A+D: Bitonisches Sortieren

H. P. Wolf

Version: 9.12.2005, file: shsort.rev

1 Schritte zum Verständnis des bitonischen Sortierens

Zur Klärung wollen wir zwei Definitionen voranstellen.

- Eine *bitonische Folge* ist eine Folge, die aus zwei zusammenhängenden monotonen Teilfolgen besteht.
- Eine *compare-exchange-Einheit* (kurz: COMP-EXCH) ist eine Einheit die zwei Inputwerte auf- oder absteigend sortiert wieder ausgibt.

Der hier diskutierte Algorithmus läßt sich wie der Algorithmus *Sortieren durch Mischen* als aus aufeinander folgenden Stufen zusammengesetzt ansehen. Für das *bitonische Sortieren* werden zunächst kleine bitonische Folgen gebildet und sortiert. Die Sortierergebnisse werden wieder zu bitonischen Folgen zusammengefügt. Dieses Spiel geht solange, bis die Input-Folge sortiert ist.
Bei dem Algorithmus *Sortieren durch Mischen* bestand ein wesentlicher Schritt darin, zwei sortierte Folgen zu Mischen. Die Betrachtung dieses Vorgangs führt uns auf die Spuren der hier vorgestellten Version des *bitonischen Sortierens*.

Schritt 1: Mischen zweier sortierter Folgen. Das Beispiel zur Abbildung 40.2 (Sedgewick, Kapitel: Parallele Algorithmen, Perfektes Mischen) macht deutlich, daß es recht einfach ist, zwei sortierte Folgen zu einer sortierten Folge zusammenzumischen.

Schritt 2: Mischen zweier sortierter Folgen durch wiederholte identische Elementverschiebungen und wechselnden *compare-exchange*-Einheiten. In der Abbildung 40.3 (wieder aus dem Sedgewick) werden die zu mischenden Folgen in einem Vektor hintereinander abgelegt, wiederholt perfekt geshuffelt und mit *compare-exchange*-Einheiten behandelt. Die spezielle Art des Element-Wechsels wird als *perfect shuffling* bezeicht. Es läßt sich leicht überprüfen, daß dieselben Operationen ausgeführt werden wie im Beispiel 40.2. Leider ist nicht ganz klar, warum das so ist. Außerdem ist für eine Umsetzung störend, daß die *compare-exchange*-Einheiten an unterschiedlichen Indexpositionen eingesetzt werden müssen. Beides wird durch den im folgenden dargestellten Ansatz hoffentlich behoben.

Schritt 3: Sortieren einer bitonischen Folge. Eine bitonische Folge ist eine Folge, deren Elemente erst ansteigen und dann abfallen oder umgekehrt. Monotone Folgen sind Spezialfälle von bitonischer Folgen. Das Sortieren bitonischer Folgen geschieht mit *teile und compare-exchange*-Operationen.

> *Teile und compare-exchange*-Operation: Teile die bitonische Folge mit den Elementen $(a_1, \ldots, a_n)$ in der Mitte, behandele jeweils die Elementpaare $(a_i, a_{n/2+i})$ mit einer *compare-exchange*-Einheit und lege die Minima nacheinander auf $(a_1, \ldots, a_{n/2})$ und die Maxima auf $(a_{n/2+1}, \ldots, a_n)$ ab.

Dann gilt:

> *Eigenschaft 1:* Die berechneten Minima sind alle kleiner als das kleinste der Maxima.

Die Elemente lassen sich also so in die kleinere und die größere Hälfte zerlegen. Die beiden Hälften besitzen noch eine weitere interessante Eigenschaft.

> *Eigenschaft 2:* Liegt bei einer zuerst ansteigenden und dann abfallenden bitonischen Input-Folge das Maximum in der Mitte, so sind die beiden Ergebnisfolgen wieder bitonische Folgen. Ist das Maximum nicht in der Mitte, so ergibt sich zumindest wieder eine bitonische Folge. Die zweite wird zu einer bitonischen Folge, wenn sie auf einem Kreis mit ihrem Endpunkt vor ihrem Anfangspunkt ablegt und dann der Kreis an geeigneter Stelle aufgetrennt wird.

Im ersten beschriebenen Fall ist klar, daß eine wiederholte Anwendung des geschilderten Prinzips: *teile und vergleiche* zum Schluß zu einer sortierten Folge führen muß. Aber auch in dem Fall, in dem das Extremum nicht in der Mitte liegt, führt das Prinzip zum Ziel. Dieses mache man sich am besten anhand einiger schöner Bleistift-Skizzen klar. Der folgende Schritt verdeutlicht, wie man das Sortieren einer bitonischen Folge mit perfect shuffling in Verbindung bringen kann.

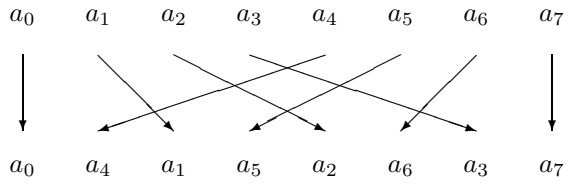
Schritt 4: Sortieren einer bitonischen Folge mit einem perfect-shuffle-Netzwerk. Betrachten wir das kleine Beispiel einer 8-elementigen bitonischen Folge. Dann werden im ersten Vergleichsschritt die Elemente mit folgenden Indizes verglichen. Als Laufbereich für die Indizes sei 0,. . .,7 angenommen.

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	4	000	–	100
1	–	5	001	–	101
2	–	6	010	–	110
3	–	7	011	–	111

Betrachtet man die binär notierten Indizes, so unterscheiden sich die der jeweils ersten Elemente, von denen der zweiten nur durch die erste Stelle, durch das erste Bit. Führt man auf den Indizes eine Bit-Shift-Operation aus, die alle Bits eine Stelle nach links schiebt und das erste wieder hinten anfügt, so werden die zu vergleichenden Elemente nebeneinander stehen.

Indexpositionen				
vorher		→	nachher	
arabisch	binär		binär	arabisch
0	000	→	000	0
1	001	→	010	2
2	010	→	100	4
3	011	→	110	6
4	100	→	001	1
5	101	→	011	3
6	110	→	101	5
7	111	→	111	7

Die Wanderung der Elemente läßt sich schnell auch graphisch darstellen:



Dieses ist das *perfect-shuffling*-Muster, das uns in der Abbildung 40.3 (Sedgewick) begegnet ist! Nach dieser Shuffle-Operation können benachbarte Elemente verglichen und ggf. vertauscht werden (*compare-exchange*). Würde man zwei weitere Shift-Operationen durchführen, ständen alle Elemente ohne die *compare-exchange*-Operation wieder genau an der ursprünglichen Stelle. Mit der *compare-exchange*-Operation sind jedoch nach dem zweimaligen Shuffeln in der ersten Hälfte die kleineren vier und in der zweiten die größeren vier Elemente zu finden, wobei beide Hälften wieder bitonische (oder fast bitonische) Folgen sind.
Die *teile und compare-exchange*-Operation läßt sich also erledigen durch folgende elementare Operationen:

SHUFFLE → COMP-EXCH → SHUFFLE → SHUFFLE

Der soeben durchgeführte Gedanke läßt sich wiederum auf die beiden bitonischen Hälften anwenden und damit beide Hälften in eine größere und eine kleinere zerlegen. Dieses wollen wir an dem Beispiel mit 8 Elementen verfolgen. Die folgende Tabelle zeigt, welche Elemente zu vergleichen sind.

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	2	000	–	010
1	–	3	001	–	011
4	–	6	100	–	110
5	–	7	101	–	111

Die zu vergleichenden Elemente unterscheiden sich nur in der zweiten Bitstelle. Wenn wir also zweimal nach links shiften, dann benachbarte mit *compare-exchange*-Einheiten vergleichen und ein weiteres Mal nach links shiften, würde für die beiden bitonischen Folgen der beiden Hälften gerade die *teile und compare-exchange*-Operation umgesetzt werden:

SHUFFLE → SHUFFLE → COMP-EXCH → SHUFFLE

Wir können natürlich die beiden *teile und compare-exchange*-Operationen Operationen verbinden und erhalten:

SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH → SHUFFLE

Als Zwischenergebnis haben wir nun vier bitonische Folgen (der Länge 2) ermittelt, wobei die Werte der ersten Folge kleiner als alle übrigen Werte, die Werte der zweiten kleiner als die Werte der dritten und vierten und die Werte der vierten größer als die ersten sechs Werte sind. Deshalb wird ein paralleler *compare-exchange*-Schritt zur sortierten Folge führen. Wieder folgt die Tabelle der zu vergleichenden Elemente:

arabisch			binär		
Element 1	–	Element 2	Element 1	–	Element 2
0	–	1	000	–	001
2	–	3	010	–	011
4	–	5	100	–	101
6	–	7	110	–	111

Jetzt unterscheiden sich die Index-Bits der zu vergleichenden Elemente nur noch an der letzten Stelle. Es ist, wie schon klar war, kein Links-Shiften erforderlich.
Zusammengefaßt erhält man aus der bitonischen Folge mit 8 Elemente durch die folgenden elementaren Operationen eine sortierte Folge:

SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH

Für bitonische Folgen der Länge 2^k benötigt man entsprechend $k = \log_2 2^k$ (SHUFFLE → COMP-EXCH) Schritte.
Damit können wir also problemlos bitonische Folgen sortieren. Nun ist noch zu klären, wie man dieses zur Sortierung nicht-bitonischer Folgen einsetzen kann.

Schritt 5: Der Gedanke zum Sortieralgorithmus Eine bitonische Folge läßt sich leicht aus einer aufsteigenden und einer absteigenden Folge zusammensetzen. Es ist also nur für eine bitonische Folge der Länge n das Problem zu lösen, zwei sortierte Folgen der Länge $n/2$ zu erstellen. Diese können wir natürlich mit dem gerade noch nicht ganz vorhandenen Sortieralgorithmus generieren.
Am Ende dieser Rekursion steht der Aufbau von bitonischen Folgen der Länge zwei, der offensichlich keinerlei Arbeit macht, da zwei Werte immer eine bitonische Folge bilden. Hieraus folgt umgekehrt das Vorgehen:

> bilde bitonische Folgen der Länge 2 → sortiere diese → bilde bitonische Folgen der Länge 4 → sortiere diese → bilde bitonische Folgen der Länge 8 → sortiere diese → ..., bis wir nur noch eine sortierte Folge vorliegen haben.

Der letzte Sortierschritt ist oben ausführlich beschrieben worden. Das Zusammenfügen von Folgen ist kein gedanklich schwieriger Akt. Was vielleicht etwas verwundert, ist, daß wieder perfect shuffling zum Einsatz kommt. Diesem Gedanken wollen wir uns im letzten Schritt zuwenden.

Schritt 6: Die Erstellung einer bitonischen Folge aus einer beliebigen. Nehmen wir wieder an, es liegen 8 unsortierte Werte vor. Dann ist das erste Ziel, 4 bitonische Folgen der Länge 2 zu erstellen und zu sortieren. Hierzu ist es nur erforderlich, die 8 Elemente als 4 Zweier-Folgen aufzufassen. Um 2 bitonische Folgen der Länge 4 zu erstellen, müssen wir jeweils zwei Werte abwechselnd auf- und absteigend sortieren. Dieses läßt sich leicht durch 4 *compare-exchange*-Einheiten bewerkstelligen, deren Sortier-Logik abwechselt. Ist die *compare-exchange*-Operation nur im Zusammenhang mit einer Shuffle-Operation erlaubt, sind zwei Shuffle-Operationen sowie eine *Shuffle-compare-exchange*-Operation nötig.
Zur aufsteigenden Sortierung der ersten bitonischen Folge der Länge 4 würden wir nach den bisherigen Überlegungen

SHUFFLE → COMP-EXCH → SHUFFLE → COMP-EXCH

einsetzen. Jedoch haben wir 8 Elemente in Bearbeitung, wobei die anderen vier absteigend sortiert werden müssen und nach *compare-exchange*-Einheiten mit absteigender Sortier-Logik verlangen. Deshalb müssen wir vor die Sortiererei eine Shuffle-Operation vorschalten. Hierdurch wird das zweite Index-Bit zum ersten und durch die nächste Shuffle-Operation, wie gewünscht, an die letzte Stelle bewegt. Leider stehen die Elemente der ersten bitonischen Folge nach der ersten Shuffle-Operation nicht mehr zusammen, sondern es wechseln sich immer zwei Elemente der ersten bitonischen Folge mit zweien der zweiten ab. Deshalb ist die Sortier-Logik der *compare-exchange*-Einheiten auch zu alternieren. Für das abschließende SHUFFLE-COMP-EXCH stehen die Elemente der bitonischen 4-er Folgen wieder beeinander, so daß zwei *compare-exchange*-Einheiten mit aufsteigender Sortier-Logik neben zwei *compare-exchange*-Einheiten mit absteigender Logik stehen. Nach deren Einsatz erhält man eine bitonische Folge mit 8 Elementen.
Hat man am Anfang mehr als 8 Werte (16, 32, ...), dann erhöht sich mit jeder 2-er Potenz die Anzahl der *Stufen* und die Anzahl der Shuffle-Operationen in jeder Stufe um 1.
Übrigens sei angemerkt, daß bei dieser Konstruktion die inneren Extremwerte der bitonischen Folgen immer in der Mitte liegen.

Literatur: Sedgewick, Cormen, Knuth III, Quinn, (Stone, Batcher).

Umsetzung: Der folgende Algorithmus setzt die obigen Gedanken um.

6 ⟨* 6⟩≡

```
shuffle<-function(x, op=rep(0,length(x)/2)){
           x <- matrix(x, length(x)/2, 2); incr<- x[,1]<x[,2]
           x[op== 1&!incr,]<-cbind(x[,2],x[,1])[op== 1&!incr,]
           x[op==-1& incr,]<-cbind(x[,2],x[,1])[op==-1& incr,]
           return(as.vector(t(x)))
}
shsort<-function(x,debug=F){
        n<-length(x); n.5<-n/2; lgn<-log(n,base=2); hh<-NULL
        for(i in 1:(lgn-1)){
           hh<-cbind(hh,matrix(0,n.5,lgn-i))
           for(j in 1:i)
             hh<-cbind(hh,as.vector(matrix(c(1,-1),2^(j-1),n.5/2^(j-1),T)))
        }
        hh<-cbind(hh,matrix(1,n.5,lgn))
        if(debug) print(hh)
        for(i in 1:dim(hh)[2])print(x<-shuffle(x,hh[,i]))
        return(x)
}
function(x=sample(1:16)){
# 1. Stufe
    for(i in 1:3){ x<-shuffle(x);print(x)}
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
# 2. Stufe
    for(i in 1:2){ x<-shuffle(x);print(x)}
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
        x<-shuffle(x,c(1,1,-1,-1,1,1,-1,-1));print(x)
# 3. Stufe
        x<-shuffle(x);print(x)
        x<-shuffle(x,c(1,-1,1,-1,1,-1,1,-1));print(x)
        x<-shuffle(x,c(1,1,-1,-1,1,1,-1,-1));print(x)
        x<-shuffle(x,c(1,1,1,1,-1,-1,-1,-1));print(x)
# 4. Stufe
    for(i in 1:4){ x<-shuffle(x,c(1,1,1,1,1,1,1,1));print(x)}
        invisible()
}
```