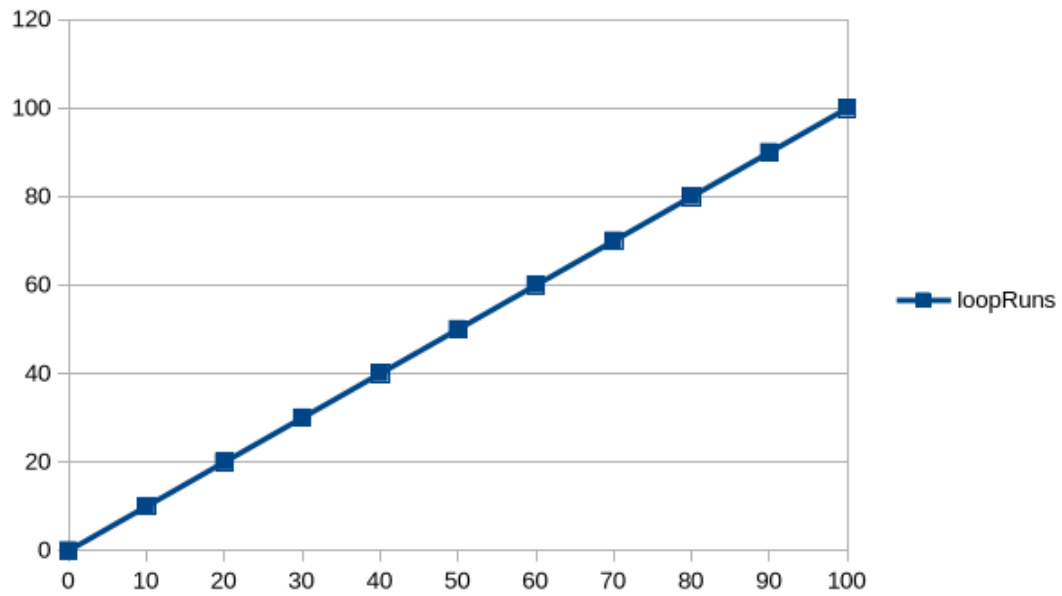
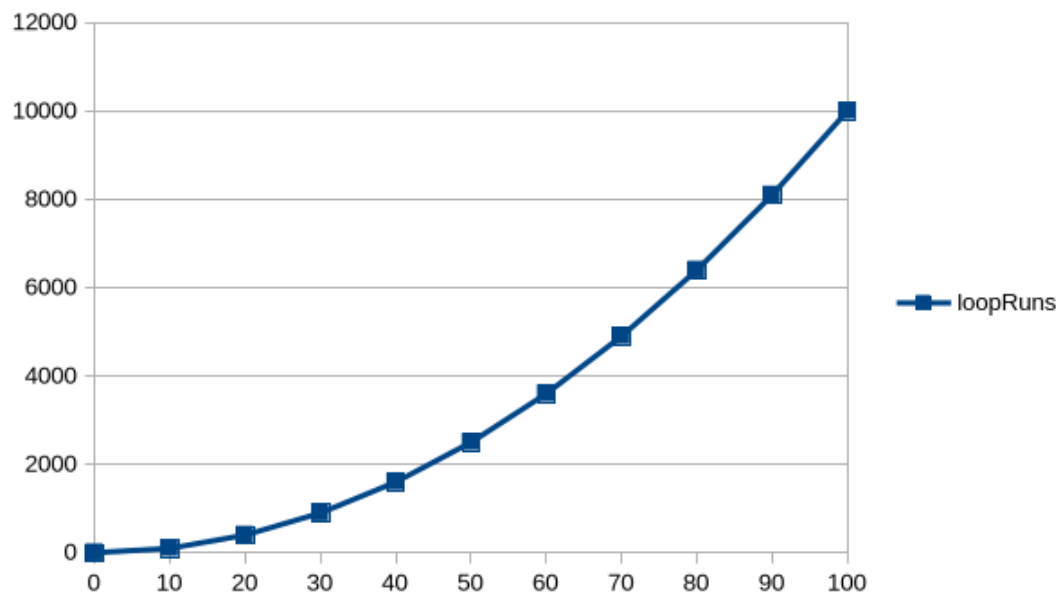


Task 1



Task 2



Task 3

Strategie & Laufzeitkomplexität

Mein Ansatz für `autoGuess()` verwendet **Binary Search** (Binäre Suche). Die Suchgrenzen starten bei `Integer.MIN_VALUE` und `Integer.MAX_VALUE`. In jedem Schritt wird die Mitte des verbleibenden Bereichs geraten und mit `isBigger()` geprüft, ob die Zielzahl größer oder kleiner ist. Dadurch halbiert sich der Suchbereich mit jedem Versuch.

Laufzeitkomplexität: $O(\log n)$, wobei n die Größe des Suchbereichs ist (hier 2^{32} für den gesamten int-Bereich). Das ergibt maximal ~ 32 Rateversuche, unabhängig von der Zielzahl.

Vergleich: Eigene Lösung vs. KI-Lösung

Die KI (Claude) hat die gleiche Aufgabe erhalten — die vollständige `NumberGuesser`-Klasse mit einer leeren `autoGuess()`-Methode zum Ausfüllen.

Ergebnis: Beide Implementierungen sind **identisch**. Sowohl meine als auch die KI-Lösung verwenden Binary Search mit denselben Suchgrenzen und derselben overflow-sicheren Berechnung der Mitte (`low + (high - low) / 2`).

Gemeinsamkeiten: - Beide nutzen Binary Search über den gesamten int-Bereich - Gleiche Initialisierung: `low = Integer.MIN_VALUE`, `high = Integer.MAX_VALUE` - Gleiche Berechnung des Mittelpunkts: `low + (high - low) / 2` (overflow-sicher) - Gleiche Abbruchbedingung: `while (low <= high)`

Unterschiede: - Keine funktionalen Unterschiede in `autoGuess()`

KI-Abschätzung der Laufzeitkomplexität

KI-Antwort: “Runtime complexity for both: $O(\log n)$ where $n = 2^{32}$ (the full int range), so at most ~ 32 iterations.”

Bewertung: Die KI liegt **richtig**. Binary Search halbiert den Suchbereich in jedem Schritt, was zu einer logarithmischen Laufzeit führt. Für den gesamten int-Bereich von 2^{32} Werten ergibt das

$\log_2(2^{32}) = 32$ Schritte im Worst Case. Dies ist deutlich effizienter als ein linearer Ansatz mit $O(n)$, der im schlimmsten Fall alle $\sim 4,3$ Milliarden Werte durchprobieren müsste.

Task 4

Strategie & Laufzeitkomplexität

Mein Ansatz für `isAnagram()` wandelt beide Wörter in char-Arrays um, sortiert diese mit `Arrays.sort()` und vergleicht die sortierten Strings. Wenn die sortierten Versionen gleich sind, handelt es sich um ein Anagramm.

Laufzeitkomplexität: $O(n \log n)$, wobei n die Länge des längeren Wortes ist. Der dominierende Faktor ist das Sortieren der char-Arrays (`Arrays.sort()` verwendet Dual-Pivot Quicksort mit $O(n \log n)$). Der anschließende Vergleich ist nur $O(n)$.

Ein effizienterer Ansatz wäre $O(n)$ mit einer HashMap/Array, die die Buchstabenhäufigkeiten zählt — allerdings ist $O(n \log n)$ für typische Wortlängen völlig ausreichend.

Vergleich: Eigene Lösung vs. KI-Lösung

Beide Lösungen verwenden den **gleichen Algorithmus**: Sortieren der Zeichen und Vergleich der sortierten Arrays.

Gemeinsamkeiten: - Beide nutzen `Arrays.sort()` auf char-Arrays - Gleicher algorithmischer Ansatz: Sort-and-Compare - Gleiche Laufzeitkomplexität: $O(n \log n)$

Unterschiede: - **Rückgabetyt:** Meine Lösung gibt `void` zurück und druckt das Ergebnis direkt auf die Konsole. Die KI gibt `boolean` zurück — sauberer, da die Methode so wiederverwendbar ist. - **Null-Check:** Die KI prüft auf null-Werte (`if (word0 == null || word1 == null)`). Meine Lösung tut dies nicht, würde also bei null-Eingaben eine `NullPointerException` werfen. - **Vergleichsmethode:** Meine Lösung erstellt neue Strings und vergleicht mit `String.equals()`. Die KI vergleicht die char-Arrays direkt mit `Arrays.equals()`, was einen

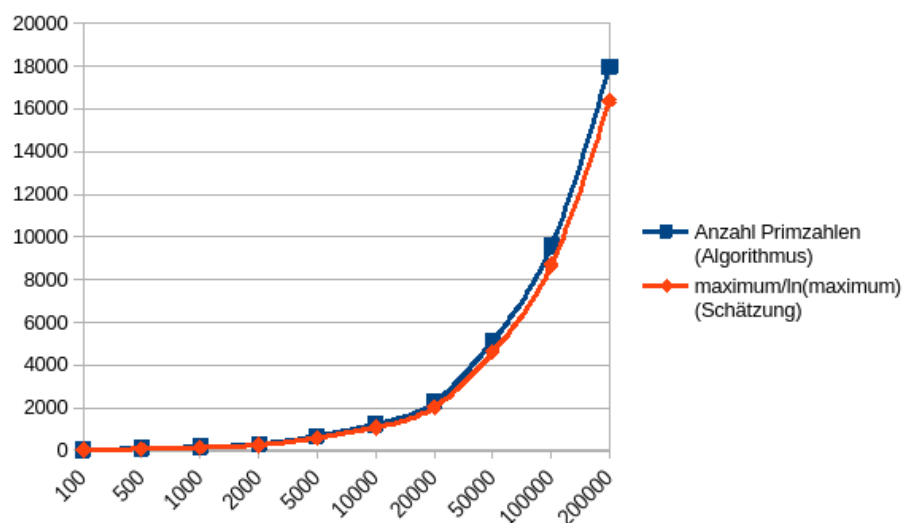
unnötigen String-Allokationsschritt spart. - **Hilfsmethode:** Meine Lösung lagert das Sortieren in eine eigene `sortString()`-Methode aus. Die KI schreibt alles inline in `isAnagram()`.

Task 5

Ergebnisse

maximum	Anzahl Primzahlen (Algorithmus)	maximum/ln(maximum) (Schätzung)
100	25	21,7
500	95	80,5
1.000	168	144,8
2.000	303	263,1
5.000	669	587,0
10.000	1.229	1.085,7
20.000	2.262	2.019,5
50.000	5.133	4.621,2
100.000	9.592	8.685,9
200.000	17.984	16.385,3

Kurvenvergleich



Primzahlen-Kurvenvergleich

Die Schätzung durch $\text{maximum}/\ln(\text{maximum})$ (Primzahlsatz) liegt durchgehend etwas **unter** der tatsächlichen Anzahl der Primzahlen, nähert sich aber mit steigendem maximum proportional an. Beide Kurven zeigen den gleichen sublinearen Wachstumsverlauf.

Aufwandsvergleich: Algorithmus vs. Formel

- **Algorithmus (Zählen durch Sieben):** Für jede Zahl bis maximum wird geprüft, ob sie prim ist. Die naive Implementierung hat eine Laufzeitkomplexität von $O(n^2)$ — für jede der n Zahlen wird im schlimmsten Fall bis n dividiert. Bei maximum = 200.000 sind das potenziell Milliarden von Divisionen.
- **Formel $\text{maximum}/\ln(\text{maximum})$:** Eine einzige Division und ein Logarithmus — $O(1)$, also konstanter Aufwand, unabhängig von der Größe von maximum.

Fazit: Die Formel ist um Größenordnungen schneller, liefert aber nur eine Schätzung. Der Algorithmus liefert das exakte Ergebnis, braucht dafür aber erheblich mehr Rechenzeit. Für große Werte ist die Formel eine sehr gute Approximation.

Vergleich: Eigene Lösung vs. KI-Lösung

Beide Lösungen liefern **identische Ergebnisse** für alle getesteten Werte. Der zentrale Unterschied liegt in der `isPrime()`-Methode:

Gemeinsamkeiten: - Gleiche Gesamtstruktur: Klasse mit `maximum`-Feld, `isPrime()`, `countPrimes()`, `printPrimeNumbers()` - Gleicher Brute-Force-Ansatz: Jede Zahl einzeln auf Primeeigenschaft prüfen

Unterschiede: - **Schleifengrenze:** Meine Lösung prüft Teiler bis `number - 1` ($O(n)$ pro Zahl). Die KI prüft nur bis $\sqrt{\text{number}}$ (`i * i <= number`), was $O(\sqrt{n})$ pro Zahl ergibt — mathematisch ausreichend, da ein Teiler $> \sqrt{n}$ immer einen Gegenteiler $< \sqrt{n}$ hat. - **Gerade Zahlen:** Die KI schließt gerade Zahlen > 2 sofort aus (`if (number % 2 == 0) return false`) und prüft danach nur ungerade Teiler (`i += 2`). Das halbiert die Anzahl der Divisionen nochmals. - **Sonderfälle:** Die KI behandelt Zahlen < 2 und die Zahl 2 explizit als Sonderfälle.

Bewertung: Die KI-Lösung ist **besser**. Durch die Optimierung der Schleifengrenze auf \sqrt{n} und das Überspringen gerader Zahlen sinkt die Gesamtkomplexität von $O(n^2)$ auf ca. $O(n\sqrt{n})$ — bei maximum = 200.000 ein erheblicher Geschwindigkeitsvorteil. Die Ergebnisse sind trotzdem identisch.

Task 6

Strategie & Laufzeitkomplexität

Mein Ansatz verwendet **Natural Merge Sort**. Der Algorithmus erkennt bereits sortierte Teilfolgen (Runs) im Array und verschmilzt jeweils benachbarte Paare. Dieser Vorgang wird wiederholt, bis das gesamte Array sortiert ist.

Laufzeitkomplexität: $O(n \log n)$ im Worst Case, $O(n)$ im Best Case (bereits sortiertes Array), da dann nur ein einziger Durchlauf ohne Merges nötig ist. Der Speicherbedarf ist $O(n)$ für das temporäre Array.

Vergleich: Eigene Lösung vs. KI-Lösung

Die KI verwendet **Introsort** — einen Hybridalgorithmus aus Quicksort, Heapsort und Insertion Sort. Dies ist der gleiche Ansatz, den die meisten Standardbibliotheken (z.B. C++ `std::sort`) verwenden.

Gemeinsamkeiten: - Beide Algorithmen haben eine Worst-Case-Komplexität von $O(n \log n)$ - Beide liefern korrekte, identische Sortier-Ergebnisse - Beide verwenden Hilfsmethoden zur Modularisierung

Unterschiede: - **Algorithmus-Familie:** Meine Lösung ist ein reiner Merge Sort (iterativ, bottom-up). Die KI kombiniert drei verschiedene Algorithmen je nach Situation. - **Adaptivität:** Meine Lösung nutzt natürliche Runs — bei teilweise vorsortierten Daten ist sie schneller. Die KI wechselt bei kleinen Teilarrays (< 16 Elemente) zu Insertion Sort und bei zu tiefer Rekursion zu Heapsort. - **Speicher:** Meine Lösung benötigt $O(n)$ zusätzlichen Speicher für das temp-Array. Die KI sortiert in-place mit nur $O(\log n)$ Speicher für den Rekursionsstack. -

Stabilität: Meine Lösung ist **stabil** (gleiche Elemente behalten ihre Reihenfolge). Die KI-Lösung ist **nicht stabil** (Quicksort-Partition kann die Reihenfolge gleicher Elemente ändern). - **Komplexität des Codes:** Meine Lösung ist deutlich kürzer und einfacher zu verstehen. Die KI-

Lösung hat 5 separate Methoden (introsort, partition, heapsort, heapify, insertionSort). - **Pivot-Wahl:** Die KI verwendet Median-of-Three für die Pivot-Wahl, was Worst-Case-Verhalten bei bereits sortierten Arrays vermeidet.

Bewertung: Beide Lösungen sind **gleichwertig gut**, mit unterschiedlichen Stärken. Meine Lösung ist einfacher, stabil und optimal bei teilweise vorsortierten Daten. Die KI-Lösung ist speichereffizienter und garantiert $O(n \log n)$ auch im Worst Case durch den Heapsort-Fallback.