Algoritmer

Emil Wiklund*

Luleå tekniska universitet
971 87 Luleå, Sverige

8 oktober 2020

Sammanfattning

Artikeln är baserad på Jon Betley's artikel från september 1983 som beskriver den vardagliga verkan som design av algoritmer kan ha på programmerare. Från en algoritmisk synvinkel på problemet kan man se att dessa kan göra ett program enklare att förstå och skriva. I denna del har vi studerat ett bidrag inom ämnet sofistikerade algoritm metoder, dessa kan ibland leda till en stor ökning prestandamässigt.

1 Introduktion

Denna delen är framtagen kring ett mindre problem med betoning på de algoritmer som ska lösa dessa problem samt teknikerna på hur man designar dom. Några av algoritmerna är lite mer komplicerade men dessa motverkas med hjälp av "big-oh" metoden. Det kommer vara tre stycken olika algoritmer som vi går igenom i denna artikel. Anledningen till varför vi väljer att undersöka detta är pågrund utav att intresset för snabbare program. Dessa program vill man alltså ska vara skrivna så att prestandan höjs.

2 Big-Oh

Betdyelsen bakom begreppet "big-oh" är att det står för en förenklad analys av en algoritms effektivitet. Detta begrepp ger oss en algoritms komplexitet baserat på invärdet, detta kallar vi för N. Det ger oss ett sätt att kunna räkna ut hur effektiv våra algortimer är. Man kan använda "big-oh" för att mäta körtiden och storleken på objekt. Med "big-oh" kan man alltså beräkna den sämsta körtiden, den bästa körtiden samt en körstid som har ett medelvärde. Generella regler som "big-oh" håller sig till är för det första att den ignorerar konstanter när N blir för stort för då så har konstanten nästan ingen

*email: emiwik-9@student.ltu.se

betydelse. För det andra så är olika termer överlägsna andra, exempelvis så är $O(N^2)$ mindre dominant än $O(N^n)$ det finns däremot flera termer och dessa är:

$$O(1) < O(log_N) < O(N) < O(Nlog_N) < O(N^2) < O(2^N) < O(N!).$$

Ett exempel där man kör kod med “big-oh” kan se ut så här:

```
for x in range (O,N) do
  for y in range (O, N) do
    print(x·y)
  end for
end for
```

Det som sker är att snurrorna kommer att byta ut x och y mot värdena inuti snurran för x respektive y innanför parantersen, i detta fall med ett naturligt tal N. Därav kommer print satsen att bli till $O(N^2)$ eftersom det blir O(N) multiplicerat med O(N) och detta är alltså körtiden för denna kod. $O(N^2)$

3 Problemet och ett simpelt program

Problemet uppstod i en endimensionellt mönsterigenkänning. Historien beskrivs senare. Invärdet är en talföljd X av N naturliga tal. Dess utvärde är den maximala summan inom en delad under talföljd av invärdet. Exempelvis om invärdet är 3 och 4 ur listan med nummer:

$$[31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$$

kommer programmet att returnera summan av $X[3...7]$, dvs 187. Problemet är enkelt när alla nummer är positiva, den största understa talföljden är lika med talföljden med invärden. Problemet uppstår när siffror är negativa. Skulle vi inkludera ett negativt nummer skulle man kunna hoppas på att ett positivt nummer skulle kunna ta ut det negativa eller med andra ord kompensera. Om alla inputs skulle vara negativa nummer skulle summan av den undre talföljden vara noll vilket också ger den totala summan 0. Det programmet man vill använda för detta är ett simpelt program som kör for each på de par av heltalen L och U där $1 \leq L \leq U \leq N$. Detta beräknar summan av $X[L...U]$ och gör en kontroll om summan är större än den summan som är störst inuläget. Koden som visas i Algorithm 1 är kort och enkel att förstå. Däremot så är den långsam. Koden tar kring 1 timme att köra om N är 1000 och 39 dagar om N är 10000. Nedan ser vi koden för Algoritm 1:

```
MaxSoFar := 0.0
for L := 1 to N do
   for U := L to N do
      Sum := 0.0
      for I := L to U do
         Sum := Sum + X[I]
      end for
      /* Sum behåller nu summan av X[L..U] */
      MaxSoFar := max(MaxSoFar, Sum)
   end for
end for
```

4 Två kvadratiska algoritmer

4.1 Algoritm 2

Vi får en annan känsla för algoritmer och hur effektiva man skulle kunna göra dom. Detta kan vi göra genom att använda oss utav "big-oh", beteckningen[1].

Uttrycken i den yttersta snurran exekveras exakt N-gånger och det i mittersta snurran exekveras som mest N-gånger i varje exekvering av den yttersta snurran. Multiplicerar dessa två faktorer inuti den mittersta snurran och visar att dessa fyra rader exekveras $O(N^2)$ antal gånger. Snurran i dessa fyra rader exekveras aldrig mer än N-gånger. Detta ger kostnad på algoritmen lika med $O(N)$. Om kostnaden multipliceras per antalet gånger som den innersta snurran körs får vi kostnaden för hela programmet och som även är proportionerligt till N^2. Så detta kan vi kalla för en kvadratisk algoritm. Med kostnad kan vi tänka oss tiden det tar att köra koden. Nedan kan vi se koden för Algoritm 2:

[1] $O(N^2)$ kan ses som proportionerligt till N^2; både $15N^2 + 100N$ och $N^2/2 - 10$ är $O(N^2)$. Sen så $f(N) = O(g(N))$ betyder att $f(N) < cg(N)$ för en konstant c och tillräckligt stora värden av N. En formell definition av beteckningen kan man hitta i de flesta böcker om design av algoritmer eller diskret matematik.

```
MaxSoFar := 0.0
for  L := 1 to N do
   for  U := 1 to N do
      Sum := 0.0
      for U := L to N do
         Sum := Sum + X[I]
      end for
      /* Sum behåller nu summan av X[L..U] */
      MaxSoFar := max(MaxSoFar, Sum)
   end for
end for
```

En kort sammanfattad förklaring angående Algoritm 2 är att den sparar variabler i sina element dessa element är dom som hittills beräknats för att få fram summan på deltalföljden som samtidigt har beräknats.

4.2 Algoritm 2b

Dessa enkla steg illustrerar tekniken av “big-oh”, analys av tiden som det tar att köra och andra fördelar men även nackdelar. Den största nackdelen med denna är att vi fortfarande inte vet exakt hur lång tid det tar programmet för en särskild input. Vi vet bara att antalet steg det tar att exekveras är $O(N^3)$. Två stycken fördelar med denna metod är att den ofta kompenserar för sina nackdelar, “big-oh” analyser är ofta användbara när man utföra sådant som vi beskrev tidigare. Sedan är den ungefärliga tiden ofta tillräcklig för att utföra den uträckning som används för att bestämma om ett program är tillräckligt effektivt för den givna uppgiften. Nedan kan vi se koden för Algoritm 2b:

```
CumArray[0] := 0.0
for I := 1 to N do
   CumArray[I] := CumArray[I - 1] + X[I]
   MaxSoFar := 0.0
   for  L := 1 to N  do
      for  U := L to N  do
         Sum := CumArray[U] - CumArray[L - 1]
      end for
      /* Sum innehåller nu X[L..U] */
      MaxSoFar := max(MaxSoFar, Sum)
   end for
end for
```

En kort sammanfattning kring Algoritm 2b är att den beräknar summan av alla föregående element och spara dem på deras dedikerade plats i CumArray’en. Detta med ett så kallat index. För att räkna ut deltalföljden plockar den fram elementets index och söker igenom i CumArray’en och ersätter summan som ligger på det index som tidigare legat på dennes plats.

Referenser

[1] Jon Bentley. *Algorithmic Design Techiniques*. Programing Pearls, September 1984.