Spiludvikling - Aflevering

JSHA og NVF

December/Januar 2025

I dette projekt skal I bygge et 4-på-stribe spil.

Fokus for projektet er algoritmer – I skal altså øve jer på at gå fra et regelsæt til kode.

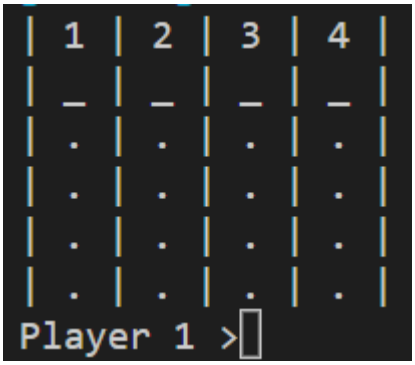
1 Hvad skal afleveres?

I skal arbejde sammen i grupper på 2-3 personer (grupperne bestemmer i selv). Hver grupe skal aflevere en zip-fil der indeholder:

- Jeres program (som .py) eller et link til jeres Github (hvor jeg og Nina er inviteret på jsha@tec.dk og nvf@tec.dk)
- Et flow-diagram til hver algoritme
- En rapport (på maks 2 sider) der indeholder:
  - En forklaring af de algoritmer der blev implementeret i jeres program
  - En forklaring af hvilke fejl i stødte på under processen, og hvordan i løste disse fejl. Svar på hvad man skal være særligt opmærksom på, når man skriver et fire-på-stribe spil.

2 Jeres program

Med udgangspunkt i det udleverede program `fire-paa-stribe.py`, skal i udvikle fire-på-stribe. For at gøre dette skal i følge denne opgave nøje. Det program jeg har givet jer, printer spilbrættet således:



Der er lavet et gameloop, hvor det er meningen at brugeren skal kunne give input 1, 2, 3 eller 4. For at få spillet til at fungere skal i implementere følgende algoritme:

1. Spilleren vælger hvilken kolonne der skal placeres en markør i.
2. Markøren ligger sig i bunden af den valgte kolonne, medmindre der allerede er en markør, så ligger markøren sig på pladsen over markøren.
3. Spiller 1 og Spiller 2 skiftes til at ligge en markør
4. Hvis der er 4 markører på stribe i vandret, lodret eller diagonal retning så vinder spilleren der lagde den 4. markør. Programmet skal gøre opmærksom på hvem der vinder, når der er en vinder.
5. Hvis der ikke er flere steder at placere markører, skal spillet gøre opmærksom på uafgjort.

3 Opgaver

1. Nærstuder det udleverede program

Start med at undersøge programmets som i har fået udleveret. Resten af algoritmen er nemmere at implementere hvis man forstår datastrukturen som hele programmet er bygget op omkring:

```
board =     [[empty, empty, empty, empty],
             [empty, empty, empty, empty],
             [empty, empty, empty, empty],
             [empty, empty, empty, empty]]
```

Strukturen er en `nested list`. Det vil sige en liste, med flere lister. Datastrukturen kan benyttes som en multidimensionel liste, også kendt som en tabel. Man kan pege på elementerne i den multidimensionelle liste som man peger på elementer i en normal liste. `board[0][0]` vil f.eks. pege på første element, fra første liste. `board[0][3]` vil pege på sidste element, fra første liste. `board[3][0]` vil pege på første element fra sidste liste osv.

Lav et forsøg hvor du sætter en af de værdier der hedder `empty` til `player1`. Skriv et print()-statement i programmet, der printer player1 ved hjælp af board-variablen.

2. At ligge markører i bunden

Jeg har lavet en funktion til jer `mark(col, player)`, som skal benyttes til at placere en markør. Funktionens parametre er `col` og `player`. `col` beskriver hvilken kolonne brugeren vil ligge sin markør i. `player` beskriver hvilken spiller der er igang med at smide en markør.

Det er jeres opgave at lave en funktion der opdatere `board`-variablen korrekt. Hvis man som spiller 1 giver input `2` betyder det at man vil ligge sin markør i kolonne 2, derfor skal der ligge en markør i `board[3][1]`, efter at funktionen er kørt. Det er vigtigt at funktionen tager højde for hvorvidt der allerede ligger en markør. Hvis der allerede ligger en markør, skal markøren placeres ovenpå markøren der ligger i vejen.

`HUSK AT LISTER TÆLLER FRA 0 OG FOR-LOOPS ER SMARTE TIL AT GENNEMGÅ LISTER`

I skal først gå til næste step når i har noget der ligner billedet nedenunder ved at give 4 inputs (1,2,2,3):

3. Multiplayer

Vi skal have 2 spillere i vores spil, derfor skal i finde ud af hvordan man kan skifte mellem markørerne, efter hvert træk.

`TIP 1: Brug en boolsk variabel til at skifte mellem spillerne. TIP 2: I skal give spiller-markøren som argument til mark()-funktionen`

4. Tjek om der er en vinder

Hver gang at der bliver lagt en brik, skal der tjekkes om spilleren har vundet. Det gøres ved at tjekke om der er fire vertikalt, horisontalt eller diagonalt forbundede markører.

Start med at løs hvordan i tjekker den horisontale retning, derefter den vertikale og til sidst de diagonale.

TIP: hvis sætningerne nedenunder er sande, vil man have en horisontal sejr;

```
[board[0][0]=="0",
 board[0][1]=="0",
 board[0][2]=="0",
 board[0][3]=="0"]
```

Det er meget brugbart at benytte for-loops ala:

```
def check_for_win(player):
    for col in range(len(board[0])):
        for row in range(len(board)):
            if board[col][row] == player:
                return True
```

På den måde kan man tjekke flere kollonner og rækker med samme logik. Logikken skal i dog selv regne ud. I eksempel-algoritmen vinder spilleren, ved at ligge en markør et hvilket som helst sted på brættet.

6. Tjek for uafgjort

Algoritmen skal også tjekke om der er uafgjort. Der er uafgjort, når der ikke er plads til en markør.

5. Udvid dit grid til 7x6

Opgaven er at udvide spillet til at have flere felter. Test om spillet stadig virker. Hvis det ikke gør, skal i undersøge hvorfor. Hvis det stadig virker, så kan i forsøge at implementere at Spiller 2 styres af computeren. Her gælder det selvfølgelig om at lave den sværeste Spiller 2.