Träd vs lista

Iley Alvarez Funcke

Vårterminen 2022

Introduktion

Målet med denna uppgift är att använda dynamisk programmering i Elixir för att kapa brädor på ett så kosteffektivt sätt som möjligt. En bräda ska kapas utifrån en lista med specifika längder där varje kapning av brädan kostar en viss summa pengar som är proportionell mot brädans längd.

Metod

Metoden som användes för att lösa uppgiften var att metodiskt skapa en funktion i taget i programmet. Vad dessa funktioner skulle göra stod tydligt i uppgifts instruktionerna så det var lätt att veta vad som skulle göras. För vissa delar som till exempel de som relaterade till dynamisk programmering användes Canvas föreläsningarna som grund för hur dessa skulle implementeras eftersom samma principer gällde.

Resultat

Den första funktionen som skapades var en som splittade en sekvens på alla möjliga sätt. Denna funktion användes inte i det slutgiltiga programmet men den la grunden för hur det slutgiltiga programmet skulle fungera.

```
def split(seq) do
   split(seq, 0, [], []) end

 def split([], l, left, right) do
   [{left, right, l}]
 end

 def split([s|rest], l, left, right) do
   split(rest, l + s, [s|left], right) ++ split(rest, l + s, left, [s|right])
 end
```

Det mesta i denna funktion fanns redan i instruktionerna så det finns inte många delar att förklara. Men det som ”split” gör är helt enkelt att ta emot en sekvens ”seq” och sedan lägga det första elementet antingen till höger eller vänster (”left” eller ”right”). Genom att göra detta rekursivt och sluta när ”seq” inte har några fler tal i sig så får man ut alla möjliga sätt en sekvens kan splittas på.

Men den viktigaste delen av programmet skapades inte föränn senare. Denna är den slutgiltiga versionen av ”cost” funktionen. Det är ”cost” funktionen som beräknar den minsta kostnaden för att kapa en bräda på det mest kosteffektiva sättet. Implementationen av ”cost” funktionen syns nedan (där delen som är kommenterad inte användes för den slutgiltiga versionen av cost men den är fortfarande viktig för en tidigare del så jag tog med den).

```
def cost([]) do
  {0, :na}
end

def cost(seq) do
  {cost, tree, _} = cost(seq, Memo.new())
  {cost, tree}
end

def cost([s], mem) do
  {0, s, mem}
end

#def cost(seq, mem) do
#  {c, t, mem} = cost(seq, 0, [], [], mem)
#  {c, t, Memo.add(mem, seq, {c, t})}
#end

def cost([s|rest] = seq, mem) do
  {c, t, mem} = cost(rest, s, [s], [], mem)
  {c, t, Memo.add(mem, seq, {c, t})}
end

def check(seq, mem) do
  case Memo.lookup(mem, seq) do
    nil ->
      cost(seq, mem)
    {c, t} ->
      {c, t, mem}
  end
```

```
end

def cost([], l, left, right, mem) do
  {costLeft, newLeft, mem} = check(left, mem)
  {costRight, newRight, mem} = check(right, mem)
  {l + costLeft + costRight, {newLeft, newRight}, mem}
end

def cost([s], l, [], right, mem) do
  {costRight, newRight, mem} = check(right, mem)
  {s + l + costRight, {s, newRight}, mem}
end

def cost([s], l, left, [], mem) do
  {costLeft, newLeft, mem} = check(left, mem)
  {s + l + costLeft, {newLeft, s}, mem}
end

def cost([s|rest], l, left, right, mem) do
  {toLeft, splitLeft, mem} = cost(rest, l+s, [s|left], right, mem)
  {toRight, splitRight, mem} = cost(rest, l+s, left, [s|right], mem)
  if toLeft < toRight do
    {toLeft, splitLeft, mem}
  else
    {toRight, splitRight, mem}
  end
end
```

Denna funktion är relativt lång så detaljerna kring den kan inte förklaras noggrant i denna rapport, men ungefär hur den fungerar kan förklaras. Den viktigaste delen av cost skulle jag säga är den sista funktionen. Det är denna del som kollar vilket sätt som är det billigaste. Detta görs med en if sats som kollar vilket alternativt som är billigast och annvänder sig av det. Detta görs rekursivt för att hitta den billigaste metoden utav alla möjliga. Där alla möjliga metoder hittas på ett liknande sätt som split metoden använde sig av. Det som görs med hjälp av dynamisk programmering är att se till att inte samma sak checkas flera gånger. Med andra ord att man inte analyserar det mest kosteffektiva sättet att kapa en bräda som man redan undersökt det på (efter att brädan blivit kapad på ett specifikt sätt).

Sist så implementerades även två olika versioner av add och lookup. Den ena använde sig av maps och binära tal och den andra använde sig inte av maps utan implementerades direkt av mig. Dessa syns i koden nedan och det som är bortkommenterat är funktionen som använde sig av maps och binära tal.

```
def new() do
    []
  end

  def add(mem, key, val) do
    [{key, val, []}|mem]
  end

  def lookup([], _) do
    nil
  end

  def lookup([{key, val}|_], key) do
    val
  end

  def lookup([_|rest], key) do
    lookup(rest, key)
  end

  #def new() do
  #  %{}
  #end

  #def add(mem, key, val) do
  #  Map.put(mem, :binary.list_to_bin(key), val)
  #end

  #def lookup(mem, key) do
  #  Map.get(mem, :binary.list_to_bin(key))
  #end
```

Diskussion

En viktig del av den här uppgiften var att få ett så effektivt program som möjligt. Min mest effektiva version av programmet var den som använde sig av add och lookup funktionerna som implementerade maps och binära tal. När jag körde programmet med dessa funktioner tog det ungefär 40 sekunder att få resultatet för en bräda av längd 16 och ungefär 130 sekunder för en av längd 17. Detta skulle säkert kunna förbättras lite med någon förbättring men det är ändå väldigt nära resultatet som stod i instruktionerna så jag anser programmet lyckat.

Denna uppgift var inte jättesvår att få att fungera normalt. Däremot

var det svåra att få programmet att vara så effektivt som möjligt. Det krävdes många förändringar av de redan färdiggjorda funktionerna för att få programmet att bli mer och mer effektivt tills det till sist var på den nivå det skulle vara. Detta tillvägagångssätt tror jag var meningen att vi skulle använda så det kändes i alla fall inte som att jag gjorde något fel direkt. I och med det tror jag inte jag kunde ha gjort så mycket annorlunda för att ha blivit klar med uppgiften på ett bättre och kanske snabbare sätt.