Aufgabe 1

Sei k die Variable aus `Partition` und c die Konstante k aus der Aufgabenstellung.

(a)

$c = n$: Da alle anderen Elemente kleiner als das Pivot sind, erhöht sich b in jedem Schritt und es gilt $b + 1 = k$ in jedem Schritt und damit haben alle Swaps ($A[b + 1]$ mit $A[k]$) keinen Effekt. A bleibt unverändert.

$c = 1$: Im ersten Schritt verbleibt A[1] auf der ersten Position und b erhöht sich. Da alle anderen Elemente größer als das Pivot sind, bleibt b danach auf Position 1. Jedes Element $A[i]$ wird also an die zweite Stelle getauscht und dann im nächsten Schritt mit dem Element $A[i+1]$ vertauscht, sodass es am Ende um ein Feld nach rechts verschoben wurde. Im letzten Schritt landet $A[n]$ an der zweiten Stelle, da es nur hingetauscht wird.

$$A = \begin{array}{|c|cccccc|} \hline \leq x & \multicolumn{6}{c|}{> x} \\ \hline 1 & n & 2 & 3 & \dots & n-2 & n-1 \\ \hline \end{array}$$

c: Für ein beliebiges c verhält sich `Partition` im Abschnitt $A[1..c]$ wie im Fall $c = n$, d.h $b + 1$ entspricht k und die Swaps sind wirkungslos. Ab der Stelle c verhält sich `Partition` wie im Fall $c = 1$, d.h alle Elemente werden zwei mal getauscht (zur Stelle c und zurück) und dabei um eins nach rechts verschoben, im letzten Schritt landet das letzte Element an der Stelle c, da es nur hingetauscht wird.

$$A = \begin{array}{|ccccc|ccccc|} \hline \multicolumn{5}{|c|}{\leq x} & \multicolumn{5}{c|}{> x} \\ \hline 1 & 2 & \dots & c-1 & c & n & c+1 & c+2 & \dots & n-2 & n-1 \\ \hline \end{array}$$

Nimmt man an, dass das Pivotelement an die letzte Stelle des Arrays verschoben wurde (wie wir z.B. in Quicksort annehmen damit das Pivot zwischen den Blöcken verbleibt), so ergibt sich am Ende für jedes c die ursprüngliche Reihenfolge.

(b)

$c = n$: Da alle anderen Element größer sind bleibt b bis zum Ende auf Position 0 wodurch sich alle Elemente durch hin- und hertauschen um eins nach rechts verschieben bevor schließlich das Pivot auf Position 1 verbleibt, da es nur hingetauscht wird:.

$$A = \begin{array}{|c|cccccc|} \hline \leq x & \multicolumn{6}{c|}{> x} \\ \hline 1 & n & n-1 & \dots & 4 & 3 & 2 \\ \hline \end{array}$$

$c = 1$: Im ersten Schritt verbleibt A[1] auf der ersten Position und b erhöht sich. Da alle anderen Elemente kleiner sind, erhöht sich b in jedem Schritt und es gilt $b+1 = k$ in jedem Schritt und damit haben alle Swaps ($A[b+1]$ mit $A[k]$) keinen Effekt. A bleibt unverändert.

c: Für ein beliebiges c verhält sich `Partition` auf den ersten $n - c$ Elementen, die größer als c sind, so, dass b auf Position 0 verbleibt und sich alle Elemente durch hin- und hertauschen um eins nach rechts verschieben. Sobald die kleineren Elemente erreicht werden erhöht sich b in jedem Schritt, die kleinen Elemente werden also eines nach dem anderen an die nächste Position gesetzt während die größeren Elemente nach hinten geswappt werden.

wenn $c \geq n/2$ ergibt sich:

$$A = \left| \begin{array}{c|c} \leq x & > x \\ \hline c, c-1, \ldots 2, 1 & n, n-1, \ldots, c+2, c+1 \end{array} \right|$$

wenn $c < n/2$ ergibt sich:

$$A = \left| \begin{array}{c|c} \leq x & > x \\ \hline c, c-1, \ldots 2, 1 & n-c, n-c-1, \ldots c+2, c+1, n, n-1, \ldots, n-c+2, n-c+1 \end{array} \right|$$

Nimmt man an, dass das Pivotelement an die letzte Stelle des Arrays verschoben wurde, so ergibt sich für $c > n/2$:

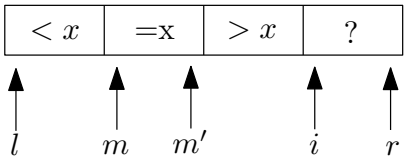
$$A = \left| \begin{array}{c|c|c} \leq x & x & > x \\ \hline c-1, c-2, \ldots 2, 1 & k & n, n-1, \ldots, c+2, c+1 \end{array} \right|$$

Aufgabe 2

```
procedure EQ_Partition(A,l,r)
      x ← A[r]
      m ← l
      m' ← m − 1
      for every i with l ≤ i ≤ r
            if A[i] < x
                  // 3-way rotate
                  tmp ← A[i]
                  A[i] ← A[m' + 1]
                  A[m' + 1] ← A[m]
                  A[m] ← tmp
                  // Update pointers
                  m ← m + 1
                  m' ← m' + 1
            else if A[i] = x
                  swap(A[m' + 1],A[i])
                  // Update pointer
                  m' ← m' + 1
            else if A[i] > x
                  // Already in a good position
      return (m,m')
```

Die Korrektheit des Programms kann durch die Korrektheit der folgenden Invarianten gezeigt werden. Sie müssen vor und nach jeder Ausführung der Schleife gelten:

$$\forall e \in A[l..m-1] : e < x \quad \wedge \quad \forall e \in A[m..m'] : e = x \quad \wedge \quad \forall e \in A[m'+1..i-1] : e > x$$



Diese Bedingung ist zu Beginn trivial erfüllt (jeweils leere Mengen). Falls $A[i] > x$, bleiben alle Invarianten direkt erfüllt. Falls $A[i] = x$, wird $A[i]$ mit dem Element $A[m']$ vertauscht, das nach Invariante $> x$ ist. Invariante 2 und 3 werden wieder erfüllt. Falls $A[i] < x$, wird zuerst $A[i]$ mit $A[m'+1] > x$ vertauscht, sodass die 3. Invariante erfüllt bleibt. Dann wird dieses Element mit $A[m] = x$ vertauscht, sodass die 1. und 2. Invariante erfüllt werden.

Da i jedes Element genau einmal referenziert, hat die Schleife genau $r - l + 1$ Durchläufe. Da eine Iteration eine konstante maximale Laufzeit hat, ist die Laufzeit[1] also $\mathcal{O}(1) + (r - l + 1) \cdot \mathcal{O}(1) + \mathcal{O}(1) \in \mathcal{O}(r - l)$. Da $r - l \leq n$, ist die Laufzeit weiterhin in $\mathcal{O}(n)$.

Aufgabe 3

Mit dem in der Lösung zu Aufgabe 4 vorgestellten Verfahren kann man tatsächlich eine bessere Laufzeit erreichen.

Aufgabe 4

Wir beschreiben zuerste eine mathematische Kernaussage, mit deren Hilfe wir das Problem vereinfachen/verkleinern können, ohne dass die relative Variante der gesuchten Eigenschaft verloren geht (d.h. wir approximieren das Problem). Danach geben wir einen Algorithmus an, der diese Kernaussage verwendet, um das Problem in seiner Größe zu reduzieren (Ausschluss-Phase). Am Ende werden wir mit einem einfachen (naiven) Algorithmus überprüfen ob es das gesuchte Element wirklich gibt, oder ob unsere Approximation ein falsches Zwischenergebnis geliefert hat (Prüf-Phase).

Die präzisen mathematischen Beweise und Formulierungen im folgenden (blau markiert) sind nicht im Ansatz gefordert und werden nur der Vollständigkeit halber und für Interessierte angegeben.

Die Erkenntnis

Es können maximal $\frac{n}{k}$ verschiedene Zahlen k-Mal im Feld vorkommen (im Folgenden oft als "Bedingung erfüllen" oder "Eigenschaft") bezeichnet. Sei $c = 1 + \left\lfloor \frac{n}{k} \right\rfloor$, sodass maximal

[1] Eine derart detaillierte Aufschlüsselung wäre nicht notwendig.

$c-1$ Zahlen die Bedingung erfüllen können. [2] Ein genauer Beweis ist nicht notwenig, da uns nur die Komplexität interessiert und $\mathcal{O}(\lceil \frac{n}{k} \rceil) = \mathcal{O}(\lfloor \frac{n}{k} \rfloor)$ gilt. Beweis:

$$\begin{aligned}(c-1)\cdot k \leq n < c\cdot k \quad &\iff \quad c-1 \leq \frac{n}{k} < c \\ &\overset{\text{mit } c\in\mathbb{N}}{\iff} \quad c-1 = \left\lfloor \frac{n}{k} \right\rfloor \\ &\iff \quad c = 1 + \left\lfloor \frac{n}{k} \right\rfloor\end{aligned}$$

Grundlegende Erkenntnis Wenn die gesuchte Eigenschaft für eine (Multi-)Menge B relativ eingehalten ist (Existenz einer Zahl z, die $\geq |B| \cdot \frac{k}{n}$ mal in B vorkommt), und nun c paarweise verschiedene Elemente entfernt werden können (also muss u.a. $|B| \geq c$ gelten), dann muss die Eigenschaft wieder relativ erfüllt sein.
Konkreter: Wir können aus dem Feld der Länge n c paarweise verschiedene Elemente entfernen, da jedes Element welches vorher $n \cdot \frac{n}{k}$-Mal vorkam nun immernoch $(n-c) \cdot \frac{n}{k}$-Mal vorkommt. Der relative Anteil der $\frac{n}{k}$ häufigen Elemente ändert sich also nicht.

Mathematische Formulierung und Beweis Sei A eine Multi-Menge (Menge mit Mehrfachvorkommen); R die Menge der zu entfernenden Elemente (mit $|R| = c$ und $R \subseteq B$); und $\#_a(X)$ die Anzahl der as in X. Dann lässt sich die „grundlegende Erkenntnis“ schreiben als:

$$\exists z \in B : \#_z(B) \geq |B| \cdot \frac{k}{n} \quad \Longrightarrow \quad \exists z \in (B \setminus R) : \#_z(A \setminus R) \geq |A \setminus R| \cdot \frac{k}{n}$$

Beweis Da R aus c paarweise verschiedenen Elementen besteht, kann die Zähl-Funktion um maximal 1 gesunken sein: $\#_a(X \setminus R) \geq \#_a(X) - 1$. Somit ist noch zu zeigen:

$$\exists z \in B : \#_z(B) \geq |B| \cdot \frac{k}{n} \quad \Longrightarrow \quad \exists z \in (B \setminus R) : \#_z(B) - 1 \geq (|B| - c) \cdot \frac{k}{n}$$

Wenn die Prämisse falsch ist, ist die Implikation trivial richtig. Auf den anderen Fall muss genauer eingegangen werden: Sei $z \in B$ also ein Element, das die Prämisse erfüllt. Dann muss es mindestens 2 Vorkommen davon geben, denn durch $|B| \geq c$ und der Definition $c > \frac{n}{k}$ gilt:

$$\#_z(B) \geq |B| \cdot \frac{k}{n} \geq c \cdot \frac{k}{n} > 1$$

[2] $c = \lceil \frac{n}{k} \rceil$ ist technisch gesehen falsch.

Wenn es mindestens 2 Vorkommnisse von z in B gibt, dann gilt auch $z \in (B \setminus R)$, wodurch sich das angenommene z für den Beweis der Folgerung benutzen lässt:

$$\begin{aligned} \#_z(B) - 1 \geq (|B| - c) \cdot \frac{k}{n} \quad &\iff \quad \#_z(B) \geq |B| \cdot \frac{k}{n} + (1 - c \cdot \frac{k}{n}) \\ &\overset{\text{Prämisse}}{\Longleftarrow} \quad |B| \cdot \frac{k}{n} \geq |B| \cdot \frac{k}{n} + (1 - c \cdot \frac{k}{n}) \\ &\iff \quad 0 \geq 1 - c \cdot \frac{k}{n} \iff c \geq \frac{n}{k} \end{aligned}$$

Datenstruktur

Für den Algorithmus wird eine Datenstruktur benötigt, die zu jedem Element eine (nicht negative) Zahl speichern kann und folgende Operationen schnell (d.h. in $\mathcal{O}(\log c)$ bei c gespeicherten Elementen) ausführen kann: Suchen, Einfügen, Wert zurücksetzen, Wert ändern und anschließend evtl. Element löschen, und Anzahl der Elemente zurückgeben. Die Datenstruktur muss nicht im Detail implementiert werden (das wäre zu viel Code und Aufwand), es sollte allerdings grob argumentiert werden, warum solche Laufzeiten möglich sind.[3] Im folgenden werden maximal $c - 1$ Elemente gespeichert.

`MapGet(Key):` Durchsucht die Datenstruktur nach dem Schlüssel `Key`. Wenn ein solcher Schlüssel existiert, liefere den damit verbundenen Wert, ansonsten -1. Laufzeit in $\mathcal{O}(\log c)$.

`MapResetCounts():` Setzt zu jedem Schlüssel den damit assoziierten Wert auf 0. Laufzeit in $\mathcal{O}(c \log c)$ ($\mathcal{O}(c)$ leicht erreichbar).

`MapDecrement(`x`):` Dekrementiert den zu jedem Schlüssel assoziierten Wert um x. Wird der Wert ≤ 0, wird der Schlüssel entfernt. Laufzeit in $\mathcal{O}(c \log c)$.

`MapPut(Key,Value):` Sucht den Schlüssel `Key`. Falls er existiert, wird der damit verbundene Wert auf `Value` gesetzt. Ansonsten wird ein neues Tupel (`Key`, `Value`) eingefügt. Laufzeit: $\mathcal{O}(\log c)$

`MapSize():` Gibt die Anzahl der aktuell gespeicherten Elemente in Zeit $\mathcal{O}(1)$ zurück.

Algorithmus

Im Grunde genommen wird diese „grundlegende Erkenntnis“ so oft angewendet, bis man eine Menge erhält, die weniger als c verschiedene Elemente enthält (z.B. $3c$ mal dasselbe Element, und sonst nichts), und damit $\leq c - 1$ Kandidaten hat (Ausschluss-Phase). Die

[3] Geeignete Kandiaten für eine solche Datenstruktur wären z.B. AVL-Bäume, Splay-Bäume, Treaps, ... Ein Teil dieser Datenstrukturen wird im Verlauf der Vorlesung noch behandelt werden.

Invariante lautet: Zu Beginn eines Schleifen-Durchlaufes ist die Multi-Menge $A[i..n] \cup$ `Map` derart, dass sie ebenso durch endlich viele Schritte der „grundlegenden Erkenntnis“ aus $A[1..n]$ hätte abgeleitet werden können, und `Map` enthält maximal $c-1$ Einträge.

```
procedure Ausschließen(k,A[1..n])
      c ← 1 + ⌊n/k⌋
      for every i in 1..n do
            if MapGet(A[i]) ≥ 0 {
                  // Notiere das Wiedersehen
                  MapPut(A[i], MapGet(A[i]) + 1)
            } else if MapSize() < c − 1 {
                  // Auffüllen des leeren Kandidaten-Slots
                  MapPut(A[i], 1)
            } else {
                  // Simuliere Wegwerfen von c Elementen:
                  MapDecrement(1) // Entfernt c − 1 Elemente
                  // Ignoriere A[i], das cte Element
            }
```

Da die grundlegende Erkenntnis eine Implikation ist, gibt es möglicherweise Elemente, welche die Eigenschaft nicht für A erfüllen. Der Algorithmus erzeugte dann sogenannte „false positives“. Diese Elemente müssen daher in einem zusätzlichen Durchlauf geprüft werden, indem ihr tatsächliches Vorkommen in A gezählt wird (Prüf-Phase):

```
procedure Prüfe(k,A[1..n])
      c ← 1 + ⌊n/k⌋
      MapResetCounts() // Einträge bleiben erhalten!
      for every i in 1..n do
            if MapGet(A[i]) ≥ 0 {
                  // Notiere das Wiedersehen
                  MapPut(A[i], MapGet(A[i]) + 1)
            }
            // Ignoriere alle anderen Elemente
      // Wirf alle Elemente weg, die es doch nicht ≥ k mal gibt:
      MapDecrement(k)
      // Map enthält alle Zahlen, die die Eigenschaft erfüllen
      return MapSize() > 0
```

Alternativ kann am Ende von `Prüfe` auch über die Abbildung iteriert werden. Jedoch soll hier das Konzept der Abbildung so einfach wie möglich gehalten werden und die Laufzeit würde sich hierdurch auch nicht verbessern.

Komplexität

Ansatz: Die Laufzeit ist stark von der genutzten Datenstruktur abhängig. Diese hängt wiederum stark von der Größe der Datenstruktur ab. Diese können wir durch die grundlegende Erkenntnis auf c beschränken. Daher kommen wir auf eine Laufzeit von $\mathcal{O}(n \log c)$ anstelle von $\mathcal{O}(n \log n)$, die mit einfachem Durchzählen oder sortieren möglich wäre.

Die Laufzeit von `Ausschließen` ist in $\mathcal{O}(1)+n\cdot\max\left\{\mathcal{O}(\log c), \mathcal{O}(\log c), \mathcal{O}(c \log c)\right\}$. Da der letzte Fall (Wegwerfen von c Elementen) allerdings nur maximal einmal in c Iterationen vorkommen kann, kürzt sich der Faktor c heraus, und wir erhalten: $\mathcal{O}(n \log c)$.

Erklärung zur Häufigkeit von MapDecrement / Wegwerfen:
Damit das Wegwerfen ausgeführt wird, müssen $c-1$ verschiedene Elemente in der `Map` sein. Diese müssen vorher erst einmal hinzugefügt worden sein, dabei werden jedoch $c-1$ mal die anderen die anderen Fälle aufgerufen. Daher kann dieser Fall nur einmal in c Schleifendurchläufen vorkommen.

Die Laufzeit von `Prüfe` ist $\mathcal{O}(1) + \mathcal{O}(c \log c) + n \cdot 2 \cdot \mathcal{O}(\log c) + \mathcal{O}(c \log c) + \mathcal{O}(1) = \mathcal{O}(c \log c) + \mathcal{O}(n \log c)$. Da $k \geq 1$ (ansonsten wäre die Frage in konstanter Zeit mit $n > 0$ zu beantworten), liegt dies in $\mathcal{O}(n \log c)$. aufgrund von $c = 1 + \left\lfloor \frac{n}{k} \right\rfloor$ ist dies gleich $\mathcal{O}(n \log(1 + \frac{n}{k}))$, was zu beweisen war.

Aufgabe 4 – Alternative Lösungen

Version 1

```
procedure vorkommen(k,A)
   m ← Median von A mit Blum, Floyd, Pratt, Rivest, und Tarjan;
   A_≤, A_=, A_≥ ← EQ_PARTITION(array=A, pivot=m);
   IF |A_=| ≥ k THEN RETURN TRUE;
   IF |A_≤| ≥ k THEN IF vorkommen(k, A_≤) THEN RETURN TRUE;
   IF |A_≥| ≥ k THEN IF vorkommen(k, A_≥) THEN RETURN TRUE;
   RETURN FALSE;
```

Die Laufzeit folgt direkt, da wir die jeweilige Rekursion abbrechen, wenn ein Teilfeld $< k$ wird. Für die Korrektheit beobachten wir, dass jedes Teilfeld zerteilt wird, bis es weniger als k Elemente hat oder wir ein häufiges Element gefunden haben. Nun angenommen ein Element kommt mindestens k-mal vor. Dann muss es irgendwann zu einem Median werden, da die Felder unterteilt werden, bis sie Größe $< k$ haben. Es müssen auch alle Vorkommen von k in diesem Teilfeld liegen. Ansonsten wäre das Element vorher schon als Median verwendet worden, um die Vorkommen aufzuteilen: Widerspruch. Somit werden die Vorkommen des Elements gezählt und der Algorithmus findet das richtige Ergebnis.

Version 2

- Wir partitionieren das Feld A der Größe n in $\lceil \frac{2n}{k} \rceil$ Teilfelder A_i der Größe $k/2$, so dass $\forall i < j : \forall a \in A_i, b \in A_j : a \leq b$ gilt. Wir nennen dies im folgenden relativ geordnet. Der Algorithmus unten benötigt $\mathcal{O}(n \log \frac{n}{k})$ Zeit dafür.
- Prüfe für jedes der A_i in $\mathcal{O}(|A_i|)$ Zeit ob es genau ein Element enthält Falls dies der Fall ist, zähle die Anzahl dieses Elementes in A_{i-1} und A_{i+1} (sofern jeweils existent). Dies benötigt Zeit $\mathcal{O}(|A_{i-1}| + |A_{i+1}|)$. Sei l und l' die Anzahl für beide Felder. Gilt nun $|A_i| + l + l' \geq k$ sind wir fertig.
- Ansonsten fahre mit A_{i+1} fort.

Gibt der Algorithmus „wahr“ zurück, muss das gesuchte Element offensichtlich existieren. Nun zur Vollständigkeit. Angenommen ein Element existiert mindestens k-mal. Durch unsere Partitionierung muss das Element ein Feld A_i komplett füllen. Anonsten könnte es aufgrund der relativen Ordnung maximal $k/2-1+k/2-1 = k-2$-mal vorkommen, da es nicht über drei Teilfelder reichen kann. Somit werden wir irgendwann dieses A_i betrachten. Danach werden die Vorkommen des Elementes in den beiden Nachbarfeldern gezählt. Da diese relativ geordnet sind, muss mindestens ein benachbartes Feld das Element enthalten. Angenommen das Feld A_{i-1} enthält o.B.d.A. $l < k/2$ Vorkommen des Elements. Dann enthält A_{i-2} *nicht* das Element. Da die Teilfelder relativ georndet sind, müssen mindestens $\hat{l} \geq k - k/2 - l = k/2 - l$ Vorkommen in den Feldern $A_{i+1}, \dots$ enthalten sein. Wiederum folgt aufgrund der relativen Ordnung, dass mindestens $k/2 - l$ Vorkommen in A_{i+1} enthalten sein müssen. Somit gibt der Algorithmus das korrekte Ergebnis zurück.

Laufzeit Das Partitionieren benötigt $\mathcal{O}(n \log \frac{n}{k})$ Zeit. Danach iterieren wir über jedes der $\mathcal{O}(\frac{n}{k})$ Teilfelder. Für jedes benötigen wir $\leq 3k$ Zeitschritte. Somit folgt eine Gesamtlaufzeit von $\mathcal{O}(n \log \frac{n}{k} + \frac{n}{k}k) = \mathcal{O}(n \log \frac{n}{k})$.

Partitionieren Wir verwenden eine Abwandlung von Quicksort, nur dass wir nach $\log \frac{2n}{k}$ Rekursionschritten abbrechen. Denn dann haben alle Teilfelder Größe $k/2$. [4] Falls $k/2 \nmid n$, können wir die $n - \lfloor \frac{2n}{k} \rfloor \frac{k}{2}$ größten Elemente im letzten Feld zusammenfassen. Für die Laufzeit von Quicksort erinnern wir uns, dass den Median von m Zahlen in $\mathcal{O}(m)$ Zeit gefunden werden kann (Blum, Floyd, Pratt, Rivest, Tarjan 1971). Ebenso können wir annehmen, dass die Felder in jedem Schritt in zwei gleich große Hälften geteilt werden (höchstens ein Größenunterschied von einem Element). Dies funktioniert, da wir den Median berechnen und folglich die Elemente, die gleich dem Median sind, beliebig auf beide Hälften verteilen können ohne unsere Eigenschaft der relativen Ordnung zu zerstören. Die Laufzeit ist dann $\mathcal{O}(n \log \frac{2n}{k}) = \mathcal{O}(n(\log 2n - \log k))$, da wir die letzten $\log k$ Rekursionen nicht durchführen.

Mit einem randomisierten Algorithmus (d.h. bei zufälliger Auwahl eines Pivot-Elements) würden wir eine *erwartete* Laufzeit von $\mathcal{O}(n \log \frac{n}{k})$ erhalten. □

[4] Streng genommen könnte es passieren, dass unsere Teilfelder am Ende $< k/2$ Elemente enthalten. Wir können aber fordern, dass die Größe zwischen $k/2$ und $k/4$ liegt. Dann müssen wir nicht nur die direkten Nachbarn, sondern auch noch deren Nachbarn prüfen. Es bleibt aber bei einer konstanten Anzahl an Nachbarn.

Aufgabe 5

Um eine obere Schranke für D zu finden, können wir die in einer vereinfachten Situation zu erwartende Anzahl D' der Würfe betrachten, so lange wir zeigen können, dass $D \leq D'$ gilt. Aus $D' \leq 2k$ folgt dann nämlich sofort $D \leq 2k$.

Wir können also ohne Weiteres davon ausgehen, dass die ersten k Bälle alle in verschiedenen Eimer landen. Die unter dieser Verschärfung zu erwartende Anzahl an Würfen ist offensichtlich nicht kleiner als D.

Wir lösen nun zunächst ein noch leichteres Problem und berechnen die erwartete Anzahl E der Würfe, die notwendig sind, ehe ein weiterer Ball in einem dieser k Eimer landet. Wir ignorieren hier also die Möglichkeit, dass beispielsweise Würfe Nummer $k+1$ und $k+2$ im selben Eimer landen. Auch das ist problemlos möglich, denn es erhöht nur die Anzahl der zu erwartenden Würfe.

Für jeden dieser (neuen) Würfe wird konstant mit einer Wahrscheinlichkeit von $\frac{k}{n}$ einer der ersten k Eimer getroffen. Dementsprechend wird mit einer Wahrscheinlichkeit von $\frac{n-k}{n}$ ein nach den initialen k Würfen leerer Eimer getroffen. Damit erhalten wir folgende Gleichung:

$$E \leq 1 + \frac{n-k}{n}E \iff E - \frac{k^2-k}{k^2}E \leq 1 \iff \frac{k^2-k^2+k}{k^2}E \leq 1 \iff E \leq k$$

Mit der Formel aus der Vorlesung folgt direkt $E \leq (\frac{k}{n})^{-1} = k$. Zusammen mit den k initialen Würfen erhalten wir $D \leq k + E \leq 2k$. □