Effiziente, stabile Sortierverfahren

Autor: Dipl.-Ing. Siegfried Sielaff

SwapSort

Inhalt:

Vorwort
I Stabile SwapSort - Sortieralgorithmen
1. Einfaches SwapSort
2. SwapSort mit zyklischem Verschieben
3. SwapSort mit zyklischem Verschieben am Stack
4. SwapSort mit zyklischem Verschieben am durch log(n) Stapelelemente begrenzten Stack
II Zyklisches Verschieben von sortierten Teilfolgen
III SwapSort als Interface für andere effiziente Sortieralgorithmen und/oder Mischverfahren
IV SwapSort als Lastaufteilungsverfahren für Multiprozessorsysteme
1. Multitasking bzw. Multithreading mit SwapSort
2. Verteilung von Fibers an Tasks
V Effizienzbetrachtungen von SwapSort

Vorwort

In diesem Artikel präsentiere ich einen mir bis dato nicht vorgekommenen Sortieralgorithmus, ich nenne ihn SwapSort, der außerordentlich effizient ist und dazu noch stabil sortiert. Ein Algorithmus **sortiert stabil**, wenn bei Gleichheit der Schlüssel zweier Elemente, die Reihenfolge der Elemente nicht verändert wird.

Da es eine technisch komplizierte Materie ist, habe ich mich sehr bemüht die Erkenntnisse genau darzulegen. Ich verwende auch eine C-ähnliche Syntax zur Beschreibung von algorithmischen Abläufen, wobei aber alles Wesentliche auch in den Kommentaren oder im Text steht.

In der Geschichte der Entwicklung von Sortieralgorithmen gab es zuerst 2 gut bekannte Sortieralgorithmen für lineare Felder, bekannt als **BubbleSort** (Sortieren durch Vertauschen) und **SortByStraightInsertion** (Sortieren durch Einfügen), welche einen Speicherbereich O(n) und eine Zeitkomplexität von O(n*n) beanspruchen. Ein anderer Algorithmus bekannt als **MergeSort** (Sortieren durch Mischen) verwendet einen Speicherbereich von O(n), hat aber eine Zeitkomplexität von O(n log(n)).

SwapSort ist ein Algorithmus, der n Elemente auf einen Computer oder in einer Apparatur, sowie in den obengenannten bekannten Algorithmen sortiert, nur mit dem entscheidenden Unterschied, dass er mit zusätzlichen verwendeten Speicher sparsam umgeht und i.a. schneller abläuft, da er weniger Bearbeitungsschritte ausführt.

Zuerst werde ich zeigen, dass dieser Algorithmus eine Verbesserung der oben erwähnten Algorithmen ist.

BubbleSort ist einfach. Wir nehmen 2 benachbarte Elemente, vergleichen sie und tauschen sie aus, falls das 2. Element kleiner als das 1. ist. In C-ähnlicher Syntax, wobei a[] ein lineares Feld von a[0] bis a[n-1] ist:

```
do for(b=0,i=1;i<n;i++) if (a[i]<a[i-1]) { v=a[i]; a[i]=a[i-1]; a[i-1]=v; b=1; } while(b); // BubbleSort
```

Dieser Algorithmus hat einen außerordentlichen Nachteil, man muss das Array (=lineares Feld) solange durchlaufen bis keine Vertauschungen mehr durchgeführt werden. Seine Verbesserung SortByStraightInsertion vergleicht 2 benachbarte Elemente und falls das 2. Element kleiner als das 1. ist, dann bewegt es sich solange nach vorne, so dass alle Elemente bei denen es sich vorbei bewegte größer als dieses Element sind und alle Elemente vor diesem Element kleiner oder gleich diesem Element sind. In C-ähnlicher Syntax:

```
for(i=1;i<n;i++) if (a[i]<a[i-1]) { for(j=i-1;j&&(a[i]<a[j-1]);j--); v=a[i]; for(k=i;k>j;k--) a[k]=a[k-1]; a[j]=v; }
```

Nun verbessert man diesen Algorithmus noch einmal. Es gibt in diesem viel zu viele unnotwendige Speicherbewegungen von links nach rechts. Es würde viel besser sein, wenn man mehr als ein Element auf den richtigen Platz platzieren könnte. In diesem Fall sammelt man mehr als ein Element und stellt diese in einem Durchgang an die richtigen Plätze. Man kann dies tun, in dem man auch den rechten Teil des Arrays sortiert. Dann nämlich kennt man alle Elemente vom rechten Teil, welche man im linken Teil des Arrays einfügen kann. Das sind zumindest alle Elemente, die kleiner als das größte Element des 1. Teilarrays sind. Und dann kann man alle einzufügenden Elemente in einem einzigen Durchgang oder auch in mehreren Durchgängen einfügen. In C-ähnlicher Syntax:

```
for(i=1;i<n;i++) if(a[i]<a[i-1]) { // jetzt ist a[0] bis a[i-1] bereits sortiert
        Sort(a[],i,n-1);                    // sortiert Teilarray von a[i] bis a[n-1]
        Merge(a[],0,i-1,i,n-1); // mische 2 sortierte Teilarrays, indem wir alle Elemente des 2. Teiles des Arrays
break; }                                // mit a[j]<a[i-1],j=i,...,n-1 in dem 1. Teil des Arrays einfügen
```

Da das Sortieren des 2. Teilarrays i.a. noch sehr aufwendig ist, lösen wir die Gesamtsortieraufgabe durch Aufteilung in kleinere Teilprobleme wie z.B.:

```
SubOfSort(a[],f,k,l) {
for(i=f;i<l && a[i+01]>a[i];i++);                   // erzeugt ein sortiertes Teilarray von a[f] bis a[i]
(*)         for(;i<k;i=j) {                        // hier ist a[f] bis a[i] bereits sortiert
                    j=((l-(i+01))<i-f)?l:i+(i-f)+01;
(**)                j=SubOfSort(a[i],i+01,j,l);    // sortiert Teilarray von a[i+1] bis a[j]
(***)               Merge(a[],f,i,i+01,j); }       // mischt 2 benachbarte sortierte Teilarrays
            return i; }
Sort(a[],f,l) { SubOfSort(a[],f,l,l); }
```

Oder so:

```
Sort(a[],f,l) { if (f<l) { k=f+(l-f)/2;            // teile in mindestens 2 Teilarrays
(*)       Sort(a[],f,k);                           // gibt ein sortiertes Teilarrays zurück
(**)      Sort(a[],k+01,l);                        // gibt noch ein weiteres sortiertes Teilarray zurück
(***)     Merge(a[],f,k,k+01,l); }}                // mischt 2 benachbarte sortierte Teilarrays
```

Nun sieht man, dass ein Spezialfall dieser Verbesserung - nämlich wenn alle im 1. Teilarray einzufügende Elemente vom 2. Teilarray in einem Durchgang im 1. Teilarray eingefügt werden - dem Sortieralgorithmus MergeSort entspricht. MergeSort leistet in den Schritten (*) bis (**) das selbe, welches der vorhergehende Algorithmus ebenfalls tut. Er teilt das lineare Array in 2 sortierte Teile. Aber MergeSort mischt in Schritt (***) die sortierten Teile des Arrays auf eine sehr aufwendige Art, indem es ein zusätzliches Array verwendet. In C-ähnlicher Syntax:

```
MergeOfMergeSort(a[],f,k,j,l) { i=f, m=0;
        while(i<=k)
                if (a[i]<=a[j]) if (m==0) i++,f++; else z[m++]=a[i++];
                else { z[m++]=a[j]; if (j++==l) break; }
         if (i<=k) for(;i<=k;i++,m++) z[m]=a[i];       // mische alles in ein zusätzliches lineares Feld z[]
         for(j=0;j<m;j++) a[f++]=z[j]; }               // kopiere m Elemente vom linearen Feld z[] zurück
```

Vorher wurde das Mischen folgendermaßen beschrieben: 2 sortierte Teilarrays mischt man, indem man alle Elemente des 2. Teilarrays, welche kleiner sind als das größte Element des 1. Teilarrays (aus dem 2. Teilarray entfernen) und in das 1. Teilarray einfügt. Bisher haben wir implizit angenommen, wenn wir 1 oder mehrere Elemente vom 2. Teilarray entfernen, so sind es stets die kleinsten Elemente und stehen somit im sortierten Teilarray auf der linken Seite, d.h. das Teilarray verkleinert sich somit indem man den untersten Index um die Anzahl der entnommenen Elemente erhöht. Hat man Elemente aus dem 2. Teilarray entnommen, so hat man Platz geschaffen, um das 1. Teilarray um genau die Anzahl der im 2. Teilarray entnommenen Elemente vergrößern zu können. D.h. man kann die aus dem 2. Teilarray entnommenen Elemente im 1. Teilarray einfügen. Dies kann man auf eine spezielle Art machen, indem man nicht wie üblich von vorne nach hinten bzw. von links nach rechts, sondern andersherum mischt, d.h. man vergleicht zuerst die hinteren dann die vorderen Elemente bzw. zuerst die rechten dann die linken Elemente. Und so kann man die entstandenen Leerplätze auf natürliche Weise auffüllen. Nun wird klar, dass man bei dieser Art von Mischen die Speichergröße, die man verwenden will, fest beschränken kann. Hat man nicht genug Speicher, um alle aus dem 2. Teilarray einzufügenden Elemente, in das 1. Teilarray mischen zu können, dann kann man den Vorgang solange wiederholen, bis man alle Elemente eingefügt hat. Man kann bei dieser Vorgehensweise des Mischens verallgemeinern, dass der Zeitaufwand beim Sortieren steigt, wenn wir weniger Speicherplatz zur Verfügung stellen.

Nun kann man diese Art des Mischens nochmals verbessern, so dass wir einen sehr leistungsfähigen Sortieralgorithmus erhalten, der bis auf einen Stack von O(log(n)) keinen zusätzlichen Speicher benötigt. Dazu wollen wir zuerst den oben beschriebenen Algorithmus anders formulieren. Wir können auch sagen: Wir mischen 2 sortierte Teilarrays, indem wir alle Elemente des 2. Teilarrays, welche kleiner sind als das größte Element des 1. Teilarrays, aus dem 2. Teilarray entfernen und mit dem 1. Teilarray im 1. Teilarray mischen. Nun bisher haben wir angenommen, dass sich die Größe eines Teilarrays noch verändern darf. Jetzt nehmen wir an, dass sich die Größe eines Teilarrays nicht mehr verändert. Dann sieht es so aus: Man mischt das 1. und das 2. sortierte Teilarray ins 1. Teilarray und mischt dann die übriggebliebenen Elemente des 1. und 2. Teilarrays ins 2. Teilarray. Egal ob man die Größe eines Arrays verändern lässt oder nicht, wir erhalten jede Mal ein sortiertes Array. Nur mit einem wesentlichen Unterschied: Man hat 1 Mischproblem in 2 Mischprobleme aufgeteilt. Und hat man erst mehrere Mischprobleme, so unterteilt man sie wieder weiter auf, solange bis man kein Mischproblem mehr aufteilen kann bzw. aufteilen braucht, da das zugehörige Teilarray bereits sortiert ist. Und nun könnte man fragen, wo bleibt hier eigentlich das Mischen, wenn man immer nur neue Mischprobleme erzeugt und zerkleinert, bis es nicht mehr nötig ist, ein sortiertes Array weiter zu unterteilen? Daher nenne ich diesen Algorithmus auch **SwapSort**, denn die Hauptaufgabe beim Aufteilen in neue Mischprobleme ist das Austauschen und zyklische Verschieben von Speicherbereichen.

Kapitel I: Stabile SwapSort - Sortieralgorithmen

1. Einfaches SwapSort

Einfaches SwapSort lässt sich in 3 verschiedene Phasen unterteilen - **Aufteilung** der Sortieraufgabe in kleinere Subsortieraufgaben, anschließendem **Swappen** (Austauschen von Speicherbereichen) und erneutem **Mischen** der sortierten Teilfolgen. Für jede einzelne Phase bietet sich das Programmierparadigma **divide & conquer** (teile und beherrsche) an. D.h. jede Phase kann durch Aufteilen auf kleinere Probleme und anschließendem Lösen der Teilprobleme gelöst werden. Die 1. Phase ist selbst lediglich nur eine Problemaufteilung.

```
SwapSort(a[],f,l) {
        if(f<l) {
                // Aufteilung - Phase 1 oder Vorsortierphase
                k=f+(l-f)/2;                    // Aufteilung des Problems in kleinere Teile
                SwapSort(a[],f,k);              // sortiert 1. Teil des Problems
                SwapSort(a[],k+1,l);            // sortiert 2. Teil des Problems

                // MergeOfSwapSort() - swappt (Phase 2) und mischt (Phase 3)
                MergeOfSwapSort(a[],f,k,k+1,l); // mischen zweier sortierter Arrays
        }}
```

MergeOfSwapSort() misht, die in der Aufteilungsphase generierten Teilprobleme. Es verwendet folgendes Problemaufteilungsverfahren: Zwei sortierte Teilfolgen einer Folge werden so miteinander gemischt, dass man eine genau zu berechnende Zahl n von Elementen einer Teilfolge, die letzten n Elemente der 1. Teilfolge sowie die ersten n Elemente der 2. Teilfolge, miteinander austauscht (swappt), so dass nachher alle Elemente der 2. Teilfolge größer oder gleich der 1. Teilfolge sind. Mischt man dann die verbliebenen Elemente der 1. Teilfolge mit den neu hinzugefügten und die verbliebenen Elemente der 2. Teilfolge mit den neu hinzugefügten, dann erhalten wir 2 sortierte Teilfolgen, wobei die 2. Teilfolge ausschließlich Elemente enthält, die größer oder gleich aller Elemente der 1. Teilfolge sind, und somit ist bei Aneinanderreihung der 1. und der 2. Teilfolge die Gesamtfolge sortiert.

Zur Veranschaulichung teilen wir ein Array in 4 Teile.

Gesamtfolge			
entspricht der 1. Sortierten Teilfolge		entspricht der 2. sortierten Teilfolge	
Teil 1	Teil 2	Teil 3	Teil 4
	Es gibt exakt n Elemente in diesen Teil des Arrays, jedes Element in diesem Teil ist >= als jedes Element von Teil 1 und > als jedes Element von Teil 3.	Es gibt exakt n Elemente in diesem Teil des Arrays, jedes Element in diesem Teil ist < als jedes Element von Teil 2 und <= als jedes Element von Teil 4.	

Die Zahl n >= 0 lässt sich exakt berechnen:

```
Calculate_n(a[],a1,a2,b1,b2) {
        n=0; do if (a[a2]>a[b1]) n++; else break; while (a1!=a2-- && b1++!=b2);
        return n;}
```

Da es bei dieser Berechnung sehr auf die Effizienz ankommt, verwenden wir hier ein Berechnungsverfahren, welches mit ca. log(Minimum(Anzahl der Elemente einer der beiden Teilfolgen)) Berechnungsschritten auskommt:

```
Calculate_n(a[],a1,a2,b1,b2){ // berechnet wie viele Elemente eingefügt werden sollen mittels
                              // eines Verfahrens, welches auf Intervallteilung beruht
u=0; if (a2-a1<b2-b1) o=a2-a1; else o=b2-b1; o++;   // Zuweisung der Intervallgrenzen
do{
        n=u+(o-u)/2;                              // Intervallhalbierung
        if (a[a2-n]<=a[b1+n]) o=k; else u=++n;    // Einengung des Intervalls
}
while (u!=o);                                     // Schleifenabbruch, wenn die Zahl n bestimmt wurde
return n; }
```

Wir nehmen nun alle n Elemente von Teil 3 und mischen diese Elemente mit Teil 1 und dann nehmen wir alle n Elemente von Teil 2 und mischen diese Elemente mit Teil 4. Dann erhalten wir ein sortiertes Array:

```
MergeOfSwapSort(a[],i,j,k,l){
n=Calculate_n(a[],i,j,k,l);  // berechnet wie viele Elemente ausgetauscht werden sollen
                             // = Anzahl der Elemente von Teil 2 bzw. Teil 3
if (!n--){                   // nur wenn noch nicht sortiert, tue  =>
                             // (n wird der Effizenz wegen um 1 reduziert)
        // Austauschen - Phase 2
        Swap(a[],j-n,j,k,k+n);                            // swap (tausche aus) Teil 2 mit Teil 3

        // Mischen - Phase 3
        if(i!=j-n) MergeOfSwapSort (a[],i,j-n-1,j-n,j);      // mischt Teil 1 mit Teil 3
        if(l!=k+n) MergeOfSwapSort (a[],k,k+n,k+n+1,l);     // mischt  Teil 2 mit Teil 4
}}
```

Wie man sieht, wird das Mischen (Phase 3) durch rekursives Teilen des Problems gelöst. Für das Swappen (Phase 2 - Austauschen) des Speicherbereiches ist eine Aufteilung in Subaufgaben erst auf Multiprozessorsystemen sinnvoll. Da der Stack (auch Stapelspeicher genannt = Speicher der lokalen Variablen und Rücksprungadressen) beim Mischen, so klein wie möglich sein soll, um einen Überlauf zu verhindern, werden wir diesen auf log(Anzahl der Elemente der Gesamtfolge) Stapelelemente beschränken, indem wir das kleinere Teilproblem zuerst lösen und dann für das andere Teilproblem keinen neuen Funktionsaufruf durchführen, sondern in der selben Funktion bleiben, indem wir die Argumente der Funktion neu belegen und zur 1. Anweisung des Funktionskörpers zurückkehren.

```
MergeOfSwapSort(a[],i,j,k,l){ for(;;) { // Endlosschleife
n=Calculate_n(a[],i,j,k,l);
if (!n--){
        Swap(a[],j-n,j,k,k+n);

        // Mischen - Phase 3
        if (j-i<=l-k) { // löse kleineres Teilproblem zuerst
                // Array von i bis j ist kleineres Teilproblem
                if(i!=j-n) MergeOfSwapSort (a[],i,j-n-1,j-n,j);      // mischt Teil 1 mit Teil 3
                if (k+n==l) break;                                  // Schleifenabbruch
                i=k, k+=n, j=k, k++;       // setze Argumente der Funktion neu
                                           // ein erneuter Aufruf der Funktion mit den neu gesetzten
                                           // Argumenten würde Teil 2 mit Teil 4 mischen, deshalb
                                           // gehe zum Anfang zurück
        }else {
                // Array von k bis l ist kleineres Teilproblem
                if(l!=k+n) MergeOfSwapSort (a[],k,k+n,k+n+1,l);     // mischt  Teil 2 mit Teil 4
                if (i+n==j) break;                                  // Schleifenabbruch
                l=j, j-=n, k=j, j--;        // setze Argumente der Funktion neu
                                           // ein erneuter Aufruf der Funktion mit den neu gesetzten
                                           // Argumenten würde Teil 1 mit Teil 3 mischen, deshalb
                                           // gehe zum Anfang zurück
}}}}
```

2. SwapSort mit zyklischem Verschieben

SwapSort mit zyklischen Verschieben baut auf dem einfachen SwapSort-Sortieralgorithmus auf, wobei das Mischen in abgeänderter Weise erfolgt, um die Verschiebungen und Vergleiche zu reduzieren. Aber wenn man es anders versucht benötigt man zusätzlichen Speicher. So werden wir jetzt eine spezielle Art zu mischen betrachten und dann diese mit dem einfachen SwapSort-Algorithmus kombinieren. Man wählt eine gewisse Zahl m. Diese Zahl ist die Maximalanzahl der Elemente, welche man direkt einfügt bzw. die Maximalanzahl sortierter Folgen, welche eingefügt wird.

Nun betrachtet man 2 Fallunterscheidungen, die 2 sortierte Arrays mischen.

linke Folge		rechte Folge	
Teil 1	Teil 2	Teil 3	Teil 4

1. Fall: Man mischt Teil 2 mit Teil 4 von **links nach rechts**. Man vergleicht maximal m mal die **kleinsten Elemente** von Teil 2 und Teil 4 und nimmt das kleinere heraus und fügt es in einem neuen Teilarray ein.

			Rest von Teil 2	Rest von Teil 4
		maximal m sortierte Elemente von Teil 2 und Teil 4. Diese Elemente sind <= jenen Elementen von den Resten des Teils 2 und Teils 4.		

2. Fall: Man mischt Teil 1 mit Teil 3 von **rechts nach links**. Man vergleicht maximal m mal die **größten Elemente** von Teil 1 und Teil 3 und nimmt das größere heraus und fügt es in einem neuen Teilarray ein.

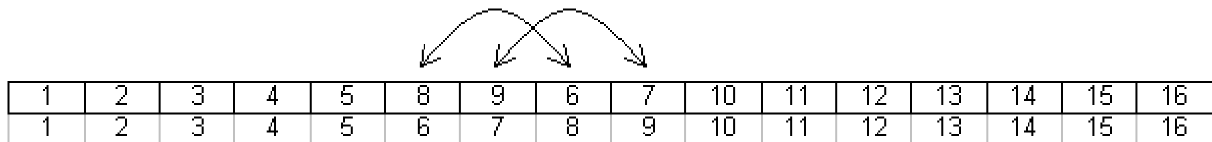
Rest von Teil 1	Rest von Teil 3			
		maximal m sortierte Elemente von Teil 1 und Teil 3. Diese Elemente sind >= jenen Elementen von den Resten des Teils 1 und Teils 3.		

In jedem Fall generiert man ein Array mit zusätzlich maximal m Elementen an ihren korrekten Plätzen. Und die Reste der Elemente, welche noch nicht aussortiert wurden, mischt man erneut. Und somit hat man die Kombination mit einfachem SwapSort hergestellt.

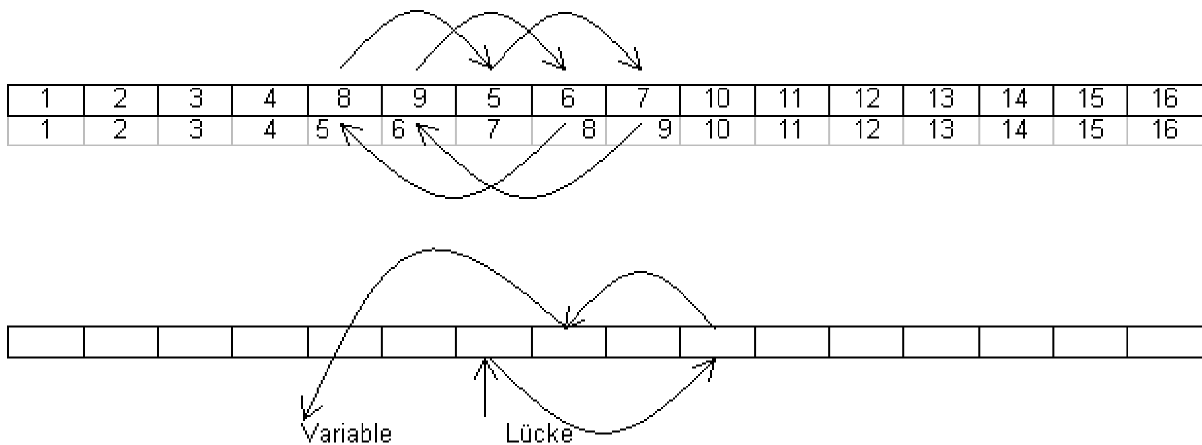
Rest von Teil 1	Rest von Teil 3	aussortierte Elemente	Rest von Teil 2	Rest von Teil 4
wird neu gemischt			wird neu gemischt	

Soeben wurde gezeigt, dass man durch Vorgabe einer Zahl m, bis zu 2*m Elemente direkt aussortieren und an die richtige Stelle setzen kann. Analoges gilt für sortierte Teilfolgen. In diesem Fall haben wir dann maximal 2*m Teilfolgen für die gilt: Alle Elemente einer Teilfolge mit Index i kleiner als Index j sind kleiner gleich jedem einzelnen Element der Teilfolge mit Index j.

Bei einfachen SwapSort haben wir die Elemente stets nur ausgetauscht.



Nun da man jetzt nicht mehr einfach wie bei einfachen SwapSort einen Speicherbereich einfach mit einem anderen austauscht, sondern einige Elemente beim Mischen gleich an ihren richtigen Plätzen einfügt, welche nicht mehr weiter gemischt werden, gibt es hier nicht nur einfache Zyklen. Wie z.B. den Zyklus 8, 6, 9, 7, 5, 8.



Beim zyklischen Verschieben rettet man ein Element vom Zyklus in einer Speichervariable. Dann hat man einen Platz, wo man ein Element hineinschreiben kann. Dann bewegt man jenes Element, das genau in diese Lücke gehört dorthin. Und dann hat man wieder eine freie Lücke. Diesen Vorgang kann man solange wiederholen, bis man das gerettete Element an seinen richtigen Platz platzieren kann.

Aber um eine minimale Anzahl von Speicherbewegungen zu erhalten, soll man gerade das Vorhandensein solcher Zyklen ausnützen. Denn zyklisches Verschieben ist viel effizienter als einfach nur Austauschen.

Swap(a,b) { c=a; b=a; b=c; } // man benötigt 3 Zuweisungen um 2 Elemente auszutauschen!

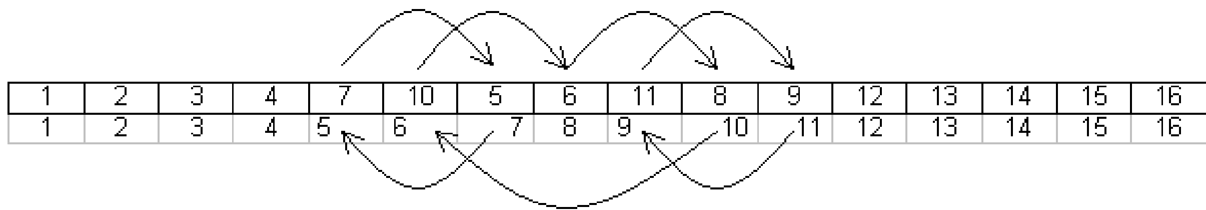
Hat man einen Zyklus mit n Elementen so gilt hingegen: Die Anzahl der Zuweisungen, um zyklisch zu verschieben, beträgt n+1. Bei der Abarbeitung eines Zyklus verwendet man ein von[] Feld, welches die Positionen enthält, von wo ein Element herkopiert wird. Die Indizes des von[] Feldes sind verschoben, so dass das 1. Element in der Indexliste jenes Element ist, welches im Feld a[] das Element mit Position j ist.

Swap(a[],von[],j) {v=a[j]; for(i=j,k=von[0];k!=j;i=k,k=von[k-j]) a[i]=a[k]; a[i]=v;} // von[] Indexliste des Zyklus

Für die Abarbeitung von einem Zyklus, wenn man Folgen aussortiert hat, verwendet noch zusätzlich ein nach[] Feld. Somit hat man die Position, wo man hineinkopieren kann. Man sieht, dass das nach[] Feld uns die Indizes des a[] Feldes liefert. Da a[nach[i]] bis a[nach[i+1]-1] einer sortierten Teilfolge entspricht, wenn das nach[] Feld sortiert wurde. Deshalb erstellen wir das nach[] Feld so, dass es in einer sortierten Reihenfolge vorliegt. Dies lässt sich auf natürliche Weise tun.

```
Swap(a[],von[],nach[],n) {                    // Abarbeitung für einen Zyklus aus der Verschiebungsliste
i=von[0]; l=k=nach[0]; d=a[k];                // rette erstes Element
do {    a[k]=a[i];                            // verschiebe
        j=binary_search(nach[],0,n-1,i);      // suche j, so dass nach[j]<=i<nach[j+1] gilt,
                                              // bzw. nach[j]<=i, falls nach[j+1] nicht existiert
        k=i; i=von[j]+(i-nach[j]);            // berechne nächsten Index
} while (i!=l);                               // wiederhole solange bis wir das gerettete Element
a[k]=d;}                                      // zurück schreiben können
```

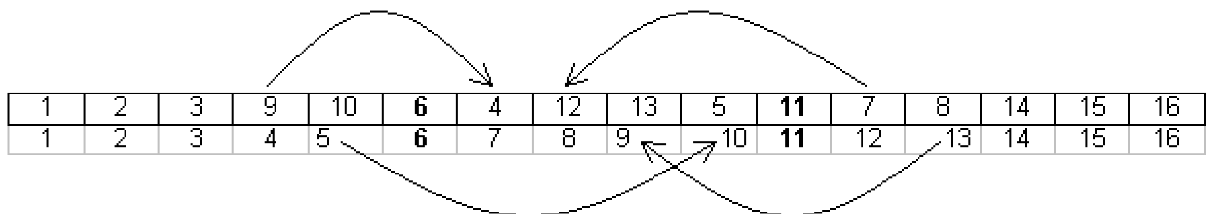
Es können auch mehrere Zyklen vorhanden sein. Dann wiederholt man die Abarbeitung solange bis alle Zyklen abgearbeitet wurden. Wie z. B. 5, 7, 5 dann 6, 10, 8, 6 dann 9, 11, 9.



Solche Zyklen lassen sich beim Mischen 2 sortierter Teilarrays immer dann finden, sofern man nicht einfach swappen (austauschen) muss oder alle Elemente des 2. Teilarrays bereits größer gleich jenen Elementen des 1. Teilarrays sind.

Kann man nur austauschen, tauscht man einfach Teil 2 mit Teil 3 aus. Andernfalls, falls man überhaupt noch etwas tun muss, verwendet man zyklisches Verschieben. Hat man die aussortierten Elemente bereits verschoben, kann in speziellen Fällen ein noch zu verschiebender Bereich übrigbleiben, falls dieser nicht schon in einem Zyklus hineinfällt oder man nicht noch zusätzlich einfache Zyklen hinzufügt, so dass diese abgearbeitet werden mussten. Wie z. B.:

<table>
<tr><td colspan="6">linke Folge</td><td colspan="6">rechte Folge</td></tr>
<tr><td colspan="2">1. Teil</td><td colspan="4">2. Teil</td><td colspan="4">3. Teil</td><td colspan="2">4. Teil</td></tr>
<tr><td></td><td>4</td><td>5</td><td colspan="3"></td><td colspan="3"></td><td>12</td><td>13</td><td></td></tr>
<tr><td>Rest von Teil 1</td><td colspan="3">Rest von Teil 3</td><td>4</td><td>12</td><td>13</td><td>5</td><td colspan="3">Rest von Teil 2</td><td>Rest von Teil 4</td></tr>
</table>



Man sieht, wenn man den Zyklus 7, 4, 9, 13, 8, 12, 7 verschiebt, dass die Elemente an Position 6 und 11 noch nicht an ihre richtigen Plätze verschoben wurden. Dies passiert dann, wenn die Anzahl der Elemente von Teil 2 weniger der Anzahl aussortierter Elemente von Teil 1, Teil 2 und Teil 3 echt größer 0 ist und die Anzahl der aussortierten Elemente von Elemente von Teil 1 gleich der Anzahl der aussortierten Elemente von Teil 4 ist. In unserem Beispiel: (4 - (1+1+1))>0 und 1=1. Wenn man aussortierte Folgen statt Elemente verwendet, dann muss man in diesem Fall die Elemente der aussortierten Folgen zählen.

In der obigen Skizze wurden die Pfeile nur für die aussortierten Elemente bzw. Folgen angegeben. Dies hat einen guten Grund, dann alle Pfeile bzw. Verschiebungen die zu einem Zyklus gehören und nicht in den Bereich der aussortierten Elemente weisen bzw. kopieren, kann man auf einfache Weise berechnen. Z. B.: Hat man eine Position die echt kleiner als die linke Position der aussortierten Elemente ist, dann braucht man zu dieser Position nur die Anzahl der Elemente von Teil 3 und die Anzahl der aussortierten Elemente von Teil 1 addieren. D.i. in diesem Fall für die Position 4: (4+4+1)=9, also kopiert man a[4]=a[9]. Hat man hingegen eine Position die echt größer als die rechte Position der aussortierten Elemente ist, dann braucht man bei dieser Position nur die Anzahl der Elemente von Teil 2 und die Anzahl der aussortierten Elemente von Teil 4 subtrahieren. D.i. in diesem Fall für die Position 12: (12-(4+1))=7, also kopiert man a[12]=a[7]. Wiederum gilt: Wenn man aussortierte Folgen statt Elemente verwendet, dann muss man in diesem Fall die Elemente der aussortierten Folgen zählen.

Oben wurde erwähnt, wenn mehrere Zyklen vorhanden sind, so bearbeitet man jeden Zyklus bis alle abgearbeitet wurden. Wenn diese obige Skizze betrachtet wird, zeigt sich, dass wenn wir als Startelement für einen Zyklus die aussortierten Elemente nehmen, dann haben wir jeden Zyklus zumindest einmal durchlaufen.

<table>
<tr><td colspan="3">linke Folge</td><td colspan="3">rechte Folge</td></tr>
<tr><td colspan="2">1. Teil</td><td>2. Teil</td><td>3. Teil</td><td colspan="2">4. Teil</td></tr>
<tr><td>Reste von Teil 1 und 3</td><td colspan="4">Aussortierte Elemente</td><td>Rest von Teil 4</td></tr>
</table>

Nun stellt sich die Frage: Ist es möglich mit weniger Startelementen für die Zyklen alle Zyklen zu durchlaufen, ohne jedes aussortierte Element auszuwählen? Dies ist in gewissen Fällen möglich. Wenn z. B. die aussortierten Elemente den Teil 2 oder den Teil 3 komplett ausfüllen. Dies ist deshalb, weil wenn z.B. der Teil 2 von aussortierten Elementen aufgefüllt wird, alle im Teil 2 vorhandenen Elemente nach Teil 3 und/oder Teil 4 verschoben werden. Da aber nun alle Elemente von Teil 2 bereits in der rechten Folge liegen und die rechte Folge nach der Verschiebung nur mehr Elemente von Teil 2 und Teil 4 enthalten darf, müssen auch alle Elemente, welche vorher in dem Teil 3 waren ebenfalls verschoben worden sein. Insbesondere müssen dann auch alle zu verschiebenden Elemente von Teil 1 betroffen sein.

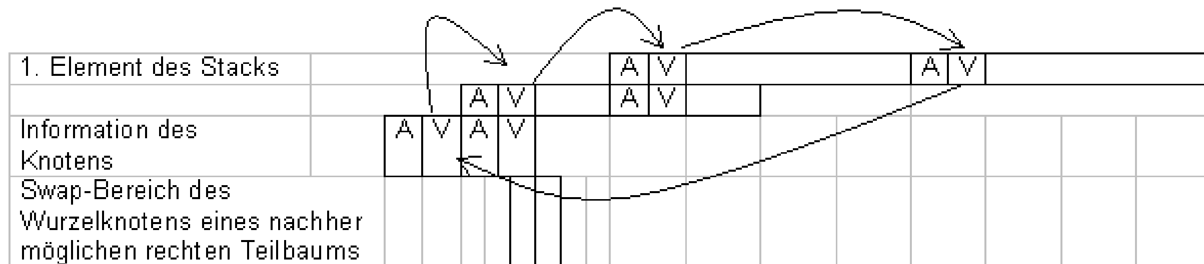
```
Swap(von[],                   // Indexliste der Zyklen
        a1,a2,a3,a4,          // Anzahl der aussortierten Elemente von Teil 1, Teil 2, Teil 3 und Teil 4
        i1,i2,j1,j2)          // linke und rechte Position von Teil 2, linke und rechte Position von Teil 3
{
        l=a1+a3;              // Anzahl aussortierter Elemente, welche in den Teil 2 verschoben werden
        r=a2+a4;              // Anzahl aussortierter Elemente, welche in den Teil 3 verschoben werden
        len=l+r;              // Anzahl aussortierter Elemente
        low=j1-l;             // linke Position der aussortierten Elemente
        up=i2+r;              // rechte Position der aussortierten Elemente
        nil=low;
                                                     // wähle die Startelemente der Zyklen:
        if (low<=i1) { len=l; k=i1-low; nil=i1; }    // Fall: alle aussortierten Elemente überdecken Teil 2
        else if (j2<=up) { k=l; len=j2-low+1; }      // Fall: alle aussortierten Elemente überdecken Teil 3
        else k=0;

        while (k<len)
        { i=index=low+k; v=a[i];              // rette Startelement des Zyklus
                for(;;)
                {
                        if(i<low)      j=i+(j2-i2)+a1; //addiere #  El. von Teil 3 + # aussortierter El. von Teil 1
                        else if(i>up)j=i-(j1-i1+a4);  //subtrahiere (#El von Teil 2 +#aussortierter El von Teil 4)
                        else
                        {
                                j=von[i-low];//hole Index j für ein nach Position i zu verschiebenden Element
                                von[i-low]=nil;     // markiere als besucht
                        }
                        if (j==index) break;        // wiederhole solange bis gerettetes Element
                                                    // zurückgeschrieben werden kann
                        a[i]=a[j]; i=j;             // verschiebe im Zyklus
                }
                a[i]=v;                             // schreibe gerettetes Element zurück
                while (++k<len && von[k]==nil); // überspringe schon bearbeitete Zyklen
        }
        // bearbeite alle noch offenen einfachen Zyklen
        if (i1+a2<=i2-l && a1==a4) Swap(i1+a2, i2-l, j1+r, j2-a3);
}
```

Im obigen Algorithmus kann man die Hauptschleife sogar noch früher verlassen, da man weiß, dass wenn alle aussortierten Elemente verschoben wurden, man die Schleife verlassen kann.

Beim Sortieren mit Folgen verwendet man 2 Felder von[] und nach[] gegenüber beim Sortieren von Elementen, welches nur ein zusätzliches Feld von[] benötigt. Weiters verwendet man beim zyklischen Verschieben, um den Index einer sortierten Folge zu berechnen, aus der man ein Element verschieben möchte, eine binäre Suchroutine. So betrachtet ist SwapSort mit sortierten Folgen gegenüber der Sortierung von Elementen i.a. erst dann effizienter, wenn das Array, dass zu sortieren vorliegt, nur leicht unsortiert ist.

3. SwapSort mit zyklischem Verschieben am Stack

Wenn man das Mischen beim einfachen SwapSort betrachtet, kann man feststellen, dass sich die rekursiven Aufrufe beim Mischen als binärer Lösungsbaum darstellen lassen, wobei zuerst der Knoten (hier swappt man), dann der linke Teilbaum und dann der rechte Teilbaum bearbeitet wird. Nun wollen wir nicht mehr sofort swappen, sondern erst dann, wenn wir mindestens 1 Element an seinen richtigen Platz platzieren können, von wo dieses Element nicht mehr verschoben wird. Hierzu macht man folgende Umstellung im Lösungsbaum: Behandelt man einen Knoten, so legt man zuerst die Information auf einen verketteten Stack, welcher beschreibt, was vertauscht werden soll, dann löst man den linken Teilbaum, nachher holt man die nötige Information wieder vom Stack und swappen, wenn noch etwas zu vertauschen ist, alle Elemente bis zum 1. Element des Stacks hinauf und dann löst man den rechten Teilbaum. Zur Veranschaulichung betrachten wir folgenden Stapel:



Das 1. Element ist jenes, welches als erstes auf den Stack gelegt wurde. Das letzte Element jenes, welches als letztes Element auf den Stack gelegt wurde.

Der Bereich A wurde bereits vom linken Teilbaum vertauscht, der Bereich V ist im Knoten zu vertauschen, d.h. dass der rechte V-Bereich vom 1. Element des Stacks zum linken V-Bereich des Knotens wandert und alle linken V-Bereiche nach rechts wandern. Der Rest wird von den anderen noch zu bearbeitenden Knoten vertauscht.

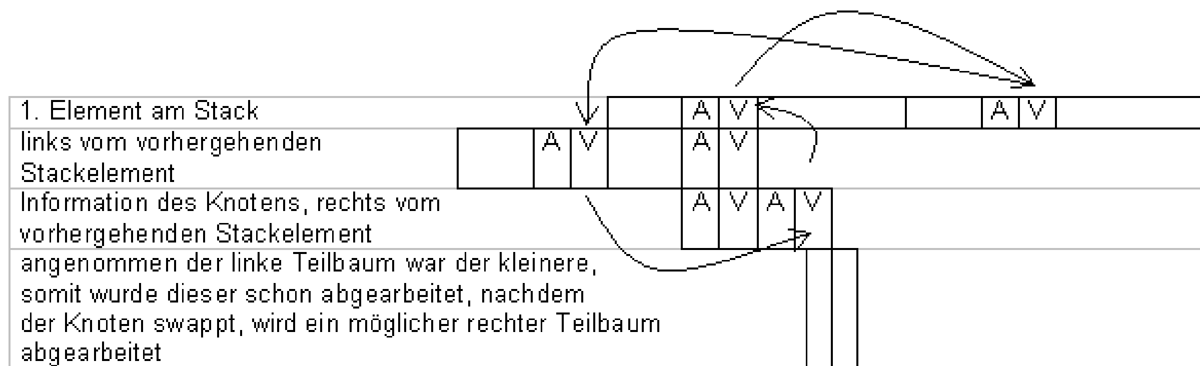
```
MergeOfSimpleStackSwapSort(a[],i1,i2,j1,j2,Stack) {
        d=Calculate_Distance(a[],i1,i2,j1,j2,Stack);
        if (d--) {
                push Swap-Bereich [i2-d,i2] und [j1,j1+d] auf den verketteten Stack
                if (i1<i2-d) MergeOfSimpleStackSwapSort(a[],i1,i2-d-1,i2-d,i2,Stack);

                vertausche noch zu vertauschende Elemente am Stack, so dass das letzte Element am Stack
                keine mehr zu vertauschenden Elemente enthält

                pop Swap-Bereich vom verketteten Stack     // [i2-d,i2] und [j1,j1+d] wurden vertauscht
                if (j1+d<j2) MergeOfSimpleStackSwapSort(a[],j1,j1+d,j1+d+1,j2,Strack);
     }}
```

Da man nun erst vertauscht, wenn man mindestens 1 Element an seine richtige Position setzen kann, muss man die Indizes bei der Berechnung der Größe d transformieren, sodass bei einem jeweiligen Vergleich, die richtigen Elemente verglichen werden. Dazu braucht man aber nur das letzte Element des Stacks, falls vorhanden, betrachten und transformieren den Index nur, wenn er in diesem Swap-Bereich enthalten ist.

Der Stapel ist in diesem Verfahren nicht durch O(log(Anzahl der Stapelelemente begrenzt)).

4. SwapSort mit zyklischem Verschieben am durch log(n) Stapelelemente begrenzten Stack

Im vorherigen Kapitel haben wir ein Verfahren beschrieben, wo mittels eines Lösungsbaumes das Mischen von SwapSort lösen. Dabei hat man zuerst immer den linken vor dem rechten Teilbaum bearbeitet. Spiegelbildlich, wenn man stets den rechten vor dem linken Teilbaum bearbeitet, funktioniert das oben beschriebene Verfahren ebenso. Da wir die Höhe des Lösungsbaumes beschränken wollen, werden wir jenen Teilbaum zuerst bearbeiten, der das kleinere Teilproblem löst, und dann erst denn anderen. Da sich die einzelnen Swap-Bereiche nicht mehr so einfach überlagern, dass der rechte Swap-Bereich eines Knotens immer unter dem linken Swap-Bereichs eines übergeordneten Knotens, falls vorhanden, liegt (oder spiegelbildlich), müssen wir in diesem Verfahren eine aufwendigere Indexberechnung durchführen. Zur Veranschaulichung betrachten wir dieses Stufenmodel:



Man sieht, dass man hier am Stack wie bereits vorher beschrieben zyklisches Verschieben anwenden kann.

```
MergeOfStackSwapSort(a[],i1,i2,j1,j2,Stack) { for (;;) {
(*)      d=Calculate_Distance(i1,i2,j1,j2,Stack);
         if (!d--) break;
         push Swap-Bereich [i2-d,i2] und [j1,j1+d] auf den verketteten Stack
         if (i2-i1<=j2-j1) // kleineres Teilproblem zuerst => Stack = O(log n)
         {
                 if (i1+d<i2) MergeOfStackSwapSort(a[],i1,i2-d-1,i2-d,i2,Stack);

(**)             vertausche noch zu vertauschende Elemente am Stack, so dass das letzte Element am Stack
                 keine mehr zu vertauschenden Elemente enthält

                 pop Swap-Bereich vom verketteten Stack     // [i2-d,i2] und [j1,j1+d] wurden vertauscht
                 if (j1+d==j2) break;
                 i1=j1, j1+=d, i2=j1, j1++;
         }
         else
         {
                 if (j1+d<j2) MergeOfStackSwapSort(a[],j1,j1+d,j1+d+1,j2,Stack);

(**)             vertausche noch zu vertauschende Elemente am Stack, so dass das letzte Element am Stack
                 keine mehr zu vertauschenden Elemente enthält

                 pop Swap-Bereich vom verketteten Stack      // [i2-d,i2] und [j1,j1+d] wurden vertauscht
                 if (i1+d==i2) break;
                 j2=i2, i2-=d, j1=i2, i2--;
         }}}
```

Die Indizes, die man in der obigen Funktion verwendet, zeigen i.a. nicht mehr auf die eigentlichen Daten, die verschoben werden sollen und müssen in den Programmteilen (*) und (**) übersetzt werden. Dabei sucht man den noch nicht übersetzten Index vom letzten Stackelement hinauf - sagen wir es so: Man steigt die Stiege die Stufen hinauf -, bis man diesen in einem Swap-Bereich findet. Nun testet man, ob er sich in einem bereits als vertauscht markierten Teil des Swap-Bereiches befindet, dann braucht man den Index nicht mehr übersetzen. Andernfalls gibt es zwei Fallunterscheidungen:

1. Der Index befindet sich im linken Teil eines Swap-Bereiches: Dann übersetzt man den Index auf dieser Stufe, so dass der Index nun das andere zu vertauschende Element im rechten Teil des Swap-Bereiches indiziert.
2. Der Index befindet sich im rechten Teil eines Swap-Bereiches: Dann übersetzt man den Index auf dieser Stufe, so dass der Index nun das andere zu vertauschende Element im linken Teil des Swap-Bereiches indiziert.

Haben wir den Index in dieser Stufe übersetzt, gehen wir die Stiege hinauf und prüfen, ob der Index weiter übersetzt werden muss, usf.

Im Teil (**) ist es nicht einmal notwendig alle Elemente, welche vertauscht wurden als vertauscht zu markieren, sondern nur jene, welche sich unmittelbar vor dem letzten Stackelement befinden. Das erspart einiges an Buchführung.

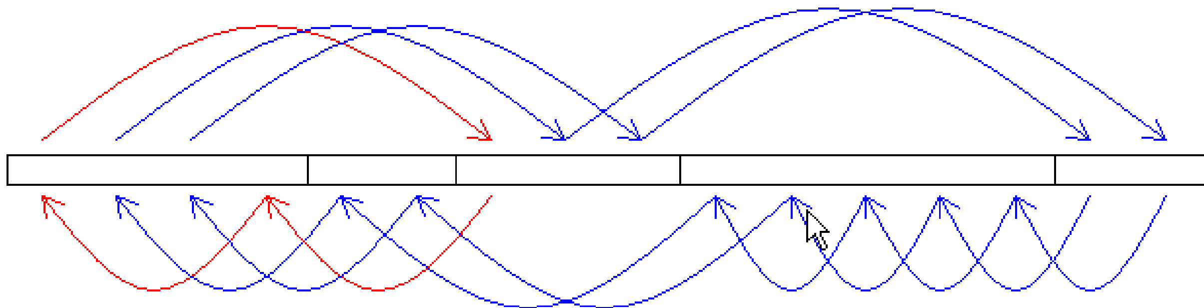
Kapitel II: Zyklisches Verschieben von sortierten Teilfolgen

Bei SwapSort mit zyklischen Verschieben wurde bei der Betrachtung von sortierten Teilfolgen erwähnt, dass man um sortierte Teilfolgen zu verschieben nur 2 lineare Felder benötigt. Dies und eine praktikable Verbesserung, um dem Aufwand beim binären Suchen zu reduzieren, werden wir hier betrachten.

Doch vorher betrachten wir noch, was passiert, wenn wir Blöcke in einem Speicherbereich verschieben.

Block 1	Block 2	Block 3	Block 4	Block 5
Block 2	Block 4	Block 1	Block 5	Block 3

Nun verschiebt man jedes einzelne Element der Blöcke der 1. Zeile mittels zyklischen Verschieben, so dass man sie in die 2. Zeile überführen kann.



Nun sieht man ganz klar, dass das Verschieben von Speicherbereichen einfacher durch zyklisches Verschieben gelöst werden kann, indem man alle Zyklen, die wir vorfinden 1 mal durchlaufen.

Man kann diese Zyklen abarbeiten ohne dass man ein zusätzliches Bitfeld benötigt, indem man die Zyklen immer in einer fest vorgegebenen Reihenfolge z.B. von links nach rechts abarbeitet und bevor man eine Verschiebung des Speicherbereiches durchführt - außer es ist der 1. Zyklus - einen Zyklustest durchführt. D.h. man macht einen Leerlauf ohne irgend etwas zu verschieben und kommt der Zyklus zur Position des Startelementes zurück ohne dass er auf eine Position vor dem Startelement stößt, so weiß man, dass dieser Zyklus noch nicht abgearbeitet wurde, andernfalls wurde er abgearbeitet. Somit benötigt man kein zusätzliches Bitfeld.

Wenn man aber zyklisches Verschieben bei sortierten Teilfolgen anwendet, stoßen wir auf das Problem, dass wenn wir denn Index der sortierten Teilfolge berechnen wollen, die ein Element enthält, das an eine bestimmte Position verschoben werden soll, diesen Index am besten durch ein Suchverfahren, welche eine Verschiebungsliste durchsucht, berechnen. Diese Berechnung kann zeitaufwendig sein, wenn man viele Teilfolgen zu verschieben hat. Daher ist es angebracht, falls es leicht durchführbar ist, schon bearbeitete Teilfolgen, sobald wie möglich aus der Verschiebungsliste zu entfernen. Arbeiten wir die Zyklen wie im oben angegeben von links nach rechts ab, so können wir Teilfolgen, die bereits abgearbeitet wurden und auf der linken Seite des Startelementes des gerade abzuarbeitenden Zyklus stehen, entfernen. Genauso gut kann man am anderen Ende der Verschiebungsliste ebenfalls Teilfolgen entfernen. Allerdings muss man dies speziell behandeln und man benötigt in diesem Fall ein Bitfeld, welches schon verschobene Elemente markiert. Durch die Markierungen erkennt man bereits verschobenen Elemente. Stehen diese Markierungen für das Ende der Verschiebungsliste, entfernt man die Teilfolgen. Man kann dieses Bitfeld aber durch eine spezielle Behandlung so ausrichten das eine gewisse Position stets mit einem noch nicht verschobenen Element übereinstimmt. Diese Bitfelder kann man wiederum selbst bei einem Zyklustest mitverwenden. Aus diesem Grund kann man auch am Anfang der Verschiebungsliste ebenfalls ein Bitfeld verwenden, um die Anzahl der Zyklustests zu verringern.

Beim SwapSort-Verfahren lassen sich sogar an 4 Stellen Teilfolgen, falls sie abgearbeitet wurden, aus der Verschiebungsliste auf einfache Weise entfernen.

Rest von Teil 1	Rest von Teil 3	Aussortierte Elemente von Teil 1 und Teil 3	Aussortierte Elemente von Teil 2 und Teil 4	Rest von Teil 2	Rest von Teil 4

D.i. beim Anfang und Ende aussortierten Elemente von Teil 1 und Teil 3 sowie am Anfang und am Ende der aussortierten Elemente von Teil 2 und Teil 4.

Kapitel II: SwapSort als Interface für andere effiziente Sortieralgorithmen und/oder Mischverfahren

Betrachtet man einfaches SwapSort:

```
SwapSort(a[],f,l) {
        if(f<l) {
                // Aufteilung - Phase 1
                k=f+(l-f)/2;                        // Aufteilung des Problems in kleinere Teile
                // wenn die 2 Teilarrays nicht stabil sortiert werden, ist i.a. die gesamte
                // Sortierung nicht mehr stabil
                sortiere a[] von Index f bis Index k      // sortiert 1. Teil des Problems
                sortiere a[] von Index k+1 bis Index l  // sortiert 2. Teil des Problems

                // MergeOfSwapSort() - swappt (Phase 2 ) und mischt (Phase 3)
                MergeOfSwapSort(a[],f,k,k+1,l); // mischen zweier sortierter Arrays
        }}
```

Man erkennt hier, dass es gar nicht vonnöten ist SwapSort() rekursiv aufzurufen, sondern man kann hier jeden beliebigen anderen Sortieralgorithmus verwenden. Ist der hierbei verwendete Sortieralgorithmus allerdings nicht stabil, dann ist i.a. eine stabile Sortierung nicht mehr gegeben. Dies könnte man z.B. einsetzen um den schlechtesten Fall von QuickSort zu verbessern. QuickSort benötigt im schlechtesten Fall ~ c*n*n Vergleiche, Phase 1 von SwapSort benötigt bei Anwendung von QuickSort im schlechtesten Fall ~ c*(n/2)*(n/2)*2 Vergleiche. Will man den schlechtesten Fall von QuickSort noch weiter verringern, teilt man durch rekursiven Aufruf von SwapSort das Problem solange, bis das Teilproblem klein genug ist.

Da QuickSort ein Sortierverfahren ist, welches selbst das Gesamtsortierproblem in weitere Unteraufgaben aufteilt, kann man QuickSort und SwapSort z.B. (mittels indirekter Rekursion) so kombinieren, dass QuickSort die Unteraufgaben SwapSort lösen lässt und SwapSort alle Sortierunteraufgaben QuickSort lösen lässt.

Auch beim Mischen von SwapSort kann man andere Mischalgorithmen verwenden. Insbesondere kann man durch Auswahl eines geeigneten Mischverfahrens weiter optimieren. Die einzelnen SwapSort Algorithmen unterscheiden sich eigentlich durch ihre Verschiedenartigkeit beim Mischen.

```
MergeOfSwapSort(a[],i,j,k,l){
n=Calculate_n(a[],i,j,k,l);
if (!n--){// Austauschen - Phase 2
        Swap(a[],j-n,j,k,k+n);                  // swap (tausche aus) Teil 2 mit Teil 3

        // Mischen - Phase 3
        if(i!=j-n) { // mischt Teil 1 mit Teil 3
                // verwende ein geeignetes Mischverfahren
                // mische stabil, falls die Sortierung stabil sein soll
                mische a[] von Index i bis Index j-n-1 mit a[] von Index j-n bis Index j
        }
        if(l!=k+n) { // mischt  Teil 2 mit Teil 4
                // verwende ein geeignetes Mischverfahren
                // mische stabil, falls die Sortierung stabil sein soll
                mische a[] von Index k bis k+n mit a[] von Index k+n+1 bis Index l
}}}
```

Kapitel III: SwapSort als Lastaufteilungsverfahren für Multiprozessorsysteme

1. Multitasking bzw. Multithreading mit SwapSort

Nun wollen wir untersuchen, welche Teile man parallel bzw. gleichzeitig ausführen kann. Dazu erweitert man SwapSort mittels eines Parameters p, welcher die Anzahl der Prozessoren angibt, welche in einem Multiprozessorsystem verfügbar sind.

```
SwapSort(a[],f,l,p) { // darf auf maximal p Prozessoren verteilt werden
        if(f<l) {
              // Aufteilung - Phase 1
              k=f+(l-f)/2;                    // Aufteilung des Problems in kleinere Teile
              if(p==1) {
                      sortiere a[] von Index f bis Index k           // sortiert 1. Teil des Problems
                      sortiere a[] von Index k+1 bis Index l         // sortiert 2. Teil des Problems
              } else parallel {
                      // erzeugt mindestens einen zusätzlichen Task, falls Ressource verfügbar
                      SwapSort(a[],f,l,p/2);       // darf p/2 Prozessoren verwenden
                      SwapSort(a[],f,l,p-p/2);    // darf p-p/2 Prozessoren verwenden
              }

              // MergeOfSwapSort() - swappt (Phase 2) und mischt (Phase 3)
              MergeOfSwapSort(a[],f,k,k+1,l,p); // mischen zweier sortierter Arrays
        }}
```

Die Befehlsabarbeitung im Block, der mit dem Schlüsselwort parallel eingeleitet wird, soll folgendes festlegen: Es sind in diesem Block 2 Funktionen bzw. Anweisungen enthalten, die gleichzeitig ausgeführt werden sollen. Dabei versucht man für die 1. Funktion einen neuen Task zu erzeugen. Wir nehmen an, dies sei immer möglich. (Andernfalls könnte man z.B., die in diesem Block enthaltenen Funktionen nacheinander abarbeiten.) Sobald der Task für die 1. Funktion erzeugt wurde, wird die Programmausführung mit der 2. Funktion, die im Block enthalten ist, fortgesetzt. Sobald man nach Ausführung der 2. Funktion am Ende des Blocks angelangt ist, muss man noch so lange warten, bis die 1. Funktion abgearbeitet bzw. der zusätzlich erzeugte Task beendet wurde.

Da die Aufteilungsphase von SwapSort, selbst die Sortieraufgabe in kleinere Sortieraufgaben teilt, die allerdings noch gemischt werden müssen, eignet sich die Aufteilungsphase zur Aufteilung der Arbeit auf mehrere Prozessoren.

Nun betrachten wir die Mischfunktion von SwapSort:

```
MergeOfSwapSort(a[],i,j,k,l,p){ // darf auf maximal p Prozessoren verteilt werden
n=Calculate_n(a[],i,j,k,l);
 if (!n--){
        // Austauschen - Phase 2
        Swap(a[],j-n,j,k,k+n,p);  // swap Teil 2 mit Teil 3 mit Hilfe von p Prozessoren

        // Mischen - Phase 3
        if(p==1) {
                if(i!=j-n) { // mischt Teil 1 mit Teil 3
                        mische a[] von Index i bis Index j-n-1 mit a[] von Index j-n bis Index j
                }
                if(l!=k+n) { // mischt  Teil 2 mit Teil 4
                         mische a[] von Index k bis k+n mit a[] von Index k+n+1 bis Index l
        }} else parallel {
                 // erzeugt mindestens einen zusätzlichen Task für die 1. Anweisung,
                 // falls Ressource verfügbar
                 if(i!=j-n) MergeOfSwapSort(a[],i,j-n-1,j-n,j,p/2);          // verwende p/2 Prozessoren
                 if(l!=k+n) MergeOfSwapSort(a[],k,k+n,k+n+1,l,p-p/2); // verwende p-p/2 Prozessoren
}}}
```

Hieraus ist ebenfalls ersichtlich, dass die Arbeitsaufteilung der Austauschphase und der Mischphase sich gut auf mehrere Prozessoren verteilen lassen.

Die Abarbeitung der Vorsortierphase und die Austauschphase reduziert sich ca. auf 1/p der Zeit. Die Abarbeitung der Mischphase reduziert sich auf beinahe 1/p der Zeit. Die Berechnung der Zahl n, welche festlegt, wie viele Elemente von der 1. mit der 2. Folge ausgetauscht werden sollen, fällt kaum ins Gewicht, da mittels Berechnung durch Intervallteilung, die Anzahl der Vergleiche auf log(Minimum(Anzahl der Elemente einer der beiden Teilfolgen)) reduziert wurde.

Allerdings gibt es bei der Abarbeitung in der Reihenfolge Sortierphase => Austauschphase => Mischphase 2 Engpässe zwischen den Phasen, da man von der einen auf die andere Phase erst übergehen kann, wenn die vorhergehende abgearbeitet wurde. Daher ist es wichtig, dass die Aufteilung der Arbeit in den einzelnen Phasen so geschieht, dass diese jeweils in annähernd gleich zeitaufwendige Unteraufgaben aufgeteilt werden.

2. Verteilung von Fibers an Tasks

Im vorhergehenden Abschnitt wurde in jeder einzelnen Phase von SwapSort, die Aufgaben so verteilt, dass hierfür immer wieder Tasks erzeugt und beendet wurden. Dieses kann zu einem merklichen Overhead führen, der sich insbesondere bei nicht umfangreichen und auf viele Prozessoren aufgeteilten, sowie bei zahlreich aufeinanderfolgenden Sortieraufgaben bemerkbar macht. Um diesen Overhead einzuschränken, kann man vor dem 1. Aufruf von SwapSort (p-1) zusätzliche Tasks erzeugen. In diesem einfachen Scheduler-Modell arbeitet ein Task gerade einen Fiber (Programmeinheit, welche in einem Task abläuft, aber vom Anwender verwaltet wird) ab, oder er befindet sich in einem Wartezustand. Ein Task bearbeitet eine Nachricht erst dann, wenn er sich wieder in dem Wartezustand befindet. Befindet ein Task sich im Wartezustand, wartet er immer solange, bis er eine Nachricht erhält. Erhält er eine Nachricht, dass er einen Fiber ausführen soll, dann führt er in diesem Fall eine in der Nachricht mitgeteilte Funktion aus. Nach der Abarbeitung des Fibers sendet dieser Task eine Nachricht jenem Task, der die Aufforderung einen Fiber abzuarbeiten sendete. Danach geht er wieder in den Wartezustand über. Ist das gesamte Sortierproblem bearbeitet, werden alle zusätzlich (p-1) erzeugten Tasks wieder beendet bzw. freigegeben.

```
ScheduleFibers() { // einer der zusätzlich erzeugten Tasks
        while(GET_MESSAGE(Message)) {                // wartet, solange bis er eine Nachricht erhält
                if (IS_SCHEDULE_MSG(Message))  {     // ist eine Nachricht in der Message-Queue,
                                                     // die einen Fiber zur Ausführung bringen soll,
                        STARTE_FIBER(Message);       // dann wird ein Fiber gestartet
                        SEND_FINISHED_MSG(Message); // nach Abarbeitung des Fibers, wird dem Task
                }}}                                  // der die Nachricht sendete, eine Nachricht
                                                     // zurückgesendet

SwapSort(a[],f,l,p[],p1,p2) {        // p[] Task-Liste
                                     // alle Tasks zwischen p1 und p2 dürfen von dieser Funktion
                                     // verwendet werden
                                     // Task p2, ist jener Task, der gerade ausgeführt wird
        if(f<l) {  // Aufteilung - Phase 1
                k=f+(l-f)/2;        // Aufteilung des Problems in kleinere Teile
                if(p1==p2) {      // wenn kein zusätzlicher Task verfügbar => tue
                        sortiere a[] von Index f bis Index k      // sortiert 1. Teil des Problems
                        sortiere a[] von Index k+1 bis Index l  // sortiert 2. Teil des Problems
                } else { // wenn ein zusätzlicher Task verfügbar => tue
                        // Task p2 sendet eine Nachricht an Task p1+(p2-p1)/2,
                        // dass die Funktion SwapSort() mit den mitgesendeten Parameterwerten
                        // ausgeführt werden soll
                        SEND_MESSAGE_START_SORT(SwapSort,a[],f,l,p[],p1,p1+(p2-p1)/2,p2);
                        SwapSort(a[],f,l,p[],p1+(p2-p1)/2+1,p2);
                        // Task p2 wartet auf eine Nachricht von Task p1+(p2-p1)/2
                        WAIT_MESSAGE(p1+(p2-p1)/2);
                }
                // Austauschen (Phase 2) und Mischen (Phase 3)  kann analog für die Ausführung von Fibers
                // programmiert werden
                mische a[] von Index f bis k mit a[] von Index k+1 bis Index l
        }}
```

Kapitel IV: Effizienzbetrachtungen

	Simple SwapSort	SwapSort	Simple StackSwapSort	StackSwapSort	MergeSort
Zeit in Sekunden mit Pentium 60 MHz	87	104	214	568	91
# Vergleiche	22846218	19291638	22846218	22846218	18675023
# Vertauschungen	48657621	456294			
# Verschiebungen		33802195	67217197	72224967	37348856
bei der Sortierung zusätzlich verwendeter Speicher, falls angegeben	log(n)	hier: log(n)+sqrt(n) sonst: log(n)+c	-	log(n)	-

In der oben angegebenen Tabelle wurden 1 Million zufällig generierte Datensätze - zu je 8 Byte, davon 4 Byte für den Schlüssel - sortiert. Die Angabe der Zeit gibt nur ungefähr einen Anhaltspunkt wie schnell ein stabiles Sortierverfahren ist, da je nach Belieben Optimierungen vorgenommen wurden. 1 Vertauschung von zwei Elementen entspricht eigentlich ungefähr dem Aufwand von 3 Verschiebungen. Hier wurde aber z.B. bereits das Vertauschen von Elementen in Maschinensprache programmiert, da es sich ohne großen Aufwand bewerkstelligen ließ.