Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Hans-Peter Lehmann, Daniel Seemaier

5. Übungsblatt zu Algorithmen II im WS 2021/2022

http://algo2.iti.kit.edu/AlgorithmenII_WS21.php
{sanders, hans-peter.lehmann, daniel.seemaier}@kit.edu

Musterlösungen

Aufgabe 1 (*Schlechter Zufall*)

Gegeben sei ein randomisierter Algorithmus badBit, der keine Eingabe liest und als Ausgabe zufällig mit Wahrscheinlichkeit p die Zahl 0 und mit Wahrscheinlichkeit $q = 1 - p$ die Zahl 1 liefert. Es sei $0 < p < 1$; der konkrete Wert von p sei aber unbekannt.
Entwerfen Sie einen randomisierten Algorithmus fairBit, der keine Eingabe liest und als Ausgabe immer zufällig eine der Zahlen 0 und 1 mit Wahrscheinlichkeit 1/2 liefert.
Was können Sie über die Laufzeit Ihres Algorithmus sagen?

Musterlösung:
Algorithmusidee:

```
1: function fairBit
2:     repeat
3:         x ← badBit()
4:         y ← badBit()
5:     until x ≠ y
6:     return x
7: end function
```

Korrektheit:
$\mathbb{P}(x = 1 | x \neq y) = \frac{\mathbb{P}(x=1 \ \wedge \ y=0)}{\mathbb{P}(x \neq y)} = \frac{pq}{2pq} = \frac{1}{2}$

Laufzeit:
Im Folgenden benutzen wir folgende Überlegung (für $z \in \mathbb{R}$ mit $0 < |z| < 1$):

$$\text{sei} \qquad R = \sum_{i=1}^{\infty} i \cdot z^{i-1} = \sum_{i=0}^{\infty} (i+1) \cdot z^i$$

$$\text{dann} \qquad Rz = \sum_{i=1}^{\infty} i \cdot z^i$$

$$\text{also} \qquad R(1-z) = R - Rz = \sum_{i=0}^{\infty} z^i = \frac{1}{1-z}$$

$$\text{also} \qquad R = \frac{1}{(1-z)^2}$$

Nun ist im Algorithmus die Wahrscheinlichkeit für $x = y$ gleich $z = p^2 + q^2$, also $1 - z = 2pq$. Wegen $0 < p < 1$ ist $0 < |z| < 1$. Daher ergibt sich für den Erwartungswert der Laufzeit:

$$\sum_{i=1}^{\infty} i \cdot 2 \cdot (p^2 + q^2)^{i-1} 2pq = 4pq \sum_{i=1}^{\infty} i \cdot z^{i-1} = 4pq \frac{1}{4p^2q^2} = \frac{1}{pq}$$

Aufgabe 2 (*Analyse: Speicherbandbreite (*)*)

Die Effizienz eines Algorithmus, der auf externem Speicher arbeitet, hängt von der gewählten Blockgröße B in Zusammenspiel mit der maximalen Bandbreite W_{max} und der durchschnittlichen Zugriffszeit T_{seek} des externen Speichers ab.
Bestimmen Sie für die folgenden Fälle die Blockgröße, für die 90% der maximalen Bandbreite ausgereizt werden kann. Sie können davon ausgehen, dass ohne Unterbrechung auf ganze Blöcke in zufälliger Reihenfolge zugegriffen wird. Etwaige Berechnungen können als asynchron angenommen werden. Daher muss für diese keine Zeit berücksichtigt werden.

a) $W_{max} = 144$ MByte$/s$, $T_{seek} = 12ms$ (*Lesen von Festplatte*)

b) $W_{max} = 550$ MByte$/s$, $T_{seek} = 100\mu s$ (*Lesen von SSD*)

c) $W_{max} = 68$ MByte$/s$, $T_{seek} = 60s$ (*Lesen von LTO Streamer*)

Musterlösung:
Ein Block kann in Zeit $T = T_{seek} + B/W_{max}$ eingelesen werden.
Mit der effektiven Bandbreite $W = B/T \stackrel{!}{=} 0.9 \cdot W_{max}$ ergibt sich

$$B = 9 \cdot W_{max} \cdot T_{seek}$$

für die gesuchte Blockgröße. Damit folgt:

a) $B = 15.552$ MByte ≈ 15 MByte

b) $B = 0.495$ MByte ≈ 500 kByte

c) $B = 36\,720$ MByte ≈ 37 GByte

Aufgabe 3 (*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 4 (*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.

Hinweis: Diese Aufgabe war ursprünglich für die Klausur vorgesehen.

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 gilt: $B < (S - \text{maximaler Speicher für PQ})/2$, wobei S den verfügbaren Hauptspeicher angibt. Die Blockgröße beim MergeSort kann größer gewählt werden, 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.

Aufgabe 5 (*Entwurf: Vorlesungs-Permutation*)

Für eine gegebene Algorithmen-Vorlesung speichert das Array F an Position $F[i]$ den Foliensatz zu Kapitel i. Die Vorlesung besitzt N Foliensätze, was zu viele sind, um diese im Hauptspeicher zu halten. Gegeben sei weiterhin ein Array R, welches die Reihenfolge angibt, in welcher die Kapitel tatsächlich gehalten wurden. Wird also beispielsweise zuerst Kapitel 3 gehalten, ist $R[0] = 3$. Array R passt ebenfalls nicht in den Hauptspeicher. Auf der Vorlesungs-Website sollen nun die Foliensätze in der Reihenfolge zur Verfügung gestellt werden, in der sie gehalten wurden.
Geben Sie einen Algorithmus im External Memory Modell an, der mit $O(\frac{N}{B} \cdot \log(\frac{N}{M}) + \frac{N}{B})$ IO-Zugriffen die Foliensätze so permutiert, dass diese in der Reihenfolge sind, in der sie gehalten wurden. Sie können davon ausgehen, dass in jedem Array-Element (sowohl in F als auch in R) noch genug Platz für eine weitere Zahl frei ist, die Sie als Hilfs-Variable benutzen können. B bezeichnet die Block-Größe des externen Speichers und M die Größe des Hauptspeichers.

Musterlösung:
Grundlegende Idee: Wir verwenden externes Sortieren und speichern die Schlüssel, nach denen wir sortieren, in den Hilfs-Variablen.

- Bestimme zunächst für jede Vorlesung ihren Rang in der Ausgabe, also als wievieltes sie gehalten wurde. Das Array R gibt an, welche Vorlesungen nacheinander gehalten wurden. Die gesuchten Ränge entsprechen also der Invertierung der Permutation, die durch Array R dargestellt ist. Dazu iteriere zunächst über das Array und nummeriere die Hilfsvariablen linear durch. Dies benötigt $O(\frac{N}{B})$ IO-Zugriffe. Führe dann einen externen Sortieralgorithmus durch (z.B. external merge sort), wobei die Hilfsvariablen die zu vergleichenden Schlüssel sind. Dies benötigt $O(sort(n)) = O(\frac{N}{B} \cdot \log \frac{N}{M})$ IO-Zugriffe. Die invertierte Permutation steht nun in den ursprünglichen Array-Elementen.
- Kopiere nun die invertierte Permutation in die Hilfsvariablen des Arrays F. Dies ist mit einem linearen Scan über beide Arrays möglich, was $O(\frac{N}{B})$ IO-Zugriffe benötigt.
- Zuletzt, sortiere das Array F mit den Foliensätzen, wobei wieder die Hilfsvariable der Schlüssel ist. Dies benötigt wieder $O(sort(n)) = O(\frac{N}{B} \cdot \log \frac{N}{M})$ IO-Zugriffe.
- Insgesamt haben wir damit $O(\frac{N}{B} \cdot \log \frac{N}{M} + \frac{N}{B})$ IO-Zugriffe.