Skizzen zu A&D

File: AuD-R-skript.rev
in: /home/wiwi/pwolf/lehre/aud/skript

Februar 2015

Inhalt

1 Vorbemerkung

In diesem Dokument sind pinselstrichartig einige vornehmlich technische Elemente zur Veranstaltung Algorithmen und Datenstrukturen sowie eine Reihe von Denkanstößen (Aufgaben) zusammengetragen worden. Für diese Elemente ist R als Notationssprache gewählt worden. Hierdurch können die eingefügten Beispiele unmittelbar ausprobiert werden.

Viele der Algorithmen gehen auf die von P. Naeve lange Zeit gehaltene Vorlesung *Algorithmen und Datenstrukturen* zurück. Manche gehören zur Menge der Algorithmen, die immer wieder in dem vorliegendem Kontext verwendet werden und sind in verschiedenen Büchern zu finden.

2 Einleitung

Frage: Warum sollte man sich mit A&D beschäftigen?

- Allgemeines Wissen: Rechner helfen, aber wie geht das?
- Algorithmen sind notierte Problemlösungen.
- Algorithmen enthalten Ideen, die an anderer Stelle nützlich sein könnten.
- Für Unterschiede zwischen theorischen und realen Ergebnisse sollte man Erklärungen kennen.
- Zum Thema Probleme lösen findet man in A&D viele Vorschläge in Reinkultur.

Beispiele:

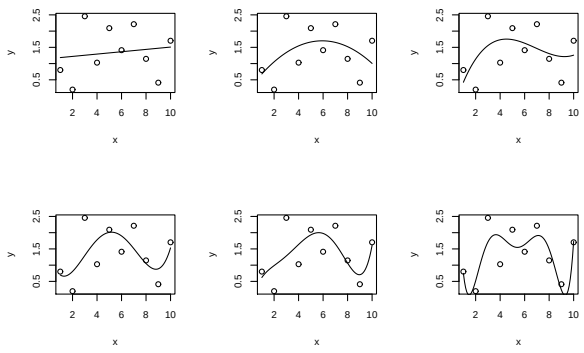
- Was halten Sie von folgendem Experiment, bei dem schulmäßig ein Polynom k-ten Grades an Daten angepasst werden soll?

1 ⟨ * 1⟩ ≡

```
anzahl.daten <- 10                        #   Daten erzeugen
x <- 1:anzahl.daten; set.seed(13)         #   x-Werte
y <- sin(x/4) + rnorm(anzahl.daten)       #   y-Werte
xx <- seq(1,10,length=100)                #   x-Werte -> Modelllinie
k.max <- 9                                #   max Polynomgrad setzen
plot(1); par(mfrow=c(3,3))                #   Darstellung vorbereiten
for(k in 1:k.max){  cat("k =",k)          #   Berechnungen umsetzen
  X <- outer(x,0:k,"^")                   #   Designmatrix Polynomgrad k
  # print(rcond(t(X)%*%X))                #   zeigt condition von X'X
  result <- solve(t(X)%*%X)%*%t(X)%*%y    #   Modell anpassen
  yy <- outer(xx,0:k,"^") %*% result      #   Modelllinie finden
  plot(x,y); lines(xx,yy)                 #   Punkte und Modell zeichnen
}
```

Wir erhalten:

```
...
k = 7
Error in solve.default(t(X) %*% X) :
  System ist fuer den Rechner singulaer: reziproke Konditionszahl = 5.59894e-19
```

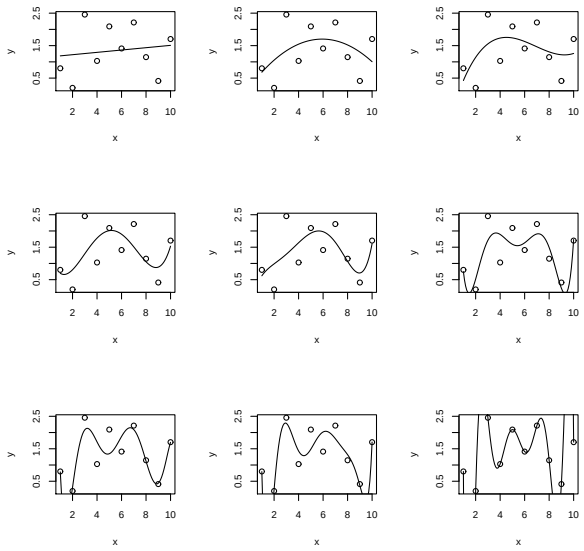


Alternative aus Anwendersicht:

2 ⟨ * 1⟩+ ≡

```
anzahl.daten <- 10                          #   Daten erzeugen
x <- 1:anzahl.daten; set.seed(13)           #   x-Werte
y <- sin(x/4) + rnorm(anzahl.daten)         #   y-Werte
xx <- seq(1,10,length=100)                  #   x-Werte -> Modelllinie
k.max <- 9                                  #   max Polynomgrad setzen
plot(1); par(mfrow=c(3,3))                  #   Darstellung vorbereiten
for(k in 1:k.max){  cat("k =",k)            #   Berechnungen umsetzen
  X <- outer(x,0:k,"^")                     #   Designmatrix Polynomgrad k
  # print(rcond(t(X)%*%X)) # zeigt condition von X'X
  result <- lm(y~X[,-1])$coef               ### Modell mit lm() anpassen
  yy <- outer(xx,0:k,"^") %*% result        #   Modelllinie finden
  plot(x,y); lines(xx,yy)                   #   Punkte und Modell zeichnen
}
```

Jetzt laufen die Berechnungen glatt durch und wir erhalten:



Es entstehen Fragen:

- Irren die Experten mit den Formeln $(X'X)^{-1}X'y$?
- Was ist der Grund für die Fehlersituation?
- Kann uns so etwas auch an anderen Stellen begegnen?
- Wie können wir solche Probleme verhindern?

Erkenntnis: die Auseinandersetzung mit Algorithmen und deren Implementierung macht Sinn.

Weitere Beispiele:

- Wie würden Sie Stichproben-Standardabweichungen ausrechnen?
- Und wie eine Chiquadrat-Statistik? Wie berechnen Sie einen p-Wert und können Sie ihrem Berechnungsergebnis glauben?

A. 2.1: Wie würden Sie die Verteilungsfunktionswerte einer binomialverteilten Zufallsvariablen berechnen? Was machen Sie, wenn n sehr groß wird?

2.1 Anzahlen von gerichteten azyklischen Graphen

Statistische Fragen können zu Fragen führen, für deren Beantwortung ein Rechner sehr hilfreich ist. Beispiel: Roy Nitze hatte Thematik: *Bootstrapping methods for causal analysis of time series data*. Dazu benötigte er die Anzahl von DAGs unter allen Graphen mit n Knoten. Dieses führte zu dem Ausführungen von `daganz`:

A. 2.2: Studiere: `daganz.rev, daganz.pdf`
`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/files/daganz.pdf`

Erkenntnis:

- Statistische Untersuchungsfragen können zu Grundfragen führen, die eher der Mathematik oder Informatik zugeordnet werden.
- Rechner sind ein Werkzeug auch für den entwickelnden Statistiker.
- Ohne Rechner sind viele (neue) Ideen kaum verfolgbar, insbesondere wenn Datensituationen ins Spiel kommen.
- Die Ausführungen in `daganz` demonstrieren den Konflikt zwischen Rechnereinsatz und Nachdenken bzw. zwischen Rechenzeit und Entwicklungszeit.
- Der riesige Speichervorrat moderne Rechner kann selbst bei einfachen Problemen begrenzend sein.
- Man erkennt, wie man Graphen durch Adjazeten-Matrizen (oder Adjazenz-Matrizen) repräsentieren kann.
- Allgemein gibt uns `daganz` einen Einblick in den Problemlösungsprozess.

A. 2.3: Überlege: Wie lässt sich die Stichprobenvarianz per Rechner berechnen? Studiere dazu insbesondere: Thisted, R.A. (1988): Elements of statistical computing → Kapitel 1.2.

A. 2.4: Es liegen die Geldbeträge von Studierenden vor, die sie im Monat zur Verfügung haben. Zu diesen sollen verschiedene Maßzahlen berechnet werden:

- Mittelwert
- Mittelwert, aber nur derjenigen, die unter 1000 Euro zur Verfügung haben
- Mittelwert, wobei Werte größer 1000 auf 1000 Euro gestutzt werden sollen
- Mittelwert, jedoch ohne die Daten, die aus dem 95%-igem zentralen Schwankungsintervall einer angepassten Normalverteilung herausfallen
- Mittelwert aus den Werten dieses Jahres sowie aus dem Mittelwert des letzten
- entsprechendes für Varianzen

Was ist zu tun, wenn diese Maßzahlen jedes Jahr berechnet werden sollen?

A. 2.5: Überlege zur Frage des Problemlösens: Wie lösen wir Sudokus?

3 Probleme lösen

Erkenntnis:

- Adjazeten-Matrizen und Matrix-Multiplikation hilft Pfade in Graphen zu finden.
- Die richtige Methode und Perspektive erleichtert die Problemlösung.
- Rätsel zu Magischen Quadraten zeigen uns bspw. elementare Zusammenhänge der Kombinatorik: Permutationen, Kombinationen.
- Brute-Force-Lösungen lassen sich oft schnell hinschreiben, versagen jedoch schon bei moderaten Problemgrößen.
- Ohne Rechner sind manche Probleme aussichtslos.

A. 3.1: Zur Sensibilisierung der Frage: *Problemlösen, wie geht das?* studiere: Polya, G (1971): How to solve it.

A. 3.2: Studiere auch: Wikipedia: → Problemlösen

A. 3.3: Beschreibe, wie sich Zahlen des 10-er Systems in Binärzahlen überführen lassen.

A. 3.4: Am Sudoku-Beispiel überlege: Wie können wir über Lösungen reden?

A. 3.5: Es gibt $n!$ Permutation von n Objekten. Doch wie können wir die einzelnen Kandidaten generieren? Wie lassen sich die Permuationen von "1,2,3,4,5" per Programm finden und lexikographisch ausgeben?

A. 3.6: Was sind magische Quadrate? Beschreiben Sie, wie Sie Rätselprobleme im Zusammenhang mit magischen Quadraten lösen?

3.1 Fragen der Repräsentation: Ein Weinpanschproblem

A. 3.7: Überlegen Sie, welche Techniken aus Ihrem bisherigem Studium bei der Lösung von Rätselproblemen helfen.

Weinbeispiel: ein Problem, das aus vielen Perspektiven betrachtet werden kann.
→ Probieren, Zustände und Handlungen, Geometrisch, Bäume, Graphen, Adjazeten-Matrizen.

Adjazeten-Matrix aus Vorlesung; Anfangszustand: `a`, gewünschter Zustand: `h`.

3 ⟨ * 1⟩+ ≡

```
am <- matrix(c(0,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,
               0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,
               0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,
```

```
                    0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,
                    0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,
                    0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,
                    1,0,1,0,1,1,0,0,1,0,1,0,1,1,0,1,
                    0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,
                    0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
                    1,1,0,1,0,1,0,1,0,0,1,1,0,0,1,1,
                    0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,
                    0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,
                    0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,0,
                    0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,
                    0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,
                    0,0,0,0,0,0,1,0,1,1,1,1,1,1,1,0),
                    16,16,dimnames=
                    list(letters[1:16],letters[1:16]))
 amp <- am; amp[amp==0] <- ""; noquote(amp)
```

```
Mon Dec 13 09:27:20 2010
  a b c d e f g h i j k l m n o p
a             1     1
b 1   1           1 1
c 1 1         1         1
d 1       1         1 1
e 1     1     1               1
f 1           1     1     1
g 1         1                     1
h 1               1 1       1
i   1         1 1                 1
j 1                     1         1
k       1     1     1             1
l     1             1     1       1
m           1 1         1         1
n             1 1               1 1
o         1         1       1   1
p             1     1
```

Ergebnis: a:j:k:d:e:o:n:h durch Rückverfolgung der Verbindung von a nach h in am6:

4 ⟨ * 1⟩+ ≡

```
 am%*%am%*%am%*%am%*%am%*%am%*%am -> am6
```

```
Mon Dec 13 09:28:42 2010
    a  b  c   d  e   f   g  h  i   j   k  l   m  n  o   p
a 148  0 10  58  3  46 299  1  1 299  46  3  58  0 10 148
b 532  3 40  96 28 237 569  1 35 605 205 34  91 22 13 533
c 539 36 13  71 34 225 594 21  1 558 238 56  95  3 12 533
d 520  3  6  85 56 234 514  1  7 536 235 34  66 22 12 525
e 519  8 13 105 33 200 601 21  1 579 227 22  96  3 40 513
f 406  2 26  92 17 169 537  1  7 545 156 23 101  2 13 411
g 359  8  2  22 30 179 234  1  0 226 178 38  27  3  1 357
h 534  3 26  96 42 238 563  1 35 604 211 34  77 36 13 535
i 535 36 13  77 34 211 604 35  1 563 238 42  96  3 26 534
j 357  3  1  27 38 178 226  0  1 234 179 30  22  8  2 359
```

```
k 411  2 13 101 23 156 545  7  1 537 169 17  92  2 26 406
l 513  3 40  96 22 227 579  1 21 601 200 33 105  8 13 519
m 525 22 12  66 34 235 536  7  1 514 234 56  85  3  6 520
n 533 22 13  91 34 205 605 35  1 569 237 28  96  3 40 532
o 533  3 12  95 56 238 558  1 21 594 225 34  71 36 13 539
p 148  0 10  58  3  46 299  1  1 299  46  3  58  0 10 148
```

A. 3.8: Wie lassen sich die Zahlen in `am6` interpretieren?

A. 3.9: Was ist zu tun, um ein anderes Umschüttproblem zu lösen?

Vgl.: Herold, H., Lurz, B., Wohlrab, J. (2007): Kap. 3.1 Rätsel: Umfüllprobleme.

4 Algorithmus und Notation

4.1 Euklid: der erste Algorithmus

Ursprünge des Begriffs *Algorithmus*: Wie fing es an?

Euklid hat uns die Idee zu folgender Funktion geliefert:

5 ⟨* 1⟩+ ≡

```
gcd<-function(m,n){
   while(n>0){
      rem <- m%%n          ## :<=> m mod n
      m   <- n
      n   <- rem
   }
   return(m)
}
```

Anschaulicher ist eine Version, die einige Infos über Zwischenzustände liefert.

6 ⟨* 1⟩+ ≡

```
gcd.demo<-function(m,n){
   while(n>0){
      rem <- m%%n          ## :<=> m mod n
      m   <- n
      n   <- rem
      cat("aktueller Inhalt von n und m:",n,m,"\n")
   }
   cat("Ergebnis: gcd =",m)
   return(m)
}
gcd.demo(24,36)
```

Die Demo liefert uns:

```
aktueller Inhalt von n und m: 24 36
aktueller Inhalt von n und m: 12 24
aktueller Inhalt von n und m: 0 12
Ergebnis: gcd = 12
```

A. 4.1: Was ist ein Algorithmus? Suche Definitionen in Lehrbüchern.

A. 4.2: Was ermittelt `gcd()`? Wie funktioniert `gcd()`?

A. 4.3: Welche Sprachelemente werden benutzt?

A. 4.4: Lässt sich sagen, wie lange der Algorithmus für das Ergebnis braucht?

A. 4.5: Wie müsste eine richtig gut kommentierte Fassung aussehen?

A. 4.6: Suche in Büchern zum Thema A&D nach dem Euklidschen Algorithmus und studiere, was an diesem demonstriert wird. Stöbere zum Beispiel in
D. E. Knuth (div Ed.): The art of computer programming: Bd. 1. – Fundamental algorithms. `FB 13 HI100 K74 / 1(3)`

A. 4.7: Schreibe ein Programm, das zu einer Zahl alle seine Teiler ermittelt.

Bemerkungen:

- Verständlichkeit für Maschine / für den Menschen
- Schriften, Ligaturen haben Einfluss auf Lesbarkeit
- Schriften auf Rechnern stehen in Verbindung mit Kurvenalgorithmen, Kompressionsverfahren
- Einkommenssteuerberechnungsanweisung stellt Algorithmus dar

5 Eigenschaften von Lösungen

5.1 Die Anzahl gesetzter Bits

Ein einfaches Beispiel-Problem mit verschiedenen Lösungen.

Problemstellung: Gegeben sei ein Bitvektor, gesucht ist die Anzahl der Bits im Bitvektor. Was ist eine gute / geeignete Lösung? Wie erhält man gute Lösungen?

L1: Die Standard-Lösung: Wir lösen das Problem, indem wir eine Schleife über die Bits konstruieren.

7 ⟨ * 1⟩+ ≡

```
L1 <- function(b){
  s <- 0; n <- length(b)
  for(i in 1:n){
     if(b[i]==1) s <- s+1
  }
  return(s)
```

```
}
L1(c(1,1,1,0,0,0,1))
```

L1a: Eine Variation erhalten wir, wenn wir die Bits addieren. Falls die Addition schneller ist als die Abfrage, macht das Sinn.

8 ⟨* 1⟩+ ≡

```
L1a <- function(b){
  n <- length(b)
  for(i in 1:n){
     s <- s+b[i]
  }
  return(s)
}
L1(c(1,1,1,0,0,0,1))
```

Folgende Lösung erfordert nur so viele Schleifendurchgänge wie es 1-en im Bitvektor gibt. Betrachten wir Algorithmus L2.

9 ⟨* 1⟩+ ≡

```
L2 <- function(B){
  ⟨konvertiere B in einen Bitvektor 10⟩
  n<-length(B)
  cat("Initial B:",(0:1)[1+B],"\n")
  s <- 0
  while(any(B>0)){
    s <- s+1
    ⟨transformiere B in die Zahl B.zehn des Zehnersystems 11⟩
    B.zehn <- B.zehn-1
    ⟨transformiere B.zehn in Bitvektor BB 12⟩
    cat("B-1:      ",BB,"\n")
    B <- B & BB
    cat("new     B:",(0:1)[1+B]," // Counter:",s)
  }
  return(s)
}
L2("1110001")
```

```
Initial B: 1 1 1 0 0 0 1
B-1:       1 1 1 0 0 0 0
new     B: 1 1 1 0 0 0 0  // Counter: 1
B-1:       1 1 0 1 1 1 1
new     B: 1 1 0 0 0 0 0  // Counter: 2
B-1:       1 0 1 1 1 1 1
new     B: 1 0 0 0 0 0 0  // Counter: 3
B-1:       0 1 1 1 1 1 1
new     B: 0 0 0 0 0 0 0  // Counter: 4
```

Der R-Code enthält einige Lösungsteile, die nicht der R-Syntax entsprechen. Hierbei handelt es sich um Konstrukte, die verbal die notwendigen Handlungen bezeichnen. Diese Bezeichnungen nennen wir Code-Chunk-Namen, die den Code repräsentieren. An Stelle der Code-Chunk-Namen muss für eine lauffähige Lösung der passende Code eingesetzt werden. Diesen Code findet man etwas über die Bezeichnungen. Zum Beispiel wir in dem folgenden Code-Stück (*Code-*

Chunk) der Code definiert, der sich hinter dem Namen: ⟨*konvertiere* `B` *in einen Bitvektor* 10⟩ verbirgt. Auf diese Weise der Lösungsbeschreibung können wir uns zunächst auf die wesentlichen Punkte konzentrieren. Es entsteht eine Mischung von Gedanken und Code-Stücken, die uns schrittweise die Lösung vermitteln. In einem späteren Kapitel wird diese Technik näher diskutiert. Hier bietet sich aber schon die folgende Aufgabe an.

A. 5.1: Recherchiere im Internet, was man unter *literate programming* versteht.

Kommen wir zurück zu dem Algorithmus `L2`. Diese Lösung erfordert die Möglichkeit der (schnelle) Interpretation des Bitvektors `B` als Zahl im Zehnersystem sowie die Umsetzung von `B-1`.

10 ⟨*konvertiere* `B` *in einen Bitvektor* 10⟩ ≡ ⊂ 9

```
h <- 1:nchar(B)
B <- c(F,T)[1+("1"==substring(B,h,h))]
```

Hier lässt sich sehen, wie der Bitvektor nach $\sum_{i=0} 2^i b_i$ in eine normale Zahl umgewandelt werden kann.

11 ⟨*transformiere* `B` *in die Zahl* `B.zehn` *des Zehnersystems* 11⟩ ≡ ⊂ 9

```
B.zehn <- 2^((n-1):0)%*%B
```

Durch Integer-Divisionen können wir Zahlen in Binär-Darstellungen überführen.

12 ⟨*transformiere* `B.zehn` *in Bitvektor* `BB` 12⟩ ≡ ⊂ 9

```
BB <- floor(B.zehn/(2^((n-1):0)))
BB <- BB - c(0,BB*2)[-n-1]
```

A. 5.2: Was sind die Vorteile, was die Nachteile der Algorithmen zum Bit-Problem?

A. 5.3: Wie geht es noch schneller?

A. 5.4: Gegeben sind auf Basis einer Umfrage die Antworten zu dem Merkmal Haarfarbe. Die Haarfarben sind in einem Ergebnisvektor als Zeichenketten abgelegt worden. Nenn Sie verschiedene Wege, um eine Häufigkeitstabelle zu erstellen?

6 Laufzeitverhalten

A. 6.1: Studiere: Levitin, A. (z.B.2003): The design & Analysis of Algorithms, genauer → Kapitel 2.1–2.3 → The analysis of algorithms, O-, Omega-, Θ-Notation.

6.1 Suche Teilvektor mit der größten Teilsumme

Problemstellung: Gegeben sei ein Zahlenvektor. Gesucht ist der zusammenhängende Teilvektor mit der größten Elementsumme.

Quellen: Peter Naeve: A+D-Skripten; Bentley, Jon (2000): Programming Pearls, Addison Wesley.

A. 6.2: Studiere: Gustav Pomberger & Heinz Dobler (2008): Algorithmen und Datenstrukturen, Pearson Studium → Kapitel 5.

Zu diesem Problem sind im Folgenden Lösungen aufgezeigt, die sich durch ihr Laufzeitverhalten unterscheiden.

Algorithmus 1: $O(n^3)$.

13 ⟨* 1⟩+ ≡

```
algo1 <- function(X){
  n <- length(X)
  MaxSoFar <- 0
  for(i in 1:n){
     for(j in i:n){
        Sum <- 0
        for(k in i:j){
           Sum <- Sum + X[k]
        }
        MaxSoFar <- max(MaxSoFar,Sum)
     }
  }
  return(MaxSoFar)
}
```

Algorithmus 1 zeigt uns eine Brute-Force-Lösung, die unschwer verbessert werden kann. Die dritte Potenz ergibt sich durch die drei ineinander geschachtelten Schleifen.

Algorithmus 2: $O(n^2)$.

14 ⟨* 1⟩+ ≡

```
algo2 <- function(X){
  n <- length(X)
  MaxSoFar <- 0
  for(i in 1:n){
     Sum <- 0
     for(j in i:n){
        Sum <- Sum + X[j]
        MaxSoFar <- max(MaxSoFar,Sum)
     }
  }
  return(MaxSoFar)
}
```

Zwei Schleifen führen zu quadratischer Laufzeit, ein ähnlicher Zeitverbrauch erfordert Algorithmus 3.

Algorithmus 3: $O(n^2)$.

15 ⟨* 1⟩+ ≡

```
algo3 <- function(X){
  n <- length(X)
  CumArray[0] <- 0
  for(i in 1:n){
```

```
      CumArray[i] <- CumArray[i-1] + X[i]
   }
   MaxSoFar <- 0
   for(i in 1:n){
      for(j in i:n){
         Sum <- CumArray[i] - CumArray[i-1]
         MaxSoFar <- max(MaxSoFar,Sum)
      }
   }
   return(MaxSoFar)
}
```

Die Besonderheit von Algorithmus 3 liegt in der Idee, zunächst einen Vorbereitungsschritt zu machen, um dann mit `CumArray` elegant auf die Lösung zu kommen. Diese Idee ist sehr nah verwandt mit der Berechnung von Intervallwahrscheinlichkeit mittels der Differenz zweier Verteilungsfunktionswerte.

A. 6.3: Formuliere die gerade geschilderte Analogie aus. Begründe, dass das Ergebnis korrekt ist.

Der Algorithmus 4 demonstriert uns, dass es noch schneller geht:

Algorithmus 4: $O(n \log(n))$.

16 ⟨* 1⟩+ ≡

```
MaxSum <- function(X,l,u){
   if(missing(l)){l <- 1; u <- length(u); n <- length(X)}
   if(l>u) return(0)
   if(l==u) return(max(0,X[i]))
   m <- (l+u)/2
   Sum <- 0
   MaxToLeft <- 0
   for(i in (m+1):u){
      Sum <- Sum + X[i]
      MaxToRight <- max(MaxToRight,Sum)
      MaxCrossing <- MaxToLeft+MaxToRight
      MaxInA <- MaxSum(l,m)
      MaxInB <- MaxSum(M+1,u)
      return(max(MaxCrossing,MaxInA,MaxInB))
   }
}
```

Dieser Ansatz ist bedeutsam, da er uns auf die Idee von divide et impera bringt. Bei sehr vielen Problemen, die sich mit Baumstrukturen in Verbindung bringen lassen, führt dieses Prinzip zu Vorteilen. Notwendig ist in der vorliegenden Umsetzung der Einsatz von Rekursion. Diese Technik führt – wenn alles ok ist –, zu elegant formulierten Lösungen. In der Entwicklungsphase können sich jedoch gravierende Fehlersituationen einstellen.

Es sei erwähnt, dass die Komplexitätsklasse $O(n \log(n))$ als Zwischenklasse zwischen linear und quadratisch sehr bedeutsam und bspw. für verschiedene Sortieransätze typisch ist.

A. 6.4: Überlege, welche negativen Punkte die rekursive Lösung besitzt.

Geradezu enttäuschend mag sein, dass die beste Lösung sehr unkompliziert aussieht. Durch sie wird das gestellte Problem mit genau der richtigen Zange angefasst:

Algorithmus 5: $O(n)$.

17 ⟨* 1⟩+ ≡

```
algo5 <- function(X){
   n <- length(X)
   MaxSoFar <- 0
   MaxEndingHere <- 0
   for(i in 1:n){
      MaxEndingHere <- max(MaxEndingHere+X[i],0)
      MaxSoFar <- max(MaxSoFar,MaxEndingHere)
   }
   return(MaxSoFar)
}
```

A. 6.5: Suche in der Literatur Gesetze zum Umgang mit der O-Notation.

A. 6.6: Studiere Komplexitätsklassen: Ermittle mit Hilfe von A&D-Büchern typische Vertreter der verschiedenen Klassen.

7 Rekursion

7.1 Towers of Hanoi mit R

Problemstellung: Gegeben sind drei Stäbe, wobei der erste mit nach oben kleiner werdenden Scheiben versehen ist. Gesucht ist eine Abfolge von Bewegungen einzelner Scheiben, so dass hinterher der erste *Turm* sich auf dem zweiten Stab befindet. Dabei ist die Bedingung zu beachten, dass niemals größere auf kleineren zu liegen kommen dürfen.

Die berühmten Türme sind sehr einprägsam, um sich den Vorteil von rekursiven Lösungen zu verdeutlichen. Man kommt fast zu der Frage, wie man denn sonst eine Lösung des Problems überhaupt beschreiben kann. An späteren Beispielen lässt sich erkennen, dass eine rekursive Lösung für den Problemlöser wunderbar ist. Jedoch kann es in praxi sehr nützlich sein, die Verwaltungsoperationen der Rekursion selbst in die Hand zu nehmen und dadurch Zeit zu sparen.

A. 7.1: Wie lassen sich die Towers of Hanoi mit R abbilden? Versuchen Sie zunächst eigene Vorschläge zu entwerfen, bevor Sie die folgende Lösung ins Auge fassen.

Eine R-Umsetzung. Wir stellen die Türme durch drei Vektoren dar, die zu einer Liste zusammengefasst werden.

18 ⟨*initialisiere die Türme* 18⟩ ≡ ⊂ 21

```
if(!exists("n")) n <- 3
```

```
towers <- list(one=1:n,two=NULL, thr=NULL)
```

Eine Scheibenbewegung von `from` nach `to` kann schnell umgesetzt werden.

19 ⟨*bewege eine Scheibe* 19⟩ ≡ ⊂ 20

```
towers[c(to,from)] <- list(c(towers[[from]][1],towers[[to]]),towers[[from]][-1])
```

Mit Hilfe dieser Operation können wir jetzt eine Funktion definieren, die das Problem löst. Der Einschub ⟨*zeige Türme* 23⟩ nur die Zwischenergebnisse, ist also für den eigentlichen Algorithmus nicht relevant.

20 ⟨*definiere eine Funktion, die einen Turm von* `from` *nach* `to` *bewegt* 20⟩ ≡ ⊂ 21

```
move.tower <- function(towers, from, to, number=1){
   if(number==1 || length(towers[[from]])==1){
     ⟨bewege eine Scheibe 19⟩
   } else {
     third <- (1:3)[-c(to,from)]
     towers <- move.tower(towers, from=from,  to=third, number=number-1)
     towers <- move.tower(towers, from=from,  to=to,    number=1)
     towers <- move.tower(towers, from=third, to=to,    number=number-1)
   }
   ⟨zeige Türme 23⟩
   return(towers)
}
```

Wir wollen zur Demonstration die Funktion aufrufen. Vorher müssen wir die Initialisierungen erledigen.

21 ⟨*starte Demonstration* 21⟩ ≡

```
n<-5
⟨initialisiere die Türme 18⟩
⟨definiere eine Funktion, die einen Turm von from nach to bewegt 20⟩
⟨initialisiere die Graphik 22⟩
⟨zeige Türme 23⟩
move.tower(towers,1,2,n)
```

Kleine Restarbeiten. Für die Darstellung ist ein Graphik-Fenster erforderlich. Weiterhin werden ein paar schöne Farben gesucht.

22 ⟨*initialisiere die Graphik* 22⟩ ≡ ⊂ 21

```
par(mfrow=c(1,3,4,5,7,10,14,20)[n]*c(1,1),mai=rep(0,4))
hcols <- rev(heat.colors(n))
```

Zum Schluss ist noch die Darstellung umzusetzen.

23 ⟨*zeige Türme* 23⟩ ≡ ⊂ 20, 21

```
plot(NULL, xlim = c(0,4), ylim = c(0,n+1),xlab="",ylab="",axes=FALSE)
box(); delta<-0.05
for(i in 1:3){
    if(0<length(towers[[i]])){
      disks <- towers[[i]]; y <- 1 + length(disks) - seq(along=disks)
      segments(i-delta*disks,y,i+delta*disks,y,lwd=10,col=hcols[disks])
      text(i,y,disks)
    }
}
```

A. 7.2: Studiere zum Thema Rekursion: Wirth, N. (z.B. 1986): Algorithmen und Datenstrukturen.

Anwendungsbeispiel. Zur Durchführung des Wilcoxon-Rangsummentests muss man die Verteilung der Prüfgröße unter H_0 kennen. Gegeben seien die Stich-

proben $x_1, ..., x_m$ und $y_1, ..., y_n$. Sei W_N mit $N = n + m$ die Rangsumme der Stichprobe der x_i $x_1, ..., x_m$. In der zweiten verbleiben damit noch $n = N - m$ Werte. Mit $P(W_N = w) =: p_{m,n}(w)$ wollen wir die Verteilung von W_N unter H_0 bezeichnen:[1]

$$(m + n) \cdot p_{m,n}(w) = m \cdot p_{m-1,n}(w - N) + n \cdot p_{m,n-1}(w)$$

Dann ließen sich die Wahrscheinlichkeiten rekursiv ermitteln:

24 ⟨* 1⟩+ ≡

```
pwmn<-function(w,m,n){
  N <- m+n; max.rang.sum <- ((m*(2*n+m+1))/2
  if(m==1) return(1/N*(w               %in% 1:N))
  if(n==1) return(1/N*(max.rang.sum-w) %in% 1:N))
  1/N*( m*pwmn(w-N,m-1,n) + n*pwmn(w,m,n-1) )
}
pwmn(w=13,m=3,n=5)*56 # Beispielaufruf
```

A. 7.3: Untersuchen Sie die vorgestellte Lösung:

- Erklären Sie die einzelnen Lösungsschritte.
- Beurteilen Sie den Lösungsansatz.
- Beschreiben Sie, wie sich das Problem ohne Rekursion lösen lässt.

7.2 Fibonacci-Folge

A. 7.4: Studiere zur Rekursion noch einmal: Levitin, A. (2003): Kapitel 2.4–2.5

Lesevorschlag – auch für Erwachsene geeignet:
Hans Magnus Enzensberger (1999): Der Zahlenteufel.

7.3 Goldener Schnitt

Die Diskussion der Fibonacci-Folge steht in Verbindung mit dem goldenen Schnitt:
`http://www.netlibrary.com/urlapi.asp?action=summary&v=1&bookid=2009570`

1.6180339887498948482045868343656381177203091798 0...

A. 7.5: Studieren Sie, wo überall der goldene Schnitt eine Rolle spielt.

A. 7.6: Erkläre, auf welchen Wegen sich Potenzen des goldenen Schnittes ausrechnen lassen und vergleiche die Ansätze.

Beispielsweise gilt:

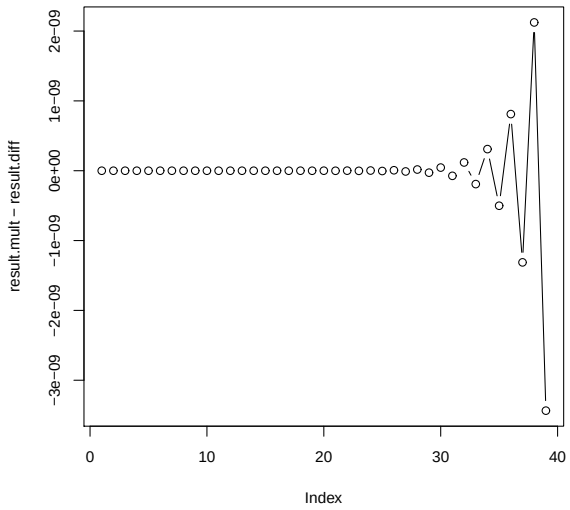
$$\phi^{n+1} = \phi \cdot \phi \cdots \phi, \qquad \phi^{n+1} = \phi^{n-1} - \phi^n$$

[1] Vgl.: Büning, H., Trenkler, G. (1979): Nichtparametrische statistische Methoden, de-Gruyter, S. 146.

Der Vergleich dieser beiden Ansätze führt zu Genauigkeitsfragen.

25 ⟨ * 1⟩+ ≡

```
phi <- (sqrt(5)-1)/2; n <- 39
vec <- c(phi,1)
for( i in 2:n){
  vec <- c(vec[2]-vec[1],vec)
}
result.diff <- rev(vec)[-1]
result.mult <- phi^(1:n)
plot(result.mult-result.diff,type="b")
```

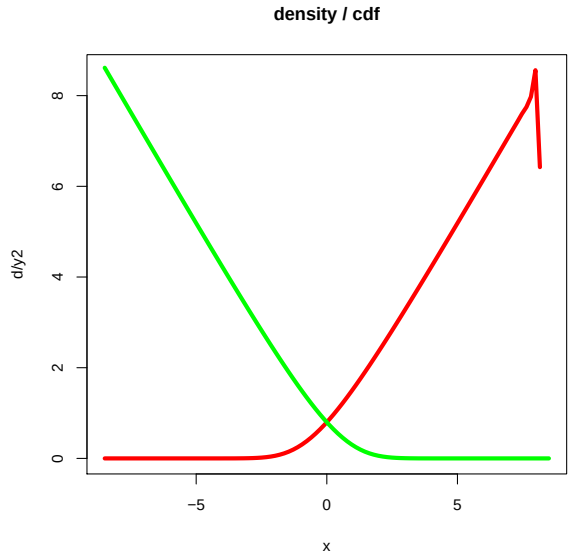


Frage: Können uns Probleme der Genauigkeit auch in statistischen Kontexten begegnen? Bemerkung: Genauigkeit der Berechnung von Dichten / Verteilungsfunktionen in R.

26 ⟨ * 1⟩+ ≡

```
x <- seq(-8.5,8.5,length=100)
y1 <- pnorm(x); y2 <- 1-y1; d <- dnorm(x)
par(mfrow=c(3,3))
plot (x, d/y2, type="l", col="red", lwd=4)
lines(x, d/y1, type="l", col="green", lwd=4)
title("density / cdf")
x <- seq(-10,10,length=100); df <- 450

for(df in c(200,250,300,350,450,500,700,900)){
  y1 <- pt(x,df=df); y2 <- 1-y1; d <- dt(x,df=df)
  plot (x, d/y2, type="l", col="red", lwd=4)
  lines(x, d/y1, type="l", col="green", lwd=4)
  title(paste("t-density / t-cdf\ndf =",df))
}
```



Diskussion von Rechengenauigkeiten, Fehlerfortpflanzung: siehe später.

Anwendung: Golden section search.

A. 7.7: Studieren Sie bei Wikipedia das Thema *golden section search*:
`http://en.wikipedia.org/wiki/Golden_section_search`

Unter der gerade angegebenen URL war im November 2010 ein Umsetzungsvorschlag zu sehen, der mit wenigen Handgriffen für R angepasst werden kann:

27 ⟨*definiere* `goldenSectionSearch()` 27⟩ ≡ ⊂ 30

```
phi <- (1 + sqrt(5)) / 2; resphi <- 2 - phi
imax <- 10; i <- 1; step <- 0.5; ystart <- 2
# x1 and x3 are the current bounds; the minimum is between them.
# x2 is the center point, which is closer to x1 than to x3
goldenSectionSearch <- function(fun, x1, x2, x3, absolutePrecision=0.0001){
    if(missing(x3)){ x3 <- x2; x2 <- x1 + resphi * (x3 - x1) }
    ⟨goldenSectionSearch: zeige Zustand 29⟩
    ⟨beende Suche bei zu großer Rekursionstiefe 28⟩
    if (abs(x1 - x3) < absolutePrecision){
      return((x1 + x3) / 2)
    }
    # Create a new possible center in the area between x2 and x3, closer to x2
    x4 = x2 + resphi * (x3 - x2)
    if(fun(x4) < fun(x2)){
      return(goldenSectionSearch(fun, x2, x4, x3, absolutePrecision))
    } else {
      return(goldenSectionSearch(fun, x4, x2, x1, absolutePrecision))
    }
```

```
}
```

Die Variable `imax` soll die Anzahl der Aufrufe begrenzen, um eventuelle Fehler abzufangen.

28 ⟨*beende Suche bei zu großer Rekursionstiefe* 28⟩ ≡ ⊂ 27

```
i <<- i + 1; if (imax < i) return( (x1 + x3) / 2 )
```

Zu Beginn jeden Aufrufs von `goldenSectionSearch()` wird die Graphik aktualisiert, die beim ersten Durchlauf konstruiert wird.

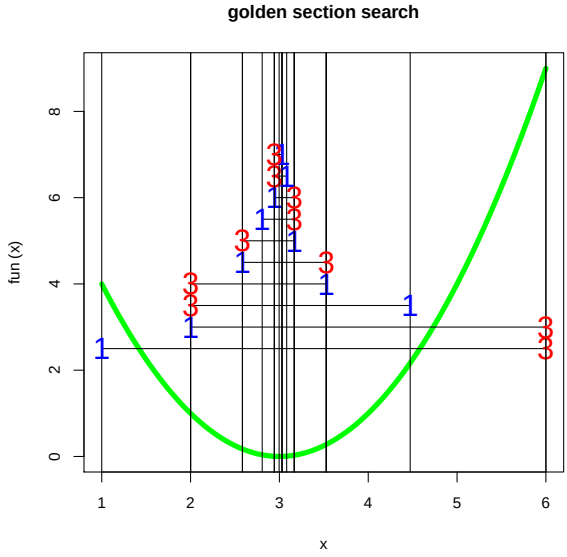
29 ⟨*goldenSectionSearch: zeige Zustand* 29⟩ ≡ ⊂ 27

```
print(c(i=i,x1=x1,x2=x2,x3=x3))
if(i==1) curve(fun,x1,x3,lwd=5,col="green", main="golden section search")
abline(v=c(x1,x2,x3)); segments(x1,ystart+i*step,x3,ystart+i*step)
text(x1,ystart+i*step,"1",col="blue",cex=2)
text(x3,ystart+i*step,"3",col="red",cex=2)
```

30 ⟨*teste Suche* 30⟩ ≡

```
⟨definiere goldenSectionSearch() 27⟩
fun <- function(x) (x-3)^2
goldenSectionSearch(fun, 1, 6)
```

```
i x1 x2 x3
1  1  2  6
       i       x1       x2       x3
2.000000 2.000000 3.527864 6.000000
       i       x1       x2       x3
3.000000 4.472136 3.527864 2.000000
       i       x1       x2       x3
4.000000 3.527864 2.944272 2.000000
       i       x1       x2       x3
5.000000 2.583592 2.944272 3.527864
       i       x1       x2       x3
6.000000 3.167184 2.944272 2.583592
       i       x1       x2       x3
7.000000 2.806504 2.944272 3.167184
       i       x1       x2       x3
8.000000 2.944272 3.029417 3.167184
       i       x1       x2       x3
9.000000 3.082039 3.029417 2.944272
        i        x1        x2        x3
10.000000  3.029417  2.996894  2.944272
Mon Dec 13 10:46:12 2010
[1] 2.986844
```



A. 7.8: Erklären Sie, wie der Vorschlag arbeitet.

8 Hartnäckige Probleme

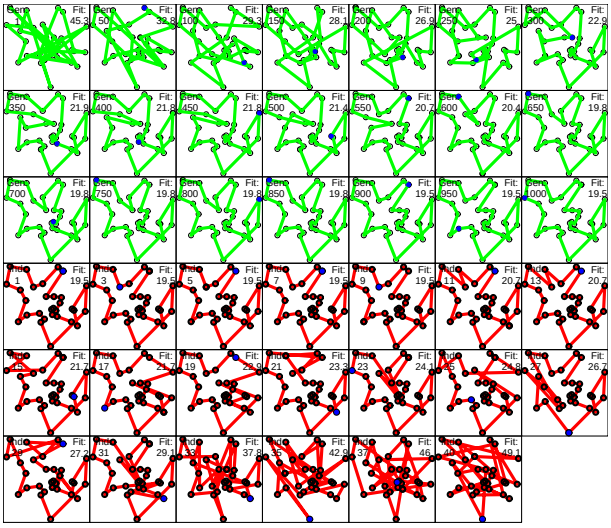
Was tun bei sehr schwierigen Problemen? Suche recht gute Lösungen. Suche nach neuen Wegen.

A. 8.1: Recherchieren Sie bei Wikipedia, was dort über das Handlungsreisenden-Problem zu finden ist.

8.1 Genetische Algorithmen

A. 8.2: Studieren Sie:
Wolf, H.P. (2010):
Handlungsreisenden-Problem per genetischem Algorithmus. Siehe:
`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/files/genetic-algos.pd`

A. 8.3: Studieren Sie bei Wikipedia, was sich dort über genetische Algorithmen finden lässt.

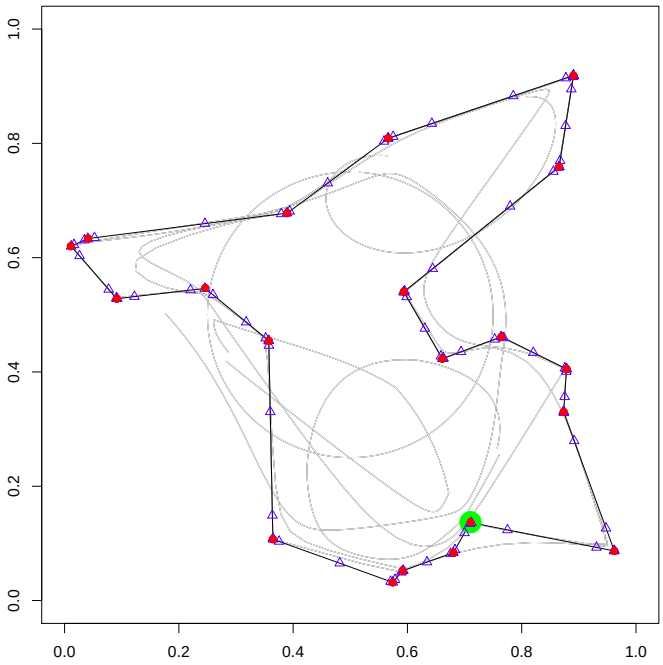


A. 8.4: Beschreiben Sie, wie sich per genetischem Ansatz Regressioinsmodelle mit einer größeren Anzahl an Koeffizienten an eine Punktewolke anpassen lassen.

A. 8.5: Überlegen Sie, wie man genetische Ansätze für die Clusteranalyse verwenden kann.

8.2 Kohonen-Karten

A. 8.6: Studieren Sie:
Wolf, H.P. (2009): Handlungsreisenden-Problem per Kohonen-Karte, siehe:
`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/fil`


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

9 Sortieren

9.1 Situationen und Prinzipien

Es fällt nicht schwer, Situationen zu nennen, in denen Sortieren relevant ist: Eisenbahnwagons, Bücher, CDs, Waren, Kunden, Klausuren, Internetstellen, Zahlen, Spielkarten.

Sortierprobleme lassen sich unterscheiden nach:

- den zu sortierenden Gegenständen
- dem zu verwendenden Sortierkriterium
- Richtung: auf- / absteigend
- den geforderten Genauigkeiten: genau / in Klassen

In Algorithmen finden wir in verschiedenen Gestaltung Prinzipien wieder, beispielsweise Folgende:

- durch Auswahl
- durch Einfügen
- durch Zählen
- durch Häufchenbildung
- durch Austauschen
- durch Mischen

Bemerkung: Sortieren kann für sich selbst ein spannendes Thema sein. Jedoch mögen die Ideen, die zu den einzelnen Ausgestaltungen führen, und deren Übertragung auf andere Situation noch relevanter sein.

A. 9.1: Blättern Sie in: D. E. Knuth (div Ed.): The Art of Computer Programming: 2. – Searching and Sorting. `FB 13 HI100 K74`

A. 9.2: Vergleichen Sie, wie Bücher zur A&D (Semesterapparat) das Thema Sortieren angehen und gliedern.

A. 9.3: Überlegen Sie, wo Sie selbst einmal etwas sortiert haben. Haben Sie sich dabei geschickt angestellt? An welcher Stelle wird Sortieren ökonomisch äußerst relevant sein?

9.2 Algorithmen fürs Sortieren durch Auswahl

Damit es praktisch etwas zu sortieren gibt, verschaffen wir uns einen Zufallsvektor der Länge 20.

31 ⟨*start* 31⟩ ≡
```
set.seed(13); a<-sample(1:20)
```

Diesen Vektor werden wir zum Sortieren als Beispielvektor verwenden.

```
[1] 15  5  8  2 16  1  9 10 11 20  7 18 14  4 13 17 19  6  3 12
```

Idee. Sortieren durch Auswahl geht von dem Gedanken aus, in der Menge der unsortierten Elemente zunächst das kleinste Element herauszusuchen. Dann kommt das zweit-kleinste Element an die Reihe und so fort, bis zum Schluss das größte Element der sortierten Folge hinzugefügt wird.

32 ⟨*start* 31⟩+ ≡

```
algo1 <- function(a,debug=FALSE){
   n <- length(a)
   for(i in 1:(n-1)){
      ⟨a1: bestimme Index k des kleinsten Elementes der a_i,...,a_n 33⟩
      ⟨a1: vertausche a_i und a_k 34⟩
      if(debug){cat("i:",substring(i+100,2),"/ a:",a,"\n")}
   }
   return(a)
}
# algo1(a,TRUE)
```

```
i: 01 / a: 1 5 8 2 16 15 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 02 / a: 1 2 8 5 16 15 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 03 / a: 1 2 3 5 16 15 9 10 11 20 7 18 14 4 13 17 19 6 8 12
i: 04 / a: 1 2 3 4 16 15 9 10 11 20 7 18 14 5 13 17 19 6 8 12
i: 05 / a: 1 2 3 4 5 15 9 10 11 20 7 18 14 16 13 17 19 6 8 12
i: 06 / a: 1 2 3 4 5 6 9 10 11 20 7 18 14 16 13 17 19 15 8 12
i: 07 / a: 1 2 3 4 5 6 7 10 11 20 9 18 14 16 13 17 19 15 8 12
i: 08 / a: 1 2 3 4 5 6 7 8 11 20 9 18 14 16 13 17 19 15 10 12
i: 09 / a: 1 2 3 4 5 6 7 8 9 20 11 18 14 16 13 17 19 15 10 12
i: 10 / a: 1 2 3 4 5 6 7 8 9 10 11 18 14 16 13 17 19 15 20 12
i: 11 / a: 1 2 3 4 5 6 7 8 9 10 11 18 14 16 13 17 19 15 20 12
i: 12 / a: 1 2 3 4 5 6 7 8 9 10 11 12 14 16 13 17 19 15 20 18
i: 13 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 16 14 17 19 15 20 18
i: 14 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17 19 15 20 18
i: 15 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 19 16 20 18
i: 16 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 17 20 18
i: 17 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 18
i: 18 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 19
i: 19 / a: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

33 ⟨a1*: bestimme Index k des kleinsten Elementes der* $a_i, \ldots, a_n$ 33⟩ ≡ ⊂ 32

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

34 ⟨a1*: vertausche* a_i *und* a_k 34⟩ ≡ ⊂ 32

```
a[k] <- a[i]; a[i] <- x
```

Was ergibt sich im Zusammenhang, also nach dem Expandieren die Code-Chunk-Namen?

35 ⟨* 1⟩+ ≡

```
algo1
```

Zusammen ergibt die folgende Funktion:

```
function(a){
   n <- length(a)
   for(i in 1:(n-1)){
      x <- a[i]; k <- i
      for(j in (i+1):n){
         if(a[j]<x){
            x <- a[j]; k <- j
         }
      }
      a[k] <- a[i]; a[i] <- x
   }
   return(a)
}
```

A. 9.4: Welche Vorteile bietet die literate bzw. die zusammenhängende Darstellung des Algorithmus?

A. 9.5: Überprüfen Sie, ob diese Funktion unseren Beispielvektor korrekt sortiert.

9.3 Algorithmen fürs Sortieren durch Einfügen

Idee. Sortieren durch Einfügen verwenden viele Kartenspieler. Sie nehmen die Karten einzeln auf und stecken sie an die passende Stelle der schon auf der Hand befindlichen Karten.

36 ⟨*start* 31⟩+ ≡

```
algo2 <- function(a,debug=FALSE){
   n <- length(a)
   for(i in 2:n){
      if(debug){cat("i:",substring(i+100,2),"/ a:",a,"\n")}
      ⟨a2: plaziere a_i geeignet in den sortierten Teilvektor ein 37⟩
   }
   return(a)
}
# algo2(a,TRUE)
```

Wir starten die Funktion und schauen, wie nach und nach von vorn der sortierte Teil beständig wächst.

```
i: 02 / a: 15 5 8 2 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 03 / a: 5 15 8 2 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 04 / a: 5 8 15 2 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 05 / a: 2 5 8 15 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 06 / a: 2 5 8 15 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 07 / a: 1 2 5 8 15 16 9 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 08 / a: 1 2 5 8 9 15 16 10 11 20 7 18 14 4 13 17 19 6 3 12
i: 09 / a: 1 2 5 8 9 10 15 16 11 20 7 18 14 4 13 17 19 6 3 12
i: 10 / a: 1 2 5 8 9 10 11 15 16 20 7 18 14 4 13 17 19 6 3 12
i: 11 / a: 1 2 5 8 9 10 11 15 16 20 7 18 14 4 13 17 19 6 3 12
i: 12 / a: 1 2 5 7 8 9 10 11 15 16 20 18 14 4 13 17 19 6 3 12
i: 13 / a: 1 2 5 7 8 9 10 11 15 16 18 20 14 4 13 17 19 6 3 12
```

```
i: 14 / a: 1 2 5 7 8 9 10 11 14 15 16 18 20 4 13 17 19 6 3 12
i: 15 / a: 1 2 4 5 7 8 9 10 11 14 15 16 18 20 13 17 19 6 3 12
i: 16 / a: 1 2 4 5 7 8 9 10 11 13 14 15 16 18 20 17 19 6 3 12
i: 17 / a: 1 2 4 5 7 8 9 10 11 13 14 15 16 17 18 20 19 6 3 12
i: 18 / a: 1 2 4 5 7 8 9 10 11 13 14 15 16 17 18 19 20 6 3 12
i: 19 / a: 1 2 4 5 6 7 8 9 10 11 13 14 15 16 17 18 19 20 3 12
i: 20 / a: 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 17 18 19 20 12
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Wir müssen noch die Einzelschritte des Algorithmus formulieren.

37 ⟨a2*: plaziere a_i geeignet in den sortierten Teilvektor ein* 37⟩ ≡ ⊂ 36
⟨a2*: bestimme den Index k im sortierten Teilvektor* $a_1, \ldots, a_i$ 38⟩
⟨a2*: verschiebe* $a_k, \ldots, a_i$ *um einen Platz und füge* a_i *ein* 39⟩

38 ⟨a2*: bestimme den Index k im sortierten Teilvektor* $a_1, \ldots, a_i$ 38⟩ ≡ ⊂ 37

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

39 ⟨a2*: verschiebe* $a_k, \ldots, a_i$ *um einen Platz und füge* a_i *ein* 39⟩ ≡ ⊂ 37

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

Wie sieht der zweite Algorithmus im Zusammenhang aus?

40 ⟨* 1⟩+ ≡

```
noquote(deparse(algo2))
```

```
 [1] function (a, debug = FALSE)
 [2] {
 [3]     n <- length(a)
 [4]     for (i in 2:n) {
 [5]         if (debug) {
 [6]             cat("i:", substring(i + 100, 2), "/ a:", a, "\\n")
 [7]         }
 [8]         k <- 1
 [9]         for (j in 1:(i - 1)) {
[10]             if (a[i] > a[j])
[11]                 k <- j + 1
[12]         }
[13]         h <- a[i]
[14]         if (i > k) {
[15]             for (j in (i - 1):k) {
[16]                 a[j + 1] <- a[j]
[17]             }
[18]         }
[19]         a[k] <- h
```

```
[20]     }
[21]     return(a)
[22] }
```

A. 9.6: Erkläre die Vorgehensweise von `algo2()` anhand der R-Anweisungen.

Der dritte Algorithmus stellt eine Variation der Idee *Sortieren durch Einfügen* dar. Daran sieht man, dass eine Idee nicht unbedingt in einer einzigen Lösung münden muss.

A. 9.7: Untersuche, was macht `algo3` anders macht als `algo2`?

41 ⟨*start* 31⟩+ ≡

```
algo3 <- function(a){
   n <- length(a)
   for(i in 2:n){
      k <- 1
      while(a[i]>a[k] && k<i){
         k <- k+1
      }
      h <- a[i]
      if(i>k){
         for(j in (i-1):k){
            a[j+1] <- a[j]
         }
      }
      a[k] <- h
   }
   return(a)
}
# algo3(a)
```

A. 9.8: Na, wie unterscheidet sich `algo4` von `algo3`?

42 ⟨*start* 31⟩+ ≡

```
algo4 <- function(a){
   n <- length(a)
   for(i in 2:n){
      k <- 1
      while(a[i]>a[k]){
         k <- k+1
      }
      h <- a[i]
      if(i>k){
         for(j in (i-1):k){
            a[j+1] <- a[j]
         }
      }
      a[k] <- h
   }
   return(a)
```

```
}
# algo4(a)
```

`algo5` benötigt gegenüber `algo4` nur zwei statt drei Schleifen-Konstrukte. Das wirkt eleganter.

43 ⟨*start* 31⟩+ ≡

```
algo5 <- function(a){
   n <- length(a)
   for(i in 2:n){
      j <- i-1
      h <- a[i]
      while(j>0 && a[j]>h){
         a[j+1] <- a[j]
         j <- j-1
      }
      a[j+1] <- h
   }
   return(a)
}
# algo5(a)
```

Idee. Der folgende Algorithmus geht davon aus, dass vor dem zu sortierenden Vektor der Platz mit dem Index 0 verwendet werden kann. Damit bekommt das erste Element die Rolle eines Wächters. Da in R `a[0]` nicht realisierbar ist, wird `a` vorn um ein Element verlängert und auf den neuen Vektor statt mit `a[i]` mit `a[adapt(i)]` zugegriffen.

44 ⟨*start* 31⟩+ ≡

```
algo6 <- function(a){
   n <- length(a)
   adapt <- function(i){i+1}
   a <- c(0,a)
   for(i in 2:n){
      j <- i-1
      a[adapt(0)] <- a[adapt(i)]
      while(a[adapt(j)]>a[adapt(0)]){
         a[adapt(j+1)] <- a[adapt(j)]
         j <- j-1
      }
      a[adapt(j+1)] <- a[adapt(0)]
   }
   return(a[-adapt(0)])
}
# algo6(a)
```

A. 9.9: Wählen Sie aus den vorgestellten Algorithmen 3 Exemplare aus und experimentieren Sie mit verschiedenen (kleinen) Inputs, um deren Unterschiede zu verdeutlichen.

A. 9.10: Fertigen Sie zu den Algorithmen `algo3` bis `algo6` literate Fas-

sungen an.

9.3.1 Shellsort.

Der folgende Algorithmus heißt Shellsort. Im innern Bereich erkennen wir Strukturen, die wir bereits kennen. Zusätzlich ist dieser Kern in eine Schleife eingebettet, so dass man sich über den Grund wundern kann.

45 ⟨*start* 31⟩+ ≡

```
algo7 <- shell.sort <- function(a){
   n <- length(a)
   h <- 1
   repeat{h <- 3*h+1;if(h>n) break}
   repeat{
      h <- floor(h/3)
      for(i in (h+1):n){
         v <- a[i]; jj <- i
         while(a[jj-h]>v){
            a[jj] <- a[jj-h]; jj <- jj-h
            if(jj<=h) break
         }
         a[jj] <- v
      }
      if(h==1) break
   }
   return(a)
}
# algo7(a)
```

A. 9.11: Vergleiche den Kern von `algo7` mit `algo5` für `h=1`. Ersetze dabei `jj` durch `j+1`. Begründen Sie dann, dass der Algoritmus wirklich seinen Input sortiert.

A. 9.12: Recherchiere im Internet, was der Witz an dem Algorithmus mit dem Namen Shellsort ist.

9.4 Algorithmen fürs Sortieren durch Zählen

Bei diesem Ansatz werden alle Zahlenpaare verglichen. Dazu müssen $\binom{n}{2}$ Vergleiche absolviert werden. Die Paarungen können den Einträgen einer oberen Dreiecksmatrix zugeordnet werden. In folgendem Algorithmus durchlaufen die beiden `for`-Schleifen gerade eine solche obere Dreiecksmatrix. Die Sortierung wird dann in linearer Zeit mit Hilfe des Zählergebnisses erledigt.

46 ⟨*start* 31⟩+ ≡

```
algo8 <- function(a){
   n <- length(a)
   b <- C <- rep(0,n)
   for(i in 1:n){
     C[i] <- 1
   }
```

```
   for(i in 1:(n-1)){
      for(j in (i+1):n){
         if(a[i]<a[j]){
            C[j] <- C[j]+1
         } else {
            C[i] <- C[i]+1
         }
      }
   }
   for(i in 1:n){
      b[C[i]] <- a[i]
   }
   return(b)
}
# algo8(a)
```

A. 9.13: Wie lässt sich der Median durch Zählen finden?

9.5 Sortieren ohne Vergessen

Ohne Vergessen meint, dass die *Urliste* der Daten nicht verloren gehen darf. Deshalb benötigt man für das Ergebnis zusätzlich einen gleich langen Vektor wie `a`. `algo9` stellt eine Variation von `algo6` dar.

47 ⟨*start* 31⟩+ ≡

```
algo9 <- function(a){
   n <- length(a); b <- c(0,a); adapt <- function(i){i+1}
   b[adapt(1)] <- a[1]
   for(i in 2:n){
      j <- i-1
      b[adapt(0)] <- a[i]
      while(b[adapt(j)] > b[adapt(0)]){
         b[adapt(j+1)] <- b[adapt(j)]
         j <- j-1
      }
      b[adapt(j+1)] <- b[adapt(0)]
   }
   return(b[-adapt(0)])
}
# algo9(a)
```

In `algo10` erkennen wir den Algorithmus `algo1`, der nach dem Prinzip *Sortieren durch Auswahl* arbeitet, wieder. Auf der Variablen `C` erkennen wir, wo ein Element anfangs gestanden hat.

48 ⟨*start* 31⟩+ ≡

```
algo10 <- function(a){
   n <- length(a); C <- a
   for(i in 1:n){
      C[i] <- i
   }
   for(i in 1:(n-1)){
```

```
      x <- a[i]; k <- i
      for(j in (i+1):n){
         if(a[j]<x){
            x <- a[j]; k <- j
         }
      }
      a[k] <- a[i]; a[i] <- x
      m <- C[i]; C[i] <- C[k]; C[k] <- m
   }
   return(rbind("Ergebnis"=a,"alter Index"=C))
}
# algo10(a)
```

Wir erhalten das folgende Ergebnis:

```
            [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
Ergebnis       1    2    3    4    5    6    7    8    9    10    11    12
alter Index    6    4   19   14    2   18   11    3    7     8     9    20
            [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
Ergebnis       13    14    15    16    17    18    19    20
alter Index    15    13     1     5    16    12    17    10
```

A. 9.14: Nennen Sie Situationen, in denen Vergessen im obigen Sinne keine gute Idee ist. Halten Datenbanken die ursprüngliche Reihefolge nach?

9.6 Beurteilung von Sortieralgorithmen

Nach der Diskussion der verschiedenen Ideen lässt sich die Qualität der Algorithmen hinterfragen. Wesentliche Größen sind:

- Anzahl der Schlüsselvergleiche
- Anzahl der Zuweisungen, bei denen die zu sortierenden Gegenstände `a[i]` bewegt werden
- Platzverbrauch

Für die Vergleiche und die Zuweisungen lassen sich wesentlich betrachten:

- best case
- worst case
- average case

Zur Durchschnittsberechnung ist natürlich eine Annahme über die Grundgesamtheit der zu sortierenden Fälle notwendig. Denn oft liegt eine gewisse Vorsortierung vor. Fast eher selten ist der Fall, dass alle Permutationen der Elemente mit gleicher Wahrscheinlichkeit vorkommen können.

A. 9.15: Vergleiche die Algorithmen bezüglich ihrer Schlüsselvergleiche und Zuweisungen.

A. 9.16: Überlegen Sie, was geeignete Antworten auf die Frage nach dem *average case* erfordert. Überlegen Sie auch, wie Sie per Experiment Durchschnittswerte ermitteln könnten.

Wir wollen diese Aufgabe anhand von drei oben vorgestellten Algorithmus exemplarisch betrachten.

Sortieren durch Zählen. Dieses Vorgehen wird nicht durch die Vorsortierung beeinflusst. Immer ergeben sich:

$$\frac{n^2 - n}{2}$$

Vergleiche und n Bewegungen.

49 ⟨*start* 31⟩+ ≡

```
algo8a <- function(a){
   n.assign <- n.vergleiche <- 0
   n <- length(a)
   b <- C <- rep(0,n)
   for(i in 1:n){
     C[i] <- 1
   }
   for(i in 1:(n-1)){
      for(j in (i+1):n){
         n.vergleiche <- n.vergleiche+1
         if(a[i]<a[j]){
            C[j] <- C[j]+1
         } else {
            C[i] <- C[i]+1
         }
      }
   }
   for(i in 1:n){
      n.assign <- n.assign + 1
      b[C[i]] <- a[i]
   }
###   return(b)
   result <- paste("n:",n,"/ Vergleiche:",n.vergleiche,"/ Zuweisungen:",n.assign)
   return(result)
}
# algo8a(1:20)
```

Hier der Output:

```
[1] "n: 20 / Vergleiche: 190 / Zuweisungen: 20"
```

Sortieren durch Auswahl. Für dieses Sortieren ist die Anzahl von Vergleichen ebenfalls vom Input unabhängig. Es werden

$$\frac{n^2 - n}{2}$$

Vergleiche benötigt. Die Anzahl der Bewegungen ist indes je nach Input unterschiedlich:

- best case (Bewegungen): $3(n-1)$
- worst case (Bewegungen): $3(n-1)+n^2/4$

50 ⟨*start* 31⟩+ ≡
```
algo1a <- function(a){
   n.assign <- n.vergleiche <- 0
   n <- length(a)
   for(i in 1:(n-1)){
      n.assign <- n.assign + 1
      x <- a[i]; k <- i
      for(j in (i+1):n){
         n.vergleiche <- n.vergleiche+1
         if(a[j]<x){
            n.assign <- n.assign + 1
            x <- a[j]; k <- j
         }
      }
      n.assign <- n.assign + 2
      a[k] <- a[i]; a[i] <- x
   }
###   return(a)
   result <- paste("n:",n,"/ Vergleiche:",n.vergleiche,"/ Zuweisungen:",n.assign)
   return(result)
}
# cat("best case", algo1a(1:20))
# cat("worst case", algo1a(20:1))
```

Was sagt uns der Praxistest?

```
best case n: 20 / Vergleiche: 190 / Zuweisungen: 57
worst case n: 20 / Vergleiche: 190 / Zuweisungen: 157
```

Sortieren durch Einfügen. Wenn der Input bereits aufsteigend sortiert ist, benötigt man sehr wenige Vergleiche bei folgendem Algorithmus.

51 ⟨*start* 31⟩+ ≡
```
algo6a <- function(a){
   n.assign <- n.vergleiche <- 0
   n <- length(a)
   adapt <- function(i){i+1}
   a <- c(0,a)
   for(i in 2:n){
      j <- i-1
      n.assign <- n.assign + 1
      a[adapt(0)] <- a[adapt(i)]
      n.vergleiche <- n.vergleiche+1
      while(a[adapt(j)]>a[adapt(0)]){
         n.vergleiche <- n.vergleiche+1
         n.assign <- n.assign + 1
         a[adapt(j+1)] <- a[adapt(j)]
```

```
         j <- j-1
      }
      n.assign <- n.assign + 1
      a[adapt(j+1)] <- a[adapt(0)]
   }
###   return(a[-adapt(0)])
   result <- paste("n:",n,"/ Vergleiche:",n.vergleiche,"/ Zuweisungen:",n.assign)
   return(result)
}
# cat("best case", algo6a(1:20))
# cat("worst case", algo6a(20:1))
```

Hier die empirischen Ergebnisse:

```
best case n: 20 / Vergleiche: 19 / Zuweisungen: 38
worst case n: 20 / Vergleiche: 209 / Zuweisungen: 228
```

Dieses passt zu folgenden Formeln für die Vergleiche ...

- best case (Vergleiche): $n - 1$
- worst case (Vergleiche): $\frac{n^2+n-2}{2}$

... und auch zu den Bewegungen:

- best case (Bewegungen): $2(n - 1)$
- worst case (Bewegungen): $\frac{n^2+3n-4}{2}$

A. 9.17: Überprüfe die Formeln!

9.7 Algorithmen fürs Sortieren durch Austausch

9.7.1 Bubblesort

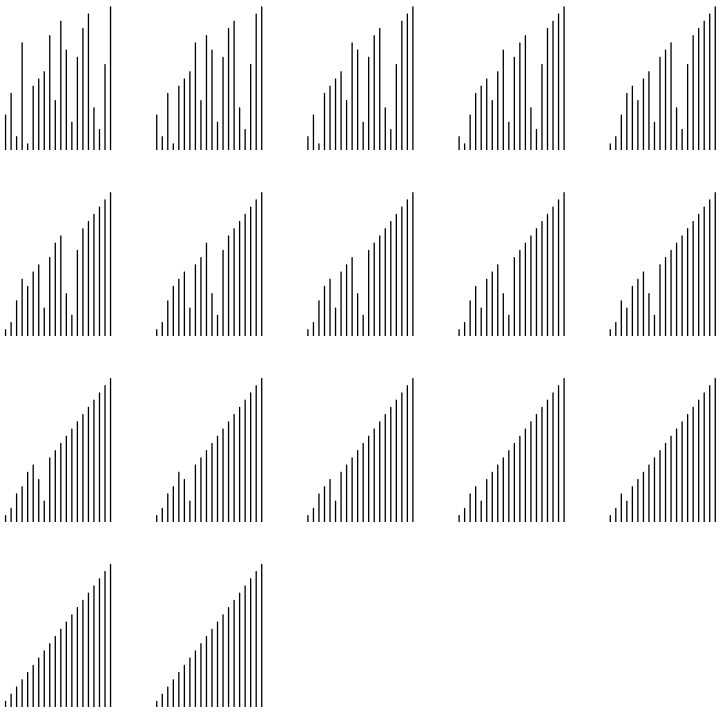
Der folgende Ansatz zeigt, dass man auch mit paarweisen Nachbarschaftsvergleichen / -Austauschen zum Ziel kommen kann.

Idee. Vorn beginnend wird ein Element solange mit seinem Nachbarn vertauscht, wie es relativ falsch steht.

52 ⟨*start* 31⟩+ ≡

```
algo11 <- bubble.sort <- function(a){
   n <- length(a)
   k <- n;      par(mfrow=c(4,5),mai=c(.3,.3,0,0))
   while(k!=1){
      l <- 1
      for(j in 1:(k-1)){
         if(a[j]>a[j+1]){
            l <- j; h <- a[j]
            a[j] <- a[j+1]; a[j+1] <- h
         }
```

```
      }
      k <- l;   plot(a,type="h",ylab="",xlab="",axes=FALSE)
   }
   return(a)
}
# algo11(a)
```



Damit sind die ganz elementaren Algorithmen abgehandelt.

A. 9.18: Was zeigt die Darstellung? Was fällt auf?

9.8 Algorithmen fürs Sortieren durch Mischen

Idee. Die Sortieraufgabe lässt sich erleichtern, indem die Menge der zu sortierenden Zahlen in zwei Teile geteilt wird. Diese Teile können von zwei unterschiedlichen Prozessoren evtl. nach unterschiedlichen Algorithmen sortiert werden. Anschließend müssen die Sortierergebnisse *zusammengemischt* werden. Wie man sich überlegen kann, wird diese Vorgehensweise Einsparungen bringen. Weitere Einsparungen folgen, wenn für jeden der zwei Teile wiederum die Idee umgesetzt wird. Letztlich bekommen wir ein rekursive Lösung des Problems: Mergesort. Eine rekursive Form von Mergesort ist schnell hingeschrieben, doch sind die Verwaltungsaufwände nicht zu unterschätzen. Insofern ist die etwas

unübersichtliche Fassung aus Rechnerperspekte überlegen. Für den Menschen besitzt dagegen die Rekursion Vorzüge.

53 ⟨*start* 31⟩+ ≡

```
mergesort <- function(a){
  # Abbruch
  if(1==(n<-length(a))) return(a)
  # Zerlegung
  m <- n %/% 2
  # Sortierung der Teile
  left  <- mergesort(a[1:m])
  right <- mergesort(a[(m+1):n])
  # Zusammenmischung
  i <- j <- 1; left <- c(left,Inf); right <- c(right,Inf)
  for(k in 1:n){
    if(left[i]<right[j]){
      a[k] <- left[i] ; i <- i+1
    } else {
      a[k] <- right[j]; j <- j+1
    }
  }
  return(a)
}
# mergesort(a)
```

Laufzeitverhalten Sei c_N die Anzahl der Schlüsselvergleiche bei N Elementen, wobei angenommen sei, dass $N = 2^l$ eine 2-er-Potenz ist. Dann gilt:

$$c_N = c_{\lfloor N/2 \rfloor} + c_{\lceil N/2 \rceil} + N \quad \text{für} \quad 2 \leq N, \qquad c_1 = 0$$

Hieraus folgt wegen $N = 2^l$:

$$c_N = c_{2^l} = c_{2^{l-1}} \cdot 2 + 2^l$$

und nach Division durch 2^l:

$$\frac{c_{2^l}}{2^l} = \frac{c_{2^{l-1}}}{2^l} \cdot 2 + 1 = \frac{c_{2^{l-1}}}{2^{l-1}} + 1$$

Den rechten Teil der Gleichung können wir nach dem selben Prinzip weiter auswerten und erhalten:

$$\frac{c_{2^l}}{2^l} = \frac{c_{2^0}}{2^0} \cdot 2 + l = l$$

Also gilt für die Anzahl der Vergleiche: $c_N = l \cdot 2^l = N \cdot \log_2 N$.

A. 9.19: Versuchen Sie für Bewegungen eine ähnliche Argumentation zu führen.

A. 9.20: Was ist an der Funktion `mergesort` aus der Perspektive von Zeitverbräuchen nicht sehr überzeugend?

Eine nicht rekursive Lösung soll dem Leser nicht vorenthalten werden.

54 ⟨*start* 31⟩+ ≡
```
algo12 <- function(a,debug=0){
   if(length(a)!=2^floor(log(length(a),base=2))) {
      return("ERROR: Inputlaenge keine 2er-Potenz\n")
   }
   if(debug>0){cat(" ");print(a)}
   if(debug==3){plot(sort(a),a);readline()}
   n <- length(a); a <- c(a,rep(0,n))
   ⟨a12: initialisiere einige Variablen 56⟩
   ⟨a12: schleife über p=1,2,4 usw. 57⟩
   ⟨a12: gebe Ergebnis aus 55⟩
}
# algo12(a[1:16])
```

55 ⟨*a12: gebe Ergebnis aus* 55⟩ ≡ ⊂ 54
```
if(!up)a <- a[-(1:n)]else a <- a[1:n]
return(a)
```

56 ⟨*a12: initialisiere einige Variablen* 56⟩ ≡ ⊂ 54
```
up <- T
p  <- 1
```

57 ⟨*a12: schleife über p=1,2,4 usw.* 57⟩ ≡ ⊂ 54
```
repeat{
  ⟨a12: initialisiere i,j,k,l in repeat-Schleife 58⟩
  h <- 1
  m <- n
  ⟨a12: mische einen Lauf von i und j nach k 59⟩
  up <- !up
  p  <- 2*p
  if(debug==1)print(a)
  if(debug==3){
    plot(sort(a[1:n]),a[(1:n)+(!up)*n])
    abline(v=0.5+(1:(n/p))*p)
    readline()
  }
  if(p>=n) break
}
```

58 ⟨*a12: initialisiere* i,j,k,l *in* repeat*-Schleife* 58⟩ ≡ ⊂ 57
```
   if(up){
     i <- 1;j <- n;k <- n+1;l <- n+n
   }else{
     k <- 1;l <- n;i <- n+1;j <- n+n
   }
```

59 ⟨*a12: mische einen Lauf von* `i` *und* `j` *nach* `k` 59⟩ ≡ ⊂ 57

```
repeat{
  m <- m-2*p; q <- p; r <- p
  while(q>0 & r>0){
     if(a[i]<a[j]){
        a[k] <- a[i]; k <- k+h; i <- i+1; q <- q-1
     }else{
        a[k] <- a[j]; k <- k+h; j <- j-1; r <- r-1
     }
  }
  ⟨a12: kopiere Rest 60⟩
  if(debug==2)print(a)
  h <- -h; t. <- k; k <- l; l <- t.
if(m==0) break}
```

60 ⟨*a12: kopiere Rest* 60⟩ ≡ ⊂ 59

```
if(q==0) {
     while(r>0){
       a[k] <- a[j];k <- k+h;j <- j-1;r <- r-1
     }
}else{
     while(q>0){
       a[k] <- a[i];k <- k+h;i <- i+1;q <- q-1
     }
}
```

A. 9.21: Studiere in der Literatur das Thema Laufzeitverhalten von Sortieralgorithmen.

9.9 Quicksort

Zur Einstimmung möge man in einem Interview mit Hoare die Stelle suchen, bei der er etwas zum Quicksort äußert:
http://cacm.acm.org/magazines/2009/3/21782-an-interview-with-car-hoare/fulltext

A. 9.22: Wie arbeitet der Algorithmus Quicksort? Verdient er seinen Namen? Recherchieren Sie diese Frage oder machen Sie selbst Laufzeittests.

Bei folgendem Vorschlag wird ein mittleres Element zur Orientierung verwendet.

61 ⟨*start* 31⟩+ ≡

```
algo14 <- quicksort <- function(a,l,r,debug=FALSE){
   n <- length(a)
   if(missing(l)){l <- 1; r <- n}
   if(l<r){
      if(debug){ cat("l",l,"r",r, "/ a =", a) }
      res <- partition(a,l,r,i,j, debug); a <- res$a; i <- res$i; j <- res$j
```

```
      a <- quicksort(a,l,j,debug)
      a <- quicksort(a,i,r,debug)
      ### if(debug){ cat("l",l,"r",r); print(a)  }
   }
   return(a)
}
partition <- function(a,l,r,i,j,debug=FALSE){
   k <- floor((l+r)/2)
   if(debug){ cat("l",l,"r",r,"a[k]",a[k]) }
   x <- a[k]
   i <- l; j <- r
   while(i<=j){
      ⟨a14: erhöhe i und verringere j 62⟩
   }
   result <- list(a=a,i=i,j=j); return(result)
}
# algo14(a, debug=TRUE)
```

62 ⟨*a14: erhöhe i und verringere j* 62⟩ ≡ ⊂ 61, 65

```
while(a[i]<x)i <- i+1
while(a[j]>x)j <- j-1
if(i<=j){
   ⟨a14: vertausche a_i, a_j 63⟩
   i <- i+1; j <- j-1
}
```

63 ⟨*a14: vertausche* a_i, a_j 63⟩ ≡ ⊂ 62

```
h <- a[i]; a[i] <- a[j]; a[j] <- h
```

Wir wollen uns die Ausgaben einmal anschauen.

```
l 1 r 20 / a = 15 5 8 2 16 1 9 10 11 20 7 18 14 4 13 17 19 6 3 12
l 1 r 20 a[k] 20
l 1 r 19 / a = 15 5 8 2 16 1 9 10 11 12 7 18 14 4 13 17 19 6 3 20
l 1 r 19 a[k] 12
l 1 r 11 / a = 3 5 8 2 6 1 9 10 11 4 7 18 14 12 13 17 19 16 15 20
l 1 r 11 a[k] 1
l 2 r 11 / a = 1 5 8 2 6 3 9 10 11 4 7 18 14 12 13 17 19 16 15 20
l 2 r 11 a[k] 3
l 2 r 3 / a = 1 3 2 8 6 5 9 10 11 4 7 18 14 12 13 17 19 16 15 20
l 2 r 3 a[k] 3
l 4 r 11 / a = 1 2 3 8 6 5 9 10 11 4 7 18 14 12 13 17 19 16 15 20
l 4 r 11 a[k] 9
l 4 r 8 / a = 1 2 3 8 6 5 7 4 11 10 9 18 14 12 13 17 19 16 15 20
l 4 r 8 a[k] 5
l 4 r 5 / a = 1 2 3 4 5 6 7 8 11 10 9 18 14 12 13 17 19 16 15 20
l 4 r 5 a[k] 4
l 6 r 8 / a = 1 2 3 4 5 6 7 8 11 10 9 18 14 12 13 17 19 16 15 20
l 6 r 8 a[k] 7
l 9 r 11 / a = 1 2 3 4 5 6 7 8 11 10 9 18 14 12 13 17 19 16 15 20
l 9 r 11 a[k] 10
l 12 r 19 / a = 1 2 3 4 5 6 7 8 9 10 11 18 14 12 13 17 19 16 15 20
l 12 r 19 a[k] 13
l 12 r 13 / a = 1 2 3 4 5 6 7 8 9 10 11 13 12 14 18 17 19 16 15 20
l 12 r 13 a[k] 13
```

```
l 14 r 19 / a = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 18 17 19 16 15 20
l 14 r 19 a[k] 17
l 14 r 16 / a = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 17 18 20
l 14 r 16 a[k] 15
l 17 r 19 / a = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 19 17 18 20
l 17 r 19 a[k] 17
l 18 r 19 / a = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 18 20
l 18 r 19 a[k] 19
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Quicksort ohne Rekursion In folgender Granularität sehen wir die Verwaltung der Stücke, die zu behandeln sind.

64 ⟨*start* 31⟩+ ≡

```
quicksort <- function(a, debug=TRUE){
  n <- length(a); y.max<-100; y<-1
  set.jobs <- list( c(1,n) )
  repeat{
    y<-y+1; if(y>y.max) break
    if( 0 == length(set.jobs) ) break
    # get job
    job <- set.jobs[[1]]; l <- job[1]; r <- job[2]
    set.jobs <- set.jobs[-1]
    ⟨handle job ab 65⟩
    if(debug) cat("job:",job,"/ pivot",x,"/ a[l:r]:",a[l:r])
    # put new jobs in set.jobs
    if(l < (i-1)) set.jobs <- c( set.jobs, list(c(l,  i-1)))
    if(i < r) set.jobs <- c( set.jobs, list(c(i,r)))
  }
  return(a)
}
# quicksort(a[1:20])
```

Ein Job beschreibt einen Ausschnitt aus dem Zahlenvektor, der zu behandeln ist.

65 ⟨*handle job ab* 65⟩ ≡ ⊂ 64

```
if(l < r){
  # kern von partition
  k <- floor((l+r)/2)
  ## if(debug){ cat("l",l,"r",r,"a[k]",a[k]) }
  x <- a[k]
  i <- l; j <- r
  while(i<=j){
    ⟨a14: erhöhe i und verringere j 62⟩
  }
}
```

Ob es geht? Hier ist der Output der Zwischenergebnisse:

```
job: 1 20 / pivot 20 / a[l:r]: 15 5 8 2 16 1 9 10 11 12 7 18 14 4 13 17 19 6 3 20
job: 1 19 / pivot 12 / a[l:r]: 3 5 8 2 6 1 9 10 11 4 7 18 14 12 13 17 19 16 15
job: 1 11 / pivot 1 / a[l:r]: 1 5 8 2 6 3 9 10 11 4 7
job: 12 19 / pivot 13 / a[l:r]: 13 12 14 18 17 19 16 15
job: 2 11 / pivot 3 / a[l:r]: 3 2 8 6 5 9 10 11 4 7
job: 12 13 / pivot 13 / a[l:r]: 12 13
job: 14 19 / pivot 17 / a[l:r]: 14 15 16 19 17 18
job: 2 3 / pivot 3 / a[l:r]: 2 3
```

```
job: 4 11 / pivot 9 / a[l:r]: 8 6 5 7 4 11 10 9
job: 14 16 / pivot 15 / a[l:r]: 14 15 16
job: 17 19 / pivot 17 / a[l:r]: 17 19 18
job: 4 8 / pivot 5 / a[l:r]: 4 5 6 7 8
job: 9 11 / pivot 10 / a[l:r]: 9 10 11
job: 14 15 / pivot 14 / a[l:r]: 14 15
job: 18 19 / pivot 19 / a[l:r]: 18 19
job: 4 5 / pivot 4 / a[l:r]: 4 5
job: 6 8 / pivot 7 / a[l:r]: 6 7 8
job: 9 10 / pivot 9 / a[l:r]: 9 10
job: 6 7 / pivot 6 / a[l:r]: 6 7
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

A. 9.23: Überlegen Sie, welche Bedeutung das *Pivot*-Element hat. Hätten Sie einen Vorschlag, wie man dieses jeweils bestimmen sollte?

9.10 Heap-Sort

Heap-Sort ist ein Beispiel dafür, dass Bäume im Geschäft des Sortierens helfen können.

66 ⟨*start* 31⟩+ ≡

```
sift <- algo16 <- function(a,l,r){
   i <- l; j <- 2*i; x <- a[i]
   while(j<=r){
      if(j<r){
         if(a[j]<a[j+1]) j <- j+1
      }
      if(x>=a[j]){
         break
      } else {
         a[i] <- a[j]; i <- j; j <- 2*i
      }
   }
   a[i] <- x
   return(a)
}
sift2 <- algo17 <- function(a,l,r){
   i <- l; j <- 2*i; x <- a[i]
   while(j<=r){
      if(j<r){
         if(a[j]>a[j+1]) j <- j+1
      }
      if(x<=a[j]){
         break
      } else {
         a[i] <- a[j]; i <- j; j <- 2*i
      }
   }
   a[i] <- x
   return(a)
}
```

67 ⟨*start* 31⟩+ ≡

```
algo18  <-  heapsort <- function(a){
   n <- length(a)
   ⟨a18: Gebe a eine Heapstruktur 68⟩
   ⟨a18: Sortiere 69⟩
   return(a)
}
# algo18(a)
```

68 ⟨*a18: Gebe* a *eine Heapstruktur* 68⟩ ≡ ⊂ 67

```
l <- floor(n/2)+1
while(l>1){
   l <- l-1
   a <- sift(a,l,n)
}
```

Führe beginned mit $k = n$ bis $k = 1$ folgende Aufgaben aus. Tausche das erste mit dem k-ten Element aus und stelle die Heap-Struktur innerhalb der ersten $k - 1$ Elemente wieder her.

69 ⟨*a18: Sortiere* 69⟩ ≡ ⊂ 67

```
r <- n
while(r>1){
   x <- a[1]; a[1] <- a[r]; a[r] <- x
   r <- r-1; a <- sift(a,1,r)
}
```

9.11 Topologisches Sortieren

Betrachten wir die beiden folgenden Situationen:

- Ermittlung einer zulässigen Belegungsreihenfolge von Kursen
- Ermittlung einer zulässigen Reihenfolge von Zellen einer Tabellenkalkulation, die nach Änderung von Einträgen aktualisiert werden müssen.

In beiden Fällen gibt es Abhängigkeiten, dass bestimmte Elemente vor anderen abgehandelt werden müssen. Gleichzeitig ist klar, dass bei solchen Fragen in der Regel mehrere Lösungen existieren. Die Sortieraufgabe enthält also Freiheitsgrade und ist zu finden unter der Überschrift *Topologisches Sortieren.* Siehe beispielsweise Cormen, Th., Leiserson, C.E., Rivest, R.L. (1992): Introduction to Algorithms, p. 485–487.

Zur Lösung erstellen wir einen Graphen und finden in der Menge der Algorithmen zu Graphen geeignete Vorschläge:

1. konstruiere einen Graphen, dessen Knoten die Elemente repräsentieren
2. zeichne für jede Abhängigkeitsbeding ein gerichtete Verbindung zwischen den betroffenen Knoten ein

Klar ist, das ein Zyklus ein Lösung der Problemstellung verhindert. Der konstruierte Graph muss also ein DAG (directed acyclic graph) sein.

Vorschlag 1:

- suche eine Quelle des Graphen
- notiere gefundenen Knoten
- entferne gefundene Knoten mit seinen (ausgehenden) Kanten
- falls noch ein Knoten im Graph ist, gehe zu Punkt 1 zurück

Vorschlag 2:

- suche eine Senke des Graphen
- notiere gefundenen Knoten vor der Menge der schon notierten Knoten
- entferne gefundene Knoten mit den Kanten, die auf ihn gerichtet sind
- falls noch ein Knoten im Graph ist, gehe zu Punkt 1 zurück

Hier folgt eine einfache Umsetzung der Idee.

70 ⟨* 1⟩+ ≡

```
topo.sort.by.removing.sinks <- function(A){
  result <- NULL; knot.set <- seq(nrow(A))
  # loop: search sink, print index and remove sink from A
  for(i in knot.set){
    # search sink
    for(z in knot.set){
       if(all(A[z,] <= 0)) { ind <- z; break } else ind <- NULL
    }
    # print knot of sink and store it in result
    result <- c(ind, result); cat("in step ",i,"sink founded is knot",ind)
    # remove sink from A
    #   by replacing row entries by  2 so that this row will not found again and
    #   by replacing col entries by -1 so that this col do not disturb the further searches
    A[ind,] <- 2; A[,ind] <- -1
  }
  # return result vector
  result
}
# define adjacency matrix as an example
A <- rbind(c(0,0,0,1,0,1),c(0,0,0,0,1,0),c(1,0,0,0,0,0),0,c(1,0,0,0,0,0),c(0,0,0,1,0,0))
print(A)
topo.sort.by.removing.sinks(A)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    1    0    1
[2,]    0    0    0    0    1    0
[3,]    1    0    0    0    0    0
[4,]    0    0    0    0    0    0
[5,]    1    0    0    0    0    0
[6,]    0    0    0    1    0    0
in step  1 sink founded is knot 4
```

```
in step  2 sink founded is knot 6
in step  3 sink founded is knot 1
in step  4 sink founded is knot 3
in step  5 sink founded is knot 5
in step  6 sink founded is knot 2
Tue Oct  1 14:10:16 2013
[1] 2 5 3 1 6 4
```

Bei beiden Vorschlägen liegt ein Problemfall (Zyklus) vor, wenn keine Quelle bzw. Senke gefunden wird, obwohl noch Knoten im Graph enthalten sind.

Bei Cormen et al. findet man den Vorschlag, zuerst eine DFS-Traversierung zu machen. Traversierungen sind Touren durch einen Graph, bei denen alle Knoten besucht werden. Bei einem Besuch können dann bestimmte Handlungen umzusetzen sein, wie notieren, nummerieren usw.

Eine DFS-Traversierung (depth first search) liefert eine Besuchstour, bei der aus der Perspektive der Quellen zunächst die am weitesten entfernten Knoten (die Senken) besucht werden. Demgegenüber versucht man bei einer Breitensuche zunächst alle Nachbarn (anfangs einer Quelle) abzuarbeiten.

Wir wollen einmal einen DFS-Ansatz nach Levitin, A. (): The design and analysis of algorithms, p. 162 ff, umsetzen. Eingearbeitet ist, dass die Enden der Sackgassen zurückgemeldet werden.

71 ⟨*definiere* DGS 71⟩ ≡ ⊂ 73, 74

```
DGS <- function(Adjazenzmatrix,levi=FALSE){
  ⟨definiere dfs 72⟩
  knoten.menge <- rep(0,dim(Adjazenzmatrix)[1])
  count <- 0; result <- NULL
  for( idx in seq(along=knoten.menge) ){
    if( knoten.menge[idx] == 0 ){
      result <- c(dfs(idx),result)
    }
    cat("Result",result)
  }
  cat("Result",result)
  result
}
```

72 ⟨*definiere* dfs 72⟩ ≡ ⊂ 71

```
dfs <- function(idx){
  count <<- count + 10
  cat("dfs-ort",idx)
  knoten.menge[idx] <<- count
  if(levi) text(xy[idx,1],xy[idx,2]+.1,as.character(count),col="red",cex=2)
  V <- which(Adjazenzmatrix[idx,]!=0)
  result <- idx
  if( 0<length(V)){
    for(w in V){
      if( knoten.menge[w] == 0 ) result<-c(result,dfs(w))
    }
  }
  return(result)
}
```

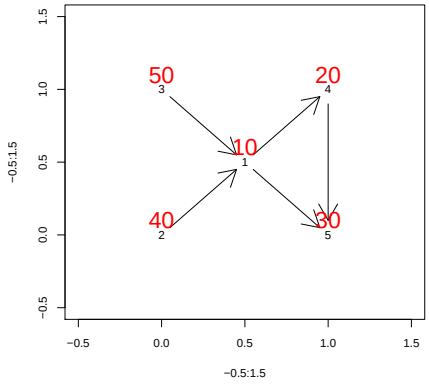
Hinweis: Die Nummerierung der Knoten weicht von Levitin ab. Andernfalls käme so etwas unklares wie 1, 2, 3, 4, 5 heraus.

73 ⟨*teste* DGS 73⟩ ≡

```
a <- matrix(0,5,5); a[1,3]<-a[2,3]<-a[3,4:5]<-a[4,5]<-1
a <- matrix(0,5,5); # a[5,3]<-a[2,3]<-a[3,c(4,1)]<-a[4,1]<-1
a[3,1]<-a[2,1]<-a[1,4:5]<-a[4,5]<-1
plot(-0.5:1.5,-0.5:1.5,col="white")
xy <- rbind(c(.5,0.5),c(0,0), c(0,1), c(1,1), c(1,0))
text(xy,as.character(1:5))
ind <- which(a==1); nach <- col(a)[ind]; von <- row(a)[ind]
eps <- 0.1*(xy[nach,]-xy[von,])
arrows(xy[von,1]+eps[,1],xy[von,2]+eps[,2],
       xy[nach,1]-eps[,1],xy[nach,2]-eps[,2])
```

⟨*definiere* DGS 71⟩

```
DGS(a,levi=TRUE)
```



Zuerst sollte 3 dann 2, 1, 4 und zuletzt 5 erledigt werden:

```
dfs-ort 1
dfs-ort 4
dfs-ort 5
Result 1 4 5
dfs-ort 2
Result 2 1 4 5
dfs-ort 3
Result 3 2 1 4 5
Result 3 2 1 4 5
Result 3 2 1 4 5
Result 3 2 1 4 5
Mon Dec 20 19:27:35 2010
[1] "ok"
```

Cormen et al., p. 486: In welcher Reihenfolge kann man sich anziehen?

74 ⟨* 1⟩+ ≡

```
a<-matrix(0,9,9)
```

```
rownames(a) <- colnames(a) <- c("belt","shirt","tie","socks","watch",
                                "shoes","undershorts","pants","jacket")
a["undershorts","shoes"]<-1
a["pants","shoes"]<-1
a["socks","shoes"]<-1
a["pants","belt"]<-1
a["undershorts","pants"]<-1
a["tie","jacket"]<-1
a["belt","jacket"]<-1
a["shirt","tie"]<-1
a["shirt","belt"]<-1
⟨definiere DGS 71⟩
rownames(a)[DGS(a)]
```

```
dfs-ort 1
dfs-ort 9
Result 1 9
dfs-ort 2
dfs-ort 3
Result 2 3 1 9
Result 2 3 1 9
dfs-ort 4
dfs-ort 6
Result 4 6 2 3 1 9
dfs-ort 5
Result 5 4 6 2 3 1 9
Result 5 4 6 2 3 1 9
dfs-ort 7
dfs-ort 8
Result 7 8 5 4 6 2 3 1 9
Result 7 8 5 4 6 2 3 1 9
Result 7 8 5 4 6 2 3 1 9
Result 7 8 5 4 6 2 3 1 9
[1] "undershorts" "pants"       "watch"       "socks"       "shoes"
[6] "shirt"       "tie"         "belt"        "jacket"
```

Die Reihenfolge klingt ganz praktikabel.

A. 9.24: Recherchieren Sie die Frage der Traversierung von Bäumen in der Literatur.

9.12 Bitonisches Sortieren

Hinweis: dieses Thema wurde nur kurz gestreift.

A. 9.25: Überlege bei den in der Vorlesung vorgestellten Algorithmen, an welchen Stellen sich etwas parallelisieren lässt.

Schritte zum Verständnis des bitonischen Sortierens. Zur Klärung wollen wir zwei Definitionen voranstellen.

- Eine *bitonische Folge* ist eine Folge, die aus zwei zusammenhängenden monotonen Teilfolgen besteht.

- Eine *compare-exchange-Einheit* (kurz: COMP-EXCH) ist eine Einheit die zwei Inputwerte auf- oder absteigend sortiert wieder ausgibt.

Der hier diskutierte Algorithmus läßt sich wie der Algorithmus *Sortieren durch Mischen* als aus aufeinander folgenden Stufen zusammengesetzt ansehen. Für das *bitonische Sortieren* werden zunächst kleine bitonische Folgen gebildet und sortiert. Die Sortierergebnisse werden wieder zu bitonischen Folgen zusammengefügt. Dieses Spiel geht solange, bis die Input-Folge sortiert ist.

Bei dem Algorithmus *Sortieren durch Mischen* bestand ein wesentlicher Schritt darin, zwei sortierte Folgen zu Mischen. Die Betrachtung dieses Vorgangs führt uns auf die Spuren der hier vorgestellten Version des *bitonischen Sortierens*.

Schritt 1: Mischen zweier sortierter Folgen. Das Beispiel zur Abbildung 40.2 (Sedgewick, Kapitel: Parallele Algorithmen, Perfektes Mischen) macht deutlich, daß es recht einfach ist, zwei sortierte Folgen zu einer sortierten Folge zusammenzumischen.

Schritt 2: Mischen zweier sortierter Folgen durch wiederholte identische Elementverschiebungen und wechselnden *compare-exchange*-Einheiten. In der Abbildung 40.3 (wieder aus dem Sedgewick) werden die zu mischenden Folgen in einem Vektor hintereinander abgelegt, wiederholt perfekt geshuffelt und mit *compare-exchange*-Einheiten behandelt. Die spezielle Art des Element-Wechsels wird als *perfect shuffling* bezeicht. Es läßt sich leicht überprüfen, daß dieselben Operationen ausgeführt werden wie im Beispiel 40.2. Leider ist nicht ganz klar, warum das so ist. Außerdem ist für eine Umsetzung störend, daß die *compare-exchange*-Einheiten an unterschiedlichen Indexpositionen eingesetzt werden müssen. Beides wird durch den im folgenden dargestellten Ansatz hoffentlich behoben.

Schritt 3: Sortieren einer bitonischen Folge. Eine bitonische Folge ist eine Folge, deren Elemente erst ansteigen und dann abfallen oder umgekehrt. Monotone Folgen sind Spezialfälle von bitonischer Folgen. Das Sortieren bitonischer Folgen geschieht mit *teile und compare-exchange*-Operationen.

> *Teile und compare-exchange*-Operation: Teile die bitonische Folge mit den Elementen $(a_1, \ldots, a_n)$ in der Mitte, behandele jeweils die Elementpaare $(a_i, a_{n/2+i})$ mit einer *compare-exchange*-Einheit und lege die Minima nacheinander auf $(a_1, \ldots, a_{n/2})$ und die Maxima auf $(a_{n/2+1}, \ldots, a_n)$ ab.

Dann gilt:

> *Eigenschaft 1:* Die berechneten Minima sind alle kleiner als das kleinste der Maxima.

Die Elemente lassen sich also so in die kleinere und die größere Hälfte zerlegen. Die beiden Hälften besitzen noch eine weitere interessante Eigenschaft.

Eigenschaft 2: Liegt bei einer zuerst ansteigenden und dann abfallenden bitonischen Input-Folge das Maximum in der Mitte, so sind die beiden Ergebnisfolgen wieder bitonische Folgen. Ist das Maximum nicht in der Mitte, so ergibt sich zumindest wieder eine bitonische Folge. Die zweite wird zu einer bitonischen Folge, wenn sie auf einem Kreis mit ihrem Endpunkt vor ihrem Anfangspunkt ablegt und dann der Kreis an geeigneter Stelle aufgetrennt wird.

Im ersten beschriebenen Fall ist klar, daß eine wiederholte Anwendung des geschilderten Prinzips: *teile und vergleiche* zum Schluß zu einer sortierten Folge führen muß. Aber auch in dem Fall, in dem das Extremum nicht in der Mitte liegt, führt das Prinzip zum Ziel. Dieses mache man sich am besten anhand einiger schöner Bleistift-Skizzen klar. Der folgende Schritt verdeutlicht, wie man das Sortieren einer bitonischen Folge mit perfect shuffling in Verbindung bringen kann.

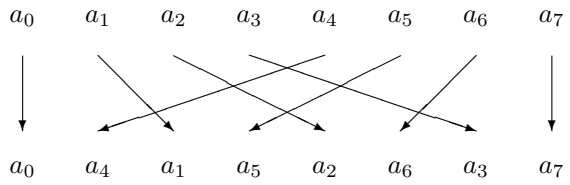
Schritt 4: Sortieren einer bitonischen Folge mit einem perfect-shuffle-Netzwerk. Betrachten wir das kleine Beispiel einer 8-elementigen bitonischen Folge. Dann werden im ersten Vergleichsschritt die Elemente mit folgenden Indizes verglichen. Als Laufbereich für die Indizes sei 0,. . .,7 angenommen.

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	4	000	–	100
1	–	5	001	–	101
2	–	6	010	–	110
3	–	7	011	–	111

Betrachtet man die binär notierten Indizes, so unterscheiden sich die der jeweils ersten Elemente, von denen der zweiten nur durch die erste Stelle, durch das erste Bit. Führt man auf den Indizes eine Bit-Shift-Operation aus, die alle Bits eine Stelle nach links schiebt und das erste wieder hinten anfügt, so werden die zu vergleichenden Elemente nebeneinander stehen.

Indexpositionen				
vorher		→	nachher	
arabisch	binär		binär	arabisch
0	000	→	000	0
1	001	→	010	2
2	010	→	100	4
3	011	→	110	6
4	100	→	001	1
5	101	→	011	3
6	110	→	101	5
7	111	→	111	7

Die Wanderung der Elemente läßt sich schnell auch graphisch darstellen:



Dieses ist das *perfect-shuffling*-Muster, das uns in der Abbildung 40.3 (Sedgewick) begegnet ist! Nach dieser Shuffle-Operation können benachbarte Elemente verglichen und ggf. vertauscht werden (*compare-exchange*). Würde man zwei weitere Shift-Operationen durchführen, ständen alle Elemente ohne die *compare-exchange*-Operation wieder genau an der ursprünglichen Stelle. Mit der *compare-exchange*-Operation sind jedoch nach dem zweimaligen Shuffeln in der ersten Hälfte die kleineren vier und in der zweiten die größeren vier Elemente zu finden, wobei beide Hälften wieder bitonische (oder fast bitonische) Folgen sind.

Die *teile und compare-exchange*-Operation läßt sich also erledigen durch folgende elementare Operationen:

SHUFFLE → COMP-EXCH → SHUFFLE → SHUFFLE

Der soeben durchgeführte Gedanke läßt sich wiederum auf die beiden bitonischen Hälften anwenden und damit beide Hälften in eine größere und eine kleinere zerlegen. Dieses wollen wir an dem Beispiel mit 8 Elementen verfolgen. Die folgende Tabelle zeigt, welche Elemente zu vergleichen sind.

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	2	000	–	010
1	–	3	001	–	011
4	–	6	100	–	110
5	–	7	101	–	111

Die zu vergleichenden Elemente unterscheiden sich nur in der zweiten Bitstelle. Wenn wir also zweimal nach links shiften, dann benachbarte mit *compare-exchange*-Einheiten vergleichen und ein weiteres Mal nach links shiften, würde für die beiden bitonischen Folgen der beiden Hälften gerade die *teile und compare-exchange*-Operation umgesetzt werden:

SHUFFLE → SHUFFLE → COMP-EXCH → SHUFFLE

Wir können natürlich die beiden *teile und compare-exchange*-Operationen Operationen verbinden und erhalten:

SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH → SHUFFLE

Als Zwischenergebnis haben wir nun vier bitonische Folgen (der Länge 2) ermittelt, wobei die Werte der ersten Folge kleiner als alle übrigen Werte, die Werte der zweiten kleiner als die Werte der dritten und vierten und die Werte der vierten größer als die ersten sechs Werte sind. Deshalb wird ein paralleler *compare-exchange*-Schritt zur sortierten Folge führen. Wieder folgt die Tabelle der zu vergleichenden Elemente:

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	1	000	–	001
2	–	3	010	–	011
4	–	5	100	–	101
6	–	7	110	–	111

Jetzt unterscheiden sich die Index-Bits der zu vergleichenden Elemente nur noch an der letzten Stelle. Es ist, wie schon klar war, kein Links-Shiften erforderlich.

Zusammengefaßt erhält man aus der bitonischen Folge mit 8 Elemente durch die folgenden elementaren Operationen eine sortierte Folge:

> SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH

Für bitonische Folgen der Länge 2^k benötigt man entsprechend $k = \log_2 2^k$ (SHUFFLE → COMP-EXCH) Schritte.

Damit können wir also problemlos bitonische Folgen sortieren. Nun ist noch zu klären, wie man dieses zur Sortierung nicht-bitonischer Folgen einsetzen kann.

Schritt 5: Der Gedanke zum Sortieralgorithmus Eine bitonische Folge läßt sich leicht aus einer aufsteigenden und einer absteigenden Folge zusammensetzen. Es ist also nur für eine bitonische Folge der Länge n das Problem zu lösen, zwei sortierte Folgen der Länge $n/2$ zu erstellen. Diese können wir natürlich mit dem gerade noch nicht ganz vorhandenen Sortieralgorithmus generieren.

Am Ende dieser Rekursion steht der Aufbau von bitonischen Folgen der Länge zwei, der offensichlich keinerlei Arbeit macht, da zwei Werte immer eine bitonische Folge bilden. Hieraus folgt umgekehrt das Vorgehen:

> bilde bitonische Folgen der Länge 2 → sortiere diese → bilde bitonische Folgen der Länge 4 → sortiere diese → bilde bitonische Folgen der Länge 8 → sortiere diese → ..., bis wir nur noch eine sortierte Folge vorliegen haben.

Der letzte Sortierschritt ist oben ausführlich beschrieben worden. Das Zusammenfügen von Folgen ist kein gedanklich schwieriger Akt. Was vielleicht etwas verwundert, ist, daß wieder perfect shuffling zum Einsatz kommt. Diesem Gedanken wollen wir uns im letzten Schritt zuwenden.

Schritt 6: Die Erstellung einer bitonischen Folge aus einer beliebigen. Nehmen wir wieder an, es liegen 8 unsortierte Werte vor. Dann ist das erste Ziel, 4 bitonische Folgen der Länge 2 zu erstellen und zu sortieren. Hierzu ist es nur erforderlich, die 8 Elemente als 4 Zweier-Folgen aufzufassen. Um 2 bitonische Folgen der Länge 4 zu erstellen, müssen wir jeweils zwei Werte abwechselnd auf- und absteigend sortieren. Dieses läßt sich leicht durch 4 *compare-exchange*-Einheiten bewerkstelligen, deren Sortier-Logik abwechselt. Ist die *compare-exchange*-Operation nur im Zusammenhang mit einer Shuffle-Operation erlaubt,

sind zwei Shuffle-Operationen sowie eine *Shuffle-compare-exchange*-Operation nötig.

Zur aufsteigenden Sortierung der ersten bitonischen Folge der Länge 4 würden wir nach den bisherigen Überlegungen

SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH

einsetzen. Jedoch haben wir 8 Elemente in Bearbeitung, wobei die anderen vier absteigend sortiert werden müssen und nach *compare-exchange*-Einheiten mit absteigender Sortier-Logik verlangen. Deshalb müssen wir vor die Sortiererei eine Shuffle-Operation vorschalten. Hierdurch wird das zweite Index-Bit zum ersten und durch die nächste Shuffle-Operation, wie gewünscht, an die letzte Stelle bewegt. Leider stehen die Elemente der ersten bitonischen Folge nach der ersten Shuffle-Operation nicht mehr zusammen, sondern es wechseln sich immer zwei Elemente der ersten bitonischen Folge mit zweien der zweiten ab. Deshalb ist die Sortier-Logik der *compare-exchange*-Einheiten auch zu alternieren. Für das abschließende SHUFFLE-COMP-EXCH stehen die Elemente der bitonischen 4-er Folgen wieder beeinander, so daß zwei *compare-exchange*-Einheiten mit aufsteigender Sortier-Logik neben zwei *compare-exchange*-Einheiten mit absteigender Logik stehen. Nach deren Einsatz erhält man eine bitonische Folge mit 8 Elementen.

Hat man am Anfang mehr als 8 Werte (16, 32, ...), dann erhöht sich mit jeder 2-er Potenz die Anzahl der *Stufen* und die Anzahl der Shuffle-Operationen in jeder Stufe um 1.

Übrigens sei angemerkt, daß bei dieser Konstruktion die inneren Extremwerte der bitonischen Folgen immer in der Mitte liegen.

Umsetzung: Der folgende Algorithmus setzt die obigen Gedanken um.

75 ⟨ * 1⟩+ ≡

```
shuffle<-function(x, op=rep(0,length(x)/2)){
           x <- matrix(x, length(x)/2, 2); incr<- x[,1]<x[,2]
           x[op== 1&!incr,]<-cbind(x[,2],x[,1])[op== 1&!incr,]
           x[op==-1& incr,]<-cbind(x[,2],x[,1])[op==-1& incr,]
           return(as.vector(t(x)))
}
shsort<-function(x,debug=F){
        n<-length(x); n.5<-n/2; lgn<-log(n,base=2); hh<-NULL
        for(i in 1:(lgn-1)){
           hh<-cbind(hh,matrix(0,n.5,lgn-i))
           for(j in 1:i)
             hh<-cbind(hh,as.vector(matrix(c(1,-1),2^(j-1),n.5/2^(j-1),T)))
        }
        hh<-cbind(hh,matrix(1,n.5,lgn))
        if(debug) print(hh)
        for(i in 1:dim(hh)[2])print(x<-shuffle(x,hh[,i]))
        return(x)
}
function(x=sample(1:16)){
# 1. Stufe
    for(i in 1:3){ x<-shuffle(x);print(x)}
```

```
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
# 2. Stufe
    for(i in 1:2){ x<-shuffle(x);print(x)}
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
        x<-shuffle(x,c(1,1,-1,-1,1,1,-1,-1));print(x)
# 3. Stufe
        x<-shuffle(x);print(x)
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
        x<-shuffle(x,c(1,1,-1,-1,1,1,-1,-1));print(x)
        x<-shuffle(x,c(1,1,1,1,-1,-1,-1,-1));print(x)
# 4. Stufe
    for(i in 1:4){ x<-shuffle(x,c(1,1,1,1,1,1,1,1));print(x)}
        invisible()
}
```

Literaturhinweis: Sedgewick, R. (2002): Algorithmen, Pearson-Studium / Addison-Wesley, Kapitel 40: Parallele Algorithmen.

10 Suchen

In diesem Kapitel werden Algorithmen zu dem Thema Suchen diskutiert. Situationen:

- Bücher, CDs, Brille, Schlüssel
- Telefon-Nummer zu Namen
- Variablen-Namen in Programmen
- Suche / Ersetze in Editoren
- Datensätze in Datenbanken, Rasterfandung
- Merkmalsträger mit bestimmten Eigenschaften
- Hauptkomponenten, Clustersuche, Suche nach Faktoren
- Nullstellen, Extrema, Lösung von Gleichungssystemen

A. 10.1: Überlegen Sie, an welchen Stellen Ihnen in der Statistik Suchprobleme begegnet sind.

Zur Einführung des Wahrscheinlichkeitsbegriffs gehen Darstellungen oft von Grundgesamtheiten und Stichprobenräumen (Ω) aus. Es werden Teilmengen von Ω betrachtet und mit diesen Operationen der Mengelehre diskutiert. Im Gleichmöglichkeitsmodell werden Elementanzahlen in Beziehung gesetzt und daraus Wahrscheinlichkeiten abgeleitet. Demgegenüber werden bei der empirischen Arbeit Datensätze ausgezählt und Häufigkeiten ermittelt. Doch mit zunehmender Größe besitzt das Auszählen auch seine Tücken. Der Zeitaufwand

für die Aufgabe: *Suche für die weiblichen Merkmalsträger des Datensatzes diejenigen, die bestimmte sonstige Eigenschaften, wie Körpergröße in einem Intervall und Haarfarbe* blond *erfüllen!* wächst mit der Größe des Datenbestandes und es ist bei wiederholten Fragen dieser Art relevant, wie man mit den Daten umgeht. In diesem Zusammenhang ist die Betrachtung des Umgang mit Indexoperationen unter der Verwendung von logischen Ausdrücken lohnenswert. Ebenfalls passt der Hinweis auf die R-Funktion `subset()`.

A. 10.2: Studiere die Hilfeseite zu der R-Funktion `subset()`.

Als zweite Situation stelle man sich vor, dass das mittlere Einkommen (arithmetisches Mittel oder Median) von Erwerbstätigen berechnet werden soll, die Daten aber fehlende Werte enthalten:

ID	Zeit	Monatseinkommen
...	...	
Meier	11/2009	1540
Meier	12/2009	NA
Meier	01/2010	1630
...	...	
Schulze	03/2008	2000
Schulze	04/2008	2100
Schulze	05/2008	NA
Schulze	06/2008	1900
Schulze	07/2008	NA
Schulze	08/2008	2200
...	...	

Wie lassen sich die fehlenden Werte ersetzen? Dabei kann man sich vorstellen, dass für die Ersetzung unterschiedliche Algorithmen zur Verfügung stehen.

Im folgenden werden ganz elementare Suchalgorithmen angesprochen, die sich zunächst auf nummerische Daten beziehen.

Eine weitere Klasse bilden Suchaufgaben, die sich auf Texte beziehen. Solche können beispielsweise in den Literaturwissenschaften, aber auch im Markting bei der Analyse von textlichen Stellungnahmen auftauchen. Kleinere Such- und Ersetzungsaufgaben wird jeder bei der Erstellung von Texten und Programmen mit Editoren kennen. Ein interessantes Beispiel ist der Knuth-Morris-Pratt-Algorithmus:

A. 10.3: Studiere: http://de.wikipedia.org/wiki/Knuth-Morris-Pratt-Algorithmus.

Wir gehen etwas später auf reguläre Ausdrücke ein, mit denen sich die Suche nach Elementen aus bestimmten Mengen formulieren und angehen lässt.

Im Falle größerer Datenbestände mögen diese in Datenbanken abgelegt sein, so dass Zugriffe auf bestimmte Teilbestände aus Anwendersicht mit Hilfe von SQL-Anweisungen formuliert und dann von der Datenbank in Form von Ergebnistabellen beantwortet werden. Hierzu stellt sich die Frage, wie es Datenbanken gelingt, so schnell zu sein. Beispielsweise ist es immer wieder erstaunlich, wie

schnell wir die Ergebnisse bei Anfragen an das eKVV der Universität Bielefeld erhalten. Eine Antwort lautet: *Index-Dateien*. Ein Blick auf Google kann hierzu erhellende Aspekte beisteuern.

Letztlich können unter der Überschrift *Suchen* die schon in der Auflistung genannten Suchprobleme Nullstellensuche und Optimumsuche thematisiert werden. Suchen ist also ein Abend füllendes Progamm.

10.1 Lineares Suchen

Der erste Algorithmus setzt eine lineare Suche um und liefert die Fundstelle(n). Algorithmus 1:

76 ⟨*start* 31⟩+ ≡
```
linear.search1<-function(x,A,n){
  if(missing(n)) n <- length(A)
  i <- 1
  while(i<=n && x!=A[i]){
    i <- i+1
  }
  result <- if(i<=n) paste(x,"in position",i) else paste(x,"not found")
  return(result)
}
# linear.search1(2,1:9)
```

Mit etwas Geschick lässt sich ein Vergleich pro Schleifen-Durchgang einsparen. Algorithmus 2:

77 ⟨*start* 31⟩+ ≡
```
linear.search2<-function(x,A,n){
  if(missing(n)) n <- length(A)
  A <- c(A,x)
  i <- 1
  while(x!=A[i]){
    i <- i+1
  }
  result <- if(i<=n) paste(x,"in position",i) else paste(x,"not found")
}
# linear.search1(10,1:9)
```

A. 10.4: Überlegen Sie, in welchen Situationen lineares Suchen sinnvoll ist und in welchen weniger brauchbar.

10.2 Binäres Suchen

In diesem Abschnitt wird davon ausgegangen, dass die Werte aufsteigend sortiert in `A` abgelegt sind. Die Vorarbeit des Sortierens führt zu Ersparnissen beim Suchen. Wir können bei der Suche zunächst die Mitte anschauen. Wenn das gesuchte Element kleiner ist als der Wert in der Mitte, suchen wir links weiter, sonst rechts. Durch Wiederholung der Idee liegt eine rekursive Lösung nahe.

10.2.1 Rekursive Lösung

Algorithmus 3:

78 ⟨*start* 31⟩+ ≡

```
binary.search <- function(x,A,low,high){
  if(missing(low)){low <- 1; high <- length(A)}
  if(low>high){
    found <- FALSE
  } else {
    m <- (low+high)%/%2  # entspricht: (low+high) div 2
    if(x==A[m]){
      found <- TRUE
    } else {
      if(x<A[m]){
        found <- binary.search(x,A,low,m-1)
      } else {
        found <- binary.search(x,A,m+1,high)
      }
    }
  }
  return(found)
}
```

A. 10.5: Modifizieren Sie `binary.search()` so, dass auch die Position ausgegeben wird.

10.2.2 Iterative Lösung

Rekursion erfordert viel Verwaltung. Deshalb wird eine iterative Lösung der rekursiven vorzuziehen sein.

Algorithmus 4:

79 ⟨*start* 31⟩+ ≡

```
binsuch <- function(a,key,size,platz=0,found){
  if(missing(size)){size <- length(a)}
  first <- 1; last <- size; loopende <- FALSE
  repeat{
    m <- (first+last)%/%2  # entspricht: (first+last) div 2
    if(key==a[m]){
      found <- TRUE; platz <- m; loopende <- TRUE
    } else {
      if(key<a[m]){
        if(m==first){
          found <- FALSE; loopende <- TRUE
        } else {
          last <- m-1
        }
      } else {
        if(m==last){
          found <- FALSE; loopende <- TRUE
        } else {
```

```
          first <- m+1
        }
      }
    }
    if(loopende) break
  }
  return(list(found=found,platz=platz))
}
```

A. 10.6: Überlegen Sie, ab wann es sich lohnt, die Daten, in denen ein Datum gesucht wird, zu sortieren und wann nicht.

10.3 Suchbäume

Suchbäume enthalten die Information in einer Form, dass die Inhalte, die nach dem Sortierkriterium vor den Eintrag eines Knotens gehören, in seinem linken Unterbaum stehen müssen. Die Einträge, die größer sind als der Knoten-Wert, müssen – wenn abgelegt – im rechten Teilbaum des Knotens stehen.

Algorithmus 6: Tree search.

80 ⟨*start* 31⟩+ ≡
```
⟨define IS.NIL 92⟩
tree.search <- function(x,k){
  if(IS.NIL(x)||k==key(x)){
    return(x)
  }
  if(k<key(x)){
    return(tree.search(left(x),k))
  } else {
    return(tree.search(right(x),k))
  }
}
```

`NIL` zeigt an, dass es keinen Unterbaum gibt, also `x` ein Blatt ist. Die Funktionen `left()` und `right]()` liefern die jeweiligen Unterbäume. `key()` gibt uns die in einem Knoten abgelegte Information aus.

A. 10.7: Diskutieren Sie den Unterschied zwischen Binärer Suche und dem Suchen mit Binärbäumen.

A. 10.8: Überlegen und Begründen Sie, in welche Komplexitätsklassen die Algorithmen zum Suchen gehören.

Für Versuche benötigen wir ein Beispiel.

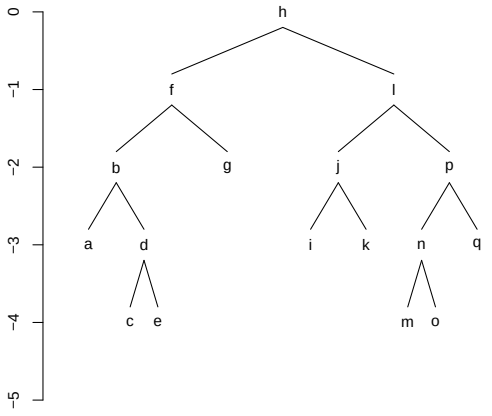
81 ⟨*start* 31⟩+ ≡
```
⟨definiere haenge.teilbaum.ein 93⟩
baum<-list("h",NIL,NIL)
baum<-haenge.teilbaum.ein(baum, list("f",NIL,NIL),"l")
baum<-haenge.teilbaum.ein(baum, list("b",NIL,NIL),"ll")
```

```
baum<-haenge.teilbaum.ein(baum, list("g",NIL,NIL),"lr")
baum<-haenge.teilbaum.ein(baum, list("a",NIL,NIL),"lll")
baum<-haenge.teilbaum.ein(baum, list("d",NIL,NIL),"llr")
baum<-haenge.teilbaum.ein(baum, list("c",NIL,NIL),"llrl")
baum<-haenge.teilbaum.ein(baum, list("e",NIL,NIL),"llrr")
baum<-haenge.teilbaum.ein(baum, list("l",NIL,NIL),"r")
baum<-haenge.teilbaum.ein(baum, list("j",NIL,NIL),"rl")
baum<-haenge.teilbaum.ein(baum, list("p",NIL,NIL),"rr")
baum<-haenge.teilbaum.ein(baum, list("i",NIL,NIL),"rll")
baum<-haenge.teilbaum.ein(baum, list("k",NIL,NIL),"rlr")
baum<-haenge.teilbaum.ein(baum, list("n",NIL,NIL),"rrl")
baum<-haenge.teilbaum.ein(baum, list("q",NIL,NIL),"rrr")
baum<-haenge.teilbaum.ein(baum, list("m",NIL,NIL),"rrll")
baum<-haenge.teilbaum.ein(baum, list("o",NIL,NIL),"rrlr")
```

Nun lässt sich der Baum zur Kontrolle plotten.

82 ⟨ * 1⟩+ ≡

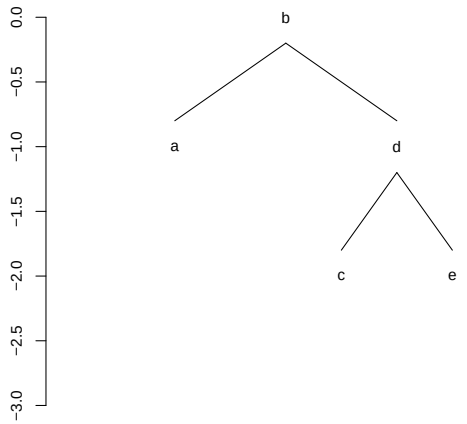
```
cat("PLOT vom Baum: baum\n")
plotbaum(baum)
```



Wir wollen in unserem Beispielbaum einmal `b` suchen.

83 ⟨ * 1⟩+ ≡

```
result <- tree.search(baum,"b")
plotbaum(result)
```



A. 10.9: Überlegen Sie die Bedeutung der Balanziertheit für die Schnelligkeit des Auffindens von Elementen.

A. 10.10: Recherchieren Sie in der Literatur, wie sich nicht balanzierte Bäume balanzieren lassen.

Der Test.

84 ⟨ * 1⟩+ ≡
```
cat("TEST: tree.search(baum,k)\n")
print("Gib Buchstaben zwischen a und q ein!")
k <- readline()
result <- tree.search(baum,k)
cat(result[[1]],"gefunden\n")
plotbaum(result)
```

Algorithmus 9. Dasselbe ohne Rekursion.

85 ⟨*start* 31⟩+ ≡
```
search.not.recur<-function(baum,k){
  while((!IS.NIL(baum)) & k!=key(baum)){
     if(k<key(baum)){
       baum<-left(baum)
     }else{
       baum<-right(baum)
     }
  }
  return(baum)
}
```

Der Test.

86 ⟨ * 1⟩+ ≡
```
cat("TEST: search.not.recur(baum,k)\n")
```

```
print("Gib Buchstaben zwischen a und q ein!")
k <- readline()
result <- search.not.recur(baum,k)
cat(result[[1]],"gefunden\n")
plotbaum(result)
```

Eine Spezialaufgabe lautet: ermittle das größte Element eines Baumes:

Algorithmus 10: Suche den maximalen Knoten.

87 ⟨*start* 31⟩+ ≡

```
max.elem<-function(baum){
  while(!IS.NIL(right(baum))){
     baum<-right(baum)
  }
  baum
}
```

Der Test.

88 ⟨* 1⟩+ ≡

```
cat("TEST: max.elem(baum)\n")
cat(max.elem(baum)[[1]],"gefunden\n")
```

Algorithmus 11: Suche den minimalen Knoten.

89 ⟨*start* 31⟩+ ≡

```
min.elem<-function(baum){
  while(!IS.NIL(left(baum))){
     baum<-left(baum)
  }
  baum
}
```

Der Test.

90 ⟨* 1⟩+ ≡

```
cat("TEST: min.elem(baum)\n")
cat(min.elem(baum)[[1]],"gefunden\n")
```

10.3.1 Implementierung der verwendeten Objekte

Für die vorgestellten Algorithmen und auch für Traversierungsalgorithmen benötigen wir ein paar kleine Hilfsfunktionen.

91 ⟨*start* 31⟩+ ≡

```
left <- function(baum){
  if(IS.NIL(baum[[2]])){
    return(NA)
  }else{
    return(baum[[2]])
  }
}
right <- function(baum){
  if(IS.NIL(baum[[3]])){
```

```
    return(NA)
  }else{
    return(baum[[3]])
  }
}
key <- function(baum){
    if(!IS.NIL(baum)) return(baum[[1]]) else return(NIL)
}
```

Technische Erfordernisse. Auf der bisherigen Abstraktionsebene sollte es keine großen Schwierigkeiten geben. Für eine praktische Demonstration mit R müssen jedoch noch ein paar Dinge bereitgestellt werden. `NIL` muss innerhalb von R natürlich einen Wert haben. Außerdem wird die Funktion `IS.NIL` verwendet. Diese sei ohne weitere Kommentare ergänzt:

92 ⟨*define* `IS.NIL` 92⟩ ≡ ⊂ 80

```
NIL <- NA
IS.NIL <- function(baum){
   if(!is.list(baum)) return(TRUE)
   if(is.na(baum[[1]])) return(TRUE)
   # if(all(is.na(unlist(baum[-1])))) return(TRUE)
   return(FALSE)
}
```

Für die Speicherung eines Baumes benötigen wir eine Struktur, in der wir ihn ablegen können.

Wir wollen einen Baum als dreielementige Liste (Wurzelknoteninhalt, linker Teilbaum, rechter Teilbaum) ablegen. Ein Teilbaum wird ebenso als Liste gehalten. Existiert kein Unterbaum, so wird das Element auf `NIL` gesetzt. Wir definieren eine kleine Funktion, mit der man in einem Baum an der Stelle, die durch `pfad` gekennzeichnet ist, einen Teilbaum `tb` einhängen kann. `pfad[i]=="l"` bedeutet: *gehe auf dem* `i`*-ten Pfadstück nach links*, `pfad[i]=="r"` bedeutet: *gehe auf dem* `i`*-ten Pfadstück nach rechts*.

93 ⟨*definiere* `haenge.teilbaum.ein` 93⟩ ≡ ⊂ 81

```
haenge.teilbaum.ein <- function(baum,tb,pfad){
  if(length(pfad)==1 && nchar(pfad)>1)
    pfad<-substring(pfad,1:nchar(pfad),1:nchar(pfad))
  wo <- c("[[2]]","[[3]]")[1+(pfad=="r")]
  eval(parse(text=paste("baum",paste(wo,collapse=""),"<-tb")))
  return(baum)
}
```

Eine zweite Funktion muss her, um einen Baum zu zeichnen. Dazu wird zuerst die Tiefe bestimmt.

94 ⟨*start* 31⟩+ ≡

```
tiefe<-function(baum,counter=0){
  l <- r <- 0
  if(!IS.NIL(baum[[2]])) l <- 1+tiefe(baum[[2]],counter)
  if(!IS.NIL(baum[[3]])) r <- 1+tiefe(baum[[3]],counter)
```

```
  return(counter+max(l,r))
}

plotknoten <- function(baum,x,y,depth){
  text(x,y,baum[[1]])
  ltb <- baum[[2]]
  rtb <- baum[[3]]
  if(!IS.NIL(ltb)){
    xneu <- x-2^(depth+y)
    plotknoten(ltb,xneu,y-1,depth)
    segments(x,y-.2,xneu,y-.8)
  }
  if(!IS.NIL(rtb)){
    xneu <- x+2^(depth+y)
    plotknoten(rtb,xneu,y-1,depth)
    segments(x,y-.2,xneu,y-.8)
  }
  if(1==length(unlist(ltb))){
    if(!is.na(ltb[[1]])){
      xneu <- x-2^(depth+y)
      text(xneu,y-1,ltb)
      segments(x,y-.2,xneu,y-.8)
    }
  }
  if(1==length(unlist(rtb))){
    if(!is.na(rtb[[1]])){
      xneu <- x+2^(depth+y)
      text(xneu,y-1,rtb)
      segments(x,y-.2,xneu,y-.8)
    }
  }
}
plotbaum <- function(baum){
  depth <- tiefe(baum)
  ylim <- c(-depth-1,0)
  xlim <- c(-1,1)*2^(depth+1)
  plot(1,xlim=xlim,ylim=ylim,type="n",xlab="",ylab="",bty="n",axes=FALSE)
  axis(2)
  plotknoten(baum,0,0,depth)
}
```

Wesentlicher Zweck der Vorstellung der Funktionen war zu zeigen, dass man die Idee binärer Suchbäume in die Tat umsetzen kann.

10.4 Hashing

Wie lassen sich Dinge schneller als beispielsweise mittels Suchbäumen finden? Antwort: Man lege sie dort ab, wo sie hingehören. Dann kann man fast blind auf sie zugreifen. Hashing setzt diesen Gedanken um, indem aus einem Objekt mit einer einfach zu berechnenden Hash-Funktion eine Adresse ermittelt wird, nämlich die Adresse, an die das Objekt hingehört. Ist ein Speicher der Größe S gegeben, dann muss die Menge der möglichen Objekte auf die Adressmenge

des Speichers, beispielsweise auf die Zahlen von 0 bis $S-1$ abgebildet werden. Das ganze funktioniert nur (gut), wenn die Anzahl der abzulegenden Objekte deutlich kleiner als S ist. Die Menge der theoretisch möglichen Objekte kann dagegen fast beliebig groß sein.

Weiter ist für das Funktionieren die Frage zu lösen, was passiert, wenn mehrere Kandidaten einem Platz zugeordnet werden, was im Rahmen der aufgelisteten Bedingungen passieren kann. Soll ein Objekt auf einem schon belegten Platz abgelegt werden, entsteht eine Kollision, und es muss eine Ausweichstrategie ergänzt werden:

- probiere einfach den nächsten Platz → linear probing
- wende eine weitere (andere) Hash-Funktion an, um einen freien Platz zu finden
- lasse ausgehend von jedem Speicherplatz eine lineare Liste starten

A. 10.11: Suche nach Anwendungen für die Idee des Hashens.

A. 10.12: Recherchiere in der Literatur nach den drei Ausweichstrategien.

A. 10.13: Argumentiere, dass sich beim *linear probing* Blöcke bilden, die bei einer zunehmenden Füllung des Speichers zu erheblichen Verlangsamungen führen.

10.5 Wie sortiert Google so schnell seine Inhalte?

Diese Frage hat sicher was mit Sortieren zu tun. Aber was?

A. 10.14: Lesen Sie: http://www.zeit.de/2005/41/Suchmaschinen_2

A. 10.15: Studieren Sie:
http://www.emeriten.ethz.ch/doku2009/Google-Emer-CAZ.pdf

11 Reguläre Ausdrücke

Dieses Thema könnte in den Abschnitt *Suchen* aufgenommen werden, doch soll aufgrund seiner vielfältigen Bedeutung ein eigenes Kapitel herhalten. Gegenstand ist die formale Beschreibung von Mengen oder Suchmustern beispielsweise für das Problem des pattern matching.

11.1 Motivation

Jeder Editor verfügt über Such- und Such-Ersetze-Funktionen. Wenn nach einer Zeichenkette gesucht werden soll, für die nur bestimmte Eigenschaften bekannt sind, wird eine Sprache erforderlich, mit der die Menge der Kandidaten beschreibbar ist.

Als zweites Beispiel kann das Problem der Überprüfung eines zulässigen Variablennamens dienen. In den meisten Zusammenhängen ist unmöglich, die Menge der möglichen Namen durch Aufzählen zu beschreiben. Die Überlegung, daß in unterschiedlichen Zusammenhängen im Prinzip immer wieder dasselbe Problem für verschiedene Mengen gelöst werden müßte, führt zu dem Verlangen, Zeichenkettenmengen durch eine einfache Sprache zu beschreiben und für Kandidaten ihre Zugehörigkeit festzustellen.

Eine Lösung für solche Fragestellung stellen *reguläre Ausdrücke* dar.

11.2 Beispiele

Je nach Standard variiert die Sprache zur Beschreibung der oben beschriebenen Mengen. Wir wollen dieses jedoch nicht thematisieren, sondern reguläre Ausdrücke aus einer erkundenden Sicht kennenlernen. Darum werden wir uns auch Schritt für Schritt an eine entsprechende Beschreibungssprache herantasten.

Als erstes Beispiel betrachten wir die Menge

$$\{a, aa, aaa, aaaa, aaaaa\}$$

Diese Menge können wir leicht durch Aufzählen beschreiben. Wir wollen diese Menge durch Nennung der Zeichenketten getrennt durch den Oder-Operator `+` beschreiben:

```
a+aa+aaa+aaaa+aaaaa
```

Der Operator ist dabei ein Metasymbol, ein Element unserer Beschreibungssprache. Ein `a` steht für ein a. Wenn die Menge etwas größer ausfällt — zum Beispiel dadurch, daß sie alle Zeichenketten, die ausschließlich aus a's bestehen, gebildet werden soll —, entsteht der Wunsch nach einer abkürzenden Schreibweise. Für

$$\{a, aa, aaa, aaaa, \ldots\}$$

wollen wir schreiben:

```
aa*
```

Das Metazeichen `*` hat die Bedeutung, daß die direkt voranstehende Einheit kein, einmal, zweimal oder noch häufiger vorkommen darf.

Ein zweites Beispiel sei die Menge aller Zeichenketten, die mit beliebig vielen a's beginnen und mit einem b abschließen oder aber aus der Zeichenkette ac bestehen. Dieses läßt sich nun leicht formulieren mittels:

```
a*b+ac
```

In unserer Beschreibungssprache besitzt also das `+` die geringste Priorität.

Ergänzen wir zu der letzten Menge noch die Forderung, daß alle Zeichenketten mit dem Zeichen d enden. Diese neue Menge läßt sich mit Hilfe von Klammern — als weitere Zeichen der Beschreibungssprache — beschreiben durch:

```
(a*b+ac)d
```

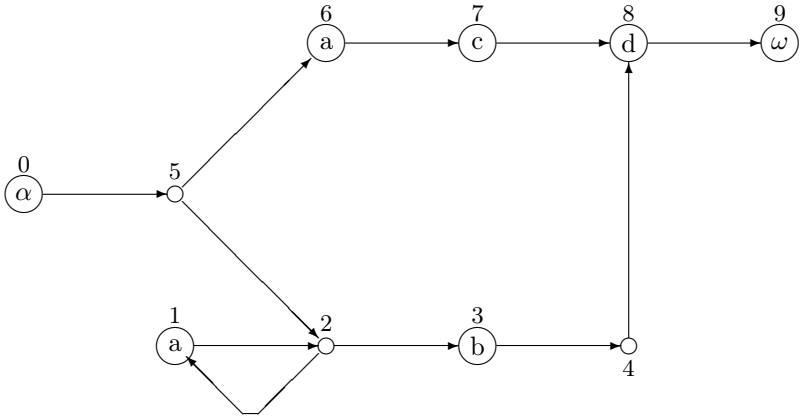
11.3 Wie lassen sich reguläre Ausdrücke finden?

Nehmen wir an, daß ein Text Zeichen für Zeichen eingelesen wird. Dann kann nach jedem Zeichen ein Vertreter aus der benannten Menge entdeckt sein. Wenn dieses der Fall ist, soll eine bestimmte Aktion erfolgen, zum Beispiel eine Ausgabemeldung über den Fund.

Bei dieser Aufgabe kann uns ein Mustererkennungsautomat helfen. Dieser Automat befindet sich in einem bestimmten Zustand. Mit jedem eingelesenen Zeichen ändert sich ggf. sein Zustand. Wenn eine der fraglichen Zeichenketten aufgespürt ist, befindet sich die Maschine in einem ganz bestimmten Zustand und kann eine Meldung absetzen. Es ergeben sich zwei Fragekomplexe:

- Wie läßt sich die Maschine beschreiben, die die gestellte Aufgabe leistet?
- Wie läßt sich eine solche Maschine bauen?

Eine Beschreibung mittels Graphen Folgender Graph beschreibt die Zustände (Knoten) und Übergänge (Kanten) unserer Maschine. Diese zeichnet sich im besonderen dadurch aus, daß sie in verschiedenen Zuständen gleichzeitig sein kann.



11.4 Wie erhalten wir den Graphen zu einem regulären Ausdruck?

Bauprinzipien der Maschine:

- Und-Verbindungen werden durch Reihenschaltung umgesetzt.
- Oder-Verbindungen werden durch Parallelschaltung umgesetzt. Dazu werden ein Knoten als Verzweigungsstelle und einer zur Vereinigung eingefügt, die aber kein Zeichen verarbeiten und ohne Zeitverlust durchlaufen werden.
- Wiederholungen erfordern eine Schleife mit einem Verzweigungsknoten.

- Wenn an einer Verzweigungsstelle zwei Pfade möglich sind, werden beide simultan weiterverfolgt.
- Wird in einem Zustand kein begehbares Pfadstück zu dem zu bearbeitenden Zeichen gefunden, wird der Zustand entfernt.
- Ist die Menge der möglichen Zustände leer, ist die Prüfung negativ beendet.
- Wird der Zustand ω erreicht, ist das letzte gefundene Zeichen das letzte Zeichen einer gefundenen Übereinstimmung im Text.

11.5 Beschreibung des Automaten mittels Datenstrukturen

Folgende Vektoren beschreiben unseren Beispielautomaten:

state	0	1	2	3	4	5	6	7	8	9
character[state]	α	a		b			a	c	d	ω
next1[state]	5	2	3	4	8	6	7	8	9	0
next2[state]	5	2	1	4	8	2	7	8	9	0

11.6 Literatur

Das Beispiel ist entnommen aus Sedgewick,R. (1992): Algorithmen, Addison-Wesley.

11.7 Simulation des Automaten

Mit diesen Strukturen ist das folgende Programm in der Lage, die Musterüberprüfung durchzuführen.

Dafür wird ein Datentyp `deque` benötigt, Beispiel: `aaabd` gesucht in `(a*b+ac)d`.

95 ⟨* 1⟩+ ≡

```
match.auto.sim<-function(a    =c("a","a","a","b","d"),j=1,
                         ch   =c(" ","a"," ","b"," "," ","a","c","d","omega"),
                         next1=c( 5,  2,  3,  4,  8,  6,  7,  8,  9,  0 ),
                         next2=c( 5,  2,  1,  4,  8,  2,  7,  8,  9,  0 )){
  ⟨definiere Funktion melde 96⟩
  SCAN<- -1;  N<-length(a<-c(a,"omega")); start<-1; IO<- 1 # Index-Anfang
  ⟨initialisiere deque: dequeinit 97⟩
  ⟨lege SCAN unten auf deque ab: put(SCAN) 98⟩
  state<-next1[0+IO];  match<-j-1
  repeat{
    melde("repeat:")
    if(state==SCAN){
      j<-j+1
      ⟨lege SCAN unten auf deque ab: put(SCAN) 98⟩
      melde("  naechstes Zeichen:",a[j])
    }else{
      if(ch[state+IO]==a[j]){
        ⟨lege next1[state+IO] unten auf deque ab: put(SCAN) 99⟩
        melde("  Zeichen gefunden:")
```

```
      }else{
        if(ch[state+IO]==" "){
          n1<-next1[state+IO]; n2<-next2[state+IO]
          ⟨lege n1 oben auf deque ab: push(n1) 100⟩
          melde("  neuer Zustand 1:")
          if(n1!=n2){
            ⟨lege n2 oben auf deque ab: push(n2) 101⟩
            melde("  neuer Zustand 2:")
          }
        }
      }
    }
    ⟨bewege oberstes Element der deque nach state: state:=pop 103⟩
    melde("  naechster Zustand:",state)
  # if( j>N || state==0 || #<ist [[deque]] leer>> ) break
    if( j>=N || state==0 )                          break
    if(⟨ist deque leer 102⟩){
      start<-start+1; j<-start
      ⟨lege SCAN unten auf deque ab: put(SCAN) 98⟩
      state<-next1[0+IO];  match<-j-1
    }
  }
  cat("j",j,"N",N,"state",state,"match",match)
  if(state==9)return(paste("found:",match))else return("nomatch")
}
match.auto.sim(c("a","a","a","b","d"))
```

96 ⟨*definiere Funktion melde* 96⟩ ≡ ⊂ 95

```
melde<-function(text="",wert=""){
  wert<-paste(as.character(wert),collapse=" ")
  wert<-paste(wert,substring("                      ",1,10-nchar(wert)))
  text<-paste(text,substring("                      ",1,20-nchar(text)))
  DEQUE<-sub("-1","<>",paste(DEQUE,collapse=" "))
  anz<-15-which(substring(DEQUE,1:10,1:10)=="<")
  if(length(anz)==0||is.na(anz))anz<-17
  DEQUE<-c(paste(rep(" ",anz),collapse=""),DEQUE)
  cat(text,wert,DEQUE)
  cat("\n")
}
```

97 ⟨*initialisiere* deque*:* dequeinit 97⟩ ≡ ⊂ 95

```
DEQUE<-NULL
```

⟨*hello* NA⟩

98 ⟨*lege* SCAN *unten auf* deque *ab:* put(SCAN) 98⟩ ≡ ⊂ 95

```
DEQUE<-c(DEQUE,SCAN)
```

99 ⟨*lege* next1[state+IO] *unten auf* deque *ab:* put(SCAN) 99⟩ ≡ ⊂ 95

```
DEQUE<-c(DEQUE,next1[state+IO])
```

100 ⟨*lege* `n1` *oben auf* `deque` *ab:* `push(n1)` 100⟩ ≡ ⊂ 95
```
DEQUE<-c(n1,DEQUE)
```

101 ⟨*lege* `n2` *oben auf* `deque` *ab:* `push(n2)` 101⟩ ≡ ⊂ 95
```
DEQUE<-c(n2,DEQUE)
```

102 ⟨*ist* `deque` *leer* 102⟩ ≡ ⊂ 95
```
0==length(DEQUE)
```

103 ⟨*bewege oberstes Element der* `deque` *nach* `state`*:* `state:=pop` 103⟩ ≡ ⊂ 95
```
state<-DEQUE[1]; DEQUE<-DEQUE[-1]
```

11.8 Ein Beispiel-Output

```
repeat:                                    <>
  neuer Zustand 1:                       6 <>
  neuer Zustand 2:                     2 6 <>
  naechster Zustand:  2                  6 <>
repeat:                                  6 <>
  neuer Zustand 1:                     3 6 <>
  neuer Zustand 2:                   1 3 6 <>
  naechster Zustand:  1                3 6 <>
repeat:                                3 6 <>
  Zeichen gefunden:                    3 6 <> 2
  naechster Zustand:  3                  6 <> 2
repeat:                                  6 <> 2
  naechster Zustand:  6                    <> 2
repeat:                                    <> 2
  Zeichen gefunden:                        <> 2 7
  naechster Zustand:  -1                      2 7
repeat:                                       2 7
  naechstes Zeichen:  a                  2 7 <>
  naechster Zustand:  2                    7 <>
repeat:                                    7 <>
  neuer Zustand 1:                       3 7 <>
  neuer Zustand 2:                     1 3 7 <>
  naechster Zustand:  1                  3 7 <>
repeat:                                  3 7 <>
  Zeichen gefunden:                      3 7 <> 2
  naechster Zustand:  3                    7 <> 2
repeat:                                    7 <> 2
  naechster Zustand:  7                      <> 2
repeat:                                      <> 2
  naechster Zustand:  -1                        2
repeat:                                         2
  naechstes Zeichen:  a                    2 <>
  naechster Zustand:  2                      <>
repeat:                                      <>
  neuer Zustand 1:                         3 <>
  neuer Zustand 2:                       1 3 <>
  naechster Zustand:  1                    3 <>
```

```
repeat:                                  3 <>
  Zeichen gefunden:                      3 <> 2
  naechster Zustand:  3                    <> 2
repeat:                                    <> 2
  naechster Zustand:  -1                      2
repeat:                                       2
  naechstes Zeichen:  b                    2 <>
  naechster Zustand:  2                      <>
repeat:                                      <>
  neuer Zustand 1:                         3 <>
  neuer Zustand 2:                       1 3 <>
  naechster Zustand:  1                    3 <>
repeat:                                    3 <>
  naechster Zustand:  3                      <>
repeat:                                      <>
  Zeichen gefunden:                          <> 4
  naechster Zustand:  -1                        4
repeat:                                         4
  naechstes Zeichen:  d                    4 <>
  naechster Zustand:  4                      <>
repeat:                                      <>
  neuer Zustand 1:                         8 <>
  naechster Zustand:  8                      <>
repeat:                                      <>
  Zeichen gefunden:                          <> 9
  naechster Zustand:  -1                        9
repeat:                                         9
  naechstes Zeichen:  omega                9 <>
  naechster Zustand:  9                      <>
repeat:                                      <>
  neuer Zustand 1:                         0 <>
  naechster Zustand:  0                      <>
[1] "found: 0"
```

12 Nullstellensuchverfahren

Aus dem Mathematik-Unterricht ist die Suche nach Nullstellen wohlbekannt, beispielsweise im Zusammenhang mit Kurvendiskussionen. Nullstellenprobleme treten auf bei der Suche nach Tabellenwerten, nach Quantilen oder bei Optimierungsproblemen nach vorhergehender Ableitung.

Wesentliche Verfahren sind:

- Bisektion
- Newton-Verfahren
- Sekanten-Verfahren
- Regula Falsi

12.1 Bisektion

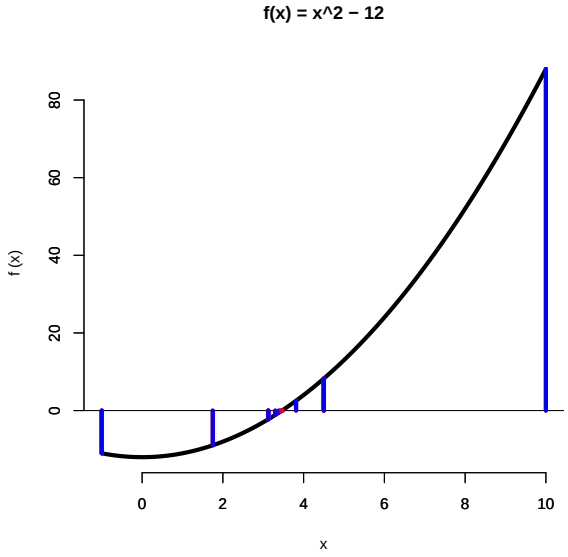
Gegeben sei eine monotone, stetige Funktion f, bei der für zwei Startwerte a und b die Vorzeichen ihrer Funktionswerte ($f(a)$ und $f(b)$) unterschiedlich sind. Folglich muss zwischen a und b eine Nullstelle liegen. Das Prinzip dürfte stark an die binäre Suche wie auch an die Verwendung des Binärbaumes erinnern.

Algorithmus Bisektion

1. berechne $h \leftarrow (a+b)/2$
2. if $0 < f(h) \cdot f(a)$ then $a := h$ else $b := h$
3. if $b - a < \epsilon$ then stop else goto 1.

104 ⟨*definiere* `bisection()` 104⟩ ≡

```
bisection <- function(ff,a,b,eps=0.01,i.max=10){
  ⟨initialisiere bisection 105⟩
  repeat{
    ⟨protokolliere Iterationsbeginn: bisection 106⟩
    h <- (a+b)/2
    if(0 < f(a)*f(h)) a <- h else b <- h
    i <- i+1
    if(i>i.max) break
    if(abs(b-a)<eps) break
    ⟨protokolliere Iterationsende: bisection 107⟩
  }
  ⟨konstruiere Ausgabe: bisection 108⟩
}
bisection(x^2-12,-1,10)
```




```
iteration: 1, interval: (a=-1,b=10), f(a)=-11, f(b)=88
iteration: 2, interval: (a=-1,b=4.5), f(a)=-11, f(b)=8.25
iteration: 3, interval: (a=1.75,b=4.5), f(a)=-8.9375, f(b)=8.25
iteration: 4, interval: (a=3.125,b=4.5), f(a)=-2.23438, f(b)=8.25
iteration: 5, interval: (a=3.125,b=3.8125), f(a)=-2.23438, f(b)=2.53516
iteration: 6, interval: (a=3.125,b=3.46875), f(a)=-2.23438, f(b)=0.03223
iteration: 7, interval: (a=3.29688,b=3.46875), f(a)=-1.13062, f(b)=0.03223
iteration: 8, interval: (a=3.38281,b=3.46875), f(a)=-0.55658, f(b)=0.03223
iteration: 9, interval: (a=3.42578,b=3.46875), f(a)=-0.26402, f(b)=0.03223
```

```
iteration: 10, interval: (a=3.44727,b=3.46875), f(a)=-0.11636, f(b)=0.03223
3.463379 -0.005006552
```

A. 12.1: Wie entwickt sich die Intervallbreite mit wachsender Iterationszahl?

A. 12.2: Studiere: http://de.wikipedia.org/wiki/Bisektion.

105 ⟨*initialisiere bisection* 105⟩ ≡ ⊂ 104

```
if(b<a) { h <- a; a <- b; b <- h }
if(a==b) return("Error: a and b not different")
body <- deparse(substitute(ff))
f <- eval(parse(text=paste("function(x)",body)))
curve(f,a,b, main=paste("f(x) =",body),bty="n",lwd=4)
axis(1); axis(2); abline(h=0)
i <- 1
```

106 ⟨*protokolliere Iterationsbeginn: bisection* 106⟩ ≡ ⊂ 104

```
cat(paste("iteration: ",i,", interval: (a=",round(a,5),",b=",round(b,5),
          "), f(a)=",round(f(a),5),", f(b)=",round(f(b),5),sep=""))
segments(c(a,b),c(0,0),c(a,b),f(c(a,b)),col="red",lwd=4)
ab.old <- c(a,b)
```

107 ⟨*protokolliere Iterationsende: bisection* 107⟩ ≡ ⊂ 104

```
Sys.sleep(0.3)
segments(ab.old,c(0,0),ab.old,f(ab.old),col="blue",lwd=4)
```

108 ⟨*konstruiere Ausgabe: bisection* 108⟩ ≡ ⊂ 104

```
cat(c((a+b)/2,f((a+b)/2)))
```

12.2 Newton-Verfahren

Gegeben sei eine monotone, stetig differenzierbare Funktion f.

Ausgehend von einem Startwert a finden wir iterativ den nächsten Wert durch die Formel:

$$a \leftarrow a - \frac{f(a)}{f'(a)}$$

Begründung: Lege in $(a, f(a))$ eine Tangente an die Kurve und nehme als neuen Wert die Stelle, an der die Tangente die x-Achse schneidet. Also: $a \leftarrow a - x$ und für die Strecke x setze:

$$\frac{f(a)}{x} = \frac{dy}{dx} = f'(a) \quad \Rightarrow \quad x = \frac{f(a)}{f'(a)}$$

Falls die Funktionsvorschrift für die Ableitung zur Verfügung steht, können wir das Newton-Verfahren schnell umsetzen.

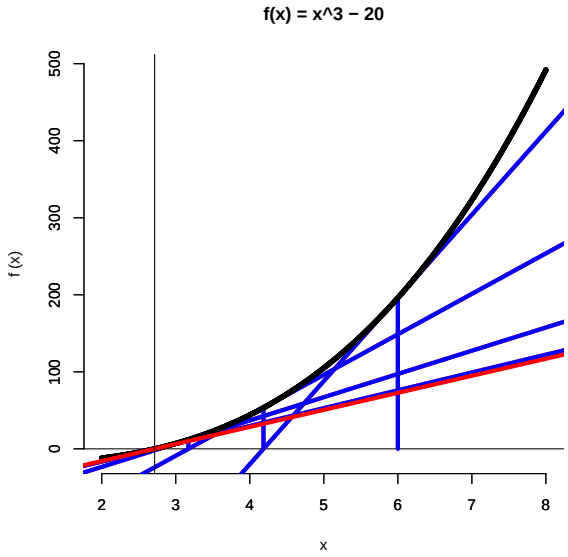
Algorithmus Newton

1. Berechne an der Stelle a den Funktionwert $f(a)$ und den Ableitungswert $f'(a)$.
2. Ermittle neue Stelle: $a \leftarrow a - \frac{f(a)}{f'(a)}$.
3. Wiederhole die ersten beiden Schritte, bis die Genauigkeit ausreicht.

109 ⟨*definiere* `newton()` 109⟩ ≡

```
newton <- function(f,df,a,x.min,x.max,i.max=10,y.eps,sleep=0.5){
  ⟨initialisiere newton 110⟩
  repeat{
    # Ableitung als Attributwert bestimmen
    f.prime <- df(a)
    ⟨protokolliere Iterationmitte: newton 111⟩
    # Newton-Iterationsschritt: neue Stelle der x-Achse ermitteln
    a <- a - f(a)/f.prime
    # Abbruch bei sehr kleinen Funktionswerten
    if(abs(f(a)) < y.eps) break
    # Abbruch bei zu vielen Iterationen
    i <- i+1; if(i>i.max) break
    ⟨protokolliere Iterationsende: newton 112⟩
  }
  ⟨konstruiere Ausgabe: newton 113⟩
}
f <- function(x)  x^3-20; df <- function(x) 3*x^2
#f <- function(x)-x^3+20; df <- function(x)-3*x^2
#newton(f,a=6,x.min=2,x.max=8,i.max=10,y.eps=0.01)
newton(x^3-20,a=6,x.min=2,x.max=8,i.max=10,y.eps=0.01)
```

Die meisten Teile der Funktion sind für die Verpackung relevant, der Kern ist letztlich in einer einzigen Zeile zu finden.



A. 12.3: Was hat das Newton-Verfahren mit der Taylorschen Reihenentwicklung zu tun?

A. 12.4: Studiere: http://de.wikipedia.org/wiki/Newton-Verfahren.

A. 12.5: Überlege, wann das Newton-Verfahren Probleme bekommt.

110 ⟨*initialisiere newton* 110⟩ ≡ ⊂ 109

```
# f:     Funktion als R-Funktion, deren Nullstelle gesucht ist
# df:    Ableitung als R-Funktion
# a:     Startwert
# x.min: Darstellungsuntergrenze
# x.max: Darstellungsobergrenze
# i.max: maximale Iterationszahl
# y.eps: Abbruch, falls Funktionswert kleiner als y.eps
# sleep: Wartezeit
# pwolf 2011/01
⟨checke Funktionsinputs 114⟩
# Kurve zeichnen
if(missing(x.min)) x.min <- a-5; if(missing(x.max)) x.max <- a+5
curve(f,x.min,x.max, bty="n",lwd=5)
axis(1); axis(2); abline(h=0); title(paste("f(x) =",deparse(f)[-1]))
# Initialisierungen
i <- 1; delta.x <- diff(par()$usr[1:2])
if(missing(y.eps)) y.eps <- diff(par()$usr[3:4])*0.001
```

111 ⟨*protokolliere Iterationmitte: newton* 111⟩ ≡ ⊂ 109

```
# Tangente in rot zeichnen und Werte merken
segments(a,0,a,f(a),col="red",lwd=4)
segments(a-delta.x,f(a)-f.prime*delta.x,
         a+delta.x,f(a)+f.prime*delta.x,col="red",lwd=4)
a.old <- a; f.prime.old <- f.prime
```

112 ⟨*protokolliere Iterationsende: newton* 112⟩ ≡ ⊂ 109

```
# Warten und rote Linien blau zeichnen
Sys.sleep(sleep)
segments(a.old,0,a.old,f(a.old),col="blue",lwd=4)
segments(a.old-delta.x,f(a.old)-f.prime.old*delta.x,
         a.old+delta.x,f(a.old)+f.prime.old*delta.x,col="blue",lwd=4)
curve(f,x.min,x.max, bty="n",lwd=5,add=TRUE)
```

113 ⟨*konstruiere Ausgabe: newton* 113⟩ ≡ ⊂ 109

```
abline(v=a);
return(paste("Iterationen:",i,
             ", Stelle:",signif(a,5),", Wert:",signif(f(a)),sep=""))
```

Exkurs: R-Tricks zur Ableitungsberechnung. In manchen Situationen liegt keine Ableitung der Problem-Funktion vor. Außerdem kann es angenehmer sein, die Funktionsvorschriften nicht als R-Funktion übermitteln zu müssen. Deshalb sind für den Aufruf von `newton()` einige Freiheitsgrade geschaffen worden. Falls keine Ableitung angegeben, wird diese mit der R-Funktion `deriv()`

ermittelt. Weiter lässt sich die Problemfunktion als R-Expression angeben. Bei der Implementation zur Eröffnung dieser Möglichkeiten kommen einige R-Tricks zur Anwendung, die hier nicht näher beleuchtet werden. Solche programmiertechnischen Spezialitäten sind zwar kein zentraler Bestandteil der Vorlesung A+D, jedoch aus konzeptioneller Sicht durchaus interessant.

114 ⟨*checke Funktionsinputs* 114⟩ ≡ ⊂ 110

```
text.to.function <- function(text){
  deparse.expression <- sub(".*~","",text)
  fun <- eval(parse(text=paste("function(x)",deparse.expression)))
}
text.to.deriv.function <-  function(text){
  deparse.expression <- paste("~",sub(".*~","",text))
  expression <- eval(parse(text=deparse.expression))
  h <- deparse(deriv(expression ,"x"))
  h <- sub(".*<-","",h[grep(".grad\\[",h)])
  fun <- eval(parse(text=paste("function(x)",h)))
}
if(!is.function(f)) f <- text.to.function(deparse(substitute(f)))
if(missing(df)) df <- text.to.deriv.function(deparse(f)[-1])
```

Alternativ lässt sich aus einem Input-Text schnell eine Funktion bauen:

115 ⟨*unused* 115⟩ ≡

```
body <- deparse(substitute(ff)); f <- eval(parse(text=paste("function(x)",body)))
```

Eine direkte Nutzung von `deriv()` ist etwas verwirrend, da diese Funktion eine Expression, z.B. `~ x^2 -5` erwartet und eine Expression zurückliefert. Für die Auswertung der Ableitung an einer speziellen Stelle ist dann die Expression geeignet zu versorgen.

116 ⟨* 1⟩+ ≡

```
body <- "x^2-5"
# ggf. Konstruktion eines geeigneten Ausdrucks
ff<- eval(parse(text=paste("~",body)))
# Ableitung als Expression
dff <- deriv(ff,"x")
# Auswertung der Ableitung an der Stelle x=a
x <- 3; f.prime <- attr(eval(dff),"gradient")
cat("Funktionsvorschrift"); print(body)
cat("Expression"); print(ff)
cat("Ableitungs-Expression / Auswertung in",x); print(dff); print(f.prime)
```

```
Funktionsvorschrift
[1] "x^2-5"
Expression
~x^2 - 5
Ableitungs-Expression
expression({
    .value <- x^2 - 5
    .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
    .grad[, "x"] <- 2 * x
    attr(.value, "gradient") <- .grad
    .value
})
Ableitung an der Stelle 3
     x
[1,] 6
```

12.3 Sekanten-Verfahren

Falls keine formale Ableitung zu einer Funktion vorliegt, kann sie durch zwei Punkte approximiert werden. Damit wird ausgehend vom Newton-Verfahren die Tangente zu einer Sekante, und es ergibt sich das Sekanten-Verfahren. Wir finden z.B. bei Wikipedia folgende Iterationsvorschrift:

$$x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \cdot f(x_n) = x_n - \frac{f(x_n)}{(f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})}$$

Algorithmus Sekante

1. Berechne zu den beiden aktuellen Stützstellen deren Funktionswerte.
2. Suche die Stelle, an der die Gerade durch die beiden Funktionswerte die x-Achse schneidet:
$$x_{n+1} = x_n - \frac{\Delta x \cdot f(x_n)}{\Delta y}.$$
3. Tausche die ältere Stützstelle gegen x_{n+1} aus.
4. Wiederhole die ersten drei Schritte, bis die Genauigkeit ausreicht.

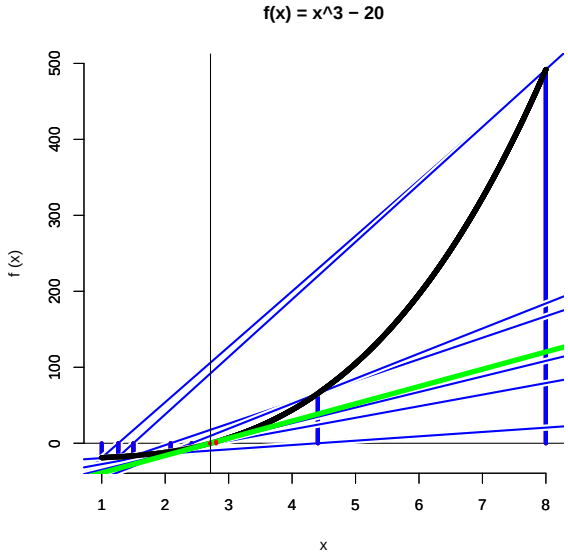
A. 12.6: Demonstriere das Sekantenverfahren mit Papier und Bleistift und erkläre dann, wie die unten vorgestellte R-Funktion arbeitet.

A. 12.7: Studiere: http://de.wikipedia.org/wiki/Sekantenverfahren.

117 ⟨*definiere* `sekante()` 117⟩ ≡

```
sekante <- function(f,a,b,x.min,x.max,i.max=10,y.eps,sleep=0.5){
  ⟨initialisiere sekante 118⟩
  repeat{
    # Ableitungsapproximation durch Sekante
    steigung <- (f(x.n) - f(x.n.minus.1))/(x.n - x.n.minus.1)
    ⟨protokolliere Iterationsmitte: sekante 119⟩
    # Iterationsschritt: neue Stelle der x-Achse ermitteln
    x.n.plus.1 <- x.n - f(x.n)/steigung
    x.n.minus.2 <- x.n.minus.1; x.n.minus.1 <- x.n; x.n <- x.n.plus.1
    # Abbruch bei sehr kleinen Funktionswerten
    if(abs(f(x.n)) < y.eps) break
    # Abbruch bei zu vielen Iterationen
    i <- i+1; if(i>i.max) break
    ⟨protokolliere Iterationsende: sekante 120⟩
  }
  ⟨konstruiere Ausgabe: sekante 121⟩
}
f <- function(x)  x^3-20
#f <- function(x)-x^3+20
sekante(f,a=1,b=8,i.max=10,y.eps=0.01)
```

```
[1] "Iterationen:8, Stelle:2.7141, Wert:-0.00713267"
```



A. 12.8: Vergleiche das Sekanten mit dem Newton-Verfahren. Was ist wann vorzuziehen?

118 ⟨*initialisiere sekante* 118⟩ ≡ ⊂ 117

```
# f:     Funktion als R-Funktion, deren Nullstelle gesucht ist
# a:     erster Startwert
# b:     zweiter Startwert
# x.min: Darstellungsuntergrenze
# x.max: Darstellungsobergrenze
# i.max: maximale Iterationszahl
# y.eps: Abbruch, falls Funktionswert kleiner als y.eps
# sleep: Wartezeit
# pwolf 2011/01
# Kurve zeichnen
if(missing(x.min)) x.min <- a; if(missing(x.max)) x.max <- b
if(x.min > x.max) { h <- x.min; x.min <- x.max; x.max <- h }
curve(f,x.min,x.max, bty="n",lwd=5)
axis(1); axis(2); abline(h=0); title(paste("f(x) =",deparse(f)[-1]))
# Initialisierungen
i <- 1; delta.x <- diff(par()$usr[1:2])
if(missing(y.eps)) y.eps <- diff(par()$usr[3:4])*0.001
x.n <- b; x.n.minus.1 <- a
```

119 ⟨*protokolliere Iterationsmitte: sekante* 119⟩ ≡ ⊂ 117

```
# Tangente in green zeichnen und Werte merken
abline(c(f(x.n)-steigung*x.n,steigung),col="green",lwd=5)
segments(x.n,0,x.n,f(x.n),col="red",lwd=4)
segments(x.n.minus.1,0,x.n.minus.1,f(x.n.minus.1),col="red",lwd=4)
```

120 ⟨*protokolliere Iterationsende: sekante* 120⟩ ≡ ⊂ 117

```
# Warten und green Linien blau zeichnen
Sys.sleep(0.3)
segments(x.n.minus.1,0,x.n.minus.1,f(x.n.minus.1),col="blue",lwd=4)
segments(x.n.minus.2,0,x.n.minus.2,f(x.n.minus.2),col="blue",lwd=4)
abline(c(f(x.n.minus.1)-steigung*x.n.minus.1,steigung),col="white",lwd=6)
abline(c(f(x.n.minus.1)-steigung*x.n.minus.1,steigung),col="blue",lwd=2)
curve(f,x.min,x.max, bty="n",lwd=5,add=TRUE)
```

121 ⟨*konstruiere Ausgabe: sekante* 121⟩ ≡ ⊂ 117

```
abline(v=x.n);
return(paste("Iterationen:",i,
             ", Stelle:",signif(x.n,5),", Wert:",signif(f(x.n)),sep=""))
```

12.4 Regula Falsi

Regula Falsi lässt sich als Kombination von Sekantenverfahren und Bisektion ansehen. Gegeben sei wieder eine monotone, stetige Funktion f. Für zwei Startwerte a und b sind die Vorzeichen ihrer Funktionswerte ($f(a)$ und $f(b)$) unterschiedlich, so dass zwischen a und b eine Nullstelle liegen muss.

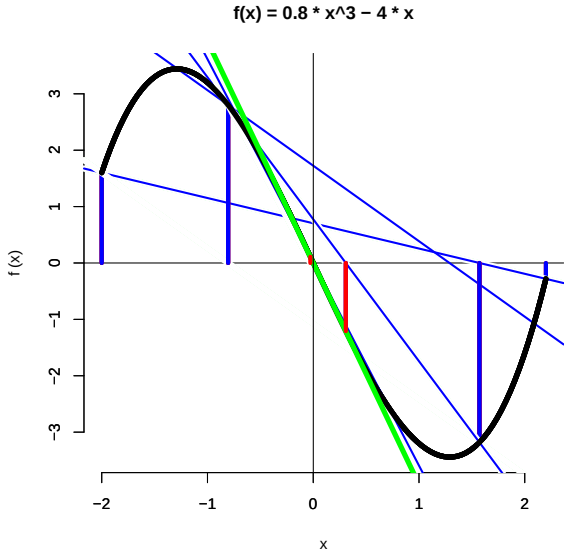
Algorithmus Regula Falsi

1. Verbinde die Funktionswerte $f(a)$ und $f(b)$ durch eine Linie .
2. Suche den Schnittpunkt c dieser Linie mit der x-Achse.
3. if $0 < f(c) \cdot f(a)$ then $a := c$ else $b := c$.
4. if $b - a < \epsilon$ then stop else goto 1.

122 ⟨*definiere* `regula.falsi()` 122⟩ ≡

```
regula.falsi <- function(f,a,b,x.min,x.max,i.max=10,y.eps,sleep=0.5){
  ⟨initialisiere regula falsi 123⟩
  repeat{
    # Ableitungsapproximation durch Sekante
    steigung <- (f(a) - f(b))/(a - b)
    ⟨protokolliere Iterationsmitte: regula.falsi 124⟩
    # Iterationsschritt: neue Stelle der x-Achse ermitteln
    neu <- a - f(a)/steigung; alt <- c(a,b)
    if( 0 < f(neu)*f(a) ) a <- neu else b <- neu
    # Abbruch bei sehr kleinen Funktionswerten
    if(abs(f(neu)) < y.eps) break
    # Abbruch bei zu vielen Iterationen
    i <- i+1; if(i>i.max) break
    ⟨protokolliere Iterationsende: regula.falsi 125⟩
  }
  ⟨konstruiere Ausgabe: regula.falsi 126⟩
}
f <- function(x)  x^3-20; f <- function(x) 0.8*x^3-4*x
regula.falsi(f,a=-2,b=2.2,i.max=10,y.eps=0.01)
```

```
[1] "Iterationen:5, Stelle:0.00048118, Wert:-0.00192471"
```



A. 12.9: Was versteht man unter dem Illinois-Verfahren?

A. 12.10: Studiere: http://de.wikipedia.org/wiki/Regula_Falsi.

A. 12.11: Überlege, wann die einfachen Nullstellenmethoden gut funktionieren und wann Probleme auftreten.

A. 12.12: Untersuche und recherchiere die Konvergenzeigenschaften anhand von Beispielen.

A. 12.13: Versuche $P(X \leq 1) =?$ für eine standardnormalverteilte Zufallsvariable mit Hilfe der vorgestellten Algorithmen und $F(x) =$ `pnorm(x)` zu ermitteln. Beurteile anschließend die Verfahren.

123 ⟨*initialisiere regula falsi* 123⟩ ≡ ⊂ 122

```
# f:     Funktion als R-Funktion, deren Nullstelle gesucht ist
# a:     erster Startwert
# b:     zweiter Startwert
# x.min: Darstellungsuntergrenze
# x.max: Darstellungsobergrenze
# i.max: maximale Iterationszahl
# y.eps: Abbruch, falls Funktionswert kleiner als y.eps
# sleep: Wartezeit
# pwolf 2011/01
```

```
# Kurve zeichnen
if(missing(x.min)) x.min <- a; if(missing(x.max)) x.max <- b
if(x.min > x.max) { h <- x.min; x.min <- x.max; x.max <- h }
curve(f,x.min,x.max, bty="n",lwd=5)
axis(1); axis(2); abline(h=0); title(paste("f(x) =",deparse(f)[-1]))
# Initialisierungen
i <- 1; delta.x <- diff(par()$usr[1:2])
if(missing(y.eps)) y.eps <- diff(par()$usr[3:4])*0.001
x.n <- b; x.n.minus.1 <- a
```

124 ⟨*protokolliere Iterationsmitte: regula.falsi* 124⟩ ≡ ⊂ 122

```
# Tangente in green zeichnen und Werte merken
abline(c(f(a)-steigung*a,steigung),col="green",lwd=5)
segments(a,0,a,f(a),col="red",lwd=4)
segments(b,0,b,f(b),col="red",lwd=4)
```

125 ⟨*protokolliere Iterationsende: regula.falsi* 125⟩ ≡ ⊂ 122

```
# Warten und green Linien blau zeichnen
Sys.sleep(sleep)
segments(alt,0,alt,f(alt),col="blue",lwd=4)
abline(c(f(alt[1])-steigung*alt[1],steigung),col="white",lwd=6)
abline(c(f(a)-steigung*a,steigung),col="blue",lwd=2)
curve(f,x.min,x.max, bty="n",lwd=5,add=TRUE)
```

126 ⟨*konstruiere Ausgabe: regula.falsi* 126⟩ ≡ ⊂ 122

```
abline(v=neu);
return(paste("Iterationen:",i,
             ", Stelle:",signif(neu,5),", Wert:",signif(f(neu)),sep=""))
```

13 Nummerische Integration

Bei vielen Problemen in den Wissenschaften, insbesondere in der Statistik, sind Integrale zu lösen.

- $P(X \leq 1) = F(1) = \int_{-\infty}^{1} f(x)dx$
- $E(g(X)) = \int_{-\infty}^{\infty} g(x)f(x)dx$
- Algorithmen auf Basis der Bayesschen Formel

Ausgehend von diese Problemen lassen sich folgende Punkte überlegen:

- grundsätzliche Ansätze zur Integration
- Polstellen: vgl. bspw. $P(X \leq 1)$ für eine χ_1^2 verteilte Zufallsvariable
- $\infty, -\infty$ als Integrationsgrenzen: vgl. bspw. $P(X \leq 1)$ für eine standardnormalverteilte Zufallsvariable.

In diesem Abschnitt werden immer wieder Bezeichnungen verwendet. Dabei sollen folgende Bedeutungen gelten:

Symbol	Bedeutung
$a = x_0 < x_1 < \cdots < x_n = b$	Stützstellen
n	oft Intervallanzahl
$(b-a)/n = h$	oft Breite von Flächenstreifen
$I, I_{(a,b)}, I_T$	u.a. bezeichnet Integrale
$f(x)$	zu integrierende Funktion

A. 13.1: Algorithmen zur Integration sind oft mit Begriffen wie: Regeln, rules, Cotes verbunden. Spüren diesen Begriffen nach!

13.1 Riemann-Summe

Approximation des Integrals durch Rechtecke.

Algorithmus Riemann-Summe

1. Teile Integrationsbereich in Flächenstreifen.
2. Berechne für jeden Flächenstreifen einen Funktionswert.
3. Berechne

$$I_R = \sum_{i_1}^{n} (x_i - x_{i-1}) \cdot f(x_i^*) \quad \text{mit} \quad x_i^* \in [x_{i-1}, x_i]$$

   und liefere diese Integralabschätzung I_R ab.

Fehlerabschätzung: absoluter Fehler$(n) = O(n^{-1})$

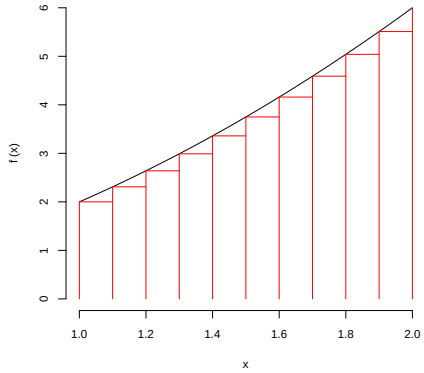
- Rechteckansatz bedeutet eine stückweise Approximation durch Polynom vom Grad 0
- Summation $\rightarrow$ Aggregationsidee
- häufige Struktur: $\sum_i$ Gewicht$_i \cdot f(x_i^*)$
- leicht implementierbar

A. 13.2: Welche Rechteecke würden Sie in welchen Fällen wählen? Beantworten Sie auch die Warum-Frage.

Den Output ...

```
Riemann, u 3.635
Riemann, m 3.8325
Riemann, o 4.035
```

... und das Bild ...



erarbeitet folgende Implementation mit anschließendem Aufruf.

127 ⟨*definiere* `I.Riemann()` 127⟩ ≡

```
I.Riemann <- function(f,a,b,n=10,type="m"){  # pwolf 2011/01
  x <- seq(a,b,length=n+1)
  curve(f,a,b,bty="n",ylim=c(0,max(f(x))))
  segments(x,0,x,f(x),col="red")
  segments(x[-1],f(x)[-n-1],x[-n-1],f(x)[-n-1],col="red")
  h <- (b-a)/n
  if( type=="u" ) x <- x[-n-1]
  if( type=="o" ) x <- x[-1]
  if( type=="m" ) x <- (x[-1]+x[-n-1])/2
  result <- sum( h * f(x))
  return(result)
}
f <- function(x) x^2 + x
a <- 1; b <- 2
cat("korrekt: ", 1/3* b^3 + 1/2* b^2 - (1/3* a^3 + 1/2* a^2))
cat("Riemann, u",I.Riemann(f,a=a,b=b,n=10,type="u"))
cat("Riemann, m",I.Riemann(f,a=a,b=b,n=10,type="m"))
cat("Riemann, o",I.Riemann(f,a=a,b=b,n=10,type="o"))
```

13.2 Trapez-Regel

Polynomgrad 1 führt zu einer stückweise linearen Approximation von $f(x)$.

$$I_T = \sum_{i_1}^{n} (x_i - x_{i-1}) \cdot \frac{f(x_i) + f(x_{i-1})}{2} = \frac{h}{2} \sum_{i=1}^{n} (f(x_i) + f(x_{i-1}))$$

also

$$I_T = \frac{h}{2} \left[f(a) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(b) \right]$$

Approximationsfehler $= O(f''(x^*)/n^2)$.

Algorithmus Trapez-Regel

1. Teile Integrationsbereich in Flächenstreifen.
2. Berechne für jede Stützstelle ihren Funktionswert $f(x)$.
3. Berechne
$$I_T = \frac{h}{2}\left(-f(a) - f(b) + 2\sum_{i=0}^{n} f(x_i))\right)$$
und liefere diese Integralabschätzung I_T ab.

A. 13.3: Erklären Sie einem Unkundigen die obige Formel und fertigen Sie dazu geeignete Zeichnungen an.

128 ⟨*definiere* `I.Trapez()` 128⟩ ≡
```
I.Trapez <- function(f,a,b,n=10,type="m"){
  # pwolf 2011/01
  x <- seq(a,b,length=n+1)
  h <- (b-a)/n
  gewichte <- h/2 * c(1,rep(2,n-1),1)
  result <- sum( gewichte * f(x))
  return(result)
}
f <- function(x) x^2 + x
a <- 1; b <- 2
cat("korrekt: ", 1/3* b^3 + 1/2* b^2 - (1/3* a^3 + 1/2* a^2))
cat("Trapez-Regel",I.Trapez(f,a=a,b=b,n=10))
```

```
korrekt:  3.833333
Trapez-Regel 3.835
```

A. 13.4: Falls Sie nicht mehr wissen, was man unter Trapezen versteht, schauen Sie bspw. in Ihrem alten Mathe-Buch die Bedeutung nach.

A. 13.5: Wie groß muss n gewählt werden, damit das Ergebnis auf 5 Stellen genau ist?

13.3 Simpson-Regel

Durch Übergang zu einer Approximation mittels Polynomen 2-ten Grades erhält man die Simpson-Regel.

$$I_S = \frac{h}{3}\left[f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(b)\right]$$

Hierbei sei unterstellt, dass $n = 2{\cdot}k$ gerade ist. Approximationsfehler $= O(n^{-4})$.

Algorithmus Simpson-Regel

1. Teile Integrationsbereich in Flächenstreifen.
2. Berechne für jede Stützstelle ihren Funktionswert $f(x)$.
3. Berechne

$$I_S = \frac{h}{3}\left(f(a) + f(b) + 4 \cdot \sum_{\substack{i \text{ ungerade} \\ i<n}} f(x_i) + 2 \cdot \sum_{\substack{i \text{ gerade} \\ 0<i<n}} f(x_i) \right)$$

   und liefere diese Integralabschätzung I_S ab.

129 ⟨*definiere* `I.Simpson()` 129⟩ ≡

```
I.Simpson <- function(f,a,b,n=10,type="m"){
  # pwolf 2011/01
  x <- seq(a,b,length=n+1)
  h <- (b-a)/n
  koeffizienten <- c(1,rep(c(4,2),(n-1)/2),4,1)
  print(koeffizienten)
  gewichte <- h/3 * koeffizienten
  result <- sum( gewichte * f(x))
  return(result)
}
f <- function(x) x^2 + x
a <- 1; b <- 2
cat("korrekt: ", format(1/3*b^3+1/2*b^2 - (1/3*a^3+1/2*a^2),digits=15))
cat("Simpson-Regel",format(I.Simpson(f,a=a,b=b,n=10),digits=15))
```

```
korrekt:  3.83333333333333
 [1] 1 4 2 4 2 4 2 4 2 4 1
Simpson-Regel 3.83333333333333
```

An dem Beispiel lässt sich erkennen, dass die Simpson-Regel für Polynome bis zum 3-Grad exakte Ergebnisse hervorbringt. Für die anderen vorgestellten Regeln gilt entsprechendes.

A. 13.6: Überprüfen Sie empirisch die Exaktheitsbemerkung auch für die anderen vorgestellten Verfahren.

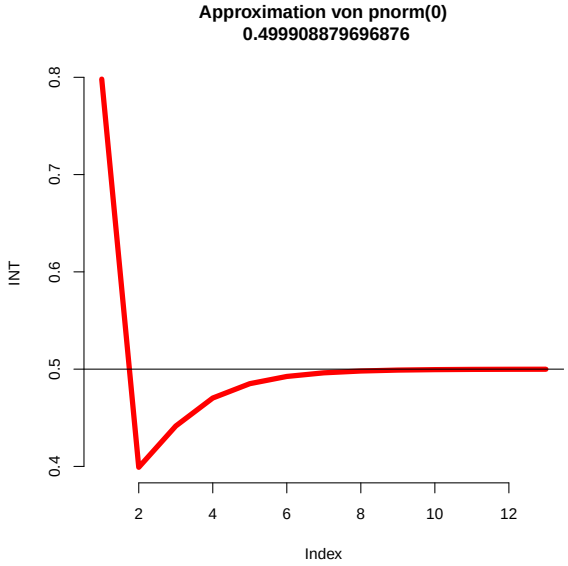
13.4 Schrittweise Genauigkeitsverbesserungen

In der Regel weiß man zunächst nicht, wie häufig man iterieren muss, bis die Genauigkeit genügend groß ist. Es ist naheliegend, solange weiterzurechnen, bis der Fehler ein gewisse geringe Größe erreicht hat. Der vorliegende Fehler lässt sich anhand der Differenzen zwischen Integral-Approximationen mit unterschiedlichem n abschätzen. Natürlich wird man sich bemühen, bereits berechnete Funktionswerte nicht noch einmal zu ermitteln, sondern diese für die nächsten Schätzungen zu verarbeiten. Auf diesen beiden Ideen baut folgender Algorithmus auf, dessen Rohversion samt weiterer Erklärungen bei Thisted (p.269) zu finden ist.

130 ⟨*definiere* `I.trapeziodal()` 130⟩ ≡

```
I.trapeziodal <- function(f,a,b,eps=0.01,K=15){
  # Trapez-Regel nach Thisted, p. 269.
  # pwolf 2011/01
  h <- b-a; n <- 1
  Int <- h * (f(a) + f(b))/2
  for(j in 1:K){
    if(n>1) p <- sum( f(a + h/2 + (1:(n-1))*h) ) else p <- 0
    Int <- c((Int[1] + h*p)/2 , Int)
    h <- h/2; n <- 2*n
    if(abs(diff(Int[1:2]))<eps)
      return(rev(Int)) else cat("j",j,"integral",Int[1])
  }
}
f <- function(x) dnorm(x,0,1)
INT<-I.trapeziodal( f, -4, 0, eps=0.0001)
plot(INT, type="l", lwd=5, col="red", bty="n"); abline(h=pnorm(0))
title(paste("Approximation von pnorm(0)\n",INT[length(INT)]))
```

```
j 1 integral 0.3990761
j 2 integral 0.4415088
j 3 integral 0.47031
j 4 integral 0.4850536
j 5 integral 0.4924843
j 6 integral 0.4962157
j 7 integral 0.4980873
j 8 integral 0.4990256
j 9 integral 0.4994959
j 10 integral 0.4997316
j 11 integral 0.4998497
```



A. 13.7: Lesen Sie den entsprechenden Abschnitt bei Thisted nach.

A. 13.8: Wie viele Funktionswerte sind für die einzelnen Approximation ausgewertet worden?

13.5 Bemerkungen

- Ansätze, bei denen eine gewichtete Summe von Funktionswerten äquidistanter x-Werte verwendet werden, heißen Newton-Cote-Regeln.
- Polstellen-Problem: Verwende offene Newton-Cotes, also Cotes, die die Endpunkte nicht auswerten.
- uneigenliche Integrale: transformiere Integral, bspw. durch Kehrwertbildung oder durch $u(x) = e^{-x}$
- Alternative: passe Polynom an, dann integriere Polynom → Lagrange-Polynome. Siehe: http://de.wikipedia.org/wiki/Polynominterpolation.

A. 13.9: Versuche $P(X \leq 1) =?$ für eine standardnormalverteilte Zufallsvariable mit Hilfe der vorgestellten Algorithmen zur Integration und $f(x) =$ `dnorm(x)` zu ermitteln. Beurteile anschließend die Verfahren.

A. 13.10: Was leistet die R-Funktion `integrate()`?

A. 13.11: Integrationsfragen kann man auch *symbolisch* angehen. Was versteht man darunter? Recherchieren Sie?

14 Gleichungssysteme und andere Berechnungen

Lineare Gleichungssysteme lassen sich nach Gesetzen der linearen Algebra einfach lösen: $\mathbf{Ax} = \mathbf{y} \Rightarrow \mathbf{x} = \mathbf{A^{-1}y}$. Für Regressionsprobleme greifen wir wie selbstverständlich auf die Formel $(\mathbf{X'X})^{-1}\mathbf{X'y}$ zurück. Doch stellt sich die Frage, ob diese Formeln geeignete Wege der Berechnung darstellen. Wir werden im Folgenden einen kurzen Blick auf die Darstellung von Zahlen werfen, sodann überlegen wir, wie man Momente berechnen kann. Anschließend wenden wir uns der Kleinste-Quadrate-Schätzung eines linearen Regressionsmodells zu.

14.1 Genauigkeit der Darstellung von Zahlen

Zahlen werden auf Rechnern in der Regel durch Fließpunktzahlen repräsentiert. Bei diesen wird die Größenordnung durch den Exponenten und die Genauigkeit durch die Länge der Mantisse bestimmt. Im 10-er System mit einer Mantissenlänge von 3 können wir die Zahl π beispielsweise als Ziffernfolge 314 mit Exponenten -2 speichern: $314 \cdot 10^{-2} = 3.14$. Bei Interpretation der Mantisse als Nachkommastellen ergäbe sich als Exponent 1. Solche Alternativen beeinflussen nicht das Problem, dass mit der Speicherung ein Fehler entsteht, hier resultiert

ein absoluter Fehler von $\pi - 3.14 = 0.00159265358979299...$ Der hantierte Wert 3.14 weicht also vom wahren ab:

$$\begin{aligned} 3.14 &= \pi - 0.00159265358979299.. = \pi\left(1 + \frac{-0.00159265358979299..}{\pi}\right) \\ &=: \pi(1+\epsilon) \end{aligned}$$

ϵ zeigt uns den relativen Fehler und besitzt für das Beispiel den Wert $|\epsilon| = |-3.14/\pi| = 3.14/\pi = 0.000506957382897213...$ Werden für die Mantisse t Stellen und ein Zahlensystem mit der Basis β verwendet, dann lässt sich der relative Fehler begrenzend abschätzen: $|\epsilon| \leq \beta/2\beta^{-t} = \beta^{1-t}/2$. Für das Beispiel ergibt sich also eine Obergrenze des relativen Fehlers von $10^{1-3}/2 = 0.01/2 = 0.005$. Bei der Verwendung des Zweiersystems ($\beta = 2$) erhalten wir bei $t = 23$ Stellen eine Grenze von $2^{1-23}/2 = 2^{-23} = \texttt{1.192093e-07}$. Eine solche Konstellation kommt zustande, wenn 32 Bits zur Darstellung einer Zahl bereitgestellt, jedoch 8 Bits für den Exponenten und ein Bit für das Vorzeichen reserviert werden.

131 ⟨* 1⟩+ ≡
```
cat(format(pi-3.14,digits=15,scientific=FALSE))
cat(format((pi-3.14)/pi,digits=15,scientific=FALSE))
cat(10^-3,2^-23)
t <- 1:7; fak <- 10^(t-1)
noquote(cbind(digits=t,"10^-t"=format(10^-t,scientific=FALSE),
              "rel.error"=format((pi - round(pi*fak)/fak)/pi,scientific=FALSE)))
```

Für Operationen können entsprechende Genauigkeitsüberlegungen angestellt werden. Statt einer Addition (+) realisiert sich auf dem Rechner die Operation $+^*$:

$$(x +^* y) = (x + y)(1 + \epsilon)$$

und statt einer Multiplikation (Division) wird $\times^*$ (bzw. $/^*$) umgesetzt. Wenn wir also den Kehrwert einer Zahl x berechnen wollen, erhalten wir statt $f(x) = 1/x$ den Wert

$$f^*(x^*) \approx f(x^*)(1+\epsilon_1) \approx f(x(1+\epsilon_2))(1+\epsilon_1) = \frac{(1+\epsilon_1)}{x(1+\epsilon_2)}$$

Im Rahmen der Fehleranalyse versucht man, die Folgewirkungen von Fehlern einzelner Operationen auf das Gesamtergebnis abzuschätzen.

A. 14.1: Verfolgen Sie, wie sich der Fehler durch wiederholtes Wurzelziehen und Quadrieren von π entwickelt. Wann wird der Fehler spürbar?

14.2 Nummerische Betrachtungen einfacher Rechnungen

Einfache Berechnungen führen auf Rechner aufgrund der begrenzten Zahlendarstellungen in der Regel nicht zu korrekten Ergebnissen. Durch Verwendung von geschickten Algorithmen lassen sich jedoch Fehler reduzieren. Wie würden Sie bspw. einen Mittelwert berechnen? Der folgende Chunk zeigt zwei mögliche Wege zur Mittelwertberechnung:

132 ⟨* 1⟩+ ≡
```
mean_naiv <- function(x){
```

```
  sum <- 0; n <- length(x)
  for(i in seq(along=x))
    sum <- sum + x[i]
  return(sum/n)
}
mean_clever <- function(x){
  sum <- 0; n <- length(x)
  for(i in seq(along=x))
    sum <- sum + x[i]
  mean_approx <- sum/n
  sum <- 0
  for(i in seq(along=x))
    sum <- sum + (x[i] - mean_approx)
  return( mean_approx + sum/n )
}
shift <- 10000000; fak <- 100; set.seed(13)
x <- rnorm(1000); x <- shift + fak * c(x,-x)
out1   <- format(m1 <- mean_naiv(x),digits=15,scientific=FALSE)
out2   <- format(m2 <- mean_clever(x),digits=15,scientific=FALSE)
error1 <- format(m1 - shift,digits=15,scientific=FALSE)
error2 <- format(m2 - shift,digits=15,scientific=FALSE)
cat("\nnaiv:  ", out1, error1 )
cat("\nclever:", out2, error2 )
```

Wir erhalten als Output:

```
naiv:   9999999.99999998 -0.0000000242143869400024
clever: 10000000 0
```

A. 14.2: Begründe, dass der zweite Algorithmus *theorisch* den korrekten Mittelwert hervorbringen sollte und warum er in manchen Situationen vorteilhaft ist.

A. 14.3: Vergleiche die beiden Algorithmen in verschiedenen Datenkonstellationen.

A. 14.4: Überlege und probiere verschiedene Algorithmen zur Berechnung der empirischen Stichprobenvarianz.

A. 14.5: Wieso kann die Berechnung von Momenten als Auswertung von inneren Produkten angesehen werden? Lassen sich die Überlegungen zur Berechnung von Mittelwerten auf die Berechnung von inneren Produkten übertragen?

14.3 Regression

14.3.1 Probleme der Implementation

Die berühmte Zauberformel der Regressionsanalyse lautet:

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

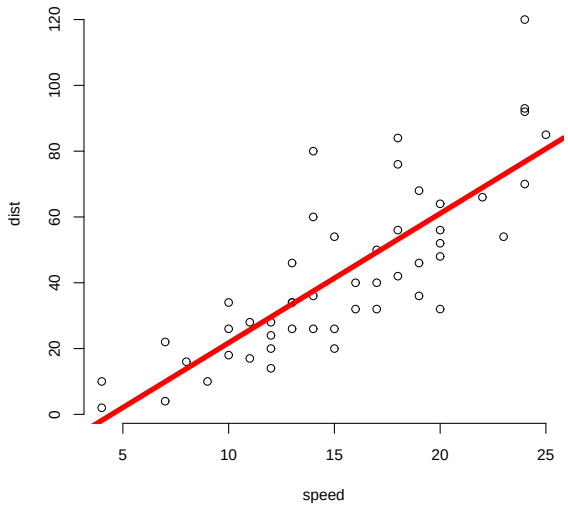
Es ist naheliegend, diese Formel zur Schätzung des Koeffizientenvektors direkt unter Verwendung entsprechender Sprachelemente umzusetzen:

133 ⟨ * 1⟩+ ≡

```
y <- cars$dist; X <- cbind(1,cars$speed)
beta.dach <- solve( t(X) %*% X ) %*% t(X) %*% y
plot(cars,bty="n"); abline(beta.dach, lwd=5, col="red")
cat("Koeffizienten:",beta.dach[1:2])
```

In der Tat erhalten wir für den Datensatz `cars` eine brauchbare Schätzung:

```
Koeffizienten: -17.57909 3.932409
```



Jedoch hat unser Einführungsbeispiel bereits auf Probleme hingewiesen. Der Kern sei zur Erinnerung noch einmal wiederholt:

134 ⟨ * 1⟩+ ≡

```
anzahl.daten <- 10                          #   Daten erzeugen
x <- 1:anzahl.daten; set.seed(13)           #   x-Werte
y <- sin(x/4) + rnorm(anzahl.daten)         #   y-Werte
k <- 8                                      #   max Polynomgrad setzen
X <- outer(x,0:k,"^")                       #   Designmatrix Polynomgrad k
result <- solve(t(X)%*%X)%*%t(X)%*%y        #   Versuch, ein Modell anzupassen
```

Leider erhalten wir eine Fehlermeldung.

```
Error in solve.default(t(X) %*% X) :
  System ist fuer den Rechner singulaer: reziproke Konditionszahl = 4.36775e-22
```

Wir können überlegen, dass sich bei der Berechnung der inneren Produkte nummerische Schwierigkeiten ergeben und könnten diese etwas näher unter die Lupe nehmen. Doch entlarvt die Fehlermeldung die Matrizen-Inversion als unüberwindliche Hürde. Von außen ist nicht zu sehen, welcher Algorithmus hinter der Invertierung verborgen ist. Dennoch wissen wir, dass dabei eine Aufgabe zu lösen ist, bei der viele elementare Operationen erforderlich sind. Beispielsweise erfordert der Algorithmus zur Invertierung mittels der Adjunkten eine Division durch die Determinante der zu invertierenden Matrix:

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\mathrm{adj}(\mathbf{A})$$

A. 14.6: Vergleiche hierzu bspw. http://de.wikipedia.org/wiki/Adjunkte und http://de.wikipedia.org/wiki/Reguläre_Matrix.

Die vielen beteiligten Operationen werden insbesondere bei ungeschickten Datenkonstellationen nummerische Probleme hervorrufen. Deshalb wollen wir über Alternativen der Minimierung von $\mathbf{u}'\mathbf{u}$ zur Schätzung der Koeffizientenvektors nachdenken:

$$\min_{\beta} \mathbf{u}'\mathbf{u} \quad \text{mit} \quad \mathbf{u} = \mathbf{y} - \mathbf{X}\beta$$

A. 14.7: Überprüfe die Orthogonalitätseigenschaften von Größen der Regression an Zahlen-Beispielen mit R.

14.3.2 Transformation des Problems

Verschiedene Ausführungen in diesem und den nächsten Abschnitten gehen stark zurück auf: *R. A. Thisted (1988): Elements of Statistical Computing, Chapman and Hall.*

Sei $\mathbf{X}$ eine $(n \times p)$ Designmatrix, $\mathbf{y}$ ein Vektor der Länge p und $\mathbf{Q}$ eine orthogonale $(n \times n)$-Matrix mit $\mathbf{Q}'\mathbf{Q} = \mathbf{Q}^{-1}\mathbf{Q} = \mathbf{I}$. Dann gilt:

$$\begin{aligned}
\mathbf{u}'\mathbf{u} &= (\mathbf{y} - \mathbf{X}\beta)'(\mathbf{y} - \mathbf{X}\beta) \\
&= (\mathbf{y} - \mathbf{X}\beta)'\mathbf{Q}'\mathbf{Q}(\mathbf{y} - \mathbf{X}\beta) \\
&= (\mathbf{Q}(\mathbf{y} - \mathbf{X}\beta))'\mathbf{Q}(\mathbf{y} - \mathbf{X}\beta) \\
&= (\mathbf{Q}\mathbf{y} - \mathbf{Q}\mathbf{X}\beta)'(\mathbf{Q}\mathbf{y} - \mathbf{Q}\mathbf{X}\beta) \\
&=: (\mathbf{y}^* - \mathbf{X}^*\beta)'(\mathbf{y}^* - \mathbf{X}^*\beta)
\end{aligned}$$

Wir sehen, uns liefert

$$\mathbf{y}^* = \mathbf{X}^*\beta + \epsilon^* \quad \text{mit} \quad \mathbf{y}^* = \mathbf{Q}\mathbf{y},\ \mathbf{X}^* = \mathbf{Q}\mathbf{X},\ \epsilon^* = \mathbf{Q}\epsilon$$

ebenfalls den gesuchten Schätzvektor $\hat{\beta}$. Wir erhalten also mit der transformierten Beziehung einen alternativen Lösungsweg. Dieser Weg ist attraktiv, wenn wir bequem solche orthogonale Matrizen $\mathbf{Q}$ ermittelt können, so dass sich bspw. die Normalgleichungen $\mathbf{X}^{*\prime}\mathbf{X}^*\beta = \mathbf{X}^{*\prime}\mathbf{y}^*$ leichter lösen lassen.

Eine Lösung fällt insbesondere dann leicht, wenn sich die $(n \times p)$-Matrix $\mathbf{X}^*$ aus einer oberen Dreiecksmatrix $(p \times p)$-Matrix $\mathbf{X}_1^*$ und einer Matrix aus Nullen zusammensetzt:

$$\mathbf{X}^* = \begin{pmatrix} \mathbf{X}_1^* \\ \mathbf{X}_2^* \end{pmatrix} = \begin{pmatrix} \mathbf{X}_1^* \\ \mathbf{0}_{(\mathrm{n-p})\times \mathrm{p}} \end{pmatrix}$$

Denn nun können wir die quadrierte Länge von $\mathbf{u}$ umformen zu:

$$\begin{aligned} \mathbf{u'u} &= (\mathbf{y}^* - \mathbf{X}^*\beta)'(\mathbf{y}^* - \mathbf{X}^*\beta) \\ &= \left[\begin{pmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2^* \end{pmatrix} - \begin{pmatrix} \mathbf{X}_1^* \\ \mathbf{0}_{(\mathrm{n-p})\times \mathrm{p}} \end{pmatrix}\beta\right]' \left[\begin{pmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2^* \end{pmatrix} - \begin{pmatrix} \mathbf{X}_1^* \\ \mathbf{0}_{(\mathrm{n-p})\times \mathrm{p}} \end{pmatrix}\beta\right] \\ &= [\mathbf{y}_1^* - \mathbf{X}_1^*\beta]'[\mathbf{y}_1^* - \mathbf{X}_1^*\beta] + |\mathbf{y}_2^*|^2 \end{aligned}$$

Für die Minimierung des inneren Produktes $\mathbf{u'u}$ ist $\mathbf{y}_2^*$ als Konstante anzusehen und muss nicht mehr weiter berücksichtigt werden. Die Lösung kann also auf Basis des Modells

$$\mathbf{y}_1^* = \mathbf{X}_1^*\beta + \epsilon_1^*$$

erarbeitet werden. Da β und $\mathbf{y}_1^*$ Vektoren der Länge p sind und $\mathbf{X}_1^*$ eine $(p \times p)$-Matrix ist, kann in normalen, nicht pathologischen Fällen ϵ_1^* auf Null gesetzt und die Gleichung $\mathbf{y}_1^* = \mathbf{X}_1^*\beta$ nach β aufgelöst werden.

Geometrische Interpretation Bei der unterstellten Transformation werden die Spaltenvektoren von $\mathbf{X}$ im $\mathcal{R}^n$ so rotiert, dass sie den Unterraum der ersten p Einheitsvektoren aufspannen. $\mathbf{y}$ wird entsprechend transformiert, und es gilt, die Linearkombination der transformierten Spaltenvektoren zu finden, mit der der Teilvektor von $\mathbf{y}^*$ bestehend aus den ersten p Koordinaten von $\mathbf{y}^*$ dargestellt werden kann. Dieser Teilvektor muss in dem von den transformierten Spaltenvektoren aufgespannten Raum liegen. Wenn die Spaltenvektoren von $\mathbf{X}$ unabhängig sind, gelingt eine solche Darstellung. Hierbei können wir $\epsilon_1^* = \mathbf{0}$ setzen, und wir haben nur die Koeffizienten der Linearkombination zu ermitteln, also $\mathbf{y}_1^* = \mathbf{X}_1^*\beta_1$ zu lösen.

In dem Fall, dass $\mathbf{X}_1^*$ eine obere Dreiecksmatrix ist, lässt sich dieses Gleichungssystem

$$\mathbf{y}_1^* = \mathbf{X}_1^*\beta$$

sukzessive lösen und der Schätzer für β in einer einfachen Schleife berechnen.

Fazit Zusammenfassend kann man also statt

$$\mathbf{y} = \mathbf{X}\beta + \epsilon$$

ebenso gut

$$\mathbf{y}^* = \mathbf{X}^*\beta + \epsilon^*$$

untersuchen oder auch nur

$$\mathbf{y}_1^* = \mathbf{X}_1^*\beta$$

lösen. Notwendig ist allerdings, vorher eine geeignete Matrix $\mathbf{Q}$ zu konstruieren.

Über die Stichwörter:

- Householder-Transformationen
- Gram-Schmidt-Methoden
- Givens-Rotationen

werden wir auf der Suche nach Konstruktionsverfahren fündig. Zur Demonstration werden wir im Folgenden kurz zum ersten und zum letzten Punkt einige Bemerkungen machen.

A. 14.8: Stöbern Sie nach dem Keywords *Householder-Transformationen* und *Gram-Schmidt* im Internet.

14.3.3 Householder-Transformationen

Ziel Zur Transformation der Regressionsfragestellung gilt es, eine Transformationsmatrix $\mathbf{Q}$ zu finden, mit der die Designmatrix $\mathbf{X}$ in die Matrix $\mathbf{X}^*$ überführt wird. Die Spalten von $\mathbf{X}$ sollen so im hinteren Teil Nullen erhalten, dass der obere Teil von $\mathbf{X}^*$ aus einer oberen Dreiecksmatrix besteht.

Idee Die Spaltenvektoren von $\mathbf{X}^*$ dürfen also im hinteren Teil nur Nullen enthalten. Dieses können wir für einen Vektor $\mathbf{x}^*$ dadurch erreichen, dass wir seine ersten $t-1$ Koordinaten unangetastet lassen und den Vektor der verbleibenden Koordinaten ohne Längenveränderung so transformieren, dass er in Richtung des t-ten Basisvektors zeigt. Nach diesem Ansatz ist der $(n-t+1)$-dimensionale Teilvektor $(x_t, \ldots, x_n)'$ zu drehen, dass nur die erste Koordinate des transformierten Teilvektor $(x_t^*, \ldots, x_n^*)'$ einen Wert verschieden von Null besitzt. Durch Rotation wird dann $\mathbf{x}$ in den Vektor $\mathbf{x}^* = \mathbf{H}_t\mathbf{x}$ überführt mit

$$\mathbf{x}^* = (x_1, \ldots, x_{t-1}, h_t, 0, \ldots 0)'$$

Damit die Länge von $\mathbf{x}^*$ der von $\mathbf{x}$ entspricht, muss gelten:

$$h_t^2 = \sum_{i=t}^{n} x_i^2$$

Verwendung Angenommen wir wären in der Lage, solche Transformationsmatrizen $\mathbf{H}_t$ konstruieren. Dann können wir $\mathbf{X}$ mit $\mathbf{H}_1$ transformieren und den ersten Spaltenvektor von $\mathbf{X}$ in den Vektor $(x_{1,1}^*, 0, \ldots, 0)'$ überführen. Mit der Transformation $\mathbf{H}_2$ wird der zweite Vektor in den Vektor $(x_{1,2}^*, x_{2,2}^*, 0, \ldots, 0)'$ transformiert usw.

Wichtig ist hierbei, dass der im ersten Schritt transformierte Vektor nicht wieder verändert wird. Da unser $\mathbf{H}_t$ die ersten $(t-1)$-Elemente unangetastet lässt, bleibt für $t = 2$ das erste Element $x_{1,1}^*$ unberührt, und die hinteren Elemente (mit den Indizes $t > 2$) nehmen den Wert 0 an, also den Wert, den sie bereits hatten. Jetzt sollten wir noch dem zweiten Element des ersten Vektors unsere Aufmerksamkeit schenken: Die Transformation $\mathbf{H}_2$ ist so konstruiert, dass die Länge des zu transformierenden Teilvektors von t bis n unverändert bleibt. Da diese Länge für $(x_{1,1}^*, 0, \ldots, 0)$ gerade 0 war, wird – wie gewünscht – das zweite

Elemente des ersten Spaltenvektors auf Null gesetzt bzw. bleibt auch das zweite Element unverändert.

Diese Überlegungen lassen sich für die weiteren Transformation mit den Matrizen $\mathbf{H}_3, ..., \mathbf{H}_p$ wiederholen und führen zu dem Ergebnis, dass die gewünschte obere Dreiecksmatrix durch Transformation von $\mathbf{X}$ mittels $\mathbf{Q}$ ersteht:

$$\mathbf{X}^* = \mathbf{QX} = \mathbf{H}_p\mathbf{H}_{p-1}\cdots\mathbf{H}_1\mathbf{X} \quad \wedge \quad \mathbf{y}^* = \mathbf{Qy} = \mathbf{H}_p\mathbf{H}_{p-1}\cdots\mathbf{H}_1\mathbf{y}$$

Konkretisierung Jetzt ist noch zu klären, wie man eine Householder-Matrix $\mathbf{H}_t$ konstruiert. Hierzu setze für vorgegebenes t

$$\mathbf{u} = (0, \ldots, 0, x_t \pm s, x_{t+1}, \ldots, x_n)' \quad \wedge \quad s^2 = \sum_{j=t}^{n} x_j^2$$

und bilde

$$\mathbf{H}_t = \left(\mathbf{I} - 2 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2}\right)$$

Offensichtlich enthält die Konstruktionsvorschrift dadurch noch einen Freiheitsgrad, dass an der Stelle $\pm$ noch ein Vorzeichen zu entscheiden ist. Zur Vermeidung von nummerischen Schwierigkeiten sollte das Vorzeichen von x_t an den fraglichen Stellen eingesetzt werden.

Zu diesem Vorschlag werden sich soforn zwei Fragen einstellen, ohne deren positive Antworten kein funktionsfähiges Verfahren resultiert.

Frage 1 Ist $\mathbf{H}_t$ eine Rotationsmatrix?

Wenn wir $\mathbf{H}_t$ – wie beschrieben – bilden, folgt $\mathbf{H}_t = \mathbf{H}_t'$ und

$$\mathbf{H}_t\mathbf{H}_t' = \mathbf{H}_t'\mathbf{H}_t = \mathbf{I},$$

denn es gilt:

$$\begin{aligned}
\mathbf{H}_t'\mathbf{H}_t &= \left(\mathbf{I} - 2 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2}\right)' \left(\mathbf{I} - 2 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2}\right) \\
&= \mathbf{I} - 2 \cdot \frac{(\mathbf{uu}')'}{|\mathbf{u}|^2} - 2 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2} + 4 \cdot \frac{\mathbf{uu}'\mathbf{uu}'}{|\mathbf{u}|^2|\mathbf{u}|^2} \\
&= \mathbf{I} - 4 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2} + 4 \cdot \frac{\mathbf{u}\left(\frac{\mathbf{u}'\mathbf{u}}{|\mathbf{u}|^2}\right)\mathbf{u}'}{|\mathbf{u}|^2} \\
&= \mathbf{I} - 4 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2} + 4 \cdot \frac{\mathbf{uu}'}{|\mathbf{u}|^2} \quad = \quad \mathbf{I}.
\end{aligned}$$

Also hat $\mathbf{H}_t$ die Eigenschaften einer Rotationsmatrix.

Frage 2 Leistet $\mathbf{H}_t$ die gewünschte Transformation?

Für die Antwort helfen zwei Zusammenhänge. Zum einen berechnen wir vorab den Wert der quadrierten Länge von $\mathbf{u}$:

$$|\mathbf{u}|^2 \quad = \quad \mathbf{u}'\mathbf{u}$$

$$
\begin{aligned}
&= (x_t \pm s)^2 + \sum_{t+1}^{n} x_i^2 \\
&= (x_t^2 \pm 2x_t \cdot s + s^2) + \sum_{t+1}^{n} x_i^2 \\
&= \pm 2x_t \cdot s + s^2 + \sum_{t}^{n} x_i^2 \\
&= 2 \cdot (\pm x_t \cdot s + s^2)
\end{aligned}
$$

Als zweites wollen wir das innere Produkt von $\mathbf{u}$ und $\mathbf{x}$ näher betrachten.

$$
\begin{aligned}
\mathbf{u}'\mathbf{x} &= (x_t \pm s) \cdot x_t + \sum_{t+1}^{n} x_i^2 \\
&= x_t^2 \pm s \cdot x_t + \sum_{t+1}^{n} x_i^2 \\
&= \pm s \cdot x_t + \sum_{t}^{n} x_i^2 \\
&= \pm s \cdot x_t + s^2
\end{aligned}
$$

Nach dieser Vorarbeit fällt folgende Umformung leicht:

$$
\begin{aligned}
\mathbf{H}_t\mathbf{x} &= \mathbf{x} - 2\mathbf{u} \cdot \frac{\mathbf{u}'\mathbf{x}}{|\mathbf{u}|^2} \\
&= \mathbf{x} - 2\mathbf{u} \cdot \frac{\pm s \cdot x_t + s^2}{2 \cdot (\pm x_t \cdot s + s^2)} \\
&= \mathbf{x} - \mathbf{u}
\end{aligned}
$$

Für das transformierte $\mathbf{x}$ können wir mit der Erinnerung an

$$\mathbf{u} = (0, \ldots, 0, x_t \pm s, x_{t+1}, \ldots, x_n)'$$

feststellen: Da die ersten $(t-1)$ Elemente von $\mathbf{u}$ Null sind, bleiben auch die ersten $(t-1)$ Elemente von $\mathbf{x}$ unverändert. Von den Elementen von $\mathbf{x}$ mit den Indizes aus $\{t+1, \ldots, n\}$ wird indes $(x_{t+1}, \ldots, x_n)$ subtrahiert, so dass sich in dem hinteren Bereich von $\mathbf{x}$ – wie gewünscht – 0-Werte einstellen. Die t-te Koordinate des Vektors $\mathbf{x}$ nimmt den Wert $x_t - (x_t \pm s) = \mp s$ an. Also ist die Struktur genau so, wie wir sie haben wollten.

Eine kurze Demonstration So kompliziert sich das Vorgehen anhört, ist eine direkte Umsetzung nicht schwer:

135 ⟨*define functions for Householder Rotations* 135⟩ ≡ ⊂ 136

```
find.u <- function(t, x){
  s <- abs(sum( x[t:n]^2 ))^0.5 * sign(x[t])
  u <- c( if(1 < t) rep(0, t-1), x[t] + s , x[-(1:t)])
  return(u)
}
```

```
find.H <- function(t, X){
  x <- X[,t]
  u <- find.u(t, x) #; print(u)
  I <- diag(n)
  H <- I - 2 * outer(u,u) / sum(u*u)
  H
}
```

Für eine Zufallsmatrix kommen wir zu folgendem Rotationsergebnis:

136 ⟨ * 1⟩+ ≡

```
⟨define functions for Householder Rotations 135⟩
set.seed(13); n <- 6; p <- 4
X <- matrix(rnorm(n*p), n, p); print(round(X,4))
cat("quadrierte Laengen der Spalten von X:")
print(colSums(X*X))
for(t in 1:p){
  H <- find.H(t,X)
  X <- H%*%X
}
print(round(X,4))
cat("quadrierte Laengen der Spalten von X^*:")
print(colSums(X*X))
```

Hier ist der Output:

```
       [,1]    [,2]    [,3]    [,4]
[1,]  0.5543  1.2295 -1.3610 -0.1144
[2,] -0.2803  0.2367 -1.8560  0.7022
[3,]  1.7752 -0.3654 -0.4399  0.2625
[4,]  0.1873  1.1051 -0.1939  1.8362
[5,]  1.1425 -1.0936  1.3964  0.3574
[6,]  0.4155  0.4619  0.1007 -1.0454
quadrierte Laengen der Spalten von X:
[1] 5.050152 4.331825 7.488358 5.167260
       [,1]    [,2]    [,3]    [,4]
[1,] -2.2473  0.3933 -0.2607 -0.2330
[2,]  0.0000 -2.0438  1.7342 -0.5758
[3,]  0.0000  0.0000  2.1007 -0.1369
[4,]  0.0000  0.0000  0.0000 -2.1824
[5,]  0.0000  0.0000  0.0000  0.0000
[6,]  0.0000  0.0000  0.0000  0.0000
quadrierte Laengen der Spalten von X^*:
[1] 5.050152 4.331825 7.488358 5.167260
```

Ergebnis:

Einsatzüberlegungen Es wäre ungeschickt, wenn wir zur Lösung unseres Regressionsproblems so viele Matrix-Operationen bewältigen müssten. Deshalb folgt noch eine Diskussion der Fragen, was überhaupt zu rechnen ist und welche Operationen als überflüssig weggelassen werden können.

Für die direkte Umsetzung durch Matrizenmultiplikationen sind p $(n \times n)$-Matrizen $\mathbf{H}_t$ aufzubauen und zu multiplizieren. Das erfordert viel Platz und viele Operationen. Weiterhin ist einzuwenden, dass auf diesem Wege sehr viele innere Produkte zu berechnen sind, die nummerisch nicht unbedingt stabil sind.

Glücklicherweise lassen sich die Ergebnisse der einzelnen Multiplikationen mit den Matrizen $\mathbf{H}_t$ überlegen und mit viel geringerem Aufwand erzielen.

Betrachten wir einen beliebigen Vektor $\mathbf{v}$, der mit $\mathbf{H}_t$ bearbeitet werden soll. Dann wissen wir, dass die ersten $(t-1)$ Elemente unverändert bleiben. Für die übrigen gilt:

$$v_j \mapsto v_j - f u_j \quad \text{mit} \quad f = 2 \cdot \frac{\mathbf{u}'\mathbf{v}}{|\mathbf{u}|^2}$$

Wir müssen also nur ein inneres Produkt ausrechnen, bei dem die ersten $(t-1)$-Elemente mit $u_i = 0$ unberücksichtigt bleiben können.

Das wollen wir mal probieren.

137 ⟨* 1⟩+ ≡

```
set.seed(13); n <- 6; p <- 4; X <- matrix(rnorm(n*p),n,p)
for(t in 1:p){
  x <- X[,t]
  u <- rep(0,n); u[t:n] <- x[t:n]
  s <- sign(x[t]) * abs( sum(x[t:n] * x[t:n]) )^0.5
  u[t] <- u[t] + s
  u.red <- u[t:n]
  u.abs.q <- sum(u.red * u.red)
  for(j in 1:p){i
    v.red <- X[t:n,j]
    f <- 2*sum(u.red * v.red)/u.abs.q
    v.red <- v.red - f * u.red
    X[t:n,j] <- v.red
  }
}
round(X,4)
```

Als Output stellt sich ein:

```
       [,1]    [,2]    [,3]    [,4]
[1,] -2.2473  0.3933 -0.2607 -0.2330
[2,]  0.0000 -2.0438  1.7342 -0.5758
[3,]  0.0000  0.0000  2.1007 -0.1369
[4,]  0.0000  0.0000  0.0000 -2.1824
[5,]  0.0000  0.0000  0.0000  0.0000
[6,]  0.0000  0.0000  0.0000  0.0000
```

Wir erkennen in der äußeren Schleife nur zwei innere Produkte, eine Addition und einen Wurzelfunktionseinsatz. In der inneren Schleife müssen zentral eine Subtraktion, die Multiplikation mit `f` und ein inneres Produkt errechnet werden. Die Multiplikation mit 2 sowie die Division durch `u.abs.q` können wir noch in die äußere Schleife verschieben, so dass bei der resultierenden Lösung wenige Operationen auszuführen sind.

138 ⟨* 1⟩+ ≡

```
set.seed(13); n <- 6; p <- 4; X <- matrix(rnorm(n*p),n,p)
for(t in 1:p){
  x <- X[,t]
  u <- rep(0,n); u[t:n] <- x[t:n]
  s <- sign(x[t]) * abs( sum(x[t:n] * x[t:n]) )^0.5
  u[t] <- u[t] + s
  u.red <- u[t:n]
```

```
  u.abs.q <- sum(u.red * u.red)
  u.red.1 <- (2/u.abs.q) * u.red
  for(j in 1:p){
    v.red <- X[t:n,j]
    f <- sum(u.red.1 * v.red)
    v.red <- v.red - f * u.red
    X[t:n,j] <- v.red
  }
}
round(X,4)
```

Oder eingedampft und per Funktion umgesetzt:

139 ⟨*define* `compute.Householder.transformations()` 139⟩ ≡ ⊂ 140, 142

```
compute.Householder.transformations <- function(X, p=ncol(X)){
  n <- nrow(X)
  for(t in 1:p){
    u.red <- x.red <- X[t:n,t]
    s <- sign(x[1]) * abs( x.red %*% x.red )^0.5 # abs k.w.
    u.red[1] <- u.red[1] + s
    u.red.1 <- (2/( u.red %*% u.red ))*u.red
    for(j in 1:ncol(X)){
      f <- u.red.1 %*% X[t:n,j]
      X[t:n,j] <- X[t:n,j] - f*u.red
    }
  }
  return(X)
}
```

... und angewendet

140 ⟨* 1⟩+ ≡

```
⟨define compute.Householder.transformations() 139⟩
set.seed(13); n <- 6; p <- 4; X <- matrix(rnorm(n*p),n,p)
print(round(X,4))
X.star <- compute.Householder.transformations(X)
print(round(X.star,4))
```

Ergebnis:

```
       [,1]    [,2]    [,3]    [,4]
[1,]  0.5543  1.2295 -1.3610 -0.1144
[2,] -0.2803  0.2367 -1.8560  0.7022
[3,]  1.7752 -0.3654 -0.4399  0.2625
[4,]  0.1873  1.1051 -0.1939  1.8362
[5,]  1.1425 -1.0936  1.3964  0.3574
[6,]  0.4155  0.4619  0.1007 -1.0454
      [,1]    [,2]    [,3]    [,4]
[1,] 2.2473 -0.3933  0.2607  0.2330
[2,] 0.0000  2.0438 -1.7342  0.5758
[3,] 0.0000  0.0000  2.1007 -0.1369
[4,] 0.0000  0.0000  0.0000  2.1824
[5,] 0.0000  0.0000  0.0000  0.0000
[6,] 0.0000  0.0000  0.0000  0.0000
```

Hilft die nahegelegte Transformation? Damit wir von der Funktionsfähigkeit überzeugt sind, müssen wir auch unser Beispielproblem für den Polynom-

grad $k = 8$ lösen können. Es gilt also, die Design-Matrix $\mathbf{X}$ und den Beobachtungsvektor $\mathbf{y}$ zu transformieren und dann das reduzierte Problem zu lösen. Für die Umsetzung können wir im ersten Schritt an die Designmatrix $\mathbf{X}$ den Vektor $\mathbf{y}$ anhängen und die Transformation umsetzen. Zunächst definieren wir aber wieder das Datenmaterial.

141 ⟨*definiere Daten zum Problembeispiel:* `x, y, k, X` 141⟩ ≡ ⊂ 144

```
anzahl.daten <- 10                          #   Daten erzeugen
x <- 1:anzahl.daten; set.seed(13)           #   x-Werte
y <- sin(x/4) + rnorm(anzahl.daten)         #   y-Werte
k <- 8                                      #   max Polynomgrad setzen
X <- outer(x,0:k,"^")                       #   Designmatrix Polynomgrad k
```

Nach der Aktivierung des letzten Chunks können wir unseren Test starten.

142 ⟨*ermittle Transformation von* $\mathbf{X}$ *und* $\mathbf{y}$ 142⟩ ≡ ⊂ 143

```
# result <- solve(t(X)%*%X)%*%t(X)%*%y      #   Versuch, ein Modell anzupassen
⟨define compute.Householder.transformations() 139⟩
hh.result <- compute.Householder.transformations(cbind(X,y), p=9)
print(round(hh.result,4))
```

Wir erhalten:

```
 [1,] -3.1623 -17.3925 -121.7477 -956.5890 -8010.9980 -69830.9964 -625626.5934
 [2,]  0.0000  -9.0830  -99.9125 -957.3430 -8972.1391 -84099.0439 -792605.5585
 [3,]  0.0000   0.0000  -22.9783 -379.1411 -4641.6066 -51184.0532 -538403.3895
 [4,]  0.0000   0.0000    0.0000  -55.5770 -1222.6934 -18247.7730 -232311.7499
 [5,]  0.0000   0.0000    0.0000    0.0000  -128.3495  -3529.6119  -62162.0075
 [6,]  0.0000   0.0000    0.0000    0.0000     0.0000   -279.2848   -9216.3984
 [7,]  0.0000   0.0000    0.0000    0.0000     0.0000      0.0000    -560.5192
 [8,]  0.0000   0.0000    0.0000    0.0000     0.0000      0.0000       0.0000
 [9,]  0.0000   0.0000    0.0000    0.0000     0.0000      0.0000       0.0000
[10,]  0.0000   0.0000    0.0000    0.0000     0.0000      0.0000       0.0000
                                        y
 [1,] -5717532.41 -53041304.726 -4.2586
 [2,] -7518370.96 -71758908.424 -0.3266
 [3,] -5528256.88 -56012811.383  0.9654
 [4,] -2723495.70 -30425169.608 -0.5634
 [5,]  -898446.66 -11638176.451 -0.8077
 [6,]  -187077.85  -3027232.407 -0.3606
 [7,]   -21579.99   -495947.423 -1.0989
 [8,]    -1003.29    -44144.747  0.7401
 [9,]        0.00     -1507.882 -0.5340
[10,]        0.00         0.000 -1.0577
```

Diese Matrix lässt sich wieder zerlegen in $\mathbf{X}^*$ und $\mathbf{y}^*$, und wir können das Gleichungssystem

$$\mathbf{X}^*\beta = \mathbf{y}^*$$

sukzessive lösen. Dabei sind von $\mathbf{X}^*$ nur die ersten p Zeilen zu berücksichtigen, denn die weitere enthalten nur 0-Werte. Wir hatten im Beispiel einen Polynomgrad von $k = 8$ gewählt, so dass hier gilt: $p = 9$.

143 ⟨*ermittle zu* $\mathbf{X}$ *und* $\mathbf{y}$ *Koeffizientenvektor* `beta.hh.dach` 143⟩ ≡ ⊂ 144

```
⟨ermittle Transformation von X und y 142⟩
p <- ncol(hh.result) - 1
X.star <- hh.result[1:p, -(p+1)]; y.star <- hh.result[1:p, p+1]
beta.hh.dach <- rep(0,p); beta.hh.dach[p] <- y.star[p]/X.star[p,p]
for(i in (p-1):1){
```

```
  beta.hh.dach[i] <-
    (y.star[i] - X.star[i,(i+1):p] %*% beta.hh.dach[(i+1):p]) / X.star[i,i]
}
round(beta.hh.dach,4)
```

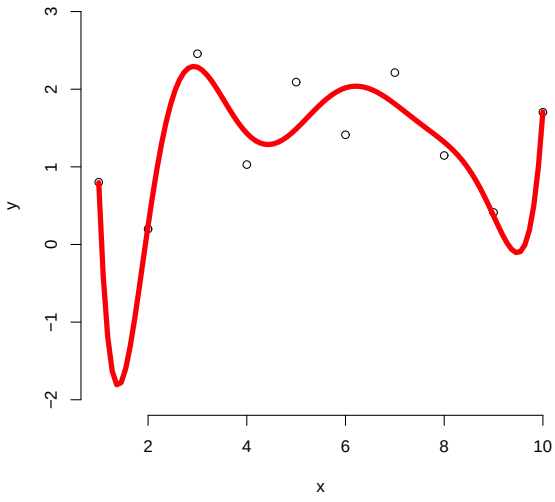
Hier ist das Ergebnis:

```
[1]   81.5181 -191.8440  172.7362  -79.7050   21.1549   -3.3642    0.3169
[8]   -0.0163    0.0004
```

Zu diesen Zahlen wollen wir natürlich das graphische Ergebnis unserer Berechnungen sehen!

144 ⟨* 1⟩+ ≡

```
⟨definiere Daten zum Problembeispiel: x, y, k, X 141⟩
⟨ermittle zu X und y Koeffizientenvektor beta.hh.dach 143⟩
xx <- seq(1,10,length=100)                    #   x-Werte -> Modelllinie
yy.dach <- outer(xx,0:k,"^") %*% beta.hh.dach #   Modelllinie finden
plot(x,y, bty="n", ylim=c(-2,3))              #   Punkte und
lines(xx,yy.dach, lwd=7, col="red")           #   ... Modell zeichnen
```



Abschließend interessiert eventuell noch die Frage, ob R-Funktionen wie `lsfit()` oder `lm()` ebenfalls obiges Resultat erarbeiten.

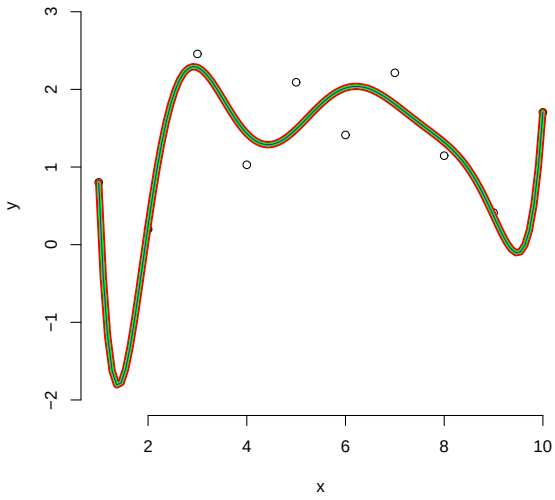
145 ⟨* 1⟩+ ≡

```
X.o.a <- X[,-1]                            # Designmatrix Polynomgrad k ohne const
coef.lm <- lm(y~X.o.a)$coef                # Koeffizientenermittlung mit lm
yy.dach.lm <- outer(xx,0:8,"^") %*% coef.lm # Modelllinie finden
lines(xx,yy.dach.lm, lwd=3, col="green")   # ... Modell zeichnen
coef.ls <- lsfit(X.o.a,y)$coef             # Koeffizientenermittlung
yy.dach.ls <- outer(xx,0:k,"^") %*% coef.ls # Modelllinie finden
lines(xx,yy.dach.ls, lwd=1, col="blue")    # ... Modell zeichnen
```

```
print(cbind(beta.hh=round(beta.hh.dach,4),
      coef.lm=round(coef.lm,4), coef.ls=round(coef.ls,4)))
```

Das zusammenfassende Ergebnis lautet:

```
             beta.hh   coef.lm   coef.ls
(Intercept)  81.5181   81.5181   81.5181
X.o.a1      -191.8440 -191.8440 -191.8440
X.o.a2       172.7362  172.7362  172.7362
X.o.a3       -79.7050  -79.7050  -79.7050
X.o.a4        21.1549   21.1549   21.1549
X.o.a5        -3.3642   -3.3642   -3.3642
X.o.a6         0.3169    0.3169    0.3169
X.o.a7        -0.0163   -0.0163   -0.0163
X.o.a8         0.0004    0.0004    0.0004
```



Damit ist exemplarisch die Funktionstüchtigkeit nachgewiesen.

14.3.4 Gram-Schmid-Verfahren

In diesem Abschnitt stellen wir nur eine fehlerkorrigierte R-Übersetzung des Algorithmus MGS von Thisted (1988, p. 73) vor, mit der der Leser eigene Experimente unternehmen möge.

146 ⟨ * 1⟩+ ≡

```
MGS <- function(X,digits=15){
  p <- ncol(X); n <- nrow(X); R <- matrix(0,p,p)
  for(j in 1:p){
    R[j,j] <- sqrt(sum(X[,j]^2))
    for(i in 1:n){
      X[i,j] <- X[i,j]/R[j,j]
    }
  }
```

```
  for(j in 1:(p-1)){ # Thisted: j in 1:p ?!
    for(k in (j+1):p){
      R[j,k] <- X[,j] %*% X[,k]
      for(i in 1:n){
        X[i,k] <- X[i,k] - X[i,k]*R[j,k]
      }
    }
  }
  return(list(R=round(R,digits), X=round(X,digits)))
}
set.seed(13); n <- 6; p <- 4; X <- matrix(rnorm(n*p),n,p)
MGS(X,4)
```

Für unsere eine Beispielmatrix erhalten wir:

```
$R
       [,1]    [,2]    [,3]    [,4]
[1,] 2.2473 -0.1890  0.0953  0.1025
[2,] 0.0000  2.0813 -0.6888  0.2447
[3,] 0.0000  0.0000  2.7365 -0.2040
[4,] 0.0000  0.0000  0.0000  2.2732
$X
        [,1]    [,2]    [,3]    [,4]
[1,]  0.2467  0.7024 -0.7599 -0.0411
[2,] -0.1247  0.1352 -1.0363  0.2521
[3,]  0.7899 -0.2087 -0.2456  0.0943
[4,]  0.0834  0.6313 -0.1083  0.6592
[5,]  0.5084 -0.6247  0.7797  0.1283
[6,]  0.1849  0.2639  0.0562 -0.3753
```

14.3.5 Givens Rotations

Geometrisch betrachtet ist $\mathbf{y}$ ein Vektor im n-dimensionalen Raum $\mathcal{R}^n$. Die p Spaltenvektoren der $(n \times p)$-Matrix $\mathbf{X}$ definieren in diesem Raum für $p < n$ einen p-dimensionalen Unterraum, sofern keine pathologischen Fälle vorliegen. Der Vektor der geschätzten y-Werte $\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$ ist eine Linearkombination der Spaltenvektoren von $\mathbf{X}$ und liegt damit im Raum von $\mathbf{X}$. Da

$$\mathbf{u}'\mathbf{u} = (\mathbf{y} - \hat{\mathbf{y}})'(\mathbf{y} - \hat{\mathbf{y}})$$

minimal ist, steht $(\mathbf{y} - \hat{\mathbf{y}})$ senkrecht auf dem Raum von $\mathbf{X}$. Also ist $\hat{\mathbf{y}}$ die orthonale Projektion von $\mathbf{y}$ auf den Raum von $\mathbf{X}$.

An dieser geometrischen Vorstellung ändert sich nichts, wenn wir den n-dimensionalen Raum rotieren, ohne dabei irgendwelche Längen und Längenverhältnisse zu verändern. Damit müsste folgender Algorithmus mittels Givens-Rotationen (nach Wallace Givens) funktionieren:

Givens-Rotationen

1. Setze i auf 1.
2. Rotiere das System so, dass alle Dimensionen kleiner i unverändert bleiben und in allen Dimensionen größer als i ein Wert von 0 entsteht.

3. Zähle i hoch.

4. Falls $i \leq p$ gehe zu Schritt 2.

Nach der ersten Bearbeitung zeigt die alte erste Spalte $\mathbf{X}$ in Richtung der ersten Koordinate. Nach der zweiten liegt der zweite Spalten-Vektor von $\mathbf{X}$ in der Ebenen der ersten beiden Koordinaten usw.

Schrittweises Annähern. Wie können wir diese einfache Idee umsetzen? Hierzu überlegen wir kurz, wie man einen Vektor im $\mathcal{R}^2$ drehen kann, dann übertragen wir diesen Ansatz auf den $\mathcal{R}^n$ und erarbeiten schließlich eine Übertragung auf unsere Problemsituation.

Rotationen eines Vektors der Länge 2 lassen sich durch Multiplikation einer geeigneten (2×2)-Matrix erzielen. Rotationsmatrizen haben die Gestalt:

$$\mathbf{G} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

Damit die erste Koordinate durch Drehung verschwindet, setzen wir:

$$\sin\theta = x_2/\sqrt{x_1^2 + x_2^2} \quad \text{und} \quad \cos\theta = x_1/\sqrt{x_1^2 + x_2^2}.$$

Als Beispielvektor drehen wir einmal $\mathbf{x}' = (3, 7)$ so, dass die zweite Koordinate eliminiert wird:

147 ⟨ * 1⟩+ ≡
```
x <- c(3,7)
si <- x[2]/sqrt(sum(x*x)); co <- x[1]/sqrt(sum(x*x))
G <- matrix(c(co,-si,si,co),2,2)
x.stern <- G %*% x
cat("x:",    x,      "/ alte Norm:",sqrt(sum(x*x)),
    "\nx.stern:",x.stern,"/ neue Norm:",sqrt(sum(x.stern*x.stern)))
```

Bis auf einen kleinen Fehler ist die zweite Koordinate verschwunden. Jedoch ist die Länge gleichgeblieben.

```
x: 3 7 / alte Norm: 7.615773
x.stern: 7.615773 -1.110223e-16 / neue Norm: 7.615773
```

A. 14.9: Begründe, warum diese Art der Transformation Längen von Vektoren nicht verändert.

Nun wollen wir die Drehidee für einen Vektor der Länge $n = 5$ probieren. Zunächst drehen wir die letzte Koordinate weg.

148 ⟨*drehe x so, dass letzte Dimension wegfällt* 148⟩ ≡
```
n  <- 5; x <- 1:5
a  <- n-1; b <- n
len<- sqrt(sum(x[a:b]^2))
si <- x[b]/len; co <- x[a]/len
G  <- diag(n); G[a:b,a:b] <- matrix(c(co,-si,si,co),2,2)
x  <- G %*% x
```

Wir erhalten:

```
             [,1]
[1,]  1.000000e+00
[2,]  2.000000e+00
[3,]  3.000000e+00
[4,]  6.403124e+00
[5,] -1.110223e-16
```

Die ersten drei Koordinaten-Werte sind unverändert, die fünfte fast 0 und die dritte wurde ein wenig vergrößert. Als nächstes versuchen wir, mehrere Koordinaten von hinten beginnend zu entfernen. Hierzu muss immer eine neue Rotationsmatrix konstruiert werden.

149 ⟨*drehe x, dass schrittweise die hinteren Koordinaten verschwinden* 149⟩ ≡

```
n <- 5; x <- 1:5; k <- 2; EM <- diag(n)
for(i in c(1:k)){
  a <- n-i; b <- n-(i-1)
  len <- sqrt(sum(x[a:b]^2))
  si <- x[b]/len; co <- x[a]/len
  G <- EM; G[a:b,a:b] <- matrix(c(co,-si,si,co),2,2)
  x <- G %*% x
  print(x)
}
```

Damit sich die Arbeit in der Schleife verfolgen lässt, wird `x` jeweils ausgegeben:

```
             [,1]
[1,]  1.000000e+00
[2,]  2.000000e+00
[3,]  3.000000e+00
[4,]  6.403124e+00
[5,] -1.110223e-16
             [,1]
[1,]  1.000000e+00
[2,]  2.000000e+00
[3,]  7.071068e+00
[4,]  8.131516e-17
[5,] -1.110223e-16
```

Die Anweisungen sehen brauchbar aus, so dass wir sie in eine Funktion gießen können. Diese soll uns für das Problem: *rotiere solange, bis die letzten k Koordinaten verschwunden sind* eine Rotationsmatrix ausgeben.

150 ⟨*definiere* `find.rotate.matrix()` 150⟩ ≡ ⊂ 151, 152

```
find.rotate.matrix <- function(x,k=1){
  n <- length(x); G.k <- EM <- diag(n)
  for(i in c(1:k)){
    a <- n-i; b <- n-(i-1)
    len <- sqrt(sum(x[a:b]^2))
    si <- x[b]/len; co <- x[a]/len
    G <- EM; G[a:b,a:b] <- matrix(c(co,-si,si,co),2,2)
    x <- G %*% x
    G.k <- G %*% G.k
  }
  G.k
}
```

Ein Test ist schnell erledigt.

151 ⟨* 1⟩+ ≡

⟨*definiere* `find.rotate.matrix()` 150⟩

```
x <- 1:5
G.k <- find.rotate.matrix(x,k=3)
G.k %*% x
```

Der Test liefert:

```
             [,1]
[1,]  1.000000e+00
[2,]  7.348469e+00
[3,] -5.551115e-17
[4,]  1.110223e-16
[5,] -1.110223e-16
```

A. 14.10: Notiere die vorgestellten Handlungen mittels mathematischer Formeln.

Wir können nun eine zusammenhängende Funktion schreiben, die die notwendige Gesamtrotationsmatrix zur Transformation unseres Regressionsproblems liefert. Die gesamte Rotation lässt sich durch eine Matrix repräsentieren, die sich durch Multiplikation der einzelnen Rotationsmatrizen ergibt.

152 ⟨*definiere* `givens()` 152⟩ ≡ ⊂ 153, 155, 156

```
givens <- function(X,y){
  ⟨definiere find.rotate.matrix() 150⟩
  n <- dim(X)[1]; p <- dim(X)[2]
  G.all <- diag(n)
  for(j in 1:p){
    G.k <- find.rotate.matrix(X[,j],n-j)
    X <- G.k %*% X
    G.all <- G.k %*% G.all
  }
  G.all
}
```

A. 14.11: Suche eine formale Beschreibung von Givens-Rotationen und versuche Elemente der Umsetzungen zu identifizieren.

Der Leser ist sicher gespannt, ob nach der Transformation auch wirklich dieselben Koeffizienten resultieren.

153 ⟨*check Rotationsansatz* 153⟩ ≡

```
n <- 7; p <- 3; set.seed(13)
X <- cbind(1, XX <- matrix(sample(1:100,n*(p-1)),n,p-1)); y <- sample(1:100,n)
⟨definiere givens() 152⟩
G.all <- givens(X); X.stern <- G.all %*% X; y.stern <- G.all %*% y
coef1 <- solve(t(X)%*%X)%*%t(X)%*%y
coef2 <- solve(t(X.stern)%*%X.stern)%*%t(X.stern)%*%y.stern
cat("Original-Problem / coef1:",  coef1,
    "\ntransformiert    / coef2:",coef2)
```

Der Output entspricht unserer Hoffnung:

```
Original-Problem / coef1: 21.66405 0.1106392 0.3646301
transformiert    / coef2: 21.66405 0.1106392 0.3646301
```

Als letzten Schritt lösen wir das Gleichungssystem im Rahmen einer Schleife, wie es schon oben angekündigt war. Diesen Job soll die Funktion `solve_beta_diag()` erledigen.

154 ⟨*definiere* `solve_beta_diag()` 154⟩ ≡ ⊂ 155, 156

```
solve_beta_diag <- function(XX,yy){
  p <- dim(XX)[2]; beta <- rep(0,p)
  for(i in p:1){
    h <- XX[i,] %*% beta
    beta[i] <- (yy[i] - h)/XX[i,i]
  }
  beta
}
```

A. 14.12: Notiere ein von der Struktur her passendes (3×3)-System und löse die Auswertaufgabe per Hand.

Ein Aufruf von `givens(X)` führt zum Ziel ...

155 ⟨* 1⟩+ ≡

```
n <- 7; p <- 3; set.seed(13)
X <- cbind(1, XX <- matrix(sample(1:100,n*(p-1)),n,p-1)); y <- sample(1:100,n)
⟨definiere givens() 152⟩
⟨definiere solve_beta_diag() 154⟩
G.all <- givens(X); X.stern <- G.all %*% X; y.stern <- G.all %*% y
X.stern.1 <- X.stern[1:p,]; y.stern.1 <- y.stern[1:p]
solve_beta_diag(X.stern.1,y.stern.1)
```

... und wir erhalten:

```
[1] 21.6640496  0.1106392  0.3646301
```

Wer hätte das gedacht? So, nun schauen wir uns unseren oben vorgestellten Testfall an:

156 ⟨*check Givens Rotationen für Problemdaten* 156⟩ ≡ ⊂ 157, 160

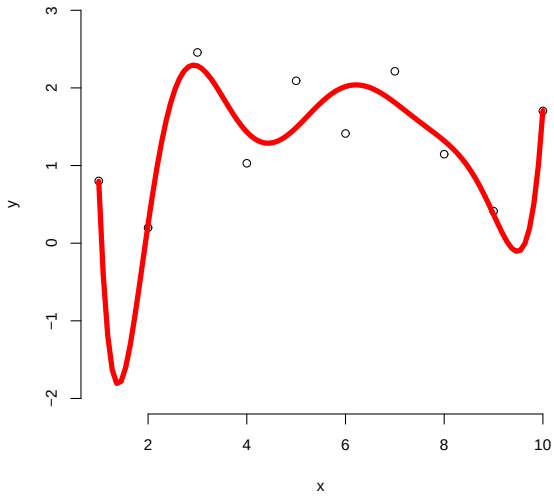
```
anzahl.daten <- 10                          #   Daten erzeugen
x <- 1:anzahl.daten; set.seed(13)           #   x-Werte
y <- sin(x/4) + rnorm(anzahl.daten)         #   y-Werte
xx <- seq(1,10,length=100)                  #   x-Werte -> Modelllinie
k <- 8                                          #   max Polynomgrad setzen
X <- outer(x,0:k,"^")                       #   Designmatrix Polynomgrad k
cat("alte Designmatrix"); print(X)          #   und ausdrucken
⟨definiere givens() 152⟩
⟨definiere solve_beta_diag() 154⟩
G.all <- givens(X)                           #   Rotationsmatrix ermitteln
X.stern <- G.all %*% X; y.stern <- G.all %*% y   # Transformation
X.stern.1 <- X.stern[1:(k+1),]              #   Reduktion
y.stern.1 <- y.stern[1:(k+1)]               #   Reduktion
beta.dach <- solve_beta_diag(X.stern.1,y.stern.1)# solve per Schleife
yy.dach <- outer(xx,0:k,"^") %*% beta.dach       #   Modelllinie finden
plot(x,y, bty="n", ylim=c(-2,3))            #   Punkte und
lines(xx,yy.dach, lwd=5, col="red")         #   ... Modell zeichnen
cat("neue Designmatrix"); print(noquote(format(X.stern.1,digits=3)))
```

```
 cat("beta.dach"); beta.dach
```

Wir erhalten:

```
alte Designmatrix
      [,1] [,2] [,3] [,4]  [,5]   [,6]    [,7]     [,8]      [,9]
 [1,]    1    1    1    1     1      1       1        1         1
 [2,]    1    2    4    8    16     32      64      128       256
 [3,]    1    3    9   27    81    243     729     2187      6561
 [4,]    1    4   16   64   256   1024    4096    16384     65536
 [5,]    1    5   25  125   625   3125   15625    78125    390625
 [6,]    1    6   36  216  1296   7776   46656   279936   1679616
 [7,]    1    7   49  343  2401  16807  117649   823543   5764801
 [8,]    1    8   64  512  4096  32768  262144  2097152  16777216
 [9,]    1    9   81  729  6561  59049  531441  4782969  43046721
[10,]    1   10  100 1000 10000 100000 1000000 10000000 100000000
neue Designmatrix
      [,1]       [,2]       [,3]       [,4]       [,5]       [,6]       [,7]
 [1,]  3.16e+00  1.74e+01  1.22e+02  9.57e+02  8.01e+03  6.98e+04  6.26e+05
 [2,] -7.77e-16  9.08e+00  9.99e+01  9.57e+02  8.97e+03  8.41e+04  7.93e+05
 [3,]  2.22e-16  4.44e-16  2.30e+01  3.79e+02  4.64e+03  5.12e+04  5.38e+05
 [4,]  1.67e-16  1.22e-15  6.88e-15  5.56e+01  1.22e+03  1.82e+04  2.32e+05
 [5,]  0.00e+00 -4.44e-16 -6.22e-15  8.88e-15  1.28e+02  3.53e+03  6.22e+04
 [6,]  1.11e-16  8.88e-16  7.11e-15  4.97e-14  6.11e-13  2.79e+02  9.22e+03
 [7,] -1.39e-17  5.27e-16  8.33e-16  4.74e-14  1.90e-13  6.45e-12  5.61e+02
 [8,]  1.25e-16  9.16e-16  1.05e-14  7.83e-14  9.11e-13  8.65e-12  5.74e-11
 [9,]  2.05e-16  1.27e-15  5.37e-15  1.06e-13  3.22e-13  2.37e-12  3.64e-11
      [,8]       [,9]
 [1,]  5.72e+06  5.30e+07
 [2,]  7.52e+06  7.18e+07
 [3,]  5.53e+06  5.60e+07
 [4,]  2.72e+06  3.04e+07
 [5,]  8.98e+05  1.16e+07
 [6,]  1.87e+05  3.03e+06
 [7,]  2.16e+04  4.96e+05
 [8,]  1.00e+03  4.41e+04
 [9,]  7.67e-12  1.51e+03
beta.dach
[1]  8.151812e+01 -1.918440e+02  1.727362e+02 -7.970501e+01  2.115488e+01
[6] -3.364243e+00  3.169288e-01 -1.631968e-02  3.541360e-04
```

... und auch eine graphische Darstellung:



A. 14.13: Überprüfe, ob `lm()` oder `lsfit()` zum selben Graphen führen. Prüfe also, was folgender Chunk liefert.

157 ⟨ * 1⟩+ ≡
⟨*check Givens Rotationen für Problemdaten* 156⟩

```
X.o.a <- outer(x,1:8,"^")                    #   Designmatrix Polynomgrad k
res.lm <- lm(y~X.o.a)$coef                   #   Koeffizientenermittlung mit lm
yy.dach.lm <- outer(xx,0:k,"^") %*% res.lm   #   Modelllinie finden
lines(xx,yy.dach.lm, lwd=3, col="green")     #   ... Modell zeichnen
res.ls <- lsfit(X.o.a,y)$coef                #   Koeffizientenermittlung
yy.dach.ls <- outer(xx,0:k,"^") %*% res.ls   #   Modelllinie finden
lines(xx,yy.dach.ls, lwd=1, col="blue")      #   ... Modell zeichnen
print(res.lm)
```

Wir stellen fest, das Ziel ist erreicht. Wir haben ein weiteres Verfahren gefunden, das uns die gesuchten Koeffizientenschätzungen liefert. Auch das Bild passt. Als große Idee merken wir uns, dass Problem-Transformationen für die Lösungsfindung sehr hilfreich sein können.

Nicht überlegt haben wir an dieser Stelle, inwieweit das Verfahren sparsam mit Rechnerressourcen umgeht und in welchen Fällen es besonders geeignet bzw. ungeeignet ist. Es wäre durchaus lohnenswert, andere Ansätze der Orthogonalisierung zu untersuchen und dann Vergleiche anzustellen. Für die Regressionsanalyse sei besonders noch darauf hingewiesen, dass bei den einzelnen Verfahren Zwischenprodukte anfallen, die sowohl für andere Zielgrößen der Regressionsanalyse hilfreich sind als auch zur Interpretation des Gegenstandes beitragen. Hier wollen wir uns jedoch nur noch mit einer Sache auseinandersetzen, nämlich der Frage, was sich hinter der immer noch etwas unverständlichen Fehlermeldung verbirgt.

A. 14.14: Stöbern Sie nach Begriffen wie Orthogonalisierung, QR-Zerlegung, singular value decomposition. Lesen Sie die Hilfe von `lsfit()` und `qr()`.

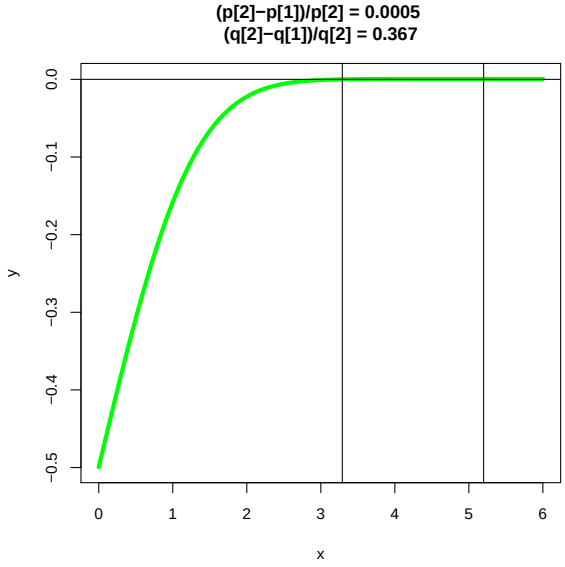
14.4 Kondition von Problemen

Es gibt Probleme, die schlecht konditioniert sind. Sie werden als `ill conditioned` bezeichnet. Diese Eigenschaft bedeutet, dass sehr kleine Änderungen des Inputs große Auswirkungen auf die Lösung haben. Eine schlechte Lösung eines Problems kann also in einem ungeeigneten Algorithmus begründet liegen, aber auch darin, dass das Problem als solches sehr labil ist. Bei der Diskussion von Fehlerfortpflanzungsfragen wird die Problematik aus Sicht der Umsetzung beleuchtet, wogegen es oft auch zweckmäßig ist, über die Konditioniertheit eine Problems vor seiner (algorithmischen) Lösung nachzudenken.

Als Beispiel können wir ein Nullstellenproblem betrachten, mit dem wir das 0.9995-Quantil der Standardnormalverteilung durch Verschiebung der Verteilungsfunktion um 0.9995 nach unten theoretisch finden können.

158 ⟨ * 1⟩+ ≡

```
x <- seq(0,6,length=100); p <- c(0.9995,0.9999999)
y <- pnorm(x) - p[1]
plot(x,y,type="l",lwd="4",col="green")
abline(h=0); abline(v=qnorm(p))
title(paste("(p[2]-p[1])/p[2] =",
            format((p[2]-p[1])/p[2],digits=3,sci=FALSE),
           "\n(q[2]-q[1])/q[2] =",
            format((qnorm(p[2])-qnorm(p[1]))/qnorm(p[2]),digits=3,sci=FALSE)))
```



Der Schnittpunkt der dicken Kurve mit der x-Achsen zeigt uns den 0.9995-Prozentpunkt der Standardnormalverteilung. Wir sehen, dass sich eine winzige (relative) Änderung der p-Werte in einer erheblichen (relativen) Änderung der Lösungswerte niederschlägt: Unabhängig vom eingesetztem Nullstellenverfahren ist damit das Problem an sich problematisch, es ist schlecht konditioniert, und

Vorsicht ist geboten.

Ähnlich problematisch sind folgende Berechnungen:

- Differenz zweier sehr ähnlich großen Zahlen.
- Vergleich von Werten, die in der Nähe von 0 liegen, mit dem Wert 0.
- Invertieren von Matrizen, deren Determinante betragsmäßig sehr klein ist. Solche Situationen treten auf, wenn Spalten oder Zeilen fast kollinear sind.

159 ⟨* 1⟩+ ≡

```
X <- matrix(c(1:3,2,4.001,6,7:9),3,3)
cat("Matrix");       print(X)
cat("Determinante"); print(det  (X))
cat("Inverse");      print(solve(X))
cat("Kondition");    print(1/rcond(X))
solve(X)
```

```
Matrix
     [,1]  [,2] [,3]
[1,]    1 2.000    7
[2,]    2 4.001    8
[3,]    3 6.000    9
Determinante
[1] -0.012
Inverse
        [,1]          [,2]          [,3]
[1,]  999.25 -2.000000e+03 1000.58333333
[2,] -500.00  1.000000e+03 -500.00000000
[3,]    0.25 -2.775558e-14   -0.08333333
Kondition
[1] 72000
```

Für die Diskussion der Kondition von Problemen ist es notwendig, das Phänomen zu quantifizieren. Das geschieht mit verschiedenen Maßzahlen zur Messung der Kondition.

A. 14.15: Studiere: http://de.wikipedia.org/wiki/Kondition_%28Mathematik%29

Konditionsbegriff. Seien f das angestrebte und $\tilde{f}$ das realisierte Berechnungsverfahren. Weiter soll x den richtigen Input und $\tilde{x}$ den auf einem Rechner realisierten Input beschreiben, dann gilt für den absoluten Fehler aufgrund der Dreiecksungleichung:

$$||f(x) - \tilde{f}(\tilde{x})|| \leq ||f(x) - f(\tilde{x})|| + ||f(\tilde{x}) - \tilde{f}(\tilde{x})|| =: C + S$$

C bezeichnet damit problemeigene Schwierigkeiten, die Kondition, wogegen S uns die Probleme durch die Umsetzung zeigt und die *Stabilität* des Algorithmus verkörpert. Durch die Transformation unseres Regressionsproblems haben wir einen stabileren Algorithmus zur Schätzung der Koeffizienten gefunden. Aber das Problem an sich dürfte sich in seinem Charakter (C) nicht verändert haben.

Betrachten wir ein paar Gedanken der Formalisierung von C. Für ein gegebenes x und kleine ϵ-Werte bedeutet ein großer Wert von

$$\frac{||f(x) - f(x+\epsilon)||}{||\epsilon||},$$

dass kleine Inputänderungen zu großen Output-Schwankungen führen. Wenn $f(x)$ sehr groß ist, kann eine kleine Änderung dieses Werte unbedeutend sein, so dass eine Normierung von Zähler und Nenner Sinn macht, und wir erhalten unter gutmütigen Bedingungen:

$$\kappa = \frac{||f(x) - f(x+\epsilon)||}{||f(x)||} \Big/ \frac{||\epsilon||}{||x||} \approx \frac{||f'(x)||}{||f(x)||} \cdot ||x||$$

Die Größe κ können wir als relative Kondition bezeichnen, wobei wir einige formale Feinheiten zugunsten des Gedanken ausgeblendet haben. Diese Größe erfasst im Problem begründete Schwierigkeiten und lässt sich nach Entscheidung für eine spezielle Norm $||\cdot||$, bspw. der L_p-Norm, berechnen und für Diskussionen nutzen.

A. 14.16: Überlege, wann die absolute und wann die relative Kondition betrachtet werden sollte.

Addition. Für eine Addition von x_1 und x_2, also mit dem Input-Vektor $(x_1, x_2)'$ ergibt sich bei Wahl der L_1-Norm, also der Manhattan-Metrik:

$$\begin{aligned}
\kappa &= \frac{||f(x) - f(x+\epsilon)||}{||f(x)||} \Big/ \frac{||\epsilon||}{||x||} \\
&= \frac{|x_1 + x_2 - (x_1 + x_2 + \epsilon_1 + \epsilon_2)|}{|x_1 + x_2|} \Big/ \frac{|\epsilon_1 + \epsilon_2|}{|x_1| + |x_2|} \\
&= \frac{|\epsilon_1 + \epsilon_2|}{|x_1 + x_2|} \cdot \frac{|x_1| + |x_2|}{|\epsilon_1 + \epsilon_2|} = \frac{|x_1| + |x_2|}{|x_1 + x_2|}
\end{aligned}$$

Dieser Quotient bestätigt obige Bemerkungen, dass Additionen, bei denen gleich große Zahlen von einander subtrahiert werden, problematisch sind. Denn dann erhält man einen Nenner nahe bei 0, gleichzeitig aber einen großen Wert für den Zähler.

A. 14.17: Versuchen Sie problematische Fälle bei der Addition empirisch zu belegen.

A. 14.18: Welche Empfehlungen ergeben sich aus den Bemerkungen für die Berechnung sehr genauer Mittelwerte / sonstiger Momente?

Lineare Abbildungen Für lineare Abbildungen $f(\mathbf{x}) = \mathbf{Ax}$ gilt:

$$\frac{||Ax - A\tilde{x}||}{||Ax||} \Big/ \frac{||\epsilon||}{||x||} = \frac{||A\epsilon||}{||Ax||} \Big/ \frac{||\epsilon||}{||x||} \leq \frac{\max_{||\hat{x}||=1} ||A\hat{x}||}{\min_{||\hat{x}||=1} ||A\hat{x}||}$$

Dieses lässt sich grob so interpretieren, dass die Kondition durch das Verhältnis größte Streckung durch A dividiert durch kleinste Streckung von A bestimmt wird. Betrachtet man zum Beispiel eine multivariate Normalverteilung, deren Varianz in Richtung ihrer ersten Hauptkomponente sehr viel größer ist als in Richtung ihrer letzten Hauptkomonente, dann hat die Kovarianzmatrix genau solche Eigenschaften. Damit ist die Betrachtung von Eigenvektorverhältnissen $\kappa = |\lambda_{max}/\lambda_{min}|$ bei symmetrischen Matrizen plausibel. Vgl. auch: http://en.wikipedia.org/wiki/Condition_number.

Mit diesen Hintergrundinformationen können wir die Fehlermeldung

```
System ist fuer den Rechner singulaer: reziproke Konditionszahl = 5.59894e-19
```

einordnen: Der Kehrwert der berechneten Konditionszahl ist fast 0, so dass die Konditionszahl selbst einen sehr hohen Wert besitzt. Die Matrix ist *fast* singulär, $\mathbf{X}$ enthält also fast kollineare Spalten und die Invertierung wird nicht mehr durchgeführt. Im vorliegendem Fall haben wir mit unserer Transformationsidee das Problem so transformiert, dass sich die Kondition der modifizierten Aufgabenstellung verbessert hat. Solche Strategien sind leider nicht immer zu finden.

Übrigens können wir mit der Funktion `rcond()` oder `kappa()` die Kondition einer Matrix feststellen. In der Hilfe lesen wir hierzu:

```
'kappa()' computes by default (an estimate of) the 2-norm
 condition number of a matrix or of the R matrix of a QR
 decomposition, perhaps of a linear fit.  The 2-norm condition
 number can be shown to be the ratio of the largest to the smallest
 _non-zero_ singular value of the matrix.
 'rcond()' computes an approximation of the *r*eciprocal
 *cond*ition number, see the details.
```

Wir sollten abschließend schauen, ob sich die Kondition unseres Beispielproblems durch die Transformation verändert. Oben hatten wir die Matrizen `X`, `X.stern` und `X.stern.1` konstruiert. Für diese ist die reziproke Kondition schnell ermittelt. Zur Erinnerung je kleiner dieser reziproke Wert ist, umso schlechter ist das Problem konditioniert.

160 ⟨ * 1⟩+ ≡
```
⟨check Givens Rotationen für Problemdaten 156⟩
cat("rcond(X) =",rcond(X),"\n")
cat("rcond(X.stern) =",rcond(X.stern),"\n")
cat("rcond(X.stern.1) =",rcond(X.stern.1))
```

Wie wir sehen, hat sich die Kondition nicht verändert:

```
rcond(X) = 6.437534e-12
rcond(X.stern) = 6.437534e-12
rcond(X.stern.1) = 6.437534e-12
```

Wir hatten nur einen verbesserten Algorithmus gefunden, um mit dem Problem umzugehen. Kleine Änderungen der Parameter unserer Problemstellung werden weiterhin zu erheblichen Ergebnisänderungen führen.

A. 14.19: Probieren Sie die Auswirkung von Änderungen der Input-Parameter aus.

A. 14.20: Berechnen Sie die Kondition einer empirischen Kovarianz-Matrix. Führen Sie dann Umskalierungen durch und berechnen Sie erneut die Kondition.

Mit diesen Bemerkungen zur Kondition ist die Kondition des Autors am Ende, und er beendet die Ausführungen mit dem Regressionsproblem vom Beginn der Vorlesung.

15 Nicht behandelt

In diesem Durchgang wurden nicht behandelt:

- Literate Programming im Detail
- Objektorientierte Sicht
- Abstrakte Datentypen
- Parsen von Ausdrücken

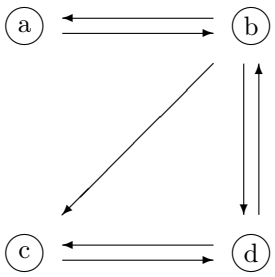
16 Eine alte Klausur

Aufbau der Klausur: Die Klausur besteht aus vier Aufgaben, von denen drei zu bearbeiten sind.

1. Aufgabe → Graphen

a) Was versteht man unter einer Adjazeten- oder Adjazenz-Matrix? Wofür kann man diese verwenden?

b) Erstellen Sie zu folgendem Graphen die Adjazeten-Matrix!



c) Wie erkennt man an der Adjazeten-Matrix eines gerichteten Graphen Quellen und Senken?

d) Wie lässt sich mit Hilfe einer Adjazeten-Matrix eines gerichteten Graphen ermitteln, ob man in zwei Schritten von einem Startpunkt zu einem speziellen Zielpunkt gelangen kann?

2. Aufgabe → Sortieren

a) Was versteht man unter *Sortieren durch Einfügen*?

b) Was versteht man unter *Sortieren durch Mischen*?

c) Stellen Sie sich vor, Sie verwalten die Mitglieder eines Vereins in einer sortierten Liste und müssen nun die neu aufgenommenen Mitglieder einarbeiten. Wie würden Sie vorgehen? – Natürlich müssen Sie auch eine Begründung angeben.

d) Es heißt: *Sortieren gehört in die Klasse* $O(n \log n)$ – was bedeutet das?

3. Aufgabe → Literate Programming

a) Was sind die Kernideen des Literate Programming?

b) Welches sind die elementaren Syntax-Elemente des Literate Programming, die also für die Formulierung einer *Literate-Programming-Lösung* notwendig sind?

c) Die im Folgenden definierte Funktion erlaubt, den Korrelationskoeffizienten

$$r = \frac{\frac{1}{n-1}\sum_i (x_i - \overline{x}) \cdot (y_i - \overline{y})}{\sqrt{\frac{1}{n-1}\sum_i (x_i - \overline{x})^2} \cdot \sqrt{\frac{1}{n-1}\sum_i (y_i - \overline{y})^2}}$$

der Vektoren x und y zu berechnen:

```
corr.coef<-function(x,y){
  m <- length(x); n <- length(y)
  if(n!=m) return("Error: Datenvektoren unterschiedlich lang!")
  mean.x <- 0;  mean.y <- 0;  s.xx <- 0;  s.yy <- 0;   s.xy <- 0
  for(i in 1:n){
    mean.x <- mean.x + x[i]; mean.y <- mean.y + y[i]
  }
  mean.x <- mean.x/n;   mean.y <- mean.y/n
  for(i in 1:n){
    s.xx <- s.xx+(x[i]-mean.x)*(x[i]-mean.x)
    s.yy <- s.yy+(y[i]-mean.y)*(y[i]-mean.y)
    s.xy <- s.xy+(x[i]-mean.x)*(y[i]-mean.y)
  }
  s.xx <- 1/(n-1)*s.xx; s.yy <- 1/(n-1)*s.yy; s.xy <- 1/(n-1)*s.xy
  corr.coefficient <- s.xy/(s.xx^0.5*s.yy^0.5)
  return(corr.coefficient)
}
```

Zwar ist die Funktion mittels R-Syntax formuliert, doch sind durch den Gebrauch allgemein bekannter Sprachkonstrukte die wesentlichen Details schnell zu erkennen. Ihre Aufgabe besteht darin, ausgehend von diesem Vorschlag eine *literate* Lösung zu skizzieren. Hier geht es primär nicht darum, den Code zu verbessern, sondern ihn in eine besser lesbare Form zu bringen. Falls Ihnen bei der Arbeit Verbesserungsvorschläge einfallen sollten, können Sie diese natürlich in Ihre Lösung integrieren.

4. Aufgabe → Suchen

Hash-Algorithmen bieten elegante Möglichkeiten, um Objekte zu speichern.

a) Welche Idee liegt diesen Algorithmen zugrunde?

b) Welche Eigenschaften sollten Hash-Funktionen haben?

c) Was macht man bei Kollisionen?

d) Für welche Probleme / Situationen sind Hash-Funktionen empfehlenswert?

A Anhang

A.1 Index der Objekte

Object Index

A.2 Index der Code-Chunks

Code Chunk Index

⟨*bewege oberstes Element der* `deque` *nach* `state:` `state:=pop` 103⟩ ⊂ 95 ..... p70
⟨*checke Funktionsinputs* 114⟩ ⊂ 110 ..... p76
⟨*check Givens Rotationen für Problemdaten* 156⟩ ⊂ 157, 160 ..... p106
⟨*check Rotationsansatz* 153⟩ p105
⟨*define functions for Householder Rotations* 135⟩ ⊂ 136 ..... p95
⟨*define* `compute.Householder.transformations()` 139⟩ ⊂ 140, 142 ..... p98
⟨*define* `IS.NIL` 92⟩ ⊂ 80 ..... p63
⟨*definiere Daten zum Problembeispiel:* `x, y, k, X` 141⟩ ⊂ 144 ..... p99
⟨*definiere eine Funktion, die einen Turm von* `from` *nach* `to` *bewegt* 20⟩ ⊂ 21 ..p17
⟨*definiere Funktion melde* 96⟩ ⊂ 95 ..... p69
⟨*definiere* `bisection()` 104⟩ p72
⟨*definiere* `dfs` 72⟩ ⊂ 71 ..... p47
⟨*definiere* `DGS` 71⟩ ⊂ 73, 74 ..... p47
⟨*definiere* `find.rotate.matrix()` 150⟩ ⊂ 151, 152 ..... p104
⟨*definiere* `givens()` 152⟩ ⊂ 153, 155, 156 ..... p105
⟨*definiere* `goldenSectionSearch()` 27⟩ ⊂ 30 ..... p21
⟨*definiere* `haenge.teilbaum.ein` 93⟩ ⊂ 81 ..... p63
⟨*definiere* `I.Riemann()` 127⟩ p83
⟨*definiere* `I.Simpson()` 129⟩ p85
⟨*definiere* `I.Trapez()` 128⟩ p84
⟨*definiere* `I.trapeziodal()` 130⟩ p86
⟨*definiere* `newton()` 109⟩ p74
⟨*definiere* `regula.falsi()` 122⟩ p79
⟨*definiere* `sekante()` 117⟩ p77
⟨*definiere* `solve_beta_diag()` 154⟩ ⊂ 155, 156 ..... p106
⟨*drehe x, dass schrittweise die hinteren Koordinaten verschwinden* 149⟩ p104
⟨*drehe x so, dass letzte Dimension wegfällt* 148⟩ p103
⟨*ermittle Transformation von* $\mathbf{X}$ *und* $\mathbf{y}$ 142⟩ ⊂ 143 ..... p99
⟨*ermittle zu* $\mathbf{X}$ *und* $\mathbf{y}$ *Koeffizientenvektor* `beta.hh.dach` 143⟩ ⊂ 144 ..... p99
⟨*goldenSectionSearch: zeige Zustand* 29⟩ ⊂ 27 ..... p22
⟨*handle job ab* 65⟩ ⊂ 64 ..... p43
⟨*initialisiere bisection* 105⟩ ⊂ 104 ..... p73
⟨*initialisiere die Graphik* 22⟩ ⊂ 21 ..... p18
⟨*initialisiere die Türme* 18⟩ ⊂ 21 ..... p16
⟨*initialisiere newton* 110⟩ ⊂ 109 ..... p75
⟨*initialisiere regula falsi* 123⟩ ⊂ 122 ..... p80
⟨*initialisiere sekante* 118⟩ ⊂ 117 ..... p78
⟨*initialisiere* `deque:` `dequeinit` 97⟩ ⊂ 95 ..... p69
⟨*ist* `deque` *leer* 102⟩ ⊂ 95 ..... p70
⟨*konstruiere Ausgabe: bisection* 108⟩ ⊂ 104 ..... p73
⟨*konstruiere Ausgabe: newton* 113⟩ ⊂ 109 ..... p75
⟨*konstruiere Ausgabe: regula.falsi* 126⟩ ⊂ 122 ..... p81
⟨*konstruiere Ausgabe: sekante* 121⟩ ⊂ 117 ..... p79
⟨*konvertiere* `B` *in einen Bitvektor* 10⟩ ⊂ 9 ..... p13
⟨*lege* `n1` *oben auf* `deque` *ab:* `push(n1)` 100⟩ ⊂ 95 ..... p70
⟨*lege* `n2` *oben auf* `deque` *ab:* `push(n2)` 101⟩ ⊂ 95 ..... p70
⟨*lege* `next1[state+IO]` *unten auf* `deque` *ab:* `put(SCAN)` 99⟩ ⊂ 95 ..... p69
⟨*lege* `SCAN` *unten auf* `deque` *ab:* `put(SCAN)` 98⟩ ⊂ 95 ..... p69
⟨*protokolliere Iterationmitte: newton* 111⟩ ⊂ 109 ..... p75
⟨*protokolliere Iterationsbeginn: bisection* 106⟩ ⊂ 104 ..... p73

⟨*protokolliere Iterationsende: bisection* 107⟩ ⊂ 104 ..... p73
⟨*protokolliere Iterationsende: newton* 112⟩ ⊂ 109 ..... p75
⟨*protokolliere Iterationsende: regula.falsi* 125⟩ ⊂ 122 ..... p81
⟨*protokolliere Iterationsende: sekante* 120⟩ ⊂ 117 ..... p79
⟨*protokolliere Iterationsmitte: regula.falsi* 124⟩ ⊂ 122 ..... p81
⟨*protokolliere Iterationsmitte: sekante* 119⟩ ⊂ 117 ..... p78
⟨*start* 31 ∪ 32 ∪ 36 ∪ 41 ∪ 42 ∪ 43 ∪ 44 ∪ 45 ∪ 46 ∪ 47 ∪ 48 ∪ 49 ∪ 50 ∪ 51 ∪ 52 ∪ 53 ∪ 54 ∪ 61 ∪ 64 ∪ 66 ∪ 67 ∪ 76 ∪ 77 ∪ 78 ∪ 79 ∪ 80 ∪ 81 ∪ 85 ∪ 87 ∪ 89 ∪ 91 ∪ 94 ∪ 161 ∪ 162 ∪ 164 ∪ 166⟩ p26
⟨*starte Demonstration* 21⟩ p17
⟨*teste Suche* 30⟩ p22
⟨*teste* `DGS` 73⟩ p48
⟨*transformiere* `B` *in die Zahl* `B.zehn` *des Zehnersystems* 11⟩ ⊂ 9 ..... p13
⟨*transformiere* `B.zehn` *in Bitvektor* `BB` 12⟩ ⊂ 9 ..... p13
⟨`a1`*: bestimme Index k des kleinsten Elementes der* $a_i, \ldots, a_n$ 33⟩ ⊂ 32 ..... p27
⟨`a1`*: vertausche* a_i *und* a_k 34⟩ ⊂ 32 ..... p27
⟨`a2`*: bestimme den Index k im sortierten Teilvektor* $a_1, \ldots, a_i$ 38⟩ ⊂ 37 ..... p29
⟨`a2`*: plaziere* a_i *geeignet in den sortierten Teilvektor ein* 37⟩ ⊂ 36 ..... p29
⟨`a2`*: verschiebe* $a_k, \ldots, a_i$ *um einen Platz und füge* a_i *ein* 39⟩ ⊂ 37 ..... p29
⟨*unused* 115⟩ p76
⟨*zeige Türme* 23⟩ ⊂ 20, 21 ..... p18

A.3 Quellen

`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/files/daganz.pdf`

`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/files/genetic-algos.pdf`

`http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/pw_files/files/tsp.pdf`

Büning, H., Trenkler, G. (1979): Nichtparametrische statistische Methoden, deGruyter.

Cormen, Th., Leiserson, C.E., Rivest, R.L. (1992): Introduction to Algorithms, McGraw-Hill.

Enzensberger, H.M. (1999): Der Zahlenteufel, DTV.

Levitin, A. (2003/2007): The design & Analysis of Algorithms, Addison-Wesley.

Naeve, P. (1995): Stochastik für Informatik, Oldenbourg.

Pomberger, G. und Dobler, H. (2008): Algorithmen und Datenstrukturen, Pearson Studium.

Sedgewick, R. (2002): Algorithmen, Pearson-Studium / Addison-Wesley.

Thisted, R.A. (1988): Elements of statistical computing, Chapman and Hall.

Wirth, N. (z.B. 1998): Algorithmen und Datenstrukturen, Teubner.