F04 - Pseudokod, komplexitetsanalys

5DV149 Datastrukturer och algoritmer
Kapitel 12

Niclas Börlin
niclas.borlin@cs.umu.se
Anna Jonsson
aj@cs.umu.se

2020-01-30 Thu

Vilken algoritm är bäst? (1)

- Vad är en algoritm?
  - Tills vidare: *En algoritm är noggrann metod för att utföra något stegvis.*

Vilken algoritm är bäst? (2)

- Vi behöver kunna jämföra algoritmer med varandra.
- Vad behöver vi?
  1. Ett sätt att beskriva algoritmer.
  2. Ett sätt att beskriva hur ”bra” en algoritm är.

Vilken algoritm är bäst? (3)

- Vad betyder ”bra”?
  - Algoritmen gör ”rätt”.
    - Korrekthet (annan kurs).
  - Algoritmen är ”snabb”.
    - Hur mycket **tid** behöver algoritmen?
  - Algoritmen går att köra på min dator.
    - Hur mycket **minne** behöver algoritmen?
- Storleken på problemet är centralt!
  - Sortera en lista på 10 element?
  - Sortera en lista på en miljon element?
  - Hur *skalar* algoritmen med större problemstorlekar?

Vilken algoritm är bäst? (4)

- Nyckelfråga:
  - Om ett problem med n element tar tiden x sekunder och y bytes minne, hur mycket resurser kräver ett problem med $2n$ element?
- Typfall

Linjärt (n)	$2n$	$\Rightarrow$	$2x$
Kvadratiskt (n^2)	$2n$	$\Rightarrow$	$2^2x = 4x$
Kubiskt (n^3)	$2n$	$\Rightarrow$	$2^3x = 8x$
Exponentiellt (k^n)	$n + 1$	$\Rightarrow$	kx
Logaritmiskt ($\log n$)	$2n$	$\Rightarrow$	$x + 1$

Exempel

- Antag:
    - 1 operation tar 1μs = 10^{-6}s.
    - $n = 10^9$ element i en lista.
- Två sorteringsalgoritmer:

1	$O(n^2))$	$T(n) = 10^{12}$ s	$\approx$	31000 år
2	$O(n \log n)$	$T(n) = 20000$ s	$\approx$	6 timmar

- Dubbelt så snabb dator: Algoritm 1 $\approx$ 15500 år.
- 1000 ggr så snabb dator: Algoritm 1 $\approx$ 32 år.
- **Snabbare algoritm viktigare än snabbare dator!**

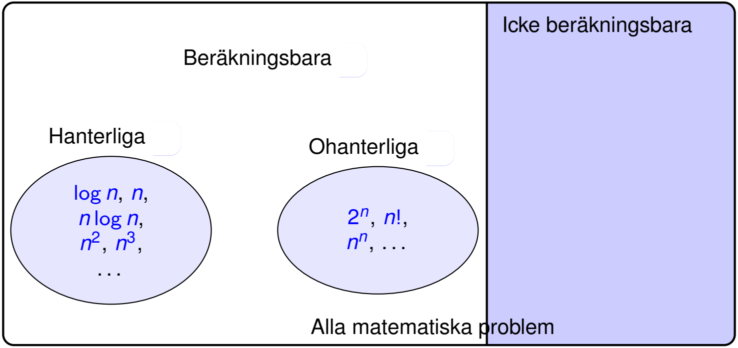
Exekveringstider — 100 000 MIPS, 10^{11} op/s

	n=10	20	50	100	300
$n \log n$	$2 \cdot 10^{-10}$ s	$6 \cdot 10^{-10}$ s	$2 \cdot 10^{-9}$ s	$5 \cdot 10^{-9}$ s	$2 \cdot 10^{-8}$ s
n^2	$1 \cdot 10^{-9}$ s	$4 \cdot 10^{-9}$ s	$2 \cdot 10^{-8}$ s	$1 \cdot 10^{-7}$ s	$9 \cdot 10^{-7}$ s
n^5	$1 \cdot 10^{-6}$ s	$3 \cdot 10^{-5}$ s	$3 \cdot 10^{-3}$ s	0.1 s	24 s
2^n	$1 \cdot 10^{-8}$ s	$1 \cdot 10^{-5}$ s	3 tim	$4 \cdot 10^{11}$ år	$6 \cdot 10^{71}$ år
n^n	0.1 s	$3 \cdot 10^{7}$ år	$3 \cdot 10^{66}$ år	$3 \cdot 10^{181}$ år	$4 \cdot 10^{724}$ år

Vad kan vi beräkna?

Beräkningsbara/hanterbara problem

- Icke beräkningsbara problem.
- Beräkningsbara, ohanterliga problem — superpolynom.
- Beräkningsbara, hanterliga problem — polynom.



Icke beräkningsbara

Beräkningsbara

Hanterliga

$\log n$, n, $n \log n$, n^2, n^3, ...

Ohanterliga

2^n, $n!$, n^n, ...

Alla matematiska problem

Ohanterbarhet

- Många triviala att förstå och viktiga att lösa:
    - Schemaläggning.
    - Handelsresande.

Hur hanterar vi ohanterbarhet?

- Heuristik!
- Lösa nästan rätt problem exakt:
  - Förenkling.
- Lösa exakt problem nästan rätt:
  - Approximation.
- Exempel: Hitta snabbaste vägen från A till B.
  - Förenkling: Sök A–motorväg–B.
  - Approximation: Dra “rakt streck” närmaste vägen A–B på kartan. Justera strecket så att det går på vägar.

NP-kompletta problem

- En speciell klass av ohanterliga problem.
- Exempel:
  - Givet en mängd {M} av heltal, finns det en icke-tom delmängd var summa är noll?
  - Generaliserad Sudoku ($n^2 \times n^2$ matris av $n \times n$-block).
- Ekvivalenta:
  - Transformeras till varandra.
  - Bevis för högst exponentiell komplexitet.
  - Saknar bevis för ohanterbarhet.
- Ett bevis att NP-kompletta problem är NP eller P (super-polynomiska eller polynomiska) är ett stort olöst problem inom datavetenskap och matematik.
- Ett av sju s.k. *Millennium Prize Problems*.

Pseudokod

Hur beskriver vi algoritmer?

- Vi behöver ett språk som:
  - Kan förstås av en människa.
  - Tillräckligt tydligt för att inte kunna missförstås.
- Tydlighet:
  - Strukturerat — ska kunna beskriva varje algoritm.
  - Formellt — ska vara självklart hur man översätter algoritmen till ett språk.
- Mindre formellt än programmeringsspråk:
  - Vi behöver t.ex. inte deklarera variabler.
- Språket liknar kod, men är det inte riktigt.
  - *Pseudokod*!

Pseudokod

- Finns ingen officiell standard.
- Alla döljer mycket av programspråkens designval.
  - Pseudokoden är *oberoende* av programspråk.
    - Ska kunna översättas till många olika språk.
- Vi använder:
  - En blandning av naturligt språk och programmeringsspråk.
  - Influenser från matematisk notation:

    $\leftarrow$ används för tilldelning.

    $=$ används för likhetsrelation.
  - Funktionsdeklarationer:
    - **Algorithm** name(*param1*, *param2*)

Pseudokod, programkonstruktioner

- Beslutsstrukturer:
  - `if ... then ... [else ...]`
- Villkorsloopar:
  - `while ... do ...`
  - `repeat ... until ...`
- Räkneloopar:
  - `for ... do ...`
- Anrop:
  - `algoritmName(` *args* `)`
- Arrayindexering:
  - `A[i]`
- Returnera värden:
  - `return` *value*

Pseudokod, Exempel

```
Algorithm arrayMean(v,n)
   input: An array v storing n integers
   output: The average of the n elements in v
sum ← v[0]
for i ← 1 to n-1 do
    sum ← sum + v[i]
return sum/n
```

Hur beskriver vi skalbarhet?

Stora Ordo (kap 12.2), definition

- Vi definierar $T(n)$ att vara $O(g(n))$ ("stort ordo av g av n") *om och endast om* det existerar konstanter $n_0, c > 0$ sådana att[1]

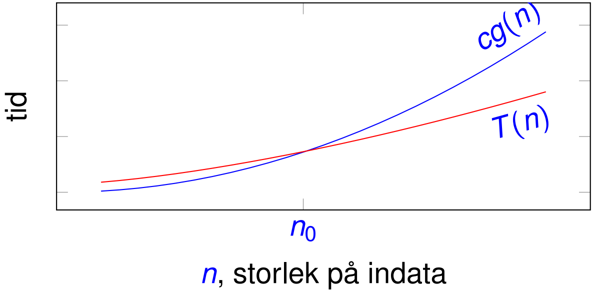
$$|T(n)| \leq cg(n), \ \forall n \geq n_0.$$

- Formellt: För $n \geq n_0$ så är $|T(n)|$ uppåt begränsad av $cg(n)$.
- Informellt: Över $n = n_0$ så växer $T(n)$ inte snabbare än $cg(n)$.
- Eng. stora ordo *Big-O*.

---

[1] Boken använder K och N i stället för c och n_0.

Stora Ordo (kap 12.2), illustration



Hur visa nåt $\forall n \geq n_0$?

- För att visa att

$$|T(n)| \leq cg(n), \ \forall n \geq n_0$$

räcker det med att visa att för

$$u(n) = cg(n) - |T(n)|$$

gäller att

$$\begin{aligned} u(n_0) &\geq 0, \\ u'(n) &\geq 0 \ \ \forall n \geq n_0. \end{aligned}$$

eller att för

$$v(n) = \frac{cg(n)}{|T(n)|}$$

gäller att

$$\begin{aligned} v(n_0) &\geq 1, \\ v'(n) &\geq 0 \ \ \forall n \geq n_0. \end{aligned}$$

$T(n)$ är $O(g(n))$

- Man kan säga
$$T(n) \in O(g(n)),$$
  då $O(g(n))$ är en mängd av funktioner.
- Vi kommer att säga/skriva att
$$T(n) \text{ är } O(g(n))$$
  eller
$$T(n) = O(g(n)).$$
- Eng. "$T(n)$ is (of) order $g(n)$."

Stora Ordo, exempel 1 (1)

- För $T(n) = 10n + 7$, är $T(n)$ $O(n)$?
- Beräkna c:

$$
\begin{aligned}
c &= \lim_{n\to\infty} \left(\frac{T(n)}{g(n)} \right) \\
&= \lim_{n\to\infty} \left(\frac{10n+7}{n} \right) \\
&= \lim_{n\to\infty} \left(10 + \frac{7}{n} \right) = 10.
\end{aligned}
$$

- Beräkna n_0:

$$
\begin{gathered}
T(n) \leq cg(n), \\
10n + 7 \leq 10n \Rightarrow \ldots?.
\end{gathered}
$$

- Gick inte: avrunda c uppåt!

Stora Ordo, exempel 1 (2)

- För $T(n) = 10n + 7$, är $T(n)$ $O(n)$?
- Beräkna c:

$$\begin{aligned} c &= \lim_{n\to\infty} \left(\frac{T(n)}{g(n)}\right) + 1 \\ &= \lim_{n\to\infty} \left(\frac{10n+7}{n}\right) + 1 \\ &= \lim_{n\to\infty} \left(10 + \frac{7}{n}\right) + 1 = 11. \end{aligned}$$

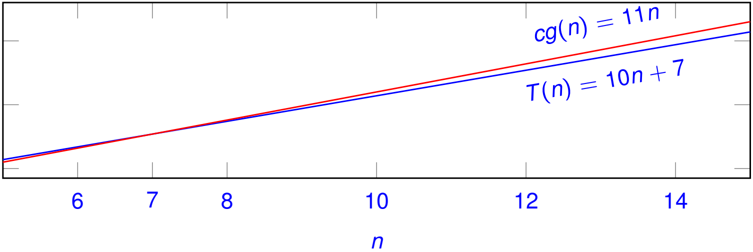
- Beräkna n_0:

$$\begin{gathered} T(n) \leq cg(n), \\ 10n + 7 \leq 11n \Rightarrow n_0 = 7. \end{gathered}$$

- Ja, $T(n)$ är $O(n)$ (med $c = 11, n_0 = 7$).

Stora Ordo, exempel 1 (3)

- $T(n) = 10n + 7$.
- $g(n) = n, c = 11, n_0 = 7$:
- $u(n) = cg(n) - T(n) = 11n - (10n + 7) = n - 7$.
- $u(n_0) = u(7) = 0$.
- $u'(n) = 1 \geq 0$.



Stora Ordo, exempel 2 (1)

- För $T(n) = 10n + 7$, är $T(n)$ $O(n^2)$?
- Beräkna c:

$$
\begin{aligned}
c &= \lim_{n\to\infty} \left(\frac{T(n)}{g(n)}\right) + 1 \\
&= \lim_{n\to\infty} \left(\frac{10n+7}{n^2}\right) + 1 \\
&= \lim_{n\to\infty} \left(\frac{10}{n} + \frac{1}{n^2}\right) + 1 = 1.
\end{aligned}
$$

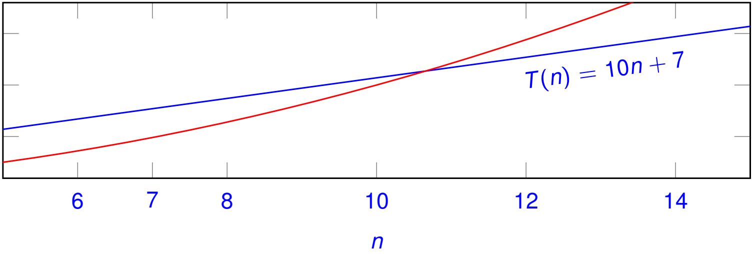
- Beräkna n_0:

$$
\begin{gathered}
T(n) \leq cg(n), \\
10n + 7 \leq n^2 \Rightarrow n_0 = 11.
\end{gathered}
$$

- Ja, $T(n)$ är $O(n^2)$ (med $c = 1, n_0 = 11$).

Stora Ordo, exempel 2 (2)

- $T(n) = 10n + 7$.
- $g(n) = n^2, c = 1, n_0 = 11$:



Vilket ordo ska man välja?

- Om $T(n)$ är $O(n)$, så är $T(n)$ också $O(n^2)$ (och $O(n^2 + n)$, $O(n^3)$, $O(2^n)$, $\ldots$).
- Underförstått att man väljer så “bra” begränsning som möjligt.
  - Viktigast för $g(n)$.
  - Vanligen mindre viktigt för c och n_0.
- Vilket c ska man välja?
  - Vid teoretisk analys av algoritmer vanligt med heltal.
  - Vid experimentell analys: ”avrunda rimligt uppåt”.
    - $c \approx 2.68 \cdot 10^{-6} \Rightarrow 3 \cdot 10^{-6}$ ok.
    - $c \approx 2.68 \cdot 10^{-6} \Rightarrow 10 \cdot 10^{-6} = 1 \cdot 10^{-5}$ troligen ok.
    - $c \approx 2.68 \cdot 10^{-6} \not\Rightarrow 1000 \cdot 10^{-6} = 1 \cdot 10^{-3}$ troligen inte ok.

Vilka parametrar är viktiga?

- När är $g(n)$ viktigast?
  - Nästan jämt!
- När är c viktigast?
  - För algoritmer med samma $g(n)$.
- När är n_0 viktigast?
  - För algoritmer med väldigt olika c.

Theta och Omega, definition

- Vi definierar $T(n)$ att vara $\Omega(g(n))$ ("omega av g av n") *om och endast om* det existerar konstanter $n_0, c > 0$ sådana att

$$|T(n)| \geq cg(n), \ \forall n \geq n_0.$$

- Formellt: För $n \geq n_0$ så är $|T(n)|$ nedåt begränsad av $cg(n)$.
- Informellt: Över $n = n_0$ så växer $T(n)$ inte långsammare än $cg(n)$.
- $T(n)$ är $\Theta(g(n))$ ("theta av g av n") *om och endast om* $T(n)$ är $O(n)$ och $\Omega(n)$.

Hur analyserar vi komplexitet?

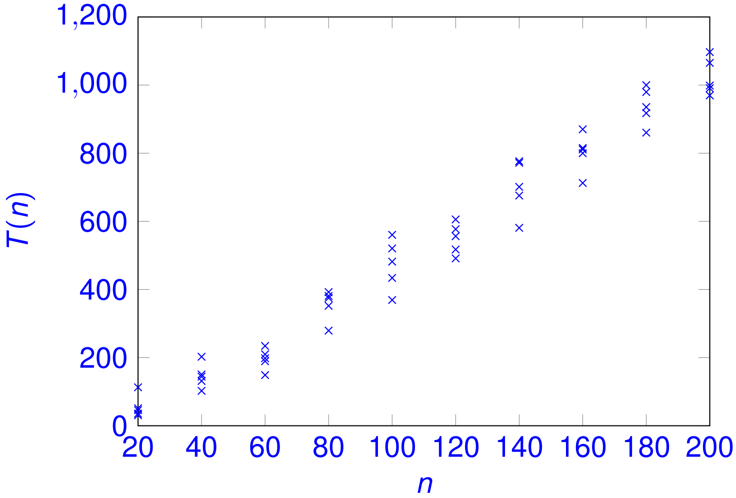
Komplexitetsanalys

- Experimentell
  - Kör programmet för olika problemstorlekar. Mät tiden.
  - Försök uppskatta trenden.
- Asymptotisk
  - Analysera algoritmen teoretiskt.
  - Undersök vad som händer då n blir stort.

Experimentell komplexitetsanalys (OU4!!!)

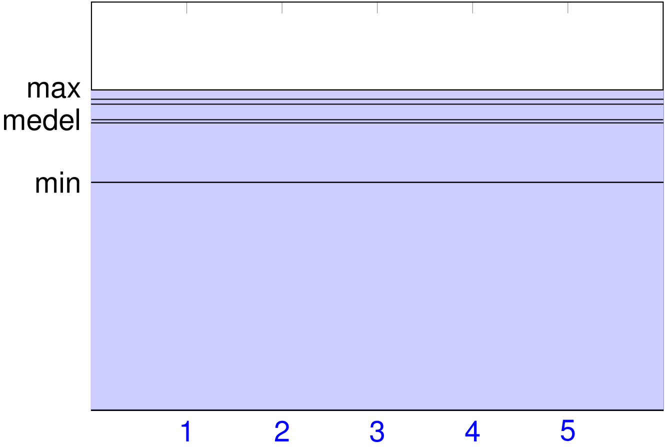
- Vid experimentell komplexitetsanalys tar vi tiden på ett givet program för olika problemstorlekar:
  1. Implementera algoritmen.
  2. Ta tiden $T(n)$ då programmet körs på olika problemstorlekar n.
  3. Ibland behövs också olika sammansättningar av data.
- Plotta $T(n)$.
  - Ansätt en hypotes, t.ex. $g(n) = n^2$.
  - Plotta $f(n) = T(n)/g(n)$.
  - Om $f(n)$ går mot positiv konstant så är hypotesen troligen korrekt.
  - Om inte, ansätt en annan hypotes, t.ex. $g(n) = n$.

Exempel

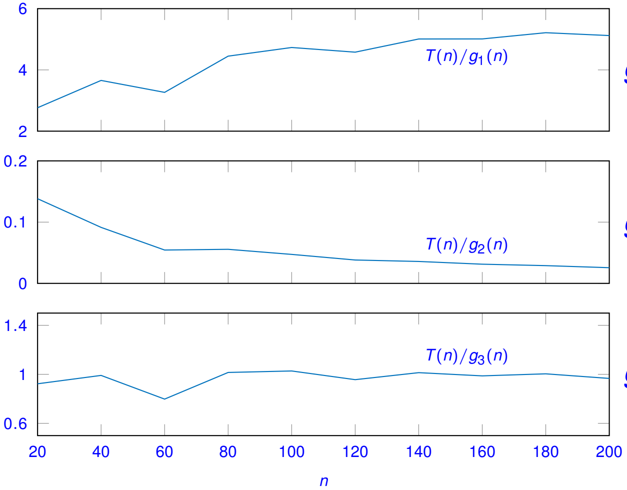

1,200
1,000
800
600
400
200
0
T(n)
20
40
60
80
100
120
140
160
180
200
n

Bästa, värsta, medel

- $T(n)$ för $n = 80$:



Testa hypoteserna



$g_1(n) = n$

$g_2(n) = n^2$

$g_3(n) = n \log n$

Experimentell analys, begränsningar

- Måste implementera och testa algoritmen.
- Svårt veta om programmet har stannat eller fast i beräkningarna.
- Experimenten kan endast utföras på en begränsad (liten) mängd data.
- Man kan missa viktiga testdata (specialfördelningar).
- Hård- och mjukvaran måste vara densamma för alla körningar.

Asymptotisk analys

- Högnivåbeskrivning av algoritmerna istället för implementation.
- Oberoende av hårdvaran och mjukvaran.
- Kan beräkna teoretiska bästa- och värsta-fallen.
- Utgå från pseudokoden.
    1. Räkna operationer:
    2. Ställ upp ett tidsuttryck $T(n)$ för antalet operationer beroende av problemstorleken n.
    3. Förenkla tidsuttrycket $T(n)$.
    4. Ta fram en funktion $g(n)$ och konstanter c, n_0 som uppfyller Ordo-definitionen.

Lite matematik behövs

- Logaritmer:

$$\begin{aligned}
\log_b xy &= \log_b x + \log_b y, \\
\log_b \frac{x}{y} &= \log_b x - \log_b y, \\
\log_b x^a &= a \log_b x, \\
\log_b a &= \frac{\log_x a}{\log_x b}.
\end{aligned}$$

- Exponenter:

$$\begin{aligned}
a^{b+c} &= a^b a^c, \\
a^{bc} &= (a^b)^c, \\
\frac{a^b}{a^c} &= a^{b-c}, \\
b &= a^{\log_a b}, \\
b^c &= a^{c \log_a b}.
\end{aligned}$$

Summor är bra att kunna…

- Generell definition

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \cdots + f(t).$$

- Geometrisk summa ($n \geq 0, 0 < a < 1$):

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}.$$

  - Konvergerar endast för $|a| < 1$.
- Aritmetisk summa: Summera alla tal från 1 till n:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n^2 + n}{2}.$$

Analys av algoritmer

- Primitiva operationer:
  - Lågnivå-beräkningar som är i stort sett oberoende av programspråk.
  - Kan definieras i termer av pseudokod.
  - Dessa operationer räknas som primitiva:
    - *Anropa* en metod/funktion.
    - *Returnera* från en metod/funktion.
    - *Utföra* en aritmetisk operation (+, -, . . .).
    - *Jämföra* två tal, etc.
    - *Referera till* (läs av eller tilldela) en variabel eller objekt.
    - *Indexera* i en array.
- Inspektera pseudokoden och räkna antalet primitiva operationer.

Analys av algoritmer (2)

- Kraftig abstraktion:
  - Vi bortser från hårdvaran (tid per operation).
  - Vi bortser från att olika operationer tar olika lång tid.
- Alternativet är att titta på de *verkliga* tiderna för de olika operationerna, ex. indexering långsamt eller $\sqrt{\cdot}$ långsamt.
  - Ger en maskin*beroende* analys.

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
   Sum all numbers 1..n (while version)

sum ← 0
i ← 1
while i <= n do
   sum ← sum + i
   i ← i + 1
return sum
```

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0
i ← 1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                        1
i ← 1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                                   1
i ← 1                                     1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                                   1
i ← 1                                     1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                         1
i ← 1                           1
while i <= n do                 $(n+1) \cdot 3$
    sum ← sum + i
    i ← i + 1
return sum
```

- Testet i lådan körs $n+1$ gånger.

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                         1
i ← 1                           1
while i <= n do                 $(n+1)\cdot 3 + n\cdot[\ ]$
    sum ← sum + i
    i ← i + 1
return sum
```

- Testet i lådan körs $n+1$ gånger.
- Loopen körs n gånger.

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                              1
i ← 1                                1
while i <= n do                      $(n+1) \cdot 3 + n \cdot [\ ]$
    sum ← sum + i                        4
    i ← i + 1
return sum
```

- Testet i lådan körs $n + 1$ gånger.
- Loopen körs n gånger.

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                              1
i ← 1                                1
while i <= n do                      $(n+1) \cdot 3 + n \cdot [\ ]$
    sum ← sum + i                        4
    i ← i + 1                            3
return sum
```

- Testet i lådan körs $n + 1$ gånger.
- Loopen körs n gånger.

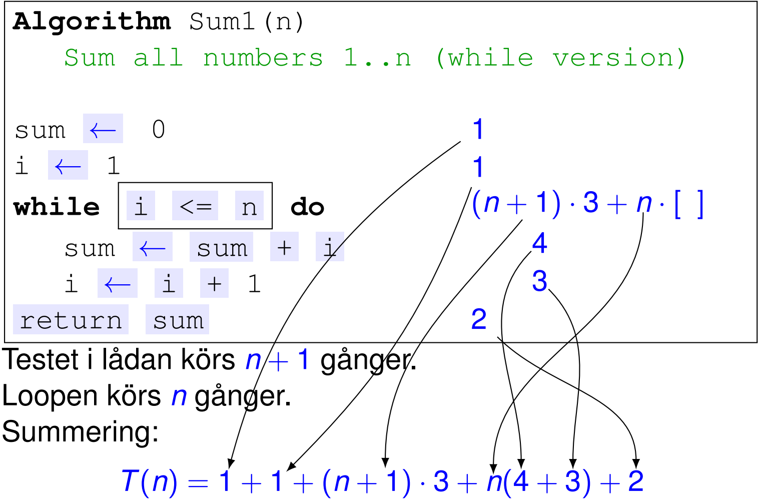
Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                          1
i ← 1                            1
while i <= n do                  $(n+1)\cdot 3+n\cdot[\ ]$
    sum ← sum + i                    4
    i ← i + 1                        3
return sum                       2
```

- Testet i lådan körs $n+1$ gånger.
- Loopen körs n gånger.

Primitiva operationer, Exempel 1



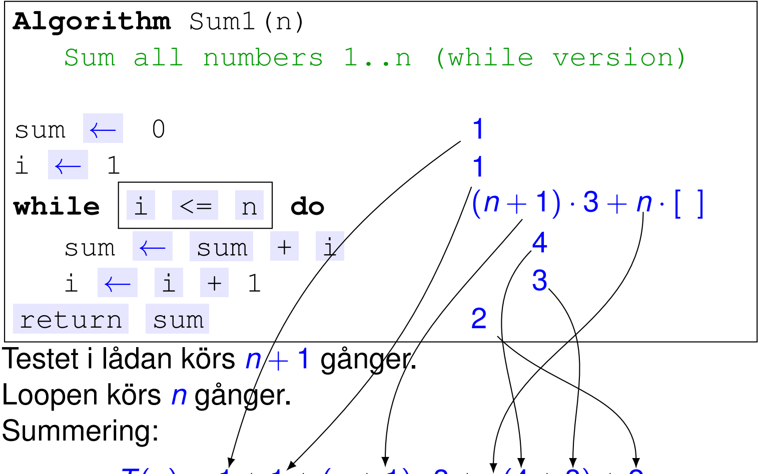
```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0
i ← 1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

- Testet i lådan körs $n+1$ gånger.
- Loopen körs n gånger.
- Summering:

$$T(n) = 1 + 1 + (n+1)\cdot 3 + n(4+3) + 2$$

Primitiva operationer, Exempel 1



```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                          1
i ← 1                            1
while i <= n do                  $(n+1) \cdot 3 + n \cdot [\ ]$
    sum ← sum + i                4
    i ← i + 1                    3
return sum                       2
```

- Testet i lådan körs $n + 1$ gånger.
- Loopen körs n gånger.
- Summering:

$$T(n) = 1 + 1 + (n+1) \cdot 3 + n(4+3) + 2$$

$$= 2 + 3n + 3 + 7n + 2$$

Primitiva operationer, Exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                  1
i ← 1                    1
while i <= n do          $(n+1) \cdot 3 + n \cdot [\ ]$
    sum ← sum + i        4
    i ← i + 1            3
return sum               2
```

- Testet i lådan körs $n+1$ gånger.
- Loopen körs n gånger.
- Summering:

$$T(n) = 1 + 1 + (n+1) \cdot 3 + n(4+3) + 2$$

$$= 2 + 3n + 3 + 7n + 2 = \boxed{10n+7}.$$

Förenklad asymptotisk analys

- Rita kurvor för $T(n)$ och jämföra är svårt och behövs ofta inte.
- Exempel: För

  $$T(n) = 10n + 7,$$

  hitta en funktion $g(n)$ som begränsar $T(n)$!
  - Finns oändligt många. Hitta den “minsta”/enklaste!
- Oftast så räcker med en förenklad asymptotisk analys:
  - Ignorera allt utom den dominerande termen, dvs. lägre ordningens termer och konstanter.
  - Använd inga koefficienter i $g(n)$.
- Exempel:
  - $T(n) = 10n + 7$ är $O(n)$.
  - $T(n) = 8n^3 + 5n^2 + n - 10$ är $O(n^3)$.

Hitta $g(n)$, c, n_0, exempel 1

- $T(n) = 10n + 7$ är $O(n)$ enligt den förenklade analysen, dvs. $\boxed{g(n) = n}$.
- Hitta c:

$$c = \lim_{n \to \infty} \left(\frac{T(n)}{g(n)} \right) + 1 = \lim_{n \to \infty} \left(\frac{10n + 7}{n} \right) + 1$$
$$= \lim_{n \to \infty} \left(10 + \frac{7}{n} \right) + 1 = \boxed{11}.$$

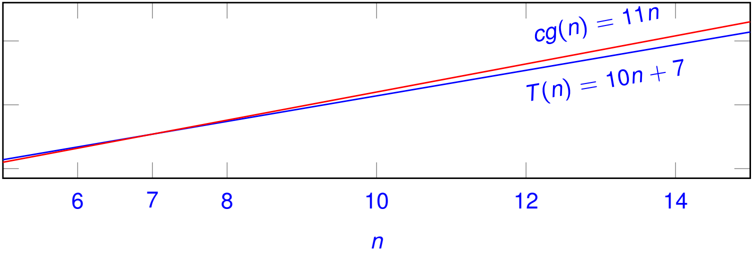
- Hitta n_0:

$$10n + 7 \leq 11n \Rightarrow \boxed{n_0 = 7}.$$

- Ok enligt ordo-definitionen!

Hitta $g(n)$, c, n_0, exempel 1

- $T(n) = 10n + 7$.
- $g(n) = n, c = 11, n_0 = 7$:



Hitta $g(n)$, c, n_0, exempel 2

- $T(n) = 7n - 3$ är $O(n)$ enligt den förenklade analysen, dvs. $\boxed{g(n) = n}$.
- Hitta c:

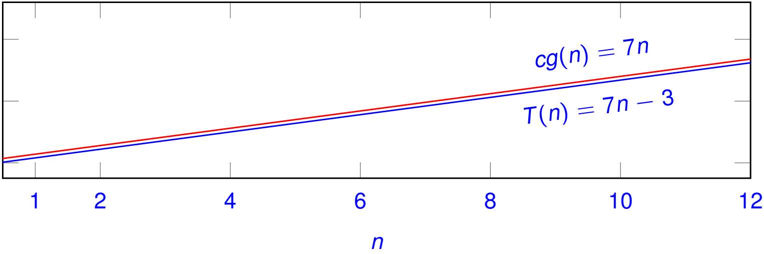
$$c = \lim_{n\to\infty} \left(\frac{T(n)}{g(n)}\right) = \lim_{n\to\infty} \left(\frac{7n-3}{n}\right)$$
$$= \lim_{n\to\infty} \left(7 - \frac{3}{n}\right) = \boxed{7}.$$

- Hitta n_0:

$$7n - 3 \leq 7n \Rightarrow \boxed{n_0 = 1}.$$

Hitta $g(n)$, c, n_0, exempel 2

- $T(n) = 7n - 3$.
- $g(n) = n, c = 7, n_0 = 1$:



Frågor

- Vad betyder $T(n) = O(1)$?
- Hur gör vi om $T(n) = 3.5n?$.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0
for i ← 1 to n do
    sum ← sum + i
return sum
```

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                    1
for i ← 1 to n do
    sum ← sum + i
return sum
```

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                    1
for i ← 1 to n do          1
    sum ← sum + i
return sum
```

- Initial tilldelning.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                        $1$
for i ← 1 to $\boxed{n}$ do    $1 + (n + 1) \cdot 3$
    sum ← sum + i
return sum
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n + 1$ gånger.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                          $1$
for i ← 1 to n do                $1 + (n + 1) \cdot 3 + 3n$
    sum ← sum + i
return sum
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n + 1$ gånger.
- Uppräkningen $i \leftarrow i + 1$ syns inte i pseudokoden, körs n gånger.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                          $1$
for i ← 1 to n do                $1 + (n + 1) \cdot 3 + 3n + n \cdot [\ ]$
    sum ← sum + i
return sum
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n + 1$ gånger.
- Uppräkningen $i \leftarrow i + 1$ syns inte i pseudokoden, körs n gånger.
- Loopen körs n gånger.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                      $1$
for i ← 1 to n do            $1 + (n+1) \cdot 3 + 3n + n \cdot [\ ]$
    sum ← sum + i               $4$
return sum
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n + 1$ gånger.
- Uppräkningen $i \leftarrow i + 1$ syns inte i pseudokoden, körs n gånger.
- Loopen körs n gånger.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                          1
for i ← 1 to n do                $1 + (n+1) \cdot 3 + 3n + n \cdot [\ ]$
    sum ← sum + i                    4
return sum                       2
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n + 1$ gånger.
- Uppräkningen $i \leftarrow i + 1$ syns inte i pseudokoden, körs n gånger.
- Loopen körs n gånger.

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                      $1$
for i ← 1 to n do            $1+(n+1)\cdot 3+3n+n\cdot[\ ]$
    sum ← sum + i                $4$
return sum                   $2$
```

- Initial tilldelning.
- Testet syns inte i pseudokoden, körs $n+1$ gånger.
- Uppräkningen $i \leftarrow i+1$ syns inte i pseudokoden, körs n gånger.
- Loopen körs n gånger.
- Summering:

$$T(n) = 1+1+3(n+1)+3n+4n+2 = \boxed{10n+7}.$$

Primitiva operationer, exempel 3

```
Algorithm SumAllEven(n)
    Sum all even numbers 2..n

sum ← 0                      1
i ← 2                        1
while i <= n do              $(n/2+1) \cdot 3 + n/2 \cdot [\ ]$
    sum ← sum + i                4
    i ← i + 2                    3
return sum                   2
```

- Loopen körs $n/2$ gånger.
- Summering:

$$T(n) = 1 + 1 + 3(n/2+1) + n/2(4+3) + 2 = \boxed{5n+7}.$$

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]
for i ← 1 to n−1 do
    if currentMax < A [ i ] then
        currentMax ← A [ i ]
return currentMax
```

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
   input: An array A storing n integers
   output: The maximum element of A
currentMax ← A [ 0 ]
for i ← 1 to n-1 do
   if currentMax < A [ i ] then
      currentMax ← A [ i ]
return currentMax
```

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]      3
for i ← 1 to n-1 do
    if currentMax < A [ i ] then
        currentMax ← A [ i ]
return currentMax
```

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
   input: An array A storing n integers
   output: The maximum element of A
currentMax ← A [ 0 ]            3
for i ← 1 to n-1 do             $1+3n+3(n-1)+(n-1)\cdot[\ ]$
   if currentMax < A [ i ] then
      currentMax ← A [ i ]
return currentMax
```

- Loopen körs $n-1$ gånger.
- Not: jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer.

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]          3
for i ← 1 to n-1 do           $1 + 3n + 3(n-1) + (n-1) \cdot [\ ]$
    if currentMax < A [ i ] then   5
        currentMax ← A [ i ]
return currentMax
```

- Loopen körs $n - 1$ gånger.
- Not: jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer.

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]                 3
for i ← 1 to n-1 do                  $1 + 3n + 3(n-1) + (n-1) \cdot [\ ]$
    if currentMax < A [ i ] then     5
        currentMax ← A [ i ]         4
return currentMax
```

- Loopen körs $n-1$ gånger.
- Not: jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer.

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]                    3
for i ← 1 to n-1 do                     $1 + 3n + 3(n-1) + (n-1) \cdot [\ ]$
    if currentMax < A [ i ] then        5
        currentMax ← A [ i ]            4
return currentMax                       2
```

- Loopen körs $n - 1$ gånger.
- Not: jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer.

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]                      3
for i ← 1 to n-1 do                       $1+3n+3(n-1)+(n-1)\cdot[\ ]$
    if currentMax < A [ i ] then          5
        currentMax ← A [ i ]              4
return currentMax                         2
```

- Loopen körs $n-1$ gånger.
- Not: jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer.
- Summering: Vi får två fall:

$$T_{\max}(n) = 3+1+6n-3+(n-1)9+2 = \boxed{15n-6}.$$

$$T_{\min}(n) = 3+1+6n-3+(n-1)5+2 = \boxed{11n-2}.$$

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)        n
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)              $n$
for i ← 0 to $\boxed{n-1}$ do        $1+3(n+1)+3n+n\cdot[\ ]$
    a ← 0
    for j ← 0 to $\boxed{i}$ do
        a ← a + X[j]
    A[i] ← a/(i+1)
return A
```

- Yttre loopen körs n gånger.

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)              $n$
for i ← 0 to $\boxed{n-1}$ do            $1 + 3(n+1) + 3n + n \cdot [\ ]$
    a ← 0                               $1$
    for j ← 0 to $\boxed{i}$ do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

- Yttre loopen körs n gånger.

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)                      $n$
for i ← 0 to n-1 do                     $1+3(n+1)+3n+n\cdot[\ ]$
    a ← 0                               $1$
    for j ← 0 to i do                   $1+3(i+2)+3(i+1)+(i+1)\cdot[\ ]$
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

- Yttre loopen körs n gånger.
- Inre loopen körs $i+1$ gånger.

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)              $n$
for i ← 0 to n-1 do             $1+3(n+1)+3n+n\cdot[\ ]$
    a ← 0                       $1$
    for j ← 0 to i do           $1+3(i+2)+3(i+1)+(i+1)\cdot[\ ]$
        a ← a + X [ j ]         $6$
    A [ i ] ← a / ( i + 1 )
return A
```

- Yttre loopen körs n gånger.
- Inre loopen körs $i+1$ gånger.

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)              $n$
for i ← 0 to n-1 do             $1 + 3(n+1) + 3n + n \cdot [\ ]$
    a ← 0                       $1$
    for j ← 0 to i do           $1 + 3(i+2) + 3(i+1) + (i+1) \cdot [\ ]$
        a ← a + X [ j ]         $6$
    A [ i ] ← a / ( i + 1 )     $7$
return A
```

- Yttre loopen körs n gånger.
- Inre loopen körs $i + 1$ gånger.

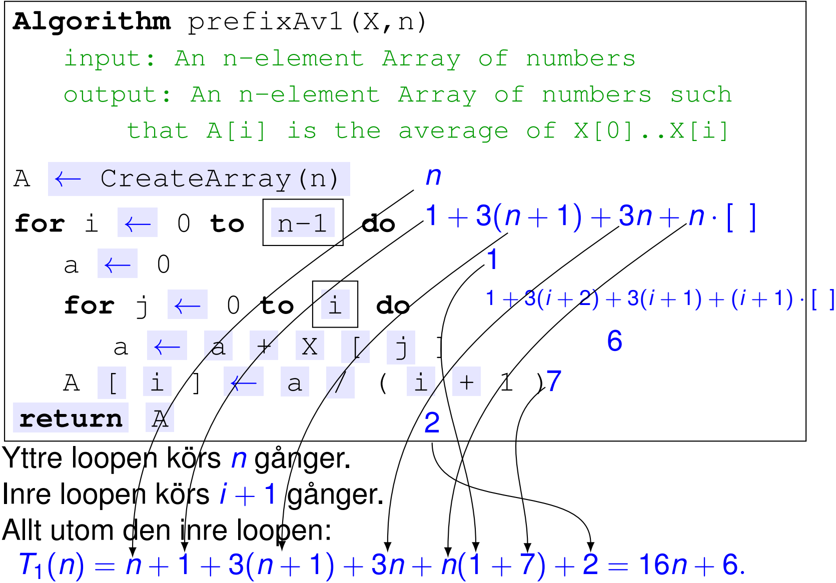
Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)                    $n$
for i ← 0 to n-1 do                   $1+3(n+1)+3n+n \cdot [\ ]$
    a ← 0                                 $1$
    for j ← 0 to i do                     $1+3(i+2)+3(i+1)+(i+1) \cdot [\ ]$
        a ← a + X [ j ]                       $6$
    A [ i ] ← a / ( i + 1 )7
return A                              $2$
```

- Yttre loopen körs n gånger.
- Inre loopen körs $i+1$ gånger.

Primitiva operationer, exempel 5



```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

n

$1+3(n+1)+3n+n\cdot[\ ]$

1

$1+3(i+2)+3(i+1)+(i+1)\cdot[\ ]$

6

7

2

- Yttre loopen körs n gånger.
- Inre loopen körs $i+1$ gånger.
- Allt utom den inre loopen:

$$T_1(n) = n + 1 + 3(n+1) + 3n + n(1+7) + 2 = 16n + 6.$$

Primitiva operationer, exempel 5 (2)

```
Algorithm prefixAv1(X,n)

for i ← 0 to $\boxed{n-1}$ do              $n \cdot [\ ]$
    for j ← 0 to $\boxed{i}$ do            $10 + 6i + (i+1) \cdot [\ ]$
        a ← a + X [ j ]                    $6$
```

Primitiva operationer, exempel 5 (2)

```
Algorithm prefixAv1(X,n)

for i ← 0 to $\boxed{n-1}$ do              $n \cdot [\ ]$
    for j ← 0 to $\boxed{i}$ do            $10 + 6i + (i+1) \cdot [\ ]$
        a ← a + X [ j ]                    $6$
```

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?

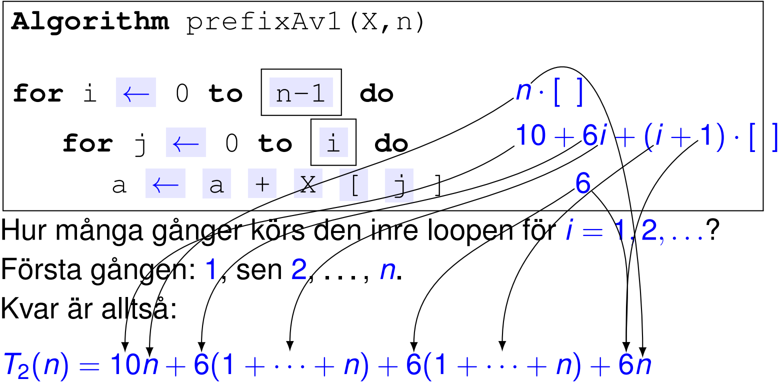
Primitiva operationer, exempel 5 (2)

```
Algorithm prefixAv1(X,n)

for i ← 0 to $\boxed{n-1}$ do                  $n \cdot [\ ]$
    for j ← 0 to $\boxed{i}$ do                $10 + 6i + (i+1) \cdot [\ ]$
        a ← a + X [ j ]                        $6$
```

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen 2, ..., n.

Primitiva operationer, exempel 5 (2)



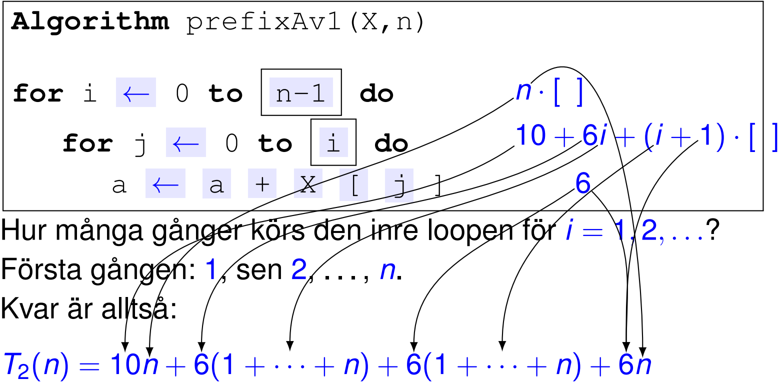
```
Algorithm prefixAv1(X,n)

for i ← 0 to n-1 do                 $n \cdot [\ ]$
    for j ← 0 to i do               $10 + 6i + (i + 1) \cdot [\ ]$
        a ← a + X [ j ]             $6$
```

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen 2, ..., n.
- Kvar är alltså:

$$T_2(n) = 10n + 6(1 + \cdots + n) + 6(1 + \cdots + n) + 6n$$

Primitiva operationer, exempel 5 (2)



```
Algorithm prefixAv1(X,n)

for i ← 0 to n−1 do
    for j ← 0 to i do
        a ← a + X [ j ]
```

$n \cdot [\ ]$

$10 + 6i + (i+1) \cdot [\ ]$

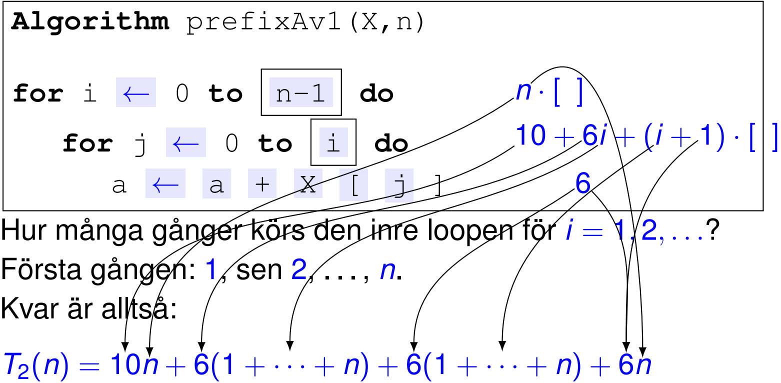
6

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen 2, …, n.
- Kvar är alltså:

$$T_2(n) = 10n + 6(1 + \cdots + n) + 6(1 + \cdots + n) + 6n$$

$$= 16n + 12 \sum_{i=1}^{n} i$$

Primitiva operationer, exempel 5 (2)



```
Algorithm prefixAv1(X,n)

for i ← 0 to n-1 do
    for j ← 0 to i do
        a ← a + X [ j ]
```

$n \cdot [\ ]$

$10 + 6i + (i+1) \cdot [\ ]$

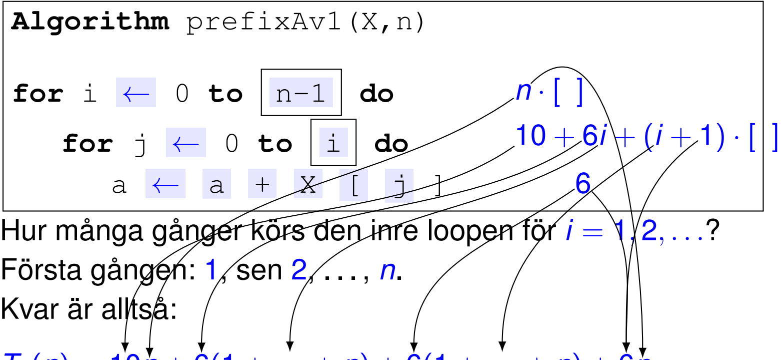
6

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen 2, …, n.
- Kvar är alltså:

$$T_2(n) = 10n + 6(1 + \cdots + n) + 6(1 + \cdots + n) + 6n$$

$$= 16n + 12\sum_{i=1}^{n} i = 16n + 12\frac{n^2+n}{2}$$

Primitiva operationer, exempel 5 (2)



```
Algorithm prefixAv1(X,n)

for i ← 0 to n−1 do
    for j ← 0 to i do
        a ← a + X[j]
```

$n \cdot [\ ]$

$10 + 6i + (i+1) \cdot [\ ]$

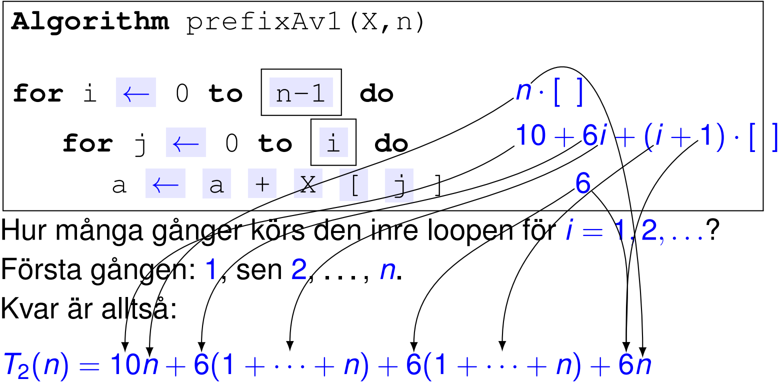
6

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen 2, …, n.
- Kvar är alltså:

$$T_2(n) = 10n + 6(1 + \cdots + n) + 6(1 + \cdots + n) + 6n$$

$$= 16n + 12\sum_{i=1}^{n} i = 16n + 12\frac{n^2+n}{2} = 6n^2 + 22n.$$

Primitiva operationer, exempel 5 (2)



```
Algorithm prefixAv1(X,n)

for i ← 0 to n-1 do
    for j ← 0 to i do
        a ← a + X [ j ]
```

$n \cdot [\ ]$

$10 + 6i + (i+1) \cdot [\ ]$

6

- Hur många gånger körs den inre loopen för $i = 1, 2, \ldots$?
- Första gången: 1, sen $2, \ldots, n$.
- Kvar är alltså:

$$T_2(n) = 10n + 6(1 + \cdots + n) + 6(1 + \cdots + n) + 6n$$

$$= 16n + 12 \sum_{i=1}^{n} i = 16n + 12 \frac{n^2 + n}{2} = 6n^2 + 22n.$$

$$T(n) = T_1(n) + T_2(n) = 16n + 6 + 6n^2 + 22n = \boxed{6n^2 + 38n + 6}$$

Primitiva operationer, exempel 6

```
Algorithm prefixAv2(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)               $n$
s ← 0                            $1$
for i ← 0 to n-1 do              $1+3(n+1)+3n+n\cdot[\ ]$
    s ← s + X [ i ]              $6$
    A [ i ] ← s / ( i + 1 )      $7$
return A                         $2$
```

- Summering:

$$T(n)=n+1+1+3n+3+3n+n(6+7)+2=\boxed{20n+5}.$$

Typalgoritmer

- Speciella klasser av algoritmer/problem:

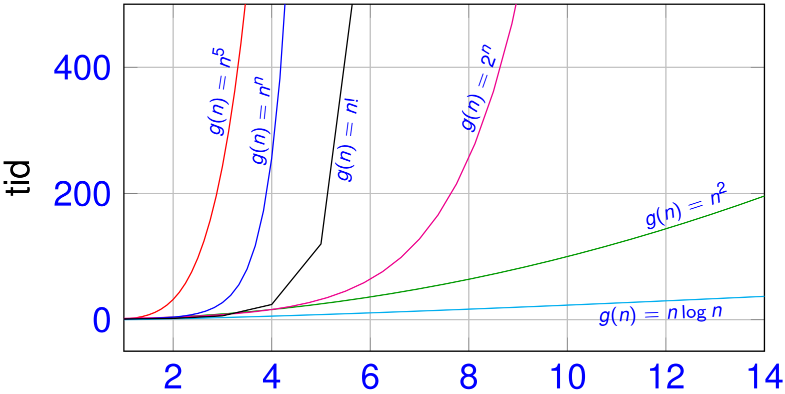
Logaritmiska	$O(\log n)$,
Linjära	$O(n)$,
Kvadratiska	$O(n^2)$,
Polynoma	$O(n^k), k \geq 1$,
Kombinatoriska	$O(n!)$,
Exponentiella	$O(a^n), a > 1$.

- Typalgoritmerna är ordnade enligt:

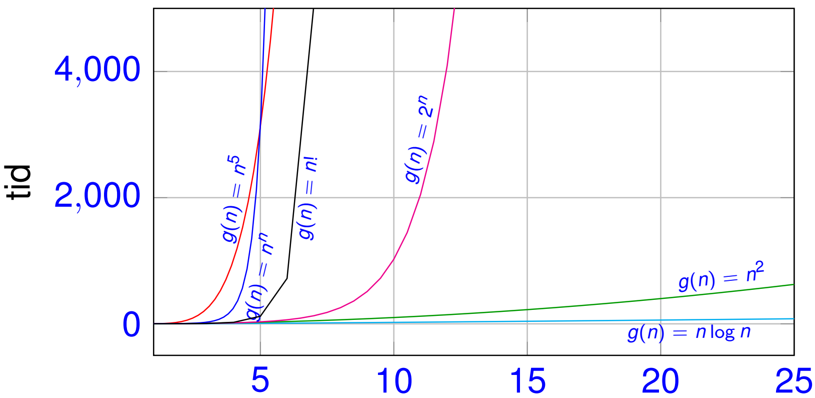
$$\log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n.$$

- Exempel:
  - $T(n) = 10n + 7$ är $O(n)$.
  - $T(n) = 8n^3 + 5n^2 + n - 10$ är $O(n^3)$.
  - $T(n) = 8n^2 \log n + 10n^2$ är $O(n^2 \log n)$.

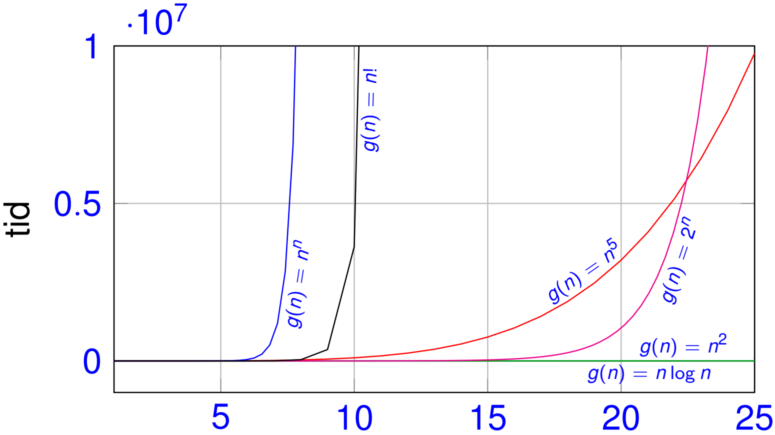
Typalgoritmer



Typalgoritmer



Typalgoritmer



- Slutsats: “Ohanterliga” problem kan vara hanterliga för små n.

Ordo, sammanfattning

- $O(n)$ används för att utrycka antalet primitiva operationer som utförs som en funktion av storleken på indata n.
- En övre gräns för tillväxt.
- `arrayMax` är en linjär algoritm dvs. $O(n)$.
- En algoritm som körs på $O(n)$ tid är bättre än en $O(n^2)$, men $O(\log n)$ är ännu bättre.

$$\log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n.$$

Ordo, varning

- Var aktsam, stora konstanter ställer till det:
  - $T_1(n) = 1000000n$ är en linjär algoritm $O(n)$.
  - $T_2(n) = 2n^2$ är en kvadratisk algoritm $O(n^2)$.
  - $T_2(n)$ är mer effektiv för “små” datamängder, $n < n_0 = 5 \cdot 10^{13}$.
- O-notationen är en stor förenkling, en övre gräns. Det finns släktingar som begränsar nedåt.
- O-notationen har tagit bort kopplingen till hårdvaran.

Ordo, genväg

- En genväg för att få en grov uppskattning av tillväxten:
- Okulärbesikta algoritmen:
  - Initiera en array är $O(n)$.
  - Nästlade loopar är $O(n) \cdot O(n) \cdots O(n) = O(n^k)$.

Grovanalys, exempel 1

```
Algorithm prefixAv1(X,n)
   input: An n-element Array of numbers
   output: An n-element Array of numbers such
       that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
   a ← 0
   for j ← 0 to i do
      a ← a + X [ j ]
   A [ i ] ← a / ( i + 1 )
return A
```

Grovanalys, exempel 1

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

- En initiering, två loopar:
  $T(n) = O(n) + O(n)O(n) = \boxed{O(n^2)}$.

Grovanalys, exempel 1

```
Algorithm prefixAv1(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
    a ← 0
    for j ← 0 to i do
        a ← a + X [ j ]
    A [ i ] ← a / ( i + 1 )
return A
```

- En initiering, två loopar:
  $T(n) = O(n) + O(n)O(n) = \boxed{O(n^2)}$.
- Detaljerad analys gav $T(n) = 6n^2 + 38n + 6 = \boxed{O(n^2)}$.

Grovanalys, exempel 2

```
Algorithm prefixAv2(X,n)
   input: An n-element Array of numbers
   output: An n-element Array of numbers such
       that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
s ← 0
for i ← 0 to n-1 do
   s ← s + X [ i ]
   A [ i ] ← s / ( i + 1 )
return A
```

Grovanalys, exempel 2

```
Algorithm prefixAv2(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
s ← 0
for i ← 0 to n-1 do
    s ← s + X [ i ]
    A [ i ] ← s / ( i + 1 )
return A
```

- En initiering, en loop: $T(n) = O(n) + O(n) = \boxed{O(n)}$.

Grovanalys, exempel 2

```
Algorithm prefixAv2(X,n)
   input: An n-element Array of numbers
   output: An n-element Array of numbers such
       that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
s ← 0
for i ← 0 to n-1 do
   s ← s + X [ i ]
   A [ i ] ← s / ( i + 1 )
return A
```

- En initiering, en loop: $T(n) = O(n) + O(n) = \boxed{O(n)}$.
- Detaljerad analys gav $T(n) = 20n + 5 = \boxed{O(n)}$.

Minneskomplexitet

- Som asymptotisk komplexitetsanalys för tid.
  - “primitiva operationer”⇒ “hur mycket minne behöver vi”?
  - Glöm inte minnet för lokala variabler, etc., som läggs upp på stacken vid rekursion.
- Ofta är minnet en “hård” begränsning medan tid är en “mjuk”.
  - Åtkomsttiden när minnet tar “slut” (disk används i stället) ökar 20–100++ ggr.
  - Ansatsen blir att räkna ut “vilket är det största problem som ryms i minnet”?
  - Ofta relativt enkelt att räkna ut c (hur många bytes som behövs per element).
- Mer komplex analys tar hänsyn till många olika begränsningar:
  - tid, minne, filåtkomst, kommunikation (nätverk, mellan CPU:er), osv.

Frågor

- Hur påverkas `prefixAv1` om `CreateArray` har komplexitet *kn* och *k* okänt?
- Hur påverkas `prefixAv2` om `CreateArray` har komplexitet *kn* och *k* okänt?

Blank

Blank

Blank

Primitiva operationer, exempel 1

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0
i ← 1
while i <= n do
    sum ← sum + i
    i ← i + 1
return sum
```

Primitiva operationer, exempel 2

```
Algorithm Sum2(n)
   Sum all numbers 1..n (for version)

sum ← 0
for i ← 1 to n do
   sum ← sum + i
return sum
```

Primitiva operationer, exempel 3

```
Algorithm SumAllEven(n)
    Sum all even numbers 2..n

sum ← 0
i ← 2
while i <= n do
    sum ← sum + i
    i ← i + 2
return sum
```

Primitiva operationer, exempel 4

```
Algorithm arrayMax(A,n)
   input: An array A storing n integers
   output: The maximum element of A
currentMax ← A [ 0 ]
for i ← 1 to n-1 do
   if currentMax < A [ i ] then
      currentMax ← A [ i ]
return currentMax
```

Primitiva operationer, exempel 5

```
Algorithm prefixAv1(X,n)
   input: An n-element Array of numbers
   output: An n-element Array of numbers such
       that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
for i ← 0 to n-1 do
   a ← 0
   for j ← 0 to i do
      a ← a + X [ j ]
   A [ i ] ← a / ( i + 1 )
return A
```

Primitiva operationer, exempel 6

```
Algorithm prefixAv2(X,n)
    input: An n-element Array of numbers
    output: An n-element Array of numbers such
        that A[i] is the average of X[0]..X[i]

A ← CreateArray(n)
s ← 0
for i ← 0 to n-1 do
    s ← s + X [ i ]
    A [ i ] ← s / ( i + 1 )
return A
```