Kapitel 7

Analys av Algoritmer

7.1 Analys av algoritmer

Genom att analysera algoritmer kommer vi att bättre förstå deras egenskaper. Vi kommer långtifrån alltid att kunna göra matematiska analyser av en algoritm och kommer då i stället att ta till empiriska metoder och approximativ analys. Resultatet av detta kommer att göra det möjligt att jämföra olika algoritmer och därmed kunna välja den bästa.

Eftersom datorernas kapacitet har ökat så enormt under de senaste femtio åren så skulle man kunna tro, att effektiviteten hos de algoritmer vi använder idag inte behöver vara så hög – att snabbare datorer kompenserar sämre program. Till en viss del kan det vara så, men i takt med att datorerna blivit kraftfullare har användaren ställt högre krav.

7.1.1 Ett exempel

Vi har en *sorteringsalgoritm* som kan sortera n tal i n^2 steg ,"bubbelsortering". Dagens persondatorer är ungefär 100 gånger snabbare än för 15 år sedan. Hur många tal kan jag sortera på samma tid som jag 1985 behövde för att sortera 1000 tal?

Om ett steg tog 1 tidsenhet 1985 så tar den 1/100 tidsenheter idag. Hela sorteringen tog då 10^6 tidsenheter. Idag kan jag på samma tid utföra 10^8 steg. Det betyder att jag kan sortera $\sqrt{10^8} = 10000$ tal på samma tid. Alltså "bara" 10 gånger så många tal kan sorteras på samma tid trots att kapaciteten ökats 100 gånger.

7.1.2 Ett exempel till

Vi har en annan algoritm där det krävs 2^n operationer (eller steg) för att finna svaret för n tal. Då $n = 100$ tar det $2^{100} \approx 10^{30}$ steg att nå målet. Eftersom jag inte hade 10^{30} tidsenheter 1985, drömde jag om att kunna lösa problemet i framtiden, med snabbare datorer. 15

år senare behövs det fortfarande 10^{28} tidsenheter och problemet är fortfarande olösligt – åtminstone med denna algoritm.

7.1.3 Ett större binärt sökträd

Från föreläsning 5 drar vi oss till minnes att ett *binärt sökträd* med höjden h kan innehålla upp till $2^h - 1$ element och att det behövs högst h operationer för att ta reda på om en givet element finns i trädet eller ej. 1985 kunde jag på 10 operationer klara av ett träd med upp till $2^{10} - 1 = 1023$ element. På samma tid kan jag idag klara av 1000 operationer vilket betyder att jag kan klara av träd på upp till $2^{1000} - 1 \approx 10^{300}$ element!

Från detta kan vi konstatera att

- för vissa typer av algoritmer behöver vi inte snabbare datorer än vi har idag. Det är knappt troligt att vi kommer att hantera ett binärt sökträd med upp till 10^{300} element.
- för vissa andra typer av algoritmer kommer datorernas kapacitet *aldrig* att räcka till. För att klara 2^{200} operationer på samma tid som 2^{100} behövs en dator som är 10^{30} gånger snabbare!
- det är viktigt att veta vilken typ av algoritm vi arbetar med och vad vi kan vänta oss av den.

7.1.4 Minsta talet i lista

Vi börjar med ett enkelt exempel

```
1 int min(int a[],int n){
2   int i,m;
3   m=0;
4   for(i=1;i<n;i++)
5     if(a[i]<a[m])
6       m=i;
7   return m;
8 }
```

Funktionen tar emot en array `a` som innehåller `n` element (heltal) och returnerar index till det minsta elementet i arrayen. Hur lång tid kommer det att ta att exekvera denna funktion?

Svaret på frågan beror dels på vilken dator och dels på vilken c-kompilator vi använder. Men allra viktigast: *Det beror på hur stort* n *är.*

Vi ska här alltså inte jämföra olika datorer, programspråk eller kompilatorer utan är endast intresserade av algoritmen och dess egenskaper.

Vi väljer ut en för algoritmen ”karaktäristisk operation” och räknar helt enkelt hur många gånger denna operation kommer att utföras för olika värden på n. Vi definierar en funktion $T(n)$, *komplexitetsfunktionen.*

Väljer vi `m=0` som den karaktäristiska operationen så blir $T(n) = 1$. Om vi däremot väljer `if`-satsen som den karaktäristiska operationen kan vi se att den kommer att utföras $n-1$ gånger under ett anrop av funktionen. Detta leder till $T(n) = n-1$.

Med `m=i` som karaktäristisk operation är det svårare att bestämma $T(n)$. Allt mellan $T(n) = 0$ och $T(n) = n-1$ är möjligt. Vi ser att det hela beror på innehållet hos `a`.

Vi väntar med att bestämma oss för $T(n)$ och tar ett nytt exempel.

7.1.5 Linjär sökning i en lista

Nedan ser vi en funktion som returnerar 1 (true) om talet `s` finns i vektorn `a` och 0 (false), annars.

```
1 int search(int a[],int n,int s){
2   int i;
3   for(i=0;i<n;i++){
4     if(a[i]==s)
5       return 1;
6   }
7   return 0;
8 }
```

Eftersom funktionen lämnas, så fort man funnit det sökta elementet så är det svårare än i förra exemplet att bestämma *komplexitetsfunktionen* $T(n)$. Funktionen bestäms i högsta grad av hur `a` ser ut och av `s`, det element som söks.

Vi kan tala om tre olika funktioner

- $W(n)$, *funktionen för sämsta tänkbara situationen.* Om elementet vi söker inte finns i `a` måste vi söka igenom hela `a`. Om den karaktäristiska operationen är `if(a[i]==s)` så blir $W(n) = n$. Denna funktion är den viktigaste av de tre som presenteras här.
- $B(n)$, *funktionen för den bästa tänkbara situationen.* Denna funktion har ingen självständig betydelse. $B(n) = 1$ i vårt exempel eftersom man alltid direkt finner det element man söker!
- $A(n)$, *funktionen som bestämmer ett medeltal för hur många gånger den karaktäristiska operationen behöver utföras.* Ska den här funktionen bestämmas exakt måste man gå igenom alla tänkbara värden hos `a` och `s`. Bestämma antalet operationer för varje uppsättning och ur detta beräkna ett medelvärde. För många algoritmer inträffar $W(n)$ så sällan att det kan vara onödigt pessimistiskt att räkna med denna funktion och därför mer realistiskt att använda sig av $A(n)$. Ett rimligt värde på $A(n)$ för vårt exempel är $A(n) = (n+1)/2$

Fortsättningsvis kommer vi att betrakta $T(n) = W(n)$. Eftersom det ofta är ganska svårt att

bestämma $A(n)$ och eftersom vi dessutom vill ha en övre gräns för hur mycket arbete som kan komma att behövas.

7.1.6 Att bestämma $T(n)$

Då vi här inte arbetar med en specifik dator kan vi inte ange hur många gånger längre tid det tar att utföra en multiplikation än en addition. Vi kan heller inte jämföra dessa operationer med `if` eller `return` och bestämmer därför att de alla tar en och samma tid att utföra.

Det spelar heller ingen större roll, kommer det att visa sig, om det är frågan om 1 eller 10 multiplikationer i följd. Om det handlar om att utföra 1 eller 10 operationer en enda gång under exekveringen anses skillnaden försumbar.

Den stora skillnaden dyker upp i *loopar* och hur många gånger, n, de ska utföras.

```
b[1]=...;

for(i=0;i<n;i++)
  b[i]=...

for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    b[i]=...
```

Hur mycket datorkraft vi får betala för dessa tre rutiner beror förstås på n. Om $n = 1$ är det ingen skillnad, alla rutinerna kommer att ta 1 tidsenhet att utföra. Men om $n = 1000$ kommer rutinerna att utföras 1, 1000 respektive 1000000 gånger.

Att exakt bestämma $T(n)$ är som sagt svårt och även om det skulle vara möjligt är det inte säkert att det ens är önskvärt. Vi är mer intresserade av att klassificera algoritmen och vill därför finna en enkel funktion som vi vet, att den egentliga komplexitetsfunktionen inte överskrider.

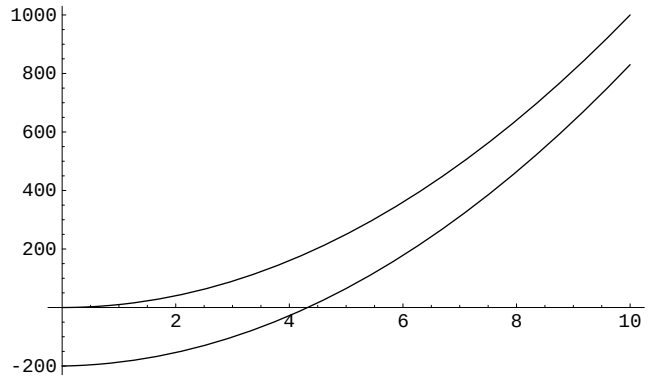
Vi studerar två funktioner $T_1(n) = 10n^2 + 3n - 200$ och $T_2(n) = 10n^2$ och visar graferna.

Av graferna förstår vi att den viktigaste av termerna i polynomen är $10n^2$. Denna term blir helt dominerande när n växer. Vi ersätter till och med $10n^2$ med n^2 och säger att $T_1(n)$ växer likt n^2 när n ökar. $T_1(n)$ är av ordningen n^2 och skriver

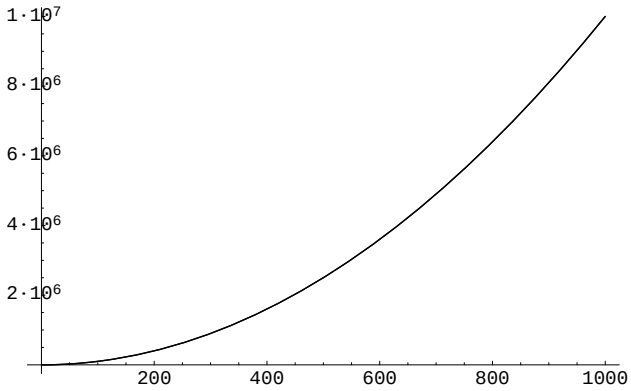
$$T_1(n) = \Theta(n^2)$$

Tre matematiska definitioner ger oss de begrepp vi behöver. $f(n)$ och $g(n)$ är två funktioner definierade för $n = 1, 2, 3 \ldots$.

- $f(n) = O(g(n))$ betyder att $c \cdot g(n)$ är en *övre gräns* för $f(n)$. Det finns en konstant c så att $f(n) \leq c \cdot g(n)$ för ett tillräckligt stort n.



Figur 7.1: *Så länge* $n <= 10$ *är det inte svårt att skilja de två funktionerna åt.*



Figur 7.2: *Men då vi studerar funktionerna i intervallet* $[0, 1000]$ *smälter de två grafer samman*

- $f(n) = \Omega(g(n))$ betyder att $c \cdot g(n)$ är en *undre gräns* för $f(n)$. Det finns en konstant c så att $f(n) \geq c \cdot g(n)$ för ett tillräckligt stort n.

- $f(n) = \Theta(g(n))$ betyder att $c_1 \cdot g(n)$ är en övre gräns för $f(n)$ och att $c_2 \cdot g(n)$ är en undre gräns för $f(n)$. Det finns alltså konstanter c_1 och c_2 sådana att $f(n) \leq c_1 \cdot g(n)$ och $f(n) \geq c_2 \cdot g(n)$.

För vår funktion $T_1(n) = 10n^2 + 3n - 200$ gäller

$$
\begin{array}{lll}
10n^2 + 3n - 200 = O(n^2) & \text{därför att} & 11n^2 > 10n^2 + 3n - 200 \\
10n^2 + 3n - 200 = O(n^3) & \text{därför att} & 0.001n^3 > 10n^2 + 3n - 200 \\
10n^2 + 3n - 200 \neq O(n) & \text{därför att} & c \cdot n < 10n^2 \text{ när } n > c \\
 & & \\
10n^2 + 3n - 200 = \Omega(n^2) & \text{därför att} & 9.99n^2 < 10n^2 + 3n - 200 \\
10n^2 + 3n - 200 \neq \Omega(n^3) & \text{därför att} & 10n^2 + 3n - 200 < n^3 \\
10n^2 + 3n - 200 = \Omega(n) & \text{därför att} & 10^{1999}n < 10n^2 + 3n - 200 \\
 & & \\
10n^2 + 3n - 200 = \Theta(n^2) & \text{därför att} & O \text{ och } \Omega \text{gäller} \\
10n^2 + 3n - 200 \neq \Theta(n^3) & \text{därför att} & \text{endast } O \text{gäller} \\
10n^2 + 3n - 200 \neq \Theta(n) & \text{därför att} & \text{endast } \Omega \text{gäller}
\end{array}
$$

Så länge $T(n)$ är ett polynom är det alltså mycket enkelt att finna en funktion $\Theta(T(n))$ Vi fastslår utan bevis det vi länge gissat

Om $a_0, a_1, a_2, \ldots, a_n$ är reella tal och $a_n \neq 0$ då är

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

$\Theta(x^n)$

7.1.7 Vilka funktioner är vanligast som $O(T(x))$

c	$\log n$	$\log^2 n$
n	$n \log n$	n^2
n^3	2^n	$n!$

För några av dem visar vi här en tabell som ska ge dig en uppfattning om hur snabbt exekveringstiden växer i takt med att n växer. Vi antar att programmet exekveras på en mycket snabb dator där den karaktäristiska operationen tar 10^{-9} sekunder att exekvera.

n	$T(n) = \lg n$	$T(n) = n$	$T(n) = n \log n$	$T(n) = n^2$	$T(n) = 2^n$	$T(n) = n!$
10	0.003µs	0.01µs	0.033µs	0.1µs	1µ	3.63ms
20	0.004µs	0.02µs	0.086µs	0.4µs	1ms	77.1år
30	0.005µs	0.03µs	0.147µs	0.9µs	1s	$8.4 \cdot 10^{15}$år
40	0.005µs	0.04µs	0.213µs	1.6µs	18.3min	
50	0.006µs	0.05µs	0.282µs	2.5µs	13dygn	
100	0.007µs	0.10µs	0.644µs	10µs	$4 \cdot 10^{13}$år	
1000	0.010µs	1.00µs	9.966µs	1ms		
10000	0.013µs	10µs	130µs	100ms		
100000	0.017µs	0.10ms	1.67ms	10s		
1000000	0.020µs	1mss	19.93ms	16.7min		
10000000	0.023µs	0.01s	0.23s	1.16dygn		
100000000	0.027µs	0.10s	2.66s	115.7dygn		
1000000000	0.030µs	1s	29.90s	31.7år		

7.1.8 Vilken komplexitet har följande algoritm?

Här ska vi se hur man med hjälp av *Maple* kan bestämma T[n].

```
1 int f(int n){
2   int i,j,k,r=0;
3   for(i=1;i<=n-1;i++)
4     for(j=i+1;j<=n;j++)
5       for(k=1;k<=j;k++)
6         r++;
7   return r;
```

Den karaktäristiska operationen är `r++`. Hur många gånger exekveras den för olika värden på n? Vi får följande värden när vi anropar funktionen

n	0	1	2	3	4	5	6	7	8	9	10
T(n)	0	0	2	8	20	40	70	112	168	240	330

Med hjälp av funktionen `Fit` i Maple kan vi bestämma en möjlig funktion – här ett polynom. Man kan känna på sig att en rutin med en trippelloop har $O(n^3)$ och ansätter därför ett sådant polynom.

```
with(Statistics}
x:=[0,1,2,3,4,5,6,7,8,9,10]
y:=[0,0,2,8,20,40,70,112,168,240,330]
Fit(a*n^3+b*n^2+c*n+d,x,y,x)
```

n och tn är två listor, med de värden vi uppmätt $(n, T(n))$. Som resultat får vi från Maple $0 - 0.333333x + 0x^2 + 0.333333x^3$. Vi tolkar det som

$$T(n) = \frac{n(n^2-1)}{3}$$

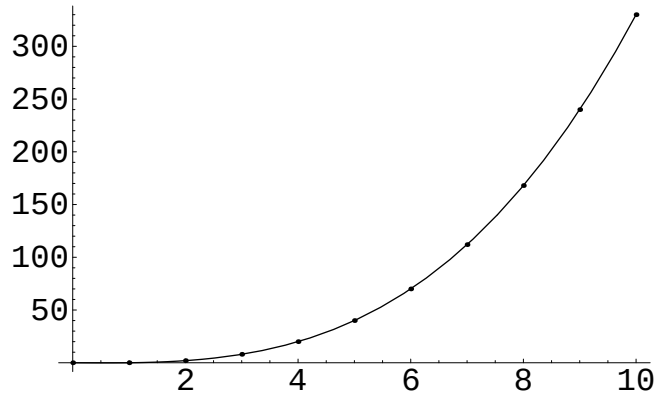
Vi är kanske inte riktigt säkra på att vi ansatt ett korrekt polynom och vill därför få en graf över både polynom och punkter.

För att kunna plotta punkterna tillsammans med kurvan använder man sig av följande teknik:

```
with(plots);
p1:=plot(x^3/3-x/3,x=0..10):
p2:=pointplot([[0,0],[1,0],[2,2],[3,8],[4,20],
    [5,40],[6,70],[7,112],[8,168],[9,240],[10,330]]):
display({p1,p2})
```

För säkerhets skull (men inte heller det är ett bevis) tar vi på två sätt reda på $T(100)$. Programmet gav $T(100) = 333300$ och det gör vår framtagna funktion också!

De gånger det går så här enkelt att bestämma $T(n)$ blir det också enkelt att bestämma $O(T[n])$ till $O(n^3)$



Figur 7.3: *Med blotta ögat ser det bra ut*

UPPGIFT 7.1
Bestäm T(n). Bestäm på ett liknade sätt som i exemplet ovan med hjälp av Maple T(n) till funktionen

```
1 int f(int n){
2   int i,j,k,m,r=0;
3   for(i=1;i<=n;i++)
4     for(j=1;j<=i;j++)
5       for(k=j;k<=i+j;k++)
6         for(m=1;m<=i+j-k;m++)
7           r++;
8   return r;
9 }
```

7.1.9 Hur effektiv är binärsökning?

Vid ett par tillfällen tidigare har vi stött på *binärsökning* utan att explicit kalla metoden så. Första gången var i föreläsning 1 i samband med ”Gissa mitt tal” och andra gången var i föreläsning 5 – binära sökträd.

I en array a finns n stycken heltal sorterade i stigande ordning. Vi vill så snabbt som möjligt ta reda på om ett visst givet tal t finns i a. Från tidigare vet vi att det är lämpligt att jämföra t med det ”mittersta” talet i arrayen.

Algoritmen för binärsökning beskriver vi så här:

Algorithm 7.1.1: BINÄRSÖKNING(a, n, t)

index ← 0
bot ← 1
top ← n
while top ≥ bot **and** index = 0
do
- mid ← $\left\lfloor \frac{\text{bot+top}}{2} \right\rfloor$
- **if** a[mid] = t
  - **then** index = mid
- **if** a[mid] > t
  - **then** top ← mid-1
  - **else** bot ← mid+1

För varje varv i loopen justeras `top` eller `bot`. Om talet t verkligen finns i a så måste förr eller senare index ≠ 0. Om `index` har värdet 0 efter att `while`-loopen har exekverats betyder det att talet t inte finns i a.

Vi påstår nu att antalet varv i loopen i *sämsta fall* är

$$T(n) = 1 + T(\lfloor n/2 \rfloor)$$

Där n är antalet tal i a och $T(1) = 1$.

Genom följande tabell får vi en uppfattning om $T(n)$.

$$
\begin{aligned}
&T(1) = 1\\
&T(2) = 1 + T(\lfloor 2/2 \rfloor) = 1 + T(1) = 1 + 1 = 2\\
&T(3) = 1 + T(\lfloor 3/2 \rfloor) = 1 + T(1) = 1 + 1 = 2\\
&T(4) = 1 + T(\lfloor 4/2 \rfloor) = 1 + T(2) = 1 + 2 = 3\\
&T(5) = 1 + T(\lfloor 5/2 \rfloor) = 1 + T(2) = 1 + 2 = 3\\
&T(6) = 1 + T(\lfloor 6/2 \rfloor) = 1 + T(3) = 1 + 2 = 3\\
&T(7) = 1 + T(\lfloor 7/2 \rfloor) = 1 + T(3) = 1 + 2 = 3\\
&T(8) = 1 + T(\lfloor 8/2 \rfloor) = 1 + T(4) = 1 + 3 = 4\\
&T(9) = 1 + T(\lfloor 9/2 \rfloor) = 1 + T(4) = 1 + 3 = 4\\
&\ldots\\
&T(15) = 1 + T(\lfloor 15/2 \rfloor) = 1 + T(7) = 1 + 3 = 4\\
&T(16) = 1 + T(\lfloor 16/2 \rfloor) = 1 + T(8) = 1 + 4 = 5\\
&\ldots
\end{aligned}
$$

När $2^i \leq n < 2^{i+1}$ så är $T(n) = i + 1$. Från matematiken får vi då att $T(n) = \lfloor \log_2 n \rfloor + 1$. Denna formel kan bevisas med induktion.

```
T:=proc(n) floor(log[2](n))+1 end proc
T(10765)
14

L:=seq([n,T(n)],n=1..100);
with(plots);
pointplot([L]);
```

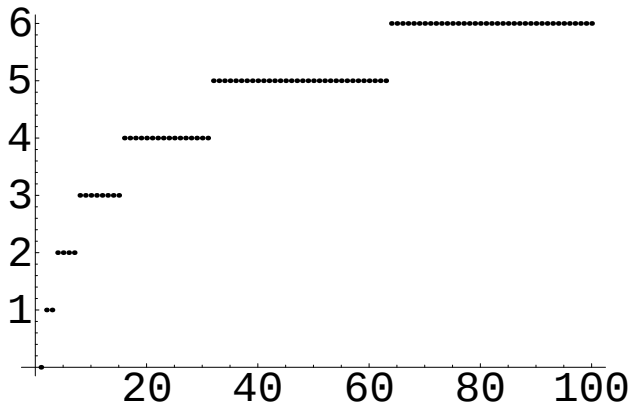
I ett av problemen i kapitlet om *binära sökträd* skulle man hantera inte mindre än 10765 ord. Genom att definiera funktionen i Maple kan vi räkna ut att trädets höjd kan som lägst bli 14.

Med hjälp av Maple får vi också en överblick över $T(n)$ (figur 7.4).

7.1.10 Profiler

Genom att använda en profiler kan man få en exakt bild över vilken del av ett program som konsumerar den mesta delen av tiden. Till Borland finns programmet `TProfilerW`. Här nedan följer ett exempel på utskrift från detta program.

Programmet bestämmer *Pythagoreiska trianglar*, rätvinkliga trianglar där alla sidorna har en heltalslängd. Vi kör programmet indata 100.



Figur 7.4: *Denna graf visualiserar* $T(n)$

```
Time  Counts
               #include <stdio.h>
0.0000 1       void main(void){
                 int small,next,last,ksmax;
0.0003 1         printf("Max för kortaste sidan: ");
6.7207 1         scanf("%d",&ksmax);
0.0000 1         small=1;
0.0550 101       while(small<=ksmax){
0.0002 100         next=small;
0.1713 5150        while(next<=ksmax){
0.2323 5050          last=next;
7.6069 176750        while(last<=ksmax){
7.5406 171700          if(last<=2*small && next<=2*small &&
                          last*last==small*small+next*next)
0.0096 27                 printf("%4d %4d %4d\n",small,next,last);
0.1439 171700          last++;
                     }
0.0042 5050          next++;
                   }
0.0000 100         small++;
                 }
0.0000 1       }
```

Till vänster i programkoden finns två kolumner. Den vänstra anger totala tidsåtgången för den satsen och den högra hur många gånger den utförts. Den innersta loopen har alltså utförts 171700 gånger och vi kan välja `if`-satsen till den *karaktäristiska operationen*.

Programmet `TProfilerW`, som är ett DOS-program, konsumerar mycket overheadtid då det körs, därför är inte de absoluta tiderna relevanta, men väl den procentuella fördelningen.

Vi är nu intresserade av att bestämma ett $O(T(n))$ och kör därför programmet för olika

värden på n. En varvräknare i den inre loopen gör samma nytta som profiler-programmet.

n	10	20	30	40	50	60	70	80	100	150	200
T(n)	220	1540	4960	11480	22100	37820	59640	88560	171700	573800	1353400

Med hjälp av Maple får vi

$$T(n) = \frac{n}{3} + \frac{n^2}{2} + \frac{n^3}{6}$$

Det var lätt att gissa att det rörde sig om ett tredjegradspolynom och därför blev ansatsen riktig från början.

UPPGIFT 7.2

Vilken komplexitetsfunktion dominerar. Här nedan har Du sex par av funktioner $T_1(n)$ och $T_2(n)$. Vilken av funktionerna ”dominerar” över den andra? Begrunda dina svar med grafer i Maple.

$T_1(n) = 2n^2$	$T_1(n) = 55n^{15} + 3n^4 + 7500$	$T_1(n) = 37n^2$
$T_2(n) = 2n$	$T_2(n) = n^{16}$	$T_2(n) = n^2 \log n$
$T_1(n) = (n+7)(n+9)$	$T_1(n) = \frac{3n+2}{n}$	$T_1(n) = n^9$
$T_2(n) = 2n^3$	$T_2(n) = 766n$	$T_2(n) = 9^n$

7.1.11 Insticksortering

Vi ska ta reda på hur snabb den sorteringsmetod som kallas *insticksortering* är.

Metoden liknar i mångt den metod en människa använder när hon ska sortera kort, papper eller brev. Man tar ett papper i taget från den osorterade bunten och ”sticker in” det på den plats där den hör hemma i den sorterade delmängd man håller i handen.

Algorithm 7.1.2: INSTICKSSORTERING(a, n)

```
for i ← 1 to n − 1
do { x ← a[i]
     j ← i − 1
     while j ≥ 0 and a[j] > x
       do { a[j + 1] ← a[j]
            j ← j − 1
     a[j + 1] ← x
```

För varje varv i `while`-loopen förbrukas en konstant tid, säg c_1. Det största antalet varv `while`-loopen kan göra är `i`. De övriga satserna i `for`-loopen antar vi tar tiden c_2. Ett varv

i `for`-loopen tar då tiden $c_2 + c_1 \cdot i$ Eftersom i varierar från 1 till $n-1$ kan vi sätta upp följande samband för hela algoritmen

$$T(n) = \sum_{i=1}^{n-1} (c_2 + c_1 \cdot i)$$

Antingen är vi bra på summering eller så tar vi Maple till hjälp.
`sum(c2+c1*i,i=1..n-1)` ger oss

$$T(n) = c_2(n-1) + \frac{c_1 \cdot n(n-1)}{2}$$

Funktionen $T(n)$ är kvadratisk i n och därför är $T(n)$ $\Theta(n^2)$. Vi har visat att *insticksortering* har kvadratisk komplexitet.

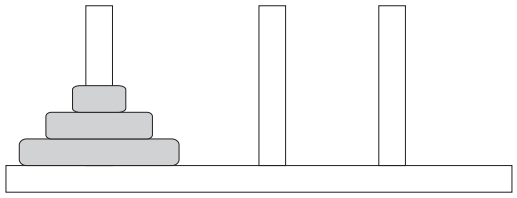
Konstanterna c_1 och c_2 är språk och datorberoende och vi kan därför inte räkna ut exekveringstiden. Även om vi skulle kunna det så skulle den bara ange tiden för den värsta tänkbara situationen för metoden – den då a från början är omvänt sorterad!

7.1.12 Analys av en rekursiv algoritm

Som arbetsexempel väljer vi *Tornen i Hanoi*. I figuren ser Du tre pinnar namngivna till a, b och c. På pinnen a finns n stycken brickor uppträdda.

Brickorna har alla olika diametrar och en större bricka får aldrig placeras ovanpå en mindre. Problemet består i att flytta brickorna från a till en annan pinne, en bricka åt gången.

Den kända algoritmen för detta problem där n anger hur många brickor som ska flyttas, från vilken pinne f, till vilken pinne t draget ska ske. x är den pinne som just nu inte är inblandad i förflyttningen.



Figur 7.5: *Tornen i Hanoi*

```
1  int hanoi(int n,int f,int t,int x){
2    if(n>0){
3      hanoi(n-1,f,x,t);
4      printf("(%d->%d)",f,t);
5      hanoi(n-1,x,t,f);
6    }
7  }
8
9  int main(void){
10   hanoi(3,1,2,3);
11 }
```

Funktionen `hanoi` är rekursiv. Den innehåller två anrop till sig själv, men nu med en bricka mindre. Vi diskuterar inte här hur den fungerar utan bara hur lång tid den tar att exekvera. $T(n) = 2T(n-1) + 1$ med $T(0) = 0$ är en bra början. Den tid det tar att ordna n brickor är lika med: *Två gånger den tid det tar att ordna* $n-1$ *brickor plus* 1. Tillägget 1 är för den tid själva anropet tar och för utskriften.

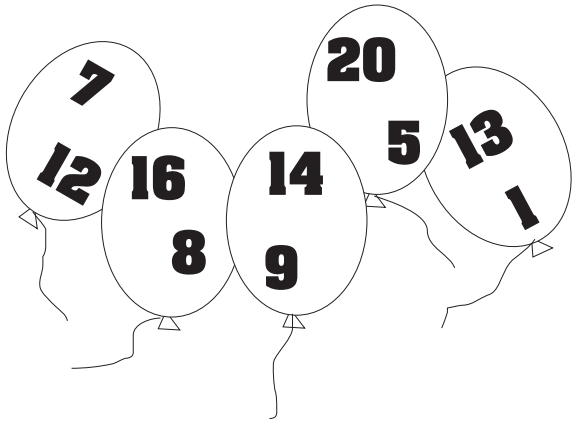
Med hjälp av $T(n)$ kan vi för hand skapa följande tabell:

n	1	2	3	4	5
T(n)	1	3	7	15	31

Adderar vi 1 till $T(n)$ får vi idel jämna tvåpotenser. Detta ger oss formeln $T(n) = 2^n - 1$. Detta är dock bara en gissning som du får bevisa i matematikkursen.

7.2 Vi löser några problem tillsammans

7.2.1 Uppgift 1 – Ballongerna



Figur 7.6:

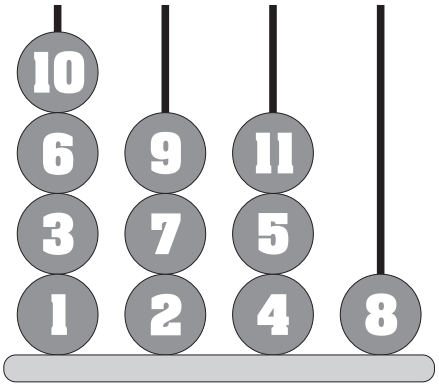
Den här uppgiften handlar om fem ballonger. På varje ballong finns tryckt fyra heltal i intervallet $1 \ldots 20$. Man kan endast se två av talen på varje ballong. Var och ett av de tjugo talen finns med precis en gång. Dessutom är summan av talen på alla ballonger densamma.

Skriv ett program som tar emot uppgift om de 10 synliga talen och bestämmer var de 10 osynliga finns.

```
Synliga tal  ? 7 12 16 8 14 9 20 5 13 1
Osynliga tal : 4 19 15 3 17 2 11 6 10 18
```

De inmatade talen hör parvis ihop, ballong för ballong. Vi kan säga från vänster till höger enligt figuren. De ska sedan skrivas ut på samma sätt, parvis, i samma ordning som indata. Det finns alltså en viss frihet vid in- och utmatning, eftersom ordningen mellan talen på samma ballong är godtycklig.

7.2.2 Uppgift 2 – Boll på pinne



Figur 7.7:

Vi har en anordning med $n, (n \leq 50)$ pinnar på vilka vi kan trä upp numrerade bollar. Numreringen börjar alltid på 1 och antalet bollar är uppåt obegränsat. Vi placerar bollarna på pinnarna i nummerordning och målet är att placera så många bollar som möjligt.

Det finns en regel: *En boll får bara placeras ovanpå en annan boll om summan av talen på bollarna är en heltalskvadrat.* Det vill säga summan kan skrivas x^2 för något tal x. Till exempel, från figuren, boll nr 11 kan placeras ovanpå nr 5 därför att $11 + 5 = 16 = 4^2$. En boll, som inte kan placeras på någon annan pinne får placeras på en tom.

Skriv ett program som frågar efter antalet pinnar och som bestämmer hur många bollar man maximalt kan trä upp. Två körningsexempel:

```
Antal pinnar ? 4
Man kan trä upp 11 bollar
```

```
Antal pinnar ? 25
Man kan trä upp 337 bollar
```

7.2.3 Uppgift 3 – Siffrorna i π

3	1	4	1	5	9	2	6	5
3	5	8	9	7	9	3	2	3
8	4	6	2	6	4	3	3	8
3	2	7	9	5	0	2	8	8
4	1	9	7	1	6	9	3	9
9	3	7	5	1	0	5	8	2
0	9	7	4	9	4	4	5	9
2	3	0	7	8	1	6	4	0
6	2	8	6	2	0	8	9	9

Figur 7.8:

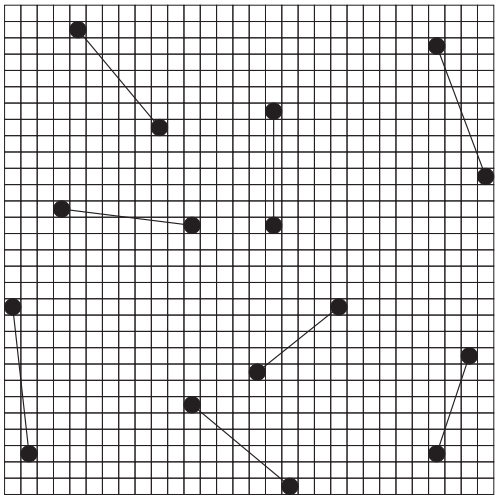
Tabellen i figur 7.8 innehåller de 81 första siffrorna i π. De svarta rutorna utgör den 'promenad' på 10 rutor som ger högsta summan. En promenad kan starta i vilken ruta som helst. Ett steg i promenaden tas alltid till närmaste ruta i någon av riktningarna: *uppåt, nedåt, höger* eller *vänster*. Man får inte gå utanför tabellen och heller inte besöka samma ruta fler än en gång.

Skriv ett program som finner den högsta summa man kan få genom en 10 rutor lång promenad enligt reglerna ovan i en 9×9 tabell där rutornas värden är positiva heltal. Data finns på filen `pi.txt`, som innehåller 9 rader med 9 tal på varje. Ett körningsexempel:

```
Högsta summan är 80
```

Observera att denna summa kan erhållas genom fyra olika promenader, som alla besöker samma rutor.

7.2.4 Uppgift 4 – Vänorter



Figur 7.9:

I landet finns n (jämnt antal) städer. Varje stad ska nu få en *vänort* (en annan stad i landet). Valet av vänorter ska göras, så att *summan av avstånden mellan alla par av vänorter blir så liten som möjligt.*

Skriv ett program, som läser data från filen `stader.txt`. Filen inleds med ett jämnt heltal $n \leq 16$ som anger antalet städer i landet. Därefter följer n rader med två tal på varje, x respektive y-koordinaten för en stad, båda heltal i $[1 \ldots 100]$. För att bestämma avståndet a mellan två städer med koordinaterna (x_i, y_i) och (x_j, y_j) används *avståndsformeln*

$$a = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Figur 7.9 visar hur valen har gjorts för de data som finns i testfilen. Ett körningsexempel:

```
Det eftersökta avståndet är 61.4
```

7.2.5 Lösning – Ballongerna

```
1 int tab[5][4],tal[21];
```

Vi väljer här några globala variabler. `tab` har plats för de 5 ballongerna och de 4 talen på varje. `tal` håller reda på talen vilka av talen [1...20] som har använts.

```
1  int main(void){
2    int i,j;
3    for(i=0;i<5;i++)
4      for(j=0;j<4;j++)
5        tab[i][j]=0;
6    for(i=0;i<21;i++)
7      tal[i]=0;
8    printf("Synliga tal ? ");
9    for(i=0;i<5;i++)
10     for(j=0;j<2;j++){
11       scanf("%d",&tab[i][j]);
12       tal[tab[i][j]]=1;
13     }
14   solve(0);
15 }
```

3-7 Nollställning av arrayerna

8-13 Inläsning av de 10 synliga talen, 2 i taget, från vänster till höger i raden.

14 Anropet `solve(0)` kommer nu att lösa problemet

```
1 solve(int ball){
2   int i,j;
3   if(ball==5){
4     printf("Osynliga tal : ");
5     for(i=0;i<5;i++)
6       for(j=2;j<4;j++)
7         printf("%d ",tab[i][j]);
8     printf("\n");
9   }
```

1 Parametern `ball` anger antalet placerade par av tal. Varje anrop avslutar en ballong.

3-9 När antalet placerade bollar är 5 har vi funnit en lösning och skriver ut resultatet. Sökningen går dock vidare för att finna fler lösningar.

```
1   for(i=1;i<=19;i++)
2     for(j=i+1;j<=20;j++)
3       if(tal[i]==0 && tal[j]==0 &&
4          i+j+tab[ball][0]+tab[ball][1]==42){
5         tal[i]=1; tal[j]=1;
6         tab[ball][2]=i;
7         tab[ball][3]=j;
8         solve(ball+1);
9         tal[i]=0; tal[j]=0;
10      }
11  }
```

10-19 En dubbelloop väljer nu ut två tal.

12 Ett krav är nu att inget av talen tidigare är använt och att summan av de 4 talen på ballongen blir 42,

$$\frac{1+2+\ldots+19+20}{5} = 42$$

14 När vi lyckats finna två kandidater bokar vi in dem som upptagna.

15-16 Talen sätts också in på rätt ballong.

17 En ballong till är klar och vi anropar `solve` för att fixa nästa.

18 När man kommer tillbaka från anropet ovan måste vi släppa de två talen vi bokade in ovan. Denna teknik kallas *backtracking*. De två `for`-looparna snurrar vidare för att hitta fler kandidater.

7.2.6 Lösning – Boll på pinne

```
1 int pinnar[50],max=0;
```

1 Två globala variabler, `pinnar`, som ska hålla reda på numret höst upp på varje pinne. `max` håller reda på det största antalet bollar vi lyckats placera.

```
1 int main(void){
2   int n,i;
3   for(i=0;i<50;i++)
4     pinnar[i]=0;
5   printf("Antal pinnar ? ");
6   scanf("%d",&n);
7   solve(1,0,n);
8   printf("Max %d\n",max-1);
9 }
```

3-6 Nollställning av array och inläsning av data.

7 Problemet är bestämt till sin storlek och vi ropar på funktionen som ska lösa problemet.

8 Väl tillbaka efter anropet vet vi svaret, lagrat i `max`.

```
1 int solve(int nr,int anvanda,int antal){
2   int i,tmp,ok;
3   if(nr>max)
4     max=nr;
5   ok=0;
6   for(i=0;i<anvanda;i++)
7     if(sqrt(pinnar[i]+nr)==(int)sqrt(pinnar[i]+nr)){
8       ok=1;
9       tmp=pinnar[i];
10      pinnar[i]=nr;
11      solve(nr+1,anvanda,antal);
12      pinnar[i]=tmp;
13    }
14  if(anvanda<antal && ok==0){
15    pinnar[anvanda]=nr;
16    solve(nr+1,anvanda+1,antal);
17    pinnar[anvanda]=0;
18  }
19 }
```

1 Parametern `nr` håller reda på vilket nummer som står i tur att placeras, `anvanda` hur många pinnar som är använda hittills och `antal` hur många pinnar det finns.

3-4 Om rekordet är slaget uppdaterar vi `max`

6-13 Vi försöker nu placera aktuell boll på pinnarna i tur och ordning från vänster till höger.

7 Om aktuellt nummer adderat till pinnens högsta nummer utgör en heltalskvadrat, accepterar vi placeringen.

8 `ok=1` innebär att en placering är möjlig och att ingen tom pinne kan komma ifråga för detta nummer.

9-10 I `tmp` lagrar vi undan högsta talet på pinnen och placerar istället aktuellt nummer där.

11 `solve` anropas med nästa tal.

12 Dags för backtracking genom att återställa numret på pinnen. Som tur var finns det i `tmp`.

14-18 Om ingen placering tidigare varit möjlig kan vi nu placera numret på en tom pinne, om det finns någon ledig.

15-16 Vi uppdaterar pinnens nummer och anropar `solve`.

17 Pinnen töms när vi återvänder hit.

7.2.7 Lösning – Siffrorna i π

```
1  int main(void){
2    FILE *fil;
3    int tab[9][9],f[9][9];
4    int i,j;
5    fil=fopen("pi3.txt","rt");
6    for(i=0;i<9;i++)
7      for(j=0;j<9;j++){
8        fscanf(fil,"%d",&tab[i][j]);
9        f[i][j]=0;
10     }
11   fclose(fil);
12   for(i=0;i<9;i++)
13     for(j=0;j<9;j++)
14       solve(i,j,tab,f,1,0);
15   printf("Maximala summan: %d\n",max);
16 }
```

3 I `tab` ska siffrorna från π lagras radvis. I `f` noterar vi med `1` att siffran använts.

5-11 Siffrorna läses in från filen och ”flaggmatrisen” nollställs.

12-14 Vi ska nu starta med var och en av de 81 siffrorna och därför ska funktionen `solve` anropas lika många gånger.

15 Dags att skriva ut den maximala summan

```
1  int solve(int rad,int kol,int tab[][9],int f[][9],int n,int sum){
2    int i;
3    sum=sum+tab[rad][kol];
4    if(n==10){
5      if(sum>max)
6        max=sum;
7    }
8    else{
9      f[rad][kol]=1;
10     if(rad>0 && f[rad-1][kol]==0)
11       solve(rad-1,kol,tab,f,n+1,sum);
12     if(rad<8 && f[rad+1][kol]==0)
13       solve(rad+1,kol,tab,f,n+1,sum);
14     if(kol>0 && f[rad][kol-1]==0)
15       solve(rad,kol-1,tab,f,n+1,sum);
16     if(kol<8 && f[rad][kol+1]==0)
17       solve(rad,kol+1,tab,f,n+1,sum);
18     f[rad][kol]=0;
19   }
20 }
```

1 Parametrarna: `rad` aktuell rad, `kol` aktuell kolumn, `tab` arrayen med siffror, `f` arrayen med flaggor (istället för att globala som i tidigare exempel), `n` antalet inräknade siffror, `sum` aktuell summa.

3 Siffran i aktuell ruta läggs till summan

4-7 Om $n = 10$ har vi nått det djup vi önskar. Om nu summan slår rekord uppdaterar vi `max`.

9 Vi markerar aktuell ruta som använd

10-17 De finns nu högst 4 riktningar att gå i. Om vi inte hamnar utanför matrisen och rutan är ledig anropar vi `solve` rekursivt med uppdaterade parametrar. Alla riktningar får chansen.

18 När vi återvänder, återställer vi flaggan i `f[rad][kol]`.

7.2.8 Lösning – Vänorter

```
1  int main(void){
2    int f[16]={0},n;
3    n=init();
4    solve(f,n,0.0,0);
5    printf("Kortaste vägen är %.1f\n",min);
6  }
```

2 `f` används som flaggor för använda orter.

3 `init` läser in data.

4 `solve` löser problemet

5 Skriver ut resultatet

```
1  float avst[16][16],min=1E10;
2
3  int init(){
4    int i,j,n,p[16][2];
5    FILE *fil;
6    fil=fopen("stadere.txt","rt");
7    fscanf(fil,"%d",&n);
8    for(i=0;i<n;i++)
9      fscanf(fil,"%d %d",&p[i][0],&p[i][1]);
10   fclose(fil);
11   for(i=0;i<n;i++)
12     for(j=i+1;j<n;j++){
13       avst[i][j]=sqrt(pow(p[i][0]-p[j][0],2)+pow(p[i][1]-p[j][1],2));
14       avst[j][i]=avst[i][j];
15     }
16   return n;
17 }
```

1 I `avst` ska vi ha samtliga avstånd mellan städerna uträknade en gång för alla. I `min` hamnar till slut det resultatet.

6-10 Indata läses in. Orternas koordinater lagras tillfälligt i `p`

11-15 Avstånden räknas ut och lagras i `avst` (på två ställen).

16 Antalet orter returneras.

```
1  void solve(int f[],int n,float vag,int d){
2    int i,j;
3    if(2*d==n){
4      if(vag<min)
5        min=vag;
6    }
7    else{
8      if(vag<min){
9        for(i=0;i<n-1;i++)
10         if(f[i]==0){
11           f[i]=1;
12           break;
13         }
14       for(j=i+1;j<n;j++)
15         if(f[j]==0){
16           f[j]=1;
17           solve(f,n,vag+avst[i][j],d+1);
18           f[j]=0;
19         }
20       f[i]=0;
21     }
22   }
23 }
24
```

1 Parametrarna: `f` för använda orter, `n` antal orter, `vag` den totala vägen, `d` antal bildade par.

3-6 Om alla orter är ihopparade, kollar vi om vi hittat ett kortare totalavstånd.

8 Denna `if`-sats är viktig! Om vi redan har ett totalavstånd som är längre än `min` är det ingen vits att söka vidare på denna gren av rekursionsträdet.

9-13 Vi söker nu upp en ”ledig” ort.

14-19 En till. När vi funnit två och bokat av dem anropar vi `solve`

18 När vi återvänder återlämnar vi den senaste orten till de ”lediga”

20 När den först valda i paret testats med alla lediga är det dagas att återlämna även denna ort.