Korrektheit

- Invariante:
  a[0…k-1] enthält nur Elemente aus a[0…k-1], aber in sortierter Reihenfolge
- Terminierung:
  - Die Schleife endet mit k=n
  - damit hätten wir nach Ende der Schleife: "a[0…n-1] enthält nur Elemente aus a[0…n-1], aber in sortierter Reihenfolge".

```
def insertionsort(a):
    for k in range( 1, len(a) ):
        x = a[k]
        i = k
        while i > 0 && a[i-1] > x:
            a[i] = a[i-1]
            i -= 1
        a[i] = x
```


- Invariante:
  a[0…k-1] enthält nur Elemente aus a[0…k-1], aber in sortierter Reihenfolge
- Initialisierung:
  - k=1
  - Invariante trivialerweise erfüllt, da a[0] sortiert ist

```
def insertionsort(a):
    for k in range( 1, len(a) ):
        x = a[k]
        i = k
        while i > 0 && a[i-1] > x:
            a[i] = a[i-1]
            i -= 1
        a[i] = x
```

- Invariante:
  a[0…k-1] enthält nur Elemente aus a[0…k-1], aber in sortierter Reihenfolge
- Schritt:
  - Nach dem Ende der inneren **`while`**-Schleife gilt
    - $a_{i-1} \leq x \ \wedge \ i \geq 0$ und
    - die Elemente, die vorher in a[i]…a[k-1] standen, stehen jetzt in a[i+1]…a[k], und
    - $x < a_{i+1}$.
  - Damit sind die Elemente a[0],…,a[i-1],x,a[i+1],…,a[k] wieder sortiert
  - Es ist kein Element aus a[0…k] verschwunden oder hinzugekommen

```
def insertionsort(a):
    for k in range( 1, len(a) ):
        x = a[k]
        i = k
        while i > 0 && a[i-1] > x:
            a[i] = a[i-1]
            i -= 1
        a[i] = x
```


Analyse

- Ist es in-place? Ist es stabil?
- Anzahl von Vergleichen: A ist sortiert → $\Theta(n)$ Vergleiche
- A ist unsortiert → $O(n^2)$ Vergleiche, denn:
  - Die maximale Anzahl von Vergleichen beim Einfügen von A_k ist (k-1)
  - Damit ist

$$C_{wc}(n) \leq \sum_{k=2}^{n} (k-1) = \sum_{j=1}^{n-1} j$$
$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

- A ist umgekehrt sortiert → $\Theta(n^2)$ Vergleiche

- im Average-Case spart man ein paar Vergleiche gegenüber Selectionsort
  - da beim Aufspüren des Zielortes im Mittel nur die Hälfte des sortierten Bereiches abgesucht werden muß
  - bei Selectionsort müssen zur Bestimmung des Minimums immer alle Elemente betrachtet werden
- Beim Verschieben müssen mehr Swaps als bei Selectionsort gemacht werden
- Insertionsort hat um so bessere Laufzeit, je besser das Array vorsortiert ist

Laufzeitvergleiche

- zufällige Daten (in Sekunden)

	N=10000	N=20000	N=30000	N=40000
Bubble	1,7	6,7	14,7	26,1
Selection	0,8	3,1	7,1	12,6
Insertion	0,4	1,25	2,9	6,7

- vorsortierte Daten (in Sekunden)

	N=10000	N=20000	N=30000	N=40000
Bubble	0	0	0	0
Selection	0,75	3,1	7,1	12,6
Insertion	0	0	0	0

- Häufig vorkommende Situation
  - unsortierter Datenbestand wird einmal aufwendig sortiert
  - danach ändert sich der Datenbestand immer nur um wenige Datensätze
    - einige Datensätze werden neu eingebracht
    - einige Datensätze werden geändert oder gelöscht
  - Neusortierung trifft daher häufig auf gut vorsortierte Datenbestände
    - Bubblesort und Insertionsort sind dann eine gute Wahl

Shellsort

- Von Donald Shell [1959]
- Idee:
  - versuche, die Datensätze durch größere Sprünge an die richtige Position zu bringen
  - parametrisiere Insertionsort um eine "Schrittweite" (*stride length*)
  - damit werden Elemente x mit größeren Schritten nach links bewegt
- Insertionsort im letzten Durchgang
  - fast nichts mehr zu tun
  - Garantie, daß die Daten anschließend sortiert sind

- Definition:
  Eine Folge $a_1, a_2, \ldots a_N$ heißt h-sortiert, wenn alle Teilfolgen

  $a_1, a_{1+h}, a_{1+2h}, \ldots$

  $a_2, a_{2+h}, a_{2+2h}, \ldots$

  $\ldots$

  $a_{h-1}, a_{2h-1}, a_{3h-1}, \ldots$

  sortiert sind.

- Methode:
  Stelle für eine abnehmende und mit 1 endende Folge von Inkrementen $h_t, h_{t-1}, \ldots h_1$ nacheinander eine h_i-sortierte Folge der Datensätze mittels parametrisiertem Insertionsort her.
- Algorithmus:
  - Bilde eine Folge von h-Werten, z.B.
    $1, 4, 13, \ldots, 3 \cdot h_{k-1}+1, \ldots$
  - starte bei dem größten h-Wert, der kleiner als N ist
  - benutze parametrisiertes Insertionsort
  - gehe zum nächst kleineren h-Wert und wiederhole
- Korrektheit: klar, da letzte Phase identisch zu Insertion-Sort ist

Beispiel

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	72	82	93	95	15 swaps

- Parametriesierung von Insertionsort, h = stride length:

```
def insertionsort( a, h ):
    for k in range( h, len(a), h ):
        x = a[k]
        i = k
        while i > h-1 && a[i-1] > x:
            a[i] = a[i-h]
            i -= h
        a[i] = x
```

```
insertionsort(a):
for k in range( 1, len(a), 1):
    x = a[k]
    i = k
    while i > 0 && a[i-1] > x:
        a[i] = a[i-1]
        i -= 1
    a[i] = x
```

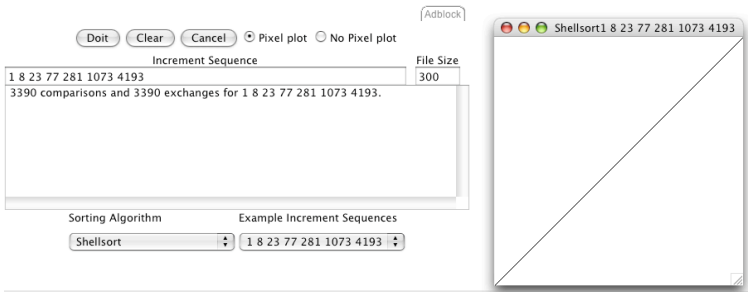
- Gesamtprogramm:

```
def shellsort(a):
    h = 1
    while h <= len(a):  h = 3*h + 1
    h /= 3
    while h > 0:
        insertionsort( a, h )
        h /= 3
```

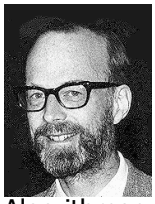
Eigenschaften

- Es gibt gute und schlechte Folgen für h
  - derzeit beste bekannte Folge
    - ... 16001, 3905, 2161, 929, 505, 209, 109, 41, 19, 5, 1
- garantiert weniger als $N^{1.5}$ Vergleiche
- Es gibt h-Folgen, für die Shellsort im worst-case $O\left(N^{1+\frac{1}{k}}\right)$ braucht
- Laufzeitverhalten wahrscheinlich $O(N \log^2 N)$ oder $O(N^{1.25})$
- Laufzeit größtenteils unabhängig von Daten

Demo



Quicksort



C. A. R. Hoare, 1960

- C.A.R. Hoare, britischer Informatiker, erfand 1960 Quicksort
- bis dahin dachte man, man müsse die einfachen Sortieralgorithmen durch raffinierte Assembler-Programmierung beschleunigen
- Quicksort zeigt, daß es sinnvoller ist, nach besseren Algorithmen zu suchen
- einer der schnellsten bekannten allgemeinen Sortierverfahren
- wird im Gegensatz zu Shellsort auch theoretisch gut verstanden
- Idee:
  - vorgegebenes Sortierproblem in kleinere Teilprobleme zerlegen
  - Teilprobleme rekursiv sortieren
  - allgemeines Algorithmen-Prinzip: *divide and conquer* (divide et impera)

Algorithmus

1. wähle irgend einen Wert W des Sortierfeldes
2. konstruiere Partitionierung des Arrays A mit folgenden Eigenschaften

- Alle Elemente von A_1 sind $\leq W$ (noch unsortiert).
- Alle Elemente von A_2 sind $\geq W$ (noch unsortiert).

3. wenn man jetzt A_1 und A_2 sortiert, ist das Problem gelöst
4. A_1 und A_2 sortiert man natürlich wieder mit Quicksort

Algo-Animation

Quicksort

P	S	E	U	D	O	M	Y	T	H	I	C	A	L
A	C	E	I	D	H	L	Y	T	O	U	S	P	M
A	C	E	I	D	H	L	Y	T	O	U	S	P	M
A	C	E	D	H	I	L	Y	T	O	U	S	P	M
A	C	E	D	H		L	Y	T	O	U	S	P	M
A	C	D	E	H		L	Y	T	O	U	S	P	M
A	C	D		H		L	Y	T					
A	C	D		H		L	Y	T