Informatik 2 mit BlueJ

Ein Kurs für die Stufe 12

von Ulrich Helmich

Lösungen und Lösungs-hinweise

1. Auflage Mai 2009

Folge 12 - Suchalgorithmen

12.1 Lineare und binäre Suche

Übung 12.1 (2 Punkte)

Berechnen Sie die durchschnittliche Suchzeit **S** für ganze Zahlen eines Arrays der Länge **N**, wenn **W** die Wahrscheinlichkeit dafür ist, dass die gesuchte Zahl im Array vorkommt ($0 < W <= 1$).

Im Lehrtext wurde bereits festgestellt, dass die Suchzeit N/2 beträgt, wenn alle gesuchten Zahlen in dem Array vorkommen. Ist eine Suchzahl nicht im Array vorhanden, so muss der gesamte Array durchsucht werden, wenn er unsortiert ist; die Suchzeit beträgt hier also N.

Ist nun W die Wahrscheinlichkeit, dass eine Suchzahl im Array vorkommt, so berechnet sich die Suchzeit für die vorkommenden Zahlen mit W * N/2. Da die Wahrscheinlichkeit, dass eine Suchzahl nicht im Array enthalten ist, genau 1-W beträgt, ist die Suchzeit für diese Zahlen (1-W) * N. Die durchschnittliche Suchzeit setzt sich aus diesen beiden Beträgen zusammen, und so kommen wir auf

S = W * N/2 + (1-W) * N.

Beispiele:

Genau die Hälfte der gesuchten Zahlen ist im Array enthalten; W = 0,5. Dann erhalten wir

S = 0,5 * N/2 + 0,5 * N = 0,75 N.

Sind 80% der gesuchten Zahlen im Array enthalten, so ergibt sich

S = 0,8 * N/2 + 0,2 * N = 0,60 N.

Übung 12.2 (12 Punkte)

Erstellen Sie eine Klasse **Zahlen** mit einem Attribut **liste**; dieses Attribut soll ein Array aus 100 zufälligen int-Zahlen im Zahlenbereich zwischen 1 und 1000 sein. Jede Zufallszahl soll höchstens einmal im Array vorkommen (2 Punkte).

Statten Sie die Klasse mit Methoden zum Erzeugen und Anzeigen der Zahlen aus - achten Sie darauf, dass die Zahlen in ansprechender Weise angezeigt werden, nicht einfach alle untereinander oder nebeneinander, sondern zum Beispiel immer 16 oder 20 Zahlen in einer Reihe (2 Punkte).

Schreiben Sie dann eine Methode

```
public int sucheLinear(int suchzahl)
```

die einen linearen Suchalgorithmus realisiert; es soll nach der Zahl `suchzahl` gesucht werden. Wenn `suchzahl` gefunden wurde, gibt die Methode den Index der Fundstelle zurück, andernfalls -1 (2 Punkte).

Entwickeln Sie dann eine Methode, die **sucheLinear()** systematisch testet und für 10 zufällig gewählte Suchzahlen die durchschnittliche Suchzeit ermittelt. Von diesen 10 Zahlen sollen 7 in dem Array vorkommen, 3 jedoch nicht (5 Punkte).

Vergleichen Sie die so experimentell ermittelte durchschnittliche Suchzeit mit dem Ergebnis, das ihre Formel aus Übung 12.1 vorhersagen würde (1 Punkt).

Die erste Teilaufgabe ist die Entwicklung einer Klasse Zahlen mit dem Attribut liste, das 100 zufällige und verschiedene int-Zahlen enthält. Wie kann man eine solche Liste generieren? Natürlich könnte man so verfahren, dass man Zahl liste[n] erzeugt und dann alle Zahlen liste[0] bis liste[n-1] mit der neu erzeugten Zahl vergleicht. Ist eine dieser Zahlen mit liste[n] identisch, wird eine neue Zufallszahl erzeugt. Dieser Vorgang wird so lange wiederholt, bis liste[n] eine Zahl enthält, die noch nicht in der Liste vorhanden ist.

Dieses Vorgehen kostet allerdings sehr viel Rechenzeit und ist auch recht aufwändig zu programmieren. Es geht viel einfacher:

Zunächst wird eine sortierte Liste erzeugt, die die Zahlen von 1 bis 100 enthält:

```
for (int i=0; i<100; i++) liste[i] = i;
```

Nun soll die Liste aber Zahlen im Bereich zwischen 1 und 1.000 enthalten und nicht Zahlen zwischen 0 und 99. Kein Problem, wir multiplizieren einfach jede Zahl mit 10:

```
for (int i=0; i<100; i++) liste[i] = i*10;
```

Die Liste enthält jetzt die Zahlen

```
0, 10, 20, 30, 40, ... , 980, 990.
```

Das sieht noch nicht sehr zufällig aus. Also addieren wir zu jeder dieser Zahlen eine Zufallszahl zwischen 0 und 9:

```
for (int i=0; i<100; i++)

   liste[i] = i*10 + zufall.nextInt(10);
```

Dabei ist **zufall** ein Objekt der Klasse **Random**, das wir natürlich vorher deklarieren und initialisieren müssen.

Diese Liste ist aber noch geordnet, während eine Zufallsliste völlig ungeordnet ist. Schreiben wir doch einfach eine Tauschroutine, wie wir sie bereits von den Sortieralgorithmen her kennen:

```
private void tauschen(int i, int j)
{
   int temp = liste[i];
   liste[i] = liste[j];
   liste[j] = temp;
}
```

Wenn wir diese Tauschroutine jetzt 100 mal auf unseren Array anwenden, sollte dieser völlig „unsortiert" und „zufällig" aussehen:

```
for (int i=0; i<100; i++)
   tauschen(zufall.nextInt(100),zufall.nextInt(100));
```

Damit wäre die erste Teilaufgabe schon einmal gelöst. Das Ausgeben dieser Zahlen sollte nun kein Problem sein:

```
public void ausgeben ()
{
   for (int i=0; i<100; i++)
      if (i % 16 == 0)
         System.out.println(liste[i]);
      else
         System.out.print(liste[i] + "\t");
}
```

Wenn i nicht durch 16 teilbar ist, wird das Arrayelement hinten an die Zeile angehängt (print-Befehl), und wenn i durch 16 teilbar ist (16, 32, 64, 96), so wird ein Zeilenvorschub erzeugt.

Die dritte Teilaufgabe verlangt, dass wir einen Suchalgorithmus schreiben, der die Zahlen linear durchsucht und den Index der Fundstelle zurückgibt. Auch das ist recht einfach:

```
public int sucheLinear(int suchzahl)
{
   for (int i=0; i<100; i++)
      if (liste[i] == suchzahl)
         return i;
   return -1;
}
```

Sobald die Suchzahl gefunden wurde, wird die sondierende Methode mit dem Index der gefundenen Zahl als Rückgabewert verlassen. Sollte die Zahl nicht im Array enthalten sein, wird die for-Schleife komplett durchlaufen. Anschließend wird die Methode mit return -1 verlassen.

Eine alternative Version dieser Methode könnte so aussehen:

```
public int sucheLinear(int suchzahl)
{
   int schritte = 1;

   for (int i=0; i<100; i++)
      if (liste[i] == suchzahl)
         break;
      else
         schritte++;
   return schritte;
}
```

Eine Methode, die **sucheLinear()** systematisch testet, könnte zunächst so aussehen:

```
public void testeSuche()
{
   int suchzahl;
   double summe=0.0;

   for (int i=1; i <= 10; i++)
   {
      suchzahl = zufall.nextInt(1000)+1;
      summe = summe + sucheLinear(suchzahl);
   }
   System.out.println("Zahl der Vergleiche
      fuer 10 Zahlen = "+summe/10.0);
}
```

Allerdings entspricht diese Methode nicht ganz der Aufgabenstellung, denn es wird ausdrücklich verlangt, dass sieben der 10 Suchzahlen in dem Array vorkommen, drei dagegen nicht. Also muss die Methode stark verändert werden:

```
public void testeSuche()
{
   int suchzahl, vergleiche;
   double summe=0.0;
   int gefunden = 0, nichtgefunden = 0;

   while (gefunden < 7)
   {
      suchzahl = zufall.nextInt(1000)+1;
      vergleiche = sucheLinear(suchzahl);
      if (vergleiche < 100)
      {
         summe = summe + vergleiche;
         gefunden++;
         System.out.println("Suchzahl = "+suchzahl+" gefunden
         nach "+vergleiche+" Schritten");
      }
   }
   while (nichtGefunden < 3)
   {
      suchzahl = zufall.nextInt(1000)+1;
      vergleiche = sucheLinear(suchzahl);
      if (vergleiche == 100)
      {
         summe = summe + vergleiche;
         nichtGefunden++;
         System.out.println("Suchzahl = "+suchzahl+"
         nicht gefunden nach "+vergleiche+" Schritten");
      }
   }

   System.out.println("Durchschnittliche Zahl der Vergleiche
   fuer 10 Zahlen = "+summe/10.0);
}
```

Das ist doch schon recht aufwändig, womit auch die 5 Punkte gerechtfertigt sind, die es für diese Teilaufgabe gibt.

Übung 12.3 (4 Punkte)

Der Array soll Zahlen zwischen 1 und 1000 enthalten und sortiert sein (einen Sortieralgorithmus haben Sie schnell eingebaut). Schreiben Sie nun eine Such-Methode, die folgende Optimierung enthält: Bei Suchzahlen kleiner als 500 beginnt die Suche links im Array, bei Suchzahlen ab 500 dagegen rechts - in diesem Fall wird der Array also von hinten ausgehend durchsucht.

```
public int sucheLinear(int suchzahl)
{
   int schritte = 1;

   if (suchzahl < 500)
   {
      for (int i=0; i<100; i++)
         if (liste[i] >= suchzahl)
            break;
         else
            schritte++;
   }
   else
   {
      for (int i=99; i>=0; i--)
         if (liste[i] <= suchzahl)
            break;
         else
            schritte++;
   }
   return schritte;
}
```

Diese Methode ist aus der alternativen Version der entsprechenden Methode aus Übung 12.2 hervorgegangen.

Messungen für 100 Zahlen, von denen 10% nicht im Array enthalten waren, ergaben eine durchschnittliche Suchzeit von ca. 25 Vergleichen - im Gegensatz zu ca. 50 Vergleichen für das konventionelle lineare Suchen in einem unsortierten Array.

Übung 12.4 (6 Punkte)

Das in 12.3 besprochene Verfahren kann man noch weiter optimieren. Bei 12.3 hatten wir *zwei* Startpositionen für die Suche, nämlich das Element mit dem Index 0 (ganz vorne) und das Element mit dem Index N-1 (ganz hinten), wobei N die Anzahl der Arrayelemente ist. Statt dieser zwei Startpositionen könnte man aber auch 10, 25 oder noch mehr Startpositionen in den Array einbauen und die Suche so enorm beschleunigen. Implementieren Sie eine derart optimierte Suche.

Die folgende Version, die auf den ersten Blick sehr plausibel aussieht und mit Sicherheit von einigen Schüler(innen) implementiert wird, arbeitet noch nicht korrekt:

```
public int sucheLinear(int suchzahl)
 {
    int schritte = 1;
    int startposition;

    if (suchzahl < 100) startposition =  0; else
    if (suchzahl < 200) startposition = 10; else
    if (suchzahl < 300) startposition = 20; else
    if (suchzahl < 400) startposition = 30; else
    if (suchzahl < 500) startposition = 40; else
    if (suchzahl < 600) startposition = 50; else
    if (suchzahl < 700) startposition = 60; else
    if (suchzahl < 800) startposition = 70; else
    if (suchzahl < 900) startposition = 80; else
                        startposition = 90;
```

```
        for (int i=startposition; i<100; i++)
              if (liste[i] >= suchzahl)
                 break;
              else
                 schritte++;

        return schritte;
     }
```

Die Auswahl der Startposition muss noch optimiert werden, hier ist sie nicht nur unnötig umständlich, sondern auch fehlerhaft. Angenommen, die Zahl 450 befindet sich an der Position 37 im Array. Der obige Algorithmus würde aber eine Startposition von 40 liefern. Die Suche nach der 450 würde rechts von der Suchzahl beginnen, und natürlich würde der Algorithmus die Zahl dann nicht in dem Array vorfinden. Er würde also den gesamten Array bis zum Ende durchsuchen und käme dann auf eine Schrittzahl von 61.

Nun weiß der Algorithmus aber nicht, wie die Elemente des Arrays beim späteren Programmdurchlauf verteilt sind. Es können daher keine festen Startpositionen festgelegt werden.

Wie wäre es mit folgendem Vorgehen: In einem ersten Schritt wird der gesamte Array durchsucht, und die Zahlen an den Positionen 10, 20, 30 und so weiter werden in einem Hilfsarray gespeichert.

Dazu müssen wir die Methode **erzeugen()** ergänzen:

```
  public void erzeugen()
  {
     for (int i=0; i<100; i++)
        liste[i] = i*10 + zufall.nextInt(10);

     for (int i=0; i<9; i++)
        hilfsarray[i] = liste[(i+1)*10];
  }
```

Und können dann die Methode **sucheLinear()** neu schreiben:

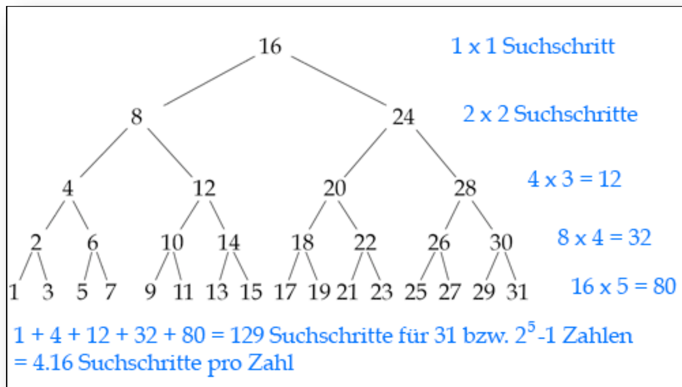
```
public int sucheLinear(int suchzahl)
{
   int schritte = 1;
   int startposition = 0;

   for (int i=0; i<9; i++)
      if (suchzahl < hilfsarray[i])
      {
         startposition = i*10;
         break;
      }

   for (int i=startposition; i<100; i++)
      if (liste[i] >= suchzahl)
          break;
      else
         schritte++;

      return schritte;
}
```

Auf diese Weise können wir - mit 9 Startpositionen - die Suchzeit noch einmal auf die Hälfte verkürzen.

Übung 12.5 (1 Punkt)

a) Wo müsste die Zahl 32 in dem Binärbaum in Abbildung 12-3 eingefügt werden?

b) Wie groß wäre die durchschnittliche Anzahl der Suchschritte pro Zahl bei dem Binärbaum mit 32 Zahlen ?



***12-3** Ein binärer Suchbaum aus 31 Zahlen*

Die Zahl 32 muss als rechter Nachfolger an die Zahl 31 angehängt werden, der Baum erhält dadurch eine zusätzliche sechste Ebene.

Die ersten 31 Zahlen können mit zusammen 129 Suchschritten gefunden werden. Zum Finden der Zahl 32 sind 6 Suchschritte erforderlich. Dadurch erhalten wir eine durchschnittliche Suchzeit von 135/32 = 4,22 Suchschritte pro Zahl.

Übung 12.6 (6 Punkte)

Ergänzen Sie die Klasse **Zahlen** um ein *rekursives* binäres Suchverfahren.

Testen Sie dann das binäre Suchverfahren mehrmals mit verschiedenen im Array enthaltenen und auch nicht enthaltenen Suchzahlen,

- ob es überhaupt funktioniert und
- welche Suchzeiten zum Finden der Suchzahlen benötigt werden.

Das Programm aus der letzten Übung kann weitgehend übernommen werden. Hier die Methode sucheBinaer():

```
public int sucheBinaer(int suchzahl)
{
  ebene = 0;
  int position = sucheBinaerRekursiv(suchzahl,0,99);

  if (position == -1)
     return ebene+1;
  else
     return ebene;
}
```

Die Methode sieht sehr kurz aus, was natürlich daran liegt, dass die Hauptarbeit von der rekursiven Methode **sucheBinaerRekursiv()** geleistet wird, die von **sucheBinaer()** aufgerufen wird. Das Attribut **ebene** muss in dem Deklarationsteil der Klasse als globales Attribut vereinbart werden.

Hier nun die rekursive Methode:

```
private int sucheBinaerRekursiv
(int suchzahl, int links, int rechts)
{
   ebene++;

   if (links > rechts)
      return -1;
   else
   {
      int mitte = (links+rechts)/2;
      if (suchzahl < liste[mitte])
         return sucheBinaerRekursiv
               (suchzahl,links,mitte-1);
      else if (suchzahl > liste[mitte])
         return sucheBinaerRekursiv
                (suchzahl,mitte+1,rechts);
      else
         return mitte;
   }
}
```

Zunächst wird die Mitte des zu untersuchenden Array-Intervalls bestimmt, beim ersten Aufruf der Methode ist dies der Wert (0+99)/2 = 99/2 = 49.

Nun wird getestet, ob die Suchzahl kleiner ist als das Arrayelement in der Mitte. Wenn ja, wird die Methode rekursiv für den Abschnitt links der Mitte aufgerufen:

```
return sucheBinaerRekursiv(suchzahl,links,mitte-1);
```

Das Rückgabeergebnis dieses Aufrufs ist die Zahl der Ebenen, die für die Suche gebraucht werden. Beim „rekursiven Abstieg" durch Aufrufen der eigenen Methode wird die Zahl der benötigten Ebenen um Eins erhöht (ebene++).

Ist die Suchzahl jedoch nicht kleiner als die mittlere Zahl des Intervalls, so wird gefragt, ob die Suchzahl größer ist. Wenn dies der Fall ist, wird entsprechend der Intervall rechts der mittleren Zahl untersucht.

Ist die Suchzahl weder kleiner noch größer als die mittlere Zahl, ist sie mit der mittleren Zahl identisch, und die Suche ist mit Erfolg beendet. Der Index der gefundenen Zahl wird zurück gegeben.

Warum wird nicht die Zahl der benötigten Rekursionsebenen zurückgegeben? Beim Suchen nach einer Zahl oder einem Element interessiert eigentlich nur, wo in dem Array das Element steht, also der Index bzw. die Position. Die Anzahl der Vergleiche oder benötigten Rekursionsebenen ist eigentlich nur von akademischem Interesse. Für unser Testprogramm benötigen wir aber gerade die Zahl der Vergleiche, daher wird hier neben der Position auch die Anzahl der Rekursionsebenen zurückgegeben. Technisch geschieht dies über eine globale Variable, nämlich ebene, da eine sondierende Methode nicht gleichzeitig zwei Werte zurückgeben kann.