Sortieren durch Mischen (MergeSort)

Divide-and-conquer-Verfahren (Teile-und-Herrsche), hier MergeSort:

1. Die **Anzahl** der zu sortierenden Elemente sei **n**.

2. Im Fall **n=1** gibt es **nichts** zu tun, denn ein einziges Element ist immer **sortiert**.

3. **Zwei Felder**, die **bereits sortiert** sind, können problemlos **zusammengefügt** werden:

   Dazu muß nur das **jeweils kleinste Element** der Felder **verglichen** und in der **entsprechenden Reihenfolge** ins Ergebnis **eingefügt** werden.

   Also **halbieren** wir unser Feld, **sortieren die Hälften rekursiv** und **fügen** das Ergebnis **zusammen**.

Der MergeSort-Algorithmus wurde 1945 von **J.v.Neumann** entdeckt.

Vergleich der Rechenzeiten zwischen Bubble-Sort und MergeSort für **n=1.000.000**

BubbleSort	ca.	**1,5 h**
MergeSort	ca.	**400 ms**

Eignet sich zum Sortieren von Dateien, die nicht in den Hauptspeicher passen. Dazu wird das Sortieren in zwei Schritte zerlegt:

1. Die Folge wird in Teile zerlegt, die jeweils in den Hauptspeicher passen und daher getrennt voneinander mit internen Verfahren sortiert werden können. Diese sortierten Teilfolgen werden wieder in Dateien ausgelagert.

2. Anschließend werden die Teilfolgen parallel gelesen und gemischt, indem jeweils das kleinste Element aller Teilfolgen gelesen und in die neue Folge (d.h. wieder in eine Datei) geschrieben wird.

Dieses **MergeSort**-Verfahren läßt sich aber auch für das interne Sortieren anwenden. Hierbei wird zunächst die zu sortierende Folge in **zwei Teilfolgen** zerlegt, diese werden durch rekursive Anwendung von **MergeSort** sortiert und anschließend gemischt.

Frage: Wie kommt man dabei zu **sortierten Teilfolgen ?** Dies wird dadurch erreicht, daß **MergeSort** solange rekursiv angewendet wird, bis die zu sortierenden Teilfolgen nur noch aus einem Element bestehen. In diesem Fall liefert das **Mischen** eine neue sortierte Folge.

Während **QuickSort** die Datenmenge immer **rekursiv in zwei Teilarrays** zerlegt, die es dann sortiert, geht **MergeSort** umgekehrt vor.

MergeSort mischt zwei **bereits rekursiv sortierte** Teilarrays.

Der Hauptvorteil des **MergeSort** ist, daß seine **Laufzeit** immer proportional **n*lg n** ist.

Sein größter **Nachteil** ist, daß es einen zu **n proportionalen** zusätzlichen Speicherplatz benötigt.

Wenn es um **Schnelligkeit** geht und **genug Speicherplatz** vorhanden ist, ist **MergeSort** dem **QuickSort** vorzuziehen.

Ein weiterer Vorteil des MergeSort ist, daß es den Zugriff auf die Daten **nahezu sequentiell** vornehmen kann. Das ist von Vorteil, wenn auf die zu sortierenden Daten nur sequentiell zugegriffen werden kann, wie z.B. beim Sortieren verketteter Listen.

Algorithmus **Mischen** von 2 Folgen F1 und F2:

```
/* Eingabe: zwei zu sortierende Folgen F1, F2
    Ausgabe: eine sortierte Folge f                          */
algorithm Merge ( F1, F2 ) {
    F := leere Folge;
    While (F1 nicht leer und F2 nicht leer) {
      Entferne das kleinere der Anfangselemente aus F1 bzw. F2;
      Füge dieses Element an F an;
    }
    Füge die verbliebene nichtleere Folge F1 oder F2 an F an;
    return F;
}
```

Mit Hilfe des Algorithmus **Merge** kann nun das eigentliche Sortieren rekursiv realisiert werden.

Das **Abbruchkriterium** ist eine einelementige Folge, die nicht mehr sortiert werden muß und daher unverändert zurückgegeben wird. Anderenfalls wird die Folge zunächst in zwei Teile **F1** und **F2** aufgeteilt, die sortiert werden. Die Ergebnisse werden schließlich mittels **Merge** gemischt.

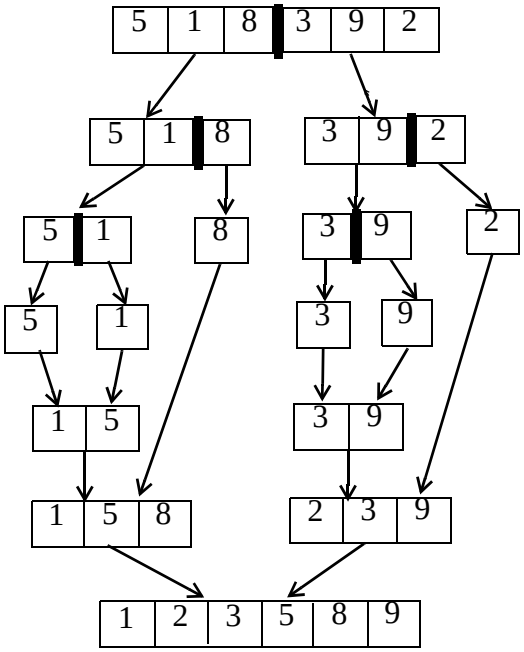
```
/* Eingabe: eine zu sortierende Folge F,   Ausgabe: eine sortierte Folge F_S */
algorithm MergeSort (F) {
    if ( F einelementig ) then return F;
    else { Teile F in F1 und F2;
              F1 := MergeSort (F1);
              F2 := MergeSort (F2);
              return Merge (F1, F2); }
}
```

Der Vorgang des Mischens erfordert in der Regel **doppelten Speicherplatz**, da eine neue Folge aus den beiden sortierten Folgen erzeugt wird. Eine Alternative ist das Mischen in einem Feld, allerdings ist hier ein aufwendigeres Verschieben notwendig.

Die folgende Abbildung demonstriert das Vorgehen beim **MergeSort**:

Zunächst wird die zu sortierende Folge in zwei Teile (**5, 1, 8**) und (**3, 9, 2**) zerlegt.

Durch die rekursiven Aufrufe werden diese Folgen weiter geteilt, bis nur noch einelementige Folgen vorliegen (Abschnitt "**Split**").



Im zweiten Abschnitt "**Merge**" werden die sortierten Teilfolgen gemischt: Zunächst also 5 und 1 sowie 3 und 9. Diese Folgen werden dann mit 8 bzw. 2 und schließlich miteinander in 1, 2, 3, 5, 8, 9 gemischt.

Die Aufteilung in die zwei Abschnitte "**Split**" und "**Merge**" entsteht durch die Rekursion im Algorithmus, indem zu Beginn jeweils das Zerlegen erfolgt und erst als letzter Schritt das eigentliche Mischen.

Für eine Implementierung des Verfahrens sind die Operationen "Entfernen" und "Anfügen" nicht sonderlich gut geeignet.

Das **Zerlegen** kann noch auf einfache Weise durch Indexe fuer Anfang und Ende des jeweiligen Teilfeldes realisiert werden.

Das **Mischen** erfordert ein Hilfsfeld, in das die neu sortierte Folge eingetragen wird.

Das folgende Programm nutzt dieses Prinzip. Da jedem Aufruf die Unter- bzw. Obergrenze des aktuellen Feldes übergeben werden müssen, ist die eigentliche Sortierroutine in der Methode msort implementiert, die von der Methode mergesort aufgerufen wird.

In **msort** werden zunächst die Mitte (**mid**) des Feldes anhand der aktuellen Grenzen **le** und **ri** bestimmt und anschließend die beiden Teilfelder durch den rekursiven Aufruf von **msort** sortiert:

```
/* Sortiert aufsteigend die l Elemente des int-Arrays array */

static void msort(int array[], int le, int ri, int l){
      int i, j, k;
      int *b = new int[l]; //Hilfsfeld anlegen

      if(ri > le){
            //zu sortierendes Feld teilen
            int mid = (ri + le)/2;

            //Teilfelder sortieren
            msort(array, le, mid, l);    //Rekursion
            msort(array, mid+1, ri, l);  //Rekursion

            //Hilfsfeld aufbauen
            for(k=le; k<=mid; k++)
                  b[k] = array[k];
            for(k=mid; k<ri; k++)
                  b[ri+mid-k]=array[k+1];

            //Ergebnisse mischen ueber Hilfsfeld b
            i=le; j=ri;
            for(k=le; k<=ri; k++)
                  if(b[i] < b[j])
                        array[k] = b[i++];
                  else
                        array[k] = b[j--];
      }
      delete [] b; b=0; //Hilfsfeld freigeben
}

static void mergesort(int array[], int l){
      msort(array, 0, l-1, l);
}
```

Im nächsten Schritt werden die beiden Hilfsfelder innerhalb des Hilfsvektors **b** aufgefüllt.

Die Elemente aus dem linken Teilfeld werden dabei auch in die linke Hälfte des Hilfsfeldes **b** kopiert, entsprechend die Elemente aus dem rechten Teilfeld in die rechte Hälfte.

Anschließend werden die beiden Teilfelder aus dem Hilfsfeld gemischt und das Ergebnis in das eigentliche Teilfeld von **array** eingetragen.

Zum Vergleich der jeweils kleinsten Elemente werden die Indexe **i** und **j** verwendet, wobei **i** beim ersten Element beginnt und aufwärts zählt und **j** vom letzten Element aus abwärts läuft.

MergeSort rekursiv

```
#include  <stdio.h>
#define N 6

/*------------------------------------------------------------- merge */
void merge(int z[], int l, int m, int r) {
   int i, j, k;
   int hilf[N];
   for (i=m+1; i>l; i--)
      hilf[i-1] = z[i-1];
   for (j=m; j<r; j++)
      hilf[r+m-j] = z[j+1];
   for (k=l; k<=r; k++)
      z[k] = (hilf[i] < hilf[j]) ? hilf[i++] : hilf[j--];
}
/*------------------------------------------------------- merge_sort */
void merge_sort(int z[], int l, int r) {
   if (l < r) {
      int mitte = (l+r)/2;
      merge_sort(z, l, mitte);
      merge_sort(z, mitte+1, r);
      merge(z, l, mitte, r);
   }
}
/*------------------------------------------------------------- main */
int  main(int argc, char *argv[]) {
      int i, zahlen[N]={5, 1, 8, 3, 9, 2};

      printf("unsortiert:\n");
      for(i=0; i<N; i++)
            printf("%d  ",zahlen[i]);
      printf("\n");

      merge_sort(zahlen, 0, N-1);

      printf("sortiert:\n");
      for(i=0; i<N; i++)
            printf("%d  ",zahlen[i]);
      printf("\n");
      getc(stdin);
      return 0;
}

/*
unsortiert:
5  1  8  3  9  2
sortiert:
1  2  3  5  8  9
*/
```

MergeSort iterativ

```
#include  <stdio.h>
#define MIN(a,b) ( ( (a) < (b) ) ? (a) : (b) )
#define N 6

/*------------------------------------------------------------- merge */
void merge(int z[], int l, int m, int r) {
   int i, j, k;
   int hilf[N];
   for (i=m+1; i>l; i--)
      hilf[i-1] = z[i-1];
   for (j=m; j<r; j++)
      hilf[r+m-j] = z[j+1];
   for (k=l; k<=r; k++)
      z[k] = (hilf[i] < hilf[j]) ? hilf[i++] : hilf[j--];
}
/*-----------------------------------------merge_sort iterativ -------*/
void merge_sort(int z[], int l, int r) {
   int i, m;
   for (m=1; m<r-l+1; m+=m)
      for (i=l; i<=r-m; i+= m+m) {
         int mitte = i+m-1;
         merge(z, i, mitte, MIN(i+m+m-1, r));
      }
}
/*------------------------------------------------------------- main */
int  main(int argc, char *argv[]) {
      int i, zahlen[N]={5, 1, 8, 3, 9, 2};

      printf("unsortiert:\n");
      for(i=0; i<N; i++)
            printf("%d  ",zahlen[i]);
      printf("\n");

      merge_sort(zahlen, 0, N-1);

      printf("sortiert:\n");
      for(i=0; i<N; i++)
            printf("%d  ",zahlen[i]);
      printf("\n");
      getc(stdin);
      return 0;
}

/*
unsortiert:
5  1  8  3  9  2
sortiert:
1  2  3  5  8  9
*/
```

4.4 Quicksort (C. A. R. HOARE)

Quicksort (von engl. quick – schnell, to sort – sortieren) ist ein **schneller**, **rekursiver** Sortieralgorithmus, der nach dem Prinzip **Teile und herrsche** (lat. Divide et impera!, engl. **Divide and conquer**) arbeitet.

Er wurde ca. 1960 von **C. Antony R. Hoare** entwickelt. Der Algorithmus hat den Vorteil, dass er über eine sehr kurze innere Schleife verfügt (was die Ausführungsgeschwindigkeit stark erhöht) und ohne zusätzlichen Speicherplatz auskommt (abgesehen von dem für die Rekursion zusätzlichen benötigten Platz auf dem Aufruf-Stack).

Verfahren:

(a) Es wird ein beliebiges Listen-/Vektorelement als sogenanntes **Pivotelement** ausgewählt (das erste Element oder das letzte Element oder zufällig oder drei zufällig, davon das mittlere Elemen, o.a.)

(b) Es werden **zwei Teillisten/Teilvektoren** erzeugt:

Die **erste Teilliste/Teilvektor** erhält alle Elemente, die **kleiner oder gleich** als das **Pivotelement** sind (außer dem Pivotelement selbst).

Die **zweite Teilliste** erhält alle anderen Elemente, d.h. diejenigen, die **größer als** das **Pivoelement** sind (außer dem Pivotelement selbst).

Innerhalb der Bereiche sind die Elemente noch nicht sortiert.

(c) Jeder der **beiden Teillisten** wird wiederum mit dem **gleichen Verfahren** sortiert. Leere oder 1-elementige Listen gelten als sortiert (und brauchen nicht weiter behandelt zu werden). Die **Abbruchbedingung** ist erreicht, wenn die Dimension der **Sequenz 1** ist.

(d) Das **Gesamtergebnis** ergibt sich, indem man an die **sortierte erste Liste** das **Pivotelement** anschließt und daran dann die **sortierte zweite Liste** anhängt.

Mittlere Anzahl von Vergleichen ~ **n*ln(n)**, wobei **n** die Anzahl der Listenelemente ist

Die Schritte (a) und (b) kann man zusammen fassen:

Bei einem Durchlauf der Liste mit Hilfe von zwei gegenläufigen Zeigern (einer vom ersten Element und einer vom letzten Element beginnend) erzeugt man innerhalb des Speicherbereiches der Liste die beiden Teillisten und wählt das dazwischen stehende Element als Pivotelement.

Beispiel: Datei **QSort.in** (Textdatei, mit Editor les- und bearbeitbar):

```
a[0] a[1] a[2] a[3] a[4] a[5] a[6]  n = 7
 345   67   13  789   90  76  -45
```

Datei **QSort.out** (Textdatei, als Ergebnis des Sortierens mit QuickSort):

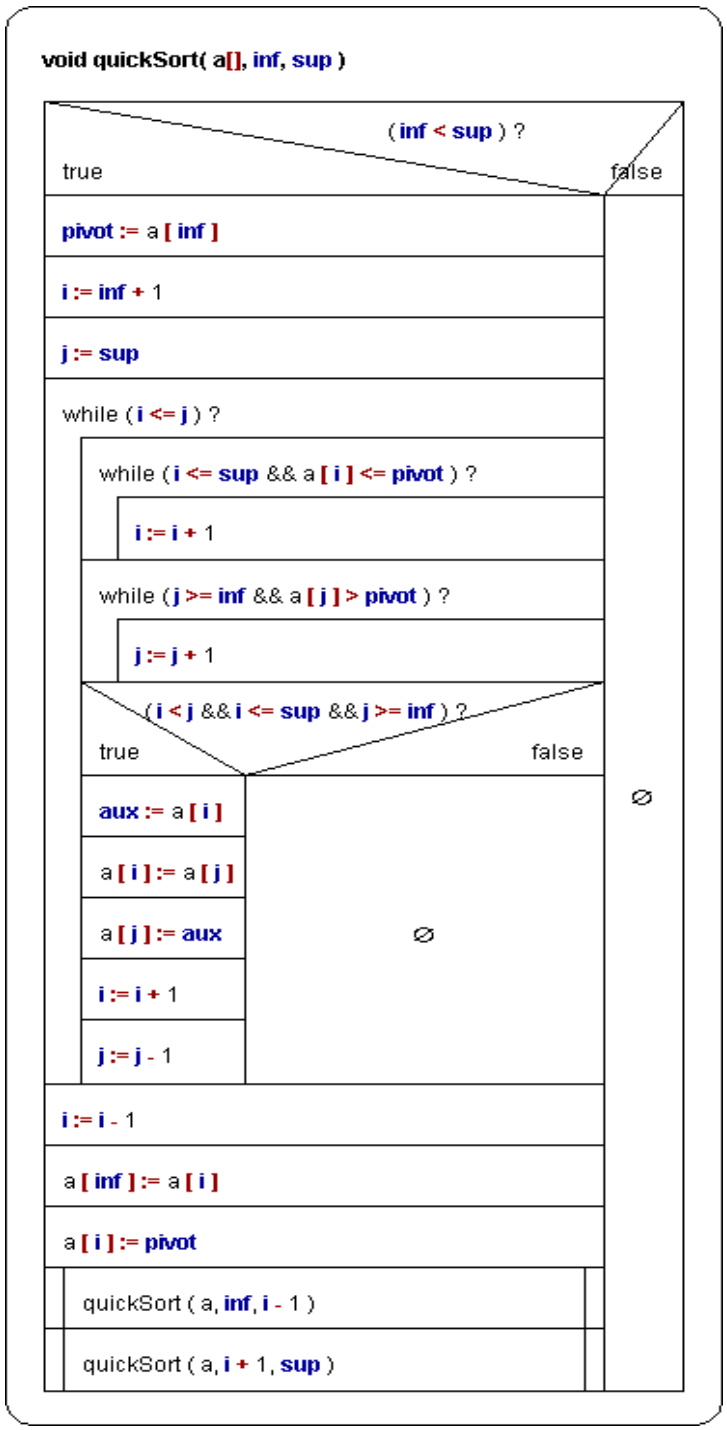
```
a[0] a[1] a[2] a[3] a[4] a[5] a[6]
 -45   13   67   76   90  345  789
```

Aufgabe: Notieren Sie auf dem Papier die Schritte für das Beispiel anhand des **Pseudocodes** auf der nächsten Seite bzw. anhand des **C++ - Codes** auf der übernächsten Seite.

Quicksort - Algorithmus als Pseudocode

```
funktion quicksort(daten, links, rechts)
     falls links < rechts dann
         teiler := teile(daten, links, rechts)
         quicksort(daten, links, teiler-1)
         quicksort(daten, teiler+1, rechts)
     ende falls
 ende

funktion teile(daten, links, rechts)
     // Starte mit i rechts vom PivotElement (PE)
     i := links + 1

     j := rechts
     pivot := daten[links]

     wiederhole solange i <= j  // solange i an j nicht
                                // vorbeigelaufen ist

      //Suche von links Element, welches größer als PE ist
         wiederhole solange i <= rechts und daten[i] <= pivot
             i := i + 1
         ende wiederhole

      //Suche von rechts Element, welches kleinergleich
      // als PE ist
         wiederhole solange j >= links und daten[j] > pivot
             j := j - 1
         ende wiederhole

         falls i < j und i<=rechts und j>=links dann
             begin
                   tausche daten[i] mit daten[j]; i++; j--;
             ende
     ende wiederhole
     i--;

  // Tausche PE mit neuer endgültiger Position (daten[i])

     tausche daten[i] mit daten[links]

  // gib die Position des Pivotelements zurück

     return i
 ende
```

Struktogramm Algorithmus QuickSort fuer C++ - Code



```
#include <fstream>              // Header-File zur Dateiarbeit
#include <vector>               // Header-File fuer vector - Template

using namespace std;

void readData(vector<int> &a){  // Funktion zum Lesen nach int-Vektor a
  int aux;
  ifstream f("QSort.in");       // Eingabestrom aus Datei QSort.in
  while(f && !f.eof() && f>>aux)// Eingabestrom exist. und nicht f.EOF
                                // und fehlerfrei Lesen von f nach aux ?
    a.push_back(aux);           // fuegt Wert von aux an Ende des Vektors an
}

void quickSort(vector<int> &a, int inf, int sup){ // Parameter a, inf, sup
  if(inf<sup){                // Falls linker Index inf kleiner als rechter Index sup
    int pivot = a[inf], aux;       // pivot := linkes Element a[inf]
    int i = inf + 1, j = sup;      // Beginne mit i: = inf+1 und j: = sup
    while(i<=j){            // Wiederhole, solange linker Index i <= rechter Index j
      while(i<=sup && a[i] <= pivot) i++; // suche von links El., welches
                                         // groesser als Pivotelement ist
      while(j>=inf && a[j]  > pivot) j--; // suche von rechts El., welches
                                         // kleinergleich als Pivotel. ist
      if(i<j && i<=sup && j>=inf){       // Falls Index i des linken El. kleiner
                                         // als Index j des rechten Elementes
                                         // und Indexe i und j zulaessig, dann
        aux = a[i];                      // vertausche a[i] mit a[j]
        a[i] = a[j];
        a[j] = aux;
        i++; j--;                        // Erhoehe i um 1, verringere j um 1
      }
    }                       // Ende while(i<=j), nach while(i<=j){...} gilt i > j
    i--;                 // i:=i-1, damit gilt i == j, a[i] wird neues Trennelement

    a[inf] = a[i]; a[i] = pivot;   // vertausche a[inf] == pivot mit a[i]

    quickSort(a, inf, i-1); // rekursiver Aufruf fuer den linken Bereich
                            // a[inf] ... a[i-1] von a
                            // a[i] ist Trennelement
    quickSort(a, i+1, sup); // rekursiver Aufruf fuer den rechten Bereich
                            // a[i+1] ... a[sup] von a
  }                         // Ende von if(inf<sup), d.h. Ende wegen
}                           // inf >= sup (einelementiger Bereich von a)

void write(vector<int> a){              // Funktion zum Schreiben nach int-Vektor a
  ofstream f("QSort.out");              // Ausgabestrom nach Datei QSort.out
  for(int i=0; i<(int)a.size(); i++)    // alle a[i] durchlaufen und in
        f << a[i] << " ";               // Datei schreiben, die mit f verbunden
}

int main(){
  vector<int> a;                        // int-Vektor a vereinbaren
  readData(a);                          // Aufruf readData(a) zum Lesen in Vektor a

  quickSort(a, 0, (int)a.size()-1);     // Aufruf Quicksort fuer Vektor a von
                                        // a[0] bis a[a.size()-1]
  write(a);                             // Aufruf von write(a) zum Schreiben
  return 0;                             //  vektor a in File
}
```

Aufgabe:

Modifizieren Sie das obige **C++ - Programm** bzw. den **Pseudocode** so, daß das **Pivot**-Element **zufällig** aus dem Intervall genommen wird.

QuickSort

Algorithmus:

- Array **z[p .. r]** wird in **zwei nichtleere disjunkte** Teilarrays **z[p ..q]** und **z[q+1 .. r]** zerlegt, so daß alle Elemente in **z[p .. q] kleiner** (oder gleich) als alle in **z[q+1 .. r]** sind.

- Funktion **partition()**, die für die Zerlegung in Teilarrays zuständig ist, liefert den **Index** des **Pivot-Elements**, das die **Trennstelle** zwischen den beiden Teilarrays ist.

- Die **Teilarrays** werden nun ihrerseits wieder nach dem **gleichen Verfahren** durch **rekursive** Aufrufe des **QuickSort** sortiert.

```
void swap(int *a, int *b) {
   int h = *a;
   *a = *b;
   *b = h;
}

int partition(int z[], int l, int r) {
    int x = z[r],                    // Pivot ist rechtes Element
        i = l-1,
        j = r;
    while (1) {
       while (z[++i] < x)
          ;
       while ((l<j) && z[--j] > x))
          ;
       if (i < j)
          swap(z+i, z+j);
       else {
          swap(z+i, z+r);
          return i;
       }
    }
}

void quick_sort(int z[], int l, int r) {
   if (l < r) {
      int pivot = partition(z, l, r);
      quick_sort(z, l, pivot-1);
      quick_sort(z, pivot+1, r);
   }
}

void main() {
      unsigned long n = 0UL;
      int *values = 0;          // Zeiger auf Vektor values == 0
      read(values, n);          // Aufruf des rekursiven Lesens von Tastatur
      quick_sort(values, 0, n-1);
      // ...
}
```

```
#include <iostream>                  // Quicksort alternativ
using namespace std;
void swap(int *a, int *b) {          // Vertauschen *a mit *b
   int h = *a;
   *a = *b;
   *b = h;
}

int partition(int z[], int l, int r) {
    int x = z[r],                    // Vergleich mit x=z[r]
        i = l-1,
        j = r;
    while (1) {
       while (z[++i] < x)
          ;
       while ((l<j) && (z[--j] > x))
          ;
       if (i < j)
          swap(z+i, z+j);
       else {
          swap(z+i, z+r);
          return i;
       }
    }
}

// alternativ zur vorherigen Funktion partition
int partition(int z[], int l, int r) {
    int x = z[l],                    //Vergleich mit x=z[l]
        i = l,
        j = r+1;
    while (1) {
       while ((i<r) && (z[++i] < x))
          ;
       while (z[--j] > x)
          ;
       if (i < j)
          swap(z+i, z+j);
       else {
          swap(z+j, z+l);
          return j;
       }
    }
}

void quick_sort(int z[], int l, int r) {
   if (l < r) {
      int pivot = partition(z, l, r);
      quick_sort(z, l, pivot-1);
      quick_sort(z, pivot+1, r);
   }
}
```

```
// Funktion zum rekursiven Einlesen von int-Zahlen ueber
// Tastatur, maximal ULONG_MAX == 4 294 967 295 (0xffffffff)
// Zahlen (falls der Stack-Speicher reicht !)

// f ist Parameter fuer Ein-/Ausgabe, beschreibt dynamisch
// vereinbarten Vektor mit n Elementen
//
// n ist Parameter fuer Ein-/Ausgabe, beschreibt die Anzahl
// der gelesenen Zahlen, dient beim Anlegen des Vektors
// als Grenze, wird auch ausserhalb von read() fuer das
// Durchlaufen des Vektors values[0..n-1] genutzt

void read(int * &f, unsigned long &n){ // 2 Referenzparameter
      int i = 0;               // lokaler Speicher fuer i
      unsigned long j = n;     // Index n nach j
      cout<<"int-Zahl = "; cin>>i; // Lesen der naechsten Zahl
      if(!cin.eof()){   // kein Dateiende gelesen
                              n++;   // n := n + 1
                              cin.clear(); cin.ignore(INT_MAX, '\n');
                              read(f, n);  // rekursiver Aufruf
      }
      if(cin.eof()){   // EOF (F6) gelesen
                  f = new int[n];// anlegen des int-Vektors f
                  cin.clear();   // cin.eof() zurueck auf false
                  if(f==0){      // Speicherplatzzuweisung fuer
                                    // Vektor f nicht moeglich !
                     cout<<"kein Speicherplatz bei n = "<<n<<endl;
                     n=0UL;      // n:=0, da kein Speicherplatz
                  }
                  return;
      }
      f[j] = i;  // in der Ebene j gelesener Wert i nach f[j]
}

// Kleiner Test: Lass es quicken!
void main() {
      unsigned long n = 0UL;
      int *values = 0;       // Zeiger auf Vektor values == 0
      read(values, n);       // Aufruf des rekursiven Lesens von Tastatur
      // Ausgabe der Werte (vorher)
      cout<<"Werte vor dem Sortieren:\n";
      for (unsigned long i = 0UL; i < n; i++)
        cout<<values[i]<<"  ";
      // Sortiere alle (count!) Elemente von values
      quick_sort(values, 0, n-1);
      // Ausgabe der Werte (nachher)
      cout<<"\n\nWerte nach dem Sortieren:\n";
      for (unsigned long i = 0UL; i < n; i++)
        cout<<values[i]<<"  ";
      delete [] values; values=0; // Freigabe des Vektors
      cin.get();
}
```

qsort in C / C++

Realisiert den Quicksort Algorithmus.

```
void qsort(                     /* Funktionsname, kein return-Wert          */
           void *base,          /* Adresse (0. El.) des zu sortierenden Arrays */
           size_t num,          /* Anzahl der Array-Elemente                */
           size_t width,        /* Größe eines Array-Elementes in Byte      */
           int (*compare )(     /* Zeiger auf Vergleichsfunktion, Typ int   */
             const void *elem1, /* 1.Arg., Zeiger auf zu sortierendes Element */
              const void *elem2 /* 2.Arg., Zeiger auf Vergleichsel. zum elem1 */
           )                    /* Ende der Parameterliste von compare      */
);                              /* Ende der Parameterliste von qsort        */
```

Routine	Required Header	Compatibility
qsort	<stdlib.h> oder <search.h>	ANSI, Windows 7

Bemerkungen:

The **qsort** function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted.

qsort overwrites this array with the sorted elements. The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort** calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

*compare***((void *)** *elem1***, (void *)** *elem2* **);**

The routine must compare the elements, then return one of the following values:

Return Value	Description
< 0	*elem1 < elem2*
0	*elem1 == elem2*
> 0	*elem1 > elem2*

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of “greater than„ and “less than„ in the comparison function.

Jeder Zeiger kann in den Typ **void *** und zurück umgewandelt werden, ohne daß Information verloren geht. Deshalb kann **qsort** aufgerufen und dabei die Argumente explizit in den Typ **void *** umgewandelt werden.

```c
#include <stdio.h>              // Beispiel qsort1
#include <stdlib.h>
#include <string.h>

int sort_str_up(const void *, const void *);
int sort_str_down(const char *, const char *);
int sort_int_up(const int *, const int *);
int sort_int_down(const void *, const void *);

char list[][4]={"cat","cha","cab","cap","can"};

int vektor[]={9, 8, -4, 5, 10, 7, 1, 12 };

void main() { int i;
  printf("Steigende Sortierung des char * - Vektors:\n");
  qsort((void *)list, sizeof(list)/sizeof(list[0]), sizeof(list[0]),
        &sort_str_up);
  for(i=0; i<sizeof(list)/sizeof(list[0]); i++) printf("%s ",list[i]);

  printf("\n\nFallende Sortierung des char * - Vektors:\n");
  qsort(list, sizeof(list)/sizeof(list[0]), sizeof(list[0]), sort_str_down);
  for(i=0; i<sizeof(list)/sizeof(list[0]); i++) printf("%s ",list[i]);

  printf("\n\nSteigende Sortierung des int - Vektors:\n");
  qsort(vektor, sizeof(vektor)/sizeof(int), sizeof(int), sort_int_up);
  for(i=0; i<sizeof(vektor)/sizeof(int); i++) printf("%d ", vektor[i]);

  printf("\n\nFallende Sortierung des int - Vektors:\n");
  qsort((void *)vektor, sizeof(vektor)/sizeof(int), sizeof(int),
        sort_int_down);
  for(i=0; i<sizeof(vektor)/sizeof(int); i++) printf("%d ", vektor[i]);
}

int sort_str_up(const void *a, const void *b)
{
      return( strcmp((char *)a, (char *)b));
}

int sort_str_down(const char *a, const char *b)
{
      return( strcmp(b, a));
}

int sort_int_up(const int *a, const int *b)
{
      return((*a<*b)?-1:((*a>*b)?1:0));
}

int sort_int_down(const void *b, const void *a)
{
      return((*(int *)a<*(int *)b)?-1:((*(int *)a>*(int *)b)?1:0));
}
```

```
/*
Steigende Sortierung des char * - Vektors:
cab can cap cat cha

Fallende Sortierung des char * - Vektors:
cha cat cap can cab

Steigende Sortierung des int - Vektors:
-4 1 5 7 8 9 10 12

Fallende Sortierung des int - Vektors:
12 10 9 8 7 5 1 -4
*/

-------------------------------------------------------------------------

// Beispiel qsort.cpp
#include<iostream>
using namespace std;

#include<cstdlib>   // enthält Prototyp von qsort()

// Definition der Vergleichsfunktion
int icmp(const void *a, const void *b)
{
    // Typumwandlung der Zeiger auf void in Zeiger auf int
    // und anschließende Dereferenzierung (von rechts lesen)
    int ia = *(int *)a;
    int ib = *(int *)b;

    // Vergleich und Ergebnisrückgabe ( > 0, = 0, oder < 0)
    if(ia == ib) return 0;

    return ia > ib ? 1 : -1;
}

void main()
{
    int ifeld[] = {100,22,3,44,6,9,2,1,8,9};

    // Die Feldgröße ist die Anzahl der Elemente des Feldes.
    // Feldgröße = sizeof(Feld) / sizeof(ein Element)

    int Groesse = sizeof(ifeld)/sizeof(ifeld[0]);

    // Aufruf von qsort():
    qsort(ifeld, Groesse, sizeof(ifeld[0]), icmp);

    // Ausgabe des sortierten Feldes:
    for (int i = 0; i < Groesse; ++i)
         cout << " " << ifeld[i];
    cout << endl;

    cin.ignore();
}

// 1 2 3 6 8 9 9 22 44 100
```