Rapport til øving 1

Kjøretidens avhengighet av n er ulik for de to algoritmene.

```
How many times the algorithms can run per second and time per second:
Algorithm 1 -  time complexity O(n):
n       Rounds per sec      Time per round
10      28203366            35 ns
100     5317475             188 ns
1000    266884              3746 ns
10000   24657               40556 ns

Algorithm 2 - time complexity O(log n):
n       Rounds per sec      Time per round
10      25750655            38 ns
100     24524520            40 ns
1000    21636699            46 ns
10000   19222151            52 ns

Power in Java - time complexity O(1):
n       Rounds per sec      Time per round
10      22363071            44 ns
100     22296024            44 ns
1000    18861112            53 ns
10000   22446460            44 ns
```

Kompleksiteten i tid for den første algoritmen er $\theta(n)$. Når algoritmen regnes ut, reduseres den med 1 i hvert rekursjonstrinn helt til den blir 0. Dermed blir altså kompleksiteten $\theta(n)$.

Ved å benytte seg av asymptotisk analyse kan man se hva som skjer når datamengden går mot uendelig, altså når $n \to \infty$.

```java
private static double pow1(double x, int n) {
    if(n==0) return 1;
    return x*pow1(x, n: n-1);
}
```

Skjer n ganger

Den første algoritmen har ingen løkker, men et rekursjonstrinn hvor metoden kalles n ganger. Kjøretiden er altså en lineær avhengighet av n. Dette kan man tydelig se i tabellen hvor tiden per runde øker lineært når n blir større.

For den andre algoritmen derimot kan man bruke mastermetoden for å undersøke tidsforbruket som funksjon av n: $T(n) = aT\left(\frac{n}{b}\right) + cn^k$. Hvor a er antall rekursive kall i metoden, og er dermed 1 siden rekursjonen blir kalt en gang hver runde. N blir dividert med 2 for hvert rekursjonstrinn, og dermed er b=2. Den siste delen av formelen $cn^k = 1$ fordi k=0. Dette skyldes at det er ingen løkker i rekursjonstrinnet. Dette gir at tidsforbruket som funksjon av n er $T(n) = T\left(\frac{n}{2}\right) + 1$. Kompleksiteten er som følgende:

$Hvis\ b^k = a, har\ vi\ T(n) \in \theta(n^k * \log(n)$. Siden $b^k = 2^0 = 1 = a$ er $T(n) \in \theta(\log n)$.

For den første algoritmen øker tiden per runde samtidig som antall runder i sekundet minker etter hvert som eksponenten blir større. Den andre algoritmen er både raskere og mer presis enn den første, ettersom at rundene i sekundet og tiden per runde holder seg mer konstant. På den måten er den andre algoritmen mindre avhengig av n, og kan dermed regne ut større eksponenter raskere enn den første algoritmen. Java sin egen metode for eksponenter derimot er betydelig raskere og mer presis enn begge algoritmene. Kompleksiteten i tid er $\theta(1)$, og tiden er dermed uavhengig av n. Det betyr at for svært store eksponenter vil Java sin metode regne ut svaret raskere enn begge algoritmene.