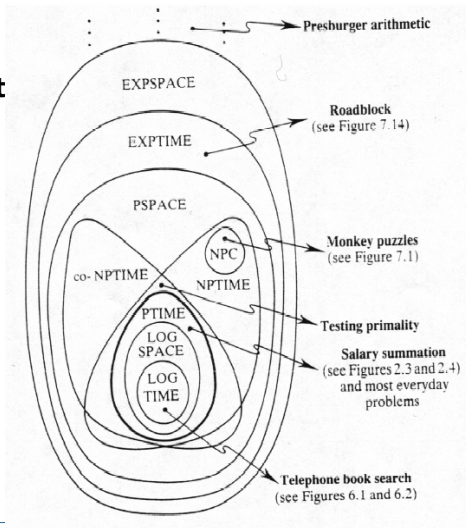
Uke 11,
Forelesning 1

Vi diskuterte: NP-komplethet



Class NPC

Among all the problems known to be in NP, there is a subset known as **NP-complete** problems, which contains the hardest problems in NP (intractable, with polynomial certificates). These have also one more property that is extreemly interesting: they all have a common fate: i.e. there exist **a polynomial time reduction** from any one problem in NPC to any other problem in NPC. Reduction can be quite simple, or it can actualy involve several intermediate reductions.

Innledning i sortering

Hva sorterer vi i praksis
- Bare tall eller tekster – eller noe mer

Hvordan definere problemet
- Krav som må være oppfylt

Empirisk testing av hastigheten til algoritmer
- Hvilke verdier (fordeling, max og min verdi – gitt antallet)

Kaller arrayen a[] før sorteringen og a'[ ] etter
- og n er lengden:dvs. a = new int [n];.

Sorteringskravet:
- a'[i] ≤ a'[i+1], i = 0,1,..,n-2

Stabil sortering
- Like elementer skal beholde sin innbyrdes rekkefølge etter sortering.
- Dvs. hvis a[i] = a[j] og i < j, så skal k < r, hvis a[i] er sortert inn på plass 'k' i a'[] og a[j] sortert inn på plass 'r' i a'[]

Sorteringsalgoritmene antar at det kan finnes like verdier i a[ ]
- I bevisene antar vi alle ai er forskjellige: a[i] ≠ a[j] , når i ≠ j.

I testkjøringene antar vi at innholdet i a[ ] er er en tilfeldig permutasjon av tallene 1...n.

Hvor mye ekstra plass bruker algoritmen?
- Et lite fast antall, et begrenset antall (eks. < 1012) heltall, eller n ekstra ord

•Bevaringskriteriet:
- •alle elementene vi hadde i a[ ], skal være i a'[ ]
- •Formelt : Skal eksistere en permutasjon, p, av tallene 0..n-1 slik at a'[p[i]] = a[i] , i = 0,1,..n-1
- •Vi skal senere se en sorteringsalgoritme basert på en slik array p

	0	1	2	3	4	5	6	7	8
a[ ] :	4	7	2	1	5	9	5	8	6

	0	1	2	3	4	5	6	7	8
a'[ ]:	1	2	4	5	5	6	7	8	9

	0	1	2	3	4	5	6	7	8
p[ ] :	2	6	1	0	3	8	4	7	5

En litt enklere kode enn boka,
antar vi sorterer heltall.

- Lar seg lett generalisere til bokas tilfelle, som antar at den sorterer en array av objekter som er av typen Comparable

BOKa:

```
Comparable [ ] a  = new Comparable [n];

tmp = a[i];

if ( tmp.compareTo( a[j])   < 0 ) {
.....
}
```

HER:

```
int [ ] a = new int [n];

tmp = a[i],

if ( tmp < a[j] ) {
.....
}
```

•Verdi-baserte :
Direkte plassering basert på verdien av hvert element – ingen sammenligninger med nabo-elementer e.l.
- •Radix
- •PSort
- •Bøtte

•Sammenligning-baserte:
Baserer seg på sammenligning av elemntene i a[ ]
- •Instikk, boble
- •Merge, Heap, Shell, Tree
- •Quicksort

tids-forbruk, millisek. – 450MHz PC

Lengde av a: 10
Boble-sort = 0,061
Innstikk-sort = 0,037
Heap - sort = 0,066
Shell-sort = 0,065
Tree - sort = 0,072

Lengde av a: 100
Boble-sort = 0,610
Innstikk-sort = 0,220
Heap - sort = 0,270
Shell-sort = 0,170
Tree - sort = 0,160

Lengde av a: 1 000
Boble-sort = 46,100
Innstikk-sort = 10,900
Heap - sort = 1,700
Shell-sort = 1,700
Tree - sort = 1,600

Lengde av a: 10 000
Boble-sort = 4735,000
Innstikk-sort = 1230,000
Heap - sort = 28,000
Shell-sort = 22,000
Tree - sort = 17,000

- La oss se på animasjoner av noen av de sorteringsalgoritmene:

  http://cs.smith.edu/~thiebaut/java/sort/demo.html
  http://www.aeriesoft.ru/Projects/SortAlg/


```
void bytt(int[] a, int i, int j)
  { int t = a[i];
    a[i]=a[j];
    a[j] = t;
  }

  void bobleSort (int [] a)
  {int i = 0, max = a.length;

   while ( i < max )
         if (a[i] > a[i+1])
         {    bytt (a, i, i+1);
              if (i > 0 )  i = i-1;
          } else {
              i = i + 1;
           }
  } // end bobleSort
```

***Ide**: Bytt om naboer hvis den som står til venstre er størst, lar den minste boble venstreover*

	0	1	2	3	4	5	6	7	8
a[ ] :	4	7	2	1	5	9	5	8	6

	0	1	2	3	4	5	6	7	8
a[ ] :									

	0	1	2	3	4	5	6	7	8
a[ ] :									

	0	1	2	3	4	5	6	7	8
a[ ] :									

En inversjon ('feil') er per def: a[i] > a[j], men i < j.

Th 7.1

Det er gjennomsnittlig n(n-1)/4 inversjoner i en array av lengde n.

Bevis

Se på en liste L og den samme listen reversert Lr. Ser vi på to vilkårlige elementer x,y i begge disse listene. I en av liste står de opplagt i gal rekkefølge (hvis x ≠y). Det er n(n-1)/2 slike par i de to listene, og i snitt står halvparten 'feil' sortert, dvs. n(n-1)/4 inversjoner i L.

- Boble er opplagt $O(n^2)$ fordi:
  - Th. 7.1 sier at det er $O(n^2)$ inversjoner, og en naboombytting fjerner bare en slik inversjon.
- Kunne også argumentert som flg.:
  - Vi går gjennom hele arrayen, og for hver som er i gal rekkefølge (halvparten i snitt) – bytter vi disse (i snitt) halve arrayen ned mot begynnelsen.
  - $n/2 \times n/2 = n^2/4 = O(n^2)$, men mange operasjoner ved å boble (nabo-ombyttinger)

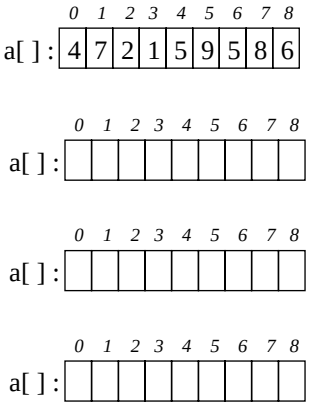
```
void insertSort(int [] a )
 {int i, t, max = a.length -1;

  for (int k = 0 ; k < max; k++)
  if (a[k] > a[k+1]) {
            t = a[k+1];
            i = k;

          do{ //  gå bakover, skyv de andre
              //  og finn riktig plass for 't'
               a[i+1] = a[i];
                i--;
           } while (i >= 0  &&  a[i] > t);

           a[i+1] = t;
        }
} // end insertSort
```

***Ide**: Ta ut ut element a[k+1] som er mindre enn a[k]. Skyv elementer k, k-1,... ett hakk til høyre til a[k+1] kan settes ned foran et mindre element.*



Spesifikasjon_: ***void*** Sort (a,n);
//a'[..] er utdata, *a[..]* er inndata -verdiene

Inndata: n tall a[0..n-1] i vilkårlig rekkefølge
Utdata: $a'[i-1] \leq a'[i],\ 0 < i < n$ **og**
$\exists$ permutasjon n P: $\forall i\ 0 \leq i < n$: $a'[P[i]] = a[i]$

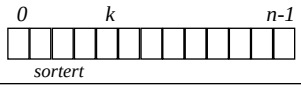
← ***Sortert-kravet***
← ***Bevar innholdet av a[0..n-1]***

Bevaringskravet leses:
Det finnes en rekkefølge P av tallene 0..n-1, slik at ved å lese utdata *a'[..]*i den rekkefølgen, er den lik inndata *a[..]*

Generell designmetodikk

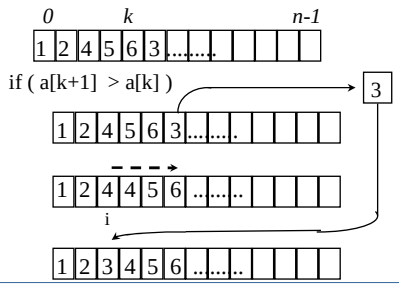
- Gitt en klar spesifikasjon med flere ledd
- En av delene i spesifikasjonen nyttes som invariant i programmets (ytterste) hovedløkke i en litt endret form.
  (løkke-invariant = noe som er sant i begynnelsen av løkka)
- Dette kravet svekkes litt (gjøres litt enklere);
  gjelder da typisk bare for en del av datastukturen
- I denne hoved-løkka, gjelder så resten av spesifikasjonene:
  - i begynnelsen av hoved-løkka
  - ødelegges ofte i løpet av løkka
  - gjenskapes før avslutning av løkka

Design, innstikksortering



Svekker Sortert-kravet til bare å gjelde a[0..i-1]
Bevaringskravet beholdes (for hele a[0..n-1])

Innstikk-sortering (for k = 0, 1,2,...,n-1:):



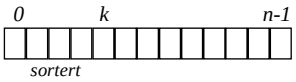
1) Ta ut det 'galt plasserte' elementet a[i]

2) Finn *i* hvor 'gamle a[k+1]' skal plasseres og skyv a[i..k] ett-hakk-til- høyre
   (*ødelegger Bevaringskravet)*

3) Sett 'gamle a[k+1]' inn på plass ***i***
   (*gjenskaper Bevaringskravet*)

```
void  insertSort ( int [ ]a )
   {  int i,  t, max = a.length –1;

1      for(int k=0; k < max; k++)
2      if (a[k] > a[k+1] )
3      {     t = a[k+1];
4            i = k;
5            do
6              { a[i+1] = a[i]; i - -;
7              } while ( i >=0 && a[i] > t)
8            a[i+1] = t;
9       }
10    }
```



Her gjelder Sortert: a[0.. k]
og Bevart a[0..n-1]

1) Ta ut det 'galt plasserte elementet a[k+1]

3) Sett 'gamle a[k+1]' inn på plass *i*
(*gjenskaper Bevaringskravet*)

Verifiseres iflg. spesifikasjonene + resonnement for *terminering* av prosedyren

Analyse av innstikk sortering

Samme analyse som Boble, men langt færre operasjoner per forflytning – O(n2)

ALMIRA KARABEG foreleser!

Vi fortsetter med sortering (kapittel 7):

- Shell sort
- Heap sort
- Quick sort
- Merge sort