y-Koordinate, 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*)

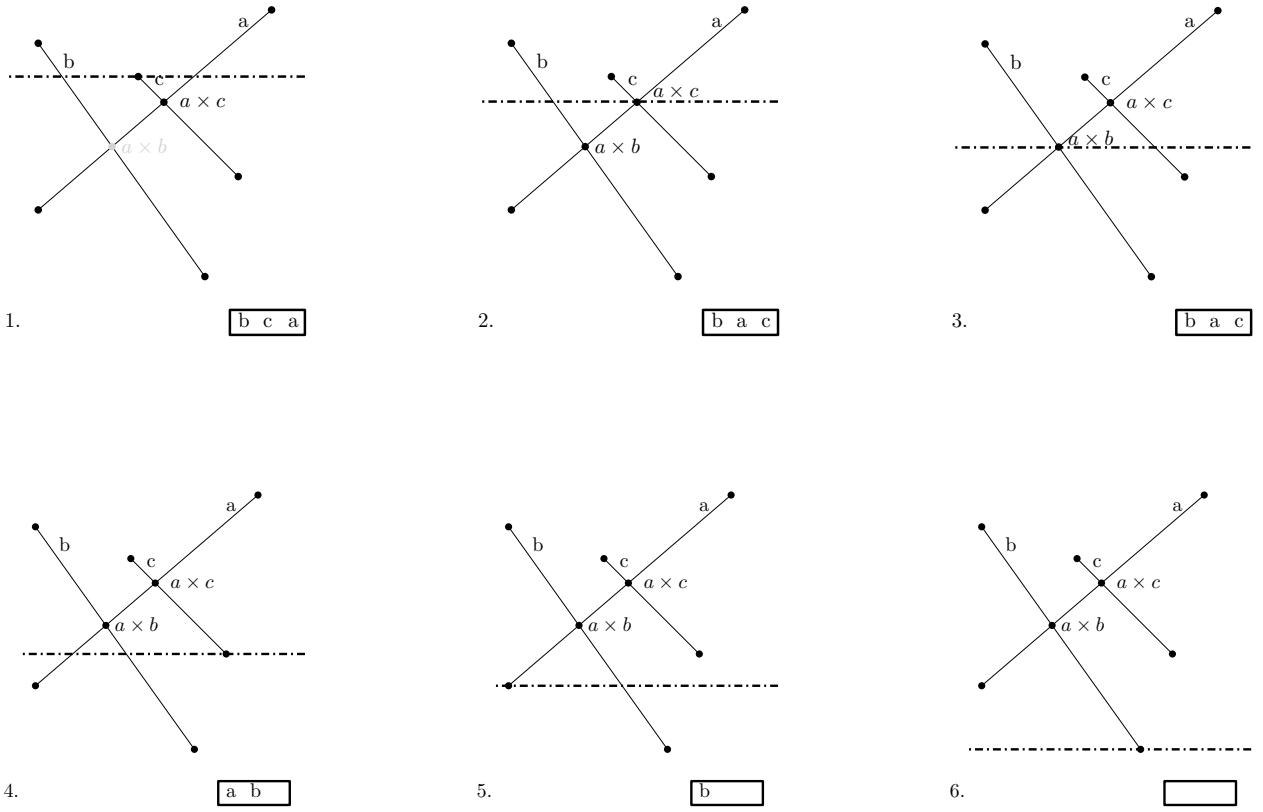
In der Vorlesung wurde ein *Sweepline*-Algorithmus zur Bestimmung der Schnitte zwischen n Liniensegmenten behandelt. Der Algorithmus arbeitet eine Liste an Ereignispunkten (Schnittpunkte, Linienanfänge und Linienenden), in Form einer nach der y-Position sortierten *Queue*, ab. Gleichzeitig führt er eine 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.

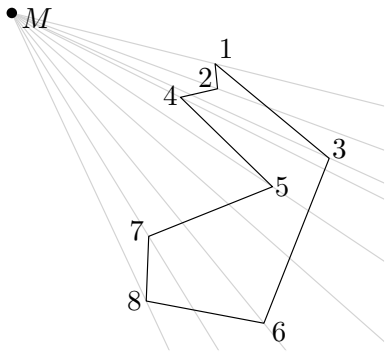
d) Eine Schneidemaschine bringt Stoffe anhand eines Schnittmusters in die gewünschte Form. Ein Schnittmuster ist dabei durch ein Polygon aus n Ecken definiert, das selbst nicht notwendigerweise konvex ist. Die Schneidemaschine kann beliebige konvexe Stoffe bearbeiten.

   Entwerfen Sie einen Algorithmus, der den minimal möglichen Verschnitt für ein gegebenes Stoffmuster in Linearzeit berechnet. Die Ausgabe des Algorithmus soll also die Gesamtfläche des Verschnitts sein, welcher anfällt, wenn ein optimales konvexes Stoffstück zum gegebenen Schnittmuster verarbeitet wird. Sie können davon ausgehen, dass die Ecken des Polygons als geschlossener Kantenzug vorliegen.

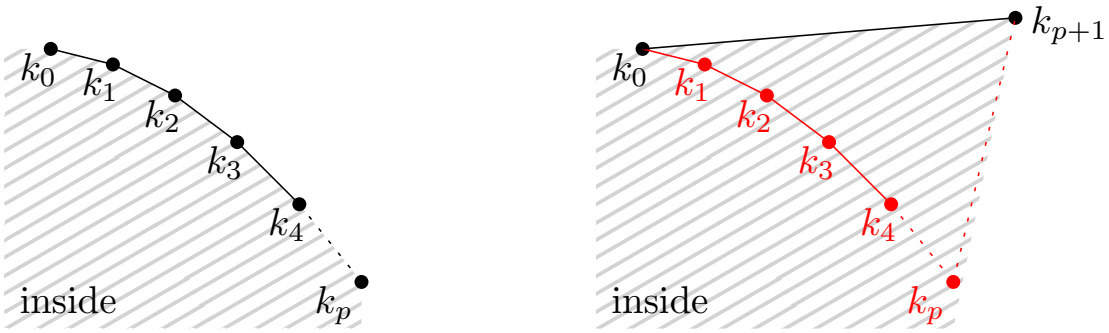
Musterlösung:

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.