Sten, sax, påse

Programmering II - Elixir

Kenan Dizdarevic

13 Februari, 2023

Inledning

Vi skall i denna uppgift lösa ett av problemen som *Advent of Code* erbjuder. Problemet vi skall lösa kommer från dag 2.

Älvorna har satt upp ett läger vid stranden. För att bestämma vem som skall vara närmast matförrådet anordnar de en **sten, sax, påse**-turnering.

Sten, sax, påse är ett spel som spelas mellan två deltagare. Varje spelare skall antingen välja sten, sax eller påse. Sten vinner över sax, påse vinner över sten och sax vinner över påse. Om båda spelarna väljer samma alternativ innebär det att matchen är oavgjord.

Implementering

Problemformulering

Som tack för att vi hjälpte älvorna med maten igår ger oss en älva en krypterad strategiguide som garanterar vinst.

Första kolumnen beskriver vad vår motståndare kommer att välja. Om det exempelvis står `A`, kommer motståndaren att välja sten. `B` representerar påse och `C` representerar sax. Den andra kolumnen beskriver vad vi skall välja. `X` representerar sten, `Y` är påse och `Z` är sax. Om vi hade vunnit varje match hade det varit suspekt. Alternativen är därför noga utvalda för att inte väcka misstanke.

Vinnaren av turneringen är den med högst poäng. Poängen man erhåller för varje runda är olika, poängen ackumuleras. Poängen baseras på om man vinner eller ej samt vilket alternativ man valde. Man erhåller 0 poäng vid förlust, 3 poäng vid oavgjort och 6 poäng vid vinst. Val av sten ger 1 poäng, påse ger 2 poäng och sax ger 3 poäng.

Vi vet dock inte om älvan försöker lura oss. Vi skall därför beräkna den totala poängen som älvans guide kommer ge oss.

Representation

Filen som innehåller alla matcher är en text-fil. Vi behöver läsa in den och dela upp den på ett adekvat sätt. Vi behöver även införa en representation för matcherna samt hur vi beräknar poängen från dem.

Vi har valt att implementera poängen som en *nyckel-värde databas* med hjälp av Elixirs egna `Map()`.

```
poäng = %{{"A", "X"} => 4, {"A", "Y"} => 8, {"A", "Z"} => 3,
          {"B", "X"} => 1, {"B", "Y"} => 5, {"B", "Z"} => 9,
          {"C", "X"} => 7, {"C", "Y"} => 2, {"C", "Z"} => 6}
```

Detta innebär om motståndaren väljer `A` och vi väljer `X` skall vi ha 4 poäng totalt, 1 poäng för sten och 3 poäng för oavgjort. Vi representerar alltså en match som en tupel med båda val.

Vi har definierat alternativen och mappat dem till poängen som vi skall ha enligt spelreglerna.

Indata

Vår nyckel-värde databas har givit oss en stabil grund att stå på. Vi har en text-fil med matcher som exempelvis ser ut som följande:
A X
B Y
C Z
Vi behöver således dela upp filen i enskilda matcher. Sedan skall varje match delas upp i två strängar med var sitt alternativ. Detta innebär att vi har möjlighet att utföra mösntermatchning mot vår databas och erhålla korrekta poäng för varje match. Vi skall sedan summera varje match och returnera totala poängen. Koden för att läsa in filen ser ut som följande:

```
{:ok, games} = File.read("C:/Users/cooltNamn/krokigVäg/games.txt")
```

Vi väljer att returnera indatan som en tupel där första elementet är `:ok`, vid en lyckad inläsning. Detta innebär att vi enklare kan hantera de fall där det uppstår problem med inläsningen.

Beräkning

Det som återstår är att formatera datan så att vi kan utföra mönstermatchning med vår nyckel-värde databas och summera alla poäng. Koden för detta ser ut som följande:

```
matches = String.split(games, "\r\n")
score = matches
|> Enum.map(fn match -> String.split(match, " ") end)
|> Enum.map(fn [opponent, me] -> points[{opponent, me}] end)
|> Enum.sum
```

I variabeln `matches` sparar vi innehållet från alla matcher men det är modifierat med hjälp av `String.split/2`. Varje sträng är således separerad där radbrytningen \r\n sker.

Vi använder oss av *pipeline-operatorn* för att koppla samman en serie av funktioner. Den första funktionen i pipelinen använder sig återigen av `String.split/2` på varje element i listan. Denna funktion returnerar en ny lista där varje innerlista består av två element. Vi har således formaterat datan till den korrekta formen. Nästa funktion i pipelinen använder sig av mönstermatchning. Den mappar varje match mot en tupel i vår nyckel-värde databas. Den sista funktionen i pipelinen summerar alla värden i den resulterande listan och returnerar totala poängen.

Vi skulle med hjälpa av älvans guide ha 12 156 poäng i slutet av turneringen.

Justering

Det visar sig att den andra kolumnen inte är det som vi bör välja. Utan det beskriver vad utfallet av matchen skall vara. `X` betyder att vi bör förlora, `Y` betyder att vi bör spela oavgjort och `Z` betyder att vi skall vinna.

Vi behöver inte modifiera koden särskilt mycket. Det enda vi bör göra är att implementera en ny nyckel-värde databas med korrekt tolkning. Den nya databasen ser ut som följande:

```
poäng = %{{"A", "X"} => 3, {"A", "Y"} => 4, {"A", "Z"} => 8,
          {"B", "X"} => 1, {"B", "Y"} => 5, {"B", "Z"} => 9,
          {"C", "X"} => 2, {"C", "Y"} => 6, {"C", "Z"} => 7}
```

Om vår motståndare väljer sten och vi skall förlora, då skall vi välja sax. Vi får 0 poäng vid en förlust och 3 poäng för val av sax. Om vår motståndare istället väljer påse och vi skall spela oavgjort, då bör vi också välja påse. Detta innebär att vi erhåller 3 poäng för oavgjort och 2 poäng för val av påse, alltså totalt 5 poäng. När vi har gått igenom varje fall och korrigerat poängen anropar vi vår funktion som läser in filen med den nya databasen. Denna gång skulle vi ha 10 835 poäng vid turneringens slut.

Slutsats

Slutligen ser vi att det är relativt simpelt att beräkna den totala poängen för båda fall. Uppgiften är inte svår att lösa tankemässigt, det kan alla klura ut. Problemet med denna uppgift är att välja en adekvat representation av matcherna och poängen.

Ett annat problem i denna uppgift är inläsningen av alla matcher. Detta skall gå hand i hand med vår implementering av nyckel-värde databasen och man behöver därför läsa in filen korrekt. Det är vitalt att dela upp datan i de beståndsdelar som krävs för att kunna använda mönstermatchning senare.