Maximale Teilsummen – Algorithmendesign

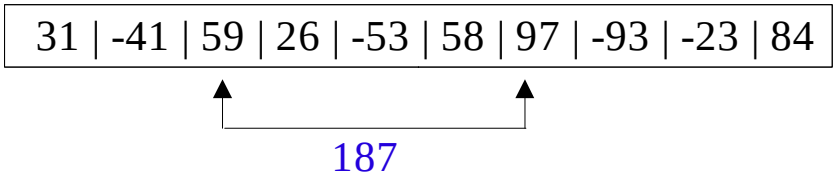
Die Geschichte beginnt, als Ulf Grenander 1977 an der Brown Universität Mustererkennungs-techniken an digitalisierten Bildern studiert. Er möchte, um Wahrscheinlichkeitsabschätzungen durchführen zu können, denjenigen rechteckigen Bereich eines Bildes herausfinden, in dem die Summe aller den Pixeln zugeordneten Zahlenwerte maximal ist. Da seine Versuche immer zu unbrauchbar hohen Laufzeiten führen (sein Programm arbeitet also einfach zu langsam), versucht er durch die Reduktion der Aufgabe auf ein eindimensionales Problem leichter Einsicht zu gewinnen. Dadurch entstand, was man heute das **Maximale-Teilsummen-Problem** nennt.

Die Aufgabe:

Gegeben ist eine Liste von Zahlen. Gesucht ist die maximale Summe, die aus benachbarten Elementen der Folge gebildet werden kann.

Beispiele:

[5,1,2,8,10]
[5,1,2,-10,8,-2,3]



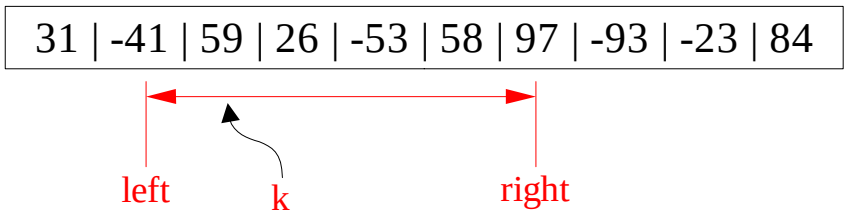
Die Aufgabe ist leicht, wenn es sich um lauter positive Zahlen handelt, denn dann ist es einfach die Summe der ganzen Liste. Schwierig wird es, wenn auch negative Zahlen (Abweichungen von einem Schwellenwert in der Bildverarbeitung) auftreten. Bereits die 7 Werte der zweiten Liste oben erfordern etwas Nachdenken. Und Grenander benötigte Listen mit einigen tausend Einträgen... Schnell entwarf er den offensichtlichen

Algorithmus 1:

Was genau ist zu tun: Wir müssen alle Teilfolgen bilden, die in einer Liste *data* enthalten sind, diese Werte jeweils aufaddieren und die maximale von all dieser Summen bestimmen. Diese Teilfolgen werden wir nicht in neue Listen stecken, da dies viel Speicherplatz kosten würde. Wir verwenden einfach je einen Zeiger auf das erste und das letzte Element der aktuell untersuchten Folge. Diese Zeiger nennen wir *left* und *right*. Lassen wir sie alle möglichen Konstellationen durchspielen und summieren die von ihnen begrenzten Teilfolgen auf.

```
maximum=0
für alle möglichen left:
    für alle möglichen right rechts davon:
        bilde die Summe data[left]+data[left+1]+...+data[right]
        wenn Summe>maximum: maximum anpassen
```

Die Summation der Werte erledigen wir mit einer eigenen kleinen Schleife.




```
def maxsum1(data):
    maximum = 0
    ## each left boundary
    for left in range(len(data)):
        ## each right boundary
        for right in range(left,len(data)):
            ## sum up all elements in between
            sum = 0
            for k in range(left,right+1):
                sum += data[k]
            ## new maximum reached?
            if sum>maximum: maximum=sum
    return maximum
```

Das war nicht schwierig. Doch das Zeitverhalten dieses Algorithmus ist erschreckend: 3 geschachtelte Schleifen, von denen jede eine Laufzeit proportional n hat, macht eine **Komplexität $O(n^3)$**. 10 mal so viele Elemente benötigen die 1000-fache Rechenzeit. Das sah natürlich auch Grenander, der sofort zwei verbesserte Versionen entwickelte:

Algorithmus 2a

Algorithmus 1 arbeitet äußerst ineffizient. Bei genauerer Betrachtung ist die innerste Schleife, die Summation der Folgenelemente, in dieser Weise unnötig.

Wurde etwa soeben mit *left*=2 und *right*=5 gearbeitet, so hat die innerste Schleife die Elemente mit Indizes 2,3,4 und 5 zusammengezählt. Nun folgt *left*=2 und *right*=6, woraufhin die innerste Schleife die Elemente 2,3,4,5 und 6 aufaddiert. Welche Verschwendung! Wir hätten doch bloß zur alten Summe das nächste Element, das mit Nummer 6, dazuzählen müssen. Und damit ist die innerste Schleife überflüssig geworden.

```
def maxsum2a(data):
    maximum = 0
    ## each possible left boundary
    for left in range(len(data)):
        ## build sums up to every right-side element
        sum = 0
        for right in range(left,len(data)):
            ## include one more right element
            sum += data[right]
            if sum>maximum: maximum=sum
    return maximum
```

Die Ordnung dieses Algorithmus? 2 geschachtelte Schleifen, also $O(n^2)$. Viel besser als Algorithmus 1! 10 mal so viele Elemente brauchen 'nur' noch 100 mal so lange.

Algorithmus 2b

Man könnte aber auch die Daten selbst umarbeiten. Statt ständig nach dem gleichen Muster Summen zu bestimmen, könnten wir diese doch gleich vorbereiten. Berechnen wir im Voraus alle Teilsummen (Partialsummen) vom untersten Element (Index 0) an.

Die ursprüngliche Liste *data*

31	-41	59	26	-53	58	97	-93	-23	84

wird damit zur Liste *partsum*

31	-10	49	75	22	80	177	84	61	145

Wollen wir nun etwa die Teilsumme von Index 3 bis Index 7 berechnen, so müssen wir *partsum[7]* nehmen, und alles was vor Index 3 kam, also *partsum[2]*, abziehen. Einen Sonderfall müssen wir beachten: wenn wir vom Index 0 an die Teilsumme wissen wollen, ist nichts zu subtrahieren.

```
def maxsum2b(data):

    ## calculate partial sums
    partsum = data[:]
    for pos in range(1,len(data)):
        partsum[pos]=partsum[pos-1]+partsum[pos]

    maximum = 0
    ## each left boundary
    for left in range(len(data)):
        ## each right boundary
        for right in range(left,len(data)):
            ## build difference of partial sums
            sum = partsum[right]
            if left>0: sum -= partsum[left-1]
            if sum>maximum: maximum=sum
    return maximum
```

Wie bei 2a ist die Ordnung $O(n^2)$.

Bemerkung für Python-Spezialisten:

- die if-Abfrage können wir mit einem Trick umgehen: Wir hängen als zusätzliches Element 0 an partsum an. *partsum.append(0)*. Nun kann Python auf *partsum[-1]* zugreifen und erhält den passenden Wert 0. (In anderen Sprachen könnte man alle Indizes um 1 verschieben und bei Index 0 eine 0 einsetzen. Ist aber nicht so glasklar...)
- Unschön ist die Tatsache, dass wir ein Hilfsfeld gleicher Größe zu *data* benötigen. Falls wir *data* verändern dürfen, können wir auf die Erzeugung von *partsum* verzichten und *data* am Platz umformen.

Diese Algorithmen schienen Grenander aber noch nicht der Weisheit letzter Schluss zu sein. Er schilderte das Problem einem Kollegen, Michael Shamos, der über Nacht folgenden Algorithmus entwickelte:

Algorithmus 3 (Divide and Conquer)

Die uns schon bekannte Technik des Divide and Conquer hat folgende Struktur:

1.) (divide) Teile das Problem in 2 ungefähr gleiche Teile

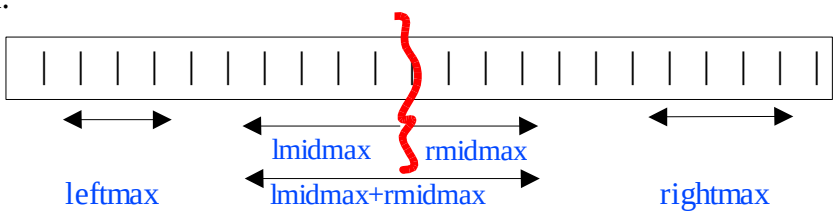
2.) (conquer) Löse die Teilprobleme (meist rekursiv)

3.) (merge) Ermittle aus den Teillösungen die Gesamtlösung

Weiters erinnern wir uns daran, dass durch die fortgesetzte 'Halbierung' der gesamte Arbeitsaufwand logarithmisch statt linear wird. Um 1024 Elemente zu berechnen, wird 10 mal geteilt und rekursiv gearbeitet. Dann ist man bei einelementigen Listen angelangt, die simpel handzuhaben sind. Wenn dann das Mergen gut klappt, kann man $O(n^2)$ auf $O(n\log n)$ drücken, was einen großen Gewinn darstellt (siehe Quicksort).

Nun ist es bei unserem Problem aber nicht so einfach – das Maximum kann zwar bei Halbierung ganz in der linken Hälfte, oder ganz in der rechten Hälfte liegen. Das gesamte Maximum ist dann das Maximum der beiden. Leider aber könnte es auch passieren, dass wir beim Zerschneiden der Liste genau die Teilfolge mit der maximalen Summe zerstückeln.

Um keine Information zu verlieren führen wir zwei weitere 'Zwischenmaxima' mit: Den Teil, der von der Trennstelle an nach links maximal ist, und der von der Trennstelle nach rechts. Beim 'Zusammenkleben' der Teile ergibt dann die Summe dieser beiden Teile den richtigen Wert für das Mittenmaximum.



Tatsächlich ist dieser Algorithmus von Komplexität O(nlogn).

Dieser Algorithmus schien auch der bestmögliche zu sein, wenn man etwa an ähnliche Aufgaben bei Sortieralgorithmen denkt.

```
def maxsum3(data):

    def maxsum3_(left,right):
        if left>right: return 0
        if left==right: return max(0,data[right])

        middle = (left+right)/2
        ## get max from middle to left side
        lmidmax = 0
        sum = 0
        for pos in range(middle,left-1,-1):
            sum += data[pos]
            lmidmax = max(lmidmax,sum)
        ## get max from middle to right side
        rmidmax = 0
        sum = 0
        for pos in range(middle+1,right+1):
            sum += data[pos]
            rmidmax = max(rmidmax,sum)
        ## get maximum in left partition
        leftmax = maxsum3_(left,middle)
        ## get maximum in right partition
        rightmax = maxsum3_(middle+1,right)
        ## maxsum is max of centerleft+centerright, insideleft, insideright
        return max(lmidmax+rmidmax,leftmax,rightmax)

    return maxsum3_(0,len(data)-1)
```

Einige Tage später schilderte Shamos auf einem Seminar das Problem und dessen Geschichte dem Statistiker Jay Kadane, der ohne langes Überlegen folgenden Algorithmus skizzierte:

Algorithmus 4 (Scanning line)

Seine Grundidee ist eine, die bei linearen Aufgaben oft weiterhilft:

Nimm an, Du hättest die Aufgabe für eine Liste mit n Elementen gelöst. Wie erhältst Du die neue Lösung, wenn ein einziges weiteres Element dazukommt?

Das endgültige Maximum kann im bereits behandelten Teil der Liste liegen. Dann ist alles leicht. Oder aber, es bildet sich erst, wenn wir nach rechts weitergehen und ein Element anfügen. (Wobei es sich auch noch später bilden kann...)

Machen wir es wie oben: Wir führen sowohl ein Maximum_Innerhalb, als auch ein Maximum_Rechts mit. Jedesmal merken wir uns zudem, was der bisher größte Wert war.

Doch auch das Maximum_Innerhalb ist einmal entstanden! Wir brauchen also keine eigene Behandlung, sondern nur die Beachtung der entstehenden Summen an der rechten Grenze.

```
def maxsum4(data):
    ## inner maximum
    maximum = 0
    ## maximum up to to right boundary
    maxright = 0
    ## repeat adding one element
    for pos in range(len(data)):
        ## update maximums
        maxright = max(maxright+data[pos],0)
        maximum = max(maximum,maxright)
    return maximum
```

Und hier staunt man: eine einzige for-Schleife über die Liste. Die Ordnung ist auf O(n) gedrückt worden!!!

Und damit ist bereits ein optimaler Algorithmus gefunden – schneller kanns nicht gehen, da jedes Element der Liste einmal berücksichtigt sein will.

Wer testen möchte: `>>> test(maxsum3)`

```
def test(proc):
    import random
    for i in range(10):
        l = random.randint(10,20)
        array = [0]*l
        for d in range(l):
            array[d]=random.randint(-50,50)
        print array
        print proc(array)
```

Zum Zeitverhalten der Algorithmen:

Folgende Werte ergaben sich für die Laufzeiten auf meinem Computer (P133 mit GnuC):
(für 2 Werte gemessen, Rest extrapoliert)

Anzahl der Werte	**Algor. 1**	**Algor. 2**	**Algor. 2a**	**Algor. 3**	**Algor. 4**
1 000	2.6 sec	20 msec	21 msec	0.8 msec	0.1 msec
100 000	1 Monat	3.4 min	3.5 min	155 msec	10 msec
1 000 000	80 Jahre	5.6 Stunden	5.6 Stunden	2 sec	0.1 sec

Nachsatz:

Das eindimensionale Problem ist damit vollständig gelöst. Leider wurde für das ursprüngliche zweidimensionale Problem Grenanders bis heute kein 'schneller' Algorithmus gefunden:

Design-Techniken

Was wir aus diesen Beispielen lernen können, lässt sich in Form einiger Regeln formulieren. Gute Programmierer haben diese im Hinterkopf, wenn sie ähnliche Algorithmen zu entwickeln versuchen:

- Speichere Zwischenergebnisse, um mehrfache Berechnungen zu vermeiden (Alg. 2a)
- Bereite die Informationen auf und finde geeignete Datenstrukturen (Alg. 2b)
- Divide and Conquer ist oft gut geeignet (Alg. 3)
- Scanning (Abtast-) Algorithmen sind oft gut geeignet (Alg. 4)