Uke 2,

Forelesning 1

- Introduksjon til
  METODEKALL
  og REKURSJON

- Introduksjon til
  KOMBINATORISKE SØK
  og KOMBINASJONER, PERMUTASJONER

- Introduksjon til
  DRONNING OPPGAVEN

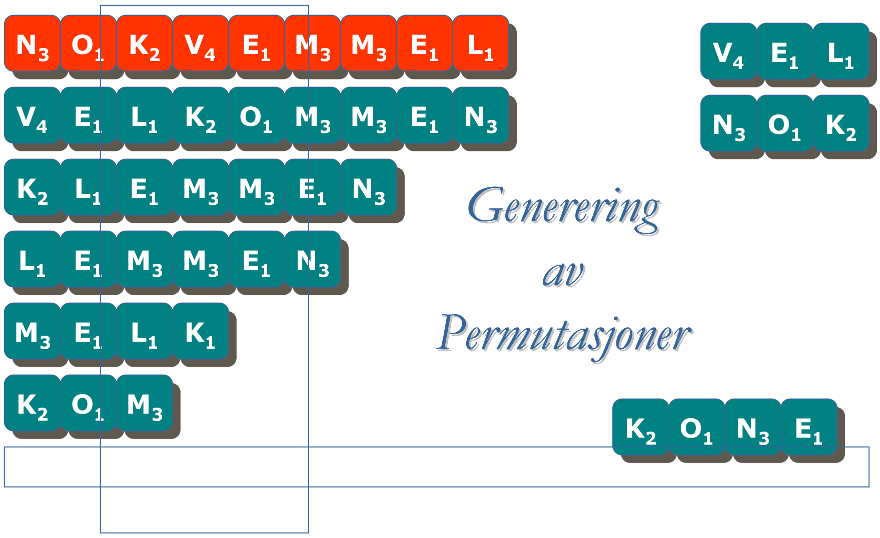
OVERSIKT – *Uke 1, Forelesning 2 (W1.L2)*

- TEMA #1:
  Hvordan en kan GENERERE PERMUTASJONER

- TEMA #2:
  Introduksjon til ANALYSE av ALGORITMER

- TEMA #3:
  Litt til om DRONNING OPPGAVEN

TEMA #1



Generering av Permutasjoner

PERMUTASJONER – *Generering av Permutasjoner #1*

- La oss se på én måte ved hjelp av et eksempel for permutasjoner av mengden {0, 1, 2, 3} for N = 4...

0	-	-	-	1	-	-	-	2	-	-	-	3	-	-	-
	1	-	-		0	-	-		0	-	-		0	-	-
		2	3			2	3			1	3			1	2
		3	2			3	2			3	1			2	1
	2	-	-		2	-	-		1	-	-		1	-	-
		1	3			0	3			0	3			0	2
		3	1			3	0			3	0			2	0
	3	-	-		3	-	-		3	-	-		2	-	-
		1	2			0	2			0	1			0	1
		2	1			2	0			1	0			1	0

Hold først, bytt siste, bytt med neste verdi deretter utover helt til første.

PERMUTASJONER – *Generering av Permutasjoner #2*

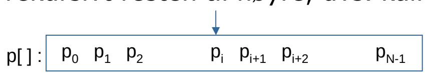
- La oss se på en mulig algoritme som tillater rekursjon:

i = siste plass i originale rekken;	0 1 2 3 ← i
Bytt siste med nest siste (i og i-1);	0 1 3 2
Bytt (i-2)^{te} og i^{te} plass;	0 2 3 1
Bytt siste med nest siste (i og i-1);	0 2 1 3
Bytt (i-2)^{te} og i^{te} plass;	0 3 1 2
Bytt siste med nest siste (i og i-1);	0 3 2 1
Bytt (i-3)^{te} og i^{te} plass;	1 3 2 0
Bytt siste med nest siste (i og i-1);	1 3 0 2
...	

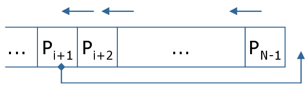
0 1 2 3	1 3 2 0	3 0 2 1	0 1 2 3
0 1 3 2	1 3 0 2	3 0 1 2	...
0 2 3 1	1 2 0 3	3 2 1 0	...
0 2 1 3	1 2 3 0	3 2 0 1	...
0 3 1 2	1 0 3 2	3 1 0 2	...
0 3 2 1	1 0 2 3	3 1 2 0	Som første kolonne!!!

PERMUTASJONER – *Generering av Permutasjoner #3*

- Hvordan generere alle permutasjoner (ombyttinger eller rekkefølger) av 0, 1, 2, ..., (N-1)?
  - Initier en array p[ ] med plass for N (indeks = 0, 1, …, N-1)
  - Et kall på **permuter(i)** skal lage alle permutasjoner av tallene p[i], ..., p[N-1] – og ved retur skal p[] være som ved kallet!
  - Anta at vi står på plass nr. i (der i < N-1) :
    - Gjennomfør (bruk) ferdig permutasjonen for i.
    - Bytt så ut nåværende element nr. i etter tur med element k = i+1, i+2, …, N-1, og for hver slik ombytting, permuter rekursivt resten til høyre, dvs. kall **permuter (k+1)**.



    - Reetabler p[]: Skift-roter alle elementene i+1, …, N-1 syklisk ett hakk til venstre (SE OGSÅ NESTE SIDE).



KODE – *for Generering av Permutasjoner*

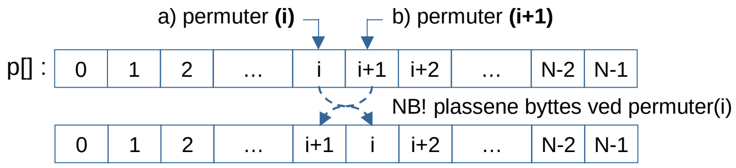
```
class Perm
{  int [] p ;
   int n;

   Perm(int num)
   {  // Konstruktor: initier p og
      // start permutasjonen
      n = num;
      p = new int[n];
      for (int i = 0; i < n ; i++)
         p[i] = i;
   }
   void roterVenstre(int i)
   {  // syklisk roter p[i..n-1] en
      //plass til venstre
      int x,k;
      x = p[i];
      for (k= i+1; k < n; k++)
         p[k-1] = p[k];
      p[n-1] = x;
   }
   void bytt(int i, int j)
   {  // bytt om p[i] og p[j]
      int t = p[i];
      p[i]=p[j];
      p[j] = t;
   }

    final void permuter (int i)
    {  // Finn neste permutasjon og
       // kall "brukPerm().
       // N.B. Permutasjonene startes
       // ved kallet: permuter(0);
     if (i == n-1)
       brukPerm();
     else
     { permuter(i+1);
       for (int t = i+1; t < n; t++)
       {   bytt (i,t);
        permuter(i+1);
       }
       roterVenstre(i);
     }
    }

    void brukPerm ()
    {  // Standard bruk ikke definert.
       // Byttes ut i subklasse.
       // NB! Skriv ut permutasjonen
       // hvis n < 5! Det kan fort bli
       // alt for mye å skrive!
    }
}
```

PERMUTASJONER – *Forklaring av Programmets Virkemåte*

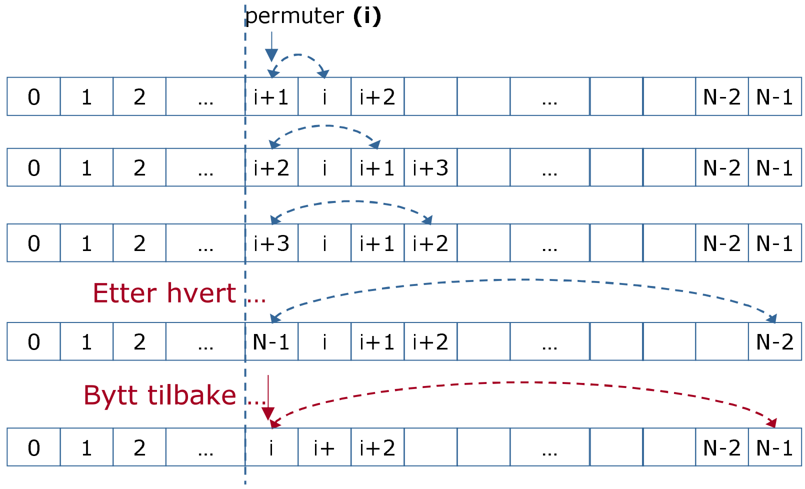


permuter (i):

Permuterer først resten, dvs. alt på plass i+1 og høyreover…

- Skifter så ut (etter tur) innholdet av det i'te elementet med det som står på plassene i+1, i+2, …, N-1.
- For hver slik bytting, permuterer resten av array'en høyreover – dvs. alt på plass i+1 og høyreover.
- NB! N-1'te kommer foran (til venstre for) i'te element, til slutt, som gjør at skift-rotering til venstre blir nødvendig (se neste side).

PERMUTASJONER – *Etter Ombyttinger…*



NB! NB! Hvordan bytte? Forsiktig med "bytt" algoritmer…

KODE #1 – *Implementasjon av brukPerm i Perm-subklasser*

```
public class PermProg
{  // start program: >java PermProg 'n'
   public static void main (String [] args)
   {  if (args.length < 2)
         System.out.println(" Riktig bruk: >java PermProg {A|B} n");
      else if (args[0].equals("A") )
         new APerm (Integer.parseInt(args[1])).permuter (0);
      else  if (args[0].equals("B") )
         new BPerm (Integer.parseInt(args[1])).permuter (0);
      else System.out.println ("Gal første parameter");
   }
}


class APerm extends Perm
{  APerm(int n) { super(n); }

   void brukPerm()
   {  // skriv ut permutasjonene
      for (int i = 0; i < n; i++)
         System.out.print  (p[i]);
      System.out.println();
   }
}
```

// FORTESSETTER NESTE SIDE…

KODE #2 – *Implementasjon av brukPerm i Perm-subklasser*

```
class BPerm extends Perm
{ int t = 1;
  int nfak;

  int fak (int i)
  {if (i == 1 ) return 1; else return i * fak(i-1);}

  BPerm(int n) { super(n); nfak = fak(n);}

  void brukPerm()
  { // skriv ut en teller hver for hver 100 000 permutasjon og til sist
    if ( (t % 100000)== 0 )
       System.out.print  ("\r Permutasjon: " + t);
    if (t == nfak )
       System.out.println  ("\r Permutasjon: " + t);
    t++;
  }
}
```

KODE – *Tolking*

```
final void permuter (int i)
{  // Finn neste permutasjon og
      // kall "brukPerm().
      // N.B. Permutasjonene startes
      // ved kallet: permuter(0);
   if (i == n-1)
      brukPerm();
   else
   {  permuter(i+1);
      for (int t = i+1; t < n; t++)
      {  bytt (i,t);
         permuter(i+1);
      }
      roterVenstre(i);
   }
}
```

```
permuter(0);                        ... i = 0
0 != 3
else
{ permuter(1)                       ... i = 1
  1 != 3
  { permuter(2)                     ... i = 2
    2 != 3
    { permuter(3)                   ... i = 3
      3 == 3                        bruk 0 1 2 3
    }
    for(t=i+1=2+1=3; t<4; t++)
    { bytt(2,3)                     ...    0 1 3 2
      permuter(3)
      3 == 3                        bruk 0 1 3 2
    }
    roterVenstre(2)                 ...    0 1 2 3
  }
  for(t=i+1=1+1=2; t<4; t++)
  { bytt(1,2)                       ...    0 2 1 3
    permuter(2)
    2 != 3
      { permuter(3)
        3 == 3                      bruk 0 2 1 3
      }
    for(t=3; ...)
    bytt(2,3)                       ...    0 2 3 1
    osv.
```

Vi prøver koden med n = 4.
permuter kalles med i = 0 først.

TEMA #2

Analyse
av
Algoritmer

- Vi vil finne uttrykk for hvordan kjøretiden øker med **n**, der **n** = Datamengden inn til algoritmen eller mål for størrelsen av problemet – f. eks. "antall byer som skal besøkes" eller "antall database records som skal sorteres"...
- Hva måler vi?
  - Gjennomsnittlig tidsforbruk
  - 'Verste tilfelle' tidsforbruk
- Alternativer :
  - Ta tiden (se programmet Tatid.java) for ulike verdier av N
  - Telle antall elementære operasjoner
  - Finne enkel funksjon f(n) som vokser 'på samme måte' som eksekveringstiden til programmet
- Det som nytter mest er alternativ 3 og 'verste tilfelle' analyse

La T(n) være programmets kjøretid...

- T(n) = O(f(n)) hvis det finnes positive konstanter c og n_0 slik at T(n) $\leq c \bullet$f(n) når n $\geq n_0$
  O (leses stor-O og forståes som "i størrelsesorden") er en øvre grense for kjøretiden

- T(n) = Ω(f(n)) hvis det finnes positive konstanter c og n_0 slik at T(n) $\geq c \bullet$f(n) når n $\geq n_0$
  Ω (leses omega og forståes også som "størrelsesorden") er en nedre grense for kjøretiden

- T(n) = Θ(f(n)) hvis og bare hvis
  T(n) **=** O(f(n)) **og** T(n) = Ω(f(n))

- O(f(n)) er overlegent mest brukt.
- Problemet er å finne f(n) 'som er minst mulig', dvs. nærmere reelle kjøretiden ovenfra.
- Vi forkorter(forenkler) funksjonen – og tar med bare det leddet (uten konstant foran) som vil dominere når 'n' blir stor - eks:
- **Enkel for-løkke:**
  T(n) er O(n), dvs. $T(n) = c_1n+c_2$, og f(n) = n
- **Dobbelt for-løkke:**
  T(n) er O(n^2), dvs. $T(n) = c_1n^2+ c_2n+c_3$ og f(n) = n^2

f(n)	**Navn**
1	Konstant
log n	Logaritmisk
n	Lineær
n log n	?
n^2	Kvadratisk
n^3	Kubisk
2^n , n!	Eksponensiell

Polynomisk tid

Raskere voksende

n! = 1 x 2 x 3 x … x n
Vokser meget raskt!

Algoritme 1: **sumNFørsteTall1(n);**
Input: n
Output: The sum of 1 + 2 + … + n
return n(n+1)/2

Algoritme 2: **sumNFørsteTall2(n);**
Input: n
Output: The sum of 1+2+..n
sum = 1
for i = 2 to n do
sum = sum + i
return sum

a) Algoritme 1
T(n) er O(1), dvs. $T(n) = c_1$, og $f(n) = 1$

b) Algoritme 2
T(n) er O(n), dvs. $T(n) = c_2n+c_3$, og $f(n) = n$


```
P1: int [] a = new int [n];
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n ; j++)
            a[k] = a[j] +1;

P2: int [] a = new int [n];
    for (int k = 0; k < n; k++)
        for (int j = 0; j < k ; j++)
            a[k] = a[j] +1;
```

n	Tid P1	Tid P2
1000	31	15
5000	672	328
10000	2735	1375

P1 har kjøretid: $n + n + n + \ldots + n = n \times n = n^2$
P2 har kjøretid: $1 + 2 + 3 + \ldots + n = n \times (n-1)/2 = n^2/2 - n/2$

N.B. Allikevel har begge O(n²) kjøretid!

KJØRETID – *Beregningsmodell, Antagelser*

- Operasjoner utføres sekvensielt
- Basisoperasjoner tar lik tid (+ * < = / - ...)
- Ingen spesialoperasjoner (som for eksempel matrisemultiplikasjon)
- Nok internlager

- Ikke helt slik for en ekte datamaskin ☹
- Operasjoner på sekundærlager (eksternlager) o.l. ikke tas til hensyn ☹
- "Page Fault" og annet (avbrudd o.l.) ikke tas til hensyn ☹

- MEN de er jo beregninger/estimater for å kunne sammenligne algoritmene, ikke målinger... ☺

NESTE GANG

- Vi fortsetter med litt mer algoritmeanalyse

- Vi avrunder introduksjonstemaene
  - matte,
  - rekursjon/permutasjoner,
  - algoritmeanalyse

- Vi kikker litt nærmere på og avrunder 1ste oblig (dronning oppgaven)