Lösungsskizzen zur Prüfung

(1) Der ursprüngliche Algorithmus hat eine exakte Laufzeit von $T(n) = 2n^3$ (siehe Möglichkeit b der Angabe). Bei dieser Aufgabe besteht keine Notwendigkeit in O-Notationen zu argumentieren bzw. zu rechnen, da die exakten Laufzeiten gegeben sind. Möglichkeit a verbessert die Laufzeit auf $T(n) = \frac{2n^3}{6} = \frac{n^3}{3}$. Variante b sorgt für eine Steigerung auf $T(n) = 2n^2$. Altenative c führt zu einer Laufzeit von $T(n) = 3n + \frac{n^3}{4}$.

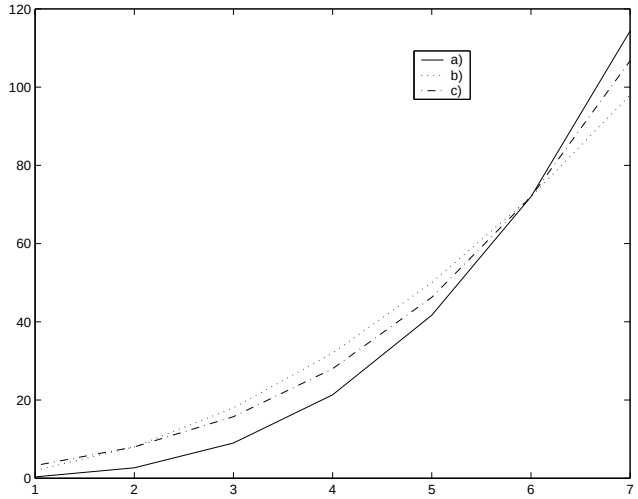
Im folgenden fragt man sich, ab welchem Werten von n welche Variante vorzuziehen ist (wir berücksichten dabei nur reale Lösungen für Inputgrößen, d.h. bei allen Bereichen von n gilt zusätzlich $n > 0$):

(a-b) $\frac{n^3}{3} < 2n^2$
$n^3 < 6n^2$
$n < 6$.
D.h. bis $n < 6$ ist Variante a der Variante b vorzuziehen, da sie schneller ist.

(b-c) $2n^2 < 3n + \frac{n^3}{4}$
$8n^2 < 12n + n^3$
$0 < n^2 - 8n + 12$ ist erfüllt für alle $n < 2$ und $n > 6$. Woraus folgt, daß Variante c für $2 < n < 6$ schneller ist als Variante b.

(a-c) $\frac{n^3}{3} < 3n + \frac{n^3}{4}$
$4n^3 < 36n + 3n^3$
$n^3 < 36$ liefert als einzige reale Lösung $n < 6$, woraus folgt, daß Möglichkeit a für alle $n < 6$ Variante c vorzuziehen ist.

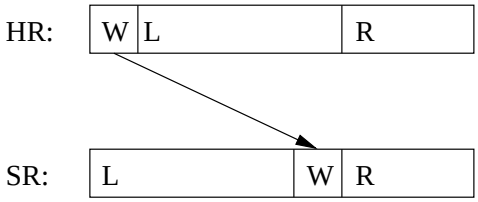
Bei $n = 6$ schneiden sich alle 3 Funktionen, was bedeutet, daß in diesem Fall alle Laufzeitverbesserungen gleich gut wären. Vor allem bei sehr großen Werten (aber natürlich schon für alle $n > 6$) sieht man einen deutlichen Laufzeitunterschied zugunsten der Möglichkeit b.



(2) Zur Unterstützung siehe Skriptum Seite 58 ff.

(a) Nein, das geht nicht. Z.B. HR='abc' und SR='cab'. Kein Binärbaum kann diese beiden beiden Reihenfolgen erzeugt haben.

(b) Ja. Kann ein Binärbaum konstruiert werden, so ist dieser immer eindeutig. Man geht dabei wie folgt vor: Als erstes Element der HR steht die Wurzel des Binärbaumes. Man sucht dieses Element dann in der SR. Damit erhält man aus der SR den linken und den rechten Teilbaum:



Nun sucht man die Elemente des linken Teilbaumes in der HR (müssen ja in HR und SR per Definition gleich viele Elemente sein und direkt nebeneinander stehen, allerdings evtl. in einer anderen Reihenfolge als in der SR). Der restliche String bildet den rechten Teilbaum. Mit diesen beiden Teilstring ruft man diesen Algorithmus rekursiv auf. Der entstehende Binärbaum ist eindeutig (oder existiert nicht).

(c) Ja, man geht dabei ähnlich vor wie bei Frage (b). Man muss beim Rekonstruktionsalgorithmus lediglich darauf achten, dass die Reihenfolge der Elemente bei der NR nicht Wurzel-Links-Rechts wie bei der HR ist, sondern dem Schema Links-Rechts-Wurzel folgt.

(d) Nein. Man betrachte einen nach links entarteten Binärbaum. Dessen HR='abcd...' und dessen SR='...dcba'

(e) Ja. Aus dem Aufbau von HR uns SR folgt, daß wenn HR und SR gleich sind, es keine linken Söhne geben kann. Der Algorithmus aus (b) folgt dann dem Schema Wurzel, rechter Teilbaum, Wurzel, rechter Teilbaum, etc. Daraus entsteht eine nach rechts entartete lineare Liste.

(3) Der gegebene Algorithmus ist ein einfacher Sortieralgorithmus.

```
Algo(A)
  n=length(A)
```

Die Länge des übergebenen Feldes wird bestimmt; Zeitbedarf: $O(1)$

```
  if n>1
```

Abbruchbedingung der Rekursion: Ist nur mehr ein Element übrig, so kann nichts mehr sortiert werden. $O(1)$

```
    min=infinity ; max=-infinity ; indmin=0 ; indmax=0
```

Initialisierung der benötigten Variablen: max und min werden auf Werte gesetzt, die sicher grösser bzw. kleiner als alle Elemente in A sind. Zeit: $O(1)$

```
for i=1 to n
  if A[i]<min then min=A[i]; indmin=i
  if A[i]>max then max=A[i]; indmax=i
end
```

Das grösste und das kleinste Element werden bestimmt. Die Werte werden in min bzw. max gespeichert, die Indizes zu A in $indmin$ bzw. $indmax$. Die Schleife wird n-mal durchlaufen. Die Anweisungen innerhalb der Schleife benötigen konstante Zeiten: $n * O(1) = O(n)$

```
if indmax=1 then indmax=indmin
hilf=A[indmin]; A[indmin]=A[1]; A[1]=hilf
hilf=A[indmax]; A[indmax]=A[n]; A[n]=hilf
```

Diese drei Zeilen dienen zur Verschiebung des kleinsten Elementes in A an die erste Position und des größten Elementes an die letzte Position im Feld. Die IF-Abfrage behandelt den Fall, daß das größte Element an der ersten Stelle im Feld steht.

```
Algo(A[2..n-1])
```

Der gleiche Algorithmus wird mit einem Teilbereich des identischen Feldes A wieder aufgerufen. Zwei Elemente ($A[1]$ und $A[n]$) fallen bei jedem Durchlauf weg: $T(n-2)$

Addiert man nun den Zeitbedarf aller Zeilen bzw. Blöcke im Programm, so erhält man:

$$T(n) = O(1) + O(1) + O(1) + O(n) + O(1) + T(n-2)$$
$$T(n) = O(n) + T(n-2)$$

Dies entspricht genau einer der rekursiven Zeitgleichungen vom Prüfungstermin vom 12.2.2001. Diese muß noch gelöst (**nicht erraten!**) werden um zum Schluss zu kommen, daß der gesamte benötigte Zeitaufwand $O(n^2)$ ist.

Eine einfache Alternative für eine schnellere Laufzeit wäre die Verwendung eines beliebigen anderen Sortieralgorithmus aus dem Skriptum.

Speicheranalyse: Das Sortieren des Feldes geschieht 'in-place', nur unter der Verwendung von einer konstanten Anzahl von Hilfsvariablen ($hilf$, $indmin$, $indmax$, min, max, i, n) und der zwangsweisen Speicherung von Rücksprungadressen. Diese Hilfsvariablen werden allerdings pro Rekursionsschritt benötigt: $S(n) = \frac{n}{2} * O(1) = O(n)$

(4) Eine Möglichkeit ist z.B. folgende: Die n Zahlen werden mit einem beliebigen Sortierverfahren – dessen worst-case Verhalten $O(n\ log\ n)$ ist – aufsteigend sortiert. Dies benötigt $O(n\ log\ n)$ Zeit. Nun wird für jede Zahl des Feldes A die Differenz zu 2002 in dem Feld mittels *BinSearch* gesucht. *BinSearch* benötigt $O(log\ n)$ Zeit, die Ergebnisabfrage $O(1)$:

```
sub FindeIndizes(A)
  n=size(A)                              O(1)
  HeapSort(A)                            O(n log n)
  for i=1 to n                           O(n)*(
    j=BinSearch(A,2002-A[i])                   O(log n)
    if (j <> -1)                               O(1)
      print "A[$i]+A[$j]=2002 ..."
      exit
    end
  end                                         )
end
```

Damit ergibt sich für die n Suchen eine Zeit von $n \cdot O(log\ n) = O(n\ log\ n)$. Die Gesamtlaufzeit des Algorithmus ist somit $O(1)+O(n\ log\ n)+n*(O(log\ n)+O(1)) = O(n\ log\ n)$.

Dieser Algorithmus arbeitet offensichtlich korrekt: zu jeder Zahl wird eine mögliche existierende Differenz zu 2002 gesucht. Ohne Sortierung würde dieses Verfahren $O(n^2)$ Zeit benötigen. Durch die Sortierung kann statt linearer Suche jedoch die Binärsuche verwendet werden, was den asymptotischen Zeitbedarf verringert.