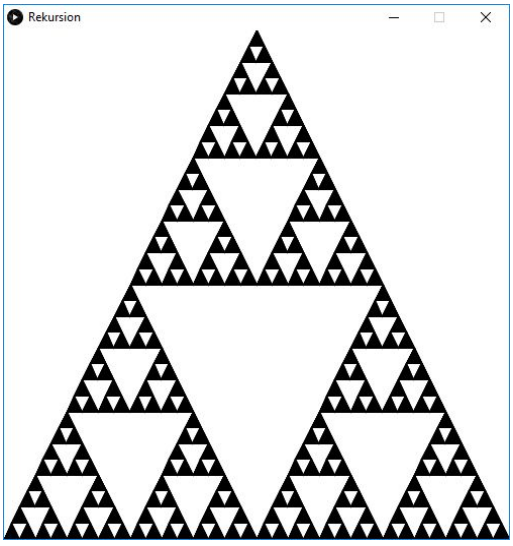
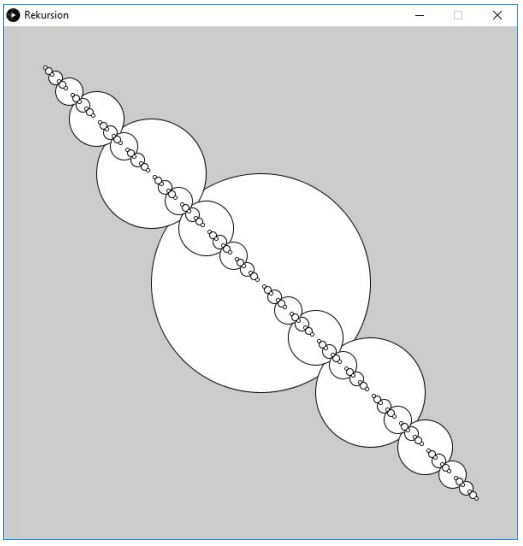
9. Rekursion

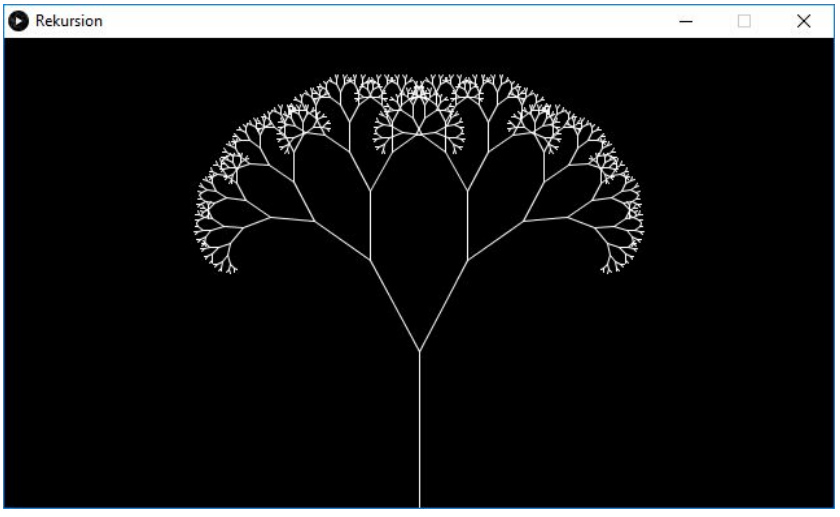
9.1 Motivation

Die bisherigen Kapitel decken ein solides Basiswissen und die Grundbausteine des Programmierens ab. Mit diesem Wissen können Sie bereits vielfältige Aufgaben und Problemstellungen mit dem Werkzeug der Programmierung meistern. Allerdings sind manche Probleme auf diese Weise noch umständlich zu lösen. Mit weiteren Konzepten, unterschiedlichen Herangehensweisen bzw. Denkweisen, können einige Aufgabenstellungen vereinfacht oder effizienter programmiert werden.

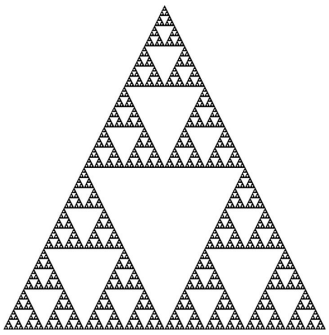
Ein solches, ganz bedeutendes Konzept ist die **Rekursion**. Sie kennen das Konzept vielleicht aus dem Grafischen bzw. Visuellen, wenn Sie die folgenden Abbildungen betrachten.



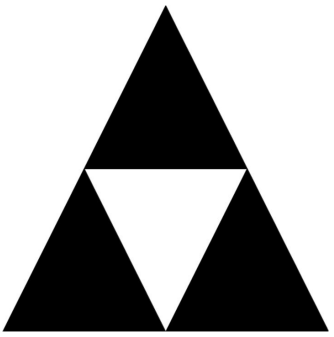




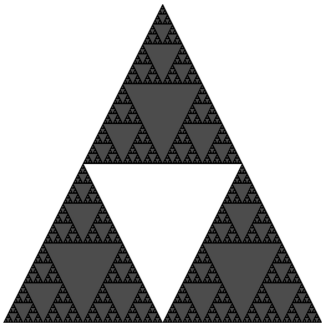
Ihnen allen ist gemeinsam, dass sie aus in sich selbst wiederholten Strukturen, Formen oder Mustern bestehen.
So besteht das Sierpinski Dreieck (links) etwa aus diesem einfachen Grundmuster (rechts):



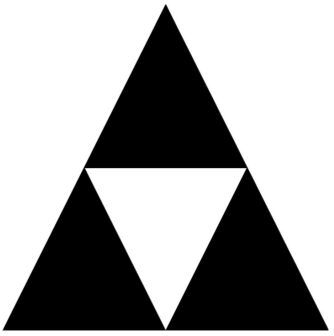
Sierpinski Dreieck



Grundmuster

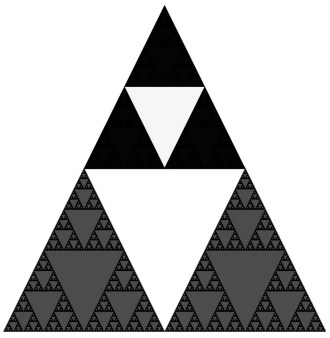


Visualisierung Grundmuster (halbtransparent)
auf Sierpinski Dreieck

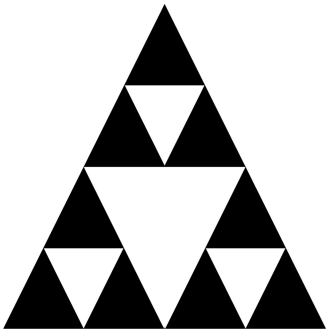


Resultat
(Tiefe 1)

Das Grundmuster wird in jedes der schwarzen Teildreiecke gezeichnet:

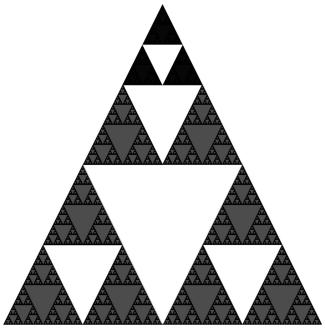


Visualisierung Grundmuster (halbtransparent)
auf Teildreieck oben

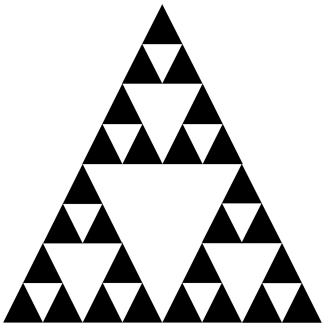


Resultat
(Tiefe 2)

Dies wird dann für jedes der neu entstehenden kleinen schwarzen Teildreiecke wiederholt.



Visualisierung Grundmuster (halbtransparent)
auf neu entstandenem Teildreieck oben



Resultat
(Tiefe 3)

Dies wird immer weiter, bis zu einem gewünschten Level bzw. theoretisch auch unendlich lange, wiederholt. Im Ausgangsbild des Beispiels wurde dies bis Level 7 durchgeführt.

Mittels Rekursion umgesetzt, können solche Grafiken in wenigen Zeilen programmiert werden.

9.2 Aufbau einer Rekursion

Dieser Aufbau aus "**in sich wiederholenden Mustern**" wird in der Programmierung umgesetzt durch eine Funktion, die sich selbst aufruft. Diese Funktion wird dementsprechend eine **rekursive Funktion** genannt. Grundidee der Rekursion ist es, ein Problem so in Teilprobleme zu teilen, dass die Teilprobleme wieder das gleiche Problem beschreiben. Die Teilprobleme sind dabei normalerweise etwas weniger komplex. Die Vorgehensweise der Rekursion könnte man dann auch wie folgt ausdrücken: "Löse die Teilprobleme mit derselben Methode wie das Gesamtproblem". Diese Denkweise ist anfangs gewöhnungsbedürftig und braucht Übung, kann aber in manchen Situationen sehr intuitiv sein.

Das folgende Beispiel soll illustrieren, dass Rekursionen natürlich nicht nur für grafische Visualisierungen eingesetzt werden, sondern auch in anderen Bereichen, wie etwa der Mathematik, Anwendung finden.

Beispiel: Es sollen die natürlichen Zahlen von 1 bis 5 aufaddiert werden. Für den Mathematiker ist die beste Lösung dazu die gaußsche Summenformel. Diese Formel führt am schnellsten zur Lösung. Wer diese aber nicht kennt, wird intuitiv das Problem vielleicht folgendermaßen lösen:

```
sum(5) = 5 + 4 + 3 + 2 + 1
```

`sum(5)` steht dabei für die Summe der natürlichen Zahlen von 1 bis 5. Allgemein würde dann mit `sum(n)` die Summe aller natürlichen Zahlen von 1 bis zur Zahl n gebildet werden.

Allerdings lässt sich die Lösung noch anders schreiben:

```
sum(5) = 5 + sum(4)
```

Das bedeutet, die Summe der natürlichen Zahlen von 1 bis 5 ist gleich 5 plus die Summe der natürlichen Zahlen von 1 bis 4. `sum(4)` steht dabei für die Summe der natürlichen Zahlen von 1 bis 4. Die Summe von 1 bis 4 kann erneut definiert werden als

```
sum(4) = 4 + sum(3)
```

Das kann wiederholt werden bis `sum(1)`, wo die Summe als "1" definiert ist.

```
sum(3) = 3 + sum(2)
sum(2) = 2 + sum(1)
sum(1) = 1
```

Diese Vorgehensweise ist rekursiv definiert. Denn das Problem wird so lange in immer kleinere gleiche Teilprobleme (hier, Summen von natürlichen Zahlen von 1 bis zu einer bestimmten Zahl) geteilt, bis es irgendwann (bei sum(1) = 1) lösbar ist.

Allgemein könnte man dies beschreiben mit:

`sum(n) = n + sum(n-1)` und `sum(1) = 1`

Die Funktion `sumUp()` im folgenden Code-Beispiel ist eine rekursive Funktion, welche genau dies in Processing umsetzt und die Zahlen von 1 bis n aufsummiert.
Achtung: Für Zahlen kleiner als 1 wird diese Funktion einen Fehler hervorrufen.

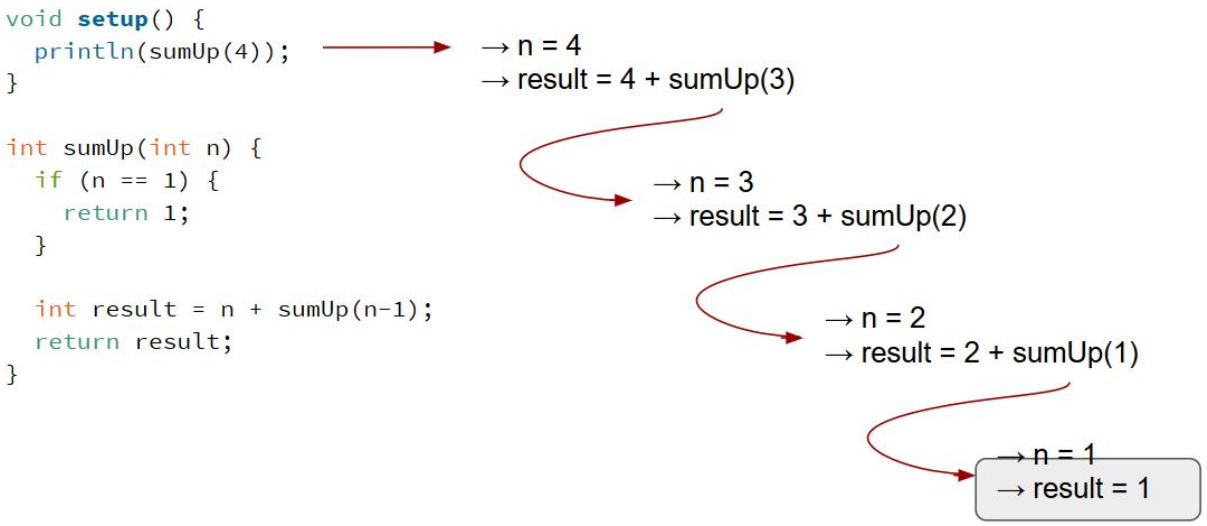
```
int sumUp(int n) {
  if (n == 1) {
    return 1;
  }

  int result = n + sumUp(n-1); //Rekursiver Funktionsaufruf sumUp(n-1)
  return result;
}

void setup() {
  println(sumUp(5));
}
```

Man beachte dabei, dass in der Funktion `sumUp()` die Funktion `sumUp()` wieder aufgerufen wird. An dieser Stelle passiert der sogenannte “rekursive Funktionsaufruf”. Die aufgerufene Funktion liefert entweder 1 zurück, wenn n bereits gleich 1 war, oder ruft sich selbst nochmals mit einem um 1 kleineren n auf. Die Anzahl dieser rekursiven Funktionsaufrufe, also wie oft die Funktion sich selbst aufruft bis endlich ein Wert (hier der Wert 1) zurückgegeben wird, nennt man auch **Rekursionstiefe**.

Die folgende Grafik visualisiert den Ablauf der Rekursion für `sumUp(4)` Schritt für Schritt im Detail. Analog funktioniert dies für alle anderen positiven ganzen Zahlen n in `sumUp(n)`.



Alternativ zu einer programmiertechnischen Erklärung, stellen Sie sich vor, es gibt vier Freunde, Alice, Bob, Carl und Dave.

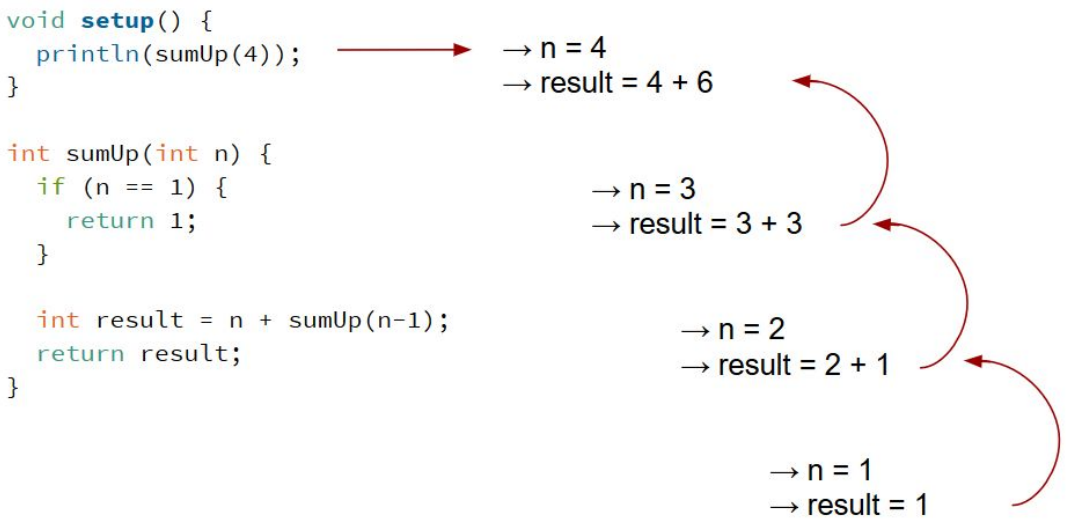
- Alice will die Summe von 1 bis 4 wissen. Sie weiß, dass die Summe `4 + sumUp(3)` ist, also 4 plus der Summe der natürlichen Zahlen von 1 bis 3. Allerdings weiß sie das Ergebnis von `sumUp(3)` nicht. Daher fragt sie Bob, ob er das für sie ausrechnen kann.
- Bob weiß, dass `sumUp(3)` gleich `3 + sumUp(2)` ist. Er weiß aber den Wert von `sumUp(2)` nicht. Daher fragt er Carl um Rat.
- Carl weiß, dass `sumUp(2)` gleich `2 + sumUp(1)` ist, aber was ist `sumUp(1)`? Carl fragt seinen Freund Dave.
- Dieser weiß, dass `sumUp(1)` gleich `1` ist.

Dies war der erste Teil des Ablaufs der Rekursion, die Zerlegung in Teilprobleme. Man kann sich das auch so vorstellen, dass man eben "in die Tiefe" arbeitet bis zum Erreichen der festgelegten Rekursionstiefe. Beim kleinsten Teilproblem (hier `sumUp(1) = 1`) angelangt, wird die Rekursion "zurückgerollt" bzw. muss man sich "zurück nach oben arbeiten", sodass schlussendlich das Ergebnis der Summe ausgegeben werden kann:

- Dave kann Carl sofort antworten und gibt ihm als Antwort "`1`".
- Damit kann Carl nun das Ergebnis von `sumUp(2) = 2 + sumup(1)` berechnen, nämlich mit `sumUp(2)` = `2 + 1`, also `3`. Dieses Ergebnis gibt er weiter an Bob.
- Jetzt weiß Bob, dass `sum(3)` gleich `3 + 3` ist, also `6`. Das erzählt er nun Alice.
- Alice kann sich endlich die Summe von 1 bis 4 ausrechnen, nämlich `4 + 6`. Damit erhält sie den Wert `10`.

An diesem Beispiel ist gut zu erkennen, dass das Problem nicht nur von einer Person gelöst wurde. Vielmehr konnten alle 4 Freunde eine Teillösung zum Problem beitragen und diese an jeweils einen weiteren Freund bzw. Freundin weiterreichen. Durch diese Vorgangsweise, das Problem immer weiter zu zerteilen und das verbleibende Teilproblem an den nächsten Freund bzw. an die nächste Freundin weiter zu reichen, haben sie gemeinsam schließlich das Problem als Ganzes lösen können.

Was passiert nun genau beim Programmablauf?



Wird die Funktion `sumUp()` für den Wert `4` aufgerufen, dann passiert in der Funktion Folgendes:

- `n` hat für den ersten Aufruf den Wert des übergebenen Parameters, nämlich `4`.
- Die if-Anweisung liefert `false`, da `4` nicht gleich `1` ist. Daher wird der `if`-Block übersprungen. Die Variable `result` erhält den Wert `4 + sumUp(3)`.
- `sumUp(3)` ist jedoch ein Funktionsaufruf, das bedeutet, die Funktion `sumUp()` wird für den Wert `3` noch einmal ausgeführt. Währenddessen pausiert die Funktion für den Wert `4`. Sie "wartet" auf den Ergebniswert von `sumUp(3)` (um das Ergebnis berechnen und zurückgeben zu können).
- Im Aufruf von `sumUp(3)` hat `n` nun den Wert `3`. Da `3` nicht `1` ist, erhält die Variable `result` diesmal den Wert "`3 + sumUp(2)`".
- In `sumUp(2)` wird dann schließlich noch die Funktion `sumUp()` für den Parameterwert `1` aufgerufen.
- Für `sumUp(1)` gilt allerdings, dass `n == 1` ist. Daher wird in der if-Anweisung `1` zurückgegeben und dieser Funktionsaufruf beim ersten `return` beendet.
- Das wird nun mit `2` zusammen addiert, das ergibt `3`. Das Resultat wird wiederum an den Aufrufer zurückgegeben.
- Damit erhält auch der Aufruf zu `sumUp(3)` sein Ergebnis für `sumUp(2)` und kann sich für `result` den Wert `6` ausrechnen.
- Dieser wird weiter an den Aufrufer in `sumUp(4)` zurückgegeben. Als `result` wird `4 + 6`, also `10`, ausgerechnet und dieses Ergebnis wird dann zurückgeliefert an den Funktionsaufruf in setup.
- Damit erhält man für den Funktionsaufruf `sumUp(4)` den Wert `10`, welcher schließlich ausgegeben wird.

Auch der Computer arbeitet in dieser ähnlichen Form. Anstatt vieler Freunde hat er jedoch einen Speicher, den sogenannten **Stack**, wo er sich die Zwischenwerte abspeichert. Während die vier Freunde jeweils für sich ein paar Werte merken mussten, wie z.B. welche Summe sie ausrechnen müssen und wen sie um Hilfe gebeten haben, so speichert sich auch der Computer bei jedem Funktionsaufruf separat alle benötigten Variablen, sowie welche Funktionen von wo aus aufgerufen wurden, um später an dieser Stelle fortfahren zu können. Im Computer bedeutet das, dass beispielsweise die Variable `result` mehrmals existiert, aber ganz unabhängig voneinander, da sie bei jedem Funktionsaufruf neu angelegt wird (siehe auch Sichtbarkeit von Variablen bei Funktionen).

9.3 Abbruchbedingung - Endlose Rekursion

Wie mit Schleifen, können auch mit Rekursionen Wiederholungen ausgedrückt werden. Daher ist es wichtig, dass auch Rekursionen eine Abbruchbedingung haben, da sie sonst zu **Endlosrekursionen** ausarten können. Die Funktion `sumUp()` hat als Abbruchbedingung die Abfrage `n == 1` und stellt bei jedem rekursiven Funktionsaufruf sicher, dass `n` immer kleiner wird.

Die Abbruchbedingung einer Rekursion ist meistens jene, die das kleinste Teilproblem löst. Das Problem wird also so oft verkleinert, bis die Lösung einfach genug ist. In `sumUp(1)` wissen wir, dass für die Summe von 1 bis 1 nur 1 rauskommen kann. Hier muss keine weitere Rekursion mehr stattfinden. Für 0 oder negative Zahlen wird hier kein Ergebnis definiert. Der Parameter n wird also bei jedem rekursiven Aufruf um 1 kleiner, bis endlich n == 1 ist und das Ergebnis der Summe bis 1 gleich 1 ist.

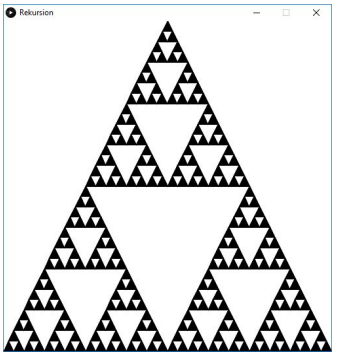
Falls eine Abbruchbedingung falsch gewählt ist, dann äußert sie sich auf zwei mögliche Arten. Zum einen kann ein falsches Ergebnis berechnet werden oder die Rekursion endlos laufen, es tut sich also für eine sehr lange Zeit einfach nichts. Bei einer solchen Endlosrekursion kann es zu einem Programmabbruch mit einer sogenannten **StackOverflowException** kommen. Das bedeutet, dass der Stack-Speicher überfüllt ist und die Rekursion daher nicht weiter stattfinden kann. Hätte jedoch der Computer unendlich viel Speicher, dann würde die Rekursion unendlich lange ohne Fehlermeldung laufen.

9.4 Unterschied Schleifen und Rekursion

Da sowohl Rekursionen als auch Schleifen Wiederholungen formulieren, stellt sich die Frage, was genau der Unterschied zwischen diesen zwei Konzepten ist bzw. welches Konzept wofür eingesetzt wird.
Grundsätzlich sind beide Varianten austauschbar. Das heißt: Für jede Art von Wiederholung lässt sich das Problem sowohl als Rekursion als auch mittels Schleifen ausdrücken. Auf die Frage, welches der beiden Konzepte für eine bestimmte Problemstellung zu bevorzugen ist, weil es sie etwa eleganter, einfacher oder effizienter löst, lautet die einfachste Antwort: Wann immer Sie merken, dass sich etwas (z.B. ein grafisches oder mathematisches Muster o.ä.) in sich selbst wiederholt, ist dies ein guter Indikator für eine Rekursion.

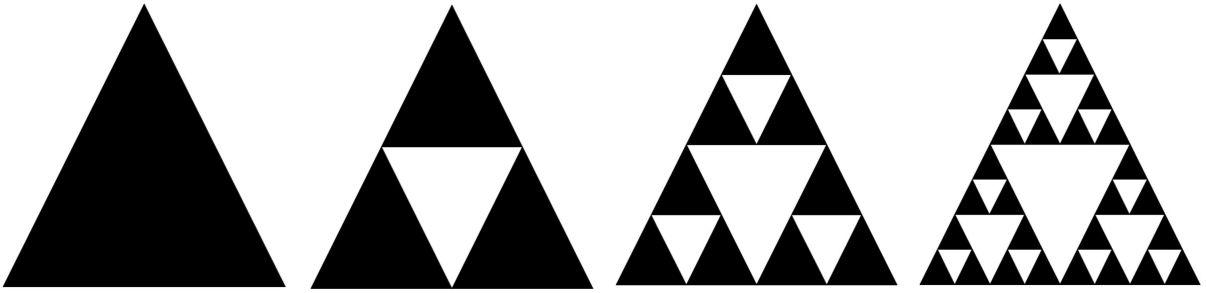
Die Herangehensweise von Schleifen und Rekursion unterscheidet sich grundlegend. Während beim rekursiven Ansatz die Problembeschreibung im Vordergrund steht, wird bei der iterativen (d.h. mit Schleifen) Variante versucht, das Problem Schritt für Schritt zu lösen.

Beispiel: Ein vereinfachter rekursiver Ansatz für das Sierpinski Dreieck, einem typischen grafischen Beispiel für Rekursionen, kann folgendermaßen zusammengefasst werden:



- Ein großes schwarzes Dreieck wird mit 3 Strecken in vier kleinere Dreiecke zerteilt.
- Das mittlere Dreieck wird “herausgeschnitten” (weiß).
- Für jedes der restlichen drei (schwarzen) Dreiecke wird dieser Vorgang wiederholt, bis der gewünschte Detailgrad erreicht ist.

Hier wird also dreimal der gleiche Vorgang wiederholt.

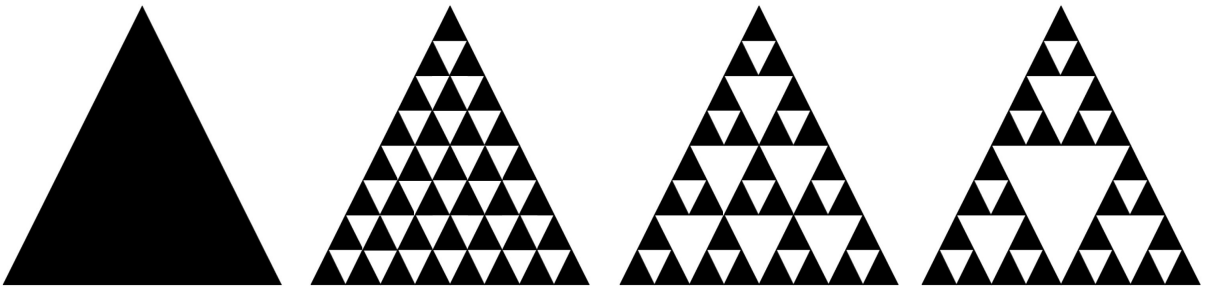


Ein iterativer Denkansatz mit Schleifen für das Sierpinski Dreieck könnte möglicherweise (vereinfacht) so aussehen:

- Beginnend bei der Spitze werden kleine weiße (gleichseitige, nach unten zeigende) Dreiecke derselben Größe in das schwarze Dreiecke gezeichnet. Bei jeder folgenden

Zeile wird ein weißes Dreieck mehr gezeichnet und die Dreiecke sind versetzt angeordnet.

- Dann wird das nächstgrößere Dreieck in gleicher Weise wiederholt über das vorhandene Bild gezeichnet.
- Das wird für alle Dreiecksgrößen wiederholt.



Vergleichen Sie die beiden Lösungsansätze und erklären Sie die Unterschiede noch einmal in eigenen Worten. Überlegen Sie sich auch für beide Varianten, welche Auswirkungen es auf den Code haben könnte, wenn Sie das Sierpinski Dreieck in weitere kleinere Dreiecke aufteilen wollen.