Algoritmkomplexitet

Koffman & Wolfgang

- kapitel 2, avsnitt 2.4

Linjär komplexitet

Om tiden ökar i proportion med antalet indata *n*, har algoritmen linjär komplexitet:

```
public static int search(int[] x, int target) {
    for(int i=0; i < x.length; i++) {
        if (x[i]==target)
            return i;
    }
    return -1;  // target not found
}
```

O(*n*)
där *n* = x.length

Flera variabler

Komplexiteten kan bero på flera olika indata:

```
public static boolean areDifferent(int[] x, int[] y) {
    for(int i=0; i < x.length; i++) {
        if (search(y, x[i]) != -1)
            return false;
    }
    return true;
}
```

O($n \times m$)
där n = x.length
och m = y.length

Kvadratisk komplexitet

Tiden är proportionerlig mot kvadraten på antalet indata n:

```
public static boolean areUnique(int[] x) {
    for(int i=0; i < x.length; i++) {
        for(int j=0; j < x.length; j++) {
            if (i != j && x[i] == x[j])
            return false;
        }
    }
    return true;
}
```

O(n^2)
där n = x.length

Ordo-notationen

O($f(n)$) kan ses som en förkortning av ”proportionerligt mot”, eller ”i storleksordningen”, eller ”order of magnitude”

Ett enkelt sätt att avgöra komplexiteten är att räkna antalet nästade loopar

Om vi antar att loopen endast innehåller enkla satser:

- en enkel loop är O(n)
- två nästade loopar är O(n^2)
- tre nästade loopar är O(n^3)
- etcetera…

Logaritmisk komplexitet

Man måste också räkna på hur många gånger en loop gås igenom:

```
for(i=1; i < x.length; i *= 2) {
    // gör någonting med x[i]
}
```

Kroppen kommer att exekvera med:

- i = 1, 2, 4, 8, 16, ..., 2^k
- så länge som $2^k \leq n$ (där n = x.length)

Alltså:

- $2^k \leq n$
- $\log 2^k \leq \log n$
- $k \leq \log n$ (eftersom $\log 2^k = k \log 2 = k$)

Alltså kommer loopen att exekvera $\log n$ gånger:

- komplexiteten blir $O(\log n)$

Ordo, formell definition

Vi säger att $T(n) = O(f(n))$, omm:

- det finns två konstanter $n_0 > 0$ och $c > 0$,
- så att för alla $n > n_0$, $T(n) \leq c \cdot f(n)$

Eller:

- $\exists n_0 > 0.\ \exists c > 0.\ \forall n > n_0.\ T(n) \leq c \cdot f(n)$

Med andra ord:

- när n blir tillräckligt stort så blir tiden $T(n)$ aldrig större än $c \cdot f(n)$

Mer ordo

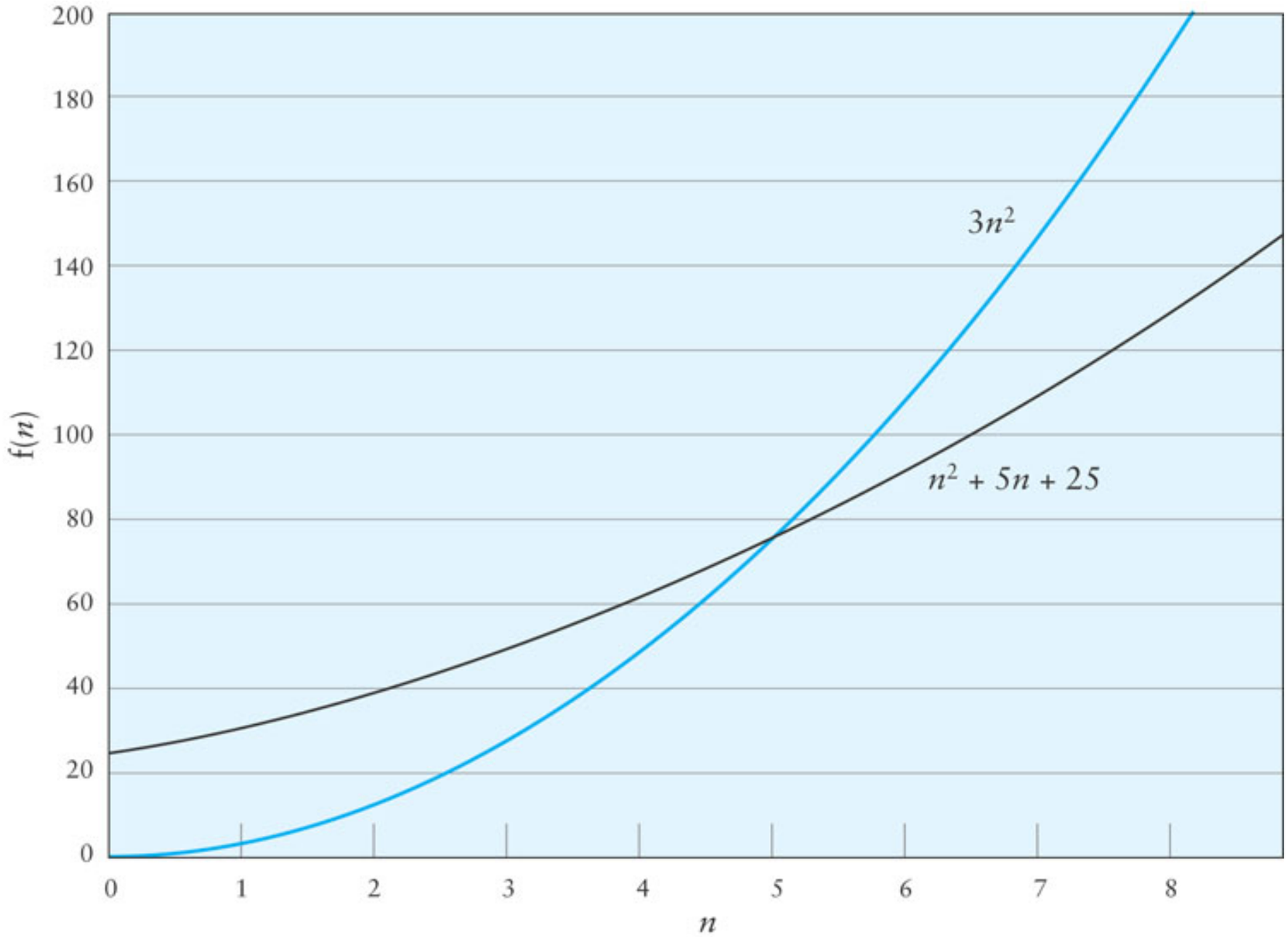
Tillväxthastigheten bestäms av den snabbast växande termen:

- $n^2 + 5n + 25 = O(n^2)$
- $0{,}0001 \cdot n^2 + 999999 \cdot n = O(n^2)$

I princip kan vi alltid ignorera alla konstanter och slänga bort termer av lägre ordning.

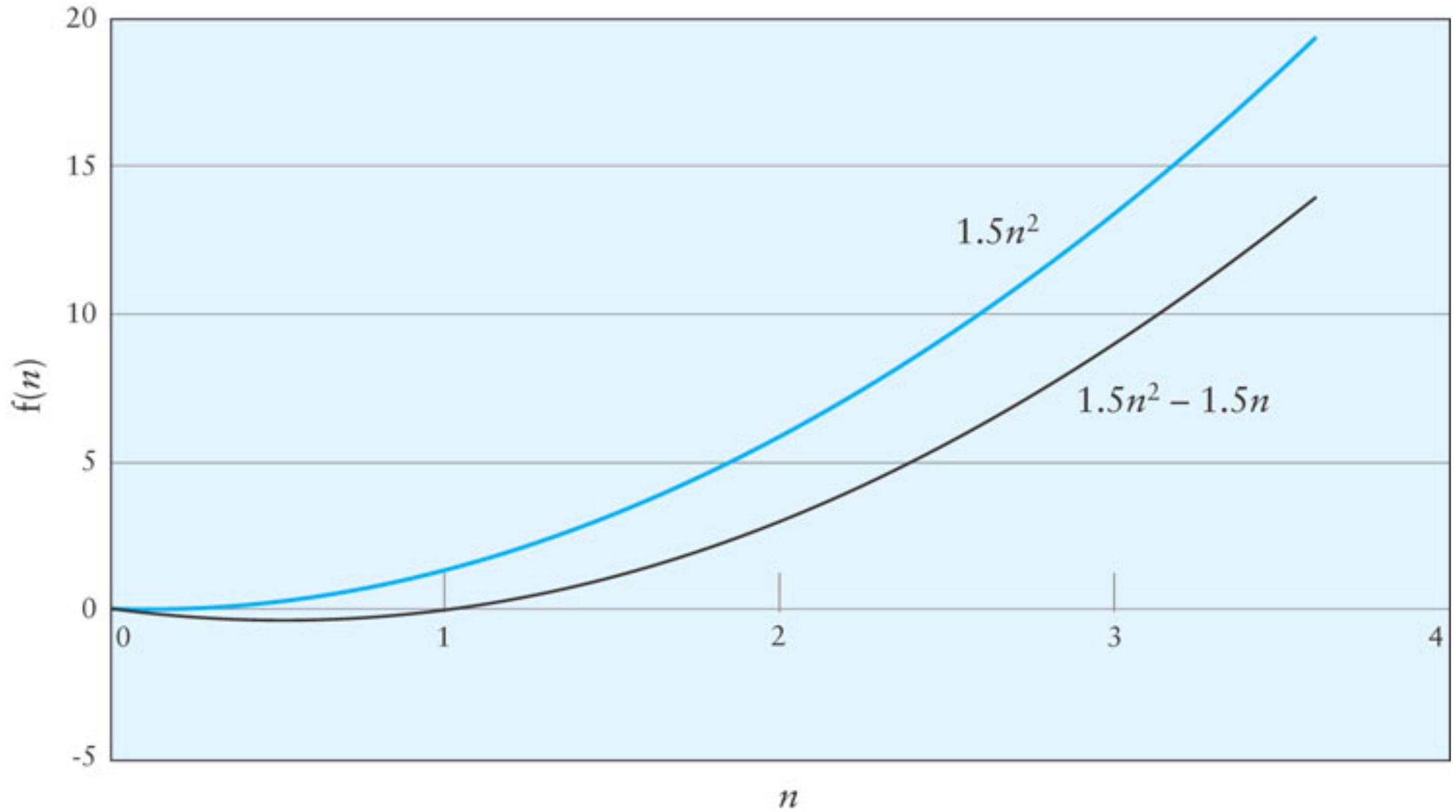
Exempel 2.8, sid 82

Om vi väljer $n_0 = 5$ och $c = 3$, så ser vi att $n^2 + 5n + 25 = O(n^2)$



Exempel 2.9, sid 83

Om vi väljer $n_0 = 1$ och $c = 1{,}5$, så ser vi att $1{,}5 \cdot n^2 - 1{,}5 \cdot n = O(n^2)$

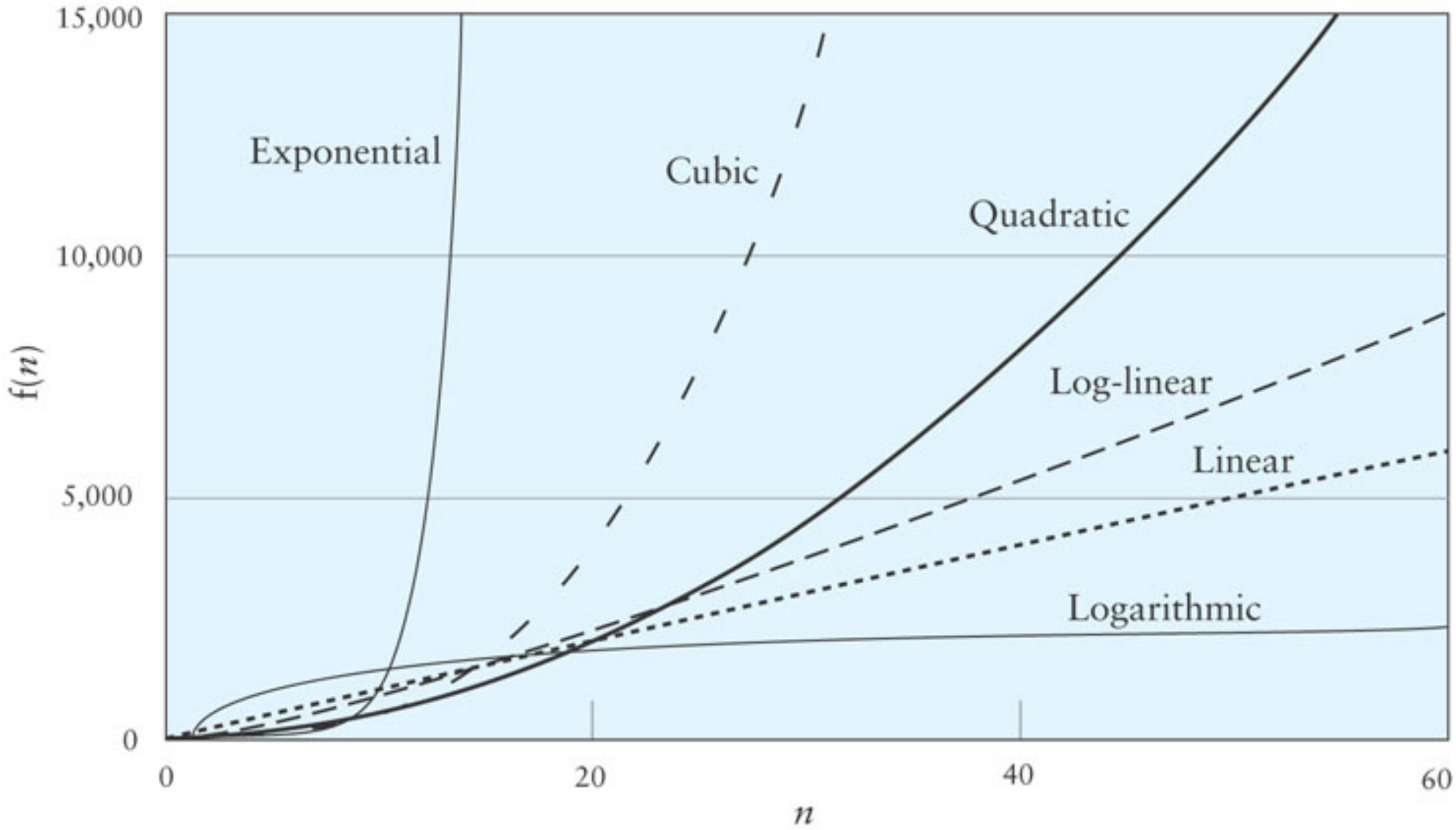


Tabell 2.3, sid 84

Vanliga tillväxthastigheter, snabbast överst:

Big-O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential
O($n!$)	Factorial

Figur 2.9, sid 85



Tabell 2.4, sid 86

Effekter av olika komplexiteter

O(f(*n*))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log *n*)	5.64	6.64	1.18
O(*n*)	50	100	2
O(*n* log *n*)	282	664	2.35
O(n^2)	2500	10,000	4
O(n^3)	12,500	100,000	8
O(2^n)	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
O(*n*!)	3.0×10^{64}	9.3×10^{157}	3.1×10^{93}

Räknelagar

Multiplikation

- $f \cdot \mathrm{O}(g) = \mathrm{O}(f) \cdot \mathrm{O}(g) = \mathrm{O}(f \cdot g)$
- $k \cdot \mathrm{O}(g) = \mathrm{O}(k \cdot g) = \mathrm{O}(g)$, om k är en konstant

Addition

- $\mathrm{O}(1) + \mathrm{O}(\log n) = \mathrm{O}(\log n)$
- $\mathrm{O}(\log n) + \mathrm{O}(n^k) = \mathrm{O}(n^k)$, om $k > 0$
- $\mathrm{O}(n^j) + \mathrm{O}(n^k) = \mathrm{O}(n^k)$, om $j \leq k$
- $\mathrm{O}(n^k) + \mathrm{O}(2^n) = \mathrm{O}(2^n)$

Exponentiella algoritmer

Algoritmer med exponentiell eller värre komplexitet har en praktisk övre gräns för storleken på indata

Om algoritmen är $O(2^n)$,
och 100 indata tar en timme, så:

- tar 101 indata 2 timmar
- tar 105 indata 32 timmar
- tar 114 indata 16 384 timmar (ca 2 år!)

Krypteringsalgoritmer

De bättre krypteringsalgoritmerna kan forceras i $O(2^n)$ tid, där n är antalet bitar i krypteringsnyckeln.

- en nyckellängd på 40 bitar kan forceras med en modern dator
- en nyckellängd på 100 bitar tar en miljard miljard (10^{18}) gånger längre tid...

Exempel, beräkna kvadratroten

Vilken komplexitet har följande program?

```
public static long squareRoot(long n) {
        long i = 0;
        long j = n+1;
        while (i + 1 != j) {
                long k = (i + j) / 2;
                if (k*k <= n)  i = k;
                else           j = k;
        }
        return i;
}
```

Exempel, beräkna kvadratroten

Vilken komplexitet har följande program?

```
public static long squareRoot(long n) {
        long i = 0;
        long j = n+1;
        while (i + 1 != j) {
                long k = (i + j) / 2;
                if (k*k <= n)  i = k;
                else           j = k;
        }
        return i;
}
```

Varje varv tar konstant tid

Exempel, beräkna kvadratroten

Vilken komplexitet har följande program?

```
public static long squareRoot(long n) {
        long i = 0;
        long j = n+1;
        while (i + 1 != j) {
                long k = (i + j) / 2;
                if (k*k <= n)  i = k;
                else           j = k;
        }
        return i;
}
```

Men hur många varv?

Varje varv tar konstant tid

Exempel, beräkna kvadratroten

Vilken komplexitet har följande program?

```
public static long squareRoot(long n) {
        long i = 0;
        long j = n+1;
        while (i + 1 != j) {
                long k = (i + j) / 2;
                if (k*k <= n)  i = k;
                else           j = k;
        }
        return i;
}
```

Men hur många varv?

Varje varv tar konstant tid

- Intervallet (j–i) halveras varje varv!

Exempel, beräkna kvadratroten

Vilken komplexitet har följande program?

```
public static long squareRoot(long n) {
        long i = 0;
        long j = n+1;
        while (i + 1 != j) {
                long k = (i + j) / 2;
                if (k*k <= n)  i = k;
                else           j = k;
        }
        return i;
}
```

Men hur många varv?

Varje varv tar konstant tid

- Intervallet (j–i) halveras varje varv!
- Alltså logaritmisk komplexitet, O(log n)