Big O Notation - Zeitkomplexität in der Informatik

Allgemeines

Big O Notation

Die Big O Notation wird in der Informatik genutzt um Algorithmen eine Zeitkomplexität zuweisen zu können. Diese wird anhand der Datensätze und Art des Algorithmus berechnet. Heirbei muss aber bedacht werden, dass die Big O Notation nur dem Worst Case Szenario entspricht. Dies bedeutet, dass ein Algorithmus mit einer gewissen Notation in der Praxis deutlich besser abschneidet als zugeordnet.

Anwendung

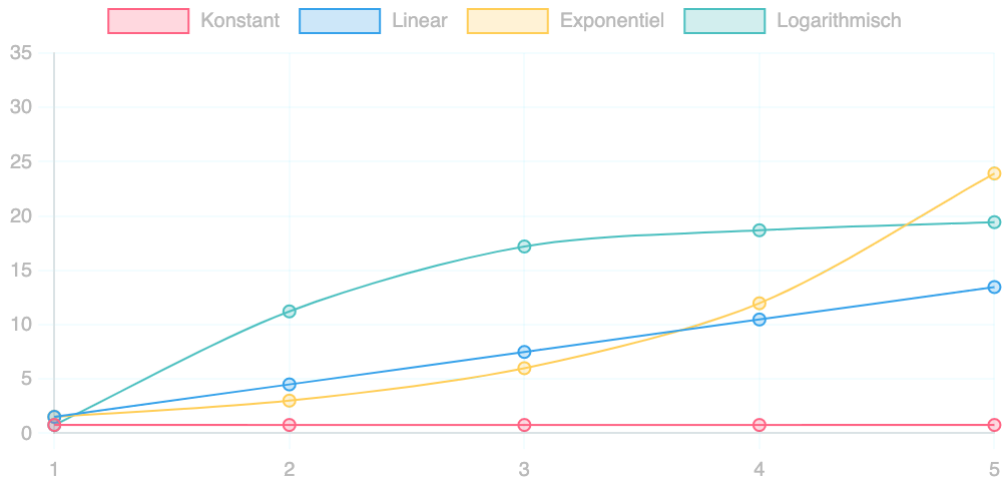
Die Anwendung der Notation beschränkt sich wie im oberen Abschnitt schon angegeben nur auf die Schreibweise des Algorithmus und die Mengen an Daten, die verarbeitet werden müssen. Hierfür werden die Variablen n und c genutzt. n steht für die Anzahl an Daten (Beispielsweise 15 Integer in einem Array) und c für die Anzahl an verschachtelten durchläufe des Array (eine for-Schleife würde also $c = 1$ entsprechen). Wenn man also beides kombiniert und ein Array mit 25 Integern in einem Algorithmus mit zwei verschachtelten for-Schleifen sortieren möchte käme die Notation $O(25^2)$ raus. Ruft man aber beispielsweise eine Zahl aus dem selben Array mit dem Index dieser ab, ist die Notation $O(1)$.

Häufig genutzte Notationen

Die Notationen, die in der Informatik am häufigsten angewand werden, sind die konstante, lineare, logaritmische und exponentiele Notation. Diese werden folgendermaßen aufgeschrieben:

- Konstant: $O(1)$
- Linear: $O(n)$
- Logarithmisch: $O(\log n)$
- Exponential: $O(n^c)$

Verbildlichung



Algorithmen und ihre Notation

Quicksort:

Best case

Der QuickSort erreicht seine optimale Performance, wenn die Arrays immer wieder in zwei gleich große Partitionen aufgeteilt werden. Dann wird bei Verdopplung der Anzahl der Elemente n nur eine einzige weitere Stufe von Partitionierungen benötigt.

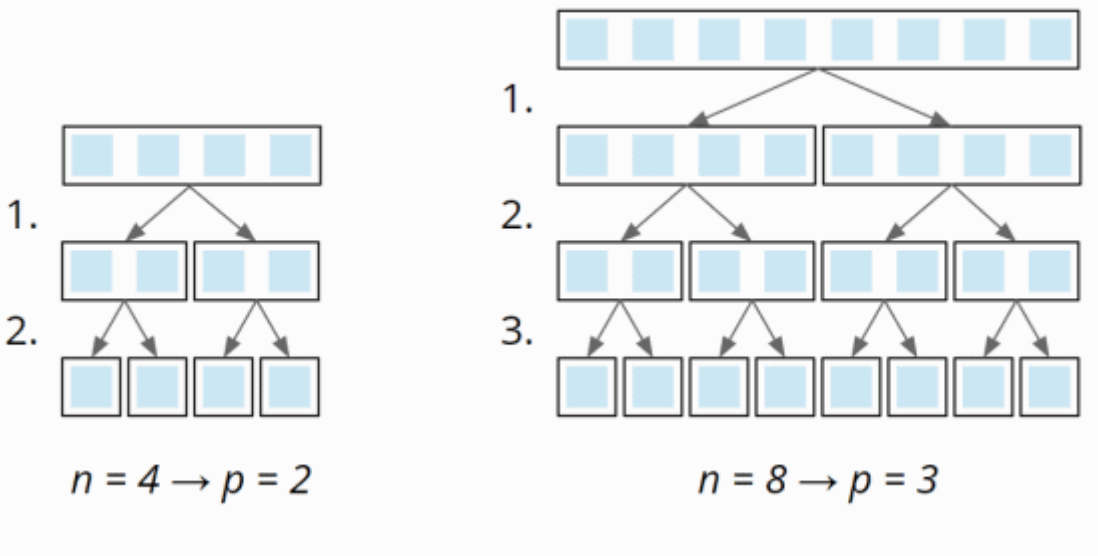


Abb 1: Schematische Darstellung des QuickSorts bei 4 und bei 8 Elementen

Die Anzahl der Partitionierungsstufen beträgt also $log_2(n)$.
Bei jeder Paritionierungsstufe werden also insgesamt n Elemnte auf die jeweils linke und rechte Parition aufgeteilt.
Erste Ebene: $1 * n$, Zweite Ebene: $2 * \frac{n}{2}$, Dritte Ebene: $4 * \frac{n}{4}$, ...

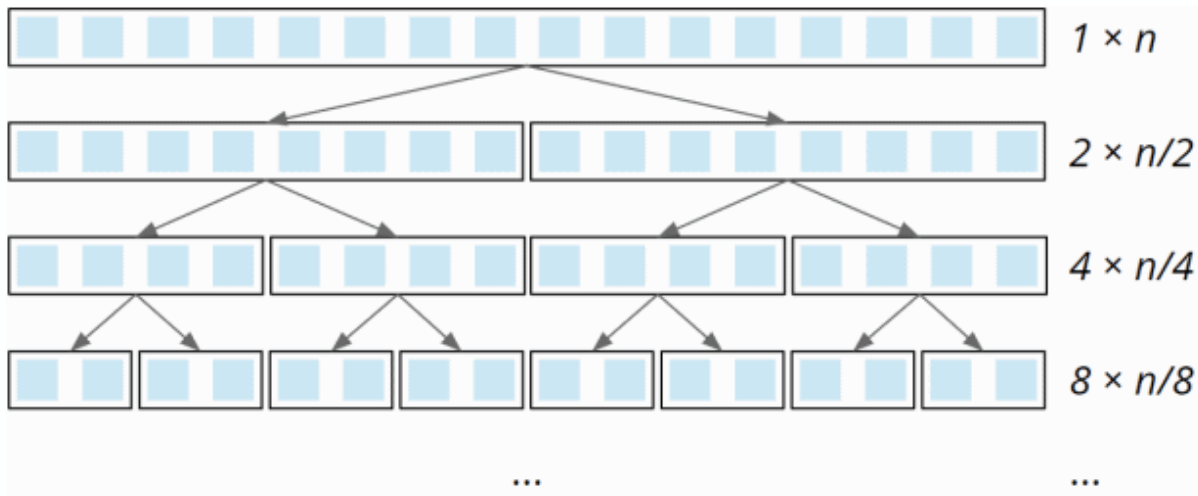


Abb 2: Darstellung der Aufteilung der Paritionierungsstufen

Da bei der Verdopplung der Array-Größe sich auf der Paritionierungsaufwand verdoppelt, ist der Gesamtaufwand bei allen Paritionierungsstufen gleich.
Wir haben also n Elemente mal $log_2(n)$ Paritionierungsstufen.
Daher beträgt die Zeitkomplexität vom Quicksort im *best case*:

$$O(n * log(n))$$

Average case

Die durchschnittliche Zeitkomplexität ist gleich wie die beste Zeitkomplexität.

$$O(n * log(n))$$

Quelle: https://en.wikipedia.org/wiki/Quicksort#Average-case_analysis

Worst case

Falls das Pivot-Element immer das kleinste oder größte Element des Arrays ist, beispielsweise bei einem vorsortierten Array, würde das Array nicht in zwei gleich große Paritionen aufgeteilt werden, sondern in eine der Länge 0 und eine der Länge $n - 1$.

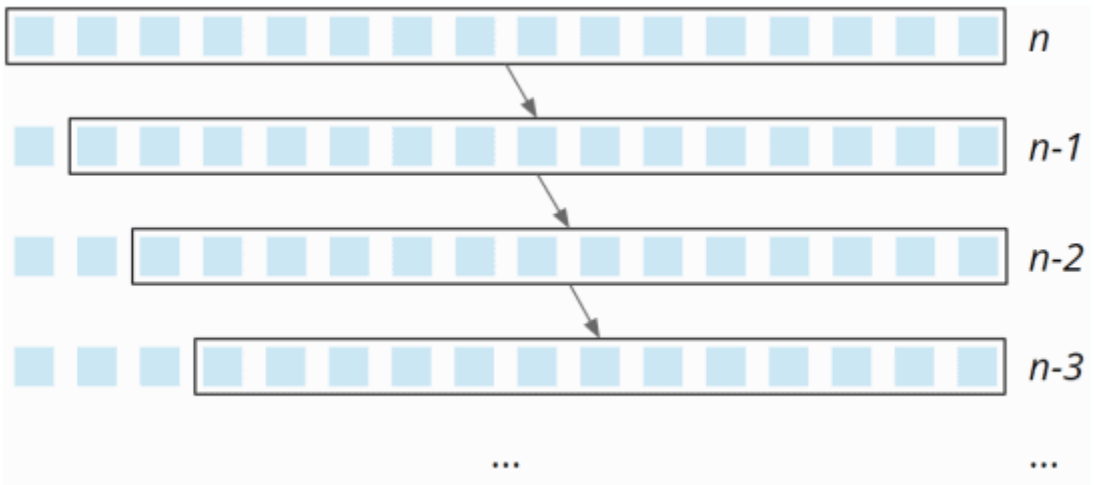


Abb 3: Paritionierungsstufen beim *worst case*

Der Paritionierungsaufwand sinkt dabei linear von n bis 0 - in der Mitte beträgt dieser daher $\frac{1}{2}n$.
Bei n Paritionierungsstufen ist der Gesamtaufwand daher $n * \frac{1}{2}n$.
Daher beträgt die Zeitkomplexit#t vom Quicksort im *worst case*:

$$O(n^2)$$

Zusammenfassung

Quicksort ist ein effizienter, instabiler Sortieralgorithmus mit einer Zeitkomplexität von $O(n * log(n))$ im *best und average case* und $O(n^2)$ im *worst case*.
Für sehr kleine n ist Quicksort langsamer als Insertion Sort und wird daher in der Praxis in der Regel mit Insertion Sort kombiniert.

Quelle: https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/

Selectionsort:

Best case

Der best case ist, wenn das gegebene Array bereits sortiert ist. Doch selbst in diesem Fall muss der Algorithmus das ganze Array durchgehen und eine Operation durchführen, bevor es weiß, dass das Array bereits sortiert ist. Dementsprechend resultiert also die Notation $O(n^2)$.

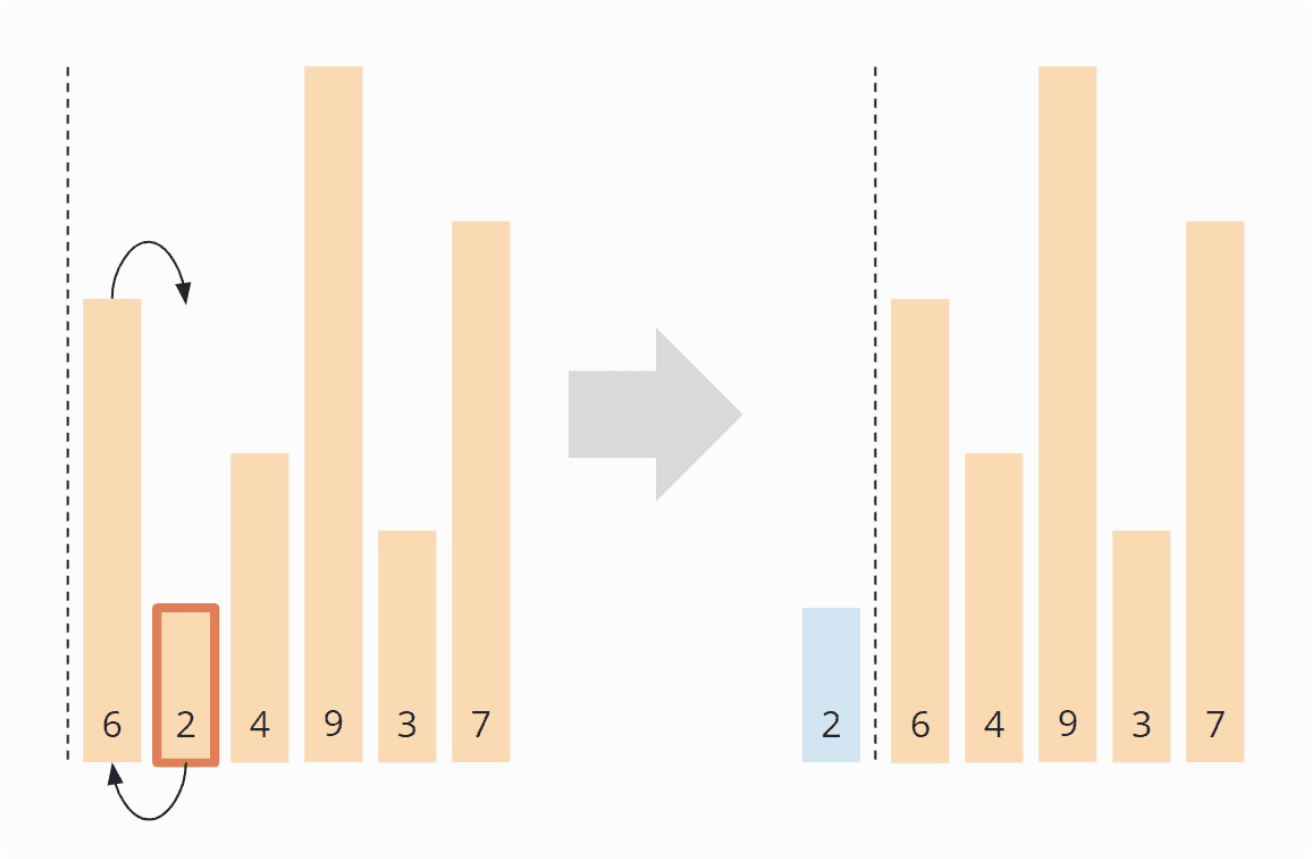


Abb. Vorgang des Selectionsorsts

Average case

Der avarage case ist, wenn ein Teil des gegebenen Arrays sortiert bereits sortiert ist. Hier auch muss der Algorithmus also wie im Best case das komplete Array durchgehen und Operationen durchführen bevor dieses vollständig sortiert ist. Die daraus resultierende Notation ist hier ebenfalls $O(n^2)$.

Worst case

Der worst case ist wenn das gegebene Array vollständig unsortiert ist. Hier muss der Algorithmus erneut das komplete Array durchlaufen und Swap Operationen durchführen, bis dieses vollständig sortiert ist. Wie auch im Best case und um Average case ist die Notation $O(n^2)$.

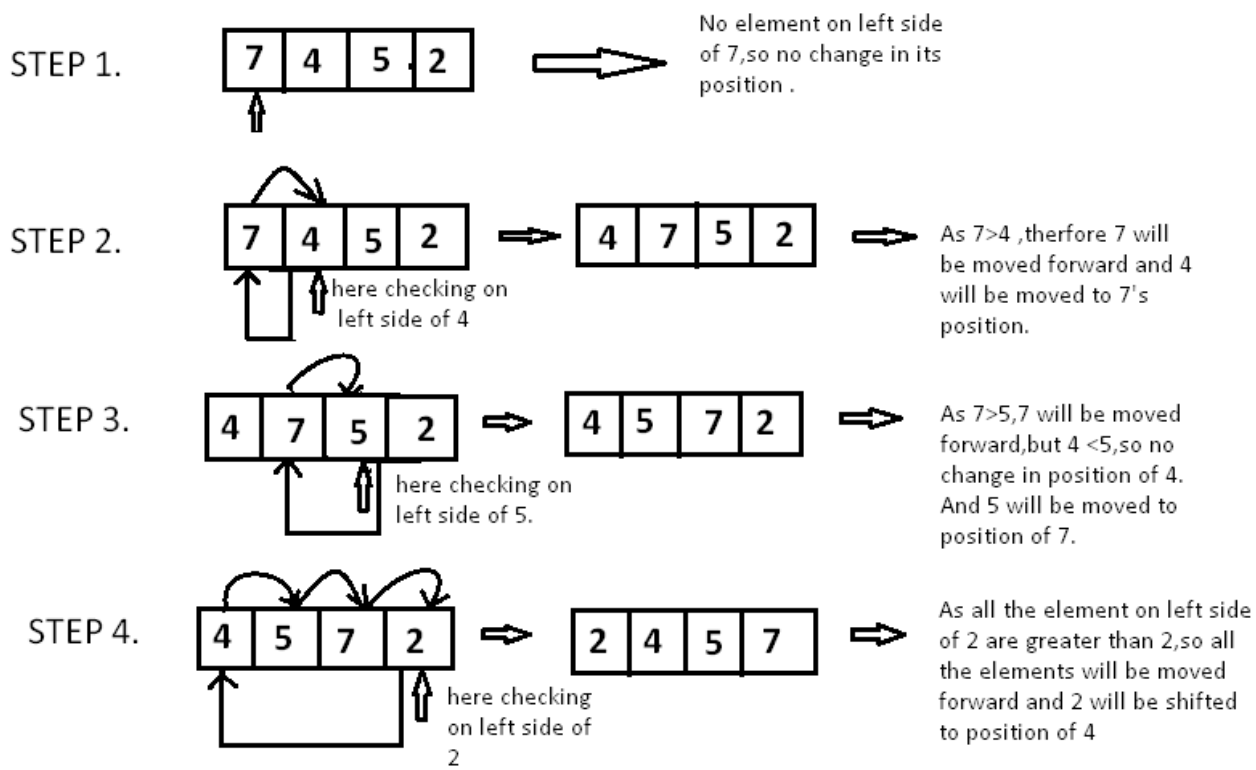
Zusammenfassung

Auch wenn die Notation des Selectionsorts in allen Fällen gleich ist kann man die Unterschiede in der Praxis deutlich merken. Das Durchlaufen eines sortierten Arrays ist deutlich schneller als das eines unsortiertes, da weniger Swap Operationen durchgeführt werden.

Quelle: https://www.scaler.com/topics/time-complexity-of-selection-sort/

Insertionsort:

Der Insertion-sort stellt einen weiteren der vier, häufig verwendeten Sortieralgorithmen dar. Dieser kann, wie auch die drei weiteren Algorithmen iterativ und rekursiv angewendet werden.



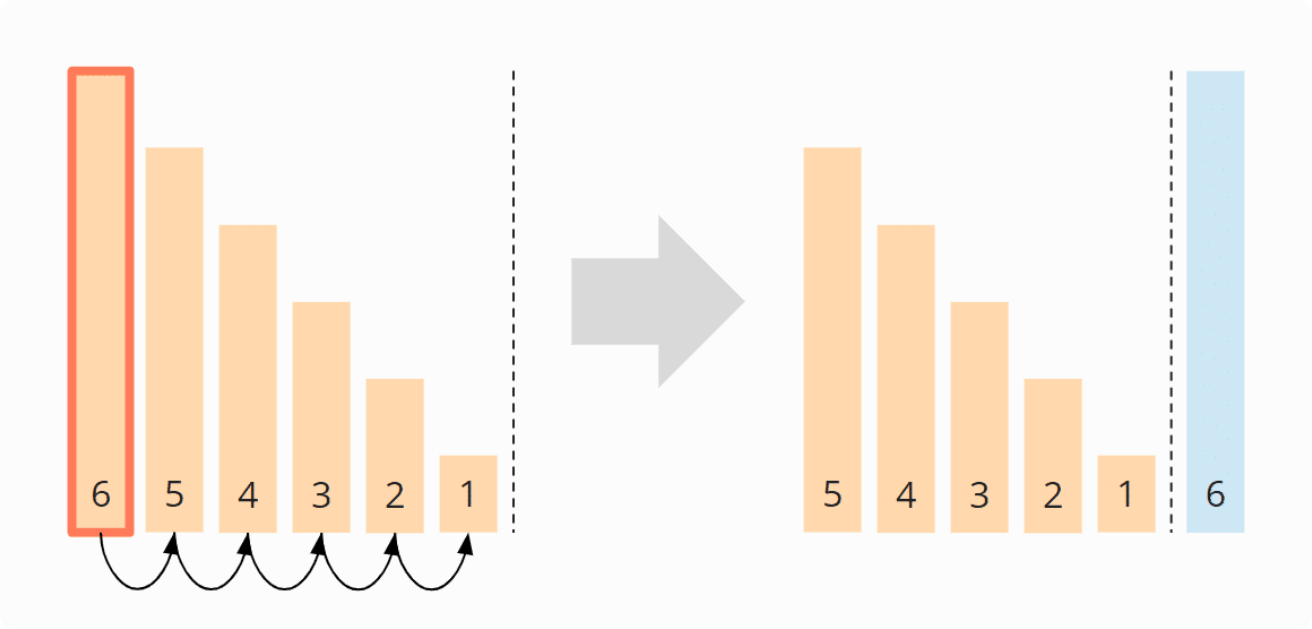
Cases

Der beste Fall (best case) zur Sortierung einer Liste liegt vor, wenn die Zahlen bereits in der geordneten Reihenfolge vorliegen. In diesem Fall müssen die

Werte miteinander verglichen, jedoch nicht vertauscht werden, wodurch die Zeitkomplexität des Algorithmus $O(n)$ beträgt. Die zwei for-Schleifen des Algorithmus deuten darauf hin, dass in dem durchschnittlichen Fall (average case) ein quadratischer Aufwand nötig ist, um die Liste vollständig zu sortieren. Der Aufwand kann zudem durch die Funktion f(n)= n \cdot (n-1) \cdot \frac{1}{4} beschrieben werden, wobei n die Anzahl der Elemente der Liste angibt, und $f(n)$ die benötigten Verschiebe-Operationen beschreibt. Diese Formel lässt sich bilden, da bei einer Liste mit n Elemente $n-1$ Verschiebe-Operationen vorgenommen werden müssen. Diese müssen jedoch in zwei Schritten durch $\frac{1}{2}$, und somit insgesamt durch $\frac{1}{4}$ geteilt werden, da in dem durchschnittlichen Fall bereits die hälfte aller Elemente der Liste an dem richtigen Platz stehen und somit nicht verschoben werden müssen, und, da das einzusortierende Element in dem Durchschnitt der Fälle nicht an den Anfang oder das Ende, sondern in die Mitte der bereits sortierten Elemente verschoben werden muss. Wird der Funktionsterm der Funktion f ausmultipliziert, so ergibt sich die Funktion f(n)= 1/4 n^2-\frac{1}{4} \cdot n. Da n^2 den größten Einfluss auf die Werte der Funktion f besitzt, und bei einer Näherung an ∞ ausschlaggebend ist, kann die Funktion f in $f(n) = n^2$ umgeformt werden. Die Zeitkomplexität in dem durchschnittlichen Fall ist somit $O(n^2)$. In dem aufwendigsten Fall (worst case) lässt sich eine ähnliche Funktion definieren. In diesem Fall wird der Wert von $n \cdot (n-1)$ jedoch nur ein einziges Mal durch $\frac{1}{2}$ geteilt, da die Wahrscheinlichkeit, dass das Element, nicht, wie bei dem durchschnittlichen Fall in die Mitte, sondern an den Anfang der Liste verschoben werden muss. Die Wahrscheinlichkeit dieser Verschiebung besitzt aufgrund des worst case-Szenarios somit einen Wert von eins. Die neue Funktion g lässt sich somit als $g(n) = \frac{1}{2}n^2 - \frac{1}{2}n$ definieren. Aufgrund derselben Ursachen, wie bei dem average case kann die Zeitkomplexität in dem worst case somit als $O(n^2)$ beschrieben werden. Die Zeitkomplexitäten der rekursiven Implementierung des insertion-sort sind identisch mit den Zeitkomplexitäten der iterativen Implementierung.

Bubblesort:

Der Bubblesort ist ein Algorithmus indem 2 verzweigte for-Schleifen das Array durchgehen weswegen dieser auch die Notation $O(n^2)$ besitzt.



Average case

Im gegensatz zum Insertion- oder Selectionsort ist der average case des Bubblesorts deutlich langsamer. Dies liegt daran, dass die Swap-Operation des Bubblesort viel Zeitintensiver als die der anderen Algorithmen ist. Beim average Case geht man davon aus, dass die hälfte des Arrays bereits sortiert ist, wodurch der Algorithmus nur noch die hälfte an Operationen durchführen muss. Die Anzahl an Swap-Operation wären in diesem Fall also:

$$\frac{1}{4}(n^2 - n)$$

Die Anzahl der Vergleichs-Operationen des Bubblesorts ist:

$$\frac{1}{2}(n^2 - n \cdot ln(n) - (y + ln(2) - 1) \cdot n) + O(\sqrt{n})$$

was vereinfacht also die Notation $O(n^2)$ ergibt.

Quelle: https://www.happycoders.eu/algorithms/bubble-sort/

Test und Ergebnisse

Allgemeines

Bei diesem Test wurde alle Sortieralgorithmen mit den selben 3 Datensätzen getestet. Hierbei gab es jeweils ein Array mit 1000, 10000 und 100000 zufällig sortieren Integern. Alle Tests wurden auf dem selben Computer durchgeführt um die Testergebnisse nicht durch unterschiedlicher Hardware verfälschen zu lassen. Bei der genutzen CPU handelt es sich hierbei um den von Apple hergestellten M2 Prozessor. Alle Testergenisse wurden in VS Code mit der Extention Code Runner durchgeführt. Die Resultierenden Zeiten wurden mit der Java Methode `System.currentTimeMillis()` und der Rechnung $stopTime - startTime$ in ms berechnet.

Quicksort

Array mit 1000 Integern:

Durchlauf	Zeit in ms
1	1
2	1
3	> 1
4	> 1
5	> 1
6	> 1
7	> 1
8	> 1
9	> 1
10	> 1

Der Durchschnitt beträgt also: $0.2ms$

Array mit 10000 Integern

Durchlauf	Zeit in ms
1	3
2	1
3	> 1
4	1
5	> 1
6	1
7	> 1
8	1
9	> 1
10	1

Der Durchschnitt beträgt also: $0.8ms$

Array mit 100000 Integern

Durchlauf	Zeit in ms
1	24
2	12
3	12
4	12
5	11
6	11
7	17
8	12
9	11
10	10

Der Durchschnitt beträgt also: $13.2ms$

Selectionsort

Array mit 1000 Integern

Durchlauf	Zeit in ms
1	3
2	2
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Der Durchschitt beträgt also: $1.3ms$

Array mit 10000 Integern

Durchlauf	Zeit in ms
1	41
2	40
3	53
4	42
5	40
6	39
7	38
8	38
9	38
10	38

Der Durchschnitt beträgt also: $40.7ms$

Array mit 100000 Integern

Durchlauf	Zeit in ms
1	3230
2	3240
3	2234
4	2045
5	2046
6	2041
7	2043
8	2028
9	2023
10	2013

Der Durschnitt beträgt also: $2093ms$

Insertionsort

Array mit 1000 Integern

Durchlauf	Zeit in ms
1	3
2	1
3	1
4	0
5	0
6	1
7	1
8	1
9	1
10	1

Der Durchschnitt beträgt: $0.9ms$

Array mit 10000 Integern

Durchlauf	Zeit in ms
1	20
2	19
3	7
4	6
5	5
6	6
7	6
8	6
9	6
10	7

Der Durschnitt beträgt: $8.2ms$

Array mit 100000 Integern

Durchlauf	Zeit in ms
1	630
2	672
3	587
4	582
5	583
6	583
7	586
8	584
9	584
10	587

Der Durschnitt beträgt also: $59.02ms$

Bubblesort

Array mit 1000 Integern

Durlauf	Zeit in ms
1	5
2	2
3	1
4	1
5	2
6	1
7	2
8	1
9	2
10	1

Der Durschnitt beträgt also: $0.18ms$

Array mit 10000 Integern

Durlauf	Zeit in ms
1	57
2	89
3	39
4	44
5	42
6	39
7	57
8	57
9	62
10	60

Der Durschnitt beträgt also: $54.6ms$

Array mit 100000 Integern

Durlauf	Zeit in ms
1	12697
2	12590
3	10431
4	10379
5	10399
6	10369
7	10383
8	10425
9	10601
10	11423

Der Durschnitt beträgt also: $10969.7ms$

Zusammenfassung

Zusammenfassend lässt sich sagen, dass die Notation gut sind um die Zeitkomplexität eines Algorithmus mit einem worst case zu beschreiben. Wie man aber anhand der Versuche erkennen kann entspricht eine Notation in der Praxis nicht immer dem Endergebnis. Auch wenn ein Algorithmus wie Wicksort die Notation $O(n^2)$ hat, sind die Resultate um ein vielfaches besser als die des Bubblesorts.