2. 5. Such- und Sortieralgorithmen

Suchalgorithmen

Gesucht wird ein Suchargument **x** in einem Array **a**; genauer: ein (oder alle) Indices **i** mit **a[i] == x**.
Wir beschränken uns auf **int**-Arrays; für andere Typen analog.

(1) Sequenzielles (lineares) Suchen

```
int i, found=0;
for (i=0; i <= n; i++)  /* n+1 = Länge von a */
   if (a[i] == x)
      {
      printf("%d\n", i);
      found = 1;
      break;
      }
if (!found)
   printf("Suche erfolglos!\n");
```

("**break**" weglassen, wenn alle Vorkommen von **x** gefunden werden sollen.)

Durchschnittlicher Rechenaufwand ("Zeitkomplexität"):
Schleife wird ca. $\frac{n+1}{2}$ mal durchlaufen.
Man spricht von einem Algorithmus mit *linearer* Komplexität (Zahl der Rechenschritte steigt linear mit *n*) und schreibt

$$O(n)$$

(Bachmann-Landausche "Groß-*O*-Notation"; keine Null gemeint!)

Allgemein heißt "Komplexität O(f(n))":

> Für große *n* ist die Zahl der Rechenschritte nach oben beschränkt durch $c \cdot |f(n)|$, c = const. > 0.

Wichtige Algorithmen-Typen:

$O(n)$ linearer Rechenaufwand
$O(n^2)$ quadratischer Rechenaufwand
$O(n^k)$ (k ganze Zahl > 0) polynomialer Rechenaufwand
$O(\ln n)$ logarithmischer Aufwand
$O(e^n)$ exponentieller Aufwand (*schlecht!*)

(2) Binäres Suchen

Voraussetzung jetzt: die Einträge des Arrays **a** seien nach der Größe aufsteigend sortiert (d.h. **i < j** $\Rightarrow$ **a[i]** $\leq$ **a[j]**).

Beispiel: Suche die Zahl 11 im Array

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
a[i]	**1**	**2**	**4**	**6**	**7**	**8**	**10**	**11**	**13**	**17**	**19**

gehe zuerst in die Mitte
(0+10)/2 = 5, dort ist 8 < 11 $\Rightarrow$ gehe nach rechts
dort wieder in die Mitte:
((5+1)+10)/2 = 8, dort 13 > 11 $\Rightarrow$ gehe nach links,
halbiere weiter... bis 11 gefunden

Algorithmus:

```
int i, j, k, found=0;
i = 0; j = n;    /* n+1 = Länge von a */
do
   {
   k = (i+j)/2;
   if (a[k] == x)
      {
      printf("%d\n", k);
      found = 1;
      break;
      }
   if (a[k] < x)
      i = k+1;
   else
      j = k+1;
   } while (!found && i<j);
```

```
if (!found)
   printf("Suche erfolglos!\n");
```

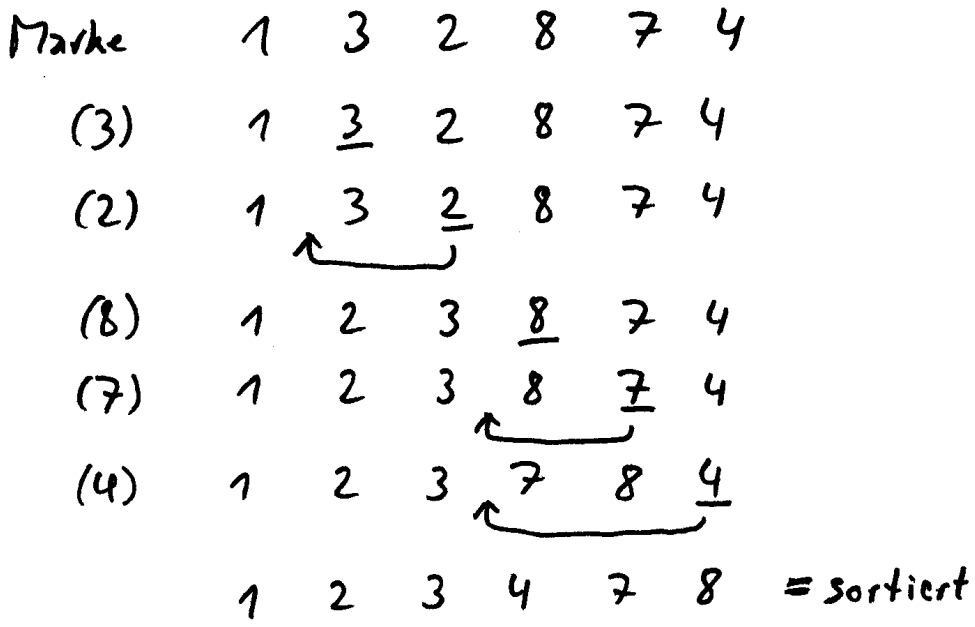
Durchschnittlicher Rechenaufwand ist hier $O(\ln n)$
$\Rightarrow$ der Algorithmus ist für große Arrays viel schneller als sequenzielles Suchen!
Nachteil: Array muss sortiert sein.

Sortieralgorithmen

Ziel: Einträge des Arrays **a** nach der Größe aufsteigend sortieren (absteigende Sortierung ist dann analog möglich).

(3) Sortieren durch direktes Einfügen (*straight insertion*)

Beispiel: Array **1 3 2 8 7 4** zu sortieren
das jeweils markierte Element wird in den schon sortierten vorderen Teil einsortiert:



Algorithmus:

das markierte Element wird in **a[0]** gespeichert (relevante Elemente des Eingabe-Arrays sollen in **a[1]**, ..., **a[n]** stehen).

```
for (i = 2; i <= n; i++)
   {
   x = a[i];    /* markiertes Element */
   a[0] = x;
   j = i – 1;
   while (x < a[j])  /* abwärts gehen */
      {
      a[j+1] = a[j];
      j--;
      }
   a[j+1] = x;
   }
```

Durchschnittliche Zahl der Rechenschritte: $O(n^2)$
Maximale Zahl der Rechenschritte
("worst case complexity"): $O(n^2)$

(4) Bubblesort

Prinzip: Austausch benachbarter Einträge, die die falsche Reihenfolge haben.

```
for (i = 1; i < n; i++)
   for (j = n–1; j >= i; j--)
      if (a[j–1] > a[j])
         {
         x = a[j–1]; a[j–1] = a[j]; a[j] = x;
             /* Tausch der Werte */
         }
```

Komplexität (durchschnittl. und maximal) auch hier $O(n^2)$.

(5) Quicksort

Grundidee:

- wähle willkürlich ein Array-Element **x**
- durchsuche Array von links, bis ein Element **a[i]>x** gefunden wird
- und von rechts, bis ein Element **a[j]<x** gefunden wird
- vertausche diese beiden Elemente

- wiederhole dies solange, bis man sich beim Durchsuchen aus beiden Richtungen irgendwo trifft
  → Resultat: Array **a** ist zerlegt in linken Teil mit Einträgen < **x** und rechten Teil mit Einträgen > **x**
- wende dieses Zerlegungsverfahren nun auf beide Teile erneut an, dann auf deren Teile usw. (Rekursion!), bis jeder Teil nur noch 1 Element enthält → **a** ist sortiert!

Algorithmus:

```
void qsort(int l, int r)
   /* Aufruf im Programm mit sort(0, n–1);
      l = linke Grenze, r = rechte Grenze  */
   {
   int i, j, x, h;
   i = l; j = r;
   x = a[(l+r)/2];
   do
      {
      while (a[i] < x) i++;
      while (x < a[j]) j--;
      if (i <= j)
         {  /* Tausch */
         h = a[i]; a[i] = a[j]; a[j] = h;
         i++; j--;
         }
      } while (i <= j);
   if (l < j)
      qsort(l, j);
   if (i < r)
      qsort(i, r);
   }
```

Komplexität: Durchschnitt $O(n \log n)$, worst case $O(n^2)$
⇒ für große n im Mittel deutlich besser als die "einfachen" Verfahren.

Bem.: Es gibt Verfahren mit $O(n \log n)$-Aufwand auch im *worst case* (*Heapsort*, s. Wirth 1983).