Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Laupichler
Nikolai Maas

4. Übungsblatt zu Algorithmen II im WS 2023/2024

https://algo2.iti.kit.edu/AlgorithmenII_WS23.php
{sanders, moritz.laupichler, nikolai.maas}@kit.edu

Musterlösungen

Aufgabe 1 (*Kleinaufgaben: Laufzeiten*)

Sei $f(n, k)$ die Laufzeit eines Algorithmus mit n der Eingabegröße des Problems und k ein beliebiger Parameter. Geben Sie an, welche der folgenden Laufzeiten ein Problem *fixed-parameter-tractable* machen. Begründen Sie Ihre Antwort jeweils kurz.

- $f_1(n, k) = 3k^2n^2$
- $f_2(n, k) = n^k \cdot k^2 \cdot \sqrt{n^e}$
- $f_3(n, k) = 3n^2 + 2nk$
- $f_4(n, k) = e^k \cdot \sqrt{n}$
- $f_5(n, k) = e^n \cdot \sqrt{k}$
- $f_6(n, k) = k^3 \cdot \log n^2$

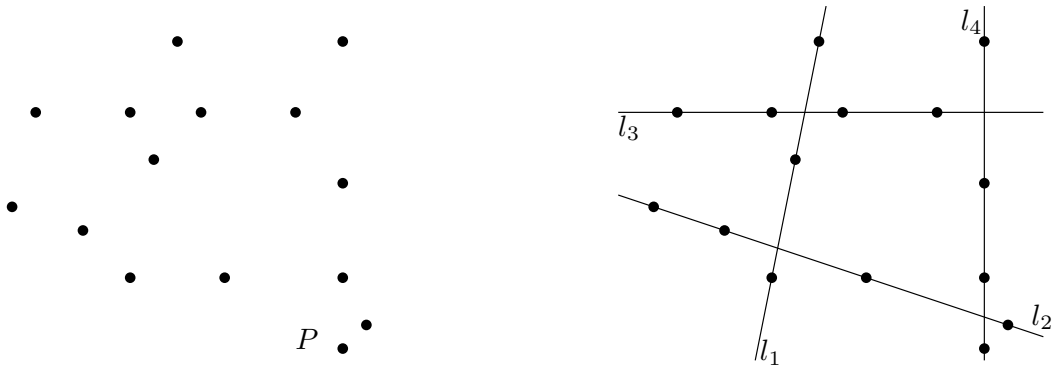
Musterlösung:

b) Ein Problem heißt *fixed parameter tractable*, falls ein Algorithmus zu dessen Lösung existiert, dessen Laufzeit durch $O(f(k) \cdot poly(n))$ abschätzbar ist. Dabei ist $poly(n)$ ein Polynom nur in n und $f(k)$ eine beliebige Funktion nur in k. Damit ergibt sich für die angegebenen Probleme:

- $f_1(n,k) = 3k^2n^2$
  Das beschriebene Problem ist *fixed parameter tractable*. $f_1(n,k)$ ist polynomiell in n (n^2) und davon unabhängig abhängig in k (k^2).
- $f_2(n,k) = n^k \cdot k^2 \cdot \sqrt{n^e}$
  Das beschriebene Problem ist nicht *fixed parameter tractable*. $f_2(n,k)$ ist durch $n^k \cdot \sqrt{n^e}$ zwar polynomiell in n, aber dies ist nicht unabhängig von k.
- $f_3(n,k) = 3n^2 + 2nk$
  Das beschriebene Problem ist *fixed parameter tractable*. Durch $poly(n) = n^2$ und $f(k) = k$ lässt sich $f_3(n,k)$ abschätzen ($3n^2 + 2nk = O(n^2k)$.
- $f_4(n,k) = e^k \cdot \sqrt{n}$
  Das beschriebene Problem ist *fixed parameter tractable*. $f_4(n,k)$ ist polynomiell in n ($\sqrt{n}$) und davon unabhängig abhängig von k (e^k).
- $f_5(n,k) = e^n \cdot \sqrt{k}$
  Das beschriebene Problem ist nicht *fixed parameter tractable*. $f_5(n,k)$ ist nicht polynomiell in n (e^n).
- $f_6(n,k) = k^3 \cdot \log n^2$
  Das beschriebene Problem ist *fixed parameter tractable*, da $f_6(n,k)$ polynomiell in n ist (da $\log n^2 = O(n) = O(poly(n))$) und davon unabhängig von k abhängig (k^3).

Aufgabe 2 (*Analyse:* line shooting *Problem*)

Gegeben seien n Punkte $P \subset \mathbb{R}^2$ in der Ebene sowie eine Zahl $k > 0$. Das *line shooting* Problem besteht darin zu bestimmen, ob es eine Menge L von k Geraden gibt, so dass jeder Punkt in P von mindestens einer dieser Geraden getroffen wird. Eine Probleminstanz wird durch das Tupel (P, k) charakterisiert.
In den Bildern sehen Sie links eine Punktmenge P und rechts eine mögliche Lösung für $(P, 4)$.



a) Begründen Sie kurz, warum eine Gerade l, die mehr als k Punkte trifft, Teil einer Lösung für die Probleminstanz (P, k) sein muss bei beliebigem P.

b) Begründen Sie kurz, warum die Instanz $(P, 3)$ des *line shooting* Problems keine Lösung besitzt mit P wie in obiger Abbildung.

c) Geben Sie einen Algorithmus an, der eine Instanz (P, k) des *line shooting* Problems exakt löst für beliebiges P. Bauen Sie dazu einen Suchbaum mit beschränkter Tiefe auf, der alle Kombination von k Geraden, die jeweils mindestens zwei Punkte treffen, generiert. Die Suchbaumtiefe und der Verzweigungsgrad sollen dabei polynomiell in k, der Aufwand pro Knoten polynomiell in $n = |P|$ sein.

   Hinweise: Verwalten Sie in jedem Knoten des Suchbaumes k Einträge, die jeweils bis zu zwei Punkte halten können (und damit eine Gerade definieren). Ein Suchbaum der Höhe $O(k)$ genügt.

d) Zeigen Sie, dass das *line shooting* Problem *fixed parameter tractable* bezüglich k ist. Geben Sie dazu die asymptotische Laufzeit Ihres Algorithmus in Abhängigkeit von n und k an.

Musterlösung:

a) Allgemein gilt, dass eine Gerade, die $K > k$ Punkte trifft, Teil einer Lösung für (P, k) sein muss. Wäre diese Gerade nicht Teil der Lösung, so würde man K andere Geraden benötigen, um diese Punkte zu treffen. Da $K > k$ wäre dies für eine Lösung von (P, k) nicht zugelassen.

b) In der rechten Abbildung sieht man drei Geraden, die jeweils 4 Punkte treffen (l_2, l_3, l_4). Diese müssen nach obiger Begründung Teil einer Lösung von $(P, 3)$ sein. Da diese Geraden aber nicht ausreichen, um alle Punkte von P treffen, existiert keine Lösung für $(P, 3)$.

c) Idee: Der Algorithmus baut systematisch alle Möglichkeiten auf, k Geraden durch je zwei Punkte aus P zu repräsentieren und prüft, ob diese Geraden alle Punkte treffen. Dies geschieht mit Hilfe eines beschränkten Suchbaumes wie im Folgenden beschrieben.

Wir betrachten die Punkte P in einer festen Reihenfolge $P = \langle p_1, p_2, \ldots, p_n \rangle$ und beginnen, wie im Hinweis angegeben, bei einem leeren Knoten mit k Einträgen. Da jeder Knoten durch eine der Geraden – definiert durch jeden der gefüllten k Einträge – überdeckt werden muss, stellt die Wahl der Reihenfolge keine Einschränkung da. Für einen beliebigen Knoten w_i der Suchbaumtiefe i können wir bis zu k Nachfolger der Suchbaumtiefe $i + 1$ generieren. Der j-te Nachfolger von w_i ($w_{i+1,j}$) wird durch eine Kopie von w_i erzeugt, bei der zusätzlich der nächste zu betrachtende Punkt in den Eintrag an Stelle j eingefügt wird. Besteht der Eintrag an Stelle j schon aus zwei Punkten, so wird $w_{i+1,j}$ verworfen und w_i erhält einen Nachfolger weniger. Wird der Eintrag an Stelle j komplett gefüllt, so wird durch die beiden Punkte eine neue Gerade definiert und alle von ihr überdeckten Knoten aus der Liste der zu bearbeitenden Punkte entfernt (für den aktuellen Teilbaum). Auf diese Weise werden alle Möglichkeiten getestet, k verschiedene Geraden aus den Punkten zu erzeugen. Jeder Blattknoten auf der Suchbaumtiefe $2k$ definiert nun k verschiedene Geraden. Sind bei einem dieser Knoten alle Punkte überdeckt, so kann eine Erfolgsmeldung ausgegeben werden.

d) Der Verzweigungsgrad des Suchbaums ist höchstens k, da es in jedem Schritt nur k mögliche Einträge zu füllen gibt. Die Suchbbaumtiefe ist höchstens $2k$, da nach Auswahl von $2k$ Knoten alle Einträge gefüllt sind. Damit hat der Suchbaum $O(k^{2k})$ Knoten. Wird kein Eintrag voll, so werden beim Erzeugen eines Knotens Kosten von $O(1)$ erzeugt. Ist dagegen eine Überprüfung der verbleibenden Knoten nötig, so entstehen Kosten von $O(n)$ für das Überprüfen der maximal $O(n)$ verbleibenden Punkte. Damit ist $O(\frac{k^{2k}}{2} \cdot n)$ eine obere Schranke für die Gesamtlaufzeit. Daraus folgt, dass das *line shooting* Problem *fixed parameter tractable* ist.

Aufgabe 3 (*Kleinaufgaben: Parallele Algorithmen*)

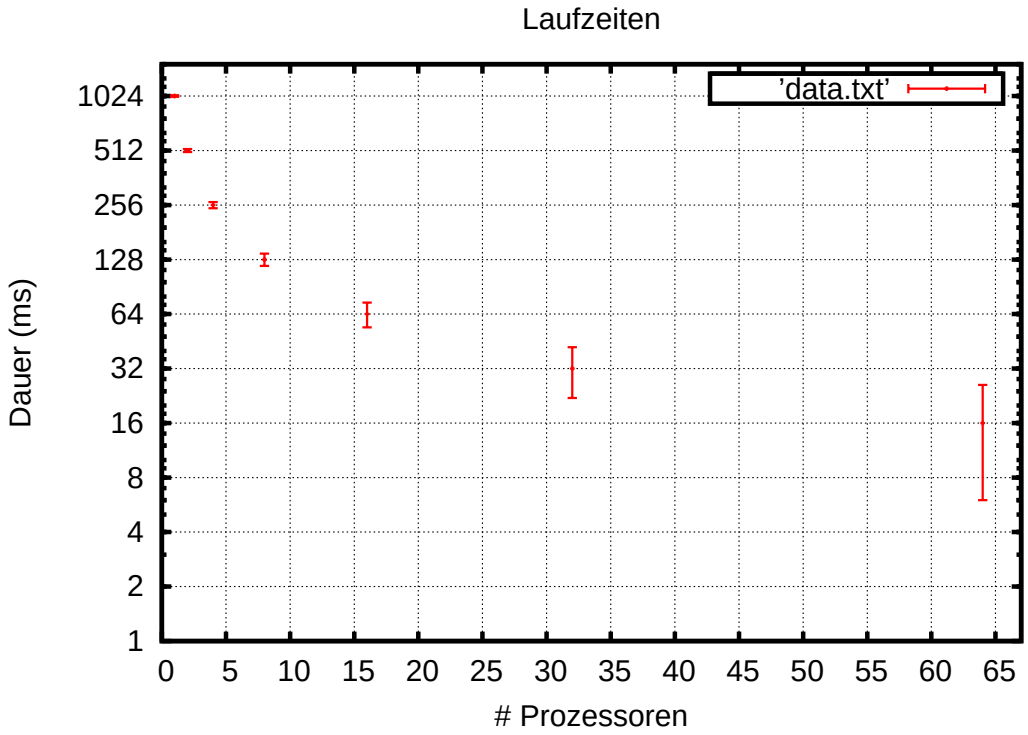
a) Gegeben sei ein paralleler vergleichsbasierter Sortieralgorithmus zum Sortieren von n komplexen Objekten auf p Prozessoren mit einer Laufzeit von

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

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 4 (*Entwurf+Analyse:* `findif`-*Anweisung*)

Gegeben sei ein Array $a[\cdot]$ im verteilten Speicher der n Objekte hält. Gesucht ist ein Algorithmus, der eine parallele `findif` Anweisung auf $a[\cdot]$ ausführt. Die Anweisung sortiert die Elemente von $a[\cdot]$ anhand eines Prädikats $pred(\cdot)$, so dass Elemente, die das Prädikat erfüllen, vorne stehen. Die relative Ordnung der Elemente untereinander soll dabei erhalten bleiben.

Bsp.: `findif( {1,4,9,7,3},` *is_bigger_than_3* `) = {4,9,7,1,3}`

a) Beschreiben Sie einen Algorithmus, der eine parallele `findif` Anweisung auf $a[\cdot]$ möglichst schnell ausführt. Sie haben $p = n$ Prozessoren zur Verfügung.

b) Untersuchen Sie die Laufzeit der Anweisung für den Fall, dass $p = n$ Prozessoren zur Verfügung stehen und das Prädikat in $T(n) = O(1),\ O(\log n)$ bzw. $O(n)$ ausgewertet werden kann.

c) Wie verhalten sich die Laufzeiten, wenn Sie nur noch $p < n$ Prozessoren zur Verfügung haben?

Musterlösung:

a) Prozessor p_i prüft Bedingung $pred(a[i])$ und schreibt das Resultat als 0 (falsch) oder 1 (wahr) in ein neues Array $s[\cdot]$ an Stelle i. Anschließend wird die Präfixsumme über $s[\cdot]$ gebildet. Aus diesen Werten kann jeder Prozessor p_i die neue Position für sein Datenelement $a[i]$ ableiten:

- $a[i] \rightarrow a[s[i] - 1]$, wenn $a[i]$ das Prädikat erfüllt,
- $a[i] \rightarrow a[s[n-1] + i - s[i]]$, wenn $a[i]$ das Prädikat nicht erfüllt.

Prozessoren werden von 0 durchnummeriert.

b) Die Ausführungszeit beträgt $O(T(n))$ für die Berechnung von $s[\cdot]$, $O(\log n)$ für die Berechnung der Präfixsumme und $O(1)$ für die Umsortierung. Gesamt ergeben sich die Laufzeiten in Abhängigkeit von $T(n)$ zu $O(\log n)$, $O(\log n)$ bzw. $O(n)$, wobei $p = n$ gilt.

c) Stehen weniger als n Prozessoren zur Verfügung, muss man die Arbeit für jeweils n/p Objekte von einem Prozessor ausführen lassen. Es ergeben sich folgende Laufzeiten für die drei Schritte $O(n/p \cdot T(n))$, $O(n/p + \log p)$ und $O(n/p)$. Insgesamt ergeben sich die Laufzeiten wieder in Abhängigkeit von $T(n)$ zu $O(n/p + \log p)$, $O(n/p \cdot \log p)$ bzw. $O(n^2/p + \log p)$.

Aufgabe 5 (*Entwurf+Analyse: Assoziative Operationen*)

Gegeben sei ein Array A im gemeinsamen Speicher bestehend aus n Objekten vom Typ X. Auf den Objekten sei ein Operator $\odot$ definiert. Es wird nach dem Ergebnis von $\bigodot_{i=1}^{n} a_i$ gesucht.

a) Sei $X = \mathbb{R}^2$ und der Operator definiert als

$$(x_1, x_2) \odot (y_1, y_2) := (x_1 y_1, x_2 + y_2)$$

Zeigen Sie, dass der Operator $\odot$ assoziativ ist.

b) Beschreiben Sie einen schnellen parallelen Algorithmus, der $\bigodot_{i=1}^{n} a_i$ berechnet, und geben Sie dessen Laufzeit $T(n, p)$ an.

c) Nun sei $\odot$ wie folgt definiert: Sei X das Alphabet $\{(,)\}$ und X^* die Menge aller möglichen Zeichenketten über X. Die Operation $x \odot y$ für $x, y \in X^*$ verknüpfe beide Zeichenketten und schiebe alle öffnenden Klammern nach links, alle schließenden Klammern nach rechts (Bsp.: $()(() \odot (()) = ((((()))))$.

Können Sie den selben Lösungsansatz wie in der vorherigen Teilaufgabe verwenden? Falls nein, geben Sie einen neuen parallelen Algorithmus an. Wie lange dauert die Ausführung?

Musterlösung:

a) Der Operator ist offensichtlich assoziativ, da beide Koordinaten unabhängig voneinander sind und die ausgeführte Addition bzw. Multiplikation auf reellen Zahlen assoziativ ist.

b) Der Operator $\odot$ ist assoziativ und in konstanter Zeit ausführbar. Daher kann $\bigodot_{i=1}^{n} a_i$ mit Hilfe des Reduktionsschemas in $T(n, p) = O(n/p + \log p)$ berechnet werden. Für $p = n$ liegt die Ausführungszeit in $T(n, p) = O(\log n)$.

c) Auch hier ist $\odot$ assoziativ, benötigt aber Zeit proportional zur Länge beider Operanden (ein Scan über beide Operanden liefert die Anzahl öffnender und schließender Klammern, die anschließend geschrieben werden können). Das Reduktionsschema ist weiterhin anwendbar, die Laufzeit erhöht sich allerdings insgesamt zu $T(n, p) = O\left(\sum_{i=0}^{\log p} \left(2^i \frac{n}{p} + 1\right)\right) = O(p \cdot n/p + \log p)$ bzw. zu $T(n, p) = O(n)$ für $p = n$.

Ein schnellerer Lösungsansatz verwendet paralleles Sortieren, indem öffnende vor schließende Klammern sortiert werden. Die gesamte Eingabe lässt sich in Zeit $T(n, p) = O(\frac{n \log n}{p} + \log^2 p)$ bzw. $T(n, p) = O(\log^3 n)$ für $p = n$ sortieren (siehe Anmerkung zu parallelem Sortieren in VL).

Ein alternativer Ansatz wäre es, zunächst in einem Vorverarbeitungsschritt aus jedem Objekt ein Tupel zu erzeugen, das die Anzahl der öffnenden und schließenden Klammern enthält. Dann kann eine normale Reduktion durchgeführt werden, bei der eine komponentenweise Addition als Operation verwendet wird.

Aufgabe 6 (*Analyse: Externer Stack*)

In der Vorlesung wurde eine Implementierung von *Stack* als externe Datenstruktur vorgestellt. Eine äquivalente Implementierung besitzt folgende Struktur: Im Speicher wird ein Puffer P der Größe $2B$ gehalten – B sei die Blockgröße beim Zugriff auf externen Speicher. Der Puffer ist in Form eines (internen) Stacks organisiert und enthält die neuesten gespeicherten Elemente. Folgende Operationen sind für die externe Datenstruktur definiert:

`pop` Falls P nicht leer, entferne das neueste Element aus P. Ansonsten, lese einen Block ein, um die Hälfte von P zu füllen bevor `pop` auf P ausgeführt wird.

`push` Falls P nicht voll, füge das neue Element direkt zu P. Ansonsten, schreibe die ältere Hälfte von P in den externen Speicher und verschiebe die aktuellere Hälfte an diese Stelle im Speicher. Anschließend führe ein `push` auf P aus.

Für die Analyse können Sie davon ausgehen, dass ein Block B Elemente des Stacks halten kann.

a) Zeigen Sie, dass die Operationen `push` und `pop` amortisiert $O(1/B)$ I/O-Operationen benötigen.

b) Warum genügt es nicht, nur einen Puffer mit Größe B zu verwenden?

Musterlösung:

a) Betrachte die minimale Anzahl an Operationen (`push` oder `pop`) bis zur nächsten I/O-Operation: Nach einer I/O-Operation ist die Hälfte des Puffers leer – bei einem `push` auf den vollen Puffer wurde die Hälfte in den externen Speicher verlagert bzw. bei einem `pop` auf einen leeren Puffer wurde der halbe Puffer aufgefüllt. Von diesem Zustand ausgehend werden mindestens B Operationen einer Art ausgeführt, bevor erneut ein I/O-Zugriff erfolgt – entweder, um bei einem `push` Daten in den externen Speicher zu schreiben, da der Puffer voll ist, oder um bei einem `pop` Daten aus dem externen Speicher zu laden, weil der Puffer leer ist.

b) Bei nur einem Puffer der Größe B könnte man sich folgende maximal schlechte Folge an Operationen überlegen: $B + 1$ `push` Operationen, gefolgt von einer Reihe von je 2 `pop` und 2 `push` Operationen. Jeweils die zweite dieser Anweisungen löst eine I/O-Operation aus, da das `pop` auf einem leeren und das `push` auf einem vollen Puffer stattfindet. Amortisiert ergeben sich $O(1)$ I/O-Operationen.

Aufgabe 7 (*Entwurf+Analyse: Telekommunikationsgesellschaft*)

Eine Telekommunikationsgesellschaft beauftragt Sie eine Anwendung zu schreiben, die monatlich die k Kunden bestimmt, bei denen sich die Rechnung im Vergleich zum Vormonat am meisten verändert hat. Diese Kunden will sich die Telekommunikationsgesellschaft noch einmal genau anschauen, um ihnen eventuell einen neuen Vertrag anzubieten.
Die zu bearbeitenden Daten werden Ihnen auf (langsamen) Bandspeichern zur Verfügung gestellt. Sie erhalten eine Liste mit den aneinandergefügten Datensätzen jeder Zweigstelle ihres Auftraggebers für den aktuellen Monat. Außerdem haben Sie eine entsprechende Liste für den Vormonat zur Verfügung. Gespeichert sind jeweils Tupel $(Kundennummer, Kosten)$.

a) Geben Sie einen Algorithmus an, der die geforderte Aufgabe erfüllt. Geben Sie außerdem die Laufzeit Ihres Algorithmus an und begründen Sie diese. Sie können davon ausgehen, dass die k zu bestimmenden Kunden in den Hauptspeicher passen.

b) Seien nun die k zu bestimmenden Kunden zu groß, um im Hauptspeicher gehalten zu werden. Ändern Sie Ihren Algorithmus so ab, dass er mit der erhöhten Datenmenge zurecht kommt. Geben Sie die Laufzeit Ihres neuen Algorithmus an und begründen Sie diese.

Musterlösung:

a) In einem ersten Schritt werden beide Listen nach aufsteigender Kundennummer sortiert mit externem MergeSort. Anschließend scannt der Algorithmus linear über beide sortierte Listen M, N. Zu diesem Zweck wird für jede Liste ein Zeiger i_M bzw. i_N mit der aktuellen Position gespeichert und ein Teil der Liste mit Größe B im Speicher gehalten, der bei Bedarf durch den folgenden ersetzt wird. Beide Zeiger starten am Anfang der jeweiligen Liste. Stimmen die Kundennummern von $M[i_M]$ und $N[i_N]$ überein, wird die Differenz ihrer Kosten gebildet. Diese wird in einer lokalen Prioritätswarteschlange PQ gespeichert. Hat PQ nach dem Einfügen mehr als k Elemente, so wird das kleinste entfernt. Stimmen die Kundennummern M_{i_M} und N_{i_N} nicht überein, wird der Zeiger um eins erhöht, der auf den Eintrag mit der kleineren Kundennummer zeigt. Nachdem die kürzere von beiden Listen vollständig abgearbeitet wurde, enthält PQ die gesuchten Elemente.

   Die Laufzeit wird von den benötigten I/O-Operationen dominiert. Sei $n := |N| + |M|$ und H die Größe des Hauptspeichers. Sortieren benötigt $\Theta(\frac{n}{B} log_{\frac{H}{B}} \frac{n}{H})$, der lineare Scan über beide Listen $O(n/B)$ I/O-Operationen. Der gesamte Algorithmus ist also vom Sortieren dominiert.

   Für die Blockgröße B beim linearen Scan muss gelten: $B < (H - \text{maximaler Speicher für PQ})/2$, wobei H den verfügbaren Hauptspeicher angibt. Die Blockgröße beim MergeSort dürfte größer sein, da die PQ zu diesem Zeitpunkt noch leer ist. Dies ändert die Anzahl I/O-Operationen aber nur um einen konstanten Faktor.

b) Verwende eine externe Prioritätswarteschlange statt einer internen. Diese benötigt bis zu $O(\frac{n}{B} \log_{\frac{H}{B}} \frac{n}{H})$ I/O-Operationen, falls jeder Wert eingefügt werden muss (Werte löschen ist amortisiert kostenlos). Dies entspricht der Anzahl, die für das initiale Sortieren benötigt wird. Die Laufzeit ändert sich also nicht.

   Für die Blockgrößen gilt die gleiche Argumentation wie in der ersten Teilaufgabe.