I-120 (h-98)

Foreleser: Michal Walicki
e-mail: michal@ii.uib.no
kontor: HiB, 4126

Gruppeleder: Peter Ølveczky
e-mail: peter@ii.uib.no

I-120 webside: http://www.ii.uib.no/
! Kurs dette semesteret
! I120

Pensum:
- ''Data Structures and Algorithms using JAVA''
  M.T.Goodrich & R.Tamassia
  (deler)
- ev. notater delt ut på forelesning

Pensum: fra boken

(H-98, foreløpig)

	unntatt	kursorisk	poenger
KAP. 1	1.4.2		**OO, ABSTRAKSJON ...**
KAP. 2			**O-NOTASJON**
KAP. 3	3.2.4, 3.5	**3.1.3, 3.2.3, 3.4**	**STABEL, KØ, LISTE, ADAPTER**
KAP. 4			**SEQ; RANK, POS; ENUM-ITERATOR**
KAP. 5	5.5	**5.4.4**	**TRÆR, B-TRÆR + BSF/DFS (PRE/POST)**
KAP. 6	6.4	**6.3.4**	**PRIORITETSKØ, HEAP; TO/COMP**
KAP. 7	7.5, 7.7	**7.6.2, 7.6.3**	**ORDBOK, BST, HASHTAB**
KAP. 8	8.1.3, 8.2, 8.6	**8.4 8.5**	**QUICKSORT (MERGE); RANDOM**
KAP. 9		**9.4.5**	**GRAPH, DIGRAPH, DFS/BFS, FW**
KAP. 10	10.1.5, 10.3 10.2.2-10.2.4		**SS-SP, MST**

Mål og Mening

Ikke programmering (i-110)

men	effektiv	og	korrekt	programmering
dvs	algoritmer	og	strukturering	

type tekst/program	boksider	tekstlinjer	listelengde
Wolfe 'Forfengelighetens fyrverkeri'	661	26 440	132m
Tolkien 'Hobbiten+Ringtrilogien+Simarillion'	1 847	73 880	222m
enkel kompilator	300	12 000	36m
lønnssystem, industri	600	24 000	80m
utlånssystem, forsikring	7 500	300 000	900m
abonnentsystem, avis	10 650	425 000	1 275m
logistikksystem, oljeplatform	18 750	750 000	2 250m

program	mengde data : n	n*10
effektiv	1 min	10 min
lite effektiv	1 min	1 000 min

Mål og Mening

Effektivitet :

Tidsforbruk - skal minimaliseres (sjeldnere også plassforbruk)

Data Struktur - velges avhengig av hvilke operasjoner programmet skal utføre

Gjenbruk, tilpassing og oversikt : **ABSTRAKSJON**

Abstrakte operasjoner - funksjonalitet (HVA) og ikke implementasjon (HVORDAN)
Programmet burde bygges v.hj.a. høy-nivå (dvs. naturlige og intuitive) begrep
som kan tilpasses forskjellige kontekster -

Strukturering - oppdeling av store programmeringsoppgaver i oversiktlige moduler
- forenkler arbaidet
- tillater gjenbruk av komponenter
- øker pålitelighet og lesbarhet

Dokumentasjon
- muliggjør gjenbruk
- øker lesbarhet

Egentlig ikke en gang programmering. . .

*men **analyse** av passende struktur for og effektivitet av intenderte programmet*
(som, i vårt tilfelle, blir tilfeldigvis skrevet i JAVA)

***Formål**: sentrale begreper om effektiv og strukturert programmering*

***Læremåte**: ikke gjennom utvikling av programmer som bare virker*
*men gjennom **analyse og systematisk design** av programmer*

Tenk, tenk, tenk . . .

design, strukturer, analyser, beskriv, dokumenter . . .

. . . programmer

+ ukentlige øvelser - "obligatoriske"

+ 3 obligatoriske oppgaver

Mer spesifikk

I. Forskjellige algoritmer for samme oppgave

hvordan måler vi algoritmes effektivitet

II. Data strukturer legger føringer på valg av algoritmer

samme oppgave kan utføres mer effektivt ved passende valg av datarepresenatasjon

III. Samme algoritme for forskjellige oppgaver !

ved å forandre enkelte aspekter
kan samme algoritme brukes for andre formål

IV. Grensesnitt + Data Struktur
algoritmer + implementasjon = Data Type

Grensesnitt operasjoner forteller HVA en modul kan gjøre
implementasjon involverer en bestemt Data Struktur - HVORDAN

V. Grensesnitt = "Abstrakt" Data Type

Kun HVA – grensesnitt metoder (funksjonalitet)
Nye moduler benytter seg kun av kunnskap om HVA andre moduler gjør (men ikke HVORDAN)
Konseptuell enkelhet
Mulighet for forskjellige implementasjoner

VI. ... og alt dette med eksempler av

mest vanlige data strukturer
og klassiske algoritmer

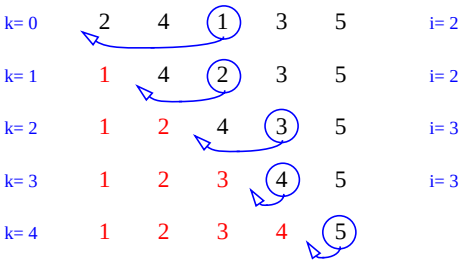
I. Pseudokode

```
public int[] SS(int[] tab) {
    int m, i;
    for (int k=0; k<tab.length;k++) {
        m= tab[k]; ind= k;
        for (int j=k+1; j<tab.length; j++) {
            if (tab[j]<m) { m= tab[j]; i= j; }
        tab[i]= tab[k];
        tab[k]= m;
    }
    return tab;
}

/**  SS - sorterer input array:
 *       @param  - int tab[0...n]
 *       @return - sortert tab
 *   for (k=0,1,2...n-1) {
 *       i = indeksen til minste elementet i tab[k...n]
 *       bytt elementene ved indeks k og i
 *       }
 */
```

Dette er nok for å se at det virker:

Løkkeinvariant: Etter k-te iterasjon er elementene 0...k riktig plassert



Algoritmeanalyse

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

Hvor mange ganger må jeg utføre basis operasjoner: sammenlikning + flytting ?

k= 0	2	4	(1)	3	5	4 + 1
k= 1	**1**	4	(2)	3	5	3 + 1
k= 2	**1**	**2**	4	(3)	5	2 + 1
k= 3	**1**	**2**	**3**	(4)	5	1 + 1
k= 4	**1**	**2**	**3**	**4**	5	0 + 0
						10 + 4 = 14

Generelt:

for en vlikårlig input tabell med lengde n:
utfører n iterasjoner (for k=1,2...n) og
i hver iterasjon går gjennom sluttsegment [k...n], dvs.

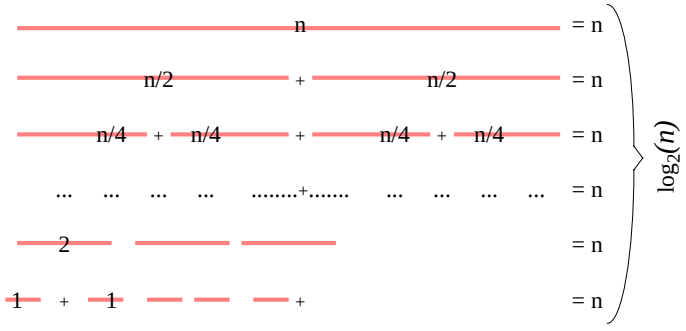
$$SS(n) \;:\; \left(n + \sum_{k=1}^{n} k = 1 + 2 + \ldots + (n-1) + n \right) = (n + n^2)/2 + n = O(n^2)$$

GREEDY


```
/*   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;
 */
```

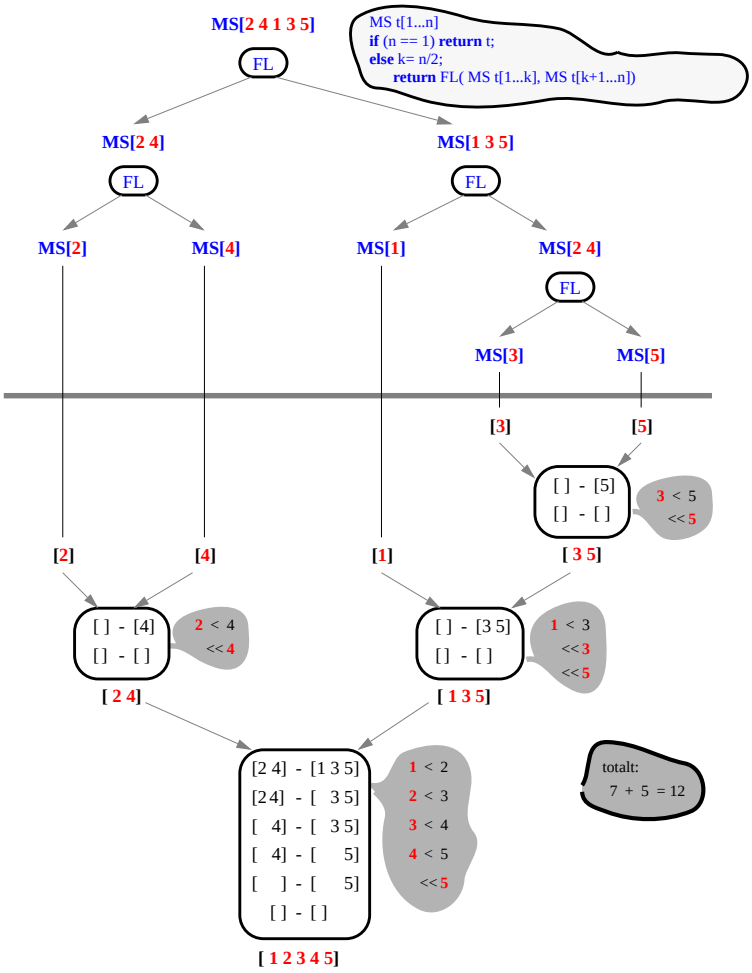
$$\mathrm{FL}(n1,n2) = O(n1+n2)$$

```
/*   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]) ); }
 */
```

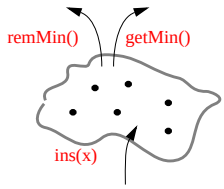


$$\mathrm{MS}(n) = O(n * \log_2(n))$$

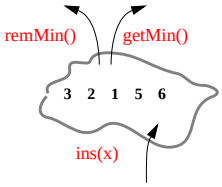
DIVIDE & CONQUER



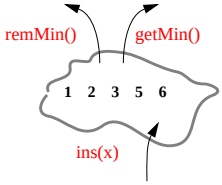
II. Vil lagre data og utføre



1. operasjon getMin()



a) usortert array (-)

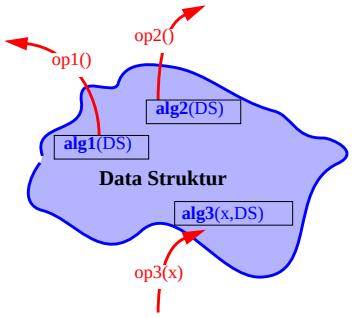


b) sortert array (+)

2... men også operasjon ins(int x)

a) usortert array (+)

b) sortert array (-)



3... og operasjon remMin()

a) usortert array (-)

b) sortert array (-)

III. Bruk, gjenbruk, tilpassing ...

Sort

```
int[] T
for (k = 0,1,2...n)
{     i = k
      for (j = k+1...n)
          if (T[j] < T[i]) i=j;
      bytt T[k] og T[i]  }
```

```
String[] T
for (k = 0,1,2...n)
 { i = k
 for (j = k+1...n)
     if (lt(T[j],T[i])) i=j;
bytt T[k] og T[i] }
```

```
public boolean lt(String s,t)
 { return (s.compareTo(t) < 0);}
```

```
{     i = k
      for (j = k+1...n)
          if (T[j] > T[i]) i=j;
      bytt T[k] og T[i]  }
```

Gjenbruk av kode – moduler for bestemte oppgaver

Tilpassing – samme ting, litt/vidt forskjellige oppgaver

```
Object[]  GS(Object[] T, Comp cp) {
            for (k = 0,1,2...n)
                {  i = k
                for (j = k+1...n)
                    if (cp.lt(T[j],T[i])) i=j;
                bytt T[k] og T[i]  }
            return T; }
```

```
public abstract class Comp {
    public abstract boolean lt(Object e,f) ;
}
```

```
public class LT extends Comp { // Integer obj
  public boolean lt(Object e,f)
    { int e1 = ((Integer) e).intValue();
      int f1 = ((Integer) f).intValue();
      return e1 < f1; }
}
```

```
public class LS extends Comp { // String objekter
  public boolean lt(Object e,f)
    { String e1 = (String) e;
      String f1 = (String) f;
return e1.compareTo(f1) < 0; } }
```

```
public class GT extends Comp {
  public boolean lt(Object e,f)
    { int e1 = ((Integer) e).intValue();
      int f1 = ((Integer) f).intValue();
      return e1 > f1; } }
```


IV. Data Type = metoder + Data Struktur

```
import Comp;
public class Min {

  public Min(int m, Comp c) {
      max= m; tab= new Object[max]; ind= -1; cp= c;  }
  public Object getMin() {
      if (!isEmpty()) { SSort(); return tab[0]; }
      else return null;  }
  public void ins(Object x) {
      if (ind < max-1) {ind= ind+1; tab[ind]= x;}   }
  public void remMin() {
      if (!isEmpty()) {
          SSort(); tab[0]= tab[ind]; ind= ind-1; }  }
  public boolean isEmpty()
      { retunr ind < 0; }

  private Object[] tab ;
  private int ind, max ;
  private Comp cp ;

  private void SSort() {
      int m, i;
      for (int k=0; k<ind; k++) {
          m= tab[k]; i= k;
          for (int j=k+1; j<ind; j++) {
              if (cp.lt(tab[j],tab[i])) { i= j; }
          m= tab[i]; tab[i]= tab[k]; tab[k]= m;
  } }
}
```

Nesten ...

```
public abstract class Comp {
        public abstract boolean lt(Object e, Object f) ;
}
```

```
public class LS extends Comp { // String objekter
   public boolean lt(Object e,f)
     { String e1 = (String) e;
       String f1 = (String) f;
return e1.compareTo(f1) < 0; } }
```

```
public class LT extends Comp {
   public boolean lt(Object e,f)
     { int e1 = ((Integer) e).intValue();
       int f1 = ((Integer) f).intValue();
       return e1 < f1; }
}
```

```
Object[] GS(Object[] T, Comp cp) {
      for (k = 0,1,2...n)
      {   i = k ;
          for (j = k+1...n)
              if (cp.lt(T[j],T[i])) i=j;
          bytt T[k] og T[i]  }
return T; }
```

Algoritmen bryr seg kun om eksistens og type av operasjonen 'lt'
– og antar at den aktuelle parameteren vil ha en slik operasjon.

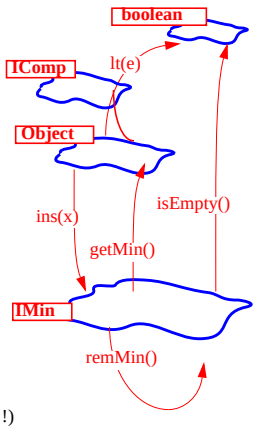
```
public class Dato  {
  int d, m, a;  }
```

```
public class DatoCp extends Comp {
    public boolean lt(Object e,f) {
      Dato e1 = (Dato) e; Dato f1 = (Dato) e2;
      if (e1.a < f1.a) return true;
      else if (e1.a == f1.a && e1.m < f1.m) return true ;
      else if (e1.a == f1.a && e1.m = f1.m && e1.d < f1.d)
          return true;
      else return false; }
}
```


V. "Abstrakt" Data Type = Data Type – Data Struktur = metoder

```
/* sammenligner elementer */
public interface IComp {
/*    @return - true hviss e er mindre enn f*/
      boolean lt(Object e, Object f) ;
}

/* sammling av sammenliknbare elementer */
public interface IMin {
/*    @return - minste element i samlingen */
      Object getMin() ;
/*    @param - nytt element som skal settes inn */
      void ins(Object x) ;
/*    @return - samlingen uten minste elementet */
      void remMin() ;
/*    @return - true hviss samlingen er tom */
      boolean isEmpty() ;
}
```



Grensesnitt vs. **implementasjon** (også av andre DT !)

```
public class MinSS implements IMin {
    public Object getMin() { ... }
    public void ins(Object x) { ... }
    public void remMin() { ... }
    public boolean isEmpty() { ... }
    public MinSS(Comp cp) { ... }
    private Comp cp ;
    private Object[] tab ;
    private int ind, max ;
    private void SSort() { ... }
}
```

```
public class MinMS implements IMin {
    public Object getMin() { ... }
    public void ins(Object x) { ... }
    public void remMin() { ... }
    public boolean isEmpty() { ... }
    public MinMS(Comp cp) { ... }
    private Comp cp ;
    private Object[] tab ;
    private int ind, max ;
    private void MSort() { ... }
    private void FL(Object[] t1,t2) {...}
}
```

Primitive Data Typer: int, boolean, ...

The "Millennium Bug"

```
MinSS store= new MinSS();
store.ins(new prod(...)) ;
....
  public void remOverdue() {
      boolean done= false;
      aa = getYear();
      mm = getMonth();
      dd = getDay();
      prod p= (prod)store.getMin();
      while (! done) {
          if ( p.a < aa || (p.a==aa && p.m < m)
              || (p.a==aa && p.m==m && p.d < dd) )
              { store.remMin();
              p= (prod)store.getMin();
          } else done = true ;
```

```
class prod {
  public int d, m, a;
  public boolean lt(prod p) {
  if ( a < p.a || (a==p.a && m < p.m)
      || (a==p.a && m==p.m && d < p.d) )
      return true;
  else return false; }
```

80% av kostnader går ikke til programutvikling
men til vedlikehold !!!

ABSTRAKT PROGRAMMERING

```
import IDate,IMin,IComp;
IMin store;
IComp dcp= new DatoCp();
store= new MinSS(dcp);
store.ins(new prod(...)) ;
....
  public void remOverdue() {
    boolean done= false;
    IDate today = getDate();
    prod p= (prod)store.getMin();
    while (! done) {
      if (dcp.lt(p.dt,today)) {
        store.remMin();
        p= (prod)store.getMin();
      } else done = true;
```

```
class prod {
    public IDate dt;
 ... }
```

```
public interface IDate {
    int d(); int m();
    int a(); setA(int aa);
```

```
public class DatoCp implements IComp {
// sammlikner IDate-objekter
  public boolean lt(Object d1,d2) {
    Dato e= (Dato)d1; Dato f= (Dato)d2;
    if (e.a() < 100) e.setA(1900+e.a());
    if (f.a() < 100) f.setA(1900+f.a());
    if (e.a<f.a || (e.a==f.a && e.m<f.m)
      ||(e.a==f.a && e.m=f.m && e.d<f.d) )
      return true;
    else return false;
 ... }
```


Programutvikling

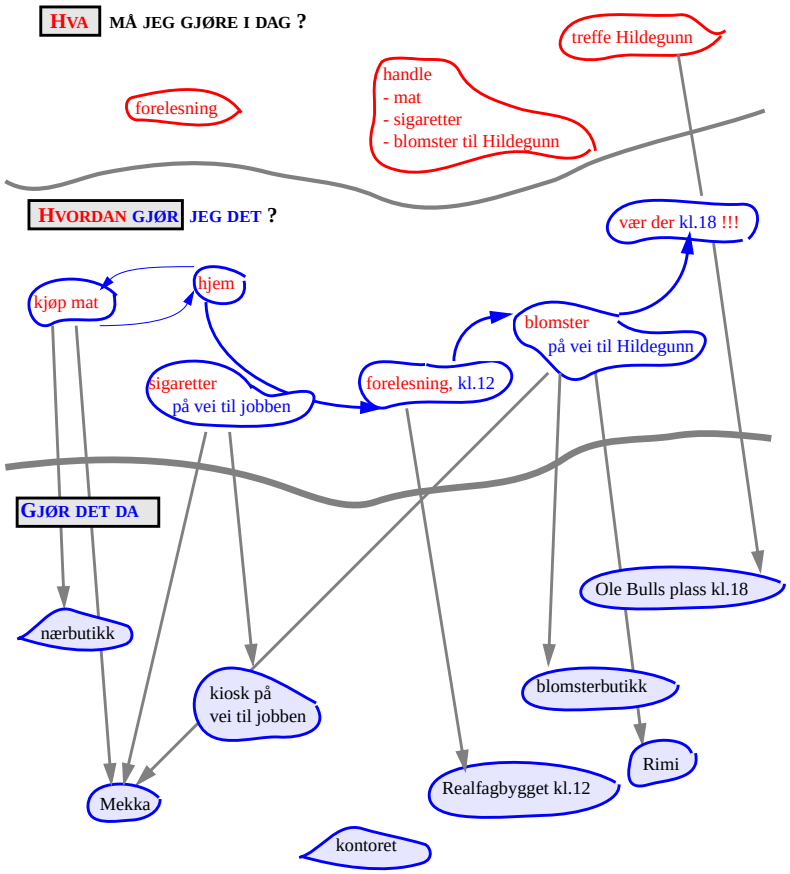
1. Finn ut hvilke moduler du trenger :
   - hvordan de skal samarbeide og
   - gjennom hvilke grensesnitt
     - **– noen vil være tilgjengelig fra eksisterende bibliotek**
     - **– andre vil du måtte lage selv**

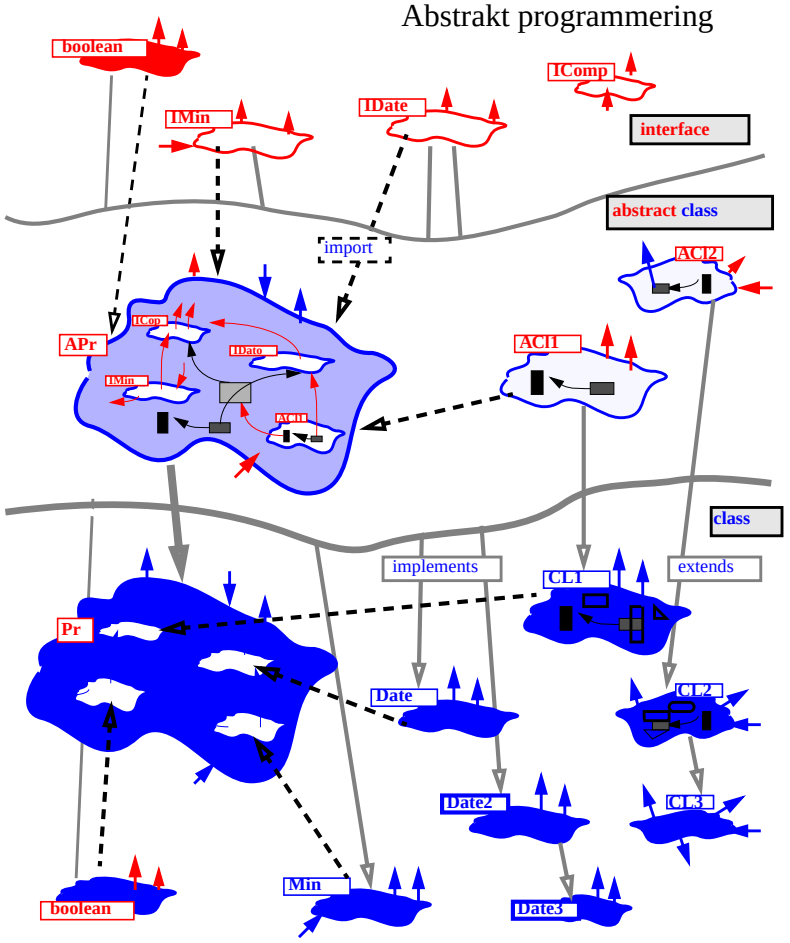
For hver enkel modul

2. velg en data struktur og
3. design nødvendige algoritmer

(Her skal du tenke effektivitet)

4. Implementer alle moduler og sett de sammen iflg. 1.


HVA MÅ JEG GJØRE I DAG ?
treffe Hildegunn
handle
- mat
- sigaretter
- blomster til Hildegunn
forelesning
HVORDAN GJØR JEG DET ?
vær der kl.18 !!!
kjøp mat
hjem
blomster
på vei til Hildegunn
sigaretter
på vei til jobben
forelesning, kl.12
GJØR DET DA
Ole Bulls plass kl.18
nærbutikk
kiosk på
vei til jobben
blomsterbutikk
Rimi
Mekka
Realfagbygget kl.12
kontoret


boolean
IComp
IMin
IDate
interface
abstract class
import
ACI2
APr
ICop
IDate
IMin
ACI1
ACI1
class
implements
CL1
extends
Pr
Date
CL2
Date2
CL3
boolean
Min
Date3

Utdeling av s-konto

i dag (25.8):	14–15
i morgen (26.8):	14–15

Institutt for Informatikk
HiB (Høyteknologisenteret), Datablokken, 4 etg.

overingeniør Ole Arntzen, rom nr. 4149

gruppepåmelding

møtt opp på første gruppe