# Tågväxling

**Programmering II**

Axel Karlsson

VT22 25 februari 2022

## Inledning

I denna rapport ska jag redovisa min lösning för inlämningsuppgift 10: train shunting i programmering II. Uppgiften gick ut på att lösa ett problem där vagnar ska sorteras i en bangård, och hur man kan modellera ett sådant problem i elixir. Jag har löst problemet på tre sätt med rekursiva algoritmer, där två är snarlika och den tredje är något mer sofistikerad.

## Modellering av problemet

Målet är att utifrån ett givet tillstånd i en bangård konstruera ett givet tåg. En bangård består av tre spår: `main`, `one` och `two`. En vagn representeras av en (unik) atom och bangårdens tillstånd representeras av en tuppel med tre listor, en för varje spår.

```
{[:a, :b], [:c], [:d, :e]}
```

Figur 1: Exempel på ett tillstånd

I 1 visas ett exempel på hur en bangård representeras i vår modell. Listorna representerar från höger till vänster `main`, `one` respektive `two`.

Vagnar kan flyttas mellan spåren på två sätt. Om en vagn flyttas från `main` till `one` eller `two` flyttas vagnen längst bak på spåret. Om en vagn flyttas från `one` eller `two` till `main` flyttas den främsta vagnen (om man funderar på hur detta skulle fungera i verkligheten kan man förstå varför).

```
{[:a, :b], [], []} -> {[:a], [:b], []}
{[], [:c, :d], []} -> {[:c], [:d], []}
```

Figur 2: Hur vagnar kan flyttas mellan spåren.

I 2 visas exempel på hur vagnar flyttas mellan de olika spåren. I det första exemplet flyttas vagn `:b` från `main` till `one`. I det andra exemplet flyttas en vagn från `one` till `main`.

En förflyttning beskrivs av antal vagnar som flyttas mellan `one` eller `two` och `main`. Detta representeras som en tuppel med två element, en atom och en siffra. Om siffran är positiv eller negativ flyttas vagnar från respektive till `main`.

```
{:one, 1}
{:one, -1}
```

Figur 3: Hur en förflyttning representeras.

I 3 ser vi hur förflyttningarna i 2 skulle beskrivas i programmet.

## Hjälpfunktioner

Ett flertal hjälpfunktioner implementerades i modulerna `Lists`, `Moves` och `Shunt` för att skapa en grund varpå en algoritm som löser problemet kunde byggas. Den mest intressanta av dessa är nog `single/2`.

### Single

Denna funktion skulle ta en förflyttning och ett tillstånd för en bangård som argument, och sedan returnera ett uppdaterat tillstånd där förflyttningen har utförts.
Först definierade jag funktionen `single/2` som tar de tidigare nämnda argumenten.

```
def single({track, num}, {main, t1, t2}) do
  case track do
    :one ->
      {main, t1} = single(num, main, t1) # Till/från one
      {main, t1, t2}
    :two ->
      {main, t2} = ... # Till/från two
  end
end
```

Denna funktion anropar sedan `single/3`, som utför de uträkningar som krävs för att flytta rätt antal vagnar och returnerar två uppdaterade spår, `main` och `one` eller `two`.

```
def single(num, main, track) do
  case num < 0 do
```

```
      true -> # Flytta från track till main
        num = num*-1
        # Modifiera listorna (spåren) med Lists modulen
        {main, train}
      false -> # Flytta main till track
        # Lists modulen igen, returnera 2 spår
    end
  end
```

I fallet `true` måste `num` måste multipliceras med `-1` eftersom vi i den exkluderade koden arbetar med listor, som självklart inte har negativa index.

## Algoritmen

Jag implementerade tre olika lösningar på problemet där den första och andra är väldigt lika. Alla tre bygger på samma princip. De går igenom alla vagnar i ett givet tåg en och en, och för varje vagn hittar de förflyttningar som i en given bangård placerar vagnen rätt plats. Dessa förflyttningar genereras utifrån en förutbestämd mall som fylls i med hjälp av `split/2`.

### Find

Den första versionen av algoritmen implementerades med funktionen `find/2`. Funktionen tar två tåg som argument, `tbeg` och `tgoal`, och returnerar en sekvens förflyttningar som transformerar bangården {`tbeg, [], []`} till {`tgoal, [], []`}.
Jag löste problemet genom att definiera en funktion `find/3` som även tar en ackumulatorlista som argument. Algoritmen går sedan igenom varje vagn i `tgoal` och "fyller i" värden i en lista av förflyttningar, som sedan läggs till i ackumulatorlistan.

```
  def find(_, [], moves) do moves end # Ingen vagn kvar i tgoal
  def find(tcurr, [gh | gt], moves) do
    {ch, ct} = split(tcurr, gh) # Antal vagnar framför och bakom gh
    # Mallen
    this_moves = [
      {:one, length(ct) + 1},
      {:two, length(ch)},
      {:one, length(ct)*-1 - 1},
      {:two, length(ch)*-1}
    ]
    tcurr = Lists.append(ch, ct) # Det nya "main" spåret
    moves = Lists.append(moves, this_moves)
    find(tcurr, gt, moves)
  end
```

### Few

Denna lösning är identisk till `find/2` förutom en förändring, en ny klausul som matchar om de första vagnarna i `tcurr` och `tgoal` är samma. I det fallet är vagnen redan på rätt plats och ingen förflyttning behövs.

```
# Den nya klausulen
def few([h | ct], [h | gt], moves) do few(ct, gt, moves) end
```

### Fewer

I denna lösning utökades informationen som algoritmen har tillgång till. Den implementerades i funktionen `fewer/4` som tar tre spår (`ms`, `os` och `ts`) samt ett tåg `[y | ys]` som argument. Till skillnad från de tidigare lösningarna kan vi alltså utnyttja det faktum att vagnarna i bangården kan ligga på tre spår. Istället för att alltid använda samma mall för `this_moves` som i `find/2` och `fewer/2` finns det nu olika mallar beroende på vilket spår vagnen `y` ligger på.

```
def fewer(ms, os, ts, [y | ys], moves) do
  cond do
  List.member(ms, y) -> # y är på main
    ...
    this_moves = [ ... ]
    ...
  List.member(os, y) -> # y är på one
    ...
    this_moves = [ ... ]
    ...
  List.member(ts, y) -> # y är på two
    ...
    this_moves = [ ... ]
    ...
  end
end
```

## Reflektion

Denna uppgift kändes som en skön omväxling efter att ha krigat med “Kapa brädor” uppgiften tidigare i veckan. En sak som slog mig när jag först gjorde uppgiften, men som är ganska uppenbart i efterhand, är att det är intressant hur våra lösningar blir bättre (färre förflyttningar) ju mer information vi låter algoritmen arbeta med. Jag undrar om det hade gått att skriva en ännu bättre algoritm som kollar på ännu fler parametrar, kanske ordningen av vagnarna på spåren. Jag funderade också på om man hade kunnat “brute

force-a" lösningen med dynamisk programmering (som i kapa brädor), men det får nog vänta tills en annan dag.