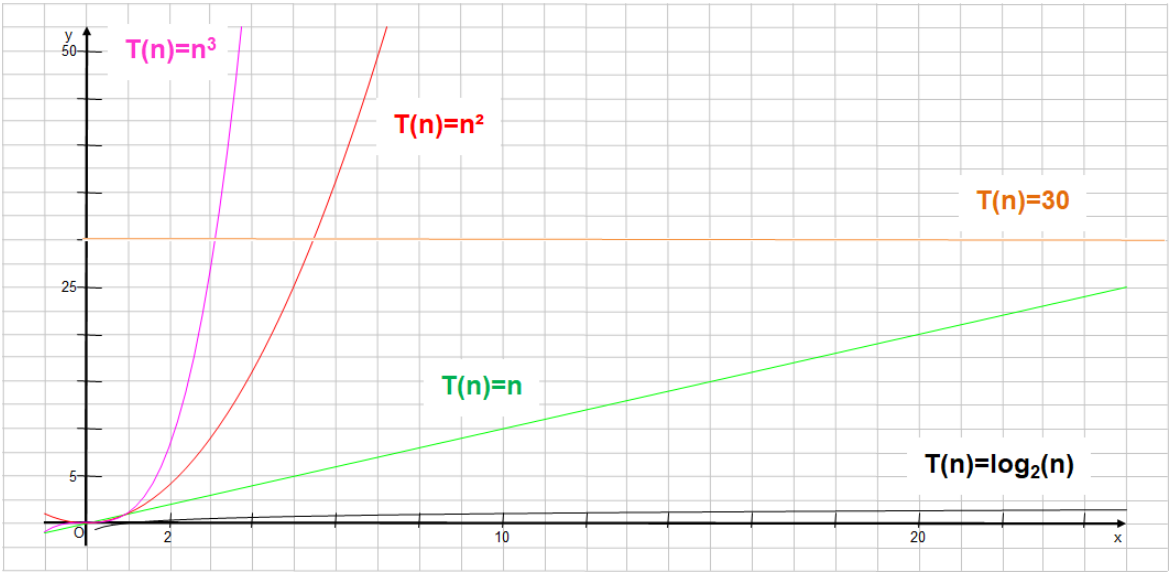
0.1 Die $\mathcal{O}$ - Notation

Grundsätzlich geht es bei der $\mathcal{O}-$ Notation darum, wie der Ressourcenbedarf unserer Algorithmus für große Eingaben skaliert (dabei kann sowohl die Laufzeit, als auch der Speicherplatzbedarf betrachtet werden). Explizit **nicht** betrachtet werden:

- Programmiersprache
- Betriebssystem
- Prozessorleistung
- Speicherausstattung
- Computerarchitektur
- etc.

Es geht rein um die Effizienz des Algorithmuses, nicht um seine konkrete Umsetzung auf einer bestimmten Hardware. Ist der Algorithmus zu „schlecht“, so ist er ggf. auf keiner Hardware praktisch umsetzbar!



Bei der Analyse eines Algorithmuses geht es in der Regel darum, eine **worst-case** oder **average-case**- Abschätzung zu machen. Wir beschränken uns auf den ersten Fall.
Grob gesprochen ist das Ziel, eine Funktion zu finden, die das Wachstum unserer Laufzeit nach oben begrenzen. Die Laufvariable der Funktion entspricht dabei in der Regel der Länge (oder Größe) der Eingabe.
Der Parameter n der Funktion oben könnte zum Beispiel für die Länge eines Strings stehen, oder für die Einträge eines Arrays. Wir wollen wissen, welche Art von Funktion als obere Grenze geeignet ist.
Natürlich sollte die Funktion dafür möglichst wenig „wachsen“. Am Besten ist demzufolge eine konstante Laufzeit (im Beispiel oben $T(n) = 30$), denn hier ist die Länge überhaupt nicht relevant!
Noch einmal als wichtiger Hinweis: es geht uns hier jetzt nicht um eine exakte Anzahl an Operationen, oder eine exakte Laufzeit, sondern nur ein **asymptotisches** Verhalten!
Einige grundlegende Abschätzungen:

- „einfache“ Operationen wie das Verrechnen von Zahlen, das Prüfen einer Bedingung, eine Zuweisung, etc. haben eine konstante Laufzeit.
- Ist eine Wiederholung im Spiel (z.B. for $i = 0$ to n), dann liefert uns dies eine lineare Laufzeit $T(n) = n$.
- Auch wenn die Wiederholung nur bis z.B. $\frac{n}{2}$ läuft spricht man von linearer Laufzeit, mathematisch: multiplikative Konstanten spielen keine Rolle, d.h. n ist eine genauso „gute“ Laufzeit wie $10000n$, da das Wachstum immer noch linear ist.

- Werden Wiederholungen ineinander geschachtelt, so erhält man höhere Polynome als Laufzeit, da beispielsweise für jeden einzelnen Listeneintrag die gesamte Liste wieder durchlaufen wird (das ergibt n^2), usw.
- Im Falle unseres Baumes kommt es häufig zu logarithmischen Laufzeiten (dem Besten Fall außerhalb der konstanten Laufzeit). Der Logarithmus ist bekanntlich die Umkehrung der Potenz, d.h.: halbiert sich die Problemgröße in jedem Schritt (z.B. durch Auswahl des linken oder rechten Teilbaums), so ergibt sich eine Laufzeit von $log_2(n)$.

Ein Beispiel:

```
int sum = 0;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n/2; i++) {
        summe += 1;
    }
}
for(int k = 0; k < n; k++) {
    sum -= 1;
}
```

Im ersten Block gibt es zwei ineinandergeschachtelte Wiederholungen, d.h. das inkrementieren wird $n \cdot \frac{n}{2} = \frac{1}{2}n^2$ mal ausgeführt. In der zweiten Wiederholung gibt es dagegen n Ausführungen. Damit insgesamt:

$$\frac{1}{2}n^2 + n$$

Operationen. Konstante multiplikative Faktoren spielen keine Rolle, ebensowenig wie das n - denn für $n \to \infty$ wächst das Quadrat viel stärker als der lineare Summand. Man würde deswegen für diesen Algorithmus folgern:

$$\mathcal{O}(n^2)$$

Der Algorithmus verhält sich für große n also näherungsweise quadratisch.
Weitere Details: siehe Studium!