```
(1)      for(int i=1; i< v.size(), i++) {
(2)          double current = v[k];
(3)          int j = k;
(4)          while (j>0) {
(5)              if (current < v[j-1]) {
(6)                  v[j] = v[j-1];
                 } else {
                     break;
                 }
(7)              j--;
             }
(8)          v[j] = current;
         }
     }
```

Annahme: Ausführung einer Programmzeile (1), ..., (8) benötigt jeweils konstante Zeit $c_1, \dots, c_8$.

Sei t_j die Anzahl der Ausführungen des Test in Zeile (4) für das aktuelle j:
Somit ergibt sich die Laufzeit als

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1)$$
$$+ c_4 \cdot \sum_{j=1}^{n-1} t_j + c_5 \cdot \sum_{j=1}^{n-1} (t_j)$$
$$+ c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1)$$
$$+ c_8 (n-1)$$

a) günstigster Fall:

Wann tritt der Fall ein? $\Rightarrow$ Die Daten sind bereits sortiert.

Dann ist $t_j = 1$ für alle j und wir erhalten:

$$T(n) = c_1 \cdot n + (c_2 + c_3 + c_8)(n-1) + (c_4 + c_5)(n-1)$$
$$= (c_1 + c_2 + c_3 + c_4 + c_5 + c_8) \cdot n - (c_2 + c_3 + c_4 + c_5 + c_8)$$
$$= \underline{a \cdot n + b} = f_1(n)$$

$$\Omega(g_1(n)) = \{ f(n) \mid \exists c > 0 \ \exists n_0 \ \forall n \geq n_0 : f(n) \geq c \cdot g(n) \}$$

Sei $g_1(n) = n$, so dass $\Omega(n)$: zz $f_1(n) \geq c \cdot n$

$$\Rightarrow a \cdot n + b \geq c \cdot n \qquad | -b$$
$$an \geq c \cdot n - b \qquad | : n$$
$$a \geq c - \underbrace{\frac{b}{n}}_{\to 0} \xrightarrow{n \to \infty} c \text{ , mit } c > 0 \text{ (nach Voraussetzung)}$$

$$\Rightarrow \underline{a \geq c} \quad \Rightarrow \underline{f_1(n) \in \Omega(n)} \quad \text{mit } g_1(n) = n$$

Warum wird Ω-Notation verwendet?

Da die Daten schon sortiert sind, ist unser anwachs linear mit n. Dies bildet eine untere Schranke, da wir mit diesem Algorithmus nie schneller werden können als eine lineare Komplexität. Aus diesem Grund wird die Ω-Notation (best case) verwendet.

b) schlechtester Fall:

Wann tritt der Fall ein? → Die Daten sind umgekehrt sortiert.
Der Test in (4) erfolgt dann für $j = 1, 2, 3, \dots, n-1$ Durchgänge mit Gaußschen Summenformel. $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$

$$T(n) = c_1 \cdot n + (c_2 + c_3)(n-1) + (c_4 + c_5 + c_6 + c_7) \cdot \frac{n(n-1)}{2} + c_8(n-1)$$

$$= (c_4 + c_5 + c_6 + c_7) \cdot \frac{n^2}{2} + (c_1 + c_2 + c_3 + c_8) \cdot n - (c_4 + c_5 + c_6 + c_7)\frac{n}{2} - (c_2 + c_3 + c_8)$$

$$= \underline{an^2 + bn + c} = f_2(n)$$

$\cancel{O}$ $O(g_2(n)) = \{ f(n) \mid \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : f(n) \leq c\, g(n) \}$

Sei $g_2(n) = n^2$, so dass $O(n^2)$: zz $f_2(n) \leq c\, g_2(n)$

$$an^2 + bn + c \leq dn^2 \qquad | -bn - c$$

$$an^2 \leq dn^2 - bn - c \qquad | : n^2$$

$$a \leq d - \underbrace{\frac{b}{n}}_{\to 0} - \underbrace{\frac{c}{n^2}}_{\to 0} \xrightarrow{n \to \infty} d > 0 \qquad \text{(nach Voraussetzung)}$$

$$\underline{a \leq d} \quad \Rightarrow \quad \underline{f_2(n) \in O(n^2)} \qquad \text{mit } g_2(n) = n^2$$

Warum muss die O-Notation verwendet werden?

Da die Daten umgekehrt sortiert sind, kann kein schlechtester Fall für dieses Problem eintreten. Dies bildet eine obere Schranke, da wir nie schlechter in der Laufzeit werden können für dieses Problem, d.h. nie schlechter als quadratische Komplexität.
Aus diesem Grund wird die O-Notation (worst case) verwendet.

c) typischer Fall:

Im Durchschnitt läuft die Schleife bis zur Hälfte durch, bevor die breake-Anweisung erreicht wird.

Somit ergeben sich nach der Gaußschen Summenformel

$$\sum_{j=1}^{n-1} \frac{j}{2} = \frac{1}{2}\sum_{j=1}^{n-1} j = \frac{1}{2}\,\frac{n(n-1)}{2} = \frac{n(n-1)}{4} \quad \text{Durchläufe}$$

$$T(n) = c_1 \cdot n + (c_2+c_3)(n-1) + (c_4+c_5+c_6+c_7)\cdot\frac{n(n-1)}{4} + c_8(n-1)$$

$$= (c_4+c_5+c_6+c_7)\frac{n^2}{4} + (c_1+c_2+c_3+c_8)n - (c_4+c_5+c_6+c_7)\frac{n}{2} - (c_2+c_3+c_8)$$

$$= an^2 + bn + c = f_3(n)$$

nach b) folgt analog mit $g_3(n) = n^2$, dass

$f_3(n) \in O(n^2)$

Somit haben wir für eine Durchschnittliche Laufzeit quadratische Komplexität.

---

d) Vektorgröße n, Sortieralgorithmus i:

(1) sort()
(2) insertion_sort_typical_time()
(3) insertion_sort_best_time()
(4) insertion_sort_worst_time()

ohne Codeoptimierung		mit Codeoptimierung	
time [s]	c_{in}	time [s]	c_{in}
$n=5000$			
(1) 0,00304	$7,1\cdot10^{-8}$	0,00040	$9,4\cdot10^{-9}$
(2) 0,16771	$6,7\cdot10^{-9}$	0,01211	$4,8\cdot10^{-10}$
(3) 0,00018	$3,6\cdot10^{-8}$	0,00002	$4\cdot10^{-9}$
(4) 0,33184	$1,3\cdot10^{-8}$	0,02478	$0,99\cdot10^{-9}$
$n=7000$			
(1) 0,00435	$7,02\cdot10^{-8}$	0,00058	$9,4\cdot10^{-9}$
(2) 0,32493	$6,6\cdot10^{-9}$	0,02403	$4,9\cdot10^{-10}$
(3) 0,00026	$3,7\cdot10^{-8}$	0,00003	$4\cdot10^{-9}$
(4) 0,64664	$1,3\cdot10^{-8}$	0,04943	$1,0\cdot10^{-9}$

~~mit~~ ohne Codeoptemierung		mit Codeoptemierung	
time [s]	c_{in}	time [s]	c_{in}
n = 9000			
(1) 0,00776	$7{,}03 \cdot 10^{-8}$	0,00077	$9{,}4 \cdot 10^{-9}$
(2) 0,53423	$6{,}6 \cdot 10^{-9}$	0,04015	$4{,}9 \cdot 10^{-10}$
(3) 0,00033	$3{,}7 \cdot 10^{-8}$	0,00003	$3{,}3 \cdot 10^{-9}$
(4) 1,06854	$1{,}3 \cdot 10^{-8}$	0,08234	$1{,}0 \cdot 10^{-9}$
n = 11000			
(1) 0,00710	$6{,}9 \cdot 10^{-8}$	0,00094	$9{,}2 \cdot 10^{-9}$
(2) 0,80311	$6{,}6 \cdot 10^{-9}$	0,06100	$5{,}0 \cdot 10^{-10}$
(3) 0,00040	$3{,}6 \cdot 10^{-8}$	0,00004	$3{,}6 \cdot 10^{-9}$
(4) 1,59602	$1{,}3 \cdot 10^{-8}$	0,12364	$1{,}02 \cdot 10^{-9}$
n = 13000			
(1) 0,00855	$6{,}9 \cdot 10^{-8}$	0,00114	$9{,}3 \cdot 10^{-9}$
(2) 1,11186	$6{,}6 \cdot 10^{-9}$	0,08500	$5{,}0 \cdot 10^{-10}$
(3) 0,00048	$3{,}7 \cdot 10^{-8}$	0,00005	$3{,}8 \cdot 10^{-9}$
(4) 2,22874	$1{,}3 \cdot 10^{-8}$	0,17332	$1{,}03 \cdot 10^{-9}$

Mittelwerte:

(1) $\bar{c}_{sort} = 7 \cdot 10^{-8}$ | $\bar{c}'_{sort} = 9{,}3 \cdot 10^{-9}$

(2) $\bar{c}_{typ} = 6{,}6 \cdot 10^{-9}$ | $\bar{c}'_{typ} = 4{,}9 \cdot 10^{-10}$

(3) $\bar{c}_{best} = 3{,}7 \cdot 10^{-8}$ | $\bar{c}'_{best} = 3{,}7 \cdot 10^{-9}$

(4) $\bar{c}_{worst} = 1{,}3 \cdot 10^{-8}$ | $\bar{c}'_{worst} = 1{,}0 \cdot 10^{-9}$

Funktionen:

(1) $t_{sort} = 7 \cdot 10^{-8} n \cdot \log(n)$ | $t'_{sort} = 9{,}3 \cdot 10^{-9} n \cdot \log(n)$

(2) $t_{typ} = 6{,}6 \cdot 10^{-9} \cdot n^2$ | $t'_{typ} = 6{,}6 \cdot 10^{-9} n^2$

(3) $t_{best} = 3{,}7 \cdot 10^{-8} \cdot n$ | $t'_{best} = 3{,}7 \cdot 10^{-9} \cdot n$

(4) $t_{worst} = 1{,}3 \cdot 10^{-8} \cdot n^2$ | $t'_{worst} = 1{,}0 \cdot 10^{-9} \cdot n^2$

Für sehr kleine n (n << 5000 siehe "Experiment") ist die Zeitdifferenz zwischen insertion_sort() und std::sort() unwesentlich.

Wie groß ist der Effekt der Codeoptimierung?

- Wie man bei den Mittelwerten erkennt ändert sich der Vorfaktor / Koeffizienten zwischen ohne und mit Codeoptemierung um eine zehner Potenz dies entspricht einer Verbesserung um 90%.