Introduktion

Marco Kuhlmann och Victor Lagerkvist

0.01 **Bubbelsortering** är en enkel sorteringsalgoritm. Vi går igenom den lista som ska sorteras upprepade gånger från början till slutet och jämför intilliggande element. När två element ligger i fel ordning byter vi plats på dem. Efter varje genomgång kommer ett nytt element från listan att ha ”bubblat upp” till sin rätta plats, varvid antalet element som vi behöver gå igenom i nästa vända minskar med ett.

0.02 Så här ser den första genomgången av bubbelsorteringen ut för listan $2\,4\,6\,1\,3\,5$. En jämförelseoperation markeras med en färgad ruta: grön om elementen ligger i rätt ordning, röd om de behöver byta plats.

$$\boxed{2\,4}\,6\,1\,3\,5 \;\to\; 2\,\boxed{4\,6}\,1\,3\,5 \;\to\; 2\,4\,\boxed{6\,1}\,3\,5 \;\to\; 2\,4\,1\,\boxed{6\,3}\,5 \;\to\; 2\,4\,1\,3\,\boxed{6\,5}$$

Efter den första genomgången har det största elementet i listan (6) ”bubblat upp” till sin rätta plats; i nästa genomgång behöver vi därför endast sortera dellistan $2\,4\,1\,3\,5$. Vi kan dra ett streck mellan den osorterade och den sorterade delen av listan.

$$\boxed{2\,4}\,1\,3\,5\,|\,6 \;\to\; 2\,\boxed{4\,1}\,3\,5\,|\,6 \;\to\; 2\,1\,\boxed{4\,3}\,5\,|\,6 \;\to\; 2\,1\,3\,\boxed{4\,5}\,|\,6$$

Den del av listan som behöver sorteras blir kortare med varje genomgång, tills den består av endast ett enda par av intilliggande element:

$$\boxed{2\,1}\,3\,4\,|\,5\,6 \;\to\; 1\,\boxed{2\,3}\,4\,|\,5\,6 \;\to\; 1\,2\,\boxed{3\,4}\,|\,5\,6 \qquad \text{(genomgång 3)}$$

$$\boxed{1\,2}\,3\,|\,4\,5\,6 \;\to\; 1\,\boxed{2\,3}\,|\,4\,5\,6 \qquad \text{(genomgång 4)}$$

$$\boxed{1\,2}\,|\,3\,4\,5\,6 \qquad \text{(genomgång 5)}$$

0.03 När vi analyserar effektiviteten hos en sorteringsalgoritm vill vi veta hur många jämförelseoperationer vi behöver göra i förhållande till listans längd. I det konkreta exemplet hade vi kunnat avbryta sorteringen efter genomgång 4, då vi inte behövde byta plats på någon av elementen. I samband med effektivitetsanalyser är vi dock ofta intresserade av det värsta fallet (eng. *worst case analysis*). Om vi tar med även omgång 5 så var antalet jämförelseoperationer

$$5 + 4 + 3 + 2 + 1 = 15\,.$$

0.04 Vad kan vi nu säga om antalet jämförelseoperationer som vi behöver göra när vi bubbelsorterar en lista med ett godtyckligt antal element? Ett sätt att svara på denna fråga är att skriva en Python-funktion `b` ("bubbelsortering") som tar antalet element i listan som argument och returnerar det totala antalet jämförelseoperationer:

```
def b(n):     # computes n-1 + ... + 1
    s = 0
    for i in range(n-1, 0, -1):     # n-1, ..., 1
        s += i
    return s
```

Så här ser talen ut för listor med upp till 10 element:

n	1	2	3	4	5	6	7	8	9	10
b(n)	0	1	3	6	10	15	21	28	36	45

0.05 Python-funktionen `b` summerar talen mellan $n-1$ och 1 i omvänd ordning. Koden blir mer kompakt om vi summerar i vanlig ordning:

```
def b(n):     # computes 1 + ... + n-1
    s = 0
    for i in range(1, n):     # 1, ..., n-1
        s += i
    return s
```

Ännu mer kompakt blir den när vi använder en generator:

```
def b(n):
    return sum(i for i in range(1, n))
```

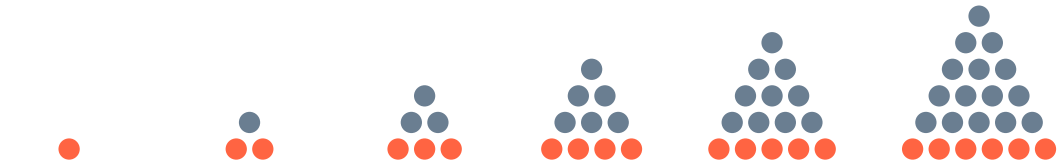
0.06 Om vi vill definiera samma funktion matematiskt kan vi använda **summanotationen**:

$$B(n) = \sum_{i=1}^{n-1} i \qquad (= 1 + \cdots + (n-1))$$

Summasymbolen kommer från den stora grekiska bokstaven Σ (sigma). Notationen utläses "summa i, då i går från 1 till $n-1$". Variabeln i är en "lokal variabel" på samma sätt som `i` var i Python-koden. Observera att gränserna för i är inklusiva, dvs. det minsta värdet för i är 1 och det största är $n-1$.[1]

0.07 I fortsättningen är det mera naturligt att inte arbeta med funktionen `b`, som ger summan av talen mellan 1 och $n-1$, utan med en funktion `t`, som ger summan av talen mellan 1 och n:

[1] I Pythons `range` är den andra gränsen *exklusiv*.



(a) $n = 1$ (b) $n = 2$ (c) $n = 3$ (d) $n = 4$ (e) $n = 5$ (f) $n = 6$

Figur 1: Illustrationer för de sex första triangeltalen.

```
def t(n):     # computes 1 + ... + n
    return sum(i for i in range(1, n+1))     # 1, ..., n
```

Samma definition med summanotation:

$$T(n) = \sum_{i=1}^{n} i \qquad (= 1 + \cdots + n)$$

Så här ser talen ut för värden mellan 1 och 10:

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	1	3	6	10	15	21	28	36	45	55

De värden som funktionen T returnerar kallas **triangeltal**. Namnet kommer av att man för dessa tal kan bilda trianglar som i figur 1. Varje punktrad i en sådan triangel svarar mot ett av talen mellan 1 och n.

0.08 Här är en annan definition av `t` som använder **rekursion**:

```
def t(n):     # computes 1 + ... + n (recursively)
    return 1 if n == 1 else t(n-1) + n
```

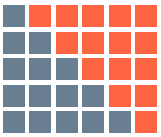
Samma definition med matematisk notation:

$$T(n) = \begin{cases} 1 & \text{om } n = 1 \\ T(n-1) + n & \text{om } n > 1 \end{cases}$$

För att övertyga oss om att denna definition är korrekt kan vi titta på figur 1 igen: Varje triangeltal kan plockas isär i en mindre triangel (grå punkter), vars storlek ges av termen $T(n-1)$, och en sista rad (röda punkter), vars storlek ges av termen n.

0.09 Här är ytterligare ett sätt att definiera funktionen `t`:

```
def t(n):     # computes 1 + ... + n (using a closed formula)
    return n * (n+1) / 2
```



Figur 2: Ett grafiskt bevis på att $T(5) = 15$.

Samma definition med matematisk notation:

$$T(n) = \frac{n \cdot (n+1)}{2}$$

Av de definitioner som vi sett hittills är denna med god marginal den mest effektiva och därför väldigt attraktiv. Däremot är det inte alls uppenbart att vi med denna definition faktiskt räknar ut rätt värde. Hur kan vi då bevisa att den är korrekt?

0.10 Ibland kan man bevisa ett påstående med hjälp av en bild. Säg till exempel att vi vill övertyga oss om att formeln i 0.09 stämmer för $n = 5$, dvs. att

$$T(5) = \frac{5 \cdot (5+1)}{2} = \frac{5 \cdot 6}{2} = 15\,.$$

Vi ritar det tionde triangeltalet som i den grå delen av figur 2: första raden består av en punkt, andra raden består av två punkter, och så vidare. Nu kopierar vi denna figur (röda punkter), vrider den 180 grader och sätter den bredvid den första. Punktraderna formar nu en rektangel bestående av $5 \cdot 6 = 30$ punkter. Detta antal behöver vi dela med 2, eftersom det triangeltal som vi vill räkna ut ju finns representerat två gånger. Detta ger oss värdet 15. Samma idé fungerar uppenbarligen med godtyckliga värden på n: Istället för 5 rader har vi då n rader, istället för 6 kolumner $(n + 1)$ kolumner. Därmed har vi bevisat formeln i 0.09 för godtyckliga värden på n.

0.11 Ett grafiskt bevis kan vara väldigt snyggt men är tyvärr inte alltid tillgängligt. En mera generell och mycket kraftfull metod att bevisa formler såsom den för triangeltalen är ett **induktionsbevis**. Här utgår vi ifrån den rekursiva definitionen i 0.08, vars korrekthet vi redan övertygat oss om. Denna definition skiljer mellan två fall: ett fall då $n = 1$ och ett fall då $n > 0$.

1. Vi börjar med fallet då $n = 1$. I detta fall säger den rekursiva definitionen att `t(n)` $= 1$, och formeln i 0.09 säger att $T(n) = 1 \cdot (1 + 1)/2 = 1$. För detta fall stämmer alltså den rekursiva definition och formeln överens med varandra.

2. Hur ser det ut i fall att $n > 1$? Då säger den rekursiva definitionen att

   $$\texttt{t(n)} = \texttt{t(n-1)} + n \qquad (\dagger)$$

   När vi exekverar koden i Python *antar* vi att det rekursiva funktionsanropet `t(n-1)` gör vad det ska – att det returnerar summan av talen mellan 1 och

$n-1$. På samma sätt *antar* vi nu att vi redan har bevisat formeln för detta anrop, dvs. att

$$\texttt{t(n-1)} = T(n-1) = \frac{(n-1)\cdot((n-1)+1)}{2} = \frac{(n-1)\cdot n}{2} = \frac{n^2-n}{2}\,. \quad (*)$$

Denna ekvation stoppar vi in i den rekursiva definitionen (†):

$$\texttt{t(n)} = \texttt{t(n-1)} + n \overset{*}{=} \frac{n^2-n}{2} + n = \frac{n^2-n}{2} + \frac{2n}{2} = \frac{n^2+n}{2} = \frac{n\cdot(n+1)}{2}$$

Därmed har vi bevisat att den rekursiva definitionen och formeln i 0.09 stämmer överens med varandra även i det andra fallet, vilket avsluter beviset.

0.12 Poängen med den här föreläsningen är att visa att matematiska resonemang behövs för att bevisa att kod gör vad den ska. Sådana resonemang kan spara oss en massa exekveringstid, en massa rader kod och därmed även en massa potentiella fel. Det är därför ni läser den här kursen.