Aufgabe 1 (Sortieren durch Vertauschen)

(a)

Die untere Schranke für das Auffinden unsortierter Positionen ist $\Omega(n)$. Dies ergibt sich aus der Tatsache, dass man jedes Element im Array betrachten muss, um eine mögliche unsortierte Position zu entdecken. Dies wiederum kann man durch einen Widerspruchsbeweis zeigen: Angenommen es gäbe einen Algorithmus F, der ein Paar unsortierter Positionen ermittelt, ohne alle Elemente des Arrays A zu betrachten. O.B.d.A. sei $i > 1$ einer der Indizes, die nicht betrachtet wurden. Falls nun A bis auf i aufsteigend sortiert ist und in i das Minimum des Arrays steht, so existiert ein unsortiertes Paar (z.B. $(1, i)$) das von F jedoch nicht erkannt wird. Widerspruch! (Im Falle von i=1 ergibt sich eine analoge Argumentation mit dem Paar $(i, 2)$, wenn man in i das Maximum annimmt.)

(b)

Man kann sich zunächst folgenden Sachverhalt klarmachen: Immer, wenn es zwei unsortierte Positionen i und j gibt, gibt es auch zwei unsortierte aufeinanderfolgende Positionen h und h+1 innerhalb des Intervalls $i..j$. Denn wenn dem nicht so wäre, dann müsste für alle $i \leq h < j$ gelten $A[h] \leq A[h+1]$. Damit aber müsste auch $A[i] \leq A[j]$ gelten, und somit wären i und j keine unsortierte Positionen. Widerspruch!

Der folgende Algorithmus sucht nach zwei aufeinanderfolgenden Elementen, die das Sortierkriterium verletzen:

```
algorithm findpair(A, i, j) : bool
j := 2;
while j ≤ n do
   if A[j] < A[j-1] then
      i := j-1;
      return true;
   end if;
   j := j+1;
end while
return false;
```

Der Algorithmus durchläuft das Array in einer Schleife bis ein unsortiertes Paar gefunden wurde oder die Array-Grenze erreicht wurde. Deshalb ergibt sich eine Laufzeit von $O(n)$. Aufgrund der unteren Schranke $\Omega(n)$ ist der Algorithmus optimal.

(c)

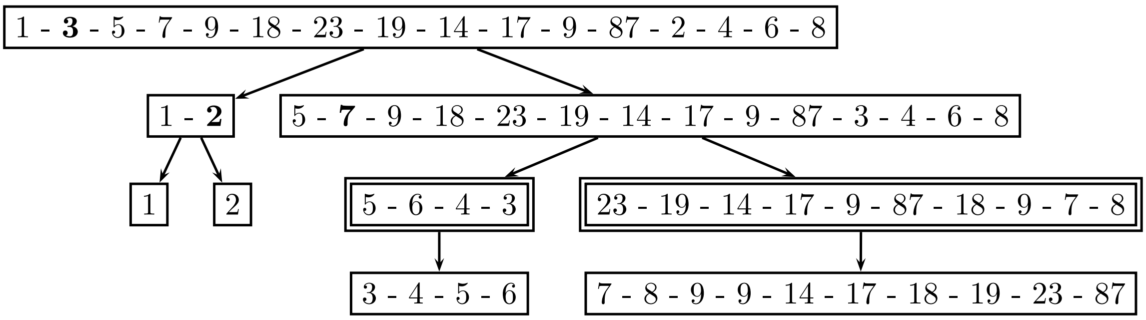
sort benötigt im worst-case $O(n^2 \cdot f(n))$ Schritte. Der worst-case ergibt sich, wenn A absteigend sortiert ist, also z.B. für A mit $A[i] = n+1-i$, und falls *findpair* immer das erste (linkeste) unsortierte Paar liefert. Dann nämlich werden n-1 Vertauschungen benötigt, um A[1] nach $A[n]$ zu bewegen. Am Beispiel n=4 kann man sehen, dass diese im Programmablauf jedoch nicht unmittelbar aufeinander folgen müssen. Um $A[2]$ nach $A[n\text{-}1]$ zu bewegen werden n-2 Vertauschungen benötigt usw. Dies ergibt insgesamt $O(n^2)$ Vertauschungen, wobei jedesmal $O(f(n))$ Schritte für *findpair* angenommen werden müssen.

Aufgabe 2

Das beschriebene Verfahren nennt sich Introsort und wurde 1997 von David R. Musser entwikkelt. Es bildet in einigen Compiler-Kollektionen (u.a. der gcc) das Standard-Sortierverfahren der Standard Template Library.

(a) Das gesuchte Verfahren muss in situ arbeiten können und eine *worst case*-Laufzeit in $O(n \cdot \log(n))$ haben. Heapsort ist der Sortieralgorithmus aus dem Kurstext, der diese Kriterien erfüllt.

(b) Der Aufrufbaum sieht folgendermaßen aus :



(c)

Die Zeit, die Quicksort auf jeder Ebene des Aufrufbaums benötigt, ist $O(n)$. Da die Tiefe des Baums durch $k \cdot \log(n)$ beschränkt wurde, ergibt sich für den Quicksort-Teil eine Laufzeit von $O(n \cdot \log(n))$. Angenommen, der alternative Sortieralgorithmus wird j mal mit Folgen der Länge $h_1,\ldots, h_j$ aufgerufen. Sei weiterhin c die Konstante aus der O-Notation für diesen Algorithmus, d.h. $c \cdot h \cdot \log(h)$ beschränkt die Laufzeit für eine Folge der Länge h. Es gilt:

$$\sum_{i=1}^{j} c \cdot h_i \cdot \log(h_i) \le c \cdot \log(n) \sum_{i=1}^{j} h_i \le c \cdot n \cdot \log(n)$$

Zunächst summieren wir die Laufzeiten der einzelnen Aufrufe des Algorithmus auf. Im nächsten Schritt schätzen wir die h_i innerhalb des Logarithmus durch n ab. Dies ist möglich, da keine Teilfolge länger als die Gesamtfolge sein kann. Diese Abschätzung ermöglicht uns, die $c \cdot \log(h_i)$ vor dem Summenzeichen zu platzieren. Wir wissen außerdem, das die Summe der Längen der Teilfolgen höchstens n sein kann, da durch die Partitionierung keine Überlappungen der Teilfolgen existieren können. Somit können wir die Summe ebenfalls durch n abschätzen.

Somit ist der Aufwand für den alternativen Teil des Algorithmus $O(n \cdot \log(n))$, was auch zu einer Gesamtlaufzeit von $O(n \cdot \log(n))$ führt.

Aufgabe 3

Nach der initialen Verteilung ergibt sich das folgende Bild:

B_ε={Nora, Carla, Ben, Uwe, Peter, Ingo, Anna, Alex}
B_n={Katrin}
B_s={Thomas}

Nach der Phase 1 haben die Behälter folgende Inhalte:

B_ε={Nora, Ben, Uwe, Ingo, Anna, Alex}
B_a={Carla, Thomas}
B_i={Katrin}
B_r={Peter}

Danach erhalten wir:

B_ε={Ben, Uwe}
B_a={Nora, Anna}
B_e={Peter}
B_l={Carla}
B_m={Thomas}
B_o={Ingo}
B_r={Katrin}
B_x={Alex}

Nach dem nächsten Schritt haben wir folgendes Bild:

B_e={Uwe, Alex}
B_g={Ingo}
B_n={Ben, Anna}
B_o={Thomas}
B_r={Nora, Carla}
B_t={Peter, Katrin}

Nach der erneuten Verteilung sehen die Behälterinhalte wie folgt aus:

B_a={Carla, Katrin}
B_e={Ben, Peter}
B_h={Thomas}
B_l={Alex}
B_n={Ingo, Anna}
B_o={Nora}
B_w={Uwe}

Im letzten Schritt ergibt sich:

B_A={Alex, Anna}
B_B={Ben}
B_C={Carla}
B_I={Ingo}
B_K={Katrin}
B_N={Nora}
B_P={Peter}
B_T={Thomas}
B_U={Uwe}

Die sortierte Folge lautet:

Alex Anna Ben Carla Ingo Katrin Nora Peter Thomas Uwe

Aufgabe 4

(a)

Sei B eine Knotenbasis, und wir nehmen an, es sei $v \in B$. Falls v nicht auf einem Zyklus liegt und einen Eingangsgrad $\neq 0$ besitzt, so muß es eine Kante (w, v) geben wobei w nicht von v aus zu erreichen ist. Da B eine Knotenbasis ist, muß entweder $w \in B$ gelten, oder es muß einen Pfad von einem Knoten in B nach w geben. In jedem Fall gibt es aber auch gleichzeitig einen Pfad von einem Knoten in B nach v, so daß man v aus B entfernen kann, ohne die Knotenbasiseigenschaft von B zu verletzen. Dies ist aber ein Widerspruch zu der Annahme, daß B minimal ist. Also gilt für Knoten v mit der beschriebenen Eigenschaft: $v \notin B$.

(b)

Da in einem azyklischen Graphen kein Knoten auf einem Zyklus liegt, folgt aus Teil (a), daß nur Knoten mit Eingangsgrad = 0 in der Knotenbasis enthalten sein können. Gleichfalls aber muß *jeder* Knoten mit Eingangsgrad = 0 in der Knotenbasis enthalten sein, da ja kein Pfad (der Länge > 0) zu ihm hinführt. Also ist die Knotenbasis in einem azyklischen Graphen eindeutig durch die Menge der Knoten mit Eingangsgrad = 0 bestimmt.