Aufgabe 1. Implementiere die Signumsfunktion `sgn(x)`, den Absolutbetrag `betrag(x)`, `cosinus(x)`, die Wurzelfunktion `wurzel(x)`, den Primzahltest `primtest(n)` und den Euklidischen Algorithmus `ggt(a,b)` (von Zettel 2) als Funktionen.

Aufgabe 2. Das folgende Programm sollte die Fakultätsfunktion implementieren und $5! + 10!$ auf der Konsole ausgeben. Zufälligerweise haben wir 6 Fehler dabei gemacht. Schnapp sie dir alle!

```
1  /* Fakultaetstest
2   * (c) 2015 Clelia und Johannes */
3
4  #include <studio.h>
5
6  int fakultaet (n) {
7     int ergebnis = 0;     /* speichert die Fakultaet */
8
9     while (n > 0)         /* verkleinere n, bis es */
10       ergebnis *= n;     /* null ist und multi- */
11       n--;               /* pliziere mit ergebnis */
12    return ergebnis;
13 }
14
15 int main () {
16          int add2fak;
17
18          add2fak = fakultaet (5) + fakulataet (10);
19          printf ("5! + 10! = %i\n", add2fak);
20          return 0,
21 }
```

Aufgabe 3. a) Implementiere für $x \in \mathbb{R}$ und $n \in \mathbb{N}$ eine Potenzfunktion $x^n =$ `power(x, n)` mit der Double-and-Add-Methode:

$$\text{power}(x,n) = \begin{cases} 1 & \text{wenn } n = 0 \\ x \cdot \text{power}\left(x^2, \frac{n-1}{2}\right) & \text{wenn } n \text{ ungerade} \\ \text{power}\left(x^2, \frac{n}{2}\right) & \text{wenn } n \text{ gerade} \end{cases}$$

zuerst mal rekursiv.

b) Implementiere eine Potenzfunktion `naiv_power(x, n)`, indem du eine Schleife von 1 bis n laufen lässt und bei jedem Durchlauf eine mit 1 initialisierte Variable mit x multipliziert. Berechne $0,9999999999^{2000000000}$ einmal mit `power(x, n)` von oben und einmal mit `naiv_power(x, n)` (es sollte ca. $0,818731$ rauskommen).

c) * Implementiere die Double-and-Add-Methode mit einer Schleife, also ohne rekursiven Aufruf.

Aufgabe 4. Diese Aufgabe wird auf eine `power(x, y)`-Funktion führen, die für beliebige $x \in \mathbb{R}^+$ und $y \in \mathbb{R}$ den Wert von x^y berechnet.

- Implementiere die Exponential-Funktion `expo(x)`, die e^x mithilfe folgender Reihendarstellung berechnet:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

- Implementiere eine Logarithmus-Funktion `logarithm(x)`, die $\ln(x)$ mithilfe folgender Reihedarstellung berechnet:

$$\ln(x) = 2 \cdot \sum_{k=0}^{\infty} \left(\frac{x-1}{x+1}\right)^{2k+1} \frac{1}{2k+1}$$

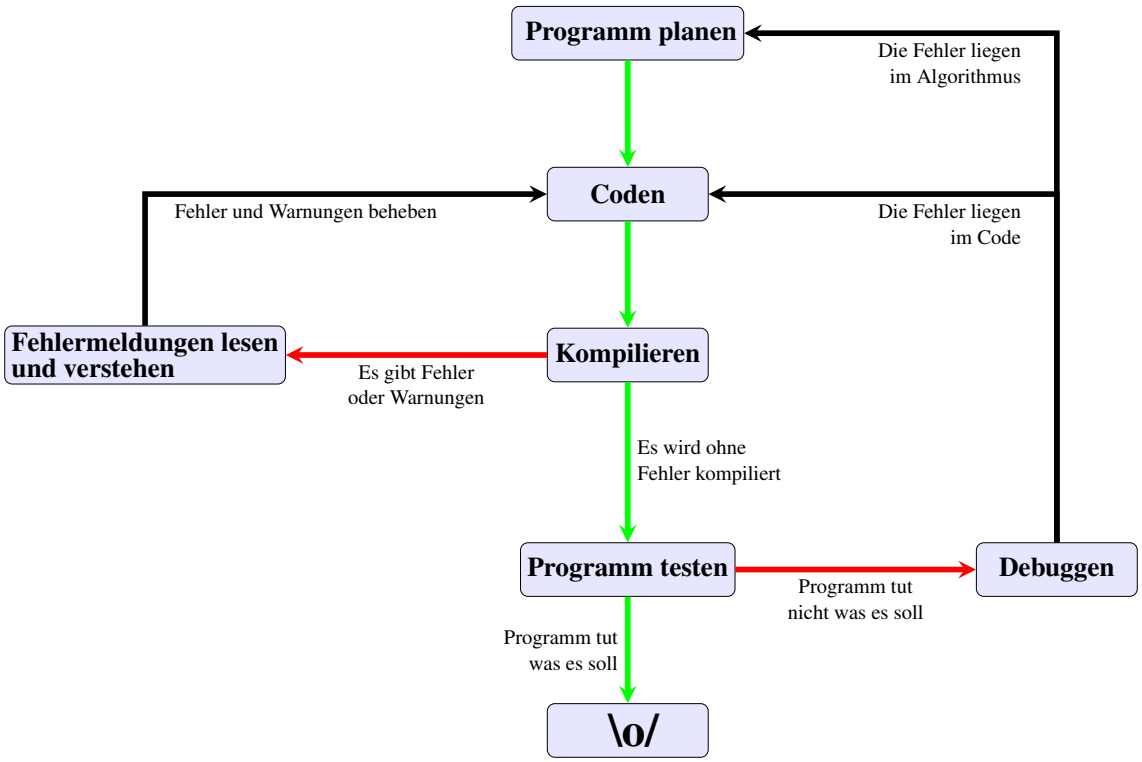
- Verwende die Formel

$$x^y = e^{y \cdot \ln(x)}$$

um `power(x, y)` zu bestimmen.

Der Programmierprozess

Wie bereits mehrfach in der Vorlesung erwähnt, ist das Schreiben des Programmcodes nur ein kleiner Teil des gesamten Programmierprozesses. Die meiste Arbeit wird bereits getan **bevor** ein Computer eingeschaltet wird. Eine sorgfältige Planung und Strukturierung des Programms macht es nicht nur einfacher zu coden, sondern verhindert auch konzeptionelle Fehler im Programm. Diese können natürlich trotzdem auftreten, was zu einem weiteren wichtigen Teil des Programmierens führt: das Debuggen.
Wie diese einzelnen Schritte zusammenhängen, haben wir für euch in folgendem Flowchart zusammengefasst. Dies gilt übrigens für so gut wie alle Programmiersprachen und nicht exklusiv für C ;).



Dabei ist bei den einzelnen Schritten folgendes zu beachten:

- **Programm planen**. Der wichtigste Schritt beim Programmieren. Wie oben schon erwähnt, ist es hier noch nicht notwendig, den Computer einzuschalten. Überlege dir genau, welche Aufgabe dein Programm lösen soll und zerlege diese gegebenenfalls in kleinere Teilaufgaben. Wie können diese gelöst werden? Was für Datenstrukturen wirst du brauchen? Kannst du eventuell bereits geschriebene Funktionen verwenden? Wie viel Speicher und wie viele Schleifen brauchst du höchstens? Und geht das auch schneller? Hierbei ist es besonders hilfreich, wenn du deine Ideen zu Papier bringst ;).

- **Coden**. In diesem Schritt setzt du deine Überlegungen aus Schritt 1 in Programmcode um. Wenn du dein Programm gut geplant hast, ist hier nicht mehr zu tun, als deine Ideen für deinen Computer zu übersetzen. Wie bei jeder Übersetzung in eine andere Sprache gilt jedoch: schlechten Code zu produzieren ist relativ leicht. Deshalb achte hierbei auf gute Lesbarkeit, ausreichende Kommentierung und eine übersichtliche Strukturierung. Das hilft nicht nur anderen, die sich dein Programm ansehen (zum Beispiel den Tutoren), sondern auch und insbesondere dir selbst (zum Beispiel beim Debuggen).

- **Kompilieren**. Mache aus deinem Code ein ausführbares Programm. An dieser Stelle gibt es zwei Möglichkeiten: entweder es wird ohne Probleme kompiliert, oder - viel wahrscheinlicher - der Compiler gibt dir Fehlermeldungen und Warnungen aus. In diesem Fall solltest du:

- **Fehlermeldungen lesen und verstehen**. Compilerfehlermeldungen werden von oben nach unten gelesen. Das bedeutet, dass du gegebenenfalls bis zur ersten Fehlermeldung hochscrollen musst. In der Fehlermeldung oder Warnung wird dir die Zeile, in der der Fehler auftritt, und die Art des Fehlers angegeben. Nachdem du die oberste Fehlermeldung oder Warnung behoben hast, solltest du erneut kompilieren, da es sein kann, dass die folgenden Meldungen ebenfalls auf den einen Fehler zurückzuführen sind. Arbeite auf diese Weise alle Fehler und Warnungen ab, bis dein Programm ohne Rückmeldung kompiliert.

- **Programm testen**. Ein fehlerfrei kompilierendes Programm muss noch lange nicht das tun, was du geplant hast. Um herauszufinden, ob es seine Aufgabe erfüllt, solltest du es ausgiebig testen. Probiere dazu verschiedene Einstellungen für die Parameter oder Inputdaten durch und versuche dabei auch, schwierige Spezialfälle zu finden und zu testen. Wie viel und wie lange du testen musst, hängt von der Art und Komplexität deines Programms ab. An dieser Stelle gibt es wieder zwei Möglichkeiten: entweder, das Programm löst die Aufgabe, für die es geschrieben wurde, oder es treten Fehler auf. Im zweiten Fall solltest du dann:

- **Debuggen**. Unter Debuggen versteht man die Suche und das Beheben von Fehlern im Programm, die der Compiler nicht findet. Es gibt verschiedene Möglichkeiten zu debuggen. In der Vorlesung gehen wir vor allem auf eine der einfachsten ein: das Debuggen mit Hilfe der `printf`-Funktion. Versuche zunächst, den einfachsten Fall zu finden, für den dein Programm nicht das tut, was es sollte. Dies könnte dafür sorgen, dass du die Fehler in deinem Programm deutlich schneller findest. Danach solltest du durch geschickte Ausgaben von Statusmeldungen den Fehler auf einen Teil des Programms bzw. eine Funktion eingrenzen. In diesem Teil bzw. dieser Funktion kannst du dich dann durch weitere Eingrenzung bzw. durch Ausgabe wichtiger Variablen dem Fehler weiter annähern. Wenn du Glück hast, liegen deine Fehler im Code und können so leicht aufgespürt werden. Falls nicht, so sind sie wahrscheinlich konzeptioneller Art und du solltest deinen Algorithmus überdenken bzw. neu planen.

- **\o/**: Wenn du alle Schritte vorher bis hierhin befolgt hast, hast du jetzt ein Programm, das ohne Fehlermeldungen und Warnungen kompiliert und genau das tut, was es soll. Das ist awesome. Sei stolz auf dich und genieße die freigewordenen Endorphine.