SelectionSort

Aufgabenstellung:

Gegeben ist ein Array **a** mit den **n** Komponenten **a[0], a[2], ... , a[n-1]** als Datenelemente, für die die Ordnungsrelationen **<** , **>** , **≤**, **≥** erklärt sind (also Komponenten z. B. vom Typ `integer`, `char` oder `string`). Die Inhalte dieser Datenelemente sind aufsteigend so anzuordnen, daß gilt:

a[0] ≤ a[2] ≤ . . . . . . ≤ a[n-1] .

In Python läßt sich ein Array **a** als Liste realisieren.

Sortieren durch direkte Auswahl („SelectionSort“)

Bei diesem Verfahren handelt es um einen typischen Vertreter eines imperativ formulierten Algorithmus’.

Der Algorithmus **SelectionSort** bestimmt

- das kleinste Element (Minimum) der Liste a[0], a[1], ... , a[n-1] und weist dieses der Komponente a[0] zu, dabei wird der Inhalt von a[0] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- das kleinste Element (Minimum) der Liste a[1], ... , a[n-1] und weist dieses der Komponente a[1] zu, dabei wird der Inhalt von a[1] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- das kleinste Element (Minimum) der Liste a[2], ... , a[n-1] und weist dieses der Komponente a[2] zu, dabei wird der Inhalt von a[2] derjenigen Komponente zugewiesen, der das Minimum entnommen wurde;
- . . . .
- . . . .
- das kleinste Element (Minimum) der Liste a[n-2], a[n-1] und weist dieses der Komponente a[n-2] zu, dabei wird der Inhalt von a[n-2] der Komponente a[n-1] zugewiesen.

Nach dem Abarbeiten der vorgenannten **n-1** Schritte ist das Array **a** aufsteigend sortiert.

In Python lassen sich die ersten vier Schritte wie folgt formulieren:

```
min = a[0]
i = 0 + 1
while i < n:
      if a[i] < min:
            min  = a[i]
            a[i] = a[0]
            a[0] = min
      i = i + 1

min = a[1]
i = 1 + 1
while i < n:
      if a[i] < min:
            min  = a[i]
            a[i] = a[1]
            a[1] = min
      i = i + 1
```

```
min = a[2]
i = 2 + 1
while i < n:
      if a[i] < min:
            min  = a[i]
            a[i] = a[2]
            a[2] = min
      i = i + 1

min = a[3]
i = 3 + 1
while i < n:
      if a[i] < min:
            min  = a[i]
            a[i] = a[3]
            a[3] = min
      i = i + 1
```

Letzter Schritt:

```
min = a[n-2]
i = n-2 + 1
while i < n:
      if a[i] < min:
            min    = a[i]
            a[i]   = a[n-2]
            a[n-2] = min
      i = i + 1
```

Zusammenfassend gilt: Der Anweisungsblock

```
min = a[j]
i = j + 1
while i < n:
      if a[i] < min:
            min  = a[i]
            a[i] = a[j]
            a[j] = min
      i = i + 1
```

ist nacheinander für j = 0, 1, 2, ... , n-2 abzuarbeiten; folglich fassen wir diesen Block als Schleifenrumpf einer weiteren Schleife (hier: for-Schleife) mit Schleifenindex j auf:

```
for j in range(0,n-1):
   min = a[j]
   i = j + 1
   while i < n:
         if a[i] < min:
               min  = a[i]
               a[i] = a[j]
               a[j] = min
         i = i + 1
```

Alternativ können wir die äußere Schleife als while-Schleife formulieren:

```
j = 0
while j <= n - 2:
   min = a[j]
   i = j + 1
   while i < n:
        if a[i] < min:
             min  = a[i]
             a[i] = a[j]
             a[j] = min
        i = i + 1
   j = j + 1
```

Das folgende Python-Programm

- weist nach Eingabe von **n** den Komponenten der Liste **a** Zufallszahlen aus dem Bereich 1, . . . , 1000000 zu,
- sortiert diese Liste **a** aufsteigend,
- ermittelt den Zeitbedarf für das Sortieren der **n** Datenelemente,
- gibt jeweils einen Teil der Quelliste und der sortierten Liste sowie den Zeitaufwand für den Sortiervorgang (in s) aus.

```
# SelectionSort

n = int(input('Anzahl der Datenelemente = '))
r = int(input('Wieviele Elemente sollen angezeigt werden? '))

a = list(range(1,n+1))

# Zuweisung von Zufallszahlen an die Komponenten der Liste a
for i in range(0,n):
    a[i]= randint(1,1000000)

# Ausgabe der Quelliste:
for i in range(0,r):
    print(a[i])

# Sortieren der Quelliste:

start = time.time()

j = 0
while j <= n-2:
   min = a[j]
   i = j + 1
   while i < n:
      if a[i] < min:
          min  = a[i]
          a[i] = a[j]
          a[j] = min
      i = i + 1
   j = j + 1

end = time.time()

# Ausgabe der sortierten Liste:
print()
print('sortierte Liste:')
for i in range(0,r):
    print(a[i])

print()
print('Zeitaufwand zum Sortieren von',n,'Elementen: {:7.3f} s'.format(end-start))
```

Aufwandsbetrachtung

Wir untersuchen den Algorithmus **SelectionSort** hinsichtlich seiner zeitlichen Komplexität, d. h. wir untersuchen, wie der Zeitbedarf zur Laufzeit sich in Abhängigkeit von der Anzahl n der zu sortierenden Datensätze verhält. Den Aufwand hinsichtlich des Speicherplatzbedarfs können wir hier vernachlässigen, da der Algorithmus SelectionSort auf dem Array **a** operiert und keinen weiteren Speicherplatz zur Laufzeit benötigt.

Hierzu betrachten wir denjenigen Programmteil, der das Sortieren ausführt:

```
j = 0
while j <= n-2:
    min = a[j]
    i = j + 1
    while i < n:
        if a[i] < min:
            min  = a[i]
            a[i] = a[j]
            a[j] = min
        i = i + 1
    j = j + 1
```

```
j = 0
while j <= n-2:
    min = a[j]
    i = j + 1
    while i < n:
        A
    j = j + 1
```

Wir fassen die Anweisungen aus dem Schleifenrumpf der inneren Schleife (hier: rot markiert) dieses Programmauszugs gedanklich zum Anweisungsblock **A** zusammen.
Um den Aufwand zu ermitteln, ein aus **n** Komponenten bestehendes array zu sortieren, fragen wir, wie oft Block **A** in Abhängigkeit von **n** abgearbeitet wird.

In folgender Tabelle gibt **z(j)** jeweils an, wie oft Block **A** in Abhängigkeit von **j** abgearbeitet wird.

Index j	**Index i**	**z(j)**
j = 0	1 ≤ i ≤ n-1	n - 1
j = 1	2 ≤ i ≤ n-1	n - 2
j = 2	3 ≤ i ≤ n-1	n - 3
j = 3	4 ≤ i ≤ n-1	n - 4
....	....	....
j = n-3	n-2 ≤ i ≤ n-1	2
j = n-2	n-1 ≤ i ≤ n-1	1

Für die Gesamtanzahl **z** der Abarbeitungen von Block **A** erhalten wir:

$$\begin{aligned} \mathbf{z} &= z(0) + z(1) + z(2) + z(3) + \ldots\ldots + z(n-3) + z(n-2) \\ &= (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots\ldots + 2 + 1 \\ &= 1 + 2 + \ldots\ldots\ldots + n-1 \\ &= \tfrac{1}{2} \cdot (n-1) \cdot n \quad \textit{(beachte untenstehenden Hinweis)} \\ &= \tfrac{1}{2} \cdot (n^2 - n) \\ &= \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n \end{aligned}$$

Für große Werte von n können wir den Summand $\frac{1}{2} \cdot n$ gegenüber dem Summand $\frac{1}{2} \cdot n^2$ vernachlässigen; somit folgt:

$\mathbf{z} \approx \frac{1}{2} \cdot n^2$

$\mathbf{z} \sim \mathbf{n^2}$

Bei SelectionSort wächst der Zeitbedarf proportional zum Quadrat der Anzahl n der zu sortierenden Datenelemente.

SelectionSort ist von polynomialer (hier: quadratischer) Komplexität.

Hinweis:
Für die Summe der ersten n natürlichen Zahlen gilt:

$$1 + 2 + \ldots\ldots + n = \tfrac{1}{2} \cdot n \cdot (n + 1)$$

Aufgaben:

1. Bestätige die quadratische Komplexität von SelectionSort experimentell anhand geeigneter Testläufe.
2. Modifiziere den Quelltext so, daß SelectionSort absteigend sortiert.
3. Sobald in der Teilliste `a[j], . . . , a[n-1]`, $0 \leq j \leq n-2$, ein Element gefunden wird, welches kleiner ist als das jeweils aktuelle Minimum, werden die Wertzuweisungen innerhalb des Blocks **`A`** ausgeführt, was für ein bestimmtes `j` ggf. auch mehrmals erfolgt. Optimiere den Algorithmus so, daß die Wertzuweisungen jeweils höchstens ein Mal für jeden Wert von `j` vorgenommen werden.
   Bestimme experimentell die Laufzeit und bestätige die (insgesamt bescheidene) Optimierung.
   *Hinweis: Ermittle zunächst den Index derjenigen Komponente, welche den kleinsten Inhalt innerhalb der Liste `a[j], . . . , a[n-1]` hat, und führe anschließend einmalig die Wertzuweisungen des Blocks **`A`** aus.*

Komplexität von Algorithmen
A(n) bezeichne den Aufwand und damit den Zeitbedarf zur Laufzeit in Abhängigkeit von **n** (z. B. n = Anzahl der zu verarbeitenden Datenelemente).

Algorithmus	Aufwand	Art der Komplexität
sequentielle oder lineare Suche	$\mathbf{A(n) \sim n}$	linear
binäre Suche	$\mathbf{A(n) \sim \log_2(n)}$	logarithmisch
SelectionSort	$\mathbf{A(n) \sim n^2}$	polynomial (hier: quadratisch)
MergeSort	$\mathbf{A(n) \sim n \cdot \log_2(n)}$	linear-logarithmisch
Fibonacci-Folge (rekursiv)	$\mathbf{A(n) \sim 2^n}$	exponentiell
Ackermann-Funktion	$\mathbf{A(3,n) \sim 2^{n+3} - 3}$ $\mathbf{A(3,n) \sim 2\uparrow(n+3) - 3}$ $\mathbf{A(4,n) \sim 2\uparrow\uparrow(n+3) - 3}$ $\mathbf{A(5,n) \sim 2\uparrow\uparrow\uparrow(n+3) - 3}$	exponentiell hyper-exponentiell

Algorithmen mit exponentieller Komplexität erweisen sich in der Praxis als unbrauchbar; selbst Algorithmen mit polynomialer Komplexität zeigen häufig ein ungünstiges Laufzeitverhalten.