n-dronningsproblemet

Obligatorisk oppgave 1

i-120, h-98

innleveringsfrist	:	fredag, 25.09, kl.18:00
innleveres:	–	listing av programkode (bruk `c2ps`) med dokumentasjon (ev. generert med `javadoc`)
	–	utskrift fra kjøring av programmet med $n = 8$ og $n = 10$ (bruk `script`)
	–	navnet på programmet og katalogen hvor det ligger (programmet skal være tilgjengelig for andre bruk `chmod a+x programnavn.*`)

Problemstilling tillatter en meget enkel løsning men oppgavens Del I går ut på å finne og programmere passende avskjæringsmekanismer for å effektivisere programmet. I Del II skal du videre forfinne programmet fra Del I.

Oppgaven skal løses ved å bruke rekursjon. Oppgaven illustrerer visse farer ved å stole ubetinget på $\mathcal{O}$-analyse. Både den enkle algoritmen og den du skal programmere har samme kompleksitet – men vi setter her noen "real-time" begrensinger nettopp for å se at den kan være av betydning.

1 Del I

1.1 Problembeskrivelse

Vi har en $n \times n$ matrise som representerer sjakkbrettet (dog, n kan velges vilkårlig). Problemet er å plassere n dronninger slik at ingen slår hverandre.

Det er meget enkelt (vi skisserer dette lenger nede) å lage en algoritme som

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

Dersom du prøver dette, vil du fort se at denne løsning er *uakseptabel* – allerede for små verdier av n vil det ta meget lang tid. (Hva er kompleksiteten til denne algoritmen?) Skal man implementere dette problemet, må man, i tillegg, komme opp med en del avkjæringsmekanismer for å redusere søkerommet.

1.2 Representasjon av brettet

Siden vi vet at i en løsning må ingen dronninger stå i samme rad eller samme kolonne, kan vi bruke følgende representasjon. En stilling (av n dronninger) på brettet representeres som en array $int[]$ $p[n]$, der posisjonen $0 \leq r < n$ representerer r-te rad og dens verdi, $p[r] = \mathbf{k}$ er et heltall $0 \leq \mathbf{k} < n$ tilsvarende kolonnen der r-te dronning er plassert. F.eks., for $n = 4$, representerer tabellen p til venstre brettet til høyre:

p :
$p[0] = \mathbf{0}$
$p[1] = \mathbf{3}$
$p[2] = \mathbf{1}$
$p[3] = \mathbf{2}$

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

Gitt n tall $0, 1...n-1$, en *permutasjon* er en oppstilling av disse tallene i en vilkårlig rekkefølge. F.eks., er $p_1 = [0, 3, 1, 2]$ og $p_2 = [2, 1, 3, 0]$ to permutasjoner av tallene $0, 1, 2, 3$. Deretter numererer vi dronningene $0, 1...n-1$, der dronning r, for $0 \leq r < n$, er den som står i r-te rad og – for en gitt permutasjon p – i $p[r]$-te kolonne.

Denne representasjonen sikrer at ingen dronninger truer 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. Den enkle løsningen antydet i 1.1 består i :

1. å generere alle mulige permutasjonene av tallene $0, 1...n-1$, og
2. for hver permutasjon, sjekke om noen dronninger står på samme diagonalen.

Det siste punktet kan sjekkes enkelt ved å observere at to dronninger r_i og r_j i en permutasjon p står på samme diagonalen

- parallel til hoveddiagonalen hviss $p[r_i] - r_i = p[r_j] - r_j$
- parallel til bi-diagonalen hviss $p[r_i] + r_i = p[r_j] + r_j$.

1.3 Avskjæring

Algoritmen over er ineffektiv fordi veldig ofte er det 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-re dronning slår hverandre.

Istedenfor å generere hele permutasjonen *før* den sjekkes, kan man sjekke den *mens* den genereres. Dette er første del av oppgaven:

1. input til programmet: n
2. programmet genererer alle permutasjoner av $0, 1...n-1$
3. og utfører passende sjekk slik at det ikke fortsetter å utforske en permutasjon som ikke kan gi en løsning

4. output av programmet: alle mulige løsningene.

Programmet *skal* være rekursivt!

Ekstra KRAV: For input $n = 10$ skal programmet kjørt på UA bruke så nært som mulig *6.0s user-time*. (Alt over 7.0s er uakseptablet!) Gitt at programmet ditt heter `queen`, skal du kjøre Unix kommando `> /bin/time java queen 10`. I tillegg til `queen 10` 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.)

Før du setter i gang med denne delen, bør du lese oppgaveteksten videre – i andre delen blir det nemlig viktig *i hvilken rekkefølge* permutasjonen genereres.

2 Del II

2.1 Speilinger og roteringer

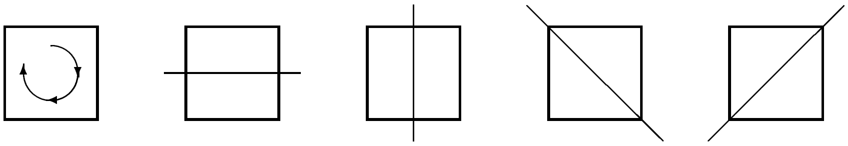
Tenk deg at du har funnet en løsning. Da kan du finne en ny løsning ved å *rotere* brettet 90°. Den nye løsningen er ikke vesensforskjellig fra den opprinnelige fordi brikkenes plassering i forhold til hverandre er uendret. Likevel er det en ny løsning i den forstand at brettet representeres ved ulike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

p_1 :	**0**	**1**	**2**	**3**	**4**
$p_1[0] = \mathbf{0}$	•				
$p_1[1] = \mathbf{2}$			•		
$p_1[2] = \mathbf{4}$					•
$p_1[3] = \mathbf{1}$		•			
$p_1[4] = \mathbf{3}$				•	

$\xRightarrow{90^\circ}$

p_2 :	**0**	**1**	**2**	**3**	**4**
$p_2[0] = \mathbf{4}$					•
$p_2[1] = \mathbf{1}$		•			
$p_2[2] = \mathbf{3}$				•	
$p_2[3] = \mathbf{0}$	•				
$p_2[4] = \mathbf{2}$			•		

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:



rotasjon $90, 180, 270^\circ$ — speiling horisontalt — speiling vertikalt — speiling hoveddiag. — speiling bi-diag.

Følgende tabell viser, for $n = 4, 5...10$ antall forskjellige løsninger (#tot) som skulle finnes i 1.3 (Del I), 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

2.2 Finn kun "vesentlige løsninger"

Du skal nå forfinne programmet fra Del I ved å utelukke 'uvesentlige løsninger'. Med andre ord: hver gang programmet generere en fullstendig løsning, skal det sjekke at denne ikke kan fåes ved noen av avbildningene fra 2.1 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 (Det er naturlig å generere de i en *leksikografisk ordning* der 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 $p_2 = [0, 2, 1, 3, 4]$ har vi $p_1(0) = p_2(0)$ og $p_1(1) = p_2(1)$. Posisjon 2 er den første som skiller disse og, siden $p_1(2) = 4 > 1 = p_2(2)$, har vi at p_2 'er mindre enn' p_1.)
2. for hver løsning L og for hver avbildning rs, konstruer bilde $rs(L)$ av L under rs;
   (F.eks. la aktuell løsning $L = p_2$ i begynnelsen av 2.1, og la rs være 90°-rotasjon, dvs. vi skal se hvilken permutasjon p_1 ville gi p_2 om vi roterte p_1 med 90°. Da må vi se hva vi får om vi roterer p_2 $-90°$. Svaret kan konstrueres fra p_2 som en $p_1 = rs_{90}(p_2)$ ved:

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

   Du skal finne tilsvarende definisjoner for alle avbildningene.
3. sjekk om, for noen rs, er $rs(L)$ 'mindre enn' L – i så fall skal ikke L være med i løsningsmengden.
   (F.eks. p_1 som vi fikk i eksemplet over er den samme p_1 som i begynnelsen av 2.1. Siden $p_1[0] = 0 < 4 = p_2[0]$, er $p_1 < p_2$ (dersom permutasjoner genereres i den leksikografiske ordningen) og p_2 er ikke med i den reduserte løsningsmengden.)[2]

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

[2] Faktisk, så 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 $p_1 = rs_{90}(p_2)$.

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

$$\begin{array}{rl} 1: & [0,4,7,5,2,6,1,3] \\ 2: & [0,5,7,2,6,3,1,4] \\ 3: & [1,3,5,7,2,0,6,4] \\ 4: & [1,4,6,0,2,7,5,3] \\ 5: & [1,4,6,3,0,7,5,2] \\ 6: & [1,5,0,6,3,7,2,4] \\ \ldots: & \textit{osv.} \end{array}$$

Ekstra KRAV: For input $n = 10$ skal programmet bruke så nært som mulig *1.1s user-time.* (Alt over 2.0s er uakseptabelt!)