

Quick-Sort (C.A.R. Hoare, 1962)

Diese Methode ist wohl die beliebteste, da sie einfach zu programmieren ist und im Allgemeinen sehr rasch läuft. Über die (seltenen) Problemfälle werden wir noch sprechen.

Die Grundidee:

Wie bei Mergesort versucht man, statt einer einzigen großen Liste viele kleine zu sortieren und dann diese Teile wieder aneinanderzusetzen. Wenn Du etwa von der Aufgabe stehst, 100 Zettel mit Zahlen zwischen 1 und 20 zu sortieren, würdest Du vermutlich zuerst einmal zwei Stapel machen: einen mit den Zetteln von 1 bis 10 und einen zweiten für die Zettel von 11 bis 20. Nun ist das Problem einfacher geworden, da jeder Stapel weniger Zettel enthält. Den selben Trick könntest Du nocheinmal machen und wiederum jeden Stapel in Teilstapel zerlegen. Wenn Du die Stapel hübsch nebeneinander legst, gewinnst Du die sortierte Liste einfach dadurch, dass Du alle Stapel passend aufeinanderlegst. Du hast das Problem zuerst in einfachere Aufgaben **geteilt** und **beherrscht** dann die Gewinnung der Antwort. Derartige Strategien heißen in der Fachsprache auch '**divide and conquer**'.

Sortiere die Liste:

- 1.) sortiere sie grob in 2 Teile, indem Du 10 als Trennwert nimmst
- 2.) sortiere den einen Teil
- 3.) sortiere den zweiten Teil
- 4.) setze die Teile zusammen

In der Praxis ist es aber ein wenig komplizierter: Du hast gewusst, dass die Zahlenwerte von 1 bis 20 gehen und damit ist 10 ein guter Teilungspunkt (englisch: **pivot**, Drehpunkt). Diese Information liegt beim Sortieren aber nicht vor. Man muss sich willkürlich auf einen Wert festlegen. Tatsächlich ist es im Quicksort auch der entscheidende Punkt, einen guten Pivot zu finden. Die Teilung der Zahlen in zwei Bereiche wird als **partition** bezeichnet.

Genauer mit Indizes:

Sortiere(Liste, von bis):

wenn bis >von:

partitioniere die Liste um das Pivot-Element (bei Index p) herum

Sortiere(Liste,von,p-1)

Sortiere(Liste,p+1,bis)

Der wesentlich Punkt dabei ist die Forderung, dass nach dem Partitionieren alle Elemente links vom pivot KLEINER (oder gleich) sind als dieser, und alle Elemente rechts davon GRÖßER (oder gleich).

Beispiel:

```

3 6 2 5 4 1 9   Ursprüngliche Liste
3 6 2 5 4 1 9   Pivotelement
3 6 2 5 4 1 9   diese Elemente stehen falsch. also umtauschen.
3 2 4 1 5 6 9   geschafft. Der Pivot musste verschoben werden.
3 2 4 1 5 6 9   Jetzt sortiere den linken Teil und den rechten getrennt
1 2 4 3 5 6 9   nach der gleichen Methode z.B. mit Pivot 2 tausche 1,3
und diese Teillisten sind nun wirklich sehr einfach...
```

Beobachtungen:

- Das 'Zusammensetzen' aller Teillisten am Ende (das große Problem bei Mergesort!) erübrigt sich, da alle Teile bereits passend stehen
- Der Pivot muss gelegentlich verschoben werden
- hätten wir beim letzten Schritt oben 1 statt 2 als Pivot genommen, hätten wir uns nicht viel Arbeit gespart.

Wir versuchen vorerst einen 'einfachen' Algorithmus zu entwickeln und sprechen nachher über Verbesserungsmöglichkeiten.

Die Partitionierung:

Sie ist, wie erwähnt, der 'Knackpunkt' des Quicksort. Wir müssen einen Listenbereich so umstellen, dass die Forderung, im linken Teil seien lauter kleine und im rechten lauter große Werte im Vergleich zu einem Pivotelement, erfüllt ist.

Da wir keine Ahnung haben, in welchem Wertebereich sich die Elemente befinden, nehmen wir ein beliebiges Element der Liste heraus und erklären es zum Pivot. Bei zufälligen Listen wird der Wert wohl auch 'zufällig' sein. Wir wählen dazu einfach einmal das Element ganz rechts.

Weiters brauchen wir mindestens zwei Zeiger: einer muss zählen, ob wir schon alle Elemente in der Teilliste untersucht haben, der zweite muss die (siehe oben variable) Grenze zwischen den 'kleinen' und den 'großen' Elementen anzeigen.

i = oberstes Element der 'kleinen' Werte

j = zeigt auf nächstes zu untersuchendes Element der Liste L .

kleine Werte: von Listenstart bis i

große Elemente: von $i+1$ bis $j-1$

wir sind fertig: j erreicht das Listenende

erster Fall: $L[j] > y$. Das neue Element ist 'groß'. Wir brauchen nur j zu erhöhen.

```

klein  gross  unsortiert  Pivot
x x x x x x x x Z x x x x x x y
      i      j
x x x x x x x x Z x x x x x x y
      i      j

```

nun gehört das neue Element zu den 'großen'

zweiter Fall: $L[j] \leq y$. Das neue Element ist 'klein'.

i muss erhöht werden. Da steht aber das erste 'große' Element im Weg. Das tauschen wir einfach mit dem neuen aus. Es kommt dann eben ans Ende der 'großen' Liste.

```

klein  gross  unsortiert  Pivot
x x x a C x x x b x x x x x x y
      i      j
x x x a b x x x C x x x x x x y
      i      j

```

nun erhöhen wir i (b gehört dann zu den kleinen) und ebenfalls j , womit C wieder bei den großen ist.

```

x x x a b x x x C x x x x x x y
      i      j

```

technisch erhöhen wir i , tauschen $L[i]$ mit $L[j]$ und erhöhen j . Sicher wird j nach jeder Bearbeitung um 1 erhöht. Ein idealer Kandidat für eine for-Schleife, die bis unter den Pivot laufen muss.

Bemerkung: das Gleichheitszeichen könnte auch bei Fall 1 stehen. Obige Version ist aber statistisch besser, da sie ein wenig in der Liste 'umrührt'.

Wenn wir alle Elemente behandelt haben, steht der Pivot immer noch am rechten Rand statt zwischen dem kleinen und großen Teil. Wir tauschen ihn einfach mit dem ersten 'großen' Element aus. Dieses

bleibt damit klarerweise 'groß'.

```
x x x Z z z z z y
x x x y z z z z Z
```

Die nun festgelegte Position des Pivot soll unsere Partitionsfunktion zurückgeben.

Die Umsetzung:

Wir müssen immer die Indizes des zu bearbeitenden Bereichs mitübergeben!

<pre>def quicksort(L, anfang, ende): if anfang < ende: p = partition(L, anfang, ende) quicksort(L, anfang, p-1) quicksort(L, p+1, ende)</pre>	<p>Quicksort</p> <ul style="list-style-type: none"> - wenn nicht: fertig sortiert oder nur 1 Element - teilen - links sortieren - rechts sortieren
<pre>def partition(L, von, bis): pivot = L[bis] i = von-1 for j in fullrange(von, bis-1): if L[j] <= pivot: i = i+1 swap(L, i, j) swap(L, i+1, bis) return i+1</pre>	<p>Partitionieren</p> <ul style="list-style-type: none"> - oberstes Element - noch kein kleines Element - kleines Element (bei großen nur $j=j+1$ in for) - pivot in Mitte tauschen - Pivotposition zurückgeben
<pre>def quick(L): quicksort(L, 0, len(L)-1)</pre>	<p>mit dieser Funktion starten wir Quicksort, ohne Grenzen selbst übergeben zu müssen.</p>

Theorie:

Quicksort ist eine Methode, die typischerweise von Ordnung $n \log(n)$ ist. Im schlechtesten Fall (**worst case**) ist es aber eine 'langsame' Methode von Ordnung n^2 ! Dieser Fall tritt ein, wenn der Pivot immer so ungünstig gewählt wird, dass er zu keiner gleichmäßigen Aufteilung der Liste in zwei etwa gleich große Teile führt. Schlimmstenfalls ergibt sich immer die ursprüngliche Liste ohne den Pivot.

Hätten wir oben das Gleichheitszeichen zu den 'großen' Elementen gestellt, dann wäre offensichtlich eine bereits sortierte Liste der worst case!!! Der pivot wäre IMMER das größte Element (weil ganz rechts) und ALLE anderen Elemente wären 'klein'!

Praxis:

- Unsere Quicksort-Routine ist rekursiv. Mit einigem zusätzlichem Aufwand lässt sie sich aber iterativ formulieren. Dann ist sie für Alltagsaufgaben besonders gut geeignet.
- Allerdings (siehe oben) zum Einsortieren einiger weniger Werte in eine bereits sortierte Liste ist Quicksort NICHT die beste Lösung. (Wie sortiert man einen einzigen neuen Wert schnellstmöglich in eine bereits sortierte Liste? Denk an den Insertion-Sort).
- Man kann bei großen Listen Rekursionen sparen, wenn man für Teillisten von Länge 5 oder weniger einen Insertion-Sort einsetzt.
- alternative Partitionierungsmethoden: man wählt den Pivot anfangs in der Mitte. Dann sucht man links

aufwärts und rechts abwärts Elemente, die im 'falschen' Teil stehen und vertauscht diese (entspricht unserem Denkbeispiel ganz oben).

- Um Pivot-Probleme zu umgehen gibt es Möglichkeiten: man wählt als Pivot ein ZUFÄLLIG ausgewähltes Element der Liste. Dieses tauscht man am Anfang der Partitionierung an die oberste Stelle. (*swap(L,Zufallsposition,bis)* als erste Zeile vor *pivot=...*)
- noch besser: wähle drei zufällige Elemente. Das mittlere von ihnen (der Median) ist der Pivot.

Um ganz präzise zu sein: bei Quicksort wird eigentlich zuerst gearbeitet und danach geteilt. Also eher (wenn auch unüblich) als **'conquer and divide'** zu bezeichnen. (Bei Mergesort ist es dann genau umgekehrt.)

Ergänzung:

Wie wir wissen, unterstützt Python die Denkweisen der funktionalen Programmierung. Mit dieser Technik können wir den Quicksort **exakt so** programmieren, wie wir es zu Anfang des Kapitels überlegt haben:

Sortiere, indem Du bezüglich eines Vergleichswertes alle kleineren sortierst, den Vergleichswert anfügt (bzw. alle, falls er mehrmals vorkommt) und mit allen sortierten größeren abschließt.

```
def quicksort(L):
    if len(L)<=1: return L

    pivot = L[len(L)/2]

    return quicksort(filter(lambda i,p=pivot: i<p, L)) + \
           filter(lambda i,p=pivot: i==p, L) + \
           quicksort(filter(lambda i,p=pivot: i>p, L))
```

Dabei haben wir das Schlüsselwort `lambda` zur Erzeugung einer Vergleichsfunktion benützt, die auf ihr letztes Element wirkt, und `filter`, um alle Elemente aufzufangen, für die der Wert 'richtig' geliefert wird.

Der einzige Nachteil dieser Funktion `quicksort`: sie erzeugt ständig neue Teil-Listen, die verkettet werden, sie geht also nicht ökonomisch mit dem Speicherplatz um. Alle obigen trickreichen Überlegungen hatten ja nur den einen Zweck, keinen zusätzlichen Speicherplatz zu benötigen.

Tip: füge nach `'pivot = ...'` eine Zeile wie

```
print L, "    pivot:",pivot
```

ein. Dann kannst Du zusehen, welche Teillisten bearbeitet werden!