# Aufwand von InsertionSort

a)
Der günstigste Fall für InsertionSort tritt dann ein, wenn das Array schon sortiert ist. Dann liefert der Vergleich `current < v[j-1]` nämlich immer `false`. Die innere `while`-Schleife wird nie durchlaufen, und es werden insgesamt nur

$$\mathrm{f}_1(n) = (n-1)$$

Vergleiche benötigt.
(Da die Laufvariable `k` des Algorithmus bei 1 beginnt, ist der erste Vergleich der von Element 2 mit Element 1, der letzte der von Element $n$ mit Element $n-1$.)
Wir suchen $\mathrm{g}_1(n) = n$. Damit gilt:

$$\mathrm{f}_1(n) = (n-1) \in \Omega(\mathrm{g}_1(n)) = \Omega(n)$$

Hier muss die $\Omega$-Notation verwendet werden, da kein besserer Fall als der Beste eintreten kann. Jede Funktion $\mathrm{g}_1$ läuft also gleich lange oder langsamer, wächst also gleich schnell oder schneller. Wir brauchen eine Abschätzung von unten, und verwenden $\Omega$.

b)
Der ungünstigste Fall für InsertionSort tritt dann ein, wenn das Array falsch herum sortiert ist. Dann liefert der Vergleich `current < v[j-1]` nämlich immer `true`, bis das Element ganz vorne ist. Für das $k$-te Element werden also $k-1$ Vergleiche durchgeführt. Insgesamt werden also

$$\mathrm{f}_2(n) = \sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2} = \frac{n^2+n}{2}$$

Vergleiche benötigt. Wir suchen $\mathrm{g}_2(n) = n^2$. Damit gilt:

$$\mathrm{f}_2(n) = \frac{n^2+n}{2} \in \mathcal{O}(\mathrm{g}_2(n)) = \mathcal{O}(n^2)$$

Hier muss die $\mathcal{O}$-Notation verwendet werden, da kein schlechterer Fall als der Schlechteste eintreten kann. Jede Funktion $g_2$ läuft also gleich lange oder schneller, wächst also gleich schnell oder langsamer. Wir brauchen eine Abschätzung von oben, und verwenden $\mathcal{O}$.

c)
Der typische Fall für InsertionSort tritt dann ein, wenn das Array zufällig angeordnet ist. Dann liefert der Vergleich `current < v[j-1]` im Durchschnitt bis zur Hälfte der Elemente `false`, und es werden insgesamt

$$f_3(n) = \sum_{k=1}^{n} \frac{k}{2} = \frac{n \cdot (n+1)}{4} = \frac{n^2+n}{4}$$

Vergleiche benötigt.
Wir suchen $g_3(n) = n^2$. Damit gilt:

$$f_3(n) = \frac{n^2+n}{4} \in \mathcal{O}(g_3(n)) = \mathcal{O}(n^2)$$

d)

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <chrono>
#include <cmath>
```

```

void insertion_sort(std::vector<int> & v){
    for(int k=1; k<v.size(); ++k) {
        int current = v[k];
        int j = k;
        while(j > 0) {
            if(current < v[j-1]) {
                v[j] = v[j-1];
            } else {
                break;
            }
            --j;
        }
        v[j] = current;
    }
}

double sort_time_a(int n) {
    std::vector<int> array;
    // guenstigster Fall: Array bereits sortiert
    for (int i = 0; i < n; i++) {
        array.push_back(i);
    }
    // Fall a)
    auto start = std::chrono::high_resolution_clock::now();
    insertion_sort(array);
    auto stop = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = stop - start;
    return diff.count();
}

double sort_time_b(int n) {
    std::vector<int> array;
    // schlechtester Fall: Array falsch herum (absteigend) sortiert
    for (int i = n; i > 0; i--) {
        array.push_back(i);
    }

    auto start = std::chrono::high_resolution_clock::now();
    insertion_sort(array);
    auto stop = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = stop - start;
    return diff.count();
}

double sort_time_c(int n) {
    std::vector<int> array;
    for (int i = 0; i < n; i++) {
        array.push_back(i);
    }
    // typischer Fall: Arrayelemente zufaellig angeordnet
    std::random_shuffle(array.begin(), array.end());
```

```
    auto start = std::chrono::high_resolution_clock::now();
    insertion_sort(array);
    auto stop = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = stop - start;
    return diff.count();
}

double sort_time_std(int n) {
    std::vector<int> array;
    for (int i = 0; i < n; i++) {
        array.push_back(i);
    }
    std::random_shuffle(array.begin(), array.end());

    auto start = std::chrono::high_resolution_clock::now();
    std::sort(array.begin(), array.end());
    auto stop = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = stop - start;
    return diff.count();
}

int main() {
    std::vector<double>lengths {5e3, 1e4, 5e4};

    std::vector<double>times_a;
    std::vector<double>times_b;
    std::vector<double>times_c;
    std::vector<double>times_std;

    for (auto n: lengths) {
        double sum_a = 0;
        double sum_b = 0;
        double sum_c = 0;
        double sum_std = 0;
        for (int i = 0; i < 3; i++) {
            sum_a += sort_time_a(n)/n;
            sum_b += sort_time_b(n)/(n*n);
            sum_c += sort_time_c(n)/(n*n);
            sum_std += sort_time_std(n)/(n*std::log(n));
        }
        times_a.push_back(sum_a/3.0);
        times_b.push_back(sum_b/3.0);
        times_c.push_back(sum_c/3.0);
        times_std.push_back(sum_std/3.0);
    }


    for (int i = 0; i < 3; i++)
    {
        std::cout << "Size: " << lengths.at(i) << ":" << std::endl;
        std::cout << "Best Case: " << times_a.at(i) << std::endl;
```

```
        std::cout << "Worst Case: " << times_b.at(i) << std::endl;
        std::cout << "Average Case: " << times_c.at(i) << std::endl;
        std::cout << "std::sort: " << times_std.at(i) << std::endl;
    }
}
```

Ausgabe auf meinem Laptop:

```
Size: 5000:
Best Case: 1.19853e-09
Worst Case: 3.78429e-10
Average Case: 1.84715e-10
std::sort: 8.45508e-09

Size: 10000:
Best Case: 1.14717e-09
Worst Case: 3.72209e-10
Average Case: 1.90475e-10
std::sort: 7.22763e-09

Size: 50000:
Best Case: 1.16427e-09
Worst Case: 3.735e-10
Average Case: 1.86723e-10
std::sort: 7.38587e-09
```

Man erkennt, dass die mit $g_i()$ normierten Laufzeiten für jedes Verfahren näherungsweise unabhängig von der Arraygröße sind. Dies zeigt, dass die Lösungen für $g_i()$ korrekt sind.

Typische Arraygrößen, bis zu denen InsertionSort mit `std::sort()` mithalten kann, liegen Rechner- und Lastabhängig etwa in der Größenordnung von $n = 100$. (Diese Werte wurden als Mittelwert über alle Abgaben der Übungsgruppe des Tutors, der die Musterlösung geschrieben hat, ermittelt.) Der Effekt der Code-Optimierung ist eine Beschleunigung um etwa den Faktor 10.