Exercises

W04H03 - To Infinity And Beyond

Points	Submission due date	Status	Difficulty	Categories
0 / 10	Nov 23, 2025 17:00	No graded result		Hausaufgabe

Tasks:

To Infinity And Beyond

In dieser Aufgabe lernst Du, wie man mithilfe von Arrays mit größeren Zahlen rechnen kann, als es die Java-Basistypen zulassen.

Die 3 Entdeckuine sind schon eine sehr lange Zeit unterwegs. Eines Tages erleben sie eine sternenklare Nacht, was wegen der Schneestürme nur selten passiert. Sie fragen sich, wie viele Sterne es im Universum gibt. Sie fangen an zu zählen und stellen fest, dass sie schnell bei der Grenze der durch `long` darstellbaren Zahlen ankommen werden. Auf einmal taucht ein Entdeckuin einer anderen Kolonie aus der Dunkelheit auf. Er heißt Florian. Die 3 begrüßen ihn und bieten ihm an sich zu ihnen zu setzen. Sie erzählen ihm von deren Problem mit der begrenzten Darstellbarkeit von Zahlen. Florian fällt ein, dass er schonmal etwas von der sagenumwobenen Datenstruktur `BigInteger` gehört hat und meint, dass man diese doch einfach nehmen könnte. Den anderen gefällt die Idee sehr, nur gibt es ein Problem: Sie haben schon angefangen zu zählen und einen `long` dafür verwendet, was aber die `BigInteger`-Klasse nicht unterstützt. Deswegen brauchen sie deine Hilfe, eine eigene `BigInteger`-Datenstruktur zu implementieren. Da sie nur positive Zahlen benötigen, haben sie sich dazu entschieden, die Klasse `BigUInt` zu nennen (Big unsigned Integer). Die Entdeckuine haben dir schon einiges an Arbeit abgenommen und viele Methoden bereits implementiert.

How to arbeiten mit großen Zahlen

Falls du dich mit den verschiedenen Darstellungsformen von Zahlen auskennst und auch bitweise Operatoren kein Fremdwort für dich sind, kannst du folgenden Teil überspringen und direkt mit der Aufgabe starten. Um die Erklärungen möglichst übersichtlich zu halten, wird alles anhand von 8-Bit-großen Zahlen erklärt. Das Prinzip lässt sich aber auch auf beliebige andere Zahlen anwenden.

1. Zahlendarstellung: Es gibt verschieden Zahlensysteme, die ihre Darstellung auf ihre jeweilige Basis zurückführen. Du benutzt bereits ganz normal im Alltag das Dezimalsystem, welches die Zahlen mit der Basis 10 darstellt. Die Zahl 123 lässt sich als $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$ darstellen. Ebenso gibt es auch die in der Informatik weit verbreiteten Binär- und Hexadezimalsysteme (Basis 2 bzw. Basis 16). Da uns in $Base_{16}$ weitere Ziffern fehlen, wird hier mit Buchstaben weitergemacht. A enspricht 10, B entspricht 11 usw. bishin zu F, was 15 entspricht. Zahlen lassen sich ganz einfach von einem Zahlensystem in ein anderes umrechnen. Beispielsweise wird die Zahl 123 in $Base_{16}$ als $7 \cdot 16^1 + 11 \cdot 16^0 = 7B$ dargestellt. Die Umrechnung zwischen $Base_2$ und $Base_{16}$ ist sogar deutlich einfacher, da eine Ziffer in $Base_{16}$ exakt 4 Ziffern in $Base_2$ entspricht. Ziffern in $Base_2$ werden auch als Bits bezeichnet. Zahlen des Hexadezimalsystems bekommen üblicherweise den Präfix `0x` (bspw. `0x7B`), während Binärzahlen den Präfix `0b` bekommen (bspw. `0b1011`).
2. Bitweise Operatoren: Es gibt eine Reihe von bitweisen Operatoren, allerdings benötigst du für die Aufgabe nur den "`<<`"-, "`>>`"-/"`>>>`"-, "`|`"- und ggf. den "`&`"-Operator.
   - "`<<`": Das ist der sog. Links-Shift-Operator. Er sorgt dafür, dass alle Bits einer Zahl um x nach links verschoben werden. Das entspricht einer Multiplikation mit der 2^x. Betrachte dazu folgendes Beispiel: $0b00110110 << 2 = 0b11011000$. Hier wurde jedes Bit um 2 Stellen verschoben, was einer Multiplikation der Zahl um den Faktor 2^2 entspricht.
   - "`>>`"/"`>>>`": Das ist der sog. Rechts-Shift-Operator. Er funktioniert fast genau so wie "`<<`", aber mit dem kleinen Unterschied, dass er die einzelnen Bits nach rechts verschiebt. Dazu gibt es noch einen Unterschied, ob man "`>>`" oder "`>>>`" benutzt. Ersterer sorgt dafür, dass von links dasselbe Bit reingeschoben wird, wie das linkeste Bit in der ursprünglichen Zahl. Letzterer sorgt dafür, dass in jedem Fall eine `0` reingeschoben wird. Betrachte dazu folgende Beispiele: $0b11001001 >> 2 = 0b11110010$, während $0b11001001 >>> 2 = 0b00110010$. Da wir in dieser Aufgabe jede Bitfolge als positive Zahl (unsigned) interpretieren, solltest du immer "`>>>`" benutzen. Andernfalls kann es zu nervigen Bugs kommen, die nur schwer zu finden sind.
   - "`|`": Das ist das sog. bitweise Oder. Dieser Operator sorgt dafür, dass von beiden Zahlen alle `1`er-Bits übernommen werden. Dabei ist es egal, in welchem Operand das Bit auf `1` gesetzt ist. Betrachte dazu folgendes Beispiel: $0b10101100 \mid 0b01101001 = 0b11101101$. Dieser Operator ist also hervorragend dazu geeignet, disjunkte Zahlenteile zu vereinen: $0b11010000 \mid 0b00001011 = 0b11011011$

- "&": Der sog. bitweise Und Operator funktioniert ähnlich zu "|", mit einem kleinen Unterschied. Er sorgt dafür, dass im Ergebnis nur eine 1 steht, wenn in beiden Operanden am selben Bit eine 1 steht. Betrachte dazu folgendes Beispiel: $0b10101100 \ \& \ 0b1101001 = 0b10001000$. Dieser Operator ist also hervorragend dazu geeignet, bestimmte Zahlenteile zu filtern: $0b10101100 \ \& \ 0b00001111 = 0b00001100$. Hier wird nur die untere Hälfte übernommen.

3. Overflow, Carry und Borrow: Bei sämtlichen Rechenoperationen kann das Ergebnis zu groß (bzw. im negativen Bereich zu klein) werden. Bei der Addition kann das Ergebnis bspw. 1 Ziffer größer werden (Carry, im Deutschen auch Übertrag genannt), bei der Multiplikation kann sich die Länge des Ergebnisses aus der Summe der Längen der beiden Operanden ergeben (Overflow) und bei der Subtraktion kann es passieren, dass der 2. Operand größer als der 1. Operand ist, sodass ein Borrow benötigt wird. Ein Carry und auch ein Borrow sind quasi nichts anderes als ein Overflow mit dem Wert 1. Um diese Fehler aufzufangen, müssen wir uns darum kümmern, dass wir diesen Overflow/Carry/Borrow nicht verlieren, da sich sonst unser Gesamtergebnis ändert. Mehr dazu dann aber in der Aufgabe.
4. least- bzw. most-significant: In der Aufgabe wird immer wieder von den beiden Begriffen die Rede sein. Sie beschreiben bei einer Zahl die niedrigst- und höchstwertigste Stelle (least- und most-significant). Die niedrigstwertigste Stelle der Zahl 123 wäre die 3, während 1 die höchstewertigste Stelle ist.

Aufgabe

Unsere `BigUInt`s basieren auf dem `long`-Array `digits`. Dabei stellt jedes `long` einen Block aus 64 Bit unserer Zahl dar. Der least-significant Block steht am Index `0` und der most-significant Block am Index `digits.length - 1`. Übertragen auf das Dezimalsystem, wo beispielsweise 2 Ziffern einen einzelnen Block bilden, wäre die Zahl 12345 im Array also $[45, 23, 01]$. Diese umgedrehte Reihenfolge erleichtert uns später die Umsetzung der Rechenoperationen. Für sich ergeben die einzelnen Blöcke wenig Sinn und müssen zusammen als Ganzes interpretiert werden. Somit ist es möglich beliebig große Zahlen darzustellen. Der limitierende Faktor ist der Arbeitsspeicher.

Da die Division zu komplex ist, haben die 3 Entdeckuine das schon für dich übernommen. Deine Aufgabe ist es nun, die Methoden zum Addieren, Subtrahieren, Multiplizieren und Exponenzieren zu implementieren. Dazu aber erst einmal ein paar allgemeine Hinweise:

- Lass dich nicht durch die Größe des Templates verunsichern. Es ist bereits einiges implementiert, um dir die Arbeit zu erleichtern.
- Benutze keine Funktionen der `java.math`-Library!
- In der Klasse `Main` findest du eine `main()`-Methode, mit der du deine Implementierung testen kannst. Hier darfst du auch gerne Elemente der `java.math`-Library benutzen, da `Main` von Artemis nicht getestet wird.
- Unsere `BigUInt`s sind immutable. Das bedeutet, dass bei jeder Rechenoperation ein neuer `BigUInt` zurückgegeben wird.
- Es gibt 4 bereits implementierte Konstruktoren, die einen neuen `BigUInt` initialisieren. Der Konstruktor mit dem `String`-Parameter wird erst funktionieren, sobald du `add()` und `mul()` korrekt implementiert hast.
- Ebenso gibt es 3 bereits implementierte `toString()`-Methoden. Allerdings wird `toDecimalString()` erst funktionieren, wenn du `sub()` und `mul()` korrekt implementiert hast. Die anderen beiden kannst du von Anfang an zum Debuggen benutzen.
- Außerdem gibt es noch eine ganze Reihe von Hilfsmethoden. Bis auf `trim()` findest du sie alle ganz unten in der Klasse.
- Da wir in der ganzen Aufgabe ohne negative Zahlen arbeiten, werden die normalen Vergleichsoperatoren "<", "<=", ">" und ">=" nicht funktionieren. Stattdessen sollst du die Methode `Long.compareUnsigned` verwenden, welche 2 `long`s annimmt und eine Zahl größer `0` zurückgibt, falls der erste Parameter größer ist, eine Zahl kleiner `0` zurückgibt, falls der zweite Parameter größer ist, oder `0` zurückgibt, falls beide Zahlen gleich sind.
- Ganz oben stehen einige `BigUInt`-Konstanten, die dir bei den Methoden helfen könnten.
- Verändere keine der Methodensignaturen!
- Du kannst davon ausgehen, dass die Parameter stets nicht `null` sind und die `BigUInts` valide sind.
- Die folgenden Teilaufgaben sind eine Möglichkeit, wie man es umsetzen könnte. Solange am Ende das erwartete Ergebnis jeder Methode vorliegt, darfst du auch gerne Alternativen ausprobieren.

Addieren

Starten wir mit der `add()`-Methode. Diese besteht aus 2 Teilaufgaben.

1. ◉ **safeAdd** No results
   Da wir mit `long`s arbeiten, kann es passieren, dass es zu unbemerkten Overflows kommt, auch bei der Addition. Das Ergebnis der Addition ist maximal 1 Bit größer als die Länge des größeren Operanden, mindestens aber genauso groß. Bei der Addition von 2 `long`-Variablen kann also eine Zahl mit 65 Bit entstehen. Deswegen kümmern wir uns zuerst darum, dass wir 2 `long`-Variablen sicher addieren können. Der Trick besteht darin, jede Variable in 2 Hälften zu teilen, beide jeweils

32 Bit groß. Um die jeweilige Hälfte zu bekommen, kannst du die beiden Hilfsmethoden `low()` und `high()` benutzen.

- Nun können wir die unteren beiden Hälften addieren. Da beide Zahlen nur 32 Bit groß sind, passt das Ergebnis garantiert in 33 Bit und somit wieder in einen `long`. In der unteren Hälfte finden wir nun unser richtiges Ergebnis, in der oberen Hälfte den Übertrag, der bei der Addition eventuell entstanden ist.
- Nach demselben Prinzip addieren wir nun die oberen beiden Hälften.
- Jetzt müssen wir noch das Endergebnis richtig zusammenbauen. Der Übertrag der ersten Addition wird auf das Ergebnis der zweiten Addition addiert (auch hier könnten die beiden Hilfsmethoden nützlich sein, um den jeweiligen Zahlenteil zu erhalten). Der Übertrag der zweiten Addition wird zusammen mit dem Endergebnis zurückgegeben. Um die beiden Teilergebnisse richtig zusammenzubauen, muss das Ergebnis der zweiten Addition wieder zurück in die obere Hälfte geschoben werden (Stichwort Links-Shift). Als nächstes können beide Hälften durch Addition einfach verrechnet werden.
- Zum Schluss sollst du ein Array mit dem Gesamtergebnis und dem zweiten Übertrag zurückgeben. Das Ergebnis soll bei Index 0 sein, der Übertrag bei Index 1.

Letztendlich schaut es also so aus:
- $Result_{low} = a_{low} + b_{low}$
- $Result_{high} = a_{high} + b_{high} + carry_{low}$
- $Carry = carry_{high}$

$carry_{low}$ und $carry_{high}$ beschreiben hier die beiden Überträge, die bei der Addition der beiden Hälften (low und high) entstanden sind. Hier ist noch ein Beispiel mit 4-Bit-großen Zahlen in binär: $0111 + 1011$. Zuerst werden die beiden unteren Hälften addiert: $11 + 11 = 0110$, wobei wir hier zum Teilergebnis 10 ein Carry haben. Danach folgt die zweite Addition (inklusive dem Carry): $01 + 10 + 01 = 0100$, wobei wir hier ebenfalls zum Ergebnis 00 ein Carry haben. Nun bauen wir unser Array aus Ergebnis und Carry zusammen, also $[0010, 0001]$.

2. ◉ **add** No results
   Diese Hilfsmethode können wir nun dazu benutzen, um die beiden `long`-Arrays zu addieren. Du kannst davon ausgehen, dass die Länge des ersten Arrays größer-gleich der Länge des zweiten Arrays ist. Da wir hier nur addieren, ist die Länge des Ergebnis-Arrays maximal 1 größer als die Länge des größeren Eingabe-Arrays, mindestens aber gleich lang. Falls du dir schwer tust, überlege dir, wie du mit der Schulmethode auf Papier 2 Zahlen miteinander Ziffer für Ziffer addieren würdest. Eine Ziffer auf dem Papier entspricht dann einem Block unserer Arrays. Nun beginnen wir mit der Addition:

   - Gehe dafür beide Arrays von vorne nach hinten durch (least-significant zu most-significant) und addiere die beiden aktuellen Blöcke mit der zuvor implementierten Hilfsmethode.
   - Denk immer daran, dass du aus der Addition des vorherigen Blocks einen Übertrag bekommen könntest, der natürlich ebenfalls verrechnet werden muss.
   - Nachdem du das Ende des kürzeren Arrays erreicht hast, muss der Rest vom größeren Array nur noch mit dem Übertrag verrechnet werden.
   - Falls ganz zum Schluss immer noch ein Übertrag übrig bleibt, wird der in den Block ganz hinten im Ergebnis-Array geschrieben.

   Letztendlich schaut das Prinzip so aus:
   - $Result_i = a_i + b_i + carry_{i-1}$
     Hier bezieht sich das `i` auf den aktuellen Index im Array. Das Carry kommt somit aus der Rechnung des vorherigen Blocks, falls dieses zu groß geworden ist. Bei der allerersten Iteration (Index 0) gibt es natürlich noch kein Carry aus der vorherigen Rechnung, also ist $carry_{-1} = 0$.

Subtrahieren

Nun geht es weiter mit der Subtraktion. Diese ist sogar etwas einfacher als die Addition.

3. ◉ **sub** No results
   Bei der Subtraktion von 2 `long`-Arrays kannst du davon ausgehen, dass der Wert des ersten Parameters größer-gleich dem Wert des zweiten Parameters ist (die Fehlerbehandlung findet bereits in der Wrapper-Methode drüber statt). Das Ergebnis der Subtraktion ist maximal so groß wie die Länge des ersten Arrays. Falls du dir schwer tust, überlege dir, wie du mit der Schulmethode auf Papier 2 Zahlen miteinander Ziffer für Ziffer subtrahieren würdest. Eine Ziffer auf dem Papier entspricht dann einem Block unserer Arrays. Du brauchst dir keine Sorgen machen, falls einer oder beide Blöcke oder das Ergebnis "negativ" sind. Du kannst hier ganz normal Minus rechnen. Nur beim Vergleich musst du aufpassen, dass du wie oben erwähnt `Long.compareUnsigned()` verwendest anstatt "`<`", "`<=`", "`>`" oder "`>=`". Nun beginnen wir mit der Subtraktion:

   - Ähnlich wie bei der Addition iterieren wir von vorne nach hinten (least-significant zu most-significant) durch die beiden Arrays durch.

- Anstatt einem Übertrag haben wir jetzt einen sogenannten Borrow. Wenn der aktuelle Block des ersten Arrays kleiner als der aktuelle Block des zweiten Arrays ist, müssen wir beim nächsten Iterationsschritt zusätzlich 1 abziehen. Denk daran diesen eventuellen Borrow bei deiner Überprüfung zu berücksichtigen!
- Nachdem wir das Ende des zweiten Arrays erreicht haben, müssen wir nur noch von den verbleibenden Blöcken des ersten Arrays eventuell den Borrow weiterverrechnen.

Letztendlich schaut das Prinzip so aus:
- $Result_i = a_i - b_i - borrow_{i-1}$, wobei $borrow \in \{0, 1\}$
Hier bezieht sich das `i` auf den aktuellen Index im Array. Das Borrow kommt somit aus der Rechnung des vorherigen Blocks, falls dort der 1. Operand kleiner als der 2. Operand gewesen ist. Bei der allerersten Iteration (Index 0) gibt es natürlich noch kein Borrow aus der vorherigen Rechnung, also ist $borrow_{-1} = 0$.

Multiplizieren

Diese Teilaufgabe ist etwas komplizierter als die bisherigen. Starten wir mit der Hilfsmethode `safeMul()`.

4. ◉ **safeMul** No results
   Wie schon bei der Addition kann auch bei der Multiplikation ein Overflow enstehen. Damit wir diesen Overflow korrekt behandeln, kümmern wir uns in `safeMul()` darum, 2 `long`-Variablen sicher zu multiplizieren. Auch hier geben wir am Schluss ein Array zurück, in dem bei Index 0 das Gesamtergebnis und bei Index 1 der Overflow gespeichert werden soll. Wie bereits zuvor könnten dir die beiden Methoden `low()` und `high()` sowie Bitshifts und der bit-wise OR Operator ("`|`") helfen.

   - Die Multiplikation ist in 4 Multiplikationen aufgeteilt. Das Gesamtergebnis ist so groß wie 2 `long`s.
   - Wir müssen
     1. die unteren beiden Hälften miteinander multiplizieren,
     2. dann die untere Hälfte von `a` mit der oberen Hälfte von `b` multiplizieren,
     3. dann die obere Hälfte von `a` mit der unteren Hälfte von `b` multiplizieren
     4. und zum Schluss die beiden oberen Hälften miteinander multiplizieren.
   - Jedes einzelne Teilergebnis für sich passt garantiert in einen `long`.
   - Das Ergebnis von 1. kann so in das Gesamtergebnis übernommen werden.
   - Das Ergebnis von 4. kann so in den Overflow übernommen werden
   - Die untere Hälften von 2. und 3. müssen auf die obere Hälfte des Gesamtergebnisses addiert werden.
   - Die oberen Hälften von 2. und 3. müssen auf die untere Hälfte des Overflows addiert werden.
   - Denk dran, dass bei jeder Addition (außer der letzten) ein Übertrag entstehen kann.

   Letztendlich schaut es also so aus:
   - $res_0 = a_{low} \cdot b_{low}$
   - $res_1 = a_{low} \cdot b_{high}$
   - $res_2 = a_{high} \cdot b_{low}$
   - $res_3 = a_{high} \cdot b_{high}$
   - $Result_{low} = (res_0)_{low}$
   - $Result_{high} = (res_0)_{high} + (res_1)_{low} + (res_2)_{low}$
   - $Overflow_{low} = (res_1)_{high} + (res_2)_{high} + (res3)_{low} + carry_{Result_{high}}$
   - $Overflow_{high} = (res_3)_{high} + carry_{Overflow_{low}}$
   Um es übersichtlicher zu halten, wird jedes Teilergebnis (res_0 bis res_3) zwischengespeichert. $carry_{Overflow_{low}}$ und $carry_{Result_{high}}$ beziehen sich hierbei auf den Übertrag, der sich aus der Addition unmittelbar im vorherigen Schritt ergibt.

5. ◉ **mul** No results
   Nun können wir die Hilfsmethode dazu benutzen, 2 `long`-Arrays zu multiplizieren. Das Gesamtergebnis wird maximal die Länge des ersten Arrays plus die Länge des zweiten Arrays haben. Falls du dir schwer tust, überlege dir, wie du mit der Schulmethode auf Papier 2 Zahlen miteinander Ziffer für Ziffer multiplizieren würdest. Eine Ziffer auf dem Papier entspricht dann einem Block unserer Arrays. Das Prinzip funktioniert wie folgt:

   - Wir nehmen uns einen Block des ersten Arrays und multiplizieren den einmal auf alle Blöcke des zweiten Arrays drauf.
   - Das Ergebnis der Multiplikation von `a[i]` und `b[j]` wird auf das Gesamtergebnis an der Position `i + j` addiert. Denk daran, dass auch hier ein Übertrag entstehen kann, der später noch wichtig wird.
   - Ebenfalls auf das Gesamtergebnis an der Position `i + j` wird der Übertrag der vorherigen Iteration addiert.
   - Der neue Übertrag für die nächste Iteration berechnet sich als Addition aus dem Overflow der Multiplikation und den beiden Überträgen der beiden Additionen.

- Nachdem du am Ende des zweiten Arrays angekommen bist und noch ein Übertrag übrig bleibt, kannst du diesen an die nächste Position im Gesamtergebnis schreiben.
- Danach wird der Übertrag auf 0 gesetzt und wir machen mit dem nächsten Block des ersten Arrays weiter.

Letztendlich schaut das Prinzip so aus:
- $Result_{i+j} = Result_{i+j} + a_i \cdot b_j + carry_{i,j-1}$
$i + j$ beschreibt den aktuellen Index im Ergebnis-Array, welcher sich aus den Indices `i` und `j` der beiden Operanden ergibt. $carry_{i,j-1}$ enspricht also dem Übertrag aus dem vorherigen Iterationsschritt. Bei der allerersten Iteration jeder einzelnen Multiplikationsreihe (also `i` ist beliebig und $j = 0$) gibt es natürlich noch kein Carry aus der vorherigen Rechnung, also ist $carry_{i,-1} = 0,\ \forall i \in [0; b.length)$. Es wird außerdem niemals passieren, dass der Übertrag so groß wird, dass er nicht mehr in den carry passt.

Exponenzieren

6. ◉ **pow** No results
   Nun haben wir das schwierigste geschafft. Die Exponierung besteht dabei einfach nur aus mehreren Multiplikationen. Da ein `BigUInt` hier keinen Sinn als Exponent macht (schon ab `10.000` dauert es spürbar lange), wird nur ein `int` übergeben. Du kannst diesen `int` zum weiteren verrechnen in einen `BigUInt` umwandeln. Nun gibt es 2 Möglichkeiten:
   1. Die einfache Variante multipliziert einfach so oft die Basis (`this`) auf, wie der Wert des Exponenten ist.
   2. Die minimal schwierigere Variante, welche aber deutlich effizienter funktioniert, multipliziert hier gleich Quadrate der Basis. Solange der Exponent größer als 0 ist, soll folgendes passieren:
      - Falls der Exponent aktuell ungerade ist, soll die aktuelle Basis auf das Zwischenergebnis multipliziert werden.
      - In jedem Fall soll danach der Exponent halbiert werden und die aktuelle Basis durch das Quadrat der Basis ersetzt werden. Nutze hierfür eine lokale Variable, da wir unsere ursprüngliche Basis (`this`) nicht modifizieren wollen.
      - Das Verfahren funktioniert, da wir für jedes Halbieren des Exponenten das Quadrat der aktuellen Basis draufmultiplizieren würden. Das Beispiel a^4 kann zu $(a^2)^{4/2}$ umgeschrieben werden. Den Fall, dass der Exponent ungerade ist, haben wir im ersten Schritt schon behandelt. Betrachte dazu folgendes Beispiel: a^5 kann zu $a \cdot (a^2)^{(5-1)/2}$ umgeschrieben werden. Das $5 - 1$ im Exponenten brauchen wir nicht explizit rechnen, da dass durch die Ganzzahldivision übernommen wird.

Beispiel

Hier wird noch ein kleiner Test durchgeführt, in dem mehrere Operationen hintereinander ausgeführt werden.

◉ **Beispiel** No results

Exercise details

Release date	Nov 14, 2025 18:00
Submission due date	Nov 23, 2025 17:00
Complaint possible	No