3.2: Store O notation.

$f(x)$ er $O(g(x))$: $f(x)$ vokser ikke hurtigere end $g(x)$.
$f(x)$ er $\Omega(g(x))$: $f(x)$ vokser mindst lige så hurtigt som $g(x)$.
$f(x)$ er $\Theta(g(x))$: $f(x)$ er $O(g(x))$ og $f(x)$ er $\Omega(g(x))$.

Hvis $f_1(x)$ er $O(g_1(x))$ og $f_2(x)$ er $O(g_2(x))$ så gælder:

- $f_1(x) + f_2(x)$ er $O(g_1(x) + g_2(x))$
- $f_1(x)f_2(x)$ er $O(g_1(x)g_2(x))$

$(\log x)^k$ er $O(x)$, for enhver konstant k.

Hvis

$$f(n) \text{ er } O(n^k)$$

og

$$g(n) \text{ er } O(n^\ell),$$

hvor k og ℓ er konstanter,
så har vi at

$$f(g(n)) \text{ er } O(n^{k\ell}).$$

3.3: Kompleksitet af algoritme.

n : mål for størrelsen af input.

$f(n)$: den tid det højst tager for algoritmen at løse problemet med denne inputstørrelse.

$f(n)$ angives normalt ikke eksakt, men med hjælp af store O notation.

procedure $bubblesort(a_1, \ldots, a_n$: reelle tal med $n \geq 2)$
for $i := 1$ **to** $n - 1$
. **for** $j := 1$ **to** $n - i$
. **if** $a_j > a_{j+1}$ **then** ombyt a_j og a_{j+1}
{ $a_1, \ldots, a_n$ er nu i voksende rækkefølge}

En sorteringsalgoritme får n tal som input.
Algoritmen sorterer de n tal på grundlag af m sammenligninger af par af tal.
(m er funktion af n. Der arbejdes med worst case: antal sammenligninger er altså højst m.)

De m sammenligninger kan give 2^m mulige udfald.
For at algoritmen kan håndtere alle $n!$ mulige rækkefølger af tallene i inputtet er det nødvendigt at

$$2^m \geq n!.$$

Altså $m = \log 2^m \geq \log n!$.
Vi ved: $\log n!$ er $\Omega(n \log n)$.

Enhver sorteringsalgoritme vil derfor have kompleksitet $\Omega(n \log n)$.

M.a.o.: en algoritme kan ikke (i worst case) sortere n tal med mindre end $c \cdot n \log n$ sammenligninger, hvor c er en konstant.