# Die Big O-Notation

In der vorhergehenden Textlektion haben wir das Konzept der Laufzeitkomplexität kennengelernt. Die Laufzeitkomplexität ist die asymptotische Laufzeit. Oft wird dabei der schlechteste Fall betrachtet, also der Fall, für den der Algorithmus am meisten Operationen benötigt. Die Schreibweise für die asymptotische Laufzeit des schlechtesten Falls wird die Big O-Notation genannt. Sie wird auch so geschrieben: ein Algorithmus könnte beispielsweise eine Laufzeitkomplexität von $O(n)$ haben.

In dieser Textlektion wollen wir die Big O-Notation etwas detaillierter betrachten. Im Verlauf des Moduls wird sie uns noch oft begegnen.

Die Big O-Notation ist die asymptotische Laufzeit im schlechtesten Fall. Um das Konzept zu verstehen, sollten wir erst einmal verstehen, was eine asymptotische Laufzeit ist. Jeder Algorithmus hat eine Eingabegröße $n$. In der asymptotischen Laufzeitbetrachtung sind wir an der Wachstumsrate der Laufzeit **in Bezug auf die Größe der Eingabe $n$** interessiert. Dabei konzentrieren wir uns auf das Verhalten des Algorithmus, wenn $n$ gegen unendlich geht.

Wenden wir uns zur Illustration wieder unserem Buchsuchalgorithmus zu. Er ließ sich folgendermaßen beschreiben:

1. Beginne ganz links im Regal mit der Suche.
2. Wenn kein Buch an der Position vorhanden ist, beende die Suche und melde, dass der gesuchte Titel nicht gefunden wurde.
3. Wenn ein Buch vorhanden ist, schaue dir den Titel des Buches an. Wenn der Titel dem gesuchten entspricht, gib das Buch und dessen Position aus und beende die Suche.
4. Bewege dich eine Position nach rechts und wiederhole alle Schritte ab Schritt 2.

In unserem Beispiel ist $n$ die Anzahl der Bücher im Regal. In der asymptotischen Laufzeitbetrachtung interessiert uns, wie viele zusätzliche Operationen wir zur Ausführung benötigen, wenn das Regal um $n$ Bücher erweitert wird, und die Bibliothek bereits unendlich viele Bücher bereithält. In der Big O-Notation interessiert uns der schlechteste Fall. In unserem Beispiel tritt er dann ein, wenn der Titel im Regal nicht gelistet ist.

Wie viele Operationen benötigen wir in diesem schlechtesten Fall? Wir könnten argumentieren, dass der erste Schritt lediglich ein Startpunkt ist. Anschließend führen wir im schlechtesten Fall dann $n$-mal die folgenden 3 Schritte aus. Bedeutet das, dass unsere Laufzeit $3n$ beträgt? Wäre das in der Big O-Notation $O(3n)$?

Die Antwort lautet: Nein. Die Laufzeitkomplexität unseres Algorithmus ist $O(n)$. Konstanten und Faktoren, die die Wachstumsrate nicht signifikant beeinflussen, werden in der Big O-Notation quasi ignoriert. Das bedeutet, dass wir uns weniger dafür interessieren, wie viele Schritte innerhalb einer Iteration genau passieren, solange diese Anzahl konstant bleibt. Es geht in erster Linie darum, wie die Gesamtzahl der Schritte mit wachsendem $n$ steigt. Wenn wir also $3n$ Schritte haben, würden wir dies in der Big O-Notation als $O(n)$ bezeichnen, da die konstante 3 das allgemeine Wachstumsverhalten nicht verändert. Nur wenn die Anzahl der Schritte innerhalb einer Iteration selbst wieder von $n$ abhängt, ist dies für die Big O-Notation relevant.

Ebenfalls nicht signifikant beeinflusst wird die Wachstumsrate von Ausdrücken, die weniger schnell wachsen als ein anderer. Laufen in einem Algorithmus beispielsweise zwei Iterationen hintereinander ab, die beide für sich genommen, unterschiedliche Laufzeitkomplexitäten aufweisen, so zählt nur die dominante Laufzeitkomplexität. Benötigt ein Algorithmus also beispielsweise im schlechtesten Fall $n + n^2$ Schritte, ist die zugehörige Laufzeitkomplexität in Big O-Notation $O(n^2)$, denn $n^2$ wächst schneller als $n$.

Merke: Ausdrücke, die die Wachstumsrate nicht signifikant beeinflussen, werden in der Big O-Notation ignoriert. Dazu gehören:

* **Faktoren**: Die Big O-Notation für $3n$ Schritte ist $O(n)$
* **Konstanten**: Die Big O-Notation für $n+50$ Schritte ist $O(n)$
* **Dominierte Ausdrücke**: Die Big O-Notation für $n^2+n$ Schritte ist $O(n^2)$.

Nun bist du dran! In der folgenden Frage wird eine unsinnige Variante des Buchsuchalgorithmus beschrieben. Deine Aufgabe ist es, die korrekte Laufzeitkomplexität in der Big O-Notation zu ermitteln.

QUIZ: In dem folgenden Algorithmus soll die suchende Person bei jeder unerfolgreichen Betrachtung eines Titels das Regal einmal komplett durchlaufen, ohne es zu durchsuchen. Was ist die Laufzeitkomplexität?

1. Beginne ganz links im Regal mit der Suche.
2. Wenn kein Buch an der Position vorhanden ist, beende die Suche und melde, dass der gesuchte Titel nicht gefunden wurde.
3. Wenn an der Position ein Buch steht, schaue dir den Titel des Buches an. Wenn der Titel dem gesuchten entspricht, gib das Buch und dessen Position aus und beende die Suche. Ansonsten merke dir die Position.
4. Gehe eine Position nach rechts.
5. Nimm das Buch aus dem Regal und stelle es (ohne es anzusehen) wieder zurück.
6. Wenn du an der gemerkten Position bist, fahre mit 7) fort. Wenn rechts kein Buch steht, beginne wieder ganz links im Regal. Ansonsten fahre mit 4) fort.
7. Bewege dich ein Buch nach rechts und wiederhole alle Schritte ab 2)

Lösung (ausklappbare Box): Wie vorhin auch ist der schlechteste Fall der, bei dem sich der Titel nicht im Regal befindet. In jedem Iterationsschritt muss die ausführende Person zusätzlich zu den „normalen“ 3 Schritten (2, 3, 7) noch das gesamte Regal durchsuchen. Hierbei entsteht eine zusätzliche Iteration innerhalb der schon vorhandenen Iteration. In dieser neuen Iteration entstehen in jedem Schritt wieder $3n$ Schritte (4, 5,6). Insgesamt zählen wir nun also $n(3+3n) = 3n+3n^2$ Schritte. Die Laufzeitkomplexität ist aber nicht $O(3n^2 + 3n)$ beziehungsweise $O(n+n^2)$, wenn wir Faktoren ignorieren. Der letzte Ausdruck beeinflusst nicht die Wachstumsrate in der asymptotischen Betrachtung. Daher ist die Laufzeitkomplexität $O(n^2)$.

**Vertiefung: die formale Definition von Big O für Laufzeitkomplexitäten.** Die Anzahl der benötigen Operationen in Abhängigkeit von der Eingabegröße $n$ können wir als Funktion $f(n)$ darstellen. Sprechen wir von Algorithmen, ist diese Funktion an allen Stellen von $n$ positiv. Wir suchen dann eine Funktion $g(n)$, die folgende Eigenschaft erfüllt:

*Es gibt Konstanten $c>0$ und $k>0$, so dass $f(n) <=cg(n)$ für alle $k>=n$ gilt.*

Ist das erfüllt, so sagen wir, dass der Algorithmus, dargestellt durch $f(n)$, eine Laufzeitkomplexität von $O(g(n))$ hat. Wir wählen die Laufzeitkomplexität immer so, dass $g(n)$ möglichst klein wird für hinreichend große $n$. Das schließt ein, dass Konstanten auf 0 und Faktoren auf 1 gesetzt werden.

Das bedeutet, dass die Anzahl der benötigten Operationen in der asymptotischen Betrachtung (hierfür steht die Bedingung $k>=n$) nie die Funktion $g(n)$ multipliziert mit einer noch so großen Konstante $c$, übersteigt. In unserem Beispiel ist die Laufzeitkomplexität $O(n)$. Die Anzahl der benötigten Operationen ist aber $3n$. Es reicht dann im Sinne der formalen Definition, Folgendes zu zeigen: $3n<= cn$ für alle $n$, die groß genug sind. In diesem Beispiel könnten wir $c=3$ oder aber auch irgendein größeres $c$ wählen.

## Lineare Laufzeiten sind grade noch so akzeptabel

Wir haben oben gesehen, dass Konstanten, Faktoren und dominierte Ausdrücke in der Big O-Notation ignoriert werden. Dadurch bilden sich einige typische Laufzeitkomplexitäten heraus. Die wichtigsten sind, in aufsteigender Reihenfolge:

$O(1)$ - konstante Laufzeitkomplexität: Die beste zu erreichende Laufzeitkomplexität ist die konstante, geschrieben als **$O(1)$.** In dem Fall ist die Anzahl an auszuführenden Schritten vollkommen unabhängig von der Datenmenge. In der formalen Definition benötigt der Algorithmus dann maximal $c$ Schritte, unabhängig davon, wie groß der Eingabewert $n$ ist.

$O(\log n)$ - logarithmische Laufzeitkomplexität: Mit solchen Laufzeitkomplexitäten haben wir es oft in Algorithmen zu tun, die auf dem so genannten *Divide-And-Conquer* Prinzip basieren. Hierbei wird das eigentliche Problem immer wieder in analoge Unterprobleme aufgeteilt. Wäre unsere Bibliothek alphabetisch aufsteigend nach Titeln sortiert, könnten wir beispielsweise folgenden Algorithmus verwenden:

1. Falls das Regal leer ist, beende die Suche. Ansonsten beginne mit dem Buch in der Mitte des Regals. Ist die Anzahl der Bücher gerade, nimm das Buch direkt links von der Mitte.
2. Prüfe den Titel des Buches. Stimmt er mit dem gesuchten Titel überein, gib den Buchtitel und seine Position im Regal aus und beende die Suche.
3. Falls nicht, entscheide basierend auf dem Anfangsbuchstaben des gesuchten Titels: Liegt dieser im Alphabet vor dem Buchtitel des aktuellen Buches, fahre mit der linken Hälfte des Regals fort. Liegt er dahinter, konzentriere dich auf die rechte Hälfte. Starte den Suchprozess in dem ausgewählten Bereich erneut von Schritt 1.

Hierbei handelt es sich um einen klassischen Binärsuchalgorithmus. Bei jeder Iteration wird hierbei die Suchmenge halbiert. Hat unsere Regal beispielsweise 8 Bücher, würden wir uns im ersten Iterationsschritt noch 4 Bücher betrachten, im zweiten 2, im dritten 1 Buch. Die maximale Anzahl an Iterationsschritten ist also 3, ob das Buch sich nun im Regal befindet oder nicht. Diesen Zusammenhang können wir auch ausdrücken als $2\*2\*2=8$ beziehungsweise $2^x=8$ mit $x=3$. Der Logarithmus zur Basis 2 beschreibt dieses $x$ und damit die Anzahl an benötigten Schritten. Die Funktion ist also nichts anderes als die inverse quadratische Funktion. In der Informatik ist mit dem Logarithmus in der Regel der Logarithmus zur Basis 2 gemeint.

$O(n)$ - lineare Laufzeitkomplexität: Ein Beispiel für einen Algorithmus mit linearer Laufzeitkomplexität haben wir bereits kennengelernt. Eine solche Laufzeitkomplexität ist sehr viel weniger wünschenswert als die konstante oder die logarithmische. In vielen Situationen müssen wir jedoch einmal alle Elemente betrachtet haben, um das gewünschte Ergebnis zu erhalten. Das Durchsuchen eines unsortierten Regals ist ein Beispiel. Wir bezeichnen diesen Algorithmus übrigens als linearen Suchalgorithmus. Andere Beispiele sind das Finden des maximalen Werts in einer numerischen Liste, oder das Bilden der Summe aller Elemente. Lineare Laufzeitkomplexitäten müssen wir in vielen Fällen in Kauf nehmen.

Laufzeitkomplexitäten jenseits der linearen sind nicht wünschenswert. Bei einer quadratischen Laufzeitkomplexität kann ein Algorithmus bei 1.000 Elementen schon 1.000.000 Operationen benötigen, bei 10.000 sind es dann bereits 100.000.000 Operationen. Selten müssen wir Laufzeitkomplexitäten jenseits der linearen in Kauf nehmen. Das Sortieren einer Liste in Python ist beispielsweise nur mit einer Laufzeitkomplexität von $O(n \log n)$ möglich.

Hier sehen wir die wichtigsten Laufzeitkomplexitäten und ihre Reihenfolge:

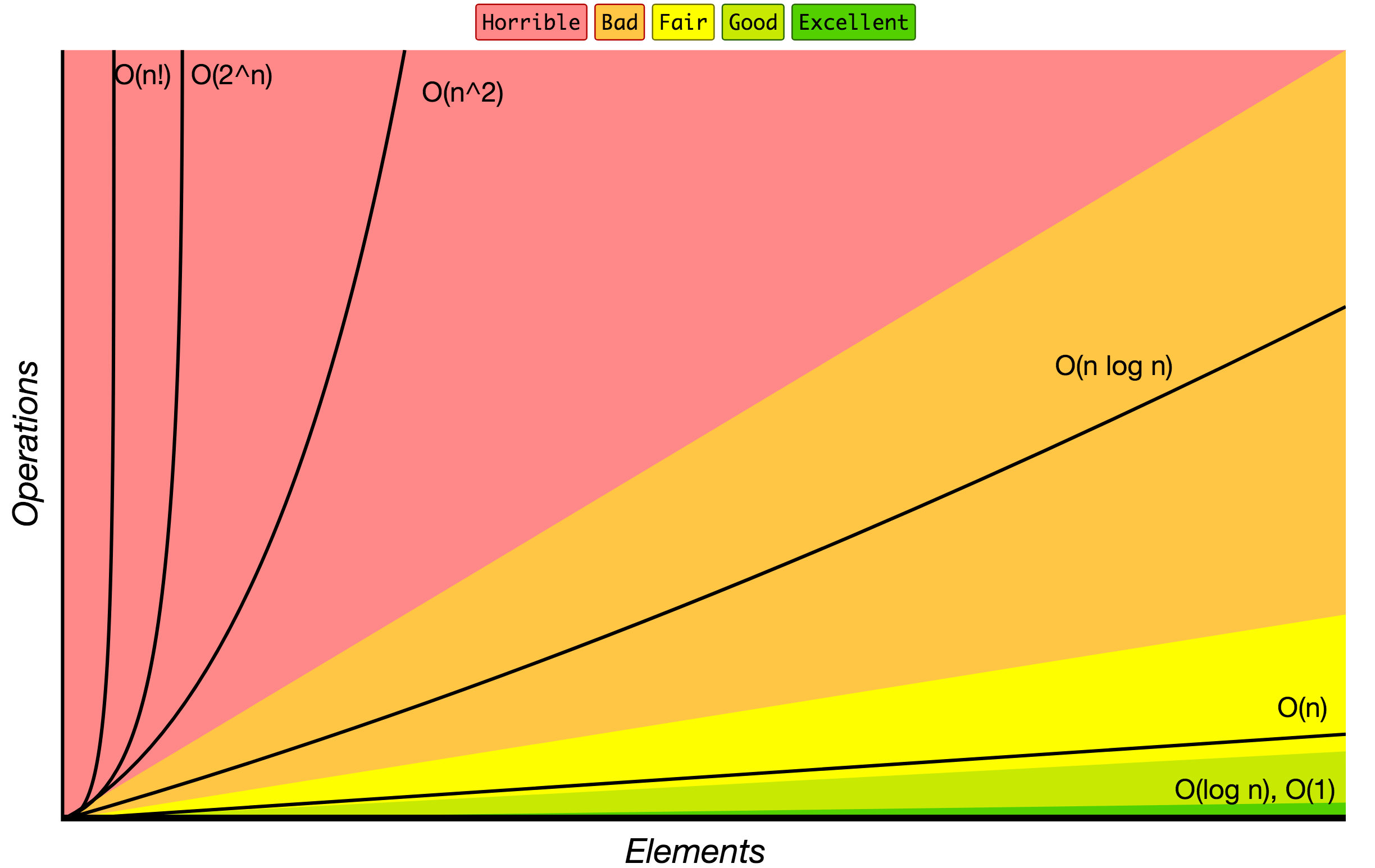


Abbildung 1: Die wichtigsten Laufzeitkomplexitäten

Wir werden uns in diesem Modul recht ausführlich mit der Laufzeiteffizienz von Algorithmen beschäftigen. Beantworte zur Vorbereitung die folgenden beiden Fragen, bevor du mit der ersten Übung beginnst!

Frage 2: Nimm an, dir stände ein Algorithmus zur Verfügung, der Bücher in $O(n \log n)$ sortieren könnte. Was ist die Laufzeitkomplexität für sortieren **und** suchen, wenn du anschließend den Binärsuchalgorithmus verwendest, der eine Laufzeitkomplexität von $O(\log n)$ hat? Ist ein solcher Algorithmus der linearen Suche vorzuziehen?

1. O(\log n) – ja, er ist besser als die lineare Suche
2. O(n \log n)\* - nein, er ist schlechter als die lineare Suche
3. O(n^2 \log n) - nein, er ist schlechter als die lineare Suche
4. O(n (\log n)^2) - nein, er ist schlechter als die lineare Suche
5. O((n+1) \log n) - nein, er ist schlechter als die lineare Suche

Lösung: wir hatten oben gesehen, dass dominierte Ausdrücke für die Wachstumsrate keine Rolle spielen. Hier dominiert das Sortieren mit $O(n \log n)$ die Suche mit $O(\log n)$. Die gesamte Laufzeitkomplexität ist daher $O(n \log n)$ und damit schlechter als die lineare Suche im unsortierten Regal.

Frage 3 (optional): Zeige anhand der formalen Definition der Big O-Notation für Algorithmen, dass die Laufzeitkomplexität in der folgenden Variante unseres Buchsuchalgorithmus $O(n^2)$ ist. Welches ist der kleinste ganzzahlige Wert von $c$, den du wählen kannst? Hinweis: Für die Lösung benötigst du die Gaußsche Summenformel: $\sum\_{i = 1}^{n} i = n\*(n+1)/2$.

1. Beginne ganz links im Regal mit der Suche.
2. Wenn kein Buch an der Position vorhanden ist, beende die Suche und melde, dass der gesuchte Titel nicht gefunden wurde.
3. Ansonsten gehe zurück zum Anfang des Regals, und wieder zurück zur zuletzt überprüften Position.
4. Bewege dich ein Buch nach rechts.
5. Wenn rechts kein Buch steht, beende die Suche, ansonsten wiederhole alle Schritte ab 2)

Lösung (aufklappbare Box): Zählen wir wieder alle Operationen einzeln, so sind an jeder Stelle wieder 3 Operationen notwendig. Zusätzlich zu diesen fallen an Position $i$ noch $2i$ Operationen an, für das Hin- und Zurücklaufen. Die Funktion $f(n)$ lässt sich dann durch $f(n)=\sum\_{i = 1}^{n} (3+2i)$ darstellen. Mit Hilfe der Gaußschen Summenformel lässt sich das zu $f(n)=4n+n^2$ umwandeln. Nun können wir das in die formale Definition von Big O einsetzen. Wir möchten zeigen, dass $4n+n^2<= cn^2$ für mindestens eine Konstante $c$, und hinreichend große Werte von $n$ gilt. Umformen ergibt, dass die Bedingung für $c=2$ und $n>=4$ erfüllt ist.

Übrigens: Manchmal gibt es in der Praxis Algorithmen, die nur in seltenen Fällen eine ungewöhnlich hohe Anzahl an Operationen erfordern. In solchen Fällen ist die Big O-Notation nicht ideal. Wir könnten beispielsweise unseren Buchsuchalgorithmus derart modifizieren, dass er nur für den Fall, dass wir einen spezifischen und selten gesuchten Titel suchen, eine extrem hohe Laufzeit aufweist. Dann wäre eine Durchschnittsanalyse aussagekräftiger.

Merke:

* Die Big O-Notation repräsentiert die obere Grenze des Wachstums einer Funktion und wird verwendet, um die Laufzeitkomplexität von Algorithmen zu charakterisieren.
* Faktoren, Konstanten und dominierte Ausdrücke werden in der Big O-Notation ignoriert.
* Der Binärsuchalgorithmus liefert mit einer Laufzeitkomplexität von $O(\log n)$ sehr schnell Ergebnisse, setzt aber sortierte Daten voraus.
* Die lineare Suche ist mit einer Laufzeitkomplexität von $O(n)$ im Vergleich weniger schnell, aber benötigt keine sortierten Daten.
* In speziellen Fällen, in denen der schlechteste Fall äußerst selten ist und eine äußerst hohe Laufzeit hat ist die Durchschnittsanalyse aussagekräftiger.