.

Ifi, UiO: INF110 - notat
Stein Krogdahl og Arne Maus:

Kombinatorisk søking, rekursjon, avskjæring

Vi skal her se på et felt som alltid har hatt et visst innpass i INF110 pensum, men ofte litt spredt og under litt uklare overskrifter. Vi tar det nå tidlig i pensum, for å kunne komme i gang med stoff som fører til litt spennende programmering, som er nokså intuitivt forståelig Det gir også en naturlig grunn til å innføre rekursiv programmering, noe som man erfaringsmessig trenger en viss modningstid for å få tak på. Dessuten setter det oss i stand til å komme i gang med en obligatorisk oppgave ganske tidlig.

Feltet er her kalt "kombinatorisk søking", og problemstillingen må på en eller annen måte være å finne en rekkefølge, gjøre et utplukk, lage en oppdeling eller liknende, ut fra et endelig antall elementer. Vi skal generelt snakke om å finne en "konstellasjon", og de konstellasjoner vi er ute etter vil typisk være karakterisert ved at de tilfredstiller et mer eller mindre krunglete krav. Noen eksempler på slike problemstillinger kan være følgende:

1. Anta at vi har en skoleklasse, og vet hvilke (par) av elevene som er uvenner. Spørsmål: Finn en måte å stille elevene på rekke slik at ingen som er uvenner blir stående ved siden av hverandre.

2. Anta at en selger må innom et antall byer, og at vi kjenner kjøreavstanden mellom ethvert par av byer. Finn den rekkefølgen av byer som gir minst total kjørelengde.

3. Plasser 8 dronninger på et sjakkbrett, slik at ingen av dronningene kan slå hverandre.

4. Finn en måte å farge landene på et kart slik at ingen naboland får samme fargen, når man bare har et begrenset antall farger (Det klarer seg alltid med 4 farger).

5. En kunstforening får en gave på like mange bilder som foreningen har medlemmer, og hvert medlem setter opp en ønskeliste med et antall bilder de gjerne vil ha. Finn, om mulig, en utdeling av bilder slik at alle får et bilde de vil ha.

Vi skal altså her finne en eller annen konstellasjon (oppstilling, ordning, rekkefølge) av et endelig antall elementer. Oppgaven kan ha forskjellige form, så som f.eks. at vi bare skal finne *en eller annen* konstellasjon som tilfredstiller kravet, man skal finne *alle* konstellasjoner som tilfredstiller kravet, eller man skal finne *den beste* slike konstellasjonen i en eller annen forstand.

Vi skal her se på en måte å løse denne type oppgave på, som essensielt er som følger: Vi deler det kravet vi har satt opp i to deler, en **enkel del** og en **vrien del**. Den enkle delen må være slik at det er rimelig greit å lage et program som systematisk genererer *alle* konstellasjoner som tilfredstiller dette kravet. Vi lager så et slikt program, og lager i tillegg en programbit som tester om en gitt konstellasjon generert av dette programmet, også tilfredstiller den vriene delen av kravet. Det er så til slutt lett å sette sammen disse bitene til et program som finner en eller annen, alle, eller den beste konstellasjonen, slik vi nevnte over.

Om vi f.eks. skal lage et program som skriver ut alle konstellasjoner som tilfredstiller det fulle kravet vårt, så kunne et første forsøk på å lage et slikt program være som følger:

```
<Generer den første konstellesjon som tilfredstiller det enkele kravet>;
IF <konstellasjon tilfredst. vrien del> THEN <skriv ut konstellasjon>;


do {
   <Generer neste konstellasjon som tilfredstiller det enkle kravet>;
   if ( <konstellasjon tilfredst. vrien del>) <skriv ut konstellasjon>;
} while(  <den konstellasjonen vi har ikke er siste> );
```

Som vi straks skal se vil de programmene vi skal skrive ofte ikke få denne enkle formen, da genereringen av konstellasjonene vanligvis foregår i et system av rekursive metoder. Mer om dette om et øyeblikk.

Effektivitet

Vi skal her umiddelbart ile til med følgende bemerkning: Den måten å løse problemene på som er skissert over (og som vi skal se på i det følgende) vil vanligvis føre til uhyre tidkrevende programmer, som gjerne er slik at eksekveringstiden øker f.eks. til det dobbelte hver gang problemstørrelsen øker med én, altså at eksekveringstiden blir $O(2^n)$. Typisk vil man da bare kunne ha praktisk nytte av programmet for uhyre små versjoner av problemet, f.eks. for skoleklasser med opp til 10 elever i det første eksempelet over.

For en del slike problemer (f.eks. oppgave 2 over) kjenner vi ikke noen bedre måte å løse problemet på enn denne metoden, som essensielt består i å gå gjennom alle mulige konstellasjoner og plukke ut de som er "brukbare" (på engelsk gjerne kalt "exhaustive search"). Som vi skal se på i det følgende kan denne metoden imidlertid programmeres mer eller mindre raffinert, og dette kan bety mye for effektiviteten, selv om tiden algoritmen bruker fremdeles oftest vil forbli "ekponensiell", altså $O(a^n)$ for en passelig konstant a. En viktig klasse problemer som vi ikke kjenner noen bedre løsningsmetode for er de såkalte *NP-komplette problemer*, som vi kortfattet skal behandle i et eget noat.

For andre slike problemer kjenner vi imidlertid mer effektive metoder enn det å gå gjennom alle mulige løsninger. Disse vil i en passelig forstand bygge opp en løsning etter en målrettet metode, og dette gjelder f.eks. oppgave 5 over, som kan løses etter en meget effektiv metode.

Vi skal imidlertid i denne delen ikke tenke så mye på om mer effektive metoder finnes, men i stedet se på de programmeringsmessige problemene omkring det å generere alle konstellasjonene ut fra et passelig sett med *enkle krav*. Her skal vi se at en programmeringsteknikk som kalles *rekursjon* får en meget naturlig anvendelse.

De metodene vi her skal se på kalles i engelsk litteratur ofte "backtracking algorithms", og i læreboka står det lite eller intet om slike metoder.

Et annet begrep som ofte brukes i forbindelse med slike metoder er "branch-and-bound". Dette er betegnelsen på en spesiell tenkemåte for å gjøre lur avskjæring (se under), når man er ute etter å finne den "beste" konstellasjonen i en eller annen betydning. Den er ikke omtalt i læreboka , men er behandlet i mange andre liknende bøker.

Avskjæring

I det følgende skal vi se på hvordan man kan være mer eller mindre raffinerte under programmeringen, og raffinementet vil da stort sett gå ut på at man ikke venter til hver gang man har generert en ferdig konstellasjon (etter det enkle kravet) med å teste om også den mer kompliserte delen av kravet er tilfredstilt. I stedet forsøker vi å filtre denne testen inn i selve genereringen av konstellasjonene, slik at vi unngår å generere en masse

konstellasjoner som likevel umulig kunne føre til noen fullstendig løsning. Dette kalles gjerne *avskjæring* (engelsk: "pruning"), i og med at vi altså avskjærer videre genering av konstellasjoner som vi innser bare er fåfengt arbeid.

Som et eksempel kan vi se på den første oppgaven over. Vi skal altså finne en rekkefølge av elevene, slik at to som er uvenner ikke kommer ved siden av hverandre. (Hvor vanskelig dette er, er selvfølgelig avhengig av hvor mye uvennskap det er i klassen, og er det riktig ille kan det godt hende at ingen slik rekkefølge finnes). Vi kan her velge den enkle delen av kravet rett og slett til å være at vi skal ha en rekkefølge av elevene, og vi gjør altså dette valget i håp om at det er forholdsvis greit å lage et program som genererer alle rekkefølger av et gitt antall elever. Den vriene delen av kravet blir da at ingen uvenner står ved siden av hverandre, og dette ser vi at det er lett å sjekke for en gitt rekkefølge, dersom vi har en passelig tabell over hvem som er uvenner.

Som et eksempel på hvordan vi kan lage en effektiv avskjæring, kan vi tenke oss at de forskjellige rekkefølgene blir generert slik at vi over en periode holder de til venstre for et visst punkt fast, mens vi til høyre for dette genererer alle mulige kombinasjoner av de resterende elevene. Over en periode under genereringen står da kanskje Anne og Britt helt til venstre i rekkefølgen, og om Anne og Britt er uvenner er det da tydeligvis bare tøv å generere alle mulige rekkefølger ut fra dette, da ingen av disse tydeligvis vil tilfredstille den vriene delen av testen. Vi skal siden se at det ofte er ganske lett å programmere slik at man ikke bruker tid på denslags tøv.

Fullstendig avskjæring

Når vi legger våre hoder i bløt og legger inn kraftigst mulig avskjæring ut fra den vriene delen av kravet, så får vi ofte det vi kunne kalle *fullstendig avskjæring*. Med det mener vi at hver gang vi kommer til en ny løsning som tilfredstiller den enkle delen av kravet, og som har sluppet gjennom det vi har lagt inn av avskjæring, så er vi *garantert* at denne løsningen også tilfredstiller den vriene delen av kravet. Ingen videre test er altså nødvendig. Vær dog klar over at det ofte kan være vanskelig å få til så kraftig avskjæring.

Generering av alle permutasjoner

I eksempel 1 over (med elevene som skulle stilles opp), valgte vi altså å la den enkle delen av kravet rett og slett være at elevene er stilt opp i en eller annen rekkefølge, og vi får derfor problemet med å skrive en programbit som genererer alle mulige rekkefølger av et gitt antall elever (eller mer generelt: av et gitt antall "elementer"). En slik rekkefølge av et gitt antall elementer kalles ofte en *permutasjon* av disse elementene, og vi skal i fortsetningen bruke dette ordet. (Ordet *permutasjon* brukes ofte også i en hårfint annerledes betydning, nemlig som betegnelse på en *omordning* fra én gitt rekkefølge til en annen. Til hver rekkefølge svarer det imidlertid naturlig en omordning, og omvendt, så det gjør liten forskjell.)

Dette med å skulle generere alle permutasjoner av et gitt antall elementer er et problem som dukker opp ganske ofte i denne type problemer (vi skal bl.a. bruke det i den første obligatoriske oppgaven), og vi skal derfor se litt spesielt på dette. Samtidig skal vi se litt på såkalt rekursiv programmering.

For enkelhets skyld kan vi tenke oss at det er n elever, og at de er nummerert fra 1 til n. Vi kan videre tenke oss at de ferdige permutasjonene skal presenteres i en **int** [] p = **new int** [n], etter hvert som de blir generert. Arrayen p kan vi tenke oss er global.

En grei måte å generere alle permutasjoner på, er igjen å se denne oppgaven som en kombinatorisk søkeoppgave, og vi må derved dele kravet å være en permuatsjon av tallene 0 til $n-1$ opp i en enkel og en vrien del. Den enkle delen kan da rett og slett være at

vi vil ha en *sekvens* med lengde n av tallene 0 til $n-1$ (i en sekvens har vi altså ikke noe krav om at tallene skal være forskjellige), og den vriene delen av kravet må da være at alle tallene i sekvensen må være forskjellige. Oppgaven er da redusert til å generere alle slike sekvenser, samt å plukke ut de av sekvensene der alle tallene er forskjellige (helst ved en passelig avskjæring).

Generering av alle sekvenser

Det å systematisk generere alle sekvenser er en oppgave som på svært mange måter tilsvarer det telle. Når vi teller fra null (som vi her bør skrive 000) til 999 generer vi alle sekvenser av lengde 3 av siffrene 0 til 9. Som vi ser går denne telleprosessen ut på å telle det første sifferet "sakte" opp, og for hvert slikt første siffer, å generere alle mulige sekvenser i de to siste posisjonene. Alle sekvenser i de to siste posisjonene framkommer så etter samme mønster. Om vi skal lage alle mulige sekvenser av lengde tre av tallene 0,1 og 2 kan det gjøres helt tilsvarende, nemlig slik (leses linjevis):

```
000  001  002    010  011  012    020  021  022

100  101  102    110  111  112    120  121  122

200  201  202    210  211  212    220  221  222
```

Om vi vil programmere dette (og vi f.eks. ønsker å *skrive ut* alle siffer-sekvensene) så kunne vi, slik som antydet over, ha en global **int** array *p[]* = **new** *int[3]* hvor sekvensene skal produseres. Vi kunne så passelig skrive tre metoder: *gen0, gen1* og *gen2*, som er ansvarlige for å genere alle kombinasjoner av siffer i henholdsvis feltene $p[0,1,2]$, $p[1,2]$ og $p[2]$, samt at *gen2* også kan sørge for utskrift. Om vi skal lage nettopp denne tellingen fra 000 til 222, så kan vi skrive metodene slik:

```
class Gen {
  int [] p = new int [3];
  int n;

  void gen0()
  {for(int siff= 0; siff < 3; siff++)
   { p[0]=siff; gen1(); }
  }

  void gen1()
  {for(int siff= 0; siff < 3; siff++)
   { p[1]=siff; gen2(); }
  }

  void gen2()
  {for(int siff= 0; siff < 3; siff++)
   { p[2] =siff; System.out.println("   " + p[0]+ p[1] + p[2]) ;}
  }
}

 public class GenProg1{

  public static void main (String [] args)
  { new Gen().gen0();
  }
 }
```

Rekursiv programmering

Om vi vil prøve å generalisere denne metoden til lengere sekvenser, møter vi problemer. En ting er at det virker nokså irriterende å måtte skrive en masse metoder som er nesten like. Et mer alvorlig problem er at om vi ikke på forhånd kjenner den maksimale lengden vi vil ha på sekvensene, så blir denne metoden prinsippielt umulig. Den rimelige måten å forsøke å løse begge disse problemene på er å skrive bare én prosedure, som da må være generalisering av alle tre, og som kan styres til å virke nøyaktig som *gen0, gen1* eller *gen2* ved hjelp av en parameter (som passelig kan ha verdiene 0, 1 eller 2). Forsøker vi å generalisere ut fra metodene over, ser vi at følgende metode virker lovende:

```
class Gen {
  int [] p = new int [3];
  int n;

  void gen(int i)
  {for(int siff= 0; siff < 3; siff++)
   { p[i]=siff;
     if (i<2 ) gen(i+1);
     else System.out.println("   " + p[0]+ p[1] + p[2]);
   }
  }
```

```
}

 public class GenProg{

  public static void main (String [] args)
  { new Gen().gen(0);
  }
 }
```

Om man forsøker å kjøre dette (eller et tilsvarende program) i et rimelig moderne språk (f.eks. Java, Simula, Pascal, C++ eller Ada), så vil man se at det virker helt strålende. Forsøker man derimot i gode gamle *Fortran* (eller i *Minila* fra IN 102) så blir det bare tull. Grunnen er at man i de førstnevnte språkene oppretter et nytt dataareal til de lokale variablene (inklusive parametrene) hver gang man gjør et metodekall. Dermed kan man godt gjøre et nytt kall på en metode som allerede har et kall i gang, uten at de to kallene kommer til å forstyrre hverandre i det hele tatt. I Fortran og Minila får imidlertid hver metode bare ett sett med variable, og om man forsøker å gjøre noe som likner på det over, vil derfor de forskjellige kallene arbeide på de samme variablene, med tilsvarende katastrofale følger.

Det at det gjøres nye kall på en metode mens gamle er i gang, kommer nettopp til å skje i utførelsen av metoden over. Først startes et kall på *gen* med parameteren 0. Fra dette vil vi så (gjentatte ganger) starte kall på den samme metoden, denne gang med parameteren 1. Endelig vil hver av disse kallene i sin tur starte et antall kall med parameteren 2, men disse siste vil ikke gjøre noen videre kall, men bare gjøre utskrift. Utførelsen går altså nøyaktig som om det var tre forskjellige metoder vi kalte, slik det var skrevet først.

Denne teknikken med å la en metode kalle seg selv, kalles gjerne *rekursiv programmering*, og de aller fleste moderne språk tillater altså dette. For programmereren kan denne teknikken virke litt forvirrende og problematisk i begynnelsen, men når man har vendt seg til tankegangen viser det seg å være et meget nyttig hjepemiddel som vi stadig skal benytte oss av.

Rekursjonsbrønner

En ting man skal passe seg for når man benytter rekursiv programmering, er å havne i en såkalt "rekursjonsbrønn". Med dette mener vi at man bare setter i gang kall etter kall, uten noen gang å avslutte noen av dem. Dersom man ved en feiltakelse lager et program som gjør dette, vil lageret snart bli fylt opp med metodekall (det trengs litt plass til hvert), og programmet vil stoppe med en beskjed om at all tilgjengelig plass for denne utførelsen er brukt opp. For å unngå slike rekursjonsbrønner, må man alltid passe på at noe forandrer seg fra kall til kall (det er gjerne parameteren som forandres, slik som i eksempelet over), og denne forandringen må være slik at man en gang kommer til et kall som *ikke* gjør nye kall (dette skjer i eksempelet i det kallet der parameteren i er 2).

Vi ser til slutt på en generalisering av programmet over, der vi kan lage sekvensene av lengde n, og der de enkelte elementer i sekvensene vi være tall fra 0 til $n-1$. I tillegg til at vi må passe på at arrayen p er (minst) n lang, er det stort sett bare å erstatte hvert tre-tall med n. Vi antar at n er en global variabel.

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

```
  Gen(int i) { n = i; p = new int [n];}

  void gen(int i)
  {for(int siff= 0; siff < n; siff++)
   { p[i]=siff;
     if (i < n-1 ) gen(i+1);
     else
     {for (int j=0; j < n; j++)
         System.out.print(""+p[j]);
      System.out.println("  ");
     }
   }
  }

}

 public class GenProg{
  // start program >java GenProg 'n'
  public static void main (String [] args)
  { new Gen(Integer.parseInt(args[0])).gen(0);
  }
 }
```

Rekursiv generering av alle permutasjoner

Vi har altså nå et program som systemetisk genererer alle sekvenser med lengde n av tallene fra 1 til n. (Det hadde ikke her vært noe vanskeligere om lengden av sekvensene og verdiområdet for tallene hadde vært forskjellige. Det er bare fordi vi skal generere permutasjoner at vi lar disse være like). Avskjæringen som skal sørge for at vi bare får ut de sekvenser som er permutasjoner viser det seg også er grei å legge inn, men la oss først skrive et program som ikke gjør noen avskjæring, bortsett fra å teste hver ny sekvens for å se om det er en permutasjon. Hvordan denne testen skal gjøres i detalj bryr vi oss ikke med, for vi skal snart gi en forbedret utgave av hele programmet. Programmet kan ta seg slik ut.

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

  Gen(int i) { n = i; p = new int [n];}

  void gen(int i)
  {for(int siff= 0; siff < n; siff++)
   { p[i]=siff;
     if (i < n-1 ) gen(i+1);
     else
     { // ny sekvens er generert i p
       < Test om denne sekvensen er en permutasjon>
       < I så fall: lever den videre til 'bruk()'>
     }
   }
  }

}

 public class GenProg{
  // start program >java GenProg 'n'
  public static void main (String [] args)
  { new Gen(Integer.parseInt(args[0])).gen(0);
  }
 }
```

Dette er altså ingen genial måte å generere alle permutasjoner av lengde n, da vi vil ende opp med svært mange ferdiglagede sekvenser som ikke er permutasjoner. Om vi ser på lista for $n = 2$ lenger opp, ser vi at metoden i alt vi generere 27 forslag, mens bare 6 av disse faktisk vil være permutasjoner. Når n blir større blir forholdet bare enda verre. Avskjæringen vi skal foreta under vil rette på dette på en dramatisk måte.

Genereringsrekkefølgen

Programmet over er imidlertid fint på en måte, nemlig med det at det er lett å se at permutasjonene blir generert i "stigende rekkefølge". Det vil si at om vi sammenlikner permutasjoner (eller sekvenser generelt) omtrent som vi sammenlikner navn i telefonkatalogen, så vil alltid en mindre permutasjon bli generert før en større. Dette følger uten videre av at dette helt selvfølgelig gjelder for de sekvenser som genereres (studér f.eks. genereringen av sekvensene fra 000 til 222), og når vi bare plukker ut de sekvenser som er permutasjoner, så vil det også gjelde dette utplukket.

Avskjæring

For å lage en versjon av dette programmet som ikke kaster bort mye tid på å produsere sekvenser som det umulig kan bli permutasjoner av, skal vi legge inn en fullstendig avskjæring. Denne skal rett og slett gå ut på at vi ikke forsøker å bygge videre på en halvferdig sekvens som allerede inneholder to like tall. Dette kan gjøres på flere måter, men den greieste

måten er å ha en tabell som viser hvilke tall som allerede er i bruk (til venstre for den posisjonen vi holder på med). Denne tabellen kan være en **boolean** [] brukt = **new boolean** [n] , og den kan være **true** for de verdier som er brukt, og **false** ellers.

Programmeringen av dette blir helt rett etter nesa. Det eneste som det kan være lett å glemme er at vi må ta vekk igjen avmerkingen i arrayen *brukt* når vi er ferdig med et tall i en gitt posisjon. Programmet kan f.eks. ta seg slik ut:

```
class Gen {
  int [] p;
  boolean [] brukt;
  int n;

  Gen(int i)
   { n = i; p = new int [n];
     brukt= new boolean[n];
     for(int j = 0; j<n; j++)
       brukt[j] = false;
   }

  void gen(int i)
  {for(int siff= 0; siff < n; siff++)
   if (! brukt[siff])
   { brukt[siff] = true;
     p[i]=siff;
     if (i < n-1 ) gen(i+1);
     else
     {   ! Ny kombinasjon er generert i p;
         ! Vi VET nå at dette er en permutasjon (fullst. avskjæring) ;
         < Lever arrayen p til videre "bruk" >;
     }
     brukt[siff] = false;
   }
  }

}

 public class AvskjProg{
  // start program >java AvskjProg 'n'
  public static void main (String [] args)
  { new Gen(Integer.parseInt(args[0])).gen(0);
  }
 }
```

Det er nå lett å observere at permutasjonene her blir generert i samme rekkefølge som i programmet over, altså også her i stigende rekkefølge.

En mer direkte generering av permutasjoner

Før vi forlater temaet med å generere permutasjoner, skal vi se på en metode som hvertfall i prinsippet er mer effektiv enn metoden diskutert over, da den ut fra sin natur aldri kan generere noe annet en permutasjoner, og derfor ikke behøver noe slags avskjæring. Den krever imidlertid noe mer administrasjon, og vil derfor i praksis neppe være særlig mye raskere enn metoden over, hvertfall ikke for rimelig små verdier av n. Når det gjelder rekursiv programmering krever denne at man holder tunga litt rettere i munnen, så vi tar den også med for treningens skyld.

Metoden skal altså programmeres som et rekursivt program, og hvilken del av arrayen p de enkelte kall har ansvaret for, skal styres nøyaktig som i programmet over. Den store forskjellen er imidlertid at arrayen p nå fra begynnelsen av skal inneholde de aktuelle elementene, og at vi hele tiden bare skal bytte rundt på disse. Dermed blir det selvfølgelig aldri snakk om å få fram noe annet enn permutasjoner.

For å kunne bytte rundt på elementene i arrayen p slik vi ønsker, skal vi definere to metoder, nemlig én som rett og slett bytter om innholdet av de to posisjonene i og j (den heter *bytt*), og én som gjør en forskyvning ett hakk mot venstre av alle elementene i feltet fra posisjon i til posisjon n, og som flytter det som stod i posisjon i opp til posisjon n. Denne operasjonen skal vi kalle en *rotasjon* mot venstre. Begge disse metodene trenger en ekstra hjelpevariabel, og de er programmert under.

Hovedideen med det programmet vi nå skal skrive er at når vi nå kaller vår rekursive metode med parameteren i, så har dette kallet til oppgave å generere alle permutasjoner av de elementene som i kalløyeblikket står i feltet fra posisjon i til posisjon n (samt å levere arrayen p til "videre bruk (f.eks. utskrift) for hver av disse permutasjoner). I tillegg skal metoden sørge for at når kontrollen leveres tilbake til kalleren, så skal elementene stå i nøyaktig samme rekkefølge som da den ble kalt.

Om hver av metodekallene holder seg til denne spesifikasjonen, så kan de gjøre sin del av jobben som følger: For å generere alle permutasjoner av elementene i "sitt" felt (fra posison i til posisjon $n-1$), så bytter den hver av disse elementene etter tur ned i posisjon i, og ber så, for hver av disse, om å få generert alle permutasjoner av elementene i feltet fra posisjon $i+1$ til posisjon $n-1$.

La oss anta at vår rekursive metode heter *gen-perm*. Om $n = 6$, så kan jobben til kallet *gen-perm(2)* passelig gjøres som følger:

```
gen_perm(3);              ! Behold først den opprinnelige i p[2] ;
bytt(2,3); gen_perm(3);   ! Bytt p[3] ned i p[2] ;
bytt(2,4); gen_perm(3);   ! Bytt p[4] ned i p[2] ;
bytt(2,5); gen_perm(3);   ! Bytt p[5] ned i p[2] ;
roterVenstre(2);         ! Rydd opp ;
```

Grunnen til at vi må gjøre en *roter-venstre* operasjon på slutten, er for å korrigere for den samlede effekt av de ombyttingene vi har gjort. Legg merke til at vi forutsetter at kallet *gen-perm(3)* hver gang returnerer med den samme sekvensen i *sitt* felt (posisjonene 3, 4 og 5) som vi gav den ved kallet, og *vi* må altså gjøre det samme overfor den som har kalt oss (for å få generert alle permutasjoner av det som står i posisjonene 2, 3, 4 og 5).

Om vi fremdeles antar at $n = 6$, så ser vi at kallet *gen-perm(5)* blir nokså spesielt. Det skal generere alle permutasjoner av den sekvensen som står i $p[5]$, og dette blir selvfølgelig bare én permutasjon. Dette kallets eneste oppgave blir derfor å sørge for at det innholdet som nå ligger i p blir levert til videre bruk (hva nå det måtte være).

Om vi summerer opp alt dette, og generaliserer det hele til en vilkårlig n, så kan metodene skrives som følger.

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

  Perm(int num)
   { // Konstruktor: initier p
     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  - byttes ut i subklasse
    // skriv ut permutasjonene
    for (int i = 0; i < n; i++)
    System.out.print  (p[i]);
    System.out.println();
  }
}

 public class PermProg{
```

```
  // start program: >java PermProg 'n'
  public static void main (String [] args)
  {  new Perm (Integer.parseInt(args[0])).permuter (0);
  }
}
```

Dette programmet har altså den egenskapen at det vil generere alle mulige permutasjoner av de elementene som stod i p fra begynnelsen, og vil avslutte med elementene satt tilbake i den opprinnelige rekkefølge. Dette vil gjelde uavhengig av om elementene er tall, blomsternavn, eller hva annet man kunne tenke seg.

Det er imidlertid interessant å observere at dersom man arbeider med heltall, og lar den opprinnelige sekvensen i arrayen p være tallene i stigende rekkefølge (slik det er gjort i programmert over), så vil også dette programmet generere permutasjonene i stigende rekkefølge, altså i nøyaktig samme rekkefølge som de tidligere programmene. Vi lar beviset av dette være en oppgave, men røper at nøkkelen til dette ligger i at hver gang metoden kalles med parameteren i, så er elementene $p[i], \ldots, p[n-1]$ i stigende rekkefølge.

Tilbake til elevene som skulle stilles opp

Vi har nå flere metoder til systematisk å generere permutasjoner, vi har trening i rekursiv programmering (!?) og vi har sett eksempler på enkel avskjæring. Vi er derfor vel rustet til å gå tilbake til noen av de opprinnelige problemene, og vi vil først se mer på problemet med elevene som skulle stilles opp på rekke, slik at ingen uvenner ble stående ved siden av hverandre.

For enkelhets skyld kan vi her anta at de som er uvenner er nettopp de som har nabonummer i den nummerrekkefølgen vi gav dem. (Vi kan tenke oss at de er nummerert i den rekkefølgen de *pleier* å bli stilt opp, og at alle er blitt uvenner med sine naboer her.) Det blir imidlertid helt tilsvarende om vi har en tabell over de som er uvenner, f.eks. som en todimensjonal boolsk matrise. Det er en passelig oppgave å gjøre de nødvendige forandringer i programmet vi kommer fram til under.

Det viser seg da å være lett, ut fra begge versjonene vi har av permutasjonsgenereringsprogrammet, å legge inn avskjæring med hensyn på dette kravet. Det er i all enkelhet ikke noe vits i å arbeide videre med et forslag som allerede har to uvenner ved siden av hverandre. Om vi velger å ta utgangspunkt i *permuter*, og vi antar at vi bare er interessert i *en eller annen* brukbar oppstillingsrekkefølge, så kan programmet da ta seg slik ut:

```
// Innledende deklarasjoner av arrayen "p" og
//  ombyttinsmetodene blir som tidligere;

  boolean ferdig = false;

  final void permuter (int i)
  { // finn neste permutasjon og kaller "brukPerm()"
    // N.B. Permutasjonene startes ved kallet: permuter(0);
   if ( ! ferdig)
   { if ( i == n-1 && (i==0 || (Math.abs(p[i]-p[i-1])) != 1) )
     // Arrayen p inneholder en lovlig oppstilling av elevene
     {  brukPerm(); ferdig = true;}
      else {
       if (i==0 || (Math.abs( p[i]-p[i-1])!= 1)) permuter(i+1);
       for (int t = i+1 ; t < n; t++)
       { bytt (i,t);
        if (i==0 || (Math.abs( p[i]-p[i-1]))!= 1)  permuter(i+1);
       }
       roterVenstre(i);
      }
    }
  }


permuter(0)  // Kallet som setter det hele i gang;
```

Dette programmet er altså skrevet med tanke på at vi bare var interessert i *en eller annen* lovlig oppstilling. Dersom vi er interessert i *alle* lovlige oppstillinger, så er det bare ta vekk **if**(! ferdig)-setningen, slik at genereringen av oppstillinger fortsetter.

Korteste reiserute

Som et siste eksempel skal vi se på problemet til en reisende som må innom 'n' byer, og som vil gjøre reisen så kort som mulig. Vi kan anta at byene han skal innom er nummerert fra 0 til $n-1$, og at han starter i by nummer 0, og at han også skal tilbake hit. Avstanden mellom ethvert par av byer kan vi tenke oss finnes i en to-dimensjonal array *avstand*.

En rekkefølge å besøke byene i kan vi se som en permutasjon av byene $1, 2, \ldots, n-1$. Vi kan derved løse oppgaven ved å generere alle slike permutasjoner, regne ut den reiselengde hver av dem representerer, for så til slutt å velge den som gav minst lengde. Det blir lett et svært stort antall permutasjoner å gå gjennom, og enhver avskjæring vil derfor være av interesse. Her er det mange mer eller mindre intrikate metoder og triks som kunne vurderes, men vi skal nøye oss med å se på en type avskjæring som typisk kommer på tale når man, som her, er ute etter et maksimum eller et minimum.

Denne avskjæringen går ut på å stoppe videre vurdering av et alternativ så fort vi innser at videre utbygging av dette alternativet må føre til en løsning som er dårligere enn den beste vi allerede har sett. I vårt konkrete tilfelle kan vi generere permutasjonene slik som over, og hele tiden holde orden på hvor lang den reiseruten vi til nå har valgt, vil være. Når vi så vurderer hvilken by vi skal reise til som den neste, kan vi rett og slett hoppe over de byene der avstanden fram til denne byen pluss avstanden derfra direkte tilbake til startstedet, er lenger enn den korteste (fullstendige) reiseruten vi allerede har sett.Dette

programmet forutsetter i sin avskjæring at den såkalte trekant-ulikheten er tilfellet mellom avstandene - dvs. at avstanden direkte til en by fra enhver annen by er kortere enn å gå til denne byen via en tredje (vilkårlig valg) by. (Prøv å forklare hvorfor)

Det er lett å se at om vi også gjør denne betrakningen for den siste byen så blir dette en fullstendig avskjæring, i den betydning at når vi først kommer ut med en ny fullstendig rekkefølge, så er dette også en kortere reiserute en den beste vi har sett til nå.

Programmeringen av dette blir derfor ganske enkel, og som en variasjon kan vi denne gangen bruke det permutsjonsprogrammet vi *først* skrev, som utgangspunkt:

```
float [][] avstand;
int [] p, minp ;

int n;
boolean [] brukt;
float lengde = 0,  minLengde = Float.MAX_VALUE;

void finnVeier (int i)
{
   for (int by= 1; by < n; by++)
   {if (! (brukt [by]) &&
           !(lengde+avstand[p[i-1]][by] + avstand[by][0]>= minLengde))
      { brukt[by] = true;
        lengde = lengde + avstand [p[i-1]][by];
        p[i] = by;

        if ( i < n -1) finnVeier(i+1);
        else
        { // nytt, bedre  mulig forslag er generert
           minLengde = lengde + avstand[by][0];
        }
        brukt[by] = false;
        lengde = lengde - avstand [p[i-1]][by];
       }
    }
  }


}
// I omliggende klasse initieres brukt[i] til false
// avstand[i][j] leses inn, og p[0]= 0;

 finnVeier (1);    // starter søket
 System.out.println(‘‘Korteste vei er: ‘‘ + minLengde);
```

Hvis vi ikke kan anta at trekantulikheten er oppfylt (f.eks ford noen veier er svingete og lengere mens andre er nesten helt rette), blir søkeprosedyren finnVeier som nedenfor. Merk at vi nå må teste mer når vi vil oppdatere *minLengde*. Initiering og oppstart som tidligere.

```
void finnVeier (int i)
   {
     for (int by= 1; by < n; by++)
     {if (! (brukt [by]) && !(lengde+avstand[p[i-1]][by] >= minLengde))
        { brukt[by] = true;
          lengde = lengde + avstand [p[i-1]][by];
          p[i] = by;

          if ( i < n -1) finnVeier(i+1);
          else if( lengde +  avstand[by][0] <  minLengde )
          { // nytt, bedre  mulig forslag er generert
             minLengde = lengde + avstand[by][0];
          }
          brukt[by] = false;
          lengde = lengde - avstand [p[i-1]][by];
         }
      }
    }
```

En annen avskjæring som alltid vil virke, er å først lage en array *korteste[..]* over summen av de i korteste avstander, i= 1,2,..,n-1, totalt i avtandsmatrisen, og så i testen for avskjæring teste om !(lengde + avstand[p[i-1]] [by] + korteste [n-i-1] >= minLengde). Denne kan legges inn i denne andre versjonen av $finnVeier$, men merk at vi likevel må beholde testen når vi oppdaterer *minLengde*.

Programmet overfor vil ikke til slutt kunne skrive ut den beste veien (som en sekvens av byer), men bare hvor lang denne veien er. Det er en passelig øvelse å utvide programmet slik at det også skriver ut selve veien som bør velges. Verdien av "uendelig" kan velges som $(n+1)$ ganger den lengste avstanden i avstandstabellen. Det vil imidlertid effektivisere programmet betraktelig om man på forhånd per hånd finner et rimelig godt veivalg, og så initialiserer *minLengde* til lengden av denne veien. Hvorfor hjelper dette?

Man kan også lett tenke seg andre muligheter som kunne forbedre programmet. F.eks. tar programmet over de aktuelle byene vi kan gå til i neste steg i en fast rekkefølge. Kanskje kunne man i stedet forsøke de nærmeste byene først, for derved tidligere å få en løsning som er rimelig god, og derved få bedre avskjæring siden. Generelt kan vi si at det kan ofte lønne seg å bruke nokså tidkrevende tester til avskjæringen dersom det er håp om at de kan oppdage håpløse delløsninger på et tidlig tidspunkt.

Permutasjoner løser ikke alle problemer

Ut fra eksemplene over kan det virke som om generering av permutasjoner alltid kommer inn i løsningen av kombinatoriske søkeproblemer. Dette er imidlertid langt fra sannheten, og vi skal forhåpentligvis i løsning av oppgaver demonstrere dette. Har man først forstått prinsippene for avskjæring etc. for permutasjoner, lar disse seg imidlertid forholdvis lett overføre også til andre situasjoner.

Oppsummering om effektiv avskjæring

Som en avslutning på dette kapittelet summerer vi opp noen tommelfingerregler man bør følge når man skal forsøke å løse et kombinatorisk søkeproblem så effektivt som mulig.

- Oppdelingen av kravet i en *enkel* og *vrien* del bør gjøres slik at mest mulig av kravet kommer inn i den enkle delen, men samtidig slik at man effektivt kan generere alle løsninger som tilfredstiller det enkle kravet.
- Måten man systematisk genererer alle løsninger på det enkle kravet bør være slik at det underveis er lettest mulig å teste ting som har med det vriene kravet å gjøre.
- At man så *tidlig som mulig* under genereringen av de enkle løsningene, klarer å kjenne igjen de del-løsninger som umulig kan utbygges til en fullstendig løsning som tilfredstiller den vriene delen av kravet (eller bli noen "bedre" løsning enn den beste vi allerede har funnet). I disse tilfellene må man da sørge for å avskjære videre generering av enkle løsninger.

Et par oppgaver:

Plassering av 9 sifre

Vi skal forsøke å finne måter å sette sammen de 9 sifrene: $1, 2, \ldots, 9$ til en permutasjon $x_1, x_2, \ldots, x_9$, slik at (det desimale) tallet x_1x_2 er delelig med 2, tallet $x_1x_2x_3$ er delelig med 3 osv., helt til at tallet $x_1x_2 \ldots x_9$ er delelig med 9. Finn alle slike måter å ordne disse sifferene på.

Tillegg: Man kan også tenke seg å utvide denne oppgaven til R-tall-systemet (i stedet for ti-tall-systemet), og forsøke å finne en rekkefølge av sifrene $1, 2, \ldots, R-1$ slik at det tilsvarende kravet blir oppfylt. Oppgavestilleren er ikke kjent med om det finnes løsninger for verdier av R forskjellige fra ti.

Magiske firkanter

Vi tenker oss at en skjema på n ganger n ruter skal fylles med alle tallene fra 1 til n^2 - en permutasjon av disse, slik at summen av alle kolonner, alle linjer og begge hoveddiagonalene blir den samme. Skriv et program som finner alle slike konstellasjoner for en gitt n.

Det programmet man naturlig ender opp med lar seg lett utføre for $n = 3$, men allerede for $n = 4$ tar det gjerne noen minutter før den første løsninger kommer, og det å få ut alle løsningene ser håpløst ut. For $n = 5$ ser det håpløst ut allerede å få ut den første løsningen. Altså: Bedre avskjæring etterlyses!