A+D: Sortieren–Dokumentationsbeispiel

H. P. Wolf

04.12.2008, file: algodocbsp.rev

Inhalt

1 Sortieren durch Einfügen

Handlungsteil

Auch als Hinweis darauf, daß dieses Papier am Rechner aktivierbar ist, beginnt es mit dem Handlungsteil, dessen Inhalt sicher erst nach dem Studieren dieses Papiers klar wird.

1 ⟨* 1⟩ ≡
⟨*starte Auswahlmenue* 45⟩

IDEE:

Die einfache Idee fürs Sortieren durch Einfügen besteht darin, immer das nächste Element an die passende Stelle schon sortierter Elemente zu plazieren. Dieses Vorgehen ist auch den meisten Skatspielern vom Sortieren ihrer Karten her bekannt. Zunächst nimmt man die erste Karte auf, dann wird die zweite links neben die erste gesteckt, wenn sie einen höheren Wert besitzt, oder rechts, wenn der Kartenwert geringer ist. Die dritte wandert je nach Wert nach ganz links, in die Mitte oder nach rechts. Diese Art des Einfügens wird fortgesetzt, bis alle Karten untergebracht sind, und der Spieler alle Karten sortiert und reizbereit in den Händen hält.

Konstruktion

Wie läßt sich diese Idee algorithmisch aufschreiben?

Zuerst definieren wir uns eine geeignete Hülle, eine Funktion, die die einzelnen Anweisungen aufzunehmen vermag. Nennen wir sie: `sort.by.insertion`. Dieser Funktion ist der zu sortierende Wertevektor zu übergeben. Er soll als formaler Parameter den Namen `a` bekommen.

2 ⟨*start* 2⟩ ≡
```
sort.by.insertion<-function(a){
    ⟨Rumpf von sort.by.insertion 9⟩
}
```

Wir wollen uns zunächst dem spannenden Kern zuwenden, der natürlich geeignet im Rumpf der Funktion untergebracht werden muß: Es muß die Idee umgesetzt werden, daß die Elemente alle *nacheinander* abgehandelt werden. Dieses führt zu einer einfachen Schleifenkonstruktion, wobei offensichtlich das erste Element a_1 beziehungsweise `a[1]` nicht eingefügt werden muß. Die Schleife beginnt also mit dem zweiten Element und handelt als letztes das `n`-te Element ab, wenn `a` genau `n` Elemente besitzt. Innerhalb jedes Schleifendurchgangs ist das jeweils nächste Element zu plazieren. Doch nicht alles auf einmal.

3 ⟨*handele alle Elemente ab,* `sort.by.insertion` 3⟩ ≡
```
for(i in 2:n){
  ⟨plaziere a_i in den sortierten Teil, sort.by.insertion 6⟩
```

```
}
```

Nehmen wir einmal an, wir hätten die Stelle gefunden, an der das nächste Element eingefügt werden muß. Nehmen wir an, das gerade einzusortierende Elemente wäre hinter das Element mit der Indexposition $k-1$, also an der Stelle k abzulegen. Dann müßten alle folgenden Elemente des sortierten Teilvektors: $a_k, \ldots, a_{i-1}$ verschoben werden. Wird a_i an der Stelle a_k abgelegt, so steht die i-te Stelle selbst wieder zur Verfügung und kann das Element a_{i-1} aufnehmen. Einen Sonderfall gilt es noch zu überlegen: Es kann passieren, daß a_i das bisher größte Element ist. In diesem Fall ist die Position zur Einfügung nicht innerhalb von $1, \ldots, i-1$ zu finden, sondern es gilt $k = i$. Dann bedarf es keiner Umspeicherung und kein Element darf verschoben werden. Mit diesen Überlegungen kommen wir zu der folgenden Umsetzung.

4 ⟨*verschiebe* $a_k, \ldots, a_{i-1}$, *füge* a_i *ein,* `sort.by.insertion` 4⟩ ≡

```
if(i>k){
   h<-a[i]
   for(j in (i-1):k){
      a[j+1]<-a[j]
   }
   a[k]<-h
}
```

Die letzte Sektion hat den Ort als bekannt vorausgesetzt, an dem a_i eingefügt werden muß. Diesen wollen wir nun ermitteln. In einer einfachen Schleife vergleichen wir a_i mit den Elementen des sortierten Teilvektors. $a_1, \ldots, a_{i-1}$. Ist a_i das größte Element, erhält k den Wert i.

5 ⟨*bestimme Index k im sortierten Teilvektor,* `sort.by.insertion` 5⟩ ≡

```
k<-1
for(j in 1:(i-1)){
   if(a[i]>a[j]) k<-k+1
}
```

Die letzten beiden Module lassen sich zum Kern zusammenfügen.

6 ⟨*plaziere* a_i *in den sortierten Teil,* `sort.by.insertion` 6⟩ ≡

```
⟨bestimme Index k im sortierten Teilvektor, sort.by.insertion 5⟩
⟨verschiebe a_k, ..., a_{i-1}, füge a_i ein, sort.by.insertion 4⟩
```

Es sind nun noch einige Restarbeiten zu erledigen. Die Variable `n` ist zu setzen, ...

7 ⟨*initialisiere* `n`, `sort.by.insertion` 7⟩ ≡

```
n<-length(a)
```

... das Resultat ist abzuliefern ...

8 ⟨*gib Ergebnis aus,* `sort.by.insertion` 8⟩ ≡

```
return(a)
```

und zum Schluß sind die Einzelteile zusammenzubinden.

9 ⟨*Rumpf von* `sort.by.insertion` 9⟩ ≡
```
⟨initialisiere n, sort.by.insertion 7⟩
⟨handele alle Elemente ab, sort.by.insertion 3⟩
⟨gib Ergebnis aus, sort.by.insertion 8⟩
```

Test

Der Algorithmus kann mit verschiedenen Inputs getestet werden. Als Input läßt sich wählen:

> `(1,5,4,2,8,7,3)`, *Input per Hand*, *generierte Werte vorgegebener Länge* oder *R-Variable*.

Der Vektor `(1,5,4,2,8,7,3)` ...

10 ⟨*teste Algorithmus* 10⟩ ≡
```
cat("Test-Auswahl:\n")
⟨bestimme a 44⟩
print(sort.by.insertion(a))
```

... liefert:

```
[1] 1 2 3 4 5 7 8
```

Laufzeiteigenschaften

Wir wollen die Funktion auf den Prüfstand stellen und einige Laufzeiteigenschaften feststellen. Wir wollen uns für die Anzahl der notwendigen Speicheroperationen und die Anzahl der Vergleiche interessieren. Dazu bauen wir in die Funktion `sort.by.insertion` an geeigneten Stellen Zähler für diese beiden Kriterien ein, die am Ende ausgewertet werden können. Die neue Funktion heißt `insertion.statistics`.

Sowohl für die Vergleiche als auch für die Bewegungen werden 2-spaltige Matrizen aufgebaut, die in der ersten Spalte die Nummer der äußeren Schleife vermerken und in der zweiten die kumulierten Anzahlen. Die Matrix der Vergleiche heißt `count.comp`, die der Bewegungen `count.move`. Das Sortierergebnis wird, falls die Schaltervariable `count` beim Aufruf nicht auf `F` gesetzt wird, zusammen mit den beiden Matrizen als Liste ausgegeben.

11 ⟨*start* 2⟩+ ≡
```
insert.statistics<-function(a,count=T){
   n<-length(a)
   ⟨initialisiere Statistiken insert.statistics 12⟩
   for(i in 2:n){
      k<-1
      for(j in 1:(i-1)){
         if(a[i]>a[j]) k<-k+1
         ⟨zaehle Vergleiche insert.statistics 13⟩
```

```
        }
        if(i>k){
           h<-a[i]
           ⟨zaehle Umspeicherungen insert.statistics 14⟩
           for(j in (i-1):k){
              a[j+1]<-a[j]
              ⟨zaehle Umspeicherungen insert.statistics 14⟩
           }
           a[k]<-h
           ⟨zaehle Umspeicherungen insert.statistics 14⟩
        }
     }
     if(count){
        return(list(a,"count.comp"=count.comp,"count.move"=count.move))
     }else{return(a)}
  }
```

Die beiden Zählmatrizen werden mit (0,0) initialisiert.

12 ⟨*initialisiere Statistiken* `insert.statistics` 12⟩ ≡
```
count.comp<-rbind(c(0,0))
count.move<-rbind(c(0,0))
```

Die neuen Werte werden als neue erste Zeile vor die Matrix `count.comp` gehängt.

13 ⟨*zaehle Vergleiche* `insert.statistics` 13⟩ ≡
```
count.comp<-rbind(c(i,count.comp[1,2]+1),count.comp)
```

14 ⟨*zaehle Umspeicherungen* `insert.statistics` 14⟩ ≡
```
count.move<-rbind(c(i,count.move[1,2]+1),count.move)
```

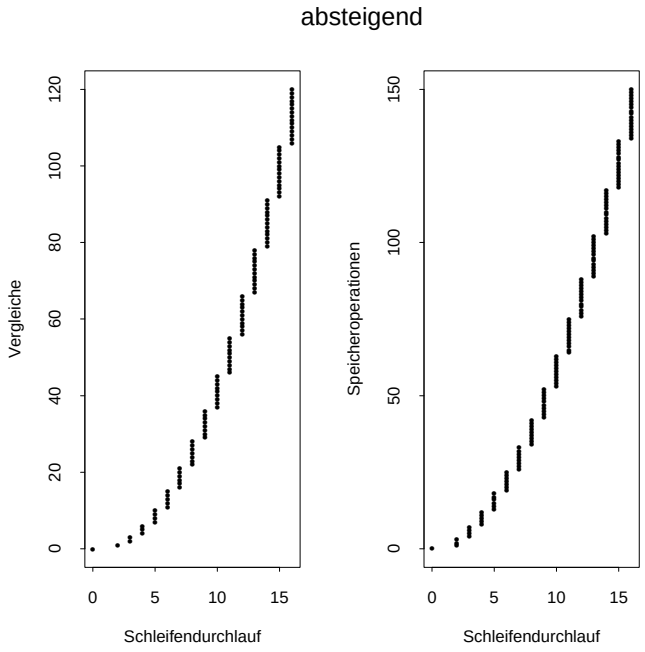
Die Funktion `insertion.statistics` wird im Rahmen von drei Experimenten eingesetzt.

2 Experiment A:

Wir wollen drei verschiedene Situationen (aufsteigend sortierter Input, absteigend, zufällig) definieren und graphisch darstellen, wie sich die Zähler bei einem Aufruf entwickeln.

15 ⟨*starte Laufzeit-Experiment A:* 15⟩ ≡
⟨*Beginn Experiment A:* 22⟩
⟨*Situationsfrage Experiment A:* 23⟩
⟨*Inputumfangsfrage Experiment A:* 24⟩
⟨*Input erstellen Experiment A:* 25⟩
⟨*Sortieralgorithmus starten Experiment A:* 26⟩
⟨*Graphische Auswertung Experiment A:* 27⟩

Wir erhalten für den Input 16:1:



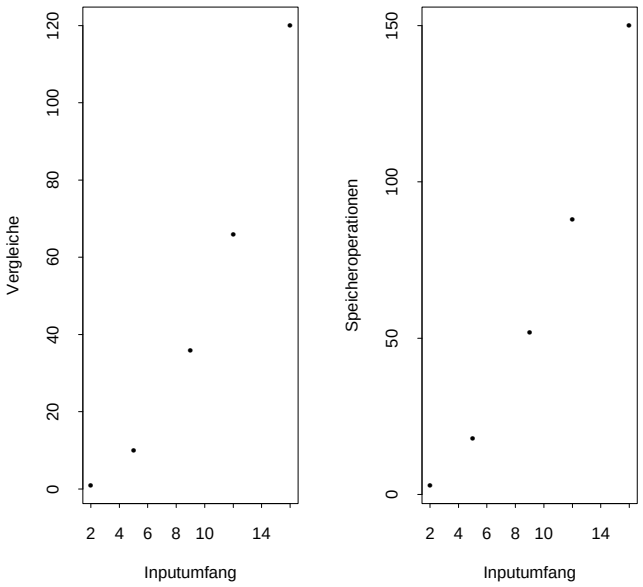
3 Experiment B.

Wir wollen sehen, wie sich die beiden Kriterien mit wachsender Größe des Input (n) entwickeln.

16 ⟨*starte Laufzeit-Experiment B:* 16⟩ ≡
⟨*Beginn Experiment B:* 28⟩
⟨*Situationsfrage Experiment B:* 29⟩
⟨*Inputumfangsfrage Experiment B:* 30⟩
⟨*Input erstellen und Experiment starten Experiment B:* 31⟩
⟨*Graphische Auswertung Experiment B:* 32⟩

Für verschiedene Umfänge, maximal 16, erhalten wir:

absteigend

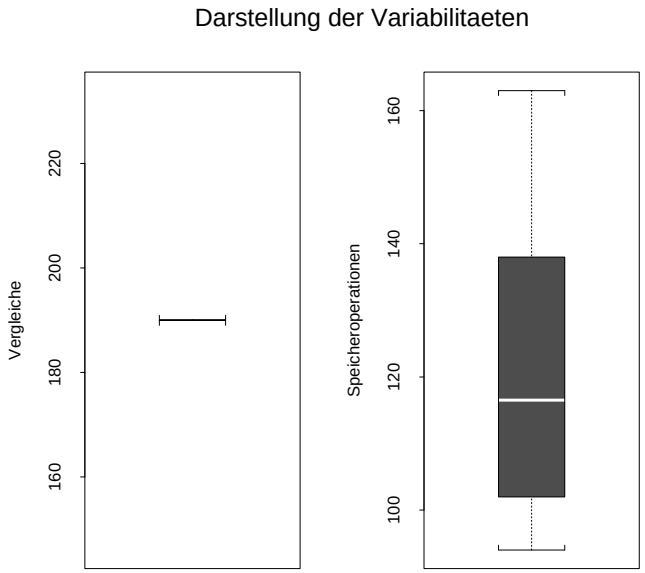


4 Experiment C.

Im dritten Experiment soll die Variabilität der Vergleiche und Speicheroperationen für eine feste Inputlänge dargestellt werden.

17 ⟨*starte Laufzeit-Experiment C:* 17⟩ ≡
⟨*Beginn Experiment C:* 33⟩
⟨*Inputumfangsfrage Experiment C:* 34⟩
⟨*Wiederholungsanzahlfrage Experiment C:* 35⟩
⟨*Input erstellen und Experiment starten Experiment C:* 36⟩
⟨*Graphische Auswertung Experiment C:* 37⟩

Für eine Werteanzahl von 20 erhalten wir bei 10 Wiederholungen das Ergebnis.



Hier bestätigt sich im linken Bild, was sicher auch die Vorüberlegungen ergeben haben: Die Anzahl der Vergleiche ist für eine feste Inputlänge auch fest.

Arbeitsweise

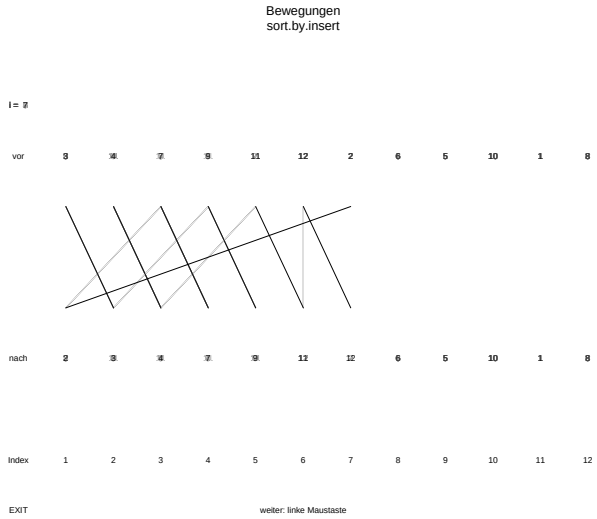
5 Demonstration der Bewegungen

In diesem Abschnitt soll die Arbeitsweise des Algorithmus demonstriert werden. Dazu zeigt die Funktion `insert.demo.move()`, wie sich die Elemente im Ablauf des Algorithmus bewegen. Die Darstellung ist im wesentlichen selbsterklärend, näheres kann aber der Funktionsdefinition entnommen werden. Technisch wählen wir interessante Stellen des Algorithmus und stellen an diesen Stellen zentrale Größen dar. Vor Beginn der eigentlichen Tätigkeit wird die Graphik initialisiert. Jeweils am Anfang und zum Ende eines Durchlauf durch die äußere Schleife wird die Graphik modifiziert. Der Bediener kann für seine Versuche verschiedene Inputs wählen.

18 ⟨*starte Element-Bewegungs-Demonstration* 18⟩ ≡

```
cat("Daten-Auswahl zur Element-Bewegungs-Demonstration:\n")
⟨bestimme a 44⟩
insert.demo.move(a)
```

Hier ist ein Bildschirmabdruck für die 12 Werte: 9, 11, 3, 4, 7, 12, 2, 6, 5, 10, 1, 8



Die Funktion `insert.demo.move()` muß natürlich noch definiert werden. Die Graphik wird nur erstellt, wenn das logische Flag `demo` den Wert `T` hat.

19 ⟨*start* 2⟩+ ≡

```
insert.demo.move<-function(a,demo=T){
   n<-length(a)
   ⟨initialisiere Bewegungs-Graphik 38⟩
   for(i in 2:n){
      ⟨zeige Belegungen vor dem Schleifendurchgang an 39⟩
      k<-1
      for(j in 1:(i-1)){
         if(a[i]>a[j]) k<-k+1
      }
      if(i>k){
         h<-a[i]
         for(j in (i-1):k){
            a[j+1]<-a[j]
         }
         a[k]<-h
      }
      ⟨zeige Belegungen nach dem Schleifendurchgang an 40⟩
   }
   return(a)
}
```

6 Demonstration der Zwischenzustände

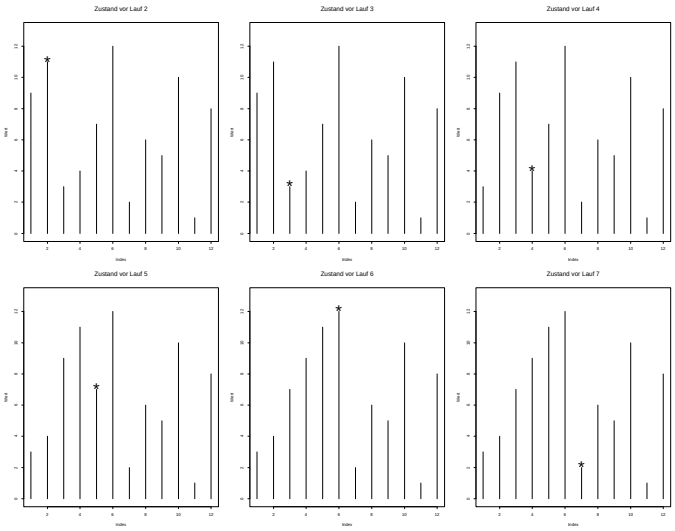
Zur Demonstration, wie sich die Zwischenzustände entwickeln, werden mit der Funktion `insert.demo.state(a)` nach jedem Schleifendurchlauf Stabdiagramme erstellt. An diesen kann der Bediener die Zustandsveränderungen verfolgen. Für seine Experimente kann er wieder zwischen verschiedenen Inputs wählen. Näheres entnehme man den Detailbemerkungen zur Funktionsdefinition.

20 ⟨*starte Zwischenzustands-Demonstration* 20⟩ ≡
```
cat("Daten-Auswahl zur Element-Bewegungs-Demonstration:\n")
⟨bestimme a 44⟩
insert.demo.state(a)
```

Die Funktion zur Demonstration von Zwischenzuständen wird nun definiert. In die schon bekannte Funktion `sort.by.insert()` werden an drei Stellen Module eingehängt, die die Graphik initialisieren, einen Plot vom Zustand vor jedem Schleifendurchlauf erstellen und einen zum Abschluß.

21 ⟨*start* 2⟩+ ≡
```
insert.demo.state<-function(a,demo=T){
   ⟨initialisiere Zustands-Graphik 41⟩
   n<-length(a)
   for(i in 2:n){
      ⟨zeichne Zustand vor Durchlauf i 42⟩
      k<-1
      for(j in 1:(i-1)){
         if(a[i]>a[j]) k<-k+1
      }
      if(i>k){
         h<-a[i]
         for(j in (i-1):k){
            a[j+1]<-a[j]
         }
         a[k]<-h
      }
   }
   ⟨erstelle letzten Plot und beende Zustands-Graphik 43⟩
   return(a)
}
```

Hier ist ein Bildschirmabdruck für die 12 Werte: 9, 11, 3, 4, 7, 12, 2, 6, 5, 10, 1, 8



Anhang

7 Laufzeitverhalten — Verfeinerungen zu Experiment A.

22 ⟨*Beginn Experiment A:* 22⟩ ≡

```
cat("Laufzeit-Experiment A\n====================\n")
cat("Darstellung der Entwicklung der Vergleiche und Speicheroperationen\n")
cat("in Abhaengigkeit der Laufvariablen der aeusseren Schleife.\n")
```

Es muß eine der drei Situationen (aufsteigend sortierter Input, absteigend, zufällig) ausgewählt werden. *Zufällig* ist die Defaultsetzung.

23 ⟨*Situationsfrage Experiment A:* 23⟩ ≡

```
cat("Welche Situation soll es sein? (Default: c)\n")
h<-menu(situation<-c("aufsteigend","absteigend","zufaellig"))
situation<-situation[ if(h==0) 3 else h ]
```

Es wird erfragt, welche Länge der zu sortierende Input haben soll. Die Defaultlänge beträgt 20.

24 ⟨*Inputumfangsfrage Experiment A:* 24⟩ ≡

```
cat("Wie viele Elemente soll der Input umfassen? (Default: 20)\n")
n<-c(scan(n=1),20)[1]
```

Der Input muß generiert werden.

25 ⟨*Input erstellen Experiment A:* 25⟩ ≡
```
if(situation=="aufsteigend") x<-1:n
if(situation=="absteigend") x<-n:1
if(situation=="zufaellig") x<-sample(1:n)
```

Der Sortieralgorithmus mit den Protokollfähigkeiten wird gestartet.

26 ⟨*Sortieralgorithmus starten Experiment A:* 26⟩ ≡
```
result<-insert.statistics(x)
```

Zur Darstellung werden zwei Graphiken erstellt. Die erste zeigt die Anzahl der kumulierten Vergleiche in Abhängigkeit vom Schleifendurchlauf, die zweite die kumulierten Speicheroperationen.

27 ⟨*Graphische Auswertung Experiment A:* 27⟩ ≡
```
par(mfrow=1:2)
plot(result[[2]],xlab="Schleifendurchlauf",ylab="Vergleiche")
plot(result[[3]],xlab="Schleifendurchlauf",ylab="Speicheroperationen")
par(mfrow=c(1,1))
title(situation)
```

8 Laufzeitverhalten — Verfeinerungen zu Experiment B.

In den nächsten Experiment wird dargestellt, wie sich die Anzahl der Vergleiche und Speicheroperationen in Abhängigkeit von der Länge des Inputs entwickelt. Es muß wieder eine von drei Situationen abgebildet werden: aufsteigend sortierter Input, absteigend, zufällig.

28 ⟨*Beginn Experiment B:* 28⟩ ≡
```
cat("Laufzeit-Experiment B\n====================\n")
cat("Darstellung der Vergleiche und Speicheroperationen\n")
cat("fuer verschieden lange Inputs.\n")
```

Siehe Experiment A. Zufällig ist die Defaultsetzung.

29 ⟨*Situationsfrage Experiment B:* 29⟩ ≡
```
cat("Welche Situation soll es sein? (Default: c)\n")
h<-menu(situation<-c("aufsteigend","absteigend","zufaellig"))
situation<-situation[ if(h==0) 3 else h ]
```

Es werden 5 verschiedene Inputlängen umgesetzt. Die größte Inputlänge ist einzugeben. Keine Eingabe setzt diese auf 20. Auf `N` stehen die einzelnen Umfänge.

30 ⟨*Inputumfangsfrage Experiment B:* 30⟩ ≡

```
cat("Bis zu welchem Umfang soll das Experiment gehen? (Default: 20)\n")
n.max<-c(scan(n=1),20)[1]
N<-floor(seq(2,n.max,length=5))
```

Die Matrix `counts` nimmt die Ergebnisse auf. Sie wird als leeres Objekt initialisiert. In der ersten Spalte stehen die Angaben zu den Vergleichen, in der zweiten zu den Speicheroperationen.

31 ⟨*Input erstellen und Experiment starten Experiment B:* 31⟩ ≡

```
counts<-NULL
for(n in N){
  if(situation=="aufsteigend") x<-1:n
  if(situation=="absteigend") x<-n:1
  if(situation=="zufaellig") x<-sample(1:n)
  result<-insert.statistics(x)[-1]
  counts<-rbind(counts,c(result[[1]][1,2],result[[2]][1,2]))
}
```

Zur Darstellung werden wieder zwei Graphiken erstellt. Die erste zeigt die Anzahl der Vergleiche in Abhängigkeit vom Inputumfang, die zweite die entsprechenden Speicheroperationen.

32 ⟨*Graphische Auswertung Experiment B:* 32⟩ ≡

```
par(mfrow=1:2)
plot(N,counts[,1],xlab="Inputumfang",ylab="Vergleiche")
plot(N,counts[,2],xlab="Inputumfang",ylab="Speicheroperationen")
par(mfrow=c(1,1))
title(situation)
```

9 Laufzeitverhalten — Verfeinerungen zu Experiment C.

In den nächsten Experiment wird dargestellt, welche Variabilität die Vergleiche und Speicheroperationen für eine feste Input-Länge haben.

33 ⟨*Beginn Experiment C:* 33⟩ ≡

```
cat("Laufzeit-Experiment C\n====================\n")
cat("Darstellung der Variabilitaet von Vergleichen und Speicheroperationen\n")
cat("fuer feste Inputlaenge.\n")
```

34 ⟨*Inputumfangsfrage Experiment C:* 34⟩ ≡

```
cat("Wie viele Elemente soll der Input umfassen? (Default: 20)\n")
n<-c(scan(n=1),20)[1]
```

Es sind die Anzahl `k` der Wiederholungen festzusetzen.

35 ⟨*Wiederholungsanzahlfrage Experiment C:* 35⟩ ≡
```
cat("Wie viele Einzelexperimente sollen gemacht werden? (Default: 10)\n")
k<-c(scan(n=1),10)[1]
```

Für eine identische Wiederholbarkeit wird der Start des Zufallszahlengenerators initialisiert.

36 ⟨*Input erstellen und Experiment starten Experiment C:* 36⟩ ≡
```
counts<-NULL
set.seed(13)
for(i in 1:k){
  x<-sample(1:n)
  result<-insert.statistics(x)[-1]
  counts<-rbind(counts,c(result[[1]][1,2],result[[2]][1,2]))
}
```

37 ⟨*Graphische Auswertung Experiment C:* 37⟩ ≡
```
par(mfrow=1:2)
boxplot(counts[,1],xlab="Inputumfang",ylab="Vergleiche")
boxplot(counts[,2],xlab="Inputumfang",ylab="Speicheroperationen")
par(mfrow=c(1,1))
title("Darstellung der Variabilitaeten")
```

10 Arbeitsweise: Verfeinerungen zur Bewegungsdemonstration

Zur Initialisierung wird ein Koordinatenfenster gesetzt, das in waagerechter Richtung von 0 bis zur Anzahl der zu sortierenden Werte geht. In vertikaler Richtung von 0 bis 5. Für etwaige Veränderungen werden verschiedene y-Werte auf Variablen `h1`,..., `h6` abgelegt. In der Höhe `h1` wird die Folge der Werte vor jedem Schleifendurchgang eingetragen, in der Höhe `h2` nach diesem. Unten wird zur Orientierung der Indexvektor vermerkt. Nach jedem Durchgang wird vom Bediener für die Fortsetzung ein Mausklick erwartet. Ein passender Text wird unten eingeblendet. Zum schnellen Ausstieg ist ein `EXIT`-Knopf vorgesehen.

38 ⟨*initialisiere Bewegungs-Graphik* 38⟩ ≡
```
if(demo==T){
  h1<-4; h2<-2; h3<-3.5; h4<-2.5; h5<-1; h6<-.5
  plot(c(0,n),c(1,h1+1),type="n",xlab="",ylab="",axes=F)
  title("Bewegungen\nsort.by.insert")
  text(0,h1,"vor"); text(1:n,h1,as.character(a))
  text(0,h2,"nach")
  text(0,h5,"Index"); text(1:n,h5,as.character(1:n))
  text(n/2,h6,"weiter: linke Maustaste")
```

```
  text(0,h6,"EXIT")
  a.alt<-a
}
```

Vor einem Durchgang werden die alten Belegungen durch Überschreiben mit der Farbe `col="white"` gelöscht und das aktuelle `a` nach dem Keyword `vor` eingetragen. Zur Information wird außerdem die Schleifenvariable `i` eingeblendet.

39 ⟨*zeige Belegungen vor dem Schleifendurchgang an* 39⟩ ≡

```
if(demo==T){
  par(col="white")
    text(0,h1+.5,paste("i = ",i-1))
    text(1:n,h1,as.character(a.alt))
    text(1:n,h2,as.character(a))
  par(col=1)
    text(1:n,h1,as.character(a))
    text(0,h1+.5,paste("i = ",i))
    a.alt<-a
}
```

Nach dem Schleifendurchgang ist das Sortierzwischenergebnis einzutragen. Damit der Betrachter die Bewegungen erkennt, werden geeignete Verbindungslinien der Art von-nach eingezeichnet. Dann wird der erwartete Mausklick abgehandelt. Die Hilfslinien werden wieder entfernt.

40 ⟨*zeige Belegungen nach dem Schleifendurchgang an* 40⟩ ≡

```
if(demo==T){
  text(1:n,h2,as.character(a))
  segments(i,h3,k,h4)
  if(i!=k) segments(k:(i-1),h3,(k+1):i,h4)
#  repeat{
#    h<-locator(n=1)
#    if(h[[2]]<(h6+.5)&h[[2]]>(h6-.5)&h[[1]]<1) return(a)
#    if(h[[2]]<(h6+.5)&h[[2]]>(h6-.5)) break
#  }
  Sys.sleep(.3)
  par(col="white")
  segments(i,h3,k,h4)
  if(i!=k) segments(k:(i-1),h3,(k+1):i,h4)
  par(col=1)
}
```

11 Arbeitsweise: Verfeinerungen zur Zwischenzustandsdemonstration

Damit der Bediener nicht immer nur einen Zustand betrachten kann, werden immer sechs auf einander folgende Plots im Graphik-Fenster generiert.

41 ⟨*initialisiere Zustands-Graphik* 41⟩ ≡
```
if(demo==T){
  par(mfrow=2:3)
}
```

Jeder Zwischenzustand wird durch ein Stabdiagramm dargestellt. An der x-Achse werden die Indizes `i` abgetragen, an der y-Achse die Inhalte `a[i]`. Das als nächste zu bearbeitende Element wird mit einem Stern markiert, die Schleifenvariable `i` wird jeweils oben links in der Ecke eingetragen. Sind sechs Plots erstellt, wird der Bediener zu einem RETURN-Druck aufgefordert, da hiernach die Bildfläche wieder gelöscht wird.

42 ⟨*zeichne Zustand vor Durchlauf* `i` 42⟩ ≡
```
if(demo==T){
   plot(a,ylim=range(a)+c(-1,1),type="n",xlab="Index",ylab="Wert")
   segments(1:n,rep(min(a)-1,n),1:n,a)
   text(i,a[i],"*",cex=2)
   title(paste("Zustand vor Lauf",i))
   if(0==((i-1)%%6)){cat("Weiter mit return!\n"); readline()}
}
```

Hier gibt es nichts zu sagen.

43 ⟨*erstelle letzten Plot und beende Zustands-Graphik* 43⟩ ≡
```
if(demo==T){
   plot(a,ylim=range(a)+c(-1,1),type="n",xlab="Index",ylab="Wert")
   segments(1:n,rep(min(a)-1,n),1:n,a)
   title("Endergebnis")
   par(mfrow=c(1,1))
}
```

Technischer Anhang

12 Datenauswahl

Dieses Modul wird an verschiedenen Stellen verwendet. Für eine identische Wiederholbarkeit wird der Start des Zufallszahlengenerators initialisiert.

44 ⟨*bestimme* `a` 44⟩ ≡
```
hh<-menu(c("c(1,5,4,2,8,7,3)", "Input per Hand",
```

```
          "generierte Werte vorgegebener Länge", "R-Variable"))
 switch(hh,{ a<-c(1,5,4,2,8,7,3)
         },{ cat("bitte Werte eingeben!\n"); a<-scan()
         },{ cat("bitte Anzahl eingeben!\n")
             set.seed(13);a<-sample(1:c(scan(n=1),20)[1])
         },{ cat("bitte Namen eingeben!\n");
             a<-eval(parse(text=scan("")))  })
 cat("Input: ",a,"\n")
```

45 ⟨*starte Auswahlmenue* 45⟩ ≡

```
 cat("Auswahl:\n")
 hh<-menu(c("Test",
            "Laufzeit-Experiment A",
            "Laufzeit-Experiment B",
            "Laufzeit-Experiment C",
            "Element-Bewegungs-Demonstration",
            "Zwischenzustands-Demonstration"
           ))
 if((!is.na(hh))&&hh>0){ switch(hh,
          { ⟨teste Algorithmus 10⟩
        },{ ⟨starte Laufzeit-Experiment A: 15⟩
        },{ ⟨starte Laufzeit-Experiment B: 16⟩
        },{ ⟨starte Laufzeit-Experiment C: 17⟩
        },{ ⟨starte Element-Bewegungs-Demonstration 18⟩
        },{ ⟨starte Zwischenzustands-Demonstration 20⟩
        })
  ⟨zeige Menü und realisiere Wunsch 46⟩
 }
```

46 ⟨*zeige Menü und realisiere Wunsch* 46⟩ ≡

```
 cmds<-"s1"
```

47 ⟨*start* 2⟩+ ≡

```
 ⟨zeige Menü und realisiere Wunsch 46⟩
```