N-dronningproblemet

Obligatorisk oppgave 1
I120, H-2000

Innleveringsfrist	:	Mandag, 2. Oktober, kl.10:00
	–	Besvarelsen legges i arkivskapet på UA i skuff merket I120
Innhold:	–	utskrift av godt dokumentert (Javadoc) programkode (bruk `c2ps` ved utskrift). Utskrift av `javadoc` er ikke nødvendig
	–	utskrift fra kjøring av løsninger med $n = 7$ og $n = 8$ (bruk `script`)
	–	navnet på programmer og katalogen hvor de ligger (programmene skal kunne eksekveres av andre; bruk `chmod a+x programnavn.*`)

Problemet er beskrevet i seksjon 1 under. Du skal skrive og levere inn 3 programmer som løser dette problemet, og disse er beskrevet i hhv seksjon 2.1, 2.2 og 2.3. Det første programmet er et rett-fram naivt løsningsforslag. Etter at denne er implementert skal den forbedres ved å konstruere passende avskjæringsmekanismer for å effektivisere programmet. Tilslutt skal denne løsningen videre forbedres ved å utelukke "like" løsninger. Alle programmene skal bruke rekursjon, med passende avskjæringer for effektivisering. I seksjon 2.5 og 2.6 beskriver vi to frivillige deler, hhv en real-time tidskonkurranse, og en mulig visualisering av problemet vha GUI.GridMatrix.

1 Problembeskrivelse

Målet med n-dronningproblemet er å plassere n dronninger på et n-dimensjonalt sjakkbrett (en $n \times n$-matrise) slik at ingen dronning kan angripe en annen. En dronning kan angripe en annen dersom de står på samme horisontale, vertikale eller diagonale linje.

1.1 Tips

1.1.1 Representasjon av sjakkbrettet

Den naive representasjonen av et sjakkbrett med dronninger på er en to-dimensjonal tabell. Men siden vi vet at i en løsning kan ingen dronninger stå i samme rad eller samme kolonne, kan vi bruke en smartere representasjon. Vi vet at det må være en, og bare en, dronning i hver rad (og hver kolonne). En stilling (av n dronninger) på brettet kan da representeres ved n verdier som sier hvilken kolonne dronningene i

de n radene befinner seg. Vi kan presentere disse n verdiene som en liste.

	0	**1**	**2**	**3**
0	•			
1				•
2		•		
3			•	

Sjakkbrettet over kan representeres som listen:

$$[0, 3, 1, 2]$$

1.1.2 Permutasjoner

Gitt en liste med n tall $[0, 1...n-1]$, er en *permutasjon* en ny liste med de samme tallene i en vilkårlig rekkefølge. F.eks., er $p_1 = [0, 3, 1, 2]$ og $p_2 = [2, 1, 3, 0]$ to permutasjoner av listen $[0, 1, 2, 3]$.

Alle listene vi er interessert i er blant de som er permutasjoner av $[0, 1...n-1]$; vi er f.eks. ikke interessert i lister som inneholder det samme tallet to ganger. Hvis vi kun ser på slike permutasjoner, sikrer vi at ingen dronninger kan angripe hverandre horisontalt eller vertikalt. I tillegg, for at en permutasjon skal være en løsning, må vi sjekke at ingen dronninger står på samme diagonal. På tegningen over står dronningene 2 og 3 på samme diagonalen og dermed er ikke permutasjonen $[0, 3, 1, 2]$ en løsning.

I flere av oppgavene i seksjon 2 er det nødvending å ha kontroll med hvilken *rekkefølge* man konstruerer permutasjoner. En måte å gjøre dette på er illustrert på figur 1.1.2(for $n = 4$). Løvnodene i dette treet er alle mulige permutasjoner av $[1, 2, 3, 4]$. De interne nodene er "ufullstendige" permutasjoner; lister der bare noen elementer er fiksert. På det øverste nivået i treet (rotnoden) er ingen elementer fiksert, på det neste er ett element fiksert, o.s.v. En slik metode som er illustrert her resulterer i en *total ordning* av permutasjoner, gitt ved å gjennomgå løvene fra venstre mot høyre. Den konkrete metoden som er brukt her gir en *leksikografisk ordning* der den første posisjonen (fra venstre) som inneholder forskjellige tall i to permutasjoner, avgjør hvilken som er minst. F.eks. for $p_1 = [0, 2, 4, 1, 3]$ og $p_2 = [0, 2, 1, 3, 4]$ er det tredje elementet det første som skiller disse og, siden $4 > 1$, har vi at p_2 'er mindre enn' p_1.

1.1.3 Diagonalt angrep

Merk at en dronning som står på rad r_1 og kolonne k_i kan angripe diagonalt en annen dronning på rad r_j og kolonne k_j hvis og bare hvis

- $k_i - r_i = k_j - r_j$ (parallel til hoveddiagonalen)
- $k_i + r_i = k_j + r_j$ (parallel til bi-diagonalen)

2 Oppgaver

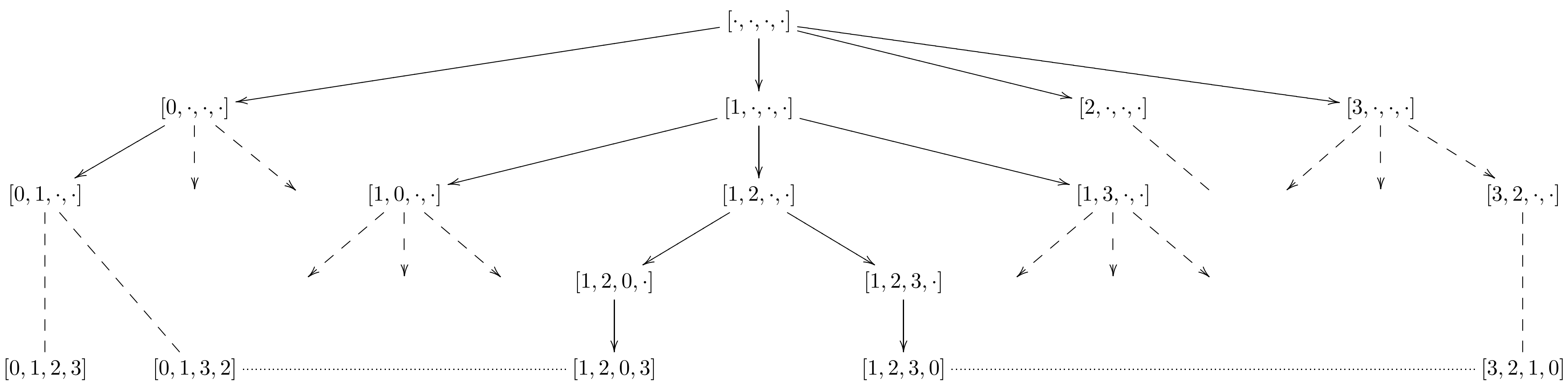
2.1 En naiv løsning

En naiv løsning på n-dronning problemet er å lage en algoritme som

1. genererer alle mulige stillinger
2. og så sjekker om en gitt stilling er en løsning

Dette kan gjøres ved følgende program:

1. input til programmet: n



Figur 1: Permutasjons-tre. I en liste angir $\cdot$ at elementet i listen ennå ikke er bestemt.

2. generer alle mulige permutasjonene av tallene $0, 1...n-1$, og
3. for hver permutasjon sjekk om noen dronninger står på den samme diagonalen
4. output av programmet: alle mulige løsninger.

Implementer denne algoritmen.

Du vil raskt se at denne løsningen er *uakseptabel* – allerede for små verdier av n vil det ta meget lang tid. (Hva er kompleksiteten til denne algoritmen?)

2.2 Avskjæring

Skal man løse dette problemet effektivt, må man i tillegg komme opp med en del avkjæringsmekanismer for å redusere søkerommet. Den første algoritmen du laget er ineffektiv fordi det veldig ofte er unødvendig å generere hele permutasjonen for å avgjøre at den ikke er en løsning. Har vi f.eks. en permutasjon som starter $[0, x, 2...]$, så vet vi at uansett hva ... inneholder, blir det ikke noen løsning siden allerede den 0-te og den 2-dre dronning kan angripe hverandre.

Istedenfor å generere hele permutasjonen *før* den sjekkes, kan man sjekke den *mens* den genereres.

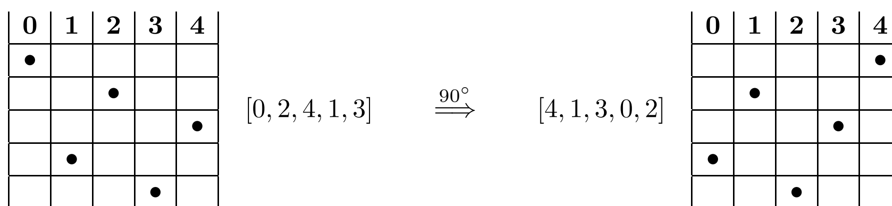
Du skal lage et program som oppfyller følgende:

1. input til programmet: n
2. generer permutasjoner av $0, 1...n-1$ rekursivt
3. utfør passende sjekk slik at rekursjonen stopper (og dermed avskjærer rekursjonstreet) når det oppdages at permutasjonen som er iferd med å bygges opp ikke kan lede til en løsning (som i eksempelet over).
4. output av programmet: alle mulige løsninger

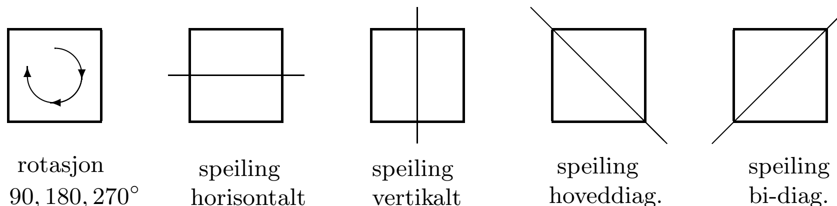
Programmet *skal* være rekursivt, og det skal være vesentlig raskere enn den naive løsningen fra 2.1!

2.3 Speiling og rotering

Tenk deg at du har funnet en løsning. Da kan du finne en ny løsning ved å *rotere* brettet 90°. Den nye løsningen er ikke vesensforskjellig fra den opprinnelige fordi brikkenes plassering i forhold til hverandre er uendret. Likevel er det en ny løsning i den forstand at brettet representeres ved ulike permutasjoner. Et eksempel viser dette:



Tilsvarende kan du generere "nye" løsninger ved å rotere 180° og 270°, samt ved å speile (vende) brettet langs horisontal-aksen, vertikal-aksen, eller en av diagonalene:



Følgende tabell viser, for $n = 4, 5...10$ antall forskjellige løsninger (#tot) som skulle finnes i oppgave 2.1, samt antall 'vesentlig forskjellige' løsninger (#ulik) der ingen kan oppnåes ved rotering eller speiling fra en annen:

n	#tot	#ulik
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92

Du skal nå videreutvikle programmet ved å utelukke 'uvesentlige løsninger'. Med andre ord: hver gang programmet genererer en fullstendig løsning, skal det sjekke at denne ikke kan fåes ved noen av avbildningene fra en løsning som allerede ble tatt med (og skrevet ut). [1] Dette kan, f.eks. gjøres ved å

1. passe på at permutasjoner genereres i en bestemt total ordning

2. for hver løsning l og for hver avbildning rs, konstruer permutasjonen, som vi kaller $\overline{rs}(l)$, som ville gitt permutasjonen l under avbildningen rs. Eksempel: La $l = p_2 = [4, 1, 3, 0, 2]$ og la rs være 90°-rotasjon. Vi skal finne en permutasjon p_1 som ville gi p_2 om vi roterte p_1 med 90°. Da må vi se hva vi får om vi roterer p_2 med $-90°$. Svaret $p_1 = \overline{rs_{90}}(p_2)$ kan dermed beregnes som følger:

   $$\texttt{for alle } k = 0, 1...n-1 : p_1[n-1-p_2[k]] = k \ ;$$

   der $p[x]$ angir elementet på plass x i listen p. Svaret blir $p_1 = [0, 2, 4, 1, 3]$. Dette tilsvarer brettene i figuren gitt i begynnelsen av denne seksjonen. Du skal finne tilsvarende definisjoner for alle avbildningene

3. sjekk, for alle rs, om $\overline{rs}(l)$ er 'mindre enn' l – i så fall har en tilsvarende løsning alt blitt skrevet ut og l skal ikke være med i løsningsmengden.
   (I eksemplet over fant vi en p_1 som var mindre enn p_2, siden $p_1[0] = 0 < 4 = p_2[0]$, (dersom permutasjoner genereres i den leksikografiske ordningen) og p_2 er dermed ikke med i den reduserte løsningsmengden.)[2]

Følger du denne oppskriften, får du, for $n = 8$, løsningene generert i følgende rekkefølge:

1 : $[0, 4, 7, 5, 2, 6, 1, 3]$
2 : $[0, 5, 7, 2, 6, 3, 1, 4]$
3 : $[1, 3, 5, 7, 2, 0, 6, 4]$
4 : $[1, 4, 6, 0, 2, 7, 5, 3]$
5 : $[1, 4, 6, 3, 0, 7, 5, 2]$
6 : $[1, 5, 0, 6, 3, 7, 2, 4]$
... : *osv.*

2.4 Følgende skal innleveres

NB: Du skal skrive 3 programmer, slik de er beskrevet i seksjonene 2.1, 2.2 og 2.3, og innlevere disse slik det er beskrevet øverst på første side.

[1] Faktisk kan dette forbedres da noen av avbildningene kan avskjæres før en fullstendig løsning foreligger.

[2] Faktisk kan man forbedre denne biten også i.o.m. at, som eksempelet viser, kan noen løsninger – som p_2 – forkastes uten å konstruere hele forbildet $p_1 = \overline{rs}_{90}(p_2)$.

2.5 Frivillig: Real-Time Tidskonkurranse

Kjøretiden til dine program på f.eks. $n = 13$ kan måles som følger: Gitt at programmet ditt heter `queen`, kjør Unix kommando `> /bin/time java queen 13`. I tillegg til `queen 13` sin output, får du da en liste med 3 tidsangivelser: `real ...`, `user ...` og `sys ...` Det er den andre, `user`-tid som vi skal se på her. Denne tiden skal – i prinsippet – være uavhengig av maskinbelastning osv. Den kan dog variere litt fra kjøring til kjøring, som dere selv vil oppdage. På hjemmesiden til kurset vil vi kontinuerlig legge ut beste kjøretiden oppnådd av en av dere, for input $n = 13$, på våre maskiner, dvs ikke hjemmemaskin. Output tar mye tid, så vi gjør tidsmålingen på følgende program med minimal output:

- Programmet fra 2.3 som beregner antall ulike løsninger for $n = 13$, dvs størrelsen på en fullstendig samling løsninger hvor ingen av disse kan oppnåes gjennom rotasjon og speiling av noen annen løsning i samlingen, og skriver kun ut dette tallet. Den som implementerer ifølge fotnotene gitt i seksjon 2.3 vil ha en stor fordel.

Såsnart du har et program som er raskere enn bestetiden som ligger ute på vår hjemmeside, sender du kjøretiden, og outputen programmet ditt ga, til agotnes@ii.uib.no, så vil vi kontakte deg for å teste ditt program og eventuellt oppdatere bestetiden. Heder, ære og en symbolsk premie til vinner(e).

2.6 Frivillig: Visualisering

Bruk GUI.GridMatrix (som du brukte i første uke til å visualisere Conway's spill) til å visualisere løsningene generert av ditt program. Utvid dette programmet slik at bruker kan gi følgende kommandoer:

- `q n` – der `n` er et heltall; finn løsningene for $n \times n$ brett; og vis første løsning på skjermen;
- `v` – vis neste løsning på sjermen (etterfølgende `v`-kommandoer skal vise etterfølgende løsninger).

Her kan altså bruker se på noen løsninger for en gitt `n` (med `q n` etterfulgt av noen `v`'er), og så skifte til en annen brettstørrelse ved å gi kommando `q m` for en `m` $\neq$ `n`.