IN1000 Oblig 8: Fotballsimulator

(Sist oppdatert: 20. April 2023 kl. 15:50)

Introduksjon

Du har fått i oppdrag å gjøre ferdig et program som skal simulere fotballsesonger. Med utgangspunkt i hvor mange mål hvert lagt slipper inn og scorer mot et gjennomsnittlig lag, skal programmet ditt regne ut hvor stor sannsynlig det er at hvert lag havner på hver plass på tabellen. Programmet ditt trenger ikke være realistisk, men det må involvere tilfeldige utfall av hver kamp, slik at det ikke blir de samme resultatene hver gang.

Du overtar programmet etter Ada, en annen programmerer, som har laget noen av klassene ferdig, og noen halvferdig. Noen av klassene fikk ikke Ada begynt på, så de må du lage helt selv.

Ada sender deg en beskjed: «Dette oppdraget er litt annerledes enn du er vant med. Her får du ikke forklart alt du skal gjøre i detalj, men må selv finne ut hvordan du vil løse en del ting. Det krever at du eksperimenterer litt og skaffer deg god oversikt over klassene etter hvert som du trenger dem. Det er også flere klasser enn du er vant til. Jeg er klar over at dette kan oppleves litt overveldende til å begynne med, men husk at dette på ingen måte må bli en realistisk simulering. Det viktigste er at det du leverer virker, så kan vi heller gjøre den mer realistisk senere. Ikke vær redd for å gjøre feil, men prøv å ha det litt gøy med at du selv kan løse mange deler av oppgaven på den måten du liker. Og pass nå på at du ikke blir sittende oppe hele natta for å få det perfekt – det viktigste med dette er det du lærer av det. Lykke til!»

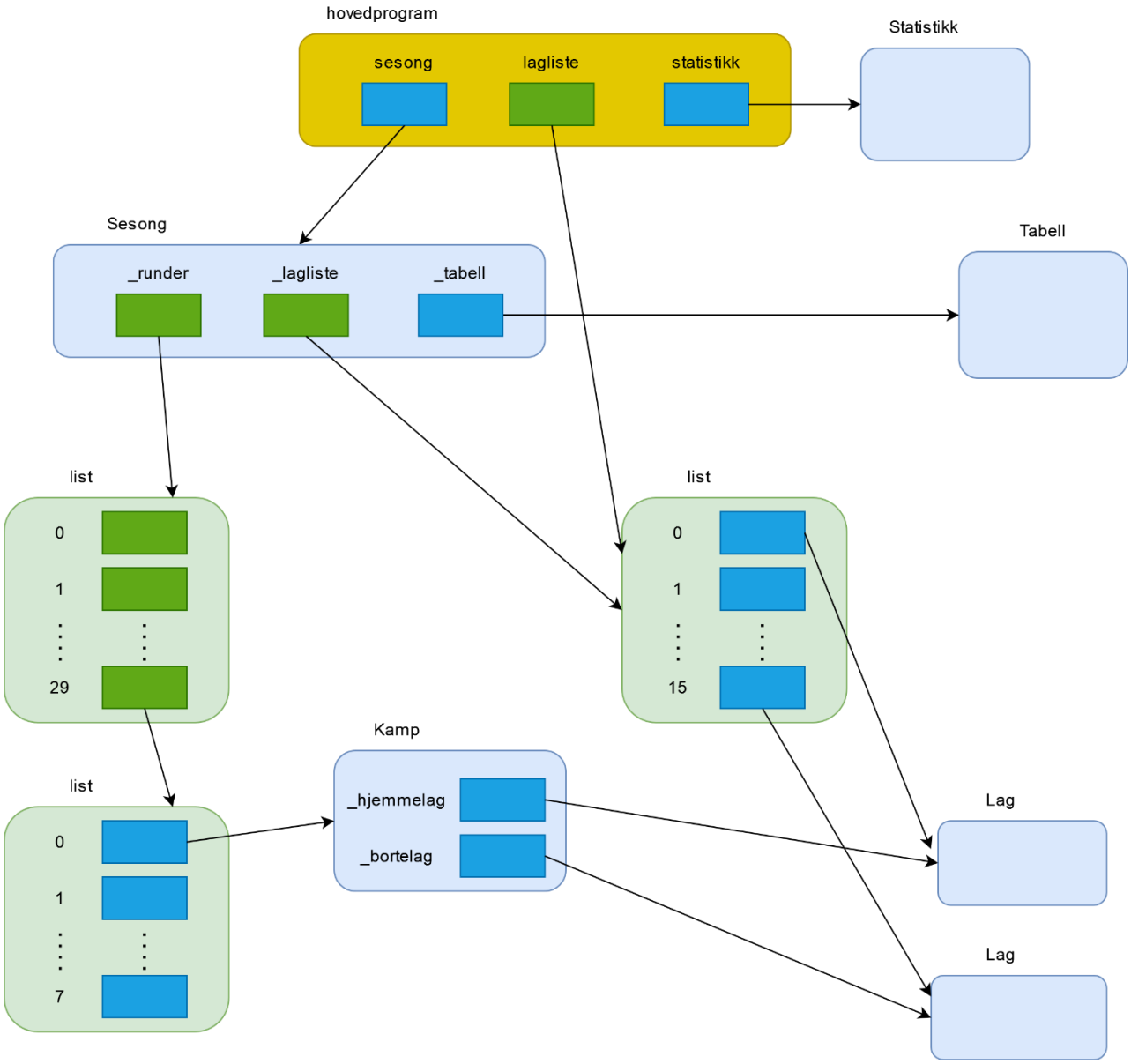
Litt senere tikker det inn enda en beskjed: «PS! Hvis du synes det er vanskelig, kom på oblig-laben for å få ideer. 😊»

Struktur

Programmet består av fem klasser og et "hovedprogram". Følgende filer skal leveres (og et kodeskjelett som inkluderer alle disse filene er lagt ved oppgaven):

hovedprogram.py	Hovedprogrammet som skal simulere 10 000 sesonger og telle antall plasseringer (1. plass, 2.plass, osv.) hvert lag oppnår i løpet av disse simuleringene
statistikk.py	Hjelpeklasse som inneholder metoder og variabler til å lagre resultatene fra hver sesong **(her trenger du ikke gjøre noen endringer)**
lagliste.py	Hjelpefil som definerer listen over lag som deltar ved hjelp av Lag-klassen du skal implementere **(her trenger du ikke gjøre noen endringer,** men når programmet ditt kjører som du skal, kan du godt eksperimentere med å endre på hvor gode lagene er, hvis du har lyst)
tabell.py	Hjelpeklasse som representerer tabellen over lagene – hvilken rekkefølge kommer de i, hvor mange poeng har de, hvor mange mål har de scoret, osv. **(her trenger du ikke gjøre noen endringer)**
sesong.py	Klasse som representerer en sesong (30 runder med 8 kamper i hver runde)
kamp.py	Klasse som representerer en enkelt fotballkamp mellom to lag, som du skal lage selv
lag.py	Klasse som representerer hvert av de 16 lagene, som du skal lage selv

Figur 1 på neste side viser noen av instansvariablene til objektene og hvilke andre objekter enkelte av dem refererer til. Dette er en forenklet framstilling som ikke inneholder alle detaljene, men som Ada har laget for at du skal få oversikt over hvordan referansene mellom objektene er tenkt til å fungere i det ferdige programmet ditt.



Figur 1: Koblinger mellom ulike typer objekter i programmet. Hovedprogrammet er vist med gul farge, lister med grønn farge, og hjemmelagde objekter med blå farge. Legg merke til at lister kan inneholde mange objekter selv om bare ett av dem er vist i figuren. Det finnes også flere andre instansvariabler i objektene enn de som er vist her.

Oppgave 1: Lag

Filnavn: *lag.py*

Ada har rådet deg til å begynne «på bunnen» i Figur 1, med en av de enkleste klassene. Denne er ikke laget, så du skal implementere den selv. Imidlertid har Ada allerede brukt Lag-objekter andre steder i programmet (se for eksempel *lagliste.py*), så denne klassen må lages på en bestemt måte, spesifisert av grensesnittet i tabellen under[1]. Hvilke instansvariabler som skal finnes for denne klassen er opp til deg – bruk de du mener er fornuftige ut fra grensesnittet under:

Metodenavn	Parametre	Returverdier	Kommentarer
__init__	navn: str mål_for: float mål_mot: float	-	Konstruktør. Gir det nye objektet et navn og informasjon om hvor mye mål de scorer og slipper inn mot et gjennomsnittlig lag.
navn	-	str	Henter navnet til laget.
mål_for	-	float	Henter hvor mange mål de bruker å score (høyere = bedre)
mål_mot	-	float	Henter hvor mange mål de bruker å slippe inn (lavere = bedre)
__gt__	other: Lag	bool	Magisk metode som kalles når vi sorterer Lag-objekter alfabetisk. Skal returnere **True** dersom *navnet* til laget er > navnet til et annet lag (other), og **False** ellers
__lt__	other: Lag	bool	Samme som __gt__, men med < isteden

(HINT: I konstruktøren må du velge fornuftige navn til objektenes instansvariabler og tilordne verdiene de skal ha. Ellers blir det vanskelig for de andre metodene å returnere verdiene de skal.)

Oppgave 2: Kamp

Filnavn: *kamp.py*

Neste steg er å lage en klasse som skal simulere én fotballkamp mellom to Lag-objekter. Som i oppgave 1 er grensesnittet gitt, og du finner det på neste side:

[1] Legg merke til at **self** ikke er tatt med blant parametrene, fordi alle metoder inkluderer denne parameteren per definisjon.

Metodenavn	Parametre	Returverdier	Kommentarer
__init__	hjemmelag: Lag bortelag: Lag	-	Konstruktør. Forteller objektet hvilke Lag som spiller
hjemmelag	-	Lag	Referanse til hjemmelaget
bortelag	-	Lag	Referanse til bortelaget
spill	-	-	Simulerer en kamp, slik at den får et resultat (se nedenfor)
mål_hjemme	-	int (eller None)	Antall mål til hjemmelaget (None dersom kampen ikke er spilt enda)
mål_borte	-	int (eller None)	Antall mål til bortelaget (None dersom kampen ikke er spilt enda)

Så til det store spørsmålet – hvordan skal du implementere **spill**-metoden?

- Du må ha med følgende:
    - Et element av tilfeldighet, slik at samme kamp kan få forskjellig resultat dersom vi simulerer flere sesonger etter hverandre. Hvordan du gjør dette, er opp til deg, men her er noen funksjoner importert fra *random*-pakken som kan være nyttige:
        - *randint(minste, største):* returnerer et tilfeldig heltall et sted fra og med *minste* til og med *største*
        - *choice(liste):* returnerer et tilfeldig valgt element fra *liste*
        - *random():* returnerer et tilfeldig flyttall mellom 0 og 1

- Det er i tillegg en fordel (men ikke påkrevd) dersom metoden tar hensyn til følgende:
    - Hvor gode lagene er til å score mål og hvor «gode» de er til å slippe inn mål (denne informasjonen har Lag-klassen fra oppgave 1 metoder som gir deg)
    - Hjemmelaget har en fordel av å spille på hjemmebane – hvis du tar med dette, må du selv finne ut hvordan du skal modellere denne fordelen og hvor stor den skal være.
    - Et realistisk antall mål – etter en spilt sesong (dvs. ved å kjøre *sesong.py* som inneholder testkode for dette), kan Tabell-objektet skrive ut hvor mange mål som ble scoret per kamp i snitt. For 2022-sesongen var dette 3.25 mål – hvis du får noe i nærheten av dette hver gang, har du en ganske realistisk simulering. (OBS: Dette får du ikke testet før du har gjort ferdig oppgave 3 og 4, så du kan vende tilbake til dette punktet da, om du har tid og lyst.)

Oppgave 3: Sesong (oppsett av kamper)

Filnavn: *sesong.py*

Denne klassen er halvferdig – her er det to ting du må gjøre. Den første er å skrive ferdig konstruktøren slik at alle lagene får en hjemmekamp og en bortekamp mot hverandre. Her kan du bruke *round-robin-algoritmen,* som fungerer slik:

Round-robin-algoritmen

For å forstå hvordan denne algoritmen fungerer, se først etter et mønster i figurene under. Vi har delt listen med lag i to – *lagliste1* og *lagliste2* – slik at ett lag fra hver liste møter ett lag fra den andre listen hver runde.

Runde 1 (og 16): Lag A møter Lag P, Lag B møter Lag O, osv.

indeks	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
lagliste1	Lag A	Lag B	Lag C	Lag D	Lag E	Lag F	Lag G	Lag H
lagliste2	Lag P	Lag O	Lag N	Lag M	Lag L	Lag K	Lag J	Lag I

Runde 2 (og 17): Lag A møter Lag O, Lag P møter Lag N, osv.

Indeks	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
lagliste1	Lag A	Lag P	Lag B	Lag C	Lag D	Lag E	Lag F	Lag G
lagliste2	Lag O	Lag N	Lag M	Lag L	Lag K	Lag J	Lag I	Lag H

(...og så videre, helt til vi nesten er tilbake ved utgangspunktet igjen...)

Runde 15 (og 30): Lag A møter Lag B, Lag C møter Lag P, osv.

Indeks	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
lagliste1	Lag A	Lag C	Lag D	Lag E	Lag F	Lag G	Lag H	Lag I
lagliste2	Lag B	Lag P	Lag O	Lag N	Lag M	Lag L	Lag K	Lag J

Ser du mønsteret? Spesielt kan det være nyttig å se på runde 15, 16 og 17 i den rekkefølgen. Legg merke til:

- Hvor i listene befinner lagene seg som blir flyttet til motsatt liste i neste runde?
- Hvor i listen befinner det ene laget som aldri bytter posisjon seg?

Din jobb i konstruktøren er å sette opp alle rundene i sesongen. Som vist i Figur 1, er instansvariabelen *self._runder* en liste hvor elementene er lister med *Kamp*-objekter som du må lage (30 runder med 8 kamper hver).
(HINT: Metoden *pop* kan være til hjelp når du skal flytte et element fra en liste til en annen.)

Hjemme- og bortekamper

Pass til slutt på å bytte om hvilen av de to listene som får hjemmekamp annenhver gang, slik ikke samme lag får mange hjemmekamper på rad (innimellom må enkelte lag få to hjemmekamper eller bortekamper på rad – det skjer når laget flyttes fra en liste til den andre, og det er OK).

Du må sjekke at alle lagene møter hverandre to ganger, og at hjemmelaget i første kamp alltid er bortelaget i siste kamp. Det finnes allerede testkode i klassen *Sesong* som du kan kjøre for å teste metodene dine.

Oppgave 4: Sesong (spill kampene)

Filnavn: *sesong.py*

I konstruktøren satte du opp rundene, men kampene er ikke spilt enda. Dette er en jobb for metoden *simuler,* som skal simulere hele sesongen, dvs. spille alle kampene, runde for runde. Her bruker du blant annet ting du laget i *Kamp*-klassen til å faktisk spille alle kampene du satte opp i konstruktøren.

Når en kamp er spilt, sørg for at resultatet av kampen legges til i *Tabell*-objektet slik at tabellen tar med resultatet. Her må du se på metodene i *Tabell*-klassen for å finne en metode som gjør det du ønsker å få til.

Kjør til slutt testkoden – hvis alt nå virker som det skal, skal du få en utskrift i terminalen som ligner dette:

```
Runde 29
--------

Bodø/Glimt - Rosenborg         2 - 0
Vålerenga - Molde              1 - 1
Lillestrøm - Brann             1 - 1
Odd - Tromsø                   1 - 0
Viking - Haugesund             1 - 0
Sarpsborg 08 - Strømsgodset    0 - 0
Sandefjord - Aalesund          6 - 2
Stabæk - HamKam                1 - 1

Runde 30
--------

Molde - Bodø/Glimt             2 - 4
Brann - Rosenborg              0 - 2
Tromsø - Vålerenga             0 - 1
Haugesund - Lillestrøm         2 - 0
Strømsgodset - Odd             1 - 1
Aalesund - Viking              0 - 0
HamKam - Sarpsborg 08          3 - 1
Stabæk - Sandefjord            2 - 0


Bodø/Glimt     30 21 2  7   75 - 38  65
Molde          30 16 6  8   71 - 44  54
Viking         30 15 5  10  51 - 39  50
Odd            30 15 4  11  54 - 47  49
Stabæk         30 13 8  9   47 - 39  47
Strømsgodset   30 12 7  11  48 - 42  43
Vålerenga      30 11 9  10  37 - 38  42
HamKam         30 13 2  15  45 - 52  41
Haugesund      30 12 3  15  39 - 51  39
Sandefjord     30 10 8  12  46 - 51  38
Rosenborg      30 10 7  13  38 - 51  37
Tromsø         30 10 6  14  51 - 52  36
Brann          30 10 6  14  39 - 54  36
Sarpsborg 08   30 9  7  14  43 - 52  34
Aalesund       30 8  10 12  37 - 48  34
```

Figur 2: Eksempel-utskrift fra å kjøre sesong.py

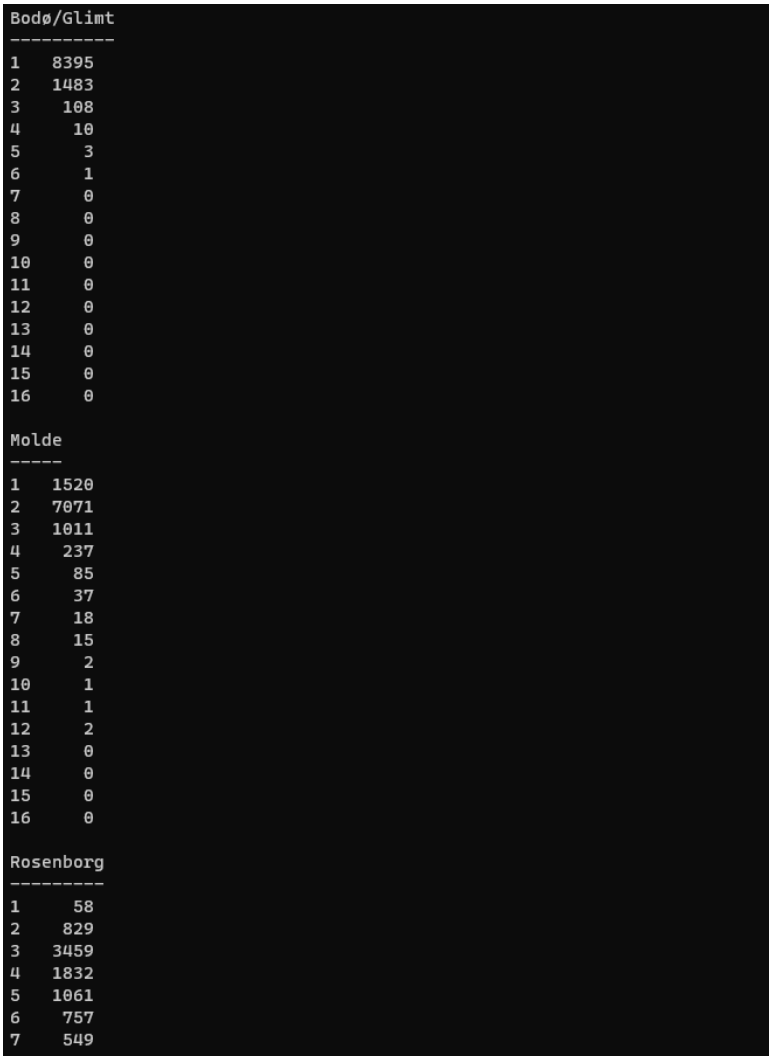
Oppgave 5: Hovedprogram

Filnavn: *hovedprogram.py*

Til slutt skal vi gjøre ferdig hovedprogrammet. Dersom alt virker som det skal i *Sesong*-klassen sin testkode, kan vi nå simulere 10 000 sesonger på rad. Det vil si at du må lage 10 000 *Sesong*-objekter med de samme lagene, og spille alle kampene i hver sesong.

Her må du i tillegg bruke metodene i hjelpeklassen *Statistikk* (som Ada rakk å lage ferdig) til å lagre tabell-resultater fra hver sesong og til å printe ut statistikken til slutt. Vær klar over at det kommer til å ta en god del sekunder å kjøre så mange simuleringer (hvor lang tid det tar kan variere med hvor rask maskinen din er). Du kan begynne med å simulere 10 sesonger mens du tester, så det går litt raskere.

(Det anbefales å gjøre som figur 1 viser og bruke en variabel *sesong* som hele tiden får et nytt *Sesong*-objekt som verdi, så lenge du passer på å lagre resultatene i *Statistikk*-objektet etter at hver sesong er simulert. Men hvis du heller foretrekker å lagre sesongene i en liste eller ordbok, er dette også greit – bare være klar over at denne tilnærmingen bruker mer av minnet til datamaskinen og slik sett er mindre effektiv.)



Figur 3: Eksempel-utskrift fra hovedprogram.py

Husk at resultatene *ikke* trenger å være realistiske. Men det er fordelaktig om antall mål lagene typisk scorer og slipper inn påvirker resultatene i alle fall i noen grad. Hvis du overhodet ikke ser noen forskjell på lagene (alle lagene vinner ca. like ofte), må du endre det du gjorde oppgave 2 (klassen *Kamp*) slik at hvor gode lagene er på en eller annen måte påvirker resultatet. Men ellers kan du gjøre det så realistisk (eller urealistisk) du vil.

Oppsummering: Hva må du gjøre?

Oppgave	Må gjøres	Frivillig
1	• Lage *Lag*-klassen • Implementere alle metodene i grensesnittet	
2	• Lage *Kamp*-klassen • Implementere alle metodene i grensesnittet • Sørge for at kamper mellom samme lag får forskjellig resultat hver gang (dvs. et element av tilfeldighet)	• Ta hensyn til hvor gode lagene er • Ta hensyn til hjemmebane-fordelen • Et realistisk antall mål pr. sesong (kan ikke testes før oppgave 3 og 4 er gjort)
3	• Gjøre ferdig konstruktøren til *Sesong* slik at alle lagene møter hverandre to ganger, en hjemme og en borte	• Lese inn faktiske resultater fra kamper som allerede er spilt (se neste side) • Lese inn den virkelige terminlisten fra fil (ekstra utfordrende; se neste side)
4	• Spill alle kampene i sesongen • Lagre resultatene i *Tabell*-objektet til sesongen	
5	• Simuler 10 000 sesonger på rad • Lagre tabell-resultatene i et *Statistikk*-objekt • Skrive ut statistikken til slutt	• Gjøre simuleringen så realistisk du har tid og lyst til

Frivillig utfordring: Faktiske resultater

Om du har tid og er interessert i å bruke denne modellen til å faktisk forutsi årets resultater (vel å merke dersom vi har gjettet riktig på hvor gode lagene er), kan du putte inn årets resultater herfra: http://rsssf.no/2023/Premier.html - det vil være snakk om 2 – 4 runder med kamper i perioden dere jobber med obligen.

En måte å gjøre dette på, er å lage en ny metode i *Sesong* som går gjennom rundene og legger til det faktiske resultatet når den kjenner igjen en kamp som er spilt (med riktig hjemme- og bortelag). Om du fortsetter å oppdatere resultatene gjennom sesongen, vil det bli en lang metode etter hvert som sesongen spilles, fordi det blir stadig flere ferdigspilte kamper.

(Om du *virkelig* har lyst på en utfording, kan du prøve å lese inn terminlisten fra nettsiden linket til ovenfor slik at kampene også kommer i den riktige rekkefølgen. Dette krever imidlertid avansert bearbeiding av tekst lest fra fil, og går såpass langt ut over det vi forventer folk skal gjøre, at vi nøyer oss med å nevne muligheten for spesielt interesserte. Du må i så fall fortsatt levere kode for round-robin-metoden fra oppgave 3, i tillegg til koden som leser fra fil.)

Utover i sesongen kan du også eventuelt oppdatere hvor gode lagene er dersom du synes at våre tidlige gjetninger ikke lenger er realistiske.