Sortieren durch Einfügen

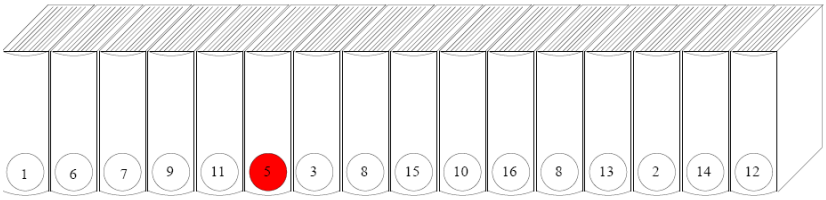
Ein Algorithmus zum Informatikjahr 2006

Prof. Dr. W. Kowalk, Universität Oldenburg, kowalk@kowalk.de

Schon wieder Aufräumen, dabei habe ich doch erst neulich... Nun ja, wenn alles durcheinander liegt, dann lohnt es sich vielleicht doch, mal eben Ordnung zu schaffen; man findet alles schneller wieder. Lass uns also mal die Bücher auf dem Regal nach ihren Titeln sortieren, so dass man jedes gleich zur Hand hat, wenn man es braucht.

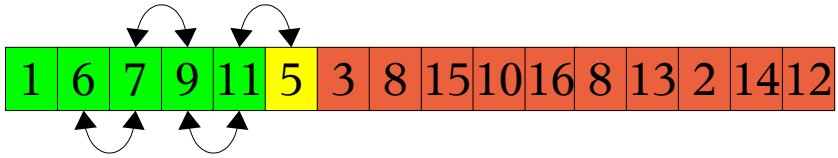
Doch wie schafft man am schnellsten Ordnung? Man kann verschiedene Ansätze verfolgen. So kann man alle Bücher durchgehen, und wenn zwei falsch stehen, vertauscht man sie eben. Das funktioniert, weil irgendwann mal keine zwei Bücher mehr verkehrt stehen, aber das ist offensichtlich nicht sehr effizient. Man kann auch zunächst das erste Buch mit dem 'kleinsten' Titel suchen, dieses nach ganz links ins Regal stellen, dann das zweite daneben usw. Auch das ist eher unnatürlich und auch ineffizient, da es sehr viel vorhandene Information nicht ausnutzt; jedes Mal müssen immer wieder die gleichen Buchtitel gelesen werden. Versuchen wir also mal was anderes.

Die folgende Idee scheint sehr natürlich zu sein. Wir bringen alle Bücher von links nach rechts im Regal fortschreitend an die richtige Position. Das erste Buch steht zunächst am richtigen Platz; dann nehmen wir das zweite Buch hinzu und vertauschen es mit dem ersten, falls es links von diesem stehen muss; dann nehmen wir das dritte Buch hinzu, vertauschen es mit dem zweiten, falls nötig, und vertauschen es anschließend gegebenenfalls mit dem ersten. Dann nehmen wir das vierte Buch hinzu, usw. Allgemein nehmen wir also an, das alle Bücher links im Bücherregal bereits sortiert sind; sodann nehmen wir das nächste hinzu und bringen es durch Vertauschen mit dem linken Nachbarn an die richtige Position.



Statt der Buchtitel schreiben wir im folgenden Nummern, weil der Algorithmus dann einfacher zu erklären ist.

In dem Bild sind die linken fünf Bücher 1, 6, 7, 9, 11 bereits sortiert; wird das Buch mit der Nummer 5 hinzu genommen, so steht es offensichtlich falsch. Also vertauschen wir es zunächst mit dem Buch mit der Nummer 11, dann mit dem mit der Nummer 9 usw. bis es einmal rechts vom Buch mit der Nummer 1 an seiner richtigen Position angekommen. Dann fahren wir mit dem Buch mit der Nummer 3 fort, usw. Offenbar kommen dadurch alle Bücher einmal an ihren richtigen Platz.

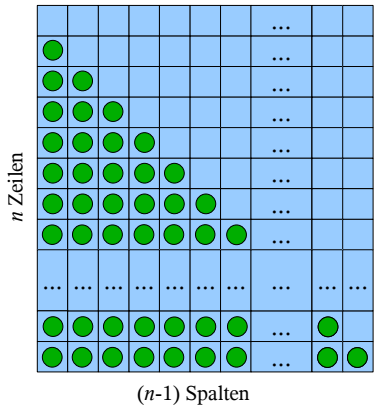


Wie kann man so etwas jetzt programmieren? Das folgende Programm macht das! Es benutzt ein sogenanntes Feld von Zahlen; in der Informatik sagt man dazu *Array*. Die Werte in einem Array werden durch einen Zahlenwert ausgewählt, der auch als *Index* bezeichnet wird. Unter `A[i]` versteht man den Wert an der *i*-ten Stelle, oder den Wert unter dem Index *i*. In der Mathematik würde man statt dessen A_i schreiben, was genau das gleiche bedeutet. Die Werte stehen also in einem Feld der Länge *n* und werden mit `A[0]`, `A[1]`, `A[2]`, ..., `A[n-1]`, `A[n-1]` bezeichnet. In der Programmiersprache Java würde der Algorithmus dann folgendermaßen aussehen.

```
int A[] = { 6, 7, 4, 9, 2, 7,  ..usw. }; // A[] enthält n Elemente
for(int i=1; i<A.length; i++)//nimmt die Bücher an den Stellen i=1 bis n-1 nacheinander hinzu
  for(int j=i; j>=1; j--)     // vergleiche die Bücher an der Stelle j=i, i-1, ... bis 1 ...
                              //  ... jeweils mit den links daneben stehenden an der Stelle j-1
    if(A[j-1]<=A[j]) break;   //  A[j] ist richtig einsortiert, denn die Stelle j-1 befindet
                     //   sich links der Stelle j, und A[j-1]≤A[j], also beende die innere Schleife j
    else { int Hand = A[j];   // sonst vertausche die Bücher an den Stellen j und j-1
           A[j] = A[j - 1];   // in der Hilfsvariablen Hand wird A[j] zwischengespeichert
           A[j - 1] = Hand; }
    // hier wird die innere Schleife j fortgesetzt, solange j>1
    // ist die innere Schleife j beendet, so wird die äußere Schleife i fortgesetzt.
```

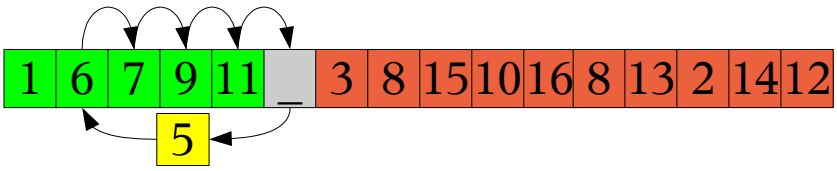
Wie lange dauert jetzt wohl das Aufräumen? Nehmen wir einmal den schlechtesten Fall an; dann sind alle Bücher genau verkehrt herum sortiert, also das mit der kleinsten Nummer steht ganz rechts, daneben das mit der zweitkleinsten Nummer usw. Was macht unser Algorithmus dann? Er fängt von links an und vertauscht das zweite Buch mit dem ersten, das dritte mit den ersten beiden, das vierte mit den ersten dreien, usw., bis schließlich das letzte mit n-1 Büchern zu vertauschen ist. Die Arbeit ist also

$$1+2+3+...+n-1=\frac{n\cdot(n-1)}{2}.$$



Diese Formel lässt sich leicht anhand des nebenstehenden Bildes einsehen. Es gibt in dem Rechteck $n\cdot(n-1)$ Felder, und davon wird gerade die Hälfte für das Vergleichen und Vertauschen verwendet. Allerdings ist unser Algorithmus im Mittel immer schneller als der schlechteste Fall, da bei zufälliger Anordnung der Bücher nur etwa die Hälfte der Vergleichs- und Schiebeoperationen nötig sind.

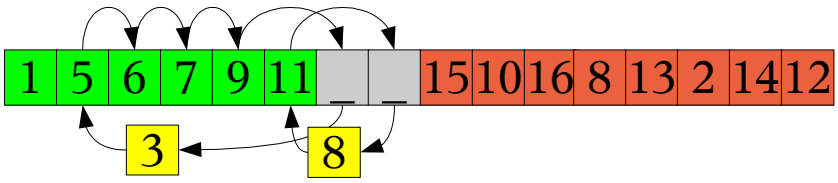
Sicherlich hast du schon bemerkt, dass unser Sortierverfahren sehr umständlich ist, weil wir benachbarte Bücher immer vertauschen. Sobald wir wissen, wo das neue Buch hingehört, schieben wir alle größeren Bücher um eine Stelle nach rechts und stellen das neue Buch an die richtige Stelle. Statt zwei Bücher k-mal zu vertauschen, verschieben wir $(k+2)$-mal ein Buch, was offenbar weniger aufwändig ist, weil einmal Vertauschen drei Verschiebeoperationen benötigt. Das spart eine ganze Menge Arbeit und Zeit! Als Algorithmus in Java sieht das vielleicht so aus:




```
int A[] = { 6, 7, 4, 9, 2, ... usw. }; // A[] enthält n Elemente
for (int i=1; i<A.length; i++) {// nimm die Bücher an den Stellen i=1 bis n-1 hinzu
  int Hand = A[i];              // zunächst nehmen wir das neue Buch in die Hand
  Schleife_J: {                 // hier beginnt ein Block mit Namen Schleife_J
    for (int j=i-1; j>=0; j--)  // vergleiche die Bücher an der Stelle j=i-1, i-2, ... bis 0 ...
                                //  ... jeweils mit dem Buch in der Hand
      if (A[j] <= Hand) {       // an j steht kleineres Buch, also Einfügestelle j+1 gefunden
        A[j+1] = Hand;     // stelle neu hinzugekommenes Buch an die Einfügestelle j+1
        break Schleife_J; // verlasse den Block Schleife_J
      } else A[j+1] = A[j];     // sonst schiebe das Buch um eine Stelle nach rechts
    A[0] = Hand;                // hier wird der Fall behandelt, dass Hand<A[0]
  }// hier endet der Block Schleife_J, der die inneren Schleife j und die letzte Anweisung umfasst
// ist der innere Block Schleife_J beendet, so wird die äußere Schleife i fortgesetzt.
```

Nun könnte man glauben, dass das Verschieben einer "beliebigen" Anzahl von Büchern nach rechts nur einen einzigen Arbeitsschritt erfordert: Wir drücken einfach die ganze Reihe von Büchern um eine Position nach rechts. Im Computer ist das aber nicht so einfach möglich. Hier müssen die Objekte (Bücher) wirklich einzeln nacheinander nach rechts geschoben werden, um eine freie Stelle für das einzufügende Objekt (Buch) zu schaffen.

Gibt es weitere Möglichkeiten Laufzeit zu sparen? Nun, deine Bücher sind sicherlich nicht so schwer, dass du nicht zwei gleichzeitig einsortieren könntest. Nimm also zwei Bücher, verschiebe alle Bücher bis zum Platz des größeren um zwei Positionen nach rechts, stelle das größere rein, schiebe die anderen bis zum Platz des kleineren um eine Position nach rechts und stelle dieses an seinen Platz. Damit hast du einiges gespart, denn du packst jetzt zwei Bücher weg, aber die Arbeit ist nicht viel mehr als wenn du nur eines richtig hinstellen würdest.



Als Java-Programm kann man es folgendermaßen implementieren.

```
int anfang = 1;            // fange mit vergleich bei Buch 1 an.
if( (A.length&1) == 0) {   // falls wir eine gerade Anzahl von Büchern haben
  anfang = 2;              // fange mit vergleich bei Buch 2 an.
  if(A[0]>A[1]) {          // vertausche ggf. die beiden Bücher ganz links
   int buch=A[0]; A[0]=A[1]; A[1]=buch;}
} // if( (A.length&1) == 0)
for (int nächster = anfang; nächster < A.length; nächster+=2) {
  int gross, klein;                  // Buch mit größerem bzw. kleinerem Titel
  if(A[nächster]<=A[nächster+1]) { // setzt die Variablen entsprechend
    klein = A[nächster]; gross= A[nächster+1];
  else {
    gross = A[nächster]; klein= A[nächster+1];
  }
  int mit = nächster-1;                  // vergleiche 'mit' diesem Index
  GROSS: {                               //  zunächst das größere Buch einfügen
    for (; mit >= 0; mit--)              // gehe alle sortierten Bücher durch
      if (A[mit] <= gross) {             // Einfügestelle für größeres Buch gefunden
        A[mit+2] = gross; break GROSS; // größeres Buch einfügen
      } else A[mit+2] = A[mit];          // sonst Buch an Stelle 'mit+2' kopieren
    A[1] = gross;                  // falls A[0]>gross: A[2]=A[0], A[1]=gross;
  }  // GROSS
  //  jetzt steht gross an richtiger Stelle, mit weist auf nächstes Element, füge also kleineres Buch ein
  KLEIN: {
    for (; mit >= 0; mit--)    // gehe alle sortierten Bücher durch
      if (A[mit] <= klein) { // Einfügestelle für kleineres Buch gefunden
        A[mit+1] = klein; break KLEIN; // kleineres Buch einfügen
      } else A[mit+1] = A[mit];// sonst Buch an Stelle 'mit+1' kopieren
      A[0] = klein;           // falls A[0]>gross: A[1]=A[0], A[0]=klein;
  }  // KLEIN
 }   // for nächster
```

Das Programm geht davon aus, immer zwei Bücher gleichzeitig einzusortieren. Daher muss die Anzahl einzufügender Bücher gerade sein; ist die Gesamtzahl an Büchern gerade, so werden die ersten beiden Bücher explizit sortiert; sonst wird das erste Buch zu Anfang alleine genommen. Danach

werden alle Bücher ab dem dritten bzw. zweiten mit dem rechts daneben stehenden zum Vergleich herausgenommen und entsprechend ihrer Größe auf zwei Variablen **gross** und **klein** kopiert.

Die nächsten beiden Schleifen arbeiten genau wie im Falle des Einfügen eines Buches, wobei in der ersten Schleife, in welcher das größere Buch eingefügt wird, jeweils zwei Schritte geschoben werden. Der Index **mit** zeigt jeweils auf das nächste zu vergleichende Buch. Wird die erste Schleife beendet, weil die Einfügestelle gefunden wurde, so zeigt **mit** auf das erste Buch, welches nicht größer als das Buch **gross** ist. Ab diesem wird also mit dem Buch **klein** verglichen. Sollte **gross** kleiner als das erste Buch **A[0]** sein, so erhält **mit** den Wert -1; die zweite innere Schleife wird gar nicht mehr ausgeführt, die Anweisung **A[0]=klein** bringt das kleinere Buch dennoch an die richtige Stelle.

Natürlich kann man auch mehr als zwei Bücher gleichzeitig einsortieren. Aber nimmt man zu viele auf einmal, 'frisst' das den Gewinn wieder auf. Man kann empirisch zeigen (d.h. durch ausprobieren und ausmessen der Laufzeit), dass ein Optimum ungefähr bei der Quadratwurzel aus *N* liegt; also wenn du 100 Bücher hättest wäre das gleichzeitige Einsortieren von 10 Büchern nahezu optimal.

Ein Programm, welches allgemein *k* Bücher gleichzeitig einsortiert könnte folgendermaßen in Java implementiert werden.

```
final int dist = (int)Math.sqrt(A.length);
int [] hand = new int[dist];  // speichere gleichzeitig zu verschiebende Bücher in hand
MAIN:
for (int nächster = 1; nächster < A.length; nächster+=dist) {
  int dist2 = Math.min(dist, A.length-nächster);  // betrachte nur Bücher in A
  hand[0] = A[nächster];                 // mindestens ein Buch ist immer in hand
  for(int i=1; i<dist2; i++) {           // nun sortiere weitere Bücher von A nach hand
    int j = i-1;
    for(;j >= 0;j--)                     // gehe die Bücher in hand[i-1..0] durch
      if(hand[j]<=A[nächster+i]) break;// Einfügestelle gefunden! Beende Schleife j
      else hand[j+1] = hand[j];          // sonst schiebe dieses Buch in hand nach rechts
    hand[j+1] = A[nächster+i];           // schiebe nächstes Buch an Einfügestelle
  } // for i
  int distanz = dist2-1;                 // vergleich A[mit] mit hand[distanz]
  for (int mit = nächster-1; mit >= 0; mit--)// Bücher in A[nächster-1..0]
    if (A[mit] <= hand[distanz]) {       // Einfügestelle gefunden
      A[mit+distanz+1] = hand[distanz];// schiebe hand[distanz] nach A
      if(distanz == 0) continue MAIN;  // höre auf, wenn letzte Buch aus hand sortiert
      mit++;                             // vergleich noch einmal mit letztem Buch aus A
    } else A[mit+distanz+1] = A[mit];  // sonst verschiebe Buch in A um distanz+1
  int i = distanz;
  while(i>=0) A[i] = hand[i--];          // bringe restliche Bücher aus hand nach A
} // for nächster
```

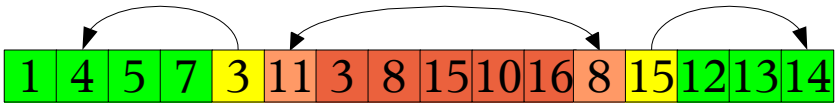
Das Programm verwendet ein Feld '**hand**', in welchem die Bücher, die gleichzeitig sortiert werden sollen, gehalten werden. Zunächst werden diese Bücher in der 'Hand' sortiert, wozu ebenfalls sortieren durch Einfügen benutzt werden kann; hier wird allerdings nur geschoben, denn die Bücher stehen zunächst nicht in dem Feld **hand**, sondern in dem Feld **A**. Das erste Buch wird also von **A** nach **hand[0]** kopierte, und für jedes weitere wird die Einfügestelle gesucht und dann dieses dort hingestellt; die anderen Bücher müssen natürlich verschoben werden.

Stehen jetzt also auf dem Feld **hand** die einzufügenden Bücher aufwärts sortiert, so werden diese vom letzten (dem größten) bis zum ersten (dem kleinsten) mit den Bücher im Regal verglichen, d.h. **hand[distanz]** mit **A[nächster-1..0]** verglichen, bis die Einfügestelle für dieses Buch gefunden wurde. Dieses wird dann an die entsprechende Stelle gebracht und der Abstand (**distanz**)

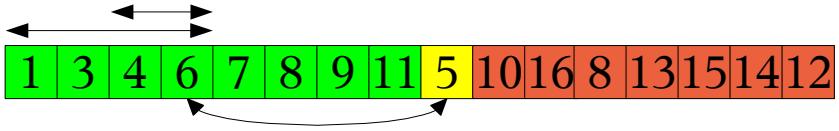
um 1 verringert. Das nächste Buch in **`hand`** muss noch einmal mit dem letzten Buch in **`A`** verglichen werden, da es ja u.U. ebenfalls rechts von diesem stehen könnte. Die innere Schleife wird beendet, wenn alle Bücher von **`hand`** nach **`A`** gebracht wurden, d.h. **`distanz==0`**. Wird jedoch das letzte Buch in **`A`** verschoben – dann sind alle restlichen Bücher in **`hand`** kleiner als das erste Buch in **`A`** – so müssen alle restlichen Bücher von **`hand`** nach **`A`** kopiert werden, was in der letzten **`while`**-Schleife geschieht.

Das Programm ist offenbar deutlich komplexer, ja es verwendet sogar als Hilfsprogramm ein Programm, welches ebenfalls sortiert, wobei man hier am sinnvollsten wieder Sortieren durch Einfügen benutzt. Daran sieht man zum einen, dass die beliebige Vergrößerung der Anzahl gleichzeitig einzufügender Bücher kontraproduktiv wäre, weil die Zeit beim Sortieren in das Feld **`hand`** verbraucht werden würde. Zum anderen sieht man aber auch, dass man häufig komplexere Algorithmen entwickeln kann, welche im wesentlichen auf einfacheren Prinzipien basieren. Die Idee, den gleichen Algorithmus zu verwenden, nur für ein einfacheres Teilproblem, kommt bei der sogenannten *rekursiven Programmierung* erst richtig zum Tragen.

Es gibt aber noch weitere Verbesserungsmöglichkeiten. Z.B. kann man die Bücher links und rechts einsortieren; sortiere beispielsweise alle Bücher, deren Titel mit 'A' bis 'L' anfangen, nach links aufsteigend, die anderen nach rechts absteigend. Warum ist das effizienter? Wir wissen bereits, dass das Sortieren der Hälfte an Büchern nur ein Viertel an Zeit kostet, weil die Bearbeitungszeit quadratisch wächst, und $(N/2)^2=N^2/4$. Du brauchst also zweimal ein Viertel der Zeit, und das ist nun mal zusammen nur noch die Hälfte. Wärst du auch auf diesen Trick gekommen? Wenn ja, dann bist du wirklich schlau!



Man kann die Bearbeitungszeit noch weiter verringern, wenn man die Einfügestelle schneller findet, denn das ständige Vergleichen kostet ja auch Zeit. Nun sind die ersten Bücher ja schon sortiert. Statt vom letzten der sortierten Bücher bis zum ersten alle zu vergleichen, guckt man erst mal in der Mitte nach (im Bild in der Mitte des Intervals [1,...,11]). Liegt die Einfügestelle jetzt links von dem mittleren Buch, so suche in dem Intervall [1,6], sonst in der rechten Hälfte [7,11] usw. Man nennt dieses Verfahren *binäres Suchen*, weil man den Suchbereich ständig halbiert (*binär* bedeutet *zwei*wertig, also in zwei Teile teilen). Hat man die Einfügestelle gefunden, so schiebt man alle Bücher ab dieser Stelle um eine Position nach rechts und kann dann das neue Buch an die richtige Position stellen. Man zeigt leicht, dass auf diese Weise das Finden der Einfügestelle logarithmisch mit der Anzahl der Bücher wächst. Selbst bei Tausend Büchern bräuchte man nicht mehr als zehn Schritte, um die Einfügestelle zu finden, und das ist schon ziemlich gut.



Unser Algorithmus ist nicht nur schnell, er hat auch einige schöne Eigenschaften. So werden bereits bestehende Sortierungen nicht verändert, wenn die Schlüssel gleich sind. Willst du z.B. dein dreibändiges Lexikon, dessen Bände bereits in der richtigen Reihenfolge stehen, verschieben ohne die Reihenfolge dieser Bücher zu verändern, so hat unser Algorithmus damit keine Probleme (außer in der vorletzten Variante!). Solche Algorithmen heißen *stabil*.

Wenn ein Algorithmus eine Vorsortierung ausnutzt, so heißt er *ordnungsverträglich*. Das bedeutet im wesentlichen, dass eine bereits sortierte Folge von Büchern nicht noch einmal sortiert wird; die Anzahl der Vergleiche sollte in diesem Fall nicht viel mehr als die Anzahl der Objekte betragen. Aber auch wenn nur ein paar Bücher verkehrt stehen, läuft unser Algorithmus deutlich schneller als wenn alle zufällig durcheinander stehen. Unser Algorithmus ist also schon ziemlich gut

und wird daher fast überall verwendet, wenn man ein paar Hundert oder wenige Tausend Objekte schnell sortieren will.

Wie du siehst, kann das Sortieren sehr schnell erledigt werden, wenn man nur den richtigen Algorithmus verwendet. Daher ist es in der Informatik so wichtig, den richtigen Algorithmus zu kennen und einzusetzen, weil der richtige Algorithmus u.U. ein Problem erst lösbar macht. Wenn du weitere Verbesserungen dieses Algorithmus kennenlernen willst, oder wenn du die verschiedenen Algorithmen animiert ablaufen lassen willst, so gehe doch einmal auf die Web-Site:

http://einstein.informatik.uni-oldenburg.de/forschung/animAlgo/