**Lemma 1.**

$$\frac{\log_a(n)}{\log_b(n)} = log_a(b)$$

*Beweis.* Es gilt $a^{\log_a(n)} = n$ und $b^{\log_b(n)} = n$, also $a^{\log_a(n)} = b^{\log_b(n)}$. Daraus folgt durch beidseitiges Anwenden von $\log_a()$ direkt $\log_a(n) = \log_a(b^{\log_b(n)}) = \log_b(n) \cdot \log_a(b)$. Daraus folgt unmittelbar $\frac{\log_a(n)}{\log_b(n)} = log_a(b)$. □

# Aufgabe 1

a) Die Zeit $t(x)$ ist proportional zur Komplexität $f(x)$, $t(x) = k \cdot f(x)$.

i) $f(2n) = \mathrm{ld}\,2 + \mathrm{ld}\,n = f(n) + 1$
$t(n) = 4s \implies t(2n) = 5s$
$t(n) = 10s \implies t(2n) = 11s$
$t(n) = 100s \implies t(2n) = 101s$

ii) $f(2n) = 2n = 2f(n)$
$t(n) = 4s \implies t(2n) = 8s$
$t(n) = 10s \implies t(2n) = 20s$
$t(n) = 100s \implies t(2n) = 200s$

iii) Näherungsweise Lösung ergibt
$t(n) = 4s \implies t(2n) = 13.49s$
$t(n) = 10s \implies t(2n) = 29.13s$
$t(n) = 100s \implies t(2n) = 244.64s$

iv) $f(2n) = \frac{f(2n)}{f(n)} f(n) = \frac{(2n)^3}{n^3} f(n) = 8f(n)$

$t(n) = 4s \implies t(2n) = 32s$
$t(n) = 10s \implies t(2n) = 80s$
$t(n) = 100s \implies t(2n) = 800s$

v) $f(2n) = 2^{2n} = (2^n)^2 = f(n)^2$
$t(n) = 4s \implies t(2n) = 16s$
$t(n) = 10s \implies t(2n) = 100s$
$t(n) = 100s \implies t(2n) = 400s$

b) $1, \log\log n, \log n, n^\epsilon, n^c, n^{\log n}, c^n, c^{c^n}, n^n$

c) 1. $\exists c, n_0$ mit $n^a \leq c \cdot n^b \quad \forall n \geq n_0$ genau dann, wenn $a - b \leq 0$. Annahme: $a - b \leq 0$. Sei nun $c = n_0^{a-b}$. Da $n \geq n_0$ und $a - b \leq 0$ ist $n^{a-b} \leq n_0^{a-b} = c \implies n^a \leq c \cdot n^b$. Nun zeigen wir die Kontraposition der Rückrichtung:
Z.Z.: $a - b \geq 0 \implies \quad \forall c, n_0 : \exists n > n_0 : n^a \geq c \cdot n^b \iff n^{a-b} \geq c \overset{a-b>0}{\iff} n \geq \sqrt[a-b]{c}$. Nach dem archimedischen Axiom gibt es stets ein solches $n$.

2. i. Z.Z.: $\log_a(x) = O(\log_b(x))$.
   Das ist äquivalent zu: $\exists c, n_0$ mit $\log_a(n) \leq c \cdot \log_b(n) \qquad \forall n > n_0$.
   Umgeformt erhalten wir: $\exists c, n_0$ mit $\frac{\log_a(n)}{\log_b(n)} \leq c \qquad \forall n > n_0$.
   Nach Lemma 1 entspricht das: $\exists c, n_0$ mit $\log_a(b) \leq c \qquad \forall n > n_0$.
   Für $c \geq \log_a(b)$ ist diese Aussage offensichtlich wahr.
   ii. Z.Z.: $\log_a(x) = \Omega(\log_b(x))$.
   Das ist äquivalent zu: $\exists c, n_0$ mit $\log_a(n) \geq c \cdot \log_b(n) \qquad \forall n > n_0$.
   Umgeformt erhalten wir: $\exists c, n_0$ mit $\frac{\log_a(n)}{\log_b(n)} \geq c \qquad \forall n > n_0$.
   Nach Lemma 1 entspricht das: $\exists c, n_0$ mit $\log_a(b) \geq c \qquad \forall n > n_0$.
   Für $c \leq \log_a(b)$ ist diese Aussage offensichtlich wahr.
   Aus (i) und (ii) folgt die Behauptung.
3. i) Z.Z.: $0 \leq a \leq b \implies a^x = O(b^x)$. Das ist äquivalent zu $\exists c, n_0$ mit $a^n \leq c \cdot b^n \qquad \forall n > n_0$.
   Es gilt: $c > 1 \implies \sqrt[n]{c} \geq 1$. Für $a \leq b$ und $c \geq 1$ folgt unmittelbar $a \leq b = 1 \cdot b \leq \sqrt[n]{c} \cdot b$. Wählt man also $c > 1$, so erhält man durch Potenzierung mit $n$ direkt $\exists n_0$ mit $a^n \leq c \cdot b^n \qquad \forall n > n_0$.
   ii) Z.Z:: $a^x = O(b^x) \implies 0 \leq a \leq b$.
   Das ist äquivalent zu $\exists c, n_0$ mit $a^n \leq c \cdot b^n \qquad \forall n > n_0 \implies 0 \leq a \leq b$. Umformung ergibt: $\exists c, n_0$ mit $1 \leq c \cdot \left(\frac{a}{b}\right)^n \qquad \forall n > n_0 \implies 0 \leq a \leq b$. Wäre einer der Werte $a$, $b$ negativ und der andere positiv, so wäre $\frac{a}{b}$ negativ und damit $\left(\frac{a}{b}\right)^n$ für gerade $x$ positiv und für ungerade $x$ negativ. Unabhängig von $c$ gäbe es also stets $x$ mit $c \cdot \left(\frac{a}{b}\right)^n \leq 0 < 1$.
   Annahme: Sowohl $a$ als auch $b$ sind negativ und es gilt $\exists c, n_0$ mit $a^n \leq c \cdot b^n \qquad \forall n > n_0$.
   Für ein gerades $n$ gilt: $a^n \leq c \cdot b^n \iff (-|a|)^n \leq c \cdot (-|b|)^n \iff -|a|^n \leq c \cdot -|b|^n$.
   Für ungerades $n$ erhält man allerdings $a^n \leq c \cdot b^n \iff (-|a|)^n \leq c \cdot (-|b|)^n \iff -|a|^n \leq c \cdot -|b|^n \iff |a|^n \geq c \cdot |b|^n$. Das ist ein Widerspruch.
   Folglich ist $a \geq 0, b \geq 0$.
   Annahme: $a > b$.
   $\forall n > n_0$ gilt $a^n \leq c \cdot b^n \iff |a|^n \leq c \cdot |b|^n \iff |a|^n \leq c \cdot |b|^n$. Durch Ziehen der $n$-ten Wurzel erhalten wir $|a| \leq |\sqrt[n]{c}| \cdot |b| \iff a \leq |\sqrt[n]{c}| \cdot b$. Da $a > b$ muss es ein $d > 0$ geben, sodass $a = (1+d)b$. Ferner gilt: $\forall d > 0 : \exists n > n_0 : |\sqrt[n]{c}| < 1 + d$. Damit können wir folgern: $\exists n < n_0 : a \leq |\sqrt[n]{c}| \cdot b < (1 + d)b = a$. Das ist allerdings ein Widerspruch und folglich ist $a \leq b$. □

# Aufgabe 2

1. Für $a \geq b > 0$ gilt nach Definition $\text{ggT}(a, b) = \text{ggT}(a - b, b)$ □
2. Ausgehend von der Eingabe $a_0 = a$, $b_0 = b$, $a, b \in N_0$ , $a + b > 0$, erzeugt der Euklidische Algorithmus eine Folge von Paaren

   $$(a_i, b_i) \qquad i \in \mathbb{N}_0$$

   Aus der Konstruktion ist sofort offensichtlich, dass $a_i + b_i \geq a_{i+1} + b_{i+1}$. Wir zeigen, dass $(a_i > 0 \wedge b_i > 0) \implies (a_i + b_i > a_{i+2} + b_{i+2})$.

   *Beweis.* Fall 1: $a_i \geq b_i$. Dann ist $a_{i+1} = a_i - b_i$, $b_{i+1} = b_i$ und somit $a_{i+1} + b_{i+1} = a_i - b_i + b_i = a_i < a_i + b_i$. Mit $a_{i+1} + b_{i+1} \geq a_{i+2} + b_{i+2}$ folgt sofort $a_i + b_i > a_{i+2} + b_{i+2}$.

Fall 2: $a_i < b_i$. Dann ist $a_{i+1} = b_i$ und $b_{i+1} = a_i$. Da nun $a_{i+1} \geq b_{i+1}$ erhalten wir $a_{i+2} = a_{i+1} - b_{i+1}$, $b_{i+2} = a_{i+1}$ und somit $a_{i+2} + b_{i+2} = a_{i+1} - b_{i+1} + b_{i+1} = a_{i+1} = b_i < a_i + b_i$.

□

# Aufgabe 3

a) Für $n = 43$ und $k = 10$ benötigt das Programm 11.610 s. Für $n = 34$ und $k = 18$ gibt das Programm -2091005866 zurück.

Listing 1: binomial.cc

```
1  #include "fcpp.hh"
2
3  int binomial (int n, int k)
4  {
5      return cond(k == 0, 1,
6                  cond(n == k, 1,
7                  cond(k >= n, 0,
8                          binomial(n - 1, k - 1) + binomial(n - 1, k))));
9  }
10
11 int main (int argc, char** argv)
12 {
13     return print (binomial(readarg_int(argc, argv, 1),
14                    readarg_int(argc, argv, 2)));
15 }
```

b) Der Gesamtaufwand $A_{n,k}$ zur Auswertung von $B_{n,k}$ ist größer gleich einer Konstanten $c_1$ multipliziert mit der Zahl $Bb_{n,k}$ der Blätter im Berechnungsbaum:

$$A_{n,k} \geq c_1 \cdot Bb_{n,k}.$$

Dann ist $Bb_{n,n} = 1$ und $Bb_{n,k} = Bb_{n-1,k-1} + Bb_{n-1,k}$. Daraus kann man $Bb_{n,k} = B_{n,k}$ folgern. Daraus folgt $A_{n,k} = \Omega(n^k)$. Der Gesamtaufwand $A_{n,k}$ zur Auswertung von $B_{n,k}$ ist kleiner gleich einer Konstanten $c_2$ multipliziert mit der Zahl $Bk_{n,k}$ der Knoten im Berechnungsbaum:

$$A_{n,k} \leq c_2 \cdot Bk_{n,k}.$$

Dann ist $Bk_{n,n} = 1$ und $Bk_{n,k} = Bk_{n-1,k-1} + Bk_{n-1,k} + 1$. Daraus folgt $Bk_{n,k} = \binom{n}{k} + \sum_{i=0}^{n}\sum_{j=0}^{i}\binom{i}{j} = \binom{n}{k} + \sum_{i=0}^{n} 2^i = \binom{n}{k} + 2^{n+1} - 1$.
Es lässt sich aber leider mit diesem Ansatz keine eindeutige Komplexität bestimmen, da die Anzahl der Knoten bei der Erzeugung sehr viel größer ist als die Anzahl der Blätter.

c) Das Programm hat eine asymptotische Komplexität von $\Theta(n)$. Für höhere Werte ist das zweite Programm deutlich schneller. Ja, das liegt daran, dass die asymptotische Komplexität deutlich geringer ist. Die berechneten Ergebnisse unterscheiden sich stark, da bereits 13! zu groß für den Datentyp int ist und es daher zu overflow Fehlern kommt.

Listing 2: binomial_fast.cc

```
1  #include "fcpp.hh"
2
3  int factorial(int f)
4  {
5      return cond( f == 0, 1,
6          cond( f== 1, 1, factorial(f - 1) * f));
7  }
8
9  int binomial_fast(int n, int k)
10 {
11     return  ( factorial( n)) / (( factorial( k)) * ( factorial( n - k)));
12 }
13
14
15 int main (int argc, char** argv)
16 {
17     return print( binomial_fast( readarg_int( argc, argv, 1),
18                         readarg_int( argc, argv, 2)));
19 }
```

d) Ein effizienter Algorithmus hat den Aufwand $\Theta(n)$, wenn er die berechneten Binomialkoeffizienten abspeichert und im Verlauf der Berechnung wieder darauf zugreifen kann. Das ist natürlich um Längen effizienter als der Algorithmus in a). Im Algorithmus aus a) werden einfach sehr viele gleiche Berechnungen sehr oft ausgeführt. Das ist in diesem Algorithmus nicht der Fall. Jede Berechnung, die man durchführen muss, wird genau einmal durchgeführt. Das erklärt die sehr viel größere Effizienz.