Algorithmen I (SoSe 18) – Übungsblatt 6
Lösungsvorschlag

A Permutationen sortieren

A.1 Sortieren in Linearzeit (1 Punkt)

```
function sortElements(A : array[1...n] of (key : ℕ, data))
    B : array[1...n] of (key : ℕ, data)    // sortiertes Array
    for i := 1 to n do
        B[A[i].key] := A[i]
    return B
```

Im Wesentlichen handelt es sich bei dem Algorithmus um Bucketsort mit n Buckets, wobei jedoch in jedem Bucket genau ein Element landet (Da die Schlüssel der Elemente eine Permutation der Zahlen von $\{1, ..., n\}$ bilden, gibt es jeden Schlüssel genau ein mal). Für jedes Element e ist somit die Position im sortierten Array $e.key$. Damit braucht man nur noch einmal durch das Feld A zu iterieren und dabei jedes Datenelement an die richtige Stelle in B schreiben, was in linearer Zeit funktioniert.

A.2 Sortieren in Linearzeit mit konstantem Speicherverbrauch (2 Punkte)

```
procedure sortElementsInPlace(A : array[1...n] of (key : ℕ, data))
    for i := 1 to n do
        while A[i].key ≠ i do
            swap(A[i], A[A[i].key])
```

Da in dem Array eine Permutation liegt, kann man diese durch Ablaufen der einzelnen Zyklen sortieren. Mit jedem **swap** wird ein Array Element an seinem endgültigen Platz positioniert. Daher gibt es höchstens n **swap**s. Es wird nur ein Index i verwendet (und ein temporärer Item-Platz, falls **swap** dies benötigt).

A.3 Schranke (1 Punkt)

Der angegebene Algorithmus ist nicht vergleichsbasiert, sondern nutzt die Ganzzahligkeit und Beschränktheit des Schlüsselraums aus. Damit haben wir hier auch keinen Widerspruch zur unteren Schranke aus der Vorlesung, da diese **nur** für vergleichsbasierte Algorithmen gilt.

B Münzen wiegen (2 Punkte)

Wir bezeichnen die Münzen mit den Zahlen 1–12, es wird also eine falsche Münze $f \in \{1 \dots 12\}$ gesucht. Die Menge $\mathcal{E}$ bezeichne alle echte Münzen. Vergleiche auf der Waage werden mit $A \,?\, B$ bezeichnet, wobei $A, B \subseteq \{1, \dots, 12\}$. Man beginnt damit die zwölf Münzen in drei Mengen $\{1,2,3,4\}$, $\{5,6,7,8\}$ und $\{9,10,11,12\}$ aufzuteilen.

Erste Wiegung: $\{1,2,3,4\} \,?\, \{5,6,7,8\}$:

1) $\{1,2,3,4\} = \{5,6,7,8\}$
   $\Rightarrow \{1,2,3,4,5,6,7,8\} \subset \mathcal{E}$, $f \in \{9,10,11,12\}$.
   Zweite Wiegung: $\{1,2,3\} \,?\, \{9,10,11\}$:
   1) $\{1,2,3\} = \{9,10,11\}$
      $\Rightarrow \{1,2,3,4,5,6,7,8,9,10,11\} \subset \mathcal{E}$ und $f = 12$, aber Gewicht unbekannt.
      Dritte Wiegung: $\{1\} \,?\, \{12\}$:
      1) $\{1\} = \{12\} \Rightarrow$ Unmöglich.
      2) $\{1\} < \{12\} \Rightarrow f = 12$ und schwerer.
      3) $\{1\} > \{12\} \Rightarrow f = 12$ und leichter.
   2) $\{1,2,3\} < \{9,10,11\}$
      $\Rightarrow 12 \in \mathcal{E}$, $f \in \{9,10,11\}$ und f schwerer.
      Dritte Wiegung: $\{9\} \,?\, \{10\}$:
      1) $\{9\} = \{10\} \Rightarrow f = 11$ und schwerer.
      2) $\{9\} < \{10\} \Rightarrow f = 10$ und schwerer.
      3) $\{9\} > \{10\} \Rightarrow f = 9$ und schwerer.
   3) $\{1,2,3\} > \{9,10,11\}$
      $\Rightarrow 12 \in \mathcal{E}$, $f \in \{9,10,11\}$ und f leichter.
      Dritte Wiegung: $\{9\} \,?\, \{10\}$:
      1) $\{9\} = \{10\} \Rightarrow f = 11$ und leichter.
      2) $\{9\} < \{10\} \Rightarrow f = 9$ und leichter.
      3) $\{9\} > \{10\} \Rightarrow f = 10$ und leichter.

2) $\{1,2,3,4\} < \{5,6,7,8\}$

$\Rightarrow \{9,10,11,12\} \subset \mathcal{E}$.

Zweite Wiegung: $\{1,2,10\}$? $\{3,4,5\}$:

1) $\{1,2,10\} = \{3,4,5\}$

$\Rightarrow \{1,2,3,4,5,9,10,11,12\} \subset \mathcal{E}$, $f \in \{6,7,8\}$ und f schwerer.

Dritte Wiegung: $\{6\}$? $\{7\}$:

1) $\{6\} = \{7\} \Rightarrow f = 8$ und schwerer.
2) $\{6\} < \{7\} \Rightarrow f = 7$ und schwerer.
3) $\{6\} > \{7\} \Rightarrow f = 6$ und schwerer.

2) $\{1,2,10\} < \{3,4,5\}$

$\Rightarrow \{3,4\} \subset \mathcal{E}$, da Verschieben dieser Münzen die Waage nicht verändert. Weil die Waage immer noch schief steht, ist $f \in \{1,2,5\}$.

Dritte Wiegung: $\{1\}$? $\{2\}$:

1) $\{1\} = \{2\} \Rightarrow f = 5$ und schwerer.
2) $\{1\} < \{2\} \Rightarrow f = 1$ und leichter.
3) $\{1\} > \{2\} \Rightarrow f = 2$ und leichter.

3) $\{1,2,10\} > \{3,4,5\}$

$\Rightarrow \{1,2,5\} \subset \mathcal{E}$, da die Waage sich verändert ohne diese Münzen zu bewegen. Weil die Waage immer noch schief steht, ist $f \in \{3,4\}$ und f leichter.

Dritte Wiegung: $\{3\}$? $\{4\}$:

1) $\{3\} = \{4\} \Rightarrow$ Unmöglich.
2) $\{3\} < \{4\} \Rightarrow f = 3$ und leichter.
3) $\{3\} > \{4\} \Rightarrow f = 4$ und leichter.

3) $\{1,2,3,4\} > \{5,6,7,8\}$

$\Rightarrow \{9,10,11,12\} \subset \mathcal{E}$.

Zweite Wiegung: $\{1,2,10\}$? $\{3,4,5\}$:

1) $\{1,2,10\} = \{3,4,5\}$

$\Rightarrow \{1,2,3,4,5,9,10,11,12\} \subset \mathcal{E}$, $f \in \{6,7,8\}$ und f leichter.

Dritte Wiegung: $\{6\}$? $\{7\}$:

1) $\{6\} = \{7\} \Rightarrow f = 8$ und leichter.
2) $\{6\} < \{7\} \Rightarrow f = 6$ und leichter.
3) $\{6\} > \{7\} \Rightarrow f = 7$ und leichter.

2) $\{1,2,10\} < \{3,4,5\}$
   $\Rightarrow \{1,2,5\} \subset \mathcal{E}$, da die Waage sich verändert ohne diese Münzen zu bewegen. Weil die Waage immer noch schief steht, ist $f \in \{3,4\}$ und f schwerer.
   Dritte Wiegung: $\{3\}$? $\{4\}$:
   1) $\{3\} = \{4\} \Rightarrow$ Unmöglich.
   2) $\{3\} < \{4\} \Rightarrow f = 3$ und schwerer.
   3) $\{3\} > \{4\} \Rightarrow f = 4$ und schwerer.
3) $\{1,2,10\} > \{3,4,5\}$
   $\Rightarrow \{3,4\} \subset \mathcal{E}$, da Verschieben dieser Münzen die Waage nicht verändert. Weil die Waage immer noch schief steht, ist $f \in \{1,2,5\}$.
   Dritte Wiegung: $\{1\}$? $\{2\}$:
   1) $\{1\} = \{2\} \Rightarrow f = 5$ und leichter.
   2) $\{1\} < \{2\} \Rightarrow f = 2$ und schwerer.
   3) $\{1\} > \{2\} \Rightarrow f = 1$ und schwerer.

C k-Wege Merging

C.1 Einfache Version (2 Punkte)

Idee: Finde in jedem Schritt durch lineare Suche das kleinste Element der k Listen, entferne es aus der entsprechenden Liste und hänge es an die Ergebnisliste R an. Um Sonderfälle zu vermeiden arbeiten wir mit dem "Dummy-Element" $\perp$ und definieren $e < \perp$ für alle Elemente.

```
function easyMerge(L : array[1...k] of List of Element)
    R : List of Element
    for i := 1 to n do
        min := ⊥
        minList := ⊥
        for j := 1 to k do
            if L[j].first < min then
                min := L[j].first
                minList := L[j]
        R.append(min)
        minList.popFront()
    return R
```

Die jeweils kleinsten Elemente der Listen $L_1, \ldots, L_k$ stehen immer am Listenanfang, so dass das kleinste Element der k Listen in $\mathcal{O}(k)$ gefunden werden kann.
Insgesamt gibt es n Elemente, so dass man das Finden des kleinsten Elements der k Listen höchstens n mal durchführen muss. Ausschneiden eines Elements und Einfügen eines Elements aus einer doppelt verketteten Liste geht in konstanter Zeit. Damit ergibt sich die Gesamtlaufzeit $\mathcal{O}(nk)$.

C.2 Bessere Version (2 Punkte)

Die zeitlich problematische Operation im obigen Algorithmus ist das Finden des kleinsten Elements der k sortierten Listen $L_1, \ldots, L_k$. Durch Verwendung von Heaps kann die Laufzeit dieser Operation auf $\mathcal{O}(\log k)$ reduziert werden.

Dazu nehme man einen binären Heap H mit Einträgen der Form (*Element, ListenID*). Dabei sei *Element* jeweils das kleinste Element aus $L_1, \ldots, L_k$ und *ListenID* sei jeweils i, wenn das jeweilige Element aus der Liste L_i stammt. Initial läuft man nun über alle k Listen, betrachtet jeweils das erste Element und fügt dessen Schlüssel zusammen mit der passenden *ListenID* in H ein. Nun befindet sich für jede Liste genau ein Element in unserem Heap H.

Um jetzt das kleinste Element der k Listen zu erhalten, braucht man nur *deleteMin* auf H auszuführen. Man erhält so ein Paar (x, i). Sodann entfernt man das erste Element aus L_i und hängt dieses am Ende der sortieren Liste L an. Sollte L_i noch nicht leer sein, so wird das neue erste Element zusammen mit i in H eingefügt.

```
function heapMerge(L : array[1...k] of List of Element)
    R : List of Element
    firsts := [(L[1].first, 1), ..., (L[k].first, k)]    // Erstes Element jeder Liste mit Index
    H := buildHeap(firsts)
    while H ≠ ∅ do
        (min, minIndex) := H.deleteMin()
        minList := L[minIndex]
        R.append(min)
        minList.popFront()
        if minList ≠ ∅ then
            H.insert(minList.first, minIndex)
    return R
```

Zu jedem Zeitpunkt befinden sich höchstens k Einträge im Heap, da für jedes entnommene Element höchstens ein neues Element in den Heap H eingefügt wird. D.h. die Operationen *insert* und *deleteMin* brauchen höchstens $\mathcal{O}(\log k)$ Zeit.

Der initiale Aufbau des Heaps erfordert ebenfalls einmalig $\mathcal{O}(k)$ Zeit, wobei $k \leq n$ ist. Insgesamt ergibt sich somit ein Zeitbedarf von $\mathcal{O}(n \log k)$.