Lektion 2

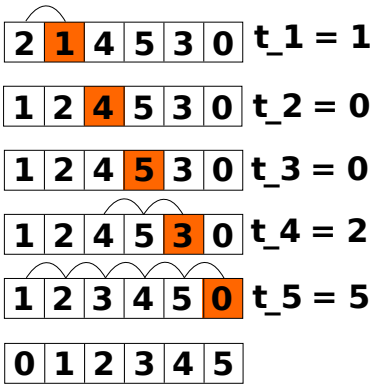
Förra lektionen tittade vi på vad ett problem är inom datavetenskapen, och vad en algoritm som löser ett problem måste uppfylla. Idag tittar vi lite mer på hur man analyserar körningstiden av en algoritm.

1 Analys av algoritmer

Vi kommer här analysera en algoritm som heter *insertion sort* och används för att sortera tal.

Antag att vi har en lista av tal $a_0, a_1, ..., a_{n-1}$ och vill ha denna lista i sorterad ordning. Ett sätt att göra detta vore att först sortera a_0 (som ju redan är sorterad), sedan sortera a_1 genom att placera in den antingen till vänster eller höger om a_0, sedan sortera a_2 genom att placera in den på rätt plats med a_0 och a_1, och så vidare.

Vi går alltså igenom varje tal i listan, ett efter ett, och placerar in det på rätt ställe i den sorterade listan.



En intressant fråga är hur lång tid det faktiskt tar att köra insertion sort. Inom datavetenskapen är man inte så ofta intresserad av den faktiskt klocktiden som algoritmen tar, utan vi försöker istället uppskatta hur körtiden växer som en funktion av indatans storlek. För en sorteringsalgoritm är indatans storlek antalet element vi sorterar, n, så vi är intresserade av hur tiden, $T(n)$ växer i förhållande till n.

För att analysera det måste vi först specifiera algoritmen i pseudokod, så att vi vet exakt vilka steg vi utför.

Algoritm 1 Insertion sort

```
1: procedure InsertionSort(A)                ▷ Sorterar listan A som har N element
2:     for i ← 1 till N − 1 do
3:         j ← i
4:         så länge j > 0 och A[j] < A[j − 1]
5:             Byt plats på A[j] och A[j − 1]
6:             j ← j − 1
```

För att analysera hur lång tid algoritmen tar börjar vi med att anta att alla småöperationer tar lika lång tid, och att den tiden är exakt 1 (av någon obestämd enhet). Vi måste vara noga med att göra rimliga antaganden av vad en liten operation är, men de flesta har en hyfsad intuition för detta.

I vårt program utför varje rad en liten operation, men de två looparna kan göra att vissa rader exekveras fler än en gång. Den yttre for-loopen kommer köras $N - 1$ gånger. Antalet gånger den inre loopen kör varierar dock beroende på hur listan vi sorterar ser ut. Vi antar därför att den för ett visst i kör t_i gånger.

Nu kan vi skriva ut hur lång tid varje rad tar, samt hur många gånger de körs:

Algoritm 2 Insertion sort

```
1: procedure InsertionSort(A)                ▷ Sorterar listan A som har N element
2:     for i ← 1 till N − 1 do                ▷ Körs N − 1 gånger, kostnad 1
3:         j ← i                              ▷ Körs N − 1 gånger, kostnad 1
4:         så länge j > 0 och A[j] < A[j − 1] ▷ Körs Σ_{i=1}^{N−1} t_j gånger, kostnad 1
5:             Byt plats på A[j] och A[j − 1] ▷ Körs Σ_{i=1}^{N−1} t_j gånger, kostnad 1
6:             j ← j − 1                      ▷ Körs Σ_{i=1}^{N−1} t_j gånger, kostnad 1
```

Vi kan nu summera kostnaden för varje rad: $T(n) = (N-1)+(N-1)+\left(\sum_{i=1}^{N-1} t_j\right)+\left(\sum_{i=1}^{N-1} t_j\right)+\left(\sum_{i=1}^{N-1} t_j\right) = 3\left(\sum_{i=1}^{N-1} t_j\right) + 2N - 2$.

Oftast är det vi är intresserade av hur algoritmen beter sig i det *värsta fallet*. Det värsta fallet för insertion sort är när listan är i omvänd ordning. Då kommer while-loopen i algoritmen köras i gånger, dvs $t_i = i$.

Övning 1. Argumentera för varje det i värsta fallet gäller att $t_i = i$.

Sätter vi in detta får vi $T(n) = 3\left(\sum_{i=1}^{N-1} i\right) + 2N - 2 = 3\frac{(N-1)(N-2)}{2} + 2N - 2 = \frac{3}{2}(N^2 - 3N + 2) + 2N - 2 = \frac{3}{2}N^2 - \frac{5}{2}N + 1$.

$T(N)$ växer alltså kvadratiskt mot antalet element N. Vi skriver detta på följande sätt: $T(N) = O(N^2)$. Detta kallas för *asymptotisk notation*, och fångar beteendet hos $T(N)$ för "stora" problem.

På samma vis, om $T(N) = 2n + 15$ växer $T(N)$ ungefär linjärt mot N, så vi säger att $T(N) = O(N)$.

Formellt defineras notationen på följande vis:

Definition 1. Vi säger att en funktion $f(n) = O(g(n))$ om $\frac{f(n)}{g(n)} \leq c$ för alla $n \geq n_0$ där c och n_0 är positiva konstanter som vi väljer.

Rent intuitivt betyder detta att $f(n)$ växer långsammare eller lika snabbt som $g(n)$. Till exempel är $an^2 + bn + c = O(n^2)$, men även $an + b = O(n^2)$. Med hjälp av denna stora-O-notation har vi alltså ett enkelt sätt att jämföra hur snabba olika algoritmer är.

Vi bevisar nu att med denna definition så är tiden som insertion sort tar i värsta fallet $O(N^2)$.

Exempel 1. Visa att $\frac{3}{2}N^2 - \frac{5}{2}N + 1 = O(N^2)$.

Bevis. När $N \geq 1$ gäller $\frac{3}{2}N^2 - \frac{5}{2}N + 1 \leq 2N^2 + N^2 + N^2 \leq 4N^2 \Rightarrow \frac{\frac{3}{2}N^2 - \frac{5}{2}N + 1}{N^2} \leq 4$ för $N \geq 2$. Med konstanterna $c = 4$ och $n_0 = 1$ uppfylls alltså villkoret i definitionen. □

Vi kallar tiden som en algoritm tar med avseende på problemstorleken för algoritmens *tidskomplexitet*. Motsvarande kan vi mäta hur mycket minne en algoritm tar med hjälp av asymptotisk notation, något vi kallar algoritmens *minneskomplexitet*. Insertion sort sparar alltid ungefär N heltal i minnet, så den har minneskomplexiteten $O(N)$.

För konstanter k säger vi att $k = O(1)$. För att avgöra om den algoritm man har kommit på är snabb nog kan man använda följande tabell. Den uppskattar hur stort problem en algoritm med en viss komplexitet kan lösa på någon sekund.

Komplexitet	n
$O(\log n)$	$2^{(10^7)}$
$O(\sqrt{n})$	10^{14}
$O(n)$	10^7
$O(n \log n)$	10^6
$O(n\sqrt{n})$	10^5
$O(n^2)$	$5 \cdot 10^3$
$O(n^2 \log n)$	$2 \cdot 10^3$
$O(n^3)$	300
$O(2^n)$	24
$O(n2^n)$	20
$O(n!)$	11

När man löser ett problem ska dock fokus alltid vara på att ha en korrekt algoritm. Ett program som går långsamt men ger rätt svar kan ge delpoäng, medan ett snabbt program som svarar att $1 + 1 = 3$ inte är mycket att ha.

Komplexitetsanalys kan också användas för att bestämma *undre gränser* till ett problem. För att resonera kring undre gränser inför vi Omega-notationen, som är motsvarande stora-O-notationen, fast åt andra hållet. Eftersom $n = O(n^2)$ säger Omega-notationen att $n^2 = \Omega(n)$. Dvs att n^2 växer alltså snabbare än n.

Notationen är vettig, för vi vill kunna konstatera att en viss algoritm måste ha en undre gräns på sin körtid. Påståendet "Algoritm X kräver minst $O(n^2)$ tid i värsta fallet" är helt värdelöst, eftersom $O(n^2)$ beskriver en *övre* gräns, inte en undre gräns. Istället vill vi säga att "Algoritm X kräver minst $\Omega(n^2)$ tid i värsta fallet".

Övning 2. Vad har insertion sort för undre körtid? Hitta en så bra gräns som möjligt, och bevisa att det är den bästa möjliga undre gränsen.

Övning 3. Ge en $O(n)$-algoritm och en $O(1)$-algoritm för att summera de n första heltalen.

Övning 4. Visa med definitionen för asymptotisk notation att $10n^2 + 7n - 5 + \log^2 n = O(n^2)$. Ge ett exempel på konstanter c, n_0 som uppfyller villkoret.

Övning 5. Visa att $f(n) + g(n) = O(max\{f(n), g(n)\})$.

Övning 6. Är $2^{n+1} = O(2^n)$? Är $2^{2n} = O(2^n)$?

Övning 7. Visa att $(n + a)^b = O(n^b)$ för positiva konstanter $a, b > 0$.