Inhaltsverzeichnis

1. Algorithmen in C

1.1. Ziele. Voraussetzungen

☑ Voraussetzungen:

- ☐ Programmierung mit C
- ☐ Arrays und verkettete Listen in der Praxis einsetzen können

☑ Ziele

- ☐ Komplexe, Algorithmen einsetzen und erstellen können

☑ Quellen:

- ☐ www.pronix.de

1.2. Was sind Algorithmen

Aus wikipedia:

Ein **Algorithmus** (auch Lösungsverfahren) ist eine formale **Handlungsvorschrift** zur Lösung eines Problems in endlich vielen Schritten.

Algorithmen können in Programmablaufplänen nach DIN 66001 oder ISO 5807 grafisch dargestellt werden.

1.3. Exkurs: Rekursion

Vgl. Wikipedia:

Als Rekursion (lat. recurrere „zurücklaufen“)
bezeichnet man die Technik, dass eine Funktion **sich selbst aufruft**.

Die Rekursion ist eine von mehreren möglichen Problemlösungsstrategien, sie führt oft zu eleganten Darstellungen. Rekursion und Iteration sind im Wesentlichen gleichmächtige Sprachmittel. In ihrer Implementierung kann es Effizienzunterschiede geben.

Wichtig ist, dass die rekursive Funktion ein sogenanntes **ABBRUCHSKRITERIUM** definieren muss.

Im Fall von einfachen rekursiven Funktionen steht es dem Programmierer frei, eine iterative oder eine rekursive Implementierung zu wählen. Dabei ist die rekursive Umsetzung meist eleganter, während die iterative Umsetzung effizienter ist (insbesondere weil der Stack weniger beansprucht wird und der Overhead für den wiederholten Funktionsaufruf fehlt); siehe auch das Programmierbeispiel unten.

1.3.1. Beispiel: summe.c

☑ Definition:

$$\mathrm{sum}(n) = \begin{cases} 0 & \text{falls } n = 0 \quad (\text{Rekursionsanfang}) \\ \mathrm{sum}(n-1) + n & \text{sonst} \quad (\text{Rekursionsschritt}) \end{cases}$$

☑ Die Summe der Zahlen von 0 bis 3 berechnet sich dann wie folgt:

$$\begin{aligned} \mathrm{sum}(3) &= \mathrm{sum}(2) + 3 && (\text{Rekursionsschritt}) \\ &= \mathrm{sum}(1) + 2 + 3 && (\text{Rekursionsschritt}) \\ &= \mathrm{sum}(0) + 1 + 2 + 3 && (\text{Rekursionsschritt}) \\ &= 0 + 1 + 2 + 3 && (\text{Rekursionsanfang}) \\ &= 6 \end{aligned}$$

☑ Die Aufruf-Kette dazu sieht so aus:

$$\mathrm{sum}(3) \rightarrow \mathrm{sum}(2) \rightarrow \mathrm{sum}(1) \rightarrow \mathrm{sum}(0).$$

☑ C-Programm

```
int sum(int n){
     if (0==n)
          return 0; //ABBRUCHSKRITERIUM

     return n + sum(n-1);
}
```

1.3.2. Beispiel: fibonacci.c

☑ Die Definition

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \quad \text{(Rekursionsanfang)} \\ 1 & \text{falls } n = 1 \quad \text{(Rekursionsanfang)} \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{sonst} \quad \text{(Rekursionsschritt)} \end{cases}$$

☑ Die dritte Fibonacci-Zahl wird zum Beispiel folgendermaßen berechnet:

$$\begin{aligned} \mathrm{fib}(3) &= \mathrm{fib}(2) + \mathrm{fib}(1) && \text{(Rekursionsschritt)} \\ &= \mathrm{fib}(1) + \mathrm{fib}(0) + \mathrm{fib}(1) && \text{(Rekursionsschritt)} \\ &= 1 + \mathrm{fib}(0) + \mathrm{fib}(1) && \text{(Rekursionsanfang)} \\ &= 1 + 0 + \mathrm{fib}(1) && \text{(Rekursionsanfang)} \\ &= 1 + 0 + 1 && \text{(Rekursionsanfang)} \\ &= 2 \end{aligned}$$

Erstellen Sie eine rekursive Version zur Berechnung einer Fibonacci-Zahl.

int fib (int n);

und schreiben Sie ein kleines Testprogramm.

1.3.3. Beispiel: dez2bin.c

Um eine Dezimalzahl in eine Dualzahl umzuwandeln, kann man folgendermaßen vorgehen:

13	:	2	=	6	REST: 1	Stelle: 0
6	:	2	=	3	REST: 0	Stelle: 1
3	:	2	=	1	REST: 1	Stelle: 2
1	:	2	=	0	REST: 1	Stelle: 3

Die Dualzahl lautet also: 1101
Kontrolle: $1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =$

```
1*8 + 1*4 + 0*2 + 1*1 =
8 + 4 + 1=
13
```

Schreiben Sie die rekursive Funktion
void dez2bin(int n)
die als Parameter eine Dezimalzahl erhält und diese auf die Konsole als Dualzahl ausgibt.

Rekursion findet man bei:

☑ Backtracking Methoden (Versuch und Irrtum): siehe unten

☑ Dynamischen Datenstrukturen: siehe Lerneinheit: „C Datenstrukturen“

☑ Sortieralgorithmen (Quicksort, …): s.u.

☑ ...

1.3.4. Aufgabe: Rekursion - ggt-pow-fast-pow

Bringen Sie die folgenden Programme zum Laufen:

DS-ALGO/01-algorithmen/02-ueben/ue-rekursion-ggt-pow-fast-pow.txt

1.4. Sortieren

Sortieralgorithmen zählen zu den Basistechnologien der Informatik.

Hier einige Typen von Sortieralgorithmen:

☑ **Internes** Sortieren - findet innerhalb des RAMs (Arbeitsspeicher) statt. Dabei werden meist Daten an das Programm geschickt und werden sortiert wieder ausgegeben.

☑ **Externes** Sortieren - hier werden externe Speicherquellen (Festplatte, Streamer, Tape ...) verwendet. Während des externen Sortierens werden zahlreiche Lese- und Schreibzugriffe auf externe Quellen ausgeführt. Externes Sortieren wird genutzt, wenn die Daten zum Sortieren nicht auf einmal im RAM verarbeitet werden können.

☑ **Vergleichendes** Sortieren - dabei wird häufig ein **Schlüssel** zum Sortieren verwendet. Dieser Schlüssel besteht meist nur aus einem kleinen Teil der Daten, der zum Auffinden der gesamten Daten dient.

☑ **Stabiles** Sortieren - stabil wird sortiert, wenn z.B. eine Arbeitnehmerliste, die nach Alphabet sortiert ist, nach Gehalt sortiert wird, ohne dass dabei die alphabetische Liste durcheinander gerät.

Im Folgenden werden häufig Arrays zum Sortieren verwendet. Diese sollten Sie sich als Schlüssel einer Datenstruktur vorstellen.

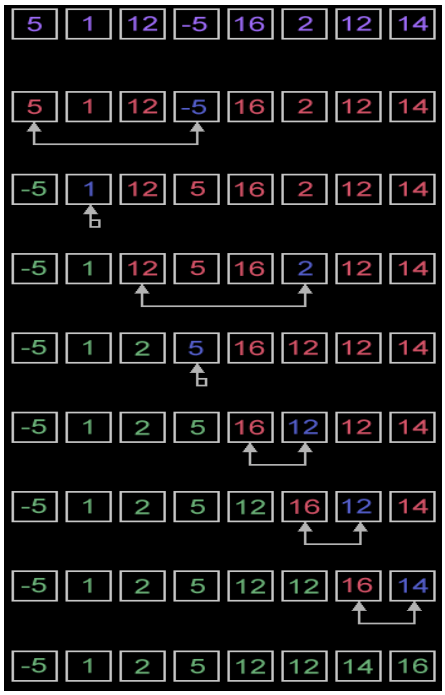
1.4.1. Selection Sort

http://www.algolist.net/Algorithms/Sorting/Selection_sort

Das einfachste Sortierverfahren ist der Selection-Sort. Es funktioniert wie folgt:

Suche das **kleinste** Element im Array und vertausche es mit dem ersten.
Suche das nächste kleine Element und vertausche es mit dem zweiten.
...

Example. Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



1.4.2. Beispiel: selectionSort.c

```
void selectionSort(int arr[], int n) {
      int i, j, minIndex, tmp;
      for (i = 0; i < n - 1; i++) {
           /* ans i-te Arrayelement soll das minimum gebracht werden */


            minIndex = i;
            for (j = i + 1; j < n; j++)
                  if (arr[j] < arr[minIndex])
                        minIndex = j;

            if (minIndex != i) {
                  tmp = arr[i];
```

```
                    arr[i] = arr[minIndex];
                    arr[minIndex] = tmp;
               }
          }
}
```

Der Vorteil von "Selektion Sort" liegt darin, dass jedes Element höchstens einmal bewegt wird.

1.4.3. Beispiel: bubbleSort.c

Der Bubble-Sort wird sehr häufig als Einführungsbeispiel für Sortierverfahren gewählt. Das Vorgehen ist relativ einfach:
Paarweise werden die Elemente verglichen und bei Bedarf vertauscht. Dies wird solange durchgeführt bis nicht mehr vertauscht werden muss.

```
/*
 * bubble_sort(int array[], int n) sortiert das Feld array[] mit
 *   n Elementen in aufsteigender Reihenfolge mit dem
 *   Bubble-Sort-Verfahren.
 */

void bubble_sort(int array[], int n)
{
     int i, j, t;
     int fertig;

     /* Schleife ueber alle Elemente bis bis nicht mehr vertauscht
      werden muss */

     fertig=0;
     while (fertig==0){
          fertig=1;
          for( i=0; i<n-1; i++ ) {
               /* Vergleichen der weiteren Elemente
                  und möglicherweise vertauschen */
               if( array[i] > array[i+1] ) {
                    t = array[i];
                    array[i] = array[i+1];
                    array[i+1] = t;
                    fertig=0;
               }
          }
     }
}
```

1.4.4. Laufzeitverhalten einfacher Sortierverfahren

Da im Prinzip alle einfachen Sortierverfahren aus zwei ineinander geschachtelten Schleifen bestehen, die im schlechtesten Fall über nahezu alle Feldelemente laufen kann man sagen, dass es sich hier um ein quadratisches Laufzeitverhalten handelt.

Alle einfachen Sortierverfahren sind im schlechtesten Fall von der Ordnung O(n^2).

Diese Notation O(n^2) wird weiter unten genauer behandelt.

1.4.5. Beispiel: quickSort.c

Einer der am häufigsten verwendeten Sortieralgorithmen ist der Quick-Sort. Die eigentliche Idee stammt aus dem Jahr 1960 von C.A.R. Hoare.

Quick-Sort arbeitet nach dem ,,Teile und Herrsche`` Prinzip. Ein Array wird dabei in zwei Teile zerlegt, die dann unabhängig voneinander sortiert werden. Die Teilung basiert dabei auf dem zu sortierenden Array. Der Algorithmus basiert also auf einer Rekursion:

```
int partition(int* array, int l, int r);

void quick_sort(int* array, int left, int right)
{
     int p;

     if( left<right ) {
          p = partition(array, left, right);
          quick_sort(array, left, p-1);
          quick_sort(array, p+1, right);
     }
     return;
}
```

Durch den Aufruf

```
quick_sort(arr, 0, n-1);
```

wird das Array arr[] sortiert.

int partition(int* array, int l, int r)

Die Schwierigkeit besteht nun in der Zerlegung des Arrays.
Dabei **muss eine Umordnung des Arrays** vorgenommen werden, so dass folgende Bedingungen erfüllt sind:

- Das Element `array[i]` befindet sich am richtigen Index `i` im Array.
- Alle Elemente `array[0]` `array[i-1]` sind kleiner als `array[i]`.

- Alle Elemente `array[i+1]` bis `array[n-1]` sind größer als `array[i]`.

Für diese Zerlegung gibt es einige Verfahren. Eines der einfacheren funktioniert nach folgendem Muster:

1. Zu Beginn wähle man zwei Indizes
   `i``=l+1` und
   `j``=r` und
   das erste Element im Array als Vergleichswert **`key`**`=array[l]`.

2. In einer Schleife **erhöhe** man **`i`** bis das Arrayelement `array[i]>key` ist.
   Dieses Arrayelement gehört in die rechte Hälfte des Arrays. Alle Elemente der rechten Hälfte sollen größer `key` sein.

3. In einer Schleife **verringere** man **`j`** bis das Arrayelement `array[j]<key` ist.
   Dieses Arrayelement gehört in die linke Hälfte des Arrays. Alle Elemente der linken Hälfte sollen kleiner `key` sein.

4. Sollte `i>=j` sein beende die Schleife.

5. Man **vertausche** die Elemente `array[i]` und `array[j]`.

6. Wiederhole alles ab Punkt 2.

7. Zum Schluß **vertausche** man das erste Element im Array (**`key`**) mit `array[j]`.
   Damit sind alle obigen Bedingungen erfüllt.

Hier ein Beispiel:

```
10      7       16      4       9       18      -3      8       12
l                                                       r
key=10

10      7       16      4       9       18      -3      8       12
                l                                       r

10      7       8       4       9       18      -3      16      12
                                …
                                …
                                …

10      7       8       4       9       -3      18      16      12
                                                l
                                                r

-3      7       8       4       9       10      18      16      12
[-------- rekursion: qsort ----]        j       [ rekursion: qsort]
```

```
int partition(int* array, int l, int r)
{
     int i, j, t, key;

     /* Schluessel - 1.Element */
```

```
     key = array[l];
     i=l+1;
     j=r;

     /* Schleife fuer Elemente
        von links und rechts */
     for(;;) {
          /* Elemente bis zum ersten
             der rechten Haelfte durchlaufen */
          while( array[i] <= key && i<r )
               i++;

          /* Elemente bis zum ersten
             der linken Seite durchlaufen */
          while( array[j] >= key && j>l )
               j--;

          /* Abbruch */
          if( i>=j )
               break;

          /* beide vertauschen */
          t = array[i];
          array[i] = array[j];
          array[j] = t;
     }

     /* letztes Element der rechten Haelfte
        mit dem ersten Element vertauschen */
     t = array[j];
     array[j] = array[l];
     array[l] = t;

     return j;
}
```

Die Performance des Quick-Sort ist charakterisiert, durch die Sätze:
☑ Quick-Sort ist im schlechtesten Fall von der Ordnung $O(n^2)$.
☑ Quick-Sort benötigt im Mittel **2 * n * ln (n)** Vergleiche.

Der schlechteste Fall tritt dann ein, wenn das Feld bereits sortiert ist. Dann wählt der Teilungsalgorithmus immer das erste Element aus, wobei die linke Hälfte leer ist und die rechte den Rest des Feldes enthält.

Der Quicksort ist abhängig von der Anordnung der Daten.

1.4.6. Beispiel: qsort.c - Die C-Standardfunktion

Im folgenden Beispiel wird die Verwendung der C-Standardbibliotheksfunktion qsort() gezeigt:

```
/*
 * anton hofmann     17.3.2000
 * file: t_qsort.c
 * gcc t_qsort.c -o t_qsort
 * demo: qsort int-array
 */
#include <stdio.h>
#include <stdlib.h>

#define MAXENTRIES 10
int feld[MAXENTRIES];

/* ---------------- vergleichsfunktion */
int comp(int* a, int* b);

int main(){
     int i;

     puts("qsort()-Demo: int array: Bitte int-zahlen eingeben:");

     /* -- lesen */
     for (i=0;i< MAXENTRIES; i++)
          scanf ("%d", &feld[i]);


     qsort ((void*) feld, MAXENTRIES, sizeof(int), (int (*)())comp);


     puts("\nNach dem sortieren:\n");

     /* -- ausgeben */
```

```
     for (i=0;i< MAXENTRIES; i++)
          printf("\n%d", feld[i]);

     return 0;
} /* main*/


/* Die Vergleichsfunktion */
int comp(const void *a, const void* b){
     int* x= (int*) a;
     int* y= (int*) b;
     return (*x - *y);
}
```

1.4.7. Zeitmessung

Eine häufig gestellte Frage lautet: Wie lange braucht der Algorithmus? Sie können dies mit folgender Funktion herausfinden:

```
#include <time.h>
clock_t clock();
```

Diese Funktion liefert die verbrauchte CPU-Zeit seit dem Programmstart zurück. Falls die CPU-Zeit nicht verfügbar ist, gibt diese Funktion -1 zurück. Wenn Sie die CPU-Zeit in Sekunden benötigen, muss der Rückgabewert dieser Funktion durch CLOCKS_PER_SEC dividiert werden.

1.4.8. Beispiel: zeitmessung.c – clock() und clock_t

```
// zeitmessung.c
//

#include <stdio.h>
#include <time.h>

int main()
{
   clock_t prgstart, prgende;
   int c;

   prgstart=clock();

   printf("Bitte geben Sie irgendetwas ein und "
          "beenden Sie mit '#'\n");
   printf("\n > ");

   while((c=getchar())!= '#')
      putchar(c);
```

```
   prgende=clock();

   printf("Die Programmlaufzeit betrug %.2f Sekunden\n",
            (float)(prgende-prgstart) / CLOCKS_PER_SEC);
   return 0;
}
```

1.4.9. Beispiel: VisualSort (in Java)

Erstellen sie ein Programm zum Visualisieren folg. Algorithmen:
☑ BubbleSort
☑ SelektionSort
☑ QuickSort

Fügen Sie einen Button mischen hinzu und zeigen sie die jeweils benötigte Zeit an

1.4.10. Beispiel: zeitSortieren.c

Sie sollen die Sortieralgorithmen analysieren. Es soll ein Programm erstellt werden, mit dem drei verschiedene Zustände von Daten sortiert werden.

☑ Zuerst sollen Daten sortiert werden, bei denen das größte Element ganz am Anfang ist und absteigend das kleinste Element ganz am Ende.

☑ Anschließend sollen Daten sortiert werden, die bereits in sortierter Form vorliegen.

☑ Im letzten Beispiel werden Daten sortiert, die mit Zufallsdaten belegt werden.

Die Anzahl der Elemente ist in einem solchen Fall natürlich auch entscheidend. Es werden dafür
☑ 10000,
☑ 100000 und
☑ 1000000 Elemente
verwendet, die nach den vorhandenen Zuständen sortiert werden sollen.

Es ist nur die Ausgabe des Programms von Interesse. Leiten Sie die Standardausgabe am besten in eine Textdatei um, indem Sie im Programm noch vor der for-Schleife in der main()- Funktion Folgendes eingeben:

freopen("benchmark.txt","a+",stdout);

```
#define MAX 100000

/*Ein Array von großen zu kleinen Werten*/
int test_array[MAX];

void init_test_array(int elements)
{
```

```
   int i,j;
   for(i=elements-1,j=0; i>=0; i--,j++)
      test_array[j]=i;
}

/*Ein bereits sortiertes Array*/
void init_test_array2(int elements)
{
   int i;
   for(i=0; i<elements; i++)
      test_array[i]=i;
}

/*Ein Array mit (Pseudo)-Zufallszahlen*/
void init_test_array3(int elements)
{
   int i;
   for(i=0; i<elements; i++)
      test_array[i]=rand();
}
```

Folgende Tabelle soll ausgegeben werden.

Zeitanalyse von Sortieralgorithmen:

Anzahl	**Datenzustand**	**Selektion**	**Bubble**	**myQsort**	**qsort**
10000	aufsteigend	??.??			
10000	absteigend				
10000	sortiert				
100000	aufsteigend	??.??			
100000	absteigend				
100000	sortiert				
1000000	aufsteigend	??.??			
1000000	absteigend				
1000000	sortiert				

Mithilfe dieser Analyse können Sie sich nun ein etwas detaillierteres Bild von der Effizienz der einzelnen Algorithmen machen. Natürlich sollten Sie diese Laufzeitmessung nicht allzu genau nehmen. Für eine exaktere und genauere Messung sollten Sie auf jeden Fall einen Profiler einsetzen. Denn das Programm zur Laufzeitmessung ist während der Ausführung sicherlich nicht das einzige Programm, welches gerade auf Ihrem System läuft.

1.4.11. +Exkurs: Profiler

Für kleinere Programme können Sie eine Laufzeitmessung mit der Funktion clock() vornehmen. Für größere

und umfangreiche Projekte ist diese Funktion aber weniger geeignet. Bei solch speziellen Fällen sollten Sie extra Programme einsetzen, die für diese geschrieben wurden, so genannte Profiler.

Ein Profiler ist ein eigenständiges Programm, welches Sie zur Laufzeitanalyse verwenden können. Der Vorteil dieses Werkzeugs ist, dass Sie mit ihm auch einzelne Funktionen analysieren können. So lässt sich schnell herausfinden, welcher Teil des Quellcodes mehr Zeit als gewöhnlich beansprucht. Ein Profiler ist ebenfalls ein Standardwerkzeug für Codetuning-Freaks.

Bei den kommerziellen Entwicklungsumgebungen ist der Profiler im Normalfall mit dabei. Es gibt aber auch einen kostenlosen Kommandozeilen-Profiler, den GNU-Profiler gprof, der für alle gängigen Systeme erhältlich ist.

☑ Als Erstes benötigen Sie einen fehlerfreien Quellcode, den Sie analysieren wollen. Dann müssen Sie den Quellcode mit dem Compilerschalter -pg übersetzen:

```
gcc -pg programmname.c
```

Jetzt befindet sich im Verzeichnis eine Datei namens "a.out" (unter Windows/MS-DOS auch "a.exe"). Diese Datei ist die ausführbare Datei für Ihren Quellcode.

☑ Starten Sie jetzt das ausführbare Programm "a.out".

```
./a.out
```

Nun werden die Profiling-Informationen in die Datei "gmon.out" geschrieben, die sich jetzt ebenfalls im Verzeichnis befindet.

☑ Nach Programmende können Sie gprof zur Auswertung der Datei "gmon.out" aufrufen. Die Ausgabe, welche häufig etwas länger ist, leiten Sie am besten in eine Datei um:

```
gprof ./a.out > test_prof.txt
```

Die Textdatei "test_prof.txt" können Sie jetzt mit einem Editor Ihrer Wahl öffnen. Diese Datei beinhaltet wiederum zwei Dateien. Der erste Teil nennt

- die verbrauchte Rechenzeit der Funktionen
- die Anzahl der Aufrufe von Funktionen

Im zweiten Teil befindet sich die Verteilung der Rechenzeit von Funktionen, auf die von ihnen aufgerufenen Unterfunktionen. Mehr zum Werkzeug gprof erfahren Sie in der entsprechenden Dokumentation.

1.5. Suchen

1.5.1. Beispiel: seqSearch.c - Sequentielles Suchen

Ähnlich dem Sortieren gibt es ein sehr einfaches Suchverfahren, bei dem Arrayelemente Position für Position mit dem Suchbegriff verglichen werden. Dies könnte wie folgt geschehen:

```
/*
 * seq_search(int a[], int n, int k) - durchsucht das Feld a[]
 *   (mit n Elementen) nach dem Schluessel k. Der
 *   Rueckgabewert ist die Position im Feld oder -1, wenn nicht
 *   vorhanden.
 */
int seq_search(int a[], int n, int k){
     int i;

     for( i=0; i<n; i++ ) {
          if( a[i] == k )
               return i;
     }

     return -1;
}
```

Sequentielles Suchen benötigt N+1 Vergleiche für eine nicht erfolgreiche Suche und im Mittel N/2 Vergleiche, für eine erfolgreiche Suche.

1.5.2. Beispiel: binSearch.c - Binäres Suchen

Statt ein geordnetes Array von vorne zu durchsuchen gibt es ein effizienteres Verfahren:

1. Vergleich des Schlüssels mit dem Element in der Mitte der Liste.
2. Falls der Schlüssel größer ist als das Element in der Mitte, wiederholt man die Suche mit der linken Hälfte.
3. Falls der Schlüssel kleiner ist, wiederholt man die Suche mit der rechten Hälfte.

Man nennt es binäres Suchen.

```
/*
 * binsearch(int a[], int left, int right, int key) - durchsucht das
 *   Array a[] in den Grenzen left bis right nach dem Wert key.
 *   Die Position ist der Rueckgabewert, oder -1
 *   wenn key nicht gefunden wurde.
 */
int binsearch(int a[], int left, int right, int key)
{
     int m;

     if( left <= right ) {
          m = (left+right)/2;

          if( key == a[m] )
               return m;

          else if( key < a[m] )
               return binsearch(a, left, m-1, key);
          else
               return binsearch(a, m+1, right, key);
     }
```

```
    return -1;
}
```

Binäres Suchen benötigt nie mehr als lg n + 1 Vergleiche.

Aufgabe: binsearch.c

Schreiben Sie obige rekursive Version der binären Suche um in eine iterative Version.
Erstellen Sie ein kl. Testprogramm.

1.5.3. +Eine Binäre Suche für beliebige Arrayelemente

Das folgende Beispiel zeigt, wie mittels Zeiger auf Funktionen die Binäre Suche für Arrays beliebigen Datentyps programmiert werden kann.

Lediglich der Vergleich zweier Arrayelemente ist Datentyp abhängig, daher verwendet man für diese Operation einen Zeiger auf die Vergleichsfunktion. Dieser Zeiger wird beim Aufruf der Funktion bin_suche() mit übergeben.

```
// anton hofmann sept. 2002
// t_ my_bin_suche.c
//
// Nachprogrammieren der C-Standardbibliotheksfunktion
// void* bsearch(const void*, const void*, size_t, size_t,
//                 int (*)(const void*, const void*));
//
// Binäres Suchen in einem Array.
// Demo: Bin.Suche und Zeiger auf Funktionen
//

#include <stdio.h>
#include <stlib.h>

// ------ forward Deklararionen
void* bin_suche (void * key, void * anArray, int nelements, int nsize,
                int (*cmp)());

// -- Vergleichsfunktion
int compare (int*, int *);


#define MAXENTRIES 10
int arr[MAXENTRIES];

int main(int argc, char *argv[])
{
 int suche;  // Wert nach dem gesucht werden soll
 int *iptr;  // Return wert von bin_suche
 int i;

 puts("\nTeste qsort und bin_suche:\nBitte int-zahlen eingeben: ");

 for (i=0;i< MAXENTRIES; i++)
     scanf ("%d", &arr[i]);

// ---------- sortieren mit der C-Standardbibliotheksfunktion
  qsort ((void*) arr, MAXENTRIES, sizeof(int), (int (*)())compare);

  puts("\nNach dem sortieren:\n");
  for (i=0;i< MAXENTRIES; i++)
      printf("\n%d", feld[i]);

// ---------- bin_suche testen
   puts ("\nbin_suche testen: key eingeben(ende=0):\n");
   scanf("%d", &suche);
   while (suche!=0){
         iptr= (int*) bin_suche ((void*) &suche, (void*)arr, MAXENTRIES,
                                 sizeof(int), (int (*)()) compare);
         if(iptr == NULL)
            printf("\nNicht gefunden");
         else
            printf("\n%d Gefunden", *iptr);
```

```
         puts ("\nkey eingeben(ende=0):\n");
         scanf("%d",&suche);
   } // while
} //main


int compare (int* a, int* b)
{
 return (*a - *b);
}




// bin_suche
// Array muss sortiert sein
// return: NULL wenn nicht gefunden
//         ptr des gefundenen Objektes, wenn gefunden
// keine Seiteneffekte

void*
bin_suche (void * key, void * anArray, int nelements, int nsize,
           int (*cmp)())
{
 int l= 0;
 int r= nelements - 1;
 int m;

 while (l<=r) {
       m= (l+r)/2;

       if ( (*cmp)(key, anArray + m*nsize) == 0)
          return (anArray + m*nsize);

       else if ((*cmp)(key, anArray + m*nsize) < 0)
           r= m - 1;
       else
           l= m + 1;
     }
     return NULL;
}
```

1.6. String Matching

Wird hier nicht behandelt.

1.7. Pattern Matching

Wird hier nicht behandelt. Siehe REGEX

1.8. Backtracking - (rücksetzen)

Quelle: http://de.wikipedia.org/wiki/Backtracking

Der Begriff Rücksetzverfahren oder englisch **Backtracking** (engl. Rückverfolgung) bezeichnet eine **Problemlösungsmethode** innerhalb der Algorithmik.

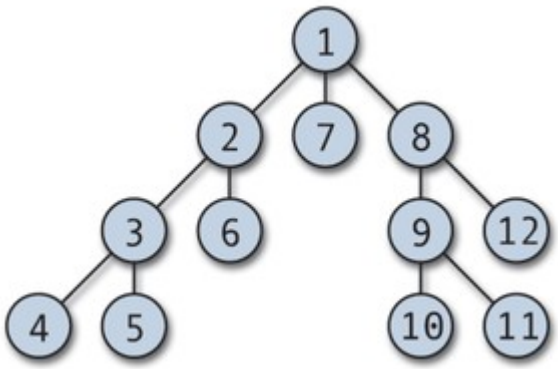
Backtracking geht nach dem **Versuch-und-Irrtum**-Prinzip (trial and error) vor, d.h. es wird versucht, eine erreichte **Teillösung schrittweise zu einer Gesamtlösung auszubauen.**

Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt bzw. die letzten **Schritte zurückgenommen**, und es werden stattdessen **alternative Wege probiert**.

Auf diese Weise ist sichergestellt, dass **alle in Frage kommenden Lösungswege ausprobiert** werden können.

Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert.

Backtracking wird **meistens am einfachsten rekursiv** implementiert und ist ein prototypischer Anwendungsfall von Rekursion.

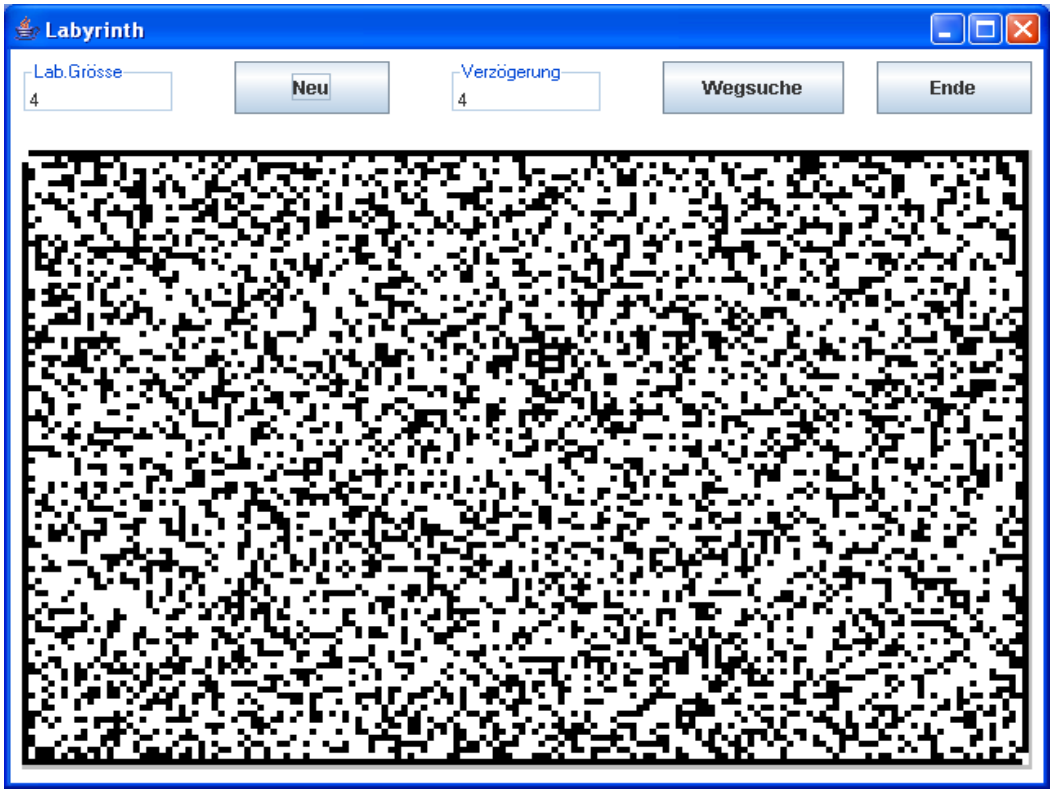


1.8.1. Wegsuche: Labyrinth

Backtracking wird auch eingesetzt für die Wegsuche von A nach B in einem Graph, z.B.

- für die Suche nach Verbindungen in einem Fahrplan oder
- zum Bestimmen einer Route in einem Routenplaner oder

- eines Weges durch ein Labyrinth.



Hier ein allg. Algorithmus zur Suche in einem Labyrinth (rekursiv formuliert):

Der erste Aufruf: Wir beginnen links, oben

```
int gefunden= wegSuche(0,0);
```

Wir wollen nun wegSuche() so programmieren, dass sie rekursiv folg. versucht:

```
☑markiere aktuellen Platz als "guten Weg", der zum Ziel führt
☑if (sucheWeg(OBEN)==1) return 1;
  else if (sucheWeg(LINKS)==1) return 1;
  else if (sucheWeg(UNTEN)==1) return 1;
  else if (sucheWeg(RECHTS)==1) return 1;
☑markiere aktuellen Platz als "schlechten Weg", der nicht zum Ziel
  führt und return -1;
```

Die obige Version muss noch mit Abbruchskriterien für die Rekursion versehen werden.

```
          // Abbruch, wenn ausserhalb des Labs
          ...
          // Abbruch, wenn Wand
```

```
          ...
          // Abbruch, wenn ich schon da war,
          ...
          //und führt nicht zum ziel
          if (labyrinth[x][y]== Color.red)
               return -1;

          // Abbruch, wenn ich schon da war,
          //und führt zum ziel
          ...

          //Abbruch, wir sind nun am Ziel
          ...
```

Nun etwas detaillierter für Java:

Zunächst der Aufruf der Funktion int wegSuche(0,0) , hier als eigener Thread.

```
java.lang.Runnable doWegSucheRunnable= new Runnable()
     {
         public void run()
         {
          try
          {
               int gefunden;

               gefunden= wegSuche(0,0);
               if (gefunden==-1)
                    JOptionPane.showMessageDialog(null,
                         "Nicht gefunden!");
               else
                    JOptionPane.showMessageDialog(null, "Gefunden!");

          }catch(Exception ex)
          {
               ex.printStackTrace(System.out);
          }
     }
};
```

```
     int wegSuche(int x, int y) throws InterruptedException{
          zeichneLab();

          // ------------------------------------
          // REKURSIONSABBRUCH; REKURSIONSABBRUCH
          // ------------------------------------

          // Abbruch, wenn ausserhalb des Labs
          if (x <0 || x >= spalten)
               return -1;
```

```
        if (y <0 || y >= zeilen)
             return -1;

        // Abbruch, wenn Wand
        if (labyrinth[x][y]== Color.black) // Wand
             return -1;

        // Abbruch, wenn ich schon da war,
        //und führt nicht zum ziel
        if (labyrinth[x][y]== Color.red)
             return -1;

        // Abbruch, wenn ich schon da war,
        //und führt nicht zum ziel
        if (labyrinth[x][y]== Color.green)
             return -1;

        //Abbruch, wir sind nun am Ziel
        if (x == spalten-1 && y== zeilen -1){
             JOptionPane.showMessageDialog(this, "AM ZIEL!!!!");
             return 1;
        }


        // -------------------------------------
        // REKURSION
        // -------------------------------------
        // suche einen neuen Weg

        // markiere momentanen Weg
        labyrinth[x][y]= Color.green;

        // suche einen Weg durch das Lab

        //---------OBEN
        if (wegSuche(x, y-1)== 1) {
             return 1;
        }
        //---------RECHTS
        else if (wegSuche(x+1, y)== 1) {
             return 1;
        }
        //---------UNTEN
        else if (wegSuche(x, y+1)== 1) {
             return 1;
        }
        //---------LINKS
        else if (wegSuche(x-1, y)== 1) {
             return 1;
        }

        // wenn wir bis hierher kommen gibts keinen weg
        labyrinth[x][y]= Color.red;
        return -1;
```

```
    }
```

1.8.2. Aufgabe: Labyrinth

Bringen Sie das folgende Java-Programm zum Laufen:

DS-ALGO/01-algorithmen/02-ueben/TINU-03-backtracking/GrfLabyrinthUE.zip

1.8.3. 8-Damen Problem

Auch beim folgenden Beispiel kann man Backtracking verwenden:

AUFGABE: 8 DAMEN
Gesucht ist eine Konfiguration von 8 Damen auf einem 8x8 Schachbrett,
sodass keine Dame eine andere bedroht.

LÖSUNG:

```
PAP:
void BT(int i)  // 1...8 : i-te Dame
     int k;

     for(k=1..8)
          if (feld[i][k] nicht bedroht)
               Setze feld[i][k] besetzt

               if(Brett ist voll)
                    Gib Lösung aus; exit
               else
                    BT(i+1)

               Nimm dame vom feld[i][k]
```

1.8.4. Aufgabe: BackTracking-DAMEN.c

Bringen Sie das folgende Programm zum Laufen.

```
/*
 ============================================================================
 Name        : BackTracking-DAMEN.c
 Author      : N.N
 Version     :
 Copyright   : open source
 Description : s.u.
 ============================================================================
 */
```

```
/*
AUFGABE: 8 DAMEN PROBLEM
Gesucht ist eine Konfiguration von 8 Damen auf einem 8x8 Schachbrett,
sodass keine Dame eine andere bedroht.


PAP:
void BT(int i)  // 1...8 : i-te Dame
     int k;

     for(k=1..8)
          if (feld[i][k] nicht bedroht)
               Setze feld[i][k] besetzt

               if(Brett ist voll)
                    Gib Lösung aus; exit
               else
                    BT(i+1)

               Nimm dame vom feld[i][k]

*/


#include <stdio.h>
#include <stdlib.h>

void BT(int i);
void display();

#define DIM 8
int feld[DIM][DIM];

int main(void) {

     puts("8 Damen: Backtracking");

     // .....

     return EXIT_SUCCESS;
}
```

1.8.5. Aufgabe: Sudoku-Puzzles

Auch das Sudoku-Puzzle kann man mit Backtracking-Techniken lösen. Löse dazu die Aufgaben in

DS-ALGO/01-algorithmen/02-ueben/TINU-03-backtracking/u-permutationen.odt

1.9. Ausblick

In den folgenden Kapiteln wollen wir die Verwendung bestehender Datenstrukturen und Algorithmen in diversen Frameworks untersuchen:

☑ C, C++ und die STL (Standard-Template-Library)

☑ Java Collection