

# Benchmark Tests

Natürlich wollen wir von allen unseren Funktionen wissen, ob sie gut funktionieren, bei welche Listen sie besser oder schlechter arbeiten und wie lang sie jeweils dafür brauchen. Also einen Benchmarktest für unsere Sortieralgorithmen.

Um bequem testen zu können, wollen wir alle interessanten Listentypen vorbereiten. Dann können wir leicht sehen, ob es Laufzeitunterschiede zwischen bereits sortierten Listen, verkehrten, zufälligen etc. für einen Algorithmus gibt. Die Funktion *listen()* enthält alle Namen, damit wir sie später in professionellen Testserien verwenden zu können.

```
#####
## Listen mit jeweils n Elementen zum Testen zur Verfügung stellen
## korrekt : bereits sortiert
## verkehrt: gestürzt
## konstant: lauter gleiche Werte
## dazu    : richtig sortiert, aber letzte 'neu' Werte wiederum klein
#####
def listen(): return ['korrekt', 'verkehrt', 'konstant', 'zufall', 'dazu']

def korrekt(n=10):
    return fullrange(1,n)

def verkehrt(n=10):
    return fullrange(n,1,-1)

def konstant(n=10):
    return [n/2]*n

def zufall(n=10):
    liste = n*[0]
    for x in range(n):
        liste[x]=random.randrange(1,n+1)
    return liste

def dazu(n=10,neu=3):
    return fullrange(1,n-neu)+fullrange(1,neu)
```

Genauso machen wir uns eine Liste aller programmierter Sortier-Funktionen:

```
def methoden():
    return ['bubble', 'bubbleclever', 'shaker', 'insertion', 'selection', \
           'shell', 'quick', 'heap', 'merge']
```

`methode=methoden[2]` ist also der **string** 'shaker'. Wir können die Ausführung der Funktion veranlassen, wenn wir Python bitten, diesen Textstring auszuwerten. Das klappt mit der Funktion `eval()`. `eval(methode)` ergibt die Adresse der Funktion, `eval(methode)()` ist der tatsächliche Aufruf.

Nebenbemerkung: An der Existenz einer solchen Funktion erkennst Du Sprachen, die Ideen von **Lisp** verwirklichen. Beispiele: Logo, Forth, Scheme, Prolog etc, also die berühmten KI-Sprachen. Bei ihnen ist es etwa möglich, dass ein Programm einen String erzeugt und diesen dann als Programm abarbeitet, was wiederum die Regeln der String-Ezeugung verändern kann. Ein sich selbst programmierendes Programm! In C(++), Pascal, Java, Basic,... kann man davon nur träumen.

Wir könnten nun eine Funktion schreiben, die mithilfe der in Python eingebauten Methode `sort()` überprüft, ob unsere Programme richtig arbeiten.

Beispiel: `kontrolle(quick,100)`

```
def kontrolle(proc,n):
    L = zufall(n)                # n zufällige Zahlen
    Ergebnis = L[:]             # Liste übernehmen
    Ergebnis.sort()             # mit eingebauter Methode sortieren
    proc(L)                      # unsere Methode
    if L==Ergebnis: pass       # korrekt
    else: print "failed!", L    # Fehler ist passiert
```

Oder wir basteln eine Funktion, die die erzeugte zufällige Liste erst anzeigt, dann sortiert und das Ergebnis zur optischen Kontrolle wieder zeigt.

Beispiel: `test(quick,10)`

```
def test(proc,n):
    L = zufall(n)
    print L
    proc(L)
    print L
```

Jetzt wird es ernst: eine **Funktion für die Zeitmessung einer Methode mit einer Liste**. Wir erhalten die Laufzeit in Sekunden zurück. Bitte darauf achten, dass keine anderen Prozesse im Hintergrund die Werte verfälschen (Browser, MediaPlayer, WindowsExplorer-Aktualisierung,...)

Beispiel: `benchmark(quick,zufall(5000))`

```
def benchmark(proc,L):
    t1 = time.clock()
    proc(L)
    t2 = time.clock()
    return t2-t1
```

Wir können auch automatisch die Laufzeit **aller unsere Methoden mit der gleichen Liste** testen.

Beispiel: `benchMethoden(zufall(1000))`

```
def benchMethoden(L1):
    for methode in methoden():
        L = L1[:]
        print "%-12s: %f"%(methode,benchmark(eval(methode),L))
```

Oder wir können **einer Methode mit allen vorbereiteten Listen** auf den Zahn fühlen.

Beispiel: `benchListen(quick,3000)`

```
def benchListen(proc,n):
    for liste in listen():
        L = eval(liste)(n)
        print "%-12s: %f"%(liste,benchmark(proc,L))
```

**Erweiterung:**

schreib eine Funktion, die nach Übergabe der gewünschten Elementzahl **ALLE** Methoden mit **ALLEN** Listen durchprobiert. (Bei langen Listen lass aber die Bubblesort-Gruppe lieber weg...)

**Fortgeschrittene Erweiterung:**

teste alle Methoden mit allen Listen und gib die Ergebnisse nach Laufzeit **SORTIERT** aus! Hängt das Ergebnis stark von der Listengröße ab? Nimm 100, 200, 1000, 2000 Elemente!