Johannes Blömer
u.v.a.m.

Paderborn, den 10. Mai

Musterlösungen zur Vorlesung

Datenstrukturen und Algorithmen

SS 2004

Blatt 2

AUFGABE 1:

1. Die Invariante für die äußere Schleife ist recht einfach.

   Invariante für Zeilen 1-5: *Vor Durchlauf der Schleife für Index i enthält $A[1..i-1]$ die $i-1$ kleinsten Zahlen der Eingabe in sortierter Reihenfolge. Die übrigen Eingabezahlen stehen in $A[i..n]$, wobei $n = length[A]$.*

   Die Invariante für die innere Schleife ist etwas komplizierter.

   Invariante für Zeilen 2-5: *Vor Durchlauf der Schleife für Index j ist $A[j]$ das Minimum der Zahlen in $A[j..n]$, wobei $n = length[A]$.*

   Wir zeigen zunächst die Invariante für die innere Schleife.

   Initialisierung: Unabhängig vom Wert der Variablen i wird die innere Schleife zunächst immer mit dem Index $j = length[A] = n$ aufgerufen. Die Invariante sagt, dass vor dem Durchlauf mit Index n die Zahl $A[n]$ das Minimum der Zahlen in $A[n..n] = A[n]$ ist. Dieses ist sicherlich korrekt.

   Erhaltung: Sei die Invariante für die Schleife in Zeilen 2-5 vor dem Durchlauf mit Index j erfüllt. Die Abfrage in Zeile 3 bestimmt dann das Minimum von $A[j]$ und $A[j-1]$. Da aber $A[j]$ nach Voraussetzung das Minimum der Zahlen in $A[j..n]$ ist, wird durch diese Abfrage das Minimum der Zahlen in $A[j-1..n]$ bestimmt. Weiter wird dieses Minimum in Zeile 5 an die Position $j-1$ gesetzt. Somit ist vor dem Durchlauf der Schleife mit Index $j-1$ die Zahl $A[j-1]$ das Minimum der Zahlen in $A[j-1..n]$ ist.

   Terminierung: Betrachten wir die innere Schleife bei Durchlauf für $j = n$ bis $j = i+1$, also wenn i der Wert des Indexes der äußeren Schleife ist. Die Invariante für die Schleife in Zeilen 2-5 sagt dann, dass $A[i]$ das Minimum der Zahlen in $A[i..n]$ ist.

   Die Invariante für die innere Schleife werden wir nun nutzen, um die Invariante der äußeren Schleife zu zeigen.

   Initialisierung der Invariante für Zeilen 1-5: Vor dem Durchlauf mit $i = 1$ besagt die Invariante, dass $A[1..0]$ die ersten 0 Zahlen der Eingabe in sortierter Reihenfolge enthält. Da das Array zu diesem Zeitpunkt leer ist, ist diese Aussage korrekt.

Erhaltung: Vor dem Durchlauf mit Index i enthält nach Voraussetzung das $A[1..i-1]$ die $i-1$ kleinsten Zahlen der Eingabe in sortierte Reihenfolge. Die übrigen Eingabezahlen stehen in $A[i..n]$. Nun sagt aber die Invariante für die innere Schleife, dass nach Terminierung der inneren Schleife das Minimum der Zahlen in $A[i..n]$ nun durch $A[i]$ gegeben ist. Damit enthält nach Durchlauf der äußeren Schliefe mit Index i und somit vor Durchlauf der äußeren Schleife mit Index $i+1$ das Teilarray $A[1..i]$ die i kleinsten Eingabezahlen in sortierter Reihenfolge. Ausserdem sind in $A[i+1..n]$ die übrigen Eingabezahlen gespeichert, da die innere Schleife nur die Reihenfolge der Zahlen in $A[i..n]$ vertauscht.

Terminierung: Die Invariante sagt, dass vor Durchlauf der äußeren Schleife mit Index $n+1$ das Teilarray $A[1..n]$ die n kleinsten Eingabezahlen in sortierter Reihenfolge enthält. Dann aber ist das gesamte Array sortiert. Algorithmus *Bubble-Sort* sortiert damit korrekt.

2. Jeder Durchlauf der Schleife in Zeilen 2-5 erfordert höchstens eine konstante Anzahl von Basisoperationen. Sei c eine obere Schranke für die Anzahl der Basisoperation, die ein Durchlauf der inneren Schleife erfordert. Für jeden Index i der äußeren Schleife wird die Zeile 3 innere Schleife genau $n-i+1$-mal durchlaufen. Die Zeile 5 wird genau $n-i$-mal durchlaufen. Alle Durchläufe der inneren Schleife für den Index i erfordern somit höchstens $c(n-i+1)$ Basisoperationen. Dieses gilt auch für den Durchlauf mit $i=n+1$, in dem nur noch festgestellt wird, dass die Schleife beendet ist. Damit ist die Gesamtzahl der Basisoperationen beschränkt durch

$$\begin{aligned}\sum_{i=1}^{n+1} c(n-i+1) &= c\sum_{i=1}^{n+1} n - c\sum_{i=1}^{n+1}(i-1) &= cn(n+1) - c\sum_{i=0}^{n} i \\ &= cn(n+1) - \tfrac{c}{2}n(n+1) &= \tfrac{c}{2}n(n+1).\end{aligned}$$

Die Gesamtzahl der Basisoperation ist also beschränkt durch an^2 für eine Konstante a.

AUFGABE 2:
Hier ist zunächst der Algorithmus in Pseudocode.

1. Da es sich um einen rekursiven Algorithmus handelt, beschreiben wir den Algorithmus für ein beliebiges Teilarray $A[p..r]$ und gesuchten Wert v. Die Eingabe für den Algorithmus besteht als aus dem Array A, zwei Indizes p, r mit $p \leq r$, sowie dem gesuchte Wert v.

BINARY-SEARCH(A, p, r, v)

```
1     if p = r
2       then if A[p] = v
3                then return p
4                else return NIL
5       else q ← ⌊(p+r)/2⌋
6            if v = A[q]
```

```
7          then return q
8          else if v < A[q]
9                  then BINARY-SEARCH(A, p, q − 1, v)
10                 else BINARY-SEARCH(A, q + 1, r, v)
```

Wir argumentieren informell, warum der Algorithmus korrekt ist, also bei Eingabe A, p, r, v entweder einen Index $p \leq i \leq r$ findet mit $A[i] = v$ oder, falls es einen solchen Index nicht gibt, den Wert NIL ausgibt. Die Zeilen 1-4 sind der Basisfall. Gilt nämlich, dass $A[p..r]$ nur noch ein Element enthält, so muss dieses v sein, damit der Index p ausgegeben wird. Sonst muss NIL ausgegeben werden. Besteht das Array damit nur noch aus einem Element arbeitet der Algorithmus korrekt.

Enthält das Array mehr als einen Eintrag, kommen die Zeilen 5-10 des Algorithmus zum Einsatz. Zunächst wird der Mittelpunkt q des Arrays berechnet. Dann gibt es drei Fälle zu betrachten. Es kann sein $A[q] = v$. Dann muss q ausgegeben werden. Dieses wird durch die Zeilen 6 und 7 sichergestellt. Gilt $A[q] \neq q$, so muss v entweder in der unteren oder der oberen Hälfte des Arrays gesucht werden. Da wir annehmen, dass das Array sortiert ist, können wir durch den Vergleich von v mit $A[q]$ entscheiden, in welcher Hälfte weiter gesucht werden muss. Der Vergleich mit $A[q]$ und die (rekursive) Suche nach v wird im Pseudocode durch die Zeilen 8-10 realisiert. Damit ist der Algorithmus korrekt. Wird er mit den Parametern $A, 1, \mathit{length}[A]$ und v aufgerufen, findet er entweder ein i mit $A[i] = v$ oder er liefert den Wert NIL, falls der Wert v nicht im Array auftaucht.

2. Nun zur Laufzeit. Sei $n = r - p + 1$, also die Anzahl der Elemente im Teilarray $A[p..r]$. Gilt $n = 1$, so werden nur die ersten vier Zeilen des Pseudocodes durchlaufen. Dieses erfordert konstant viele Basisoperationen.

   Ist $n > 1$, so erfordern die Zeilen 1-8 insgesamt höchstens konstant viele Basisoperationen. Hinzu kommt dann aber noch ein rekursiver Aufruf des Algorithmus. Allerdings hat das Teilarray für den der rekursive Aufruf erfolgt, Größe kleiner als $n/2$. Für die worst-case Laufzeit $T(n)$ des Algorithmus BINARY-SEARCH erhalten wir damit die folgende Rekursionsgleichung.

$$T(n) \leq \begin{cases} c & \text{falls } n \leq 1 \\ T(n/2) + c & \text{sonst} \end{cases}$$

   Hierbei ist c eine Konstante, die sowohl den Aufwand für den Fall $n = 1$ als auch den Aufwand für die Zeilen 1-8 im Fall $n > 1$ abschätzt.

   Wir erhalten

$$\begin{array}{rcl@{\quad}rcl@{\quad}rcl} T(n) & \leq & T(n/2) + c & \leq & T(n/4) + c + c & = & T(n/4) + 2c \\ & \leq & T(n/8) + c + 2c & = & T(n/8) + 3c & & \\ & \vdots & & & \vdots & & \\ & \leq & T(n/2^i) + i \cdot c. & & & & \end{array}$$

   Für $i = \lceil \log(n) \rceil$ gilt $n/2^i \leq 1$. Damit erhalten wir

$$T(n) \leq T(1) + \lceil \log(n) \rceil c = c(\lceil \log(n) \rceil + 1).$$