



i Hinweis: KI-generierte Musterlösung - bei Unklarheiten gerne nochmal im Buch nachschlagen.

AUFGABE 1a

TIPP: Bei Insertionsort betrachtest du das Array von links nach rechts. Der Teil links vom aktuellen Element gilt als "bereits sortiert", das aktuelle Element wird an der richtigen Stelle in diesen sortierten Teil eingefügt.

SCHRITT 1: Start: 6, 2, 9, 4, 1. Das erste Element (6) gilt als sortierter Anfangsbereich, da ein einzelnes Element immer sortiert ist.

SCHRITT 2: Durchgang 1 - Einfügen der 2: Die 2 wird mit der 6 verglichen. Da 2 kleiner ist als 6, wird die 6 nach rechts verschoben und die 2 davor eingefügt.

ZWISCHENERGEBNIS: 2, 6, 9, 4, 1

SCHRITT 3: Durchgang 2 - Einfügen der 9: Die 9 wird mit der 6 verglichen. Da 9 größer ist als 6, bleibt die 9 an ihrer Stelle stehen, es muss nichts verschoben werden.

ZWISCHENERGEBNIS: 2, 6, 9, 4, 1

SCHRITT 4: Durchgang 3 - Einfügen der 4: Die 4 wird zuerst mit der 9 verglichen (4 kleiner, 9 rückt nach rechts), dann mit der 6 (4 kleiner, 6 rückt nach rechts), dann mit der 2 (4 größer als 2, Einfügen direkt dahinter).

ZWISCHENERGEBNIS: 2, 4, 6, 9, 1

SCHRITT 5: Durchgang 4 - Einfügen der 1: Die 1 wird mit 9, 6, 4 und 2 verglichen, ist jeweils kleiner, sodass alle vier Werte um eine Position nach rechts rutschen. Die 1 wird ganz vorne eingefügt.

ZWISCHENERGEBNIS: 1, 2, 4, 6, 9

ENDERGEBNIS: Zustände nach jedem Durchgang: 2,6,9,4,1 / 2,6,9,4,1 / 2,4,6,9,1 / 1,2,4,6,9. Das Array ist am Ende vollständig sortiert: 1, 2, 4, 6, 9.





AUFGABE 1b

TIPP: Zähle für jedes einzufügende Element separat mit, wie viele bereits einsortierte Elemente tatsächlich verglichen und bei Bedarf um eine Position nach rechts verschoben werden mussten.

SCHRITT 1: Einfügen der 2: Die 6 muss um eine Position verschoben werden, damit die 2 davor passt. Das sind 1 Verschiebung.

SCHRITT 2: Einfügen der 9: Die 9 ist bereits größer als die 6 davor, also ist keine Verschiebung nötig. Das sind 0 Verschiebungen.

SCHRITT 3: Einfügen der 4: Hier müssen die 9 und die 6 jeweils eine Position nach rechts rutschen, bevor die 4 hinter der 2 eingefügt wird. Das sind 2 Verschiebungen.

SCHRITT 4: Einfügen der 1: Hier müssen die 9, die 6, die 4 und die 2 jeweils eine Position nach rechts rutschen, bevor die 1 ganz vorne eingefügt wird. Das sind 4 Verschiebungen.

SCHRITT 5: Addiere alle Verschiebungen der vier Durchgänge: $1 + 0 + 2 + 4 = 7$.

ENDERGEBNIS: Insgesamt waren 7 Verschiebeoperationen nötig (gezählt wird dabei jede einzelne Verschiebung eines Elements um eine Position nach rechts, nicht erst das endgültige Einfügen).

AUFGABE 2a

TIPP: Überlege dir, was mit der Laufzeit passiert, wenn du n verdoppelst - einmal bei $O(n)$, einmal bei $O(n^2)$.

SCHRITT 1: $O(n)$ bedeutet, dass die Laufzeit linear mit der Eingabegröße wächst: Verdoppelt sich n , verdoppelt sich ungefähr auch die Laufzeit.

SCHRITT 2: $O(n^2)$ bedeutet, dass die Laufzeit quadratisch wächst: Verdoppelt sich n , vervierfacht sich die Laufzeit (da $2 \text{ hoch } 2 = 4$ ist).

SCHRITT 3: Für sehr große n wächst n^2 also deutlich schneller als n . Ein $O(n)$ -Algorithmus bleibt bei großen Eingaben praktisch immer schneller bzw. skaliert deutlich besser als ein $O(n^2)$ -Algorithmus.

SCHRITT 4: Die Landau-Notation beschreibt dabei nur das grundsätzliche Wachstumsverhalten (die Größenordnung) der Laufzeit in Abhängigkeit von n , nicht die exakte Laufzeit in Sekunden und nicht das Verhalten bei kleinen n .

ENDERGEBNIS: $O(n)$ wächst linear, $O(n^2)$ wächst quadratisch. Für sehr große Eingabegrößen n ist ein $O(n)$ -Algorithmus deutlich effizienter, da seine Laufzeit viel langsamer ansteigt als die eines $O(n^2)$ -Algorithmus.





AUFGABE 2b

TIPP: Erwinnere dich an die typischen Eigenschaften der drei Algorithmen: Wie viele verschachtelte Schleifen braucht Bubblesort, wie funktioniert das Teilen-und-Verschmelzen bei Mergesort, und wie halbiert die binäre Suche den Suchraum?

SCHRITT 1: Bubblesort vergleicht im schlechtesten Fall jedes Element mit (fast) jedem anderen Element durch zwei ineinander verschachtelte Schleifen. Das führt zu quadratischem Wachstum.

SCHRITT 2: Mergesort teilt das Array immer wieder in der Mitte (das ergibt $\log n$ Teilungsebenen) und muss auf jeder Ebene alle n Elemente beim Verschmelzen durchlaufen. Das ergibt n mal $\log n$.

SCHRITT 3: Die binäre Suche halbiert bei jedem Schritt den verbleibenden Suchbereich, sodass nur noch logarithmisch viele Schritte nötig sind, bis nur noch ein Element übrig ist.

ENDERGEBNIS: Bubblesort gehört zu $O(n^2)$, Mergesort gehört zu $O(n \cdot \log n)$, Binäre Suche (auf sortiertem Array) gehört zu $O(\log n)$.

AUFGABE 3a

TIPP: Denke an das Prinzip "Teile und Herrsche" - zuerst wird zerlegt, dann wird Stück für Stück wieder zusammengesetzt.

SCHRITT 1: In der Teilen-Phase (Divide) wird das Array so lange in der Mitte halbiert, bis nur noch einzelne Elemente übrig sind, die für sich genommen automatisch sortiert sind.

SCHRITT 2: In der Verschmelzen-Phase (Merge) werden jeweils zwei bereits sortierte Teil-Arrays zusammengeführt, indem man ihre kleinsten noch verbliebenen Elemente miteinander vergleicht und das kleinere jeweils in das Ergebnis-Array übernimmt.

SCHRITT 3: Dieses Verschmelzen wird auf jeder Ebene wiederholt, bis am Ende wieder ein einziges, nun vollständig sortiertes Array entsteht.

SCHRITT 4: Das Grundprinzip ist also rekursiv: Ein großes Sortierproblem wird in viele kleine, leicht lösbare Teilprobleme zerlegt, deren Lösungen anschließend systematisch wieder zusammengefügt werden.

ENDERGEBNIS: Mergesort zerlegt das Array rekursiv so lange, bis einzelne Elemente übrig bleiben (Teilen), und führt diese anschließend paarweise sortiert wieder zusammen, bis das gesamte Array sortiert ist (Verschmelzen).





AUFGABE 3b

TIPP: Vergleiche die Laufzeitklassen beider Algorithmen aus Aufgabe 2b und überlege, was das für sehr große Datenmengen praktisch bedeutet.

SCHRITT 1: Bubblesort hat im schlechtesten Fall eine Laufzeit von $O(n^2)$, Mergesort dagegen nur $O(n \cdot \log n)$.

SCHRITT 2: Bei kleinen Datenmengen fällt der Unterschied kaum auf, aber bei sehr großen n wächst n^2 extrem viel schneller als $n \cdot \log n$.

SCHRITT 3: Das bedeutet in der Praxis: Bei Millionen von Datensätzen würde Bubblesort unrealistisch lange brauchen, während Mergesort auch bei solchen Datenmengen noch in vertretbarer Zeit fertig wird.

ENDERGEBNIS: Mergesort wird bei sehr großen Datenmengen bevorzugt, weil seine Worst-Case-Laufzeit $O(n \cdot \log n)$ deutlich langsamer wächst als die quadratische Worst-Case-Laufzeit $O(n^2)$ von Bubblesort, wodurch Mergesort bei großen n erheblich schneller ist.

AUFGABE 3c

TIPP: Die Landau-Notation beschreibt nur das Verhalten für sehr große n - überlege, ob das auch für kleine n automatisch gelten muss.

SCHRITT 1: Die Aussage bezieht sich auf das asymptotische Verhalten, also auf das Verhalten für sehr große Eingabegrößen n .

SCHRITT 2: Für sehr große n stimmt es, dass ein $O(n^2)$ -Algorithmus irgendwann langsamer wird als ein $O(n \cdot \log n)$ -Algorithmus, da n^2 schneller wächst als $n \cdot \log n$.

SCHRITT 3: Für kleine n kann es jedoch anders aussehen: Ein $O(n^2)$ -Algorithmus kann durch einen einfacheren Aufbau (kleinere konstante Faktoren, weniger Verwaltungsaufwand) bei kleinen Eingaben sogar schneller sein als ein komplizierterer $O(n \cdot \log n)$ -Algorithmus.

ENDERGEBNIS: Die Aussage des Schülers ist in dieser absoluten Form nicht korrekt. Die Landau-Notation trifft nur eine Aussage über das Verhalten bei sehr großen n (asymptotisches Verhalten). Für kleine Eingabegrößen kann ein $O(n^2)$ -Algorithmus trotzdem schneller sein als ein $O(n \cdot \log n)$ -Algorithmus, da konstante Faktoren und der Verwaltungsaufwand dort noch eine größere Rolle spielen.





AUFGABE 4a

TIPP: Prüfe, wie sich die Laufzeit verändert, wenn sich n verdoppelt - das verrät dir, ob linear oder quadratisches Wachstum vorliegt.

SCHRITT 1: Betrachte den Übergang von $n=100$ auf $n=200$ (eine Verdopplung von n): Die Laufzeit steigt von 0,01 s auf 0,04 s, also auf das Vierfache.

SCHRITT 2: Betrachte den Übergang von $n=200$ auf $n=400$ (ebenfalls eine Verdopplung von n): Die Laufzeit steigt von 0,04 s auf 0,16 s, also wieder auf das Vierfache.

SCHRITT 3: Bei $O(n)$ müsste sich die Laufzeit bei einer Verdopplung von n ebenfalls nur verdoppeln. Bei $O(n^2)$ müsste sie sich vervierfachen (da $2 \text{ hoch } 2 = 4$ ist).

SCHRITT 4: Da sich die Laufzeit in beiden Fällen tatsächlich jeweils vervierfacht hat, passt der gemessene Verlauf zur quadratischen Laufzeitklasse.

ENDERGEBNIS: Die Messwerte passen zu $O(n^2)$, da eine Verdopplung von n jeweils zu einer Vervierfachung der Laufzeit führt - genau das erwartete Verhalten bei quadratischem Wachstum.

AUFGABE 4b

TIPP: Nutze dieselbe Gesetzmäßigkeit wie in Aufgabe a): Was passiert mit der Laufzeit, wenn n erneut verdoppelt wird?

SCHRITT 1: Von $n=400$ auf $n=800$ verdoppelt sich n erneut.

SCHRITT 2: Da die Laufzeitklasse $O(n^2)$ vorliegt, vervierfacht sich die Laufzeit bei jeder Verdopplung von n .

SCHRITT 3: Die Laufzeit bei $n=400$ betrug 0,16 s. Multipliziert man diese mit 4, ergibt sich die zu erwartende Laufzeit bei $n=800$.

ZWISCHENERGEBNIS: $0,16 \text{ s} \cdot 4 = 0,64 \text{ s}$

ENDERGEBNIS: Bei $n=800$ ist eine Laufzeit von etwa 0,64 Sekunden zu erwarten.

