Aufgabe 1

Idee:
Wir nutzen den Median, um unsere Liste in die Hälften der kleineren und größeren Zahlen zu spalten und überprüfen dann, ob die Hälfte der größeren Zahlen ausreicht. Falls die größeren Elemente zu A aufsummieren, geben wir diese Hälfte zurück. Falls die Summe größer ist, suchen wir rekursiven die benötigten Elemente aus dieser Hälfte. Sollte sie jedoch kleiner sein, so nutzen wir die komplette Hälfte der größeren Zahlen und müssen Zahlen aus der kleineren Hälfte ergänzen. Dazu suchen wir rekursiven die Zahlen aus der kleineren Hälfte die benötigt werden den Restwert abzudecken.

Bevor wir den eigentlichen Algorithmus arbeiten lassen, überprüfen wir noch drei Dinge in folgender Reihenfolge (in $\mathcal{O}(n)$ machbar):

- Ist $A \leq 0$ geben wir die leere Menge zurück.
- Elemente mit einem Wert ≤ 0 können wir direkt entfernen
- Ist S die leere Menge, geben wir S direkt zurück.

Im Folgenden sei Split eine Routine, die die Menge anhand des Medians aufteilt und dabei Elemente, die gleich dem Median sind, so in die Mengen verteilt, dass die Kardinalitäten entweder gleich sind oder sich nur um 1 unterscheiden.

```
procedure MinSet(A,S)
      if Length(S) = 1 {
            return S
      }
      m ← Median(S)
      (S_≤, S_≥) ← Split(m, S)
      sum ← Sum(S_≥)
      if sum = A {
            return S_≥
      }
      if sum > A {
            return MinSet(A,S_≥)
      }
      return MinSet(A−Sum(S_≥),S_≤) ∪ S_≥
```

Laufzeit:

- Length(S) in $\mathcal{O}(1)$
- Median(S) in $\mathcal{O}(|S|)$
- Split(m, S) in $\mathcal{O}(|S|)$
- Sum$(S_{\geq})$ in $\mathcal{O}(|S|)$

Damit ergibt sich:

$$T(A,S) = \begin{cases} \mathcal{O}(1) & |S| = 1 \\ 3\mathcal{O}(|S|) & \textsc{Sum}(S_{\geq}) = A \\ 3\mathcal{O}(|S|) + T(A', S') & \text{sonst} \end{cases}$$

Wobei $|S'| \leq 1 + |S|/2$. Somit halbiert sich in jedem Schritt die Arbeit, die wir verrichten müssen. Im schlechtesten Fall gilt außerdem nie $\textsc{Sum}(S_{\geq}) = A$. Wir lösen $\mathcal{O}(|S|)$ auf zu $c \cdot |S|$.

$$T(A,S) \leq \sum_{i=0}^{\lceil \log |S| \rceil} 3c \left(\frac{|S|}{2^i} + 1 \right) \leq 3c \lceil \log |S| \rceil + 3c|S| \sum_{i=0}^{\infty} \frac{1}{2^i} \overset{\text{geom. Reihe}}{=} 3c \lceil \log |S| \rceil + 6c|S|$$

Somit läuft der Algorithmus in $\mathcal{O}(|S|) = \mathcal{O}(n)$.

Korrektheit:

Wir nutzen die Invariante, $W_S \geq A$ und zeigen, dass wir immer die kleinstmögliche Menge zurückgeben.

- $|S| = 1$:

  Nach Invariante ist das einzige Element in S größer als A. Außerdem gilt $A > 0$, somit ist die Menge die kleinste Teilmenge mit Summe größer A.

- $W_{S_{\geq}} = A$:

  Wir geben die kleinste Menge, die zu mindestens A aufsummiert zurück, da wir auf kein Element in $S_{\geq}$ verzichten können und jedes Austauschen mit weniger verbleibenden Zahlen die Summe verkleinern würde.

- $\textsc{Sum}(S_{\geq}) > A$:

  Da wir wissen, dass diese Menge ausreicht, können wir bedenkenlos die kleineren Zahlen ignorieren. Jede zu A aufsummierende Menge die Zahlen aus der kleineren Hälfte nutzt hat eine Menge gleicher Größe und größerer Summe, bei der alle Zahlen aus der kleineren Hälfte durch Zahlen aus der größeren ausgetauscht wurden. Die Invariante gilt nach Beschreibung des Falls. Korrektheit folgt aus der Induktionshypothese.

- $\textsc{Sum}(S_{\geq}) < A$:

  Unsere Invariante garantiert uns, dass die Zahlen in $S_{\leq}$ ausreichen, um unsere Menge $S_{\geq}$ zu erweitern. Wieder gilt, dass die Zahlen aus der größeren Hälfte nicht gewinnbringend mit Zahlen aus der kleineren Hälfte getauscht werden können. Wir suchen nach Induktionshypothese die kleinstmögliche Anzahl aus der kleineren Hälfte um insgesamt A zu erreichen.

Aufgabe 2

(a)

- **Insertionsort:**

  Unser Algorithmus fasst die Elemente der Reihe nach an (vom kleinem zum großen Index) und platziert jedes Element an der größten passenden Position im sortierten Teil, sodass ein Element hinter gleichen Elementen mit kleinerem Index platziert wird.

  Dementsprechend ist der Algorithmus stabil.

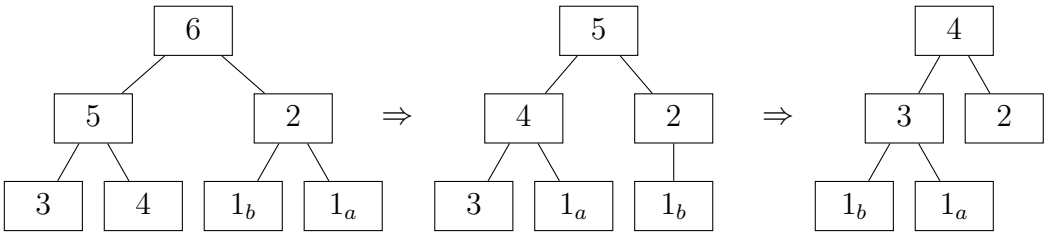
- **Selectionsort:**

  Auch hier fassen wir eigentlich die Elemente der Reihe nach an. Allerdings tauschen wir um Platz zu sparen Elemente, wodurch die Reihenfolge zwischen zwei Elementen geändert werden kann. Dies lässt sich aber leicht beheben, indem wir das Maximum einfach in einem zusätzlichen zweiten Feld platzieren. Es gilt nur darauf zu achten gleiche Elemente als neues Maximum zu vermerken wenn man vom kleinem zum großen Index über das Feld läuft.

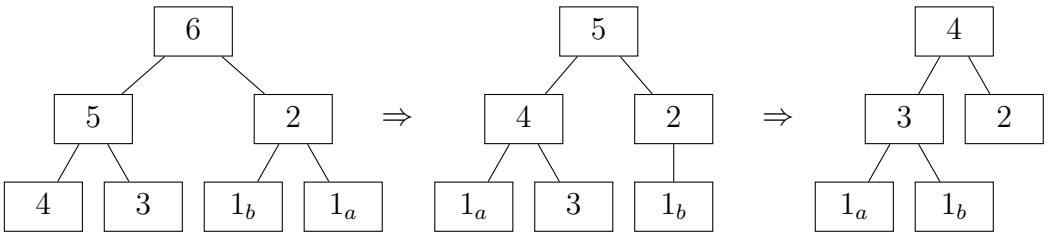
- **Heapsort:**

  Hier lässt sich kein einfacher Ansatz für stabiles Sortieren finden. Betrachtet man unseren Algorithmus wird klar, dass beim Aufbauen des Heaps und dem Vertauschen jegliche Information über relative Positionen verlorengeht. Die Position eines Elementes nach einer Heapify Operation hängt maßgeblich davon ab welche Kinder auf dem Pfad nach unten größer sind.

  Z.B. wird im folgenden Heap die Reihenfolge zwischen den Elementen 1_a und 1_b durch zwei Heapsort Schritte beibehalten:



  Ändern sich die Werte der beiden linken Blätter, so tauschen 1_a und 1_b ihre Position. Da die Heaps ab diesem Zeitpunkt äquivalent sind entspricht dies auch einem Vertauschen in der sortierten Ausgabe unabhängig davon wie Heapsort implementiert ist.



(b) Unsere Variante, die das Array zu Beginn permutiert, ist offensichtlich nicht stabil, da jede Information oder die ursprünglichen Positionen verloren geht.

Auch die Variante, die vor jedem `Partition` ein zufälliges Element als Pivot wählt, ist nicht stabil, da das ausgewählte Pivot über gleichen Schlüsseln platziert wird unabhängig davon, ob der Schlüssel zuvor unter oder über dem Pivot gestanden hat.

Zusätzlich zerstört `Partition` die Reihenfolge der Elemente wie in Aufgabe 1 auf Übungsblatt 3 zu sehen war unabhängig von der Auswahl des Pivots.

(c) Sei A das zu sortierende Array und B das zur Verfügung stehende Array der gleichen Größe. Wir wählen ein zufälliges Element als Pivot und zerteilen A in die Elemente die kleiner, gleich und größer als das Pivot sind. Dazu bestimmen wir zunächst die Größe dieser Teile und füllen dann die entsprechenden Bereiche in B jeweils von unten nach oben, während wir A von unten nach oben durchlaufen. Dadurch bleibt die relative Reihenfolge erhalten.

Am Ende unserer Methode soll A sortiert sein und zusätzlich B eine Kopie des sortierten A. Auf diese Weise sparen wir uns das Kopieren des sortierten B nach den Rekursiven Aufrufen.

```
1  quicksort(A[i..j], B[k..l])
2    assert l-k = j-i
3    if j<i then return
4    if j=i then
5      B[k] = A[i]
6      return
7    x = key(A[random(i,j)]) // pivot
8    // Partition in drei Teile <x, =x, >x
9    n, m = 0, 0 // relativer Start der Teile =x, >x
10   for c = i to j do
11     if key(A[c]) <  x then n++
12     if key(A[c]) <= x then m++
13   d, e, f = k, k + n, k + m // naechste absolute Position in B
14   for c = i to j do
15     if      key(A[c]) < x then B[d++] = A[c]
16     elseif key(A[c]) = x then B[e++] = A[c]
17     else                      B[f++] = A[c]
18   quicksort(B[k..k+n-1], A[i..i+n-1])
19   A[i+n..i+m-1] = B[k+n..k+m-1]
20   quicksort(B[k+m..l]), A[i+m..j])
```

Das Partitionieren und Kopieren hat lineare Laufzeit in der Größe des Arrays, die [erwartete] Laufzeit hat sich nicht verändert.

Aufgabe 3

Das folgende Sortierverfahren funktioniert ähnlich wie RadixSort. Wir können jedoch nicht von hinten nach vorne (von niedrigstwertiger Stelle zu höchstwertiger Stelle) sortieren, da UnstableCountingSort nicht stabil ist. Stattdessen arbeiten wir von vorne nach hinten (höchstwertige zu niedrigstwertige Stelle). Wir nehmen uns immer eine Ziffernstelle, sortieren nach ihr, gruppieren alle Zahlen nach ihrer Ziffer an dieser Stelle und sortieren anschließend für jede Gruppe rekursiv.

```
procedure Sort(array, digit):
  if digit = 0 or array.length = 1:
    return
  UnstableCoutingSort(array, digit)
  for partition in FindPartitions(array, digit):
    Sort(partition, digit-1)
```

Die Funktion FindPartitions kann offensichtlich so implementiert werden, dass sie linear in der Länge des Feldes die Partionen findet. Außerdem soll sie nur die Partitionen liefern, welche nicht leer sind. Damit erhalten wir für ein $c > 0$

$$T(A,d) = \begin{cases} \mathcal{O}(1) & |A| = 1 \vee d = 0 \\ \mathcal{O}(|A|+K) + \mathcal{O}(|A|) + \sum\limits_{\text{Partition } A' \text{ von } A \text{ nach } d} T(A', d-1) & \text{sonst} \end{cases}$$
$$\leq \begin{cases} c(|A|+K) & |A| = 1 \vee d = 0 \\ c(|A|+K) + \sum\limits_{\text{Partition } A' \text{ von } A \text{ nach } d} T(A', d-1) & \text{sonst} \end{cases}$$

Wir sehen, dass T einen Rekursionsbaum aufbaut. Die Menge aller Partitionen, für die T mit einer Ziffernstelle d aufgerufen wird, sei bezeichnet durch $\mathcal{P}_d$ (wonach für den initialen Aufruf $T(A,d)$ diese Menge also genau $\{A\}$ ist). Wir erhalten damit

$$\begin{aligned} T(A,d) &\leq \sum_{i=0}^{d} \sum_{P \in \mathcal{P}_i} c(|P|+K) \\ &= \sum_{i=0}^{d} c \left(\sum_{P \in \mathcal{P}_i} |P| + \sum_{P \in \mathcal{P}_i} K \right) \end{aligned}$$

Wir wissen, dass $\sum_{P \in \mathcal{P}_i} |P| \leq n$ gilt, da jedes der n Elemente in maximal einer Partition vorkommt. Ferner gilt $|\mathcal{P}_i| \leq n$, da FindPartitions nur nicht-leere Partitionen liefert.

$$\begin{aligned} &\leq \sum_{i=0}^{d} c\,(n + nK) \\ &= c(d+1)(n+nK) \\ &= cdn + cn + cdnK + cnK \\ &= \mathcal{O}(dnK) \end{aligned}$$

Nun zur Korrektheit des Algorithmus. Für eine Zahl x bezeichne x_k die Ziffer an Stelle k und $x_{k..\ell}$ die Ziffernfolge zwischen k und ℓ (inklusive). Wir behaupten, dass unsere Funktion für beliebige d die folgende Eigenschaft erfüllt (wobei $\textsc{Sort}(A, d)$ den Zustand des Feldes nach dem Sortieren bezeichnet):

$$\forall A : \forall 1 \leq i < j \leq |A| : \textsc{Sort}(A, d)[i]_{d..1} \leq \textsc{Sort}(A, d)[j]_{d..1}$$

Falls alle Zahlen nur d Stellen haben, erhalten wir:

$$\forall A : \forall 1 \leq i < j \leq |A| : \textsc{Sort}(A, d)[i] \leq \textsc{Sort}(A, d)[j]$$

Also Sort sortiert tatsächlich (streng genommen müsste man noch zeigen, dass Sort keine Elemente löscht oder neue erfindet). Wir zeigen das mit Induktion über d.

Für $d = 0$ ist nichts zur zeigen.

Wir zeigen nun die Aussage für $d > 0$. Falls $|A| = 1$, ist wieder nichts zu zeigen. Andernfalls beginnt unser Algorithmus mit UnstableCountingSort nach der d-ten Ziffer. Wir erhalten damit A' mit

$$\forall 1 \leq i < j \leq |A| : A'[i]_d \leq A'[j]_d$$

nun finden wir die Partionen $P_0 \dots P_{K-1}$ (möglicherweise gibt es weniger), A' bleibt gleich. Aus dem Partitionieren ist klar für $0 \leq p < p' \leq K - 1$, dass

$$\forall 1 \leq i \leq |P_p| : \forall 1 \leq j \leq |P_{p'}| : P_p[i]_d < P_{p'}[j]_d$$

woraus folgt (was leicht zu zeigen ist)

$$\forall 1 \leq i \leq |P_p| : \forall 1 \leq j \leq |P_{p'}| : P_p[i]_{d..1} < P_{p'}[j]_{d..1} \qquad \bigstar$$

Wir rufen für jede Partition Sort auf, sodass wir nach Induktionshypothese für $d - 1$ und $0 \leq p \leq K - 1$ nach allen Aufrufen (also bei Funktionsbeendigung) erhalten

$$\forall 1 \leq i < j \leq |P_p| : \textsc{Sort}(P_p, d-1)[i]_{d-1..1} \leq \textsc{Sort}(P_p, d-1)[j]_{d-1..1} \qquad \spadesuit$$

Sei nun $1 \leq i < j \leq |A|$ beliebig. Wir müssen nun zeigen

$$\textsc{Sort}(A, d)[i]_{d..1} \leq \textsc{Sort}(A, d)[j]_{d..1}$$

Angenommen i, j befinden sich in verschiedenen Partitionen. Dann folgt die Behauptung unmittelbar aus $\bigstar$. Andernfalls befinden sie sich in der gleichen Partition p. Dann wissen wir (nach der definierenden Eigenschaft unserer Partition)

$$\textsc{Sort}(A, d)[i]_d = \textsc{Sort}(A, d)[j]_d$$

also die Ziffer d gleich ist. Es reicht also, die Ziffern $d-1..1$ zu vergleichen. Da die Partition bei beiden p ist, ist dies genau die Aussage in $\spadesuit$.

Aufgabe 4

(a) Beweis durch strukturelle Induktion:

Die Wurzel des binären Baums hat entweder kein Kind, ein Kind oder zwei Kinder.

0) Hat die Wurzel keine Kinder, so besitzt der Baum nur einen Knoten (N = 1) und die Höhe der Wurzel ist $0 \geq \log_2(1+1) - 1 = \log_2(2) - 1 = 1 - 1 = 0$

1) Hat die Wurzel ein Kind, so befindet sich nach Induktionshypothese in diesem Teilbaum mit $N-1$ Knoten ein Knoten der Höhe mindestens $\log_2(N-1+1)-1 = \log_2(N)-1$, der im gesamten Baum nun mindestens die Höhe $\log_2(N)-1+1 = \log_2(2N)-1$ besitzt. Es gilt $\log_2(2N)-1 \geq \log_2(N+1)-1$ da $N > 1$ und log monoton steigend ist.

2) Seien N_1 und N_2 die Anzahl der Knoten in den beiden Teilbäumen und $N_1 \geq N_2$, dann gilt $N = N_1+N_2+1 \Rightarrow N_1 \geq \frac{N-1}{2}$. Nach Induktionshypothese befindet sich im Teilbaum mit N_1 Knoten ein Knoten der Höhe mindestens $\log_2(N_1+1)-1$ da log monoton steigend ist ergibt sich aus der unteren Schranke für N_1 eine Höhe von mindestens $\log_2(\frac{N-1}{2}+1)-1$. Dieser Knoten hat im gesamten Baum nun mindestens die Höhe $\log_2(\frac{N-1}{2}+1)-1+1 = \log_2(N-1+2)-1 = \log_2(N+1)-1$.

(b) Wir schätzen zuerst die Summe der Höhen aller Knoten ab.

Wir betrachten in jedem Schritt den höchsten verbleibenden Knoten: Der höchste Knoten besitzt nach (a) mindestens eine Höhe von $\log_2(N+1)-1$. Entfernt man gedanklich diesen Knoten (bei dem es sich um ein Blatt handeln muss, da seine Kinder größere Höhe besäßen) so bilden die restlichen $N-1$ Knoten einen Teilbaum aus $N-1$ Knoten der gleichen Höhen, deren Summe wir auf gleiche Weise abschätzen. Die Summe aller Höhen ist also mindestens

$$\sum_{n=N}^{1} \log_2(n+1) - 1 = \sum_{n=1}^{N} \log_2(n+1) - 1$$

Betrachtet man diese Summe als Obersumme über der Funktion $\log_2(x+1)-1$, so ergibt sich

$$\sum_{n=1}^{N} \log_2(n+1) - 1 \geq \int_1^N \log_2(x+1) - 1 \,\mathrm{d}x = \left[(x+1)\log_2(x+1) - x - \frac{x}{\ln(2)}\right]_1^N$$

da

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}(x+1)\log_2(x+1) - x - \frac{x}{\ln(2)} &= \log_2(x+1) + (x+1)\frac{1}{\ln(2)(x+1)} - 1 - \frac{1}{\ln(2)} \\ &= \log_2(x+1) - 1\end{aligned}$$

Es ergibt sich also eine untere Schranke für die Summe aller Höhen von

$$(N+1)\log_2(N+1) - N - \frac{N}{\ln(2)} - 2 + 1 + \frac{1}{\ln(2)} \geq N\log_2(N+1) - cN$$

für $c = 1 + \frac{1}{\ln(2)} \approx 2,44$.

Die durchschnittliche Höhe eines Knoten beträgt also mindestens $\log(N+1) - c$