Implementasjon

EN ADT AV EN ANNEN DT

- Data representasjon
- Data Invariant

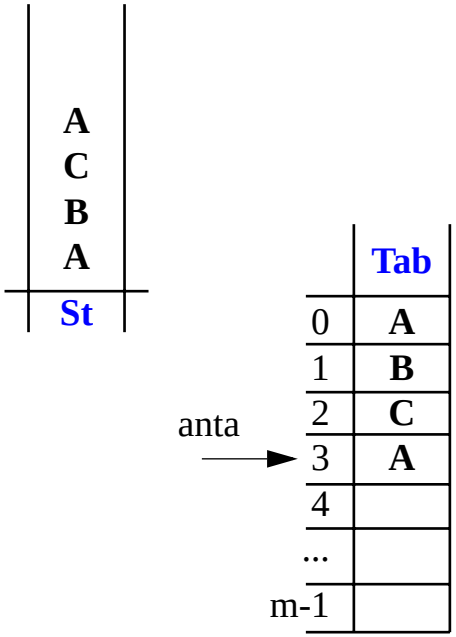
ALGORITMEANALYSE

- Asymptotisk notasjon

Kap. 3

Implementasjon (***Stack*** med array)

```
public interface Stack {
   void push(Object o);
   Object pop() throws
      EmptyStackException;
   Object peek() throws
      EmptyStackException;
   boolean empty();
...}
```



I DATA REPRESENTASJON

Hvordan skal en stabel representeres ?

- **en array med en peker tilsvarende peek()**
- **rekkefølge i array omvendt av den i Stabel**

II DATA STRUKTUR

Hvilke konkrete data trenger jeg å holde rede på ?

class Stab

- **Object[] Tab**
- **int antall**
- **int max**

III DATA INVARIANT

DS kan holde forskjellige "inkonsistente" verdier – de må bindes på en måte som tilsvarer Stabel og denne "binding" må opprettholdes

antall – Tab[antall] er toppen av stabel – hvis antall+1 = antall elementer > 0

Må passe på at **antall** *ikke går forbi* **Tab.length** *!*

max = Tab.length > antall >= –1

Opprettholdelse av DataInvarianten

antall – Tab[antall] er toppen av stabel – hvis
antall+1 = antall elementer > 0
max = Tab.length > antall >= –1

```
public class Stab implements Stack {
    private Object[] Tab;
    private int antall, max=10;

    public Stab() {
        Tab= new Object[max]; antall= –1;
    }                                   // er DI ok etter initialisering ?

    public void push(Object o) {        // hvis: DI ok ved kall
        if (antall=max-1) {
            Object[] temp= new Object[max];
            Copy(Tab, temp);
            max= 2*max; Tab= new Object[max];
            Copy(temp,Tab);  }
        antall++;
        Tab[antall]= o;
    }                                   //  er DI ok ved retur ?

    public Object pop()                 // hvis: DI ok ved kall
        throws EmptyStackException {
        if (empty())
            throw new EmptyStackException("Tom");
        else { antall--; return Tab[antall+1]; }
    }                                   //  er DI ok ved retur ?

    public Object peek()                // hvis: DI ok ved kall
        throws EmptyStackException {
        if (empty())
            throw new EmptyStackException("Tom");
        else return Tab[antall];
    }                                   // er resultatet ok  ?

    public boolean empty() {            // hvis: DI ok ved kall
        return antall < 0;
    }                                   // er resultatet ok ?

    private void Copy(Object[] fra, Object[] til) {
                                        // hvis: DI ok ved kall
        for (int k=0; k<fra.length; k++)  til[k]= fra[k];
    }                                   // er DI ok ved retur  ?
}
```

DataInvarianten kan – og bør – implementeres

antall – Tab[antall] er toppen av stabel – hvis
antall+1 = antall elementer > 0
max = Tab.length > antall >= –1

```
public class Stab implements Stack {
  private Object[] Tab; private int antall, max=10;
  public Stab() throws DIExc {  Tab= new Object[max]; antall= –1;
   if (!DI()) throw new DIExc("Init feil");
  }
  public void push(Object o) throws DIExc {
   if (!DI()) throw new DIExc("DI feil før push");
   if (antall=max-1) {
     Object[] temp= new Object[max];
     Copy(Tab, temp);
     max= 2*max; Tab= new Object[max];
     Copy(temp,Tab);  }
   antall++;
   Tab[antall]= o;
   if (!DI()) throw new DIExc("DI feil etter push");
  }
  public Object pop()  throws DIExc,
     EmptyStackException {
   if (!DI()) throw new DIExc("DI feil før pop");
   if (empty())
     throw new EmptyStackException("Tom");
   else { antall--;
     if (!DI()) throw new DIExc("DI feil etter pop");
     else return Tab[antall+1]; }
  }
```

```
protected boolean DI() {
  if (antall >= –1 && antall < max && max==Tab.length)
     return true;
  else return false;
}
public Object peek() throws DIExc,
      EmptyStackException {
  if (!DI()) throw new DIExc("DI feil før peek");
  if (empty())
     throw new EmptyStackException("Tom");
  else return Tab[antall];
}
public boolean empty() throws DIExc {
  if (!DI()) throw new DIExc("DI feil ved empty");
  return antall < 0;
}
private void Copy(Object[] fr, Object[] ti) throws DIExc {
  if (!DI()) throw new DIExc("DI feil");
  for (int k=0; k<fr.length; k++)  ti[k]= fr[k];
  if (!DI()) throw new DIExc("DI feil etter kopi");
}
```

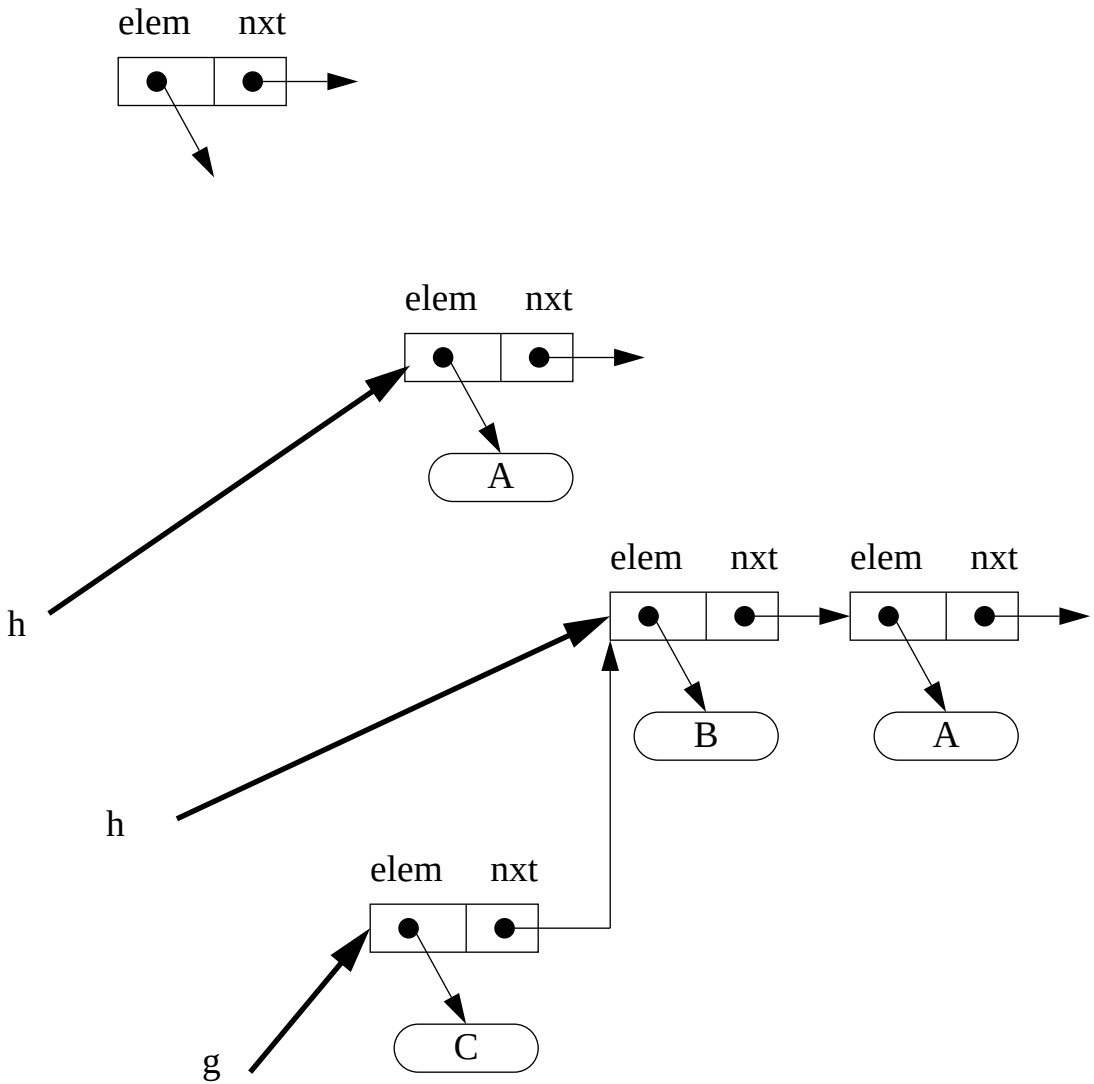
En-veis Lenket Liste - datastruktur

```
public class Node {
   private Object elem;
   private Node next;
   public Node(Object o, Node n) {
       elem= o; next= n;}
   public Node() { this(null, null); }
   public Object getElem() { return elem; }
   public Node getNext() { return next; }
   public void setElem(Object o) { elem= o; }
   public void setNext(Node n) { next= n; }
}
```

Node h = new Node(A,null);

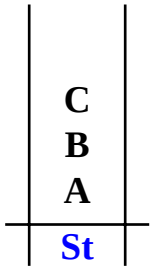
h = new Node(B,h);

Node g = new Node(C,h)



Implementasjon (*Stack* med LenketListe)

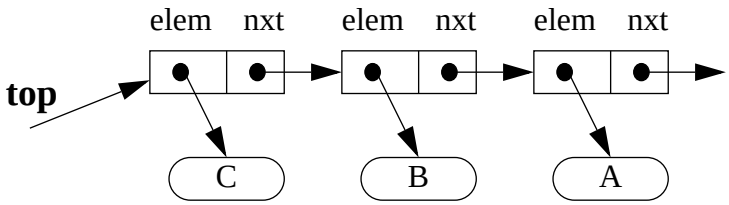
```
public interface Stack {
   void push(Object o);
   Object pop() throws
      EmptyStackException;
   Object peek() throws
      EmptyStackException;
   boolean empty();
...}
```



I Data Representasjon

Hvordan skal en stabel representeres ?

- **en klasse med en peker til top-node = peek()**
- **innsetting skjer i starten av listen**



II Data Struktur

Hvilke konkrete data trenger jeg å holde rede på ?

class StackLL

– Node top

III Data Invariant

DS kan holde forskjellige "inkonsistente" verdier – de må bindes på en måte som tilsvarer Stabel og denne "binding" må opprettholdes

top == peker alltid på toppen av stabel

== null hviss stabel er tom

.nxt == peker på elementet "under"

Opprettholdelse av DataInvarianten

```
public class StackLL implements Stack {
  private Node top;

  public StackLL() { top= null; }
                              // er DI ok etter initialisering ?

  public void push(Object o) {        //hvis: DI ok ved kall
    top= new Node(o,top);
  }                                   // er DI ok ved retur ?

  public Object pop()                 // hvis: DI ok ved kall
    throws EmptyStackException{
    if (empty()) throw
      new EmptyStackException("Tom ved pop");
    else {
      Object u = top.getElem();
      top = top.getNext();
      return u;
  } }                                 // er DI ok ved retur ?
```

top == peker på toppen av stabel
== null hviss stabel er tom
.nxt == peker på elementet "under"

```
  public Object peek()  // hvis: DI ok ved kall
   throws EmptyStackException {
      if (empty()) throw
                  new EmptyStackException("Tom");
      else return top.getElem();
  }                                   // er resultatet ok ?

  public boolean empty()  {
                                // hvis: DI ok ved kall
      return (top == null);
  }                                   // er resultatet ok ?
}
```

Oppsummering: implementasjon (av en ADT med en konkret klasse)

1. ***DataRepresentasjon:*** *hvordan kan* ***ADT****-elementene representeres v.hj.a.* ***DT****-elementene*

– det kan godt hende at du trenger flere konkrete DT'er, ***I1****,* ***I2****...****In****, for å holde all nødvendig informasjon*

2. ***DataStruktur:*** *velg en måte å binde sammen de relevante aspektene fra* ***I1****...****In***

– beskriv abstraksjonsfunksjonen ***Abs***
(denne avbilder kun de instansene av ***DT*** *som oppfyller* ***DataInvarianten****)*

3. ***DataInvarianten:*** *for din DataStruktur; spesifiser*

– hvilke instanser av DataStruktur faktisk representerer abstrakte verdier fra ***ADT***

– hvordan karakteriseres disse med relasjoner mellom attributter av forskjellige ***I1****...****In***

4. Implementer operasjonene fra ***ADT*** *på den valgte DataStrukturen*

– implementasjonen skal oppfylle spesifikasjonen (gitt i interface ; for- og bakbetingelser)

– verifiser, evt. implementer ***DataInvarianten****, forsikre deg om at*

- ***DI*** *holder etter initialisering*

og for hver implementert/konkret operasjon ***f*** *i din* ***DT****:*

- *dersom* ***DI*** *holder før* ***f*** *kalles, så holder den også etter at* ***f*** *returnerer*

5. Se hvor ***effektiv*** *din implementasjon er*

Effektivitet = tidskompleksitet

Tidsforbruk = *antall primitive (atomære) operasjoner*

Primitive operasjoner:

- *sammenlikning*
- *aritmetikk (+, /, ...)*
- *aksess (A[x], P.k)*
- *tilordning,*
- *metodekall,*
- *return*
- . . .

```
                                                          P(op)
Stab(){ Tab= new Object[max]; antall= –1;}                  3
void push(Object o) { if (antall==max-1) {                  2
      Object[]temp = new Object[max];                       2
      Copy(Tab,tab); max= 2*max;                          max+2
      Tab = new Object[max];                                2
      Copy(tab,Tab);  }                                   max
   antall++; Tab[antall] = o;                               3
}                                                    5 / 11+ 2*max
Object pop() { if (empty()) return null;                  1+1
   else { antall--; return Tab[antall+1];}                  3
}                                                          2/4
Object peek() { if (empty()) return null;                 1+1
   else return Tab[antall];                                 2
}                                                          2/3
boolean empty()  { return (antall < 0); }                   2
```

```
                                                          P(op)
StackLL() { top= null; }                                    1
void push(Object o) {
   top= new Node(o,top);  }                                 2
Object pop() {
   if (empty()) return null;                              1+1
   else { Object u= top.getElem();                          3
      top= top.getNext(); return u;                         4
} }                                                        2/8
Object peek() { if (empty()) return null;                 1+1
   else return top.getElem();                               3
}                                                          2/4
boolean empty() {return (top==null);}                       2
```

- **P**(Stab.push(**n**)) = 11+2*max > 2 = **P**(LL.push(**n**))
- *Alt i LL tar konstant – uavhengig av* **n** *– tid*
- Stab.push *avhenger av* **max** *– ikke av* **n**

Tidskompleksitet = tidsforbruk *som funksjon av inputstørrelse* **n**

En operasjon *op* er *av orden 1*, *op* = **O**(1),
hviss den tar konstant tid for tilstrekkelig stor input

$$\exists n_0 \ \exists k \ \forall n \ (n > n_0 \rightarrow \mathbf{P}(op(n)) \leq k*1)$$

For push/pop/peek i LL: k=2/8/4 og n_0=1

Algoritmeanalyse

```
/*    SS - sorterer input array (SeleksjonSort)
 *       @param  - int tab[0...n]
 *       @return - sortert tab
 *
 *    for (k = 0,1,2...n) {
 *       m = k
 *       for ( j = k+1...n)
 *          if (tab[j] < tab[m]) m = j;
 *       bytt elementene ved indeks k og m
 *       }
 */
```

Hvor mange ganger må jeg utføre basis operasjoner ?
sammnlikn. + flytting

k= 0	2	4	**1**	3	5	4 + 2
k= 1	**1**	4	**2**	3	5	3 + 2
k= 2	**1**	**2**	4	**3**	5	2 + 2
k= 3	**1**	**2**	**3**	**4**	5	1 + 1
k= 4	**1**	**2**	**3**	**4**	**5**	0 + 0
						10 + 7 = 17

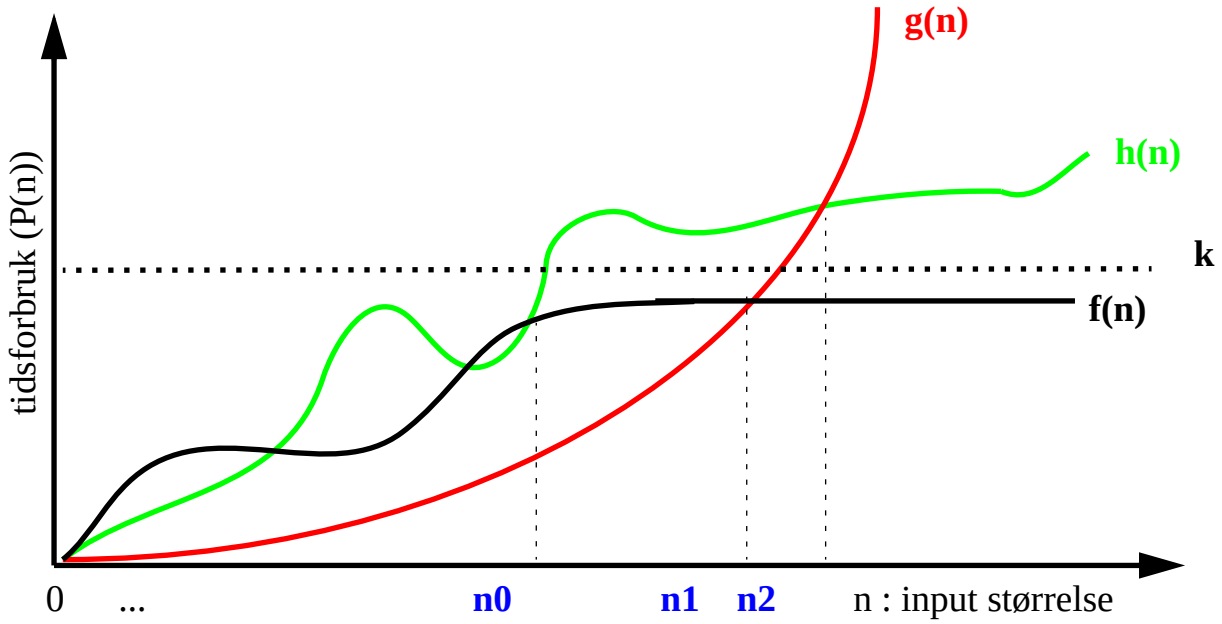
for en vlikårlig input tabell med lengde **n***:*

- *utfører* **n** *iterasjoner (for k=1,2...***n***) og*
- *i hver iterasjon utfører 2 flytting/tilordninger :* **2 * n**, *og*
- *i hver iterasjon går gjennom sluttsegment* **[k...n]**, *(for j=k+1...n), med så mange sammenlikninger:*

$$\left(\sum_{k=1}^{n} k = 1 + 2 + \ldots + (n-1) + n = \frac{n + n^2}{2} \right) + 2 * n = \mathbf{2.5 * n + 0.5 * n^2} = \mathbf{O(n^2)}$$

Asymptotisk notasjon: f(n) = **O**(g(n))

En funksjon f(n) *er av orden* g(n), f(n)= O(g(n)), *hviss*
det finnes konstanter **k** *og* **n0** s.a. ∀n (n > **n0** → f(n) ≤ **k** * g(n))



for alle n > **n1** : **f(n)** ≤ **1** * **g(n)** :
f(n) = **O**(g(n))

for alle n > **n0** : **f(n)** ≤ **1** * **h(n)** :
f(n) = **O**(h(n))

for alle n > **0** : **f(n)** ≤ **k** * **h(n)** :
f(n) = **O**(h(n))

for alle n > **0** : **f(n)** ≤ **1** * **k** :
f(n) = **O**(k)

for alle n > **n2** : **h(n)** ≤ **1** * **g(n)** :
h(n) = **O**(g(n))

for enhver konstant **c** >= 1:

for alle n > **0** : f(n) ≤ **k** * **c** – f(n) = **O**(**c**)
for alle n > **0** : f(n) ≤ **k** * **1** – f(n) = **O**(**1**) **!!!**
for alle n > **n2** : h(n) ≤ **1** * **g(n)** – h(n) = **O**(**g(n)**)

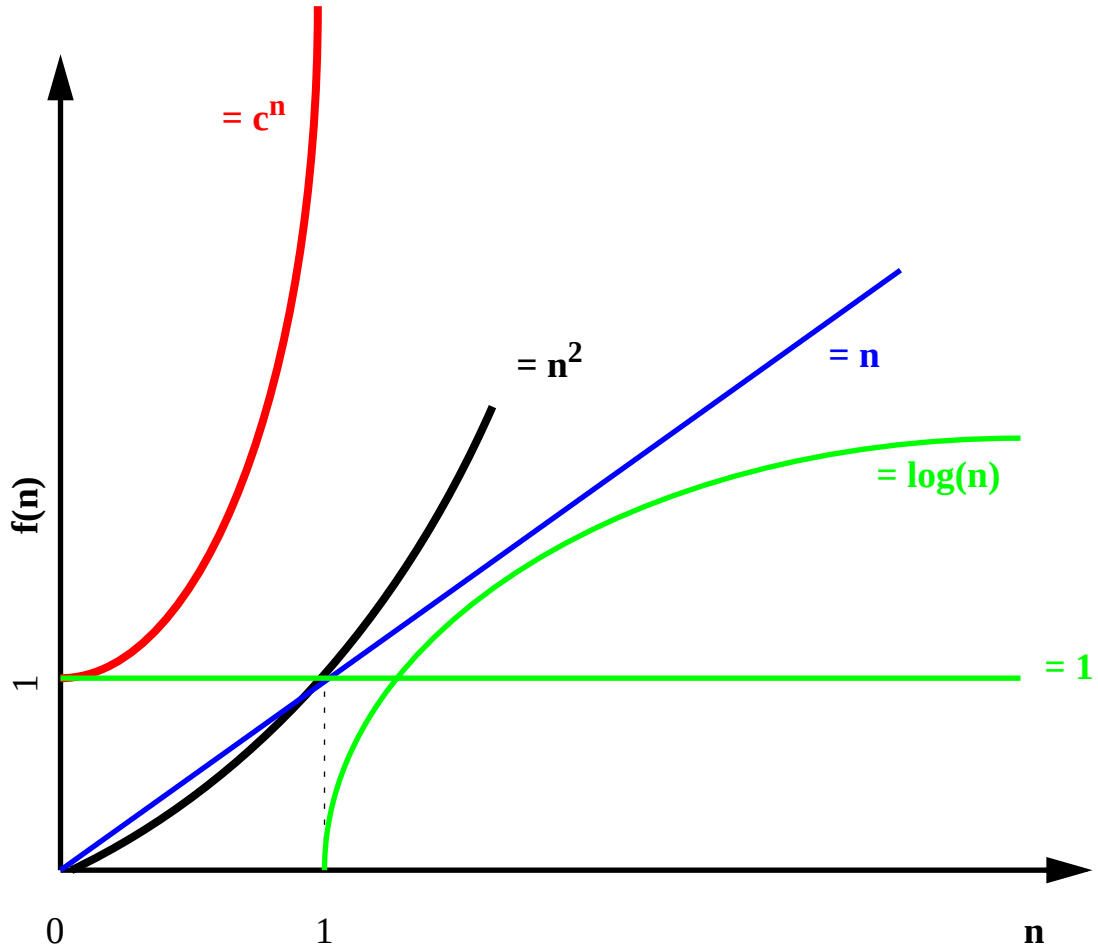
En tommelfinger regel:

Glem lavereordens termer og konstanter:

f(n) = 2 * $\mathbf{n^3}$ + 16 * n^2 + 5000 = ***O***(n^3)

f(n) = 8 * $\mathbf{n^2}$ * **log(n)** + 3*n = ***O***(n^2 * log(n))

Mest vanlige kompleksitets-klasser



konstant $O(1)$

$1 =_O 5 =_O 100 =_O 2^{100} =_O \ldots$

$<_O$ **logaritmisk** $O(\log(n))$

$5 * \mathbf{log(n)} =_O 2^{100} * \mathbf{log(n)} =_O \ldots$

$<_O$ **linær** $O(n)$

$5 * \mathbf{n} =_O 400 * \mathbf{n} - 3 =_O \mathbf{n} + \log(n) =_O \ldots$

$<_O$ **kvadratisk** $O(n^2)$

$5 * \mathbf{n^2} =_O 125 * \mathbf{n^2} + 5 * n - 3 =_O \ldots$

$<_O$ **polynomisk** $O(n^k)$ **k >1**

$5 * \mathbf{n^3} + 5 * n^2 + 5 * n + 3 <_O n^4 <_O n^6 <_O ..$

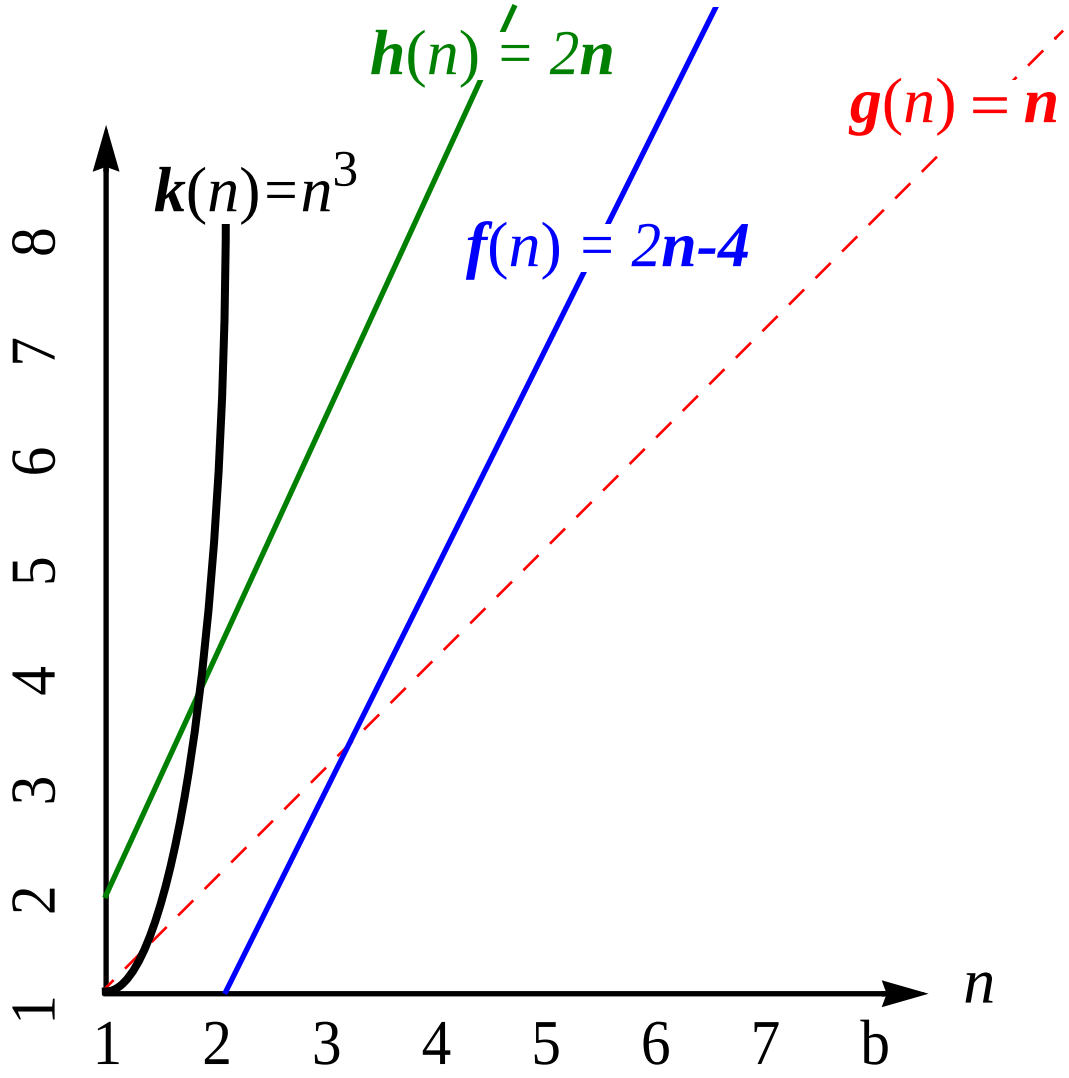
$<_O$ **eksponensielt** $O(k^n)$ **k >1**

$5 * \mathbf{2^n} <_O \mathbf{3^n} =_O 3^n + n <_O \mathbf{5^n} <_O \ldots$

Asymptotisk notasjon

- NB: Det er sant at
  $2n - 4 = O(n^3)$,
  men en bedre påstand er at
  $2n - 4 = O(n)$

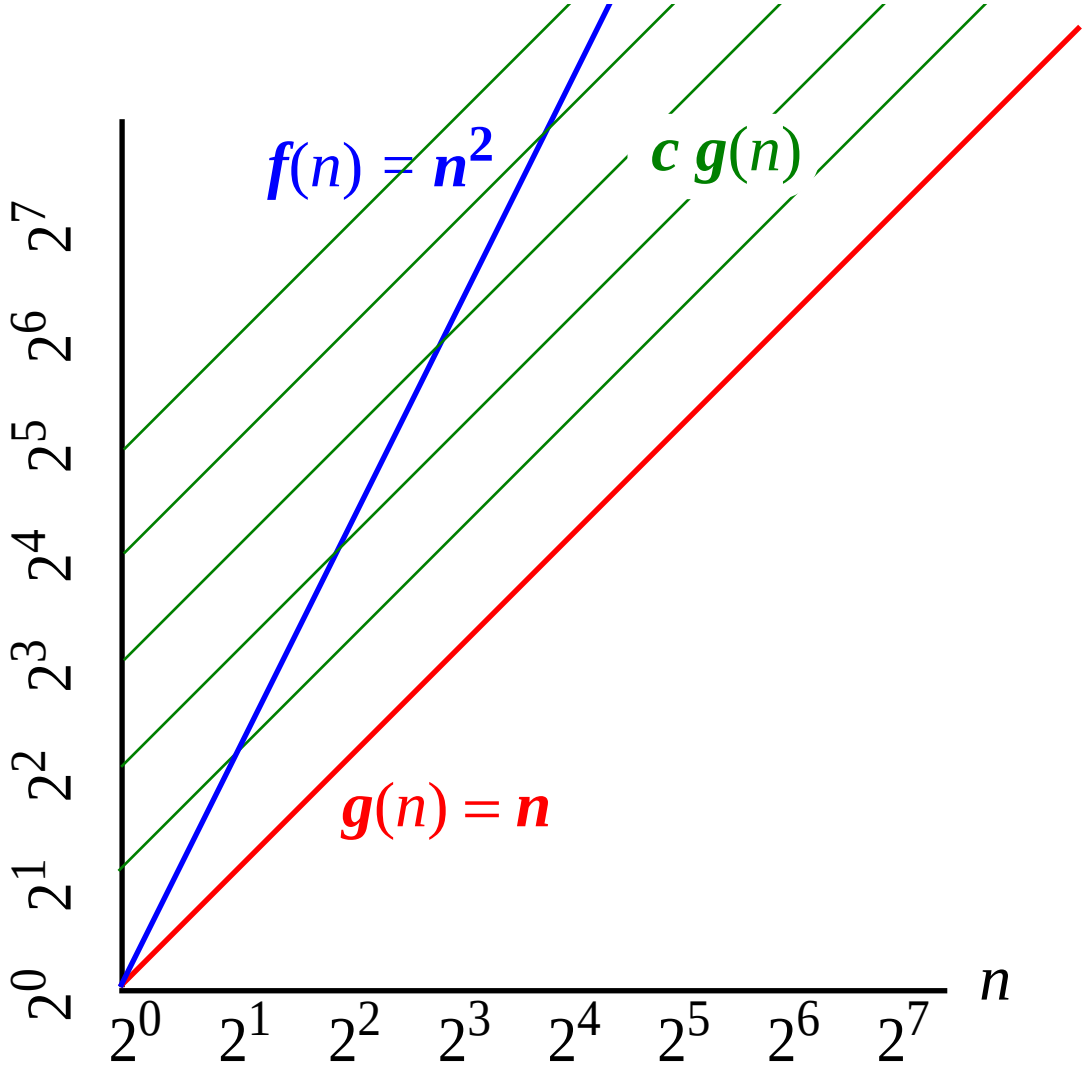
- approksimer så tett som mulig



Asymptotisk notasjon

- n^2 **er ikke** O(n)

- det finnes ikke c og n_0 alik at $n^2 \leq c\,n$ for $n \geq n_0$



Konstanter teller!

Gitt 3 algoritmer A, B og C med kjøretid hhv. O(n), O(n*logn) og $O(n^2)$, men forskjellige "skjulte" konstanter, f.eks., hhv. 400*n, 20*n*logn og $2*n^2$.

Stor-O forteller oss at asymptotisk, dvs når n vokser, så er A bedre enn B bedre enn C. I praksis vil dette avhenge av inputstørrelsen n.

	***O*( )**	faktisk	n =		
			10	100	1000
A	n	400 * n	4 000	40 000	400 000
B	n * log(n)	$20*n*\log_{10}(n)$	200	4 000	60 000
C	n^2	$2 * n^2$	200	20 000	2 000 000
	n^4	n^4	10 000	100000000	....
	2^n	2^n	1 024	....	....

Konstanter teller ... men ikke SÅ mye

I det lange løp vil alltid stor-O notasjon skille korrekt.
I særdeleshet dersom vi vil se på virkningen av å skaffe en raskere maskin:

	***O*( )**	faktisk	problemstørrelse – løses på 1 sekund		
			dagens maskin	60-gngr raskere	3600-gngr raskere
A	n	400 * n	2 500	150 000	9 000 000
B	n * log(n)	20*n*log(n)	4 000	170 000	8 000 000
C	n^2	$2 * n^2$	700	5 500	42 000
	n^4	n^4	30	90	250
	2^n	2^n	20	25	30

Med en maskin som er 3600-ganger raskere så løser vi et problem (i samme tid) som er

$O(n)$: 3600-ganger (= 9.000.000 / 2.500) større.

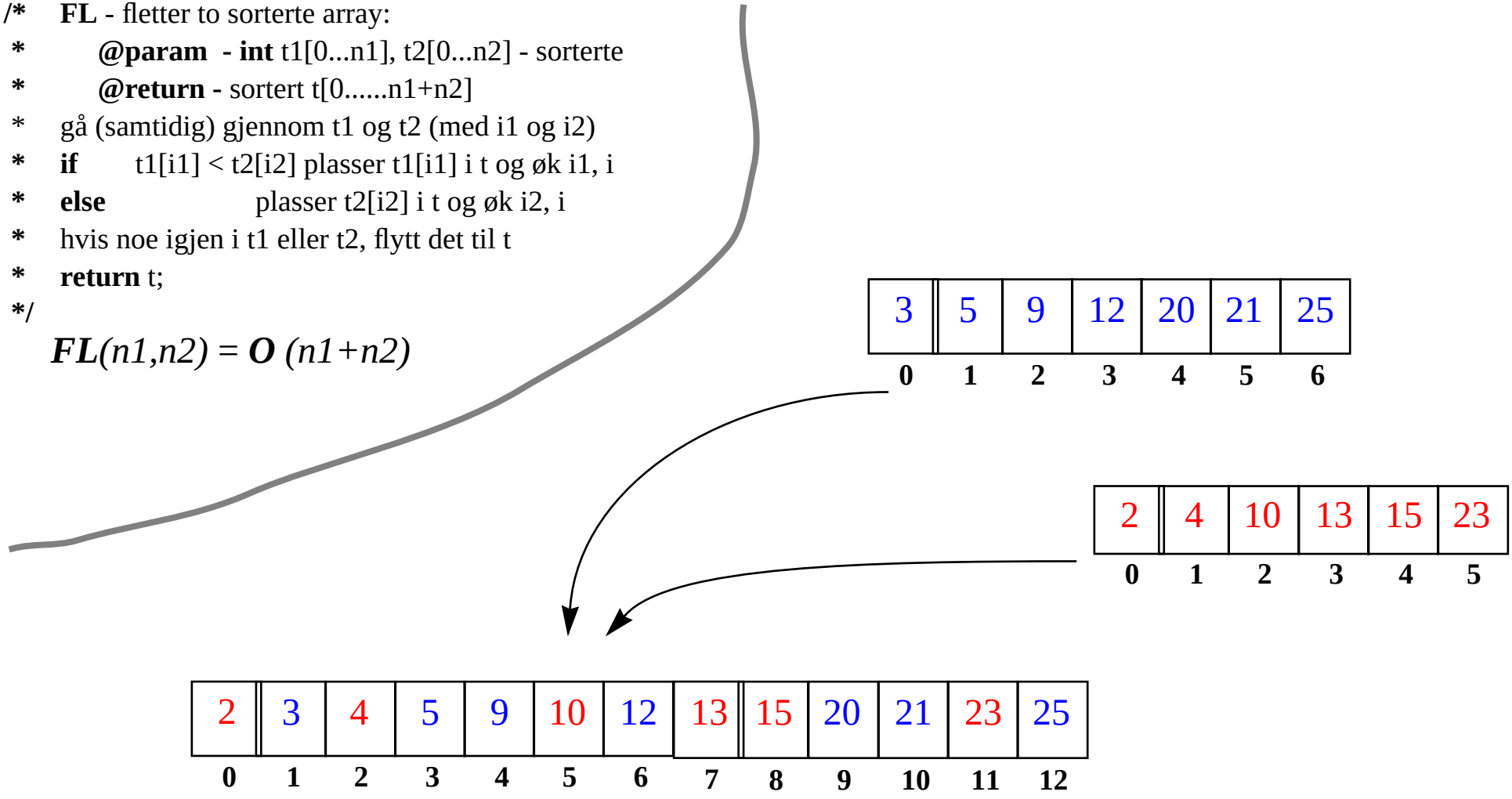
$O(n^2)$: 60-ganger (= 42.000 / 700) større.

$O(2^n)$: kun 10 enheter større – økningen er kun additiv!

Flett

```
/*    FL - fletter to sorterte array:
 *        @param  - int t1[0...n1], t2[0...n2] - sorterte
 *        @return - sortert t[0.....n1+n2]
 *    gå (samtidig) gjennom t1 og t2 (med i1 og i2)
 *    if       t1[i1] < t2[i2] plasser t1[i1] i t og øk i1, i
 *    else                     plasser t2[i2] i t og øk i2, i
 *    hvis noe igjen i t1 eller t2, flytt det til t
 *    return t;
 */
```

$$FL(n1,n2) = O\ (n1+n2)$$

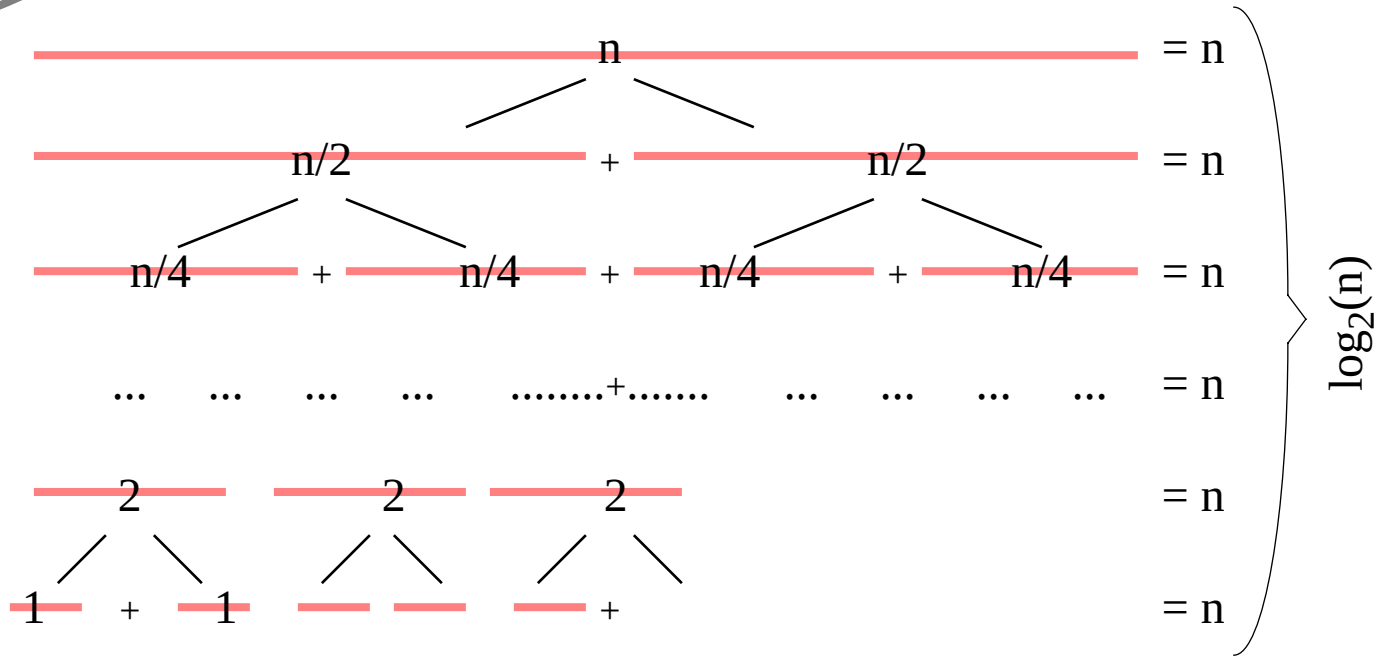


MergeSort

```
/*   FL - fletter to sorterte array:
 *       @param  - int t1[0...n1], t2[0...n2] - sorterte
 *       @return - sortert t[0.....n1+n2]
 *   gå (samtidig) gjennom t1 og t2 (med i1 og i2)
 *   if      t1[i1] < t2[i2] plasser t1[i1] i t og øk i1, i
 *   else                    plasser t2[i2] i t og øk i2, i
 *   hvis noe igjen i t1 eller t2, flytt det til t
 *   return t;
 */
```

***FL**(n1,n2) = **O** (n1+n2)*

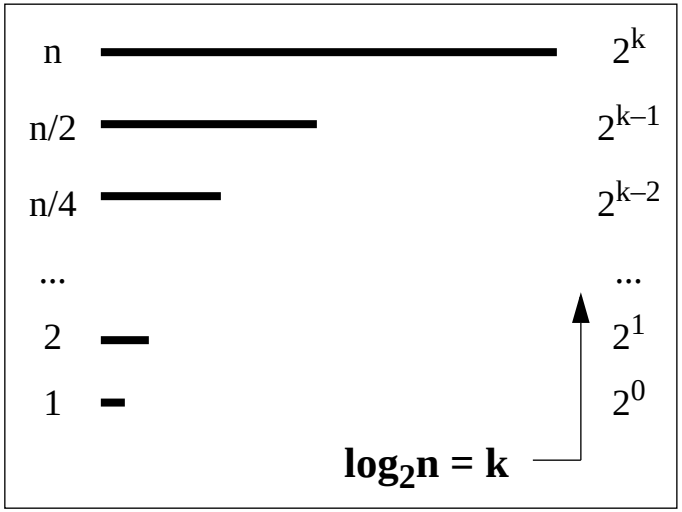
```
/*   MS - sorterer input array:
 *       @param  - int tab[0...n-1]
 *       @return - sortert tab
 *   if (n == 1) return tab
 *   else { k= n/2;
 *       return FL ( MS(tab[0...k]), MS(tab[k+1..n-1]) ); }
 */
```

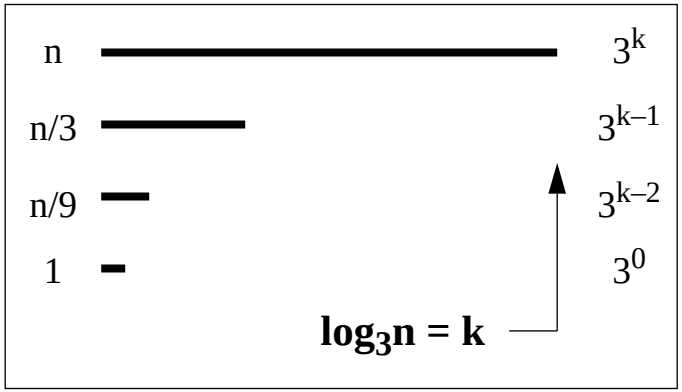


***MS**(n) = **O** (n * log$_2$(n))*

Noen enkle fakta

$\log_b n = k$ hviss **$b^k = n$** (**$n = b^k$**) $\log_b n$





$$\log_b n = \frac{\log_c n}{\log_c b}$$

$$\log_3 n = \frac{\log_2 n}{\log_2 3} = O$$

$$\log(n * m) = \log(n) + \log(m)$$

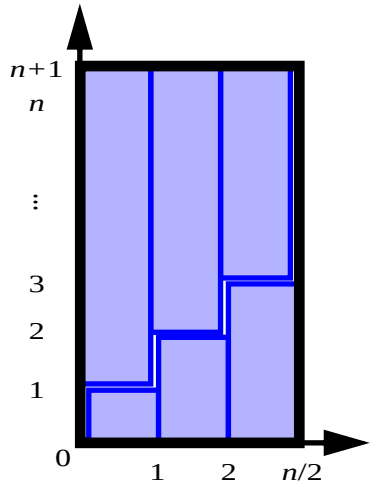
$$\log(n^c) = c * \log(n) = O$$

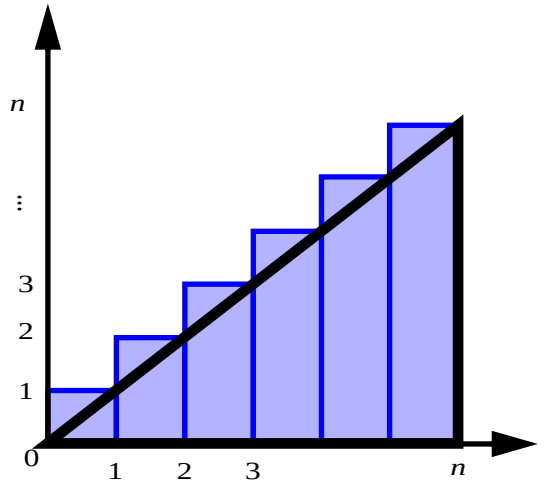
$$\log(n/m) = \log(n) - \log(m)$$

$$1 + 2 + 3 + \ldots + n = \sum_{k=1}^{n} k = \frac{n^2 + n}{2}$$

$$a^0 + a^1 + \ldots + a^n = \sum_{k=0}^{n} a^k = \frac{1 - a^{n+1}}{1 - a}$$

($0 < a \neq 1$)





O-notasjon

Gitt heltallsfunksjoner f, g, h : N → N

- f(n) = $\boldsymbol{O}$(g(n)) det finnes c, n0: n > n0 ! f(n) ≤ c * g(n) $f \leq_{\boldsymbol{O}} g$
  $\boldsymbol{o}$(g(n)) $f <_{\boldsymbol{O}} g$
- f(n) = Ω(g(n)) hvis g(n) = $\boldsymbol{O}$(f(n)) $f \geq_{\boldsymbol{O}} g$
  ω(g(n)) $f >_{\boldsymbol{O}} g$
- f(n) = Θ(g(n)) hvis f(n) = $\boldsymbol{O}$(g(n)) og g(n) = $\boldsymbol{O}$(f(n)) $f =_{\boldsymbol{O}} g$

***O** forekommer ikke på venstre side av =*

- Hvis f(n) = $\boldsymbol{O}$(h(n)) og h(n) = $\boldsymbol{O}$(g(n)) så også f(n) = $\boldsymbol{O}$(g(n))

  f(n) = $\boldsymbol{O}$(h(n))

  h(n) = $\boldsymbol{O}$(g(n))

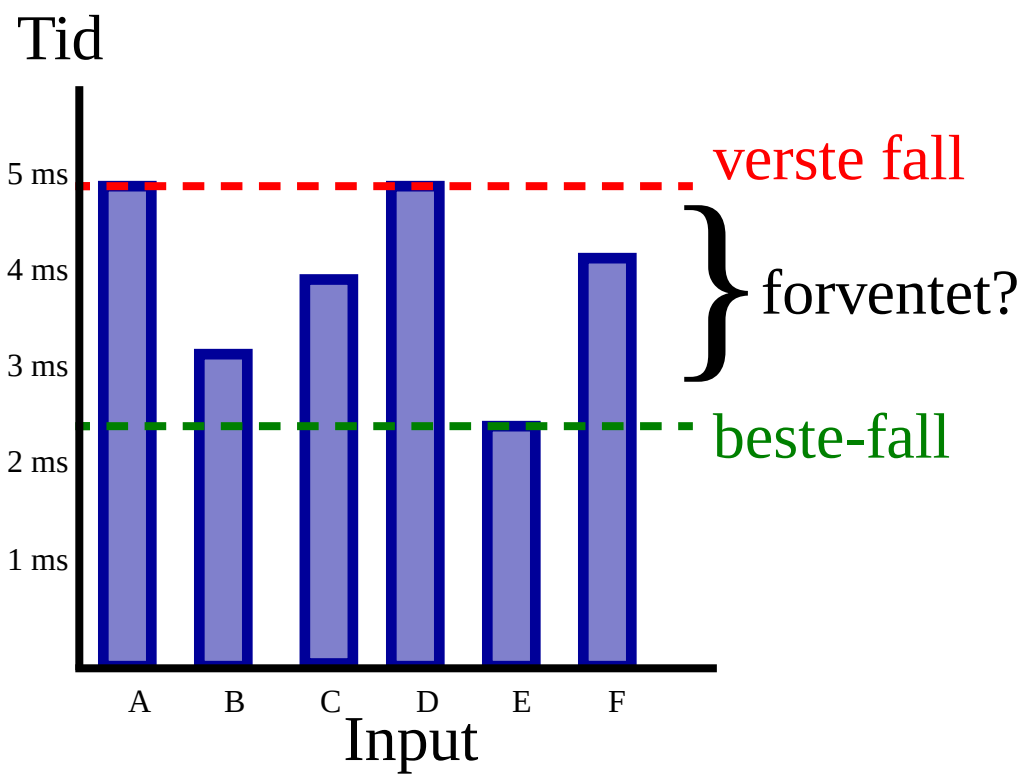
  f(n) = $\boldsymbol{O}$(g(n))
- f(n) + g(n) = $\boldsymbol{O}$(max{ f(n), g(n) })
- Hvis f(n) = $\boldsymbol{O}$(h(n)) så f(n) + g(n) = $\boldsymbol{O}$(h(n) + g(n))
- Hvis f(n) = $\boldsymbol{O}$(h(n)) så f(n) * g(n) = $\boldsymbol{O}$(h(n) * g(n))
- for en gitt c: $\log(n^c) = \boldsymbol{O}(\log(n))$

Typisk ja, men i Verste Fall ...

Finn indeks til et element x i en array A[0...n]

```
/* int finn(x)
 *       i= 0; r= –1; funnet= false;
 *       while (!funnet) {
 *           if   (Tab[i] == x)
 *               r = i; funnet = true;
 *           else i++;
 *           if (i == Tab.lenght)
 *               funnet = true;   }
 *       return r ;
 */
```

	Tab
0	**A**
1	**B**
2	**C**
3	**Z**
4	**V**
5	**Y**
n= 6	**X**



finn(A) : 1

i beste fall : 1 ***O*** (1)

finn(Z) : 3

gjennomsnittlig : n/2 ***O*** (n)

finn(X) : 7

i verste fall : n ***O*** (n)

O-notasjon brukes nesten utelukkende for verste-fall analyse

Rekursjon og kompleksitet

Rekursjon som en generell strategi for problemløsning og algoritmedesign

Gitt en instans **n** *av et problem* **P** *:*

1. *hva gjør jeg når* **n** *er basis tilfelle*
2. *hvordan konstruere løsning for* **n** *utfra løsninger for noen instanser* **mindre enn n**

P = sorter input array A

```
/* int[] SS(int[] A,k) {
 * initielt kall med SS(A,0)
 * n = A.length;
 *    if (k==n-1) { return A; }
 *    else {
 *       i= indeksen til minste elementet
 *          i A[k...n-1];
 *       bytt A[k] med A[i];
 *       return SS(A, k+1); } }   */
```

$$\sum_{k=1}^{n} k = (n + n^2)/2 = O(n^2)$$

```
/* int[] MergeSort ( int[] A )  {
 *    int n = A.length;
 *    if (n == 1) { return A; }
 *    else {
 *       del A i midten i
 *       t1 = A[0...n/2] og t2 = A[n/2+1...n];
 *       sorter rekursivt begge (mindre) array
 *             r1 = MergeSort ( t1 ) og
 *             r2 = MergeSort ( t2 )
 *       return flettet resultat av disse FL(r1,r2)
 *    } }
 * FL - fletter to sorterte array i en sortert array */
```

$O(n \log n)$

Rekursjon og kompleksitet

Rekursjon som en generell strategi for problemløsning og algoritmedesign

Gitt en instans **n** *av et problem* **P** *:*

1. *hva gjør jeg når* **n** *er basis tilfelle*
2. *hvordan konstruere løsning for* **n** *utfra løsninger for noen instanser* **mindre enn n**

P = finn et gitt element x i en array A

Hvis A *er usortert :*

sjekk A[n]*;*
hvis x ikke er der, lett i A[0...n–1]

```
/*   initielt kall med FE(A, x?, A.length–1)   */
int FE(int[] A, x. k) {
    if (k < 0 )  return –1;
    else if (A[k]==x)  return k;
    else return FE(A, x, k–1);   }

FE(A, 7, 11)
```

1	3	5	7	8	9	10	11	13	15	19	20
0	1	2	3	4	5	6	7	8	9	10	11

O(n)

Hvis A er sortert ...

```
/*   initielt kall med
* BS(A, x?, 0, A.length) */
int BS(int[] A, x, l, h) {
   m = (l+h) / 2;
   if (l > h) return -1;
   else if (A[m] == x) return m;
   else if (A[m] < x)
       return BS(A, x, m+1, h);
   else return BS(A, x, l, m–1); }
```

O(log n)

n

1	3	5	7	8	9	10	11	13	15	19	20
0	1	2	3	4	**5**	6	7	8	9	10	11

n/2

1	3	5	7	8
0	1	**2**	3	4

n/4

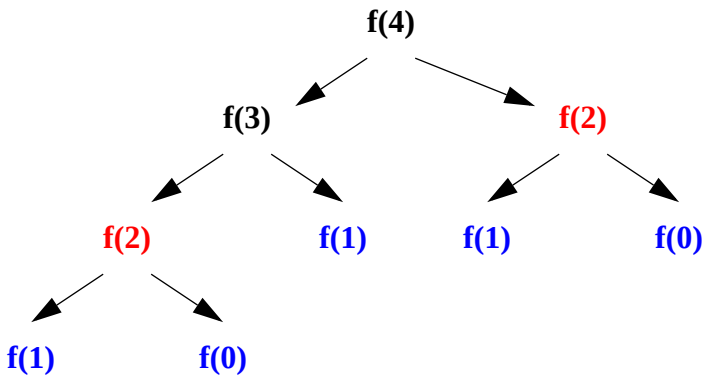
7	8
3	4

Rekursjon og effektivitet

– *Reduser antall rekursive kall* –

```
int fib(int n) {
   if (n==0 || n==1) return 1;
   else return fib(n-1)+fib(n-2);
}
```

$O(1.6^n)$



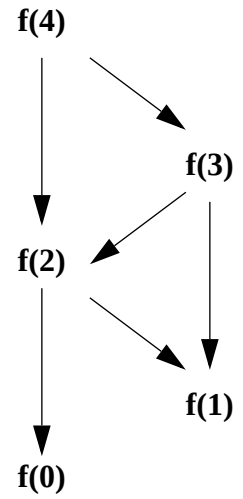
1. "Memoisering" :

*Istedenfor gjentatte rekursive kall til f(k) med samme k, kan **i dette tilfelle** resultatet av f(k) lagres for senere bruk:*

O(n)

```
int Fib(int n) {
   int[] ar= new int[n+1];
   ar[0]=1; ar[1]=1;
   return fibo(n, ar);
}

int fibo(int n, int[] ar) {
   if (ar[n] > 0) {
      return ar[n]; }
   else {
      int z= fibo(n–1,ar) + fibo(n–2,ar);
      ar[n]= z;
      return z;
} }
```



Kompleksitet av en rekursiv funksjon

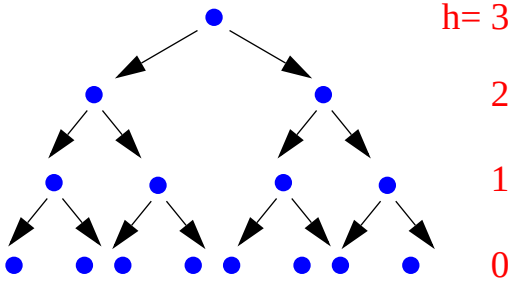
avhenger av

Analyse vha REKURSJONSTRE

- *"**størrelsen på steget**" i hvert rekursivt kall (høyden av treet)*
- ***antall rekursive kall** i hvert steg ("bredden" av forgreninger)*
- *arbeidsmengden ved "sammensetting" av resultater fra rekursive kall. **Anta dette O(1) i eksemplene under.***

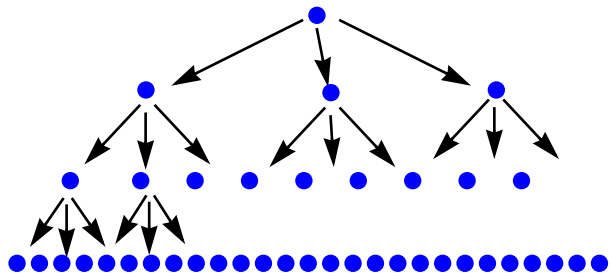
R(n) har 2 rekursive kall til R(n-1)

R(**n+1**) = R(**n**) + R(**n**) $O(2^{n+1} - 1)$



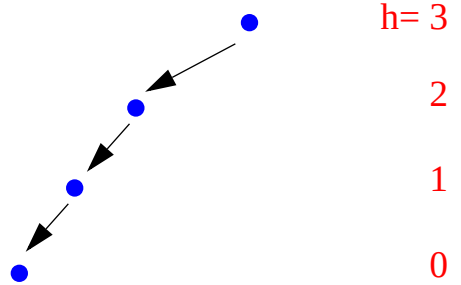
R(n) har 3 rekursive kall til R(n-1)

R(**n+1**) = R(**n**) + R(**n**) + R(**n**) $O(3^{n+1} - 1)$



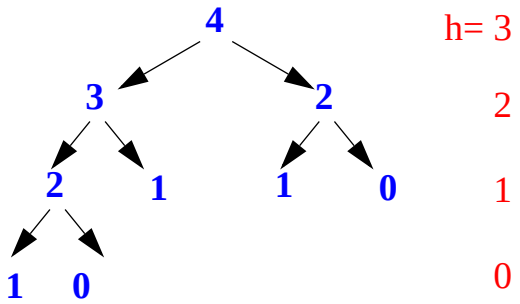
R(n) har 1 rek. kall til R(n-1)

R(**n+1**) = R(**n**) + c $O(n)$

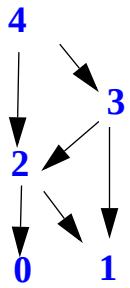


R(n) har rek kall til R(n-1) og R(n-2)

R(**n+2**) = R(**n+1**) + R(**n**) $O(1.6^n)$



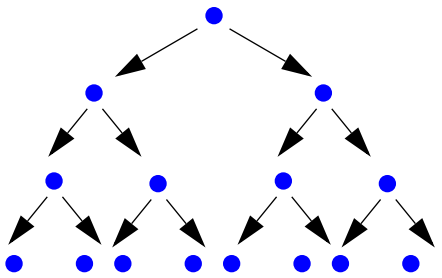
Fibonacci kan dog forenkles til: $O(n)$



R(n) har 2 rek. kall til R(n/2)

R(**n**) = R(**n/2**) + R(**n/2**)

$2^{\log(n)+1} - 1 = 2n - 1 = O(n)$



Oppsummering

Implementasjon av (A)DT

- *Data Representasjon*
- *Data Struktur*
- *Data Invariant*
- *Korrekthet*
  - *opprettholdelse av Data Invarianten*
  - *korrekt implementasjon av abstrakte operasjoner*

Effektivitet:

- *tidskompleksitet = hvordan avhenger tidsforbruket av størrelsen på input*
- *kompleksitets-klasser*
  - *linær*
  - *logaritmisk*
  - *polynomisk (kvadratisk)*
  - *eksponensiell*
- *verste-fall analyse*