Kapitel 10

Söndra och Härska

10.1 Största värdet i en array

Algoritmer i detta kapitel följer en mycket viktig paradigm – *söndra och härska* eller *divide and conquer* som är den engelska termen. Tekniken är ganska vanlig vid konstruerandet av algoritmer. Att starta med ett stort problem P. Dela det i två eller flera små delproblem p. När man fått lösningen på delproblemen kombineras deras lösningar till lösningen på det stora problemet. Men hur får man då lösningen till delproblemen? Dela dem vidare i ännu mindre problem tills problemen blir så små att de blir triviala att lösa!

Tycker du att detta låter som *rekursion*? Inte överraskande är rekursiva funktioner det enklaste sättet att implementera *söndra och härska*

Vi börjar med ett exempel som vi redan har en enkelt lösning till. En lösning som dessutom inte kan göras bättre. Trots det:

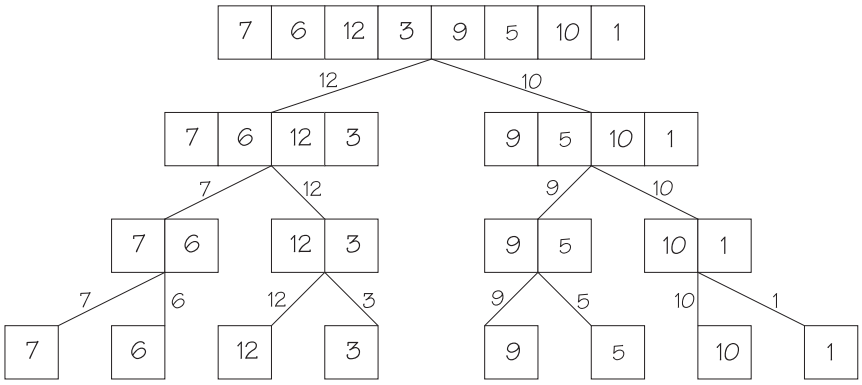
```
1  int max(int a[],int low,int high){
2    int m1,m2;
3    if(low==high)
4      return a[low];
5    else{
6      m1=max(a,low,(low+high)/2);
7      m2=max(a,(low+high)/2+1,high);
8      if (m1>m2)
9        return m1;
10     else
11       return m2;
12   }
13 }
```

Vi har en array a med n heltal och vill ha reda på det största av dessa. Första anropet av

funktionen ovan blir `max(a,0,n-1)`, sök reda på det största talet i `a`, mellan index `0` och index $n-1$, hela arrayen.

Om det skulle vara så att `low` och `high` vore lika, är det bara ett tal det handlar om, och problemet är enkelt. Detta tal returneras av funktionen.

Om däremot `low<high` finns det flera kandidater. Problemet delas då upp på två ungefär lika stora delproblem. Resultatet från vart och ett returneras och *kombineras* till lösningen på det större problemet.



Figur 10.1:

10.2 Exponentiering

För att bestämma 123^2 behöver man utföra *en* multiplikation och för att bestämma 123^3 måste man göra *två*. Men hur många multiplikationer behöver man göra för att bestämma 123^4?

Här följer en algoritm som beräknar a^b, båda positiva heltal.

Algorithm 10.2.1: EXP(a, b)

$z \leftarrow 1$
if $b = 0$
  then return (1)
if $b = 1$
  then return (a)
$z \leftarrow$ EXP$(a, \lfloor b/2 \rfloor)$
if even(b)
  then return $(z * z)$
  else return $(z * z * a)$

Hur många multiplikationer behövs för olika värden på b?

b	5	8	31	123
multiplikationer	3	3	8	11

10.3 Äntligen en snabb sortering

Redan innan vi kommer till kapitlet om sortering ska vi här presentera *mergesort*. Anledningen till det är att den är uppbyggd efter paradigmen *divide and conquer*.

Algorithm 10.3.1: MERGESORT(x, n)

```
if n = 1
  then return ( )
if n = 2
  then { if x[0] > x[1]
         then { t ← x[0]
                x[0] ← x[1]
                x[1] ← t
  else { m ← ⌊n/2⌋
         for i ← 0 to m − 1
           do a[i] ← x[i]
         for i ← m to n − 1
           do b[i − m] ← x[i]
         MERGESORT(a, m)
         MERGESORT(b, n − m)
         i ← 0
         j ← 0
         for k ← 0 to n − 1
           do { if a[i] ≤ b[j]
                then { x[k] ← a[i]
                       i ← i + 1
                else { x[k] ← b[j]
                       j ← j + 1
```

Det som ska sorteras finns i x, som innehåller n objekt. Om $n = 1$ eller $n = 2$ är det hela snabbt avklarat. I annat fall kopierar vi hälften av elementen i x till a och den andra hälften till b. Dessa två arrayer a och b sorteras nu rekursivt. Detta är *divide*-delen av algoritmen.

När a och b så småningom är sorterade var och en för sig återstår att sammanföra dem till x. Detta kallas *samsortera* eller *merge*. Två sorterade arrayer med n respektive m objekt kan samsorteras i $O(n + m)$ steg. Detta är algoritmens *conquer*-del.

Det är endast ett objekt i taget från var och en av a och b som kan komma i fråga att flyttas över till x. När detta element är överfört ökas motsvarande index och samsorteringen kan fortsätta.

UPPGIFT 10.1
Implementera mergesort efter algoritmen ovan tillsammans med *enkel sortering*. Generera arrayer att sortera, med hjälp av slumpen. Använd samma osorterade array till båda sorteringarna. Ge dem en storlek tillräcklig för att känna skillnaden i effektivitet mellan de två metoderna.

Observera att *merge*-delen i praktiken är lite knepigare än som den teoretiskt visas i algoritmrutan

Skriv en liten rutin som testar att sorteringen verkligen fungerar.

10.4 Maximal delsekvens

Givet är en vektor a innehållande n heltal, såväl positiva som negativa. Vi söker den följd av element i a som ger största summan. Ett exempel får förklara.

31	-41	**59**	**26**	**-53**	**58**	**97**	-93	-23	84

Summerar vi talen med fet stil får vi summan 187. Ingen annan kontinuerlig sekvens av element i denna array ger en högre summa. Vi är nu på jakt efter en algoritm, som kan lösa problemet för godtyckliga a och n. Vi inser direkt att om alla tal är *icke negativa* så blir den maximala summan, summan av alla elementen i arrayen. På samma sätt, om samtliga element är *icke positiva*, så blir den maximala summan 0, vi tar helt enkelt inte med något element alls.

10.4.1 Ett första försök

Den mest närliggande algoritmen är kanske:

Algorithm 10.4.1: F1(a, n)

```
for l ← 1 to n
do { for h ← l to n
     do { sum ← 0
          for i ← l to h
            do sum ← sum + a[i]
          m ← max(m, sum)
return (m)
```

Vi väljer alltså alla tänkbara l (lägsta index för sekvensen) och alla tänkbara h (högsta index för sekvensen) där $l \leq h$, vilket ger samtliga möjliga sekvenser. För varje sådan sekvens bestämmer vi summa *sum* och jämför den med ett maxvärde m. Funktionen *max* returnerar den största av de två parametrarna. Det slutliga m returneras och är samtidigt vårt resultat.

Då vi mäter tiden för olika värden på n får vi följande tabell i μs.

n	5	10	20	40	80	200	400	800	1500
tid	3	12	48	316	2473	29890	228791	1802198	11816483

Från de erfarenheter vi har förstår vi att algoritmen löser problemet i $O(n^3)$ tid. Låter vi Mathematica anpassa tabellens data får vi $T(n) = 0.00350n^3$, alltså $O(n^3)$

10.4.2 Kan detta göras bättre?

Javisst här kommer två algoritmer som gör jobbet mycket snabbare för stora värden på n.

Algorithm 10.4.2: F2(a, n)

```
m ← 0
for l ← 1 to n
do { sum ← 0
     for h ← l to n
     do { sum ← sum + a[h]
          m ← max(m, sum)
return (m)
```

Precis som i första förslaget låter vi det undre index l stega igenom alla arrayens index. För varje l får sedan h (även det som tidigare) anta alla index $h \geq l$. Men, och det är här skillnaden uppstår, passar vi samtidigt på att summera och testa. Detta betyder att vi sparar in en loop och får en mycket snabbare funktion.

n	20	40	80	150	300	600	1500	3000	6000	12000
tid	11	55	198	615	2692	10330	65495	260989	1087912	4510989

Det handlar givetvis om en $O(n^2)$ algoritm. Mathematica föreslår $T(n) = 0.0313n^2$, alltså $O(n^2)$. Ett alternativ till denna algoritm skulle kunna vara

Algorithm 10.4.3: F3(a, n)

```
c[0] ← 0
for i ← 1 to n
  do c[i] ← c[i − 1] + a[i]
m ← 0
for l ← 1 to n
  do { for h ← l to n
       do { sum ← c[h] − c[l − 1]
            m ← (m, sum)
return (m)
```

Skillnaden ligger i att i en extra array c bestäms de ackumulerade summorna. I c_i finns $\sum_{j=1}^{i} a_j$. Funktionen innehåller visserligen tre for-satser men eftersom de inte alla är nästlade kommer denna algoritm ungefär att vara lika snabb som 10.4.2 och betydligt bättre än 10.4.1

n	20	40	80	150	300	600	1500	3000	6000	12000
tid	22	66	220	703	2747	11099	68352	272527	1125275	4560440

Att den är något sämre än 10.4.2 beror förstås på den extra loopen som bygger upp arrayen c. Med Mathematica kommer vi fram till $T(n) = 0.0316n^2$, även den $O(n^2)$

10.4.3 Kan detta göras bättre?

Eftersom dagens föreläsning handlar om *divide and conquer* blir det förstås inte speciellt överraskande med denna algoritm

Algorithm 10.4.4: F4(a, l, h)

```
if l > h
  then return (0)
if l = h
  then return (max(0, a[l]))
c ← ⌊(l + h)/2⌋
sum ← 0
mleft ← 0
for i ← c downto l
  do { sum ← sum + a[i]
     { mleft ← max(mleft, sum)
sum ← 0
mright ← 0
for i ← c + 1 to h
  do { sum ← sum + a[i]
     { mright ← max(mright, sum)
m ← mleft + mright
mleft ← F4(a, l, c)
mright ← F4(a, c + 1, h)
return (max(m,mleft,mright))
```

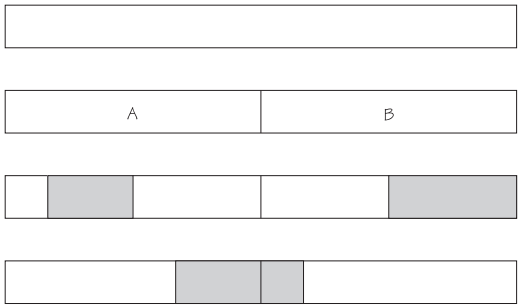
Innan vi nu tar sats för att förstå algoritmen upprepar vi

För att lösa ett problem av storleken n*, löser vi först rekursivt två delproblem av ungefär storleken* $n/2$*. Vi kombinerar sedan lösningarna från dessa två problem till en lösning av hela problemet*

Vi ser att algoritmen är rekursiv och att varje anrop leder till två nya anrop med hälften så många element. När vi kommer tillbaka från anropen har någon av de två situationerna i figur 10.2 uppstått.

Vår algoritm måste därför speciellt intressera sig för fallet då då den maximala summan ligger över gränsen. Den största delen av koden går åt till detta specialfall. Men den är inte så komplicerad. Vi vet ju att de två elementen närmast ”gränsen” måste ingå i en eventuell maximal delsumma och börjar därför där för addera mer och mer avlägsna element.

De två for-looparna ger oss två maximala delsummor som kan adderas till den största summa som ligger över gränsen. Detta maximum m jämförs sedan med de två maximala delsummorna *mleft* och *mright*. Största summan av dessa är lösningen på (del)problemet.



Figur 10.2: *Ursprungsproblemet delas i två delproblem. Resultatet från dessa kan antingen leda fram till två maximala delsummor, som inte har något element gemensamt, eller till en maximal delsumma, som ligger över gränsen mellan delarna.*

Tittar vi på exekveringstiden för olika värden på n får vi denna tabell.

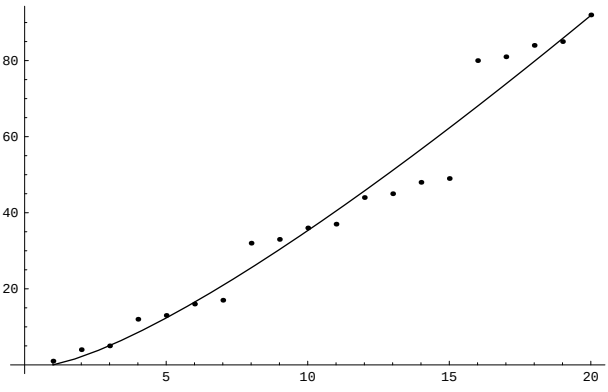
n	200	400	1000	2000	4000	10000	20000	50000	100000	200000
tid	203	407	1231	2330	4945	13297	27033	74176	149451	324176

Komplexitetsfunktionen $T(n) = 2T(n/2) + n$ förklaras av att problemet med n tal delas upp i två problem med $n/2$ tal. Att kombinera två lösningar kostar högst n.

Löser vi denna rekursiva formel där $T(1) = 1$ får vi genom Mathematica:

```
t[1]=1;
t[n_]:=2*t[Floor[n/2]]+n;
Table[t[i],{i,1,20}]
```

talföljden för $i = 1 \ldots 10$ blir $1, 4, 5, 12, 13, 16, 17, 32, 33, 36, 37, 44, 45, 48, 49, 80, 81, 84, 85, 92$ vilket Mathematica anser bäst anpassas av $T(n) = 1.53332n \log n$,alltså $O(n \log n$. Plottar vi punkterna tillsammans med denna funktion får vi



Figur 10.3:

10.4.4 Som om det vore slut med detta

Det finns faktiskt ett ännu kraftfullare sätt att lösa problemet.

Algorithm 10.4.5: F5(a, n)

```
m ← 0
me ← 0
for i ← 1 to n
  do { me ← max(me + a[i], 0)
     { m ← max(m, me)
return (m)
```

Det handlar om den allra enklaste algoritm som opererar på en array – den som startar med $a[1]$ och besöker varje element precis en gång fram till $a[n]$. Hur är detta möjligt? På vägen genom arrayen håller algoritmen reda på den maximala summan ”så långt”.

Anta att vi har löst problemet för $a[1, \ldots, i-1]$. Hur kan denna lösning utvidgas till $a[1, \ldots, i]$? Den maximala summan hos $a[1, \ldots, i]$ är antingen lika med den maximala summan hos $a[1, \ldots, i-1]$ (m) eller så är det en ny summa $a[j, \ldots, i]$ (me), där $a[i]$ ingår.

Följer vi det inledande exemplet kommer m och me att anta följande värden under exekveringen och det slutliga värdet hos m blir slutgiltiga resultatet.

i	1	2	3	4	5	6	7	8	9	10
m	31	31	59	85	85	90	187	187	187	187
me	31	0	59	85	32	90	187	94	71	155

n	500	1000	2000	4000	8000	20000	40000	80000	150000	300000
tid	33	66	220	352	440	1978	3297	7143	18681	27473

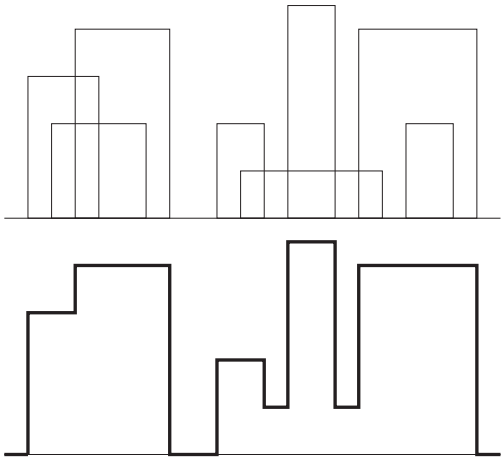
Algoritmen har alltså komplexiteten $O(n)$.

10.5 Skyline-problemet

Överst i figur 10.4 ser vi 8 (genomskinliga) hus. Underst ser vi den silhuett dessa hus bildar. Problemet består i att från de data som beskriver husens höjd och placering bestämma silhuetten. För varje hus behöver vi två x-koordinater och en y-koordinat. Vi bestämmer oss för ordningen (x_1, y, x_2) och kan beskriva alla husen med följande data

```
(0,30,15),(5,20,25),(10,40,30),(40,20,50),
(45,10,75),(55,45,65),(70,40,95),(80,20,90)
```

Resultatet vill vi presentera på följande sätt



Figur 10.4:

```
0,30,10,40,30,0,40,20,50,10,55,45,65,10,70,40,95,0
```

Vi startar med en x-koordinat och varvar sedan y- och x-koordinater tills hela silhuetten är beskriven.

Detta problem används ofta för att demonstrera Söndra och Härska.

```
1 int data[8][4]={{0,30,15,0} ,{5,20,25,0} ,{10,40,30,0},
2                 {40,20,50,0},{45,10,75,0},{55,45,65,0},
3                 {70,40,95,0},{80,20,90}};
```

H Vi överför data från vårt exempel till en hårdkodad matris.

```
1 int main(void){
2   int i,j=0,a[32];
3   skyline(a,0,7);
4   printf("(%d,%d)",a[0],a[1]);
5   for(i=3;i<32;i=i+2)
6     if(a[i]!=a[i-2])
7       printf("(%d,%d)",a[i-1],a[i]);
8 }
```

3 Anropet som ska lösa problemet. Resultatet får vi tillbaka i arrayen `a`. Parametrarna 0 och 7 anger det intervall av hus som ska ingå lösningen. Motsvarande uppgifter finns i matrisen `data`.

5-7 När vi återvänder innehåller `a` 16 par av x- och y-koordinater. I de falla två intilliggande par har samma y-koordinat kan vi utesluta det senare.

```
1 void skyline(int a[],int low,int high){
2   int a1[32],a2[32],i1,i2,i3,i,n,h1,h2;
3   if(high-low==0)
4     for(i=0;i<4;i++)
5       a[i]=data[low][i];
6   else{
7     skyline(a1,low,(low+high)/2);
8     skyline(a2,(low+high)/2+1,high);
```

H Funktionen `skyline` är förstås programmets hjärta.

7-8 Söndra delen. Vi halverar hela tiden antalet hus som ska hanteras.

3-5 Förr eller senare kommer `high-low` att bli lika med `0` och då är det bara ett hus som hanteras. Vi fyller då på `a` med data för detta hus, fyra heltal.

```
1     i1=0; i2=0; i3=0; h1=0; h2=0;
2     n=2*(high-low+1)-1;
3     while(i1<=n || i2<=n){
4       if(i1<=n && (a1[i1]<a2[i2] || i2>=n)){
5           h1=a1[i1+1];
6           a[i3++]=a1[i1++];
7           if(h1>=h2)
8             a[i3++]=a1[i1++];
9           else{
10            a[i3++]=h2;
11            i1++;
12          }
13        }
14      else
15        if(i2<=n && (a2[i2]<=a1[i1] || i1>=n)){
16          h2=a2[i2+1];
17          a[i3++]=a2[i2++];
18          if(h2>=h1)
19            a[i3++]=a2[i2++];
20          else{
21            a[i3++]=h1;
22            i2++;
23          }
24        }
25    }
26  }
27 }
```

H Härska-delen ser i stort sett ut som *mergesort*. Vi samkör de två arrayerna `a1` och `a2` till `a`, så att x-koordinaterna kommer i stigande ordning. Med hjälp av variablerna `h1` och `h2` kan vi hålla reda på vilket `y`-koordinat som ska gälla (den största)

2 `n` håller reda på högsta index i de två arrayerna. Funktionen kommer bara att fungera då antalet hus är en 2-potens, eftersom man annars någon gång kommer att få olika antal hus i de två arrayerna `a1` och `a2`.

3 Så länge det finns hus kvar i någon array snurrar `while`-loopen vidare.

4-24 Beroende på i vilken array den minsta x-koordinaten finns kommer antingen $5-12$ eller $16-23$ att exekveras. Om en array redan är tömd kommer den andra att kopieras över till `a`.

Hur snabbt är då programmet? Formeln

$$T(n) = 2T(n/2) + n$$

Ett problem delas upp i två, hälften så stora, problem under *söndringen* Vid *härskandet* har vi en loop som utför n steg. Vi har inte lärt oss att lösa den här typen rekursionsekvationer. I Mathematica skriver vi

```
RSolve[{T[n]==2*T[n/2]+n, T[1]==0}, T[n], n]
  T[n]->(n Log[n])/Log[2]
```

som svar får vi

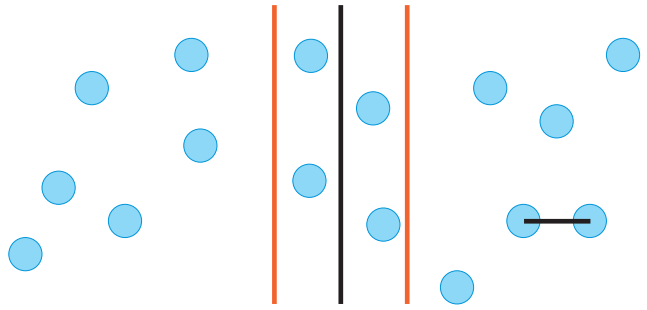
$$T_n = \frac{n \log n}{\log 2}$$

$T(1) = 0$ eller $T(1) = 1$ hur lång tid man nu tycker att det tar att lösa problemet för ett hus. Kvar blir det glädjande resultatet $O(n \ln n)$ eller $O(n \log n)$. Problemet går alltså att lösa lika snabbt som den snabbaste sortering!

10.6 Closest Pair

Givet n punkter i planet, där vi söker de två punkter som ligger närmast varandra. Med brutal kraft kan man bestämma alla $n(n-1)/2$ avstånd mellan punkterna och därmed också det minsta avståndet. Denna algoritm har komplexiteten $O(n^2)$. Kan jobbet utföras snabbare?

```
1 #include <stdio.h>
2 int  m[16][2]={{2,8},{5,10},{7,3},{11,0},{13,10},{17,-2},
3              {18,3},{21,4},{28,-5},{31,7},{35,9},
4              {37,2},{42,6},{53,15},{58,12},{64,5}};
5
6 float avst(int x1,int y1,int x2,int y2){
7   return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
8 }
```



Figur 10.5: *De två punkter som ligger närmast varandra är utmärkta med det avstånd* d *som ligger mellan dem. Mittlinjen bestäms av de två ytterlighetspunkterna i de två mängderna på var sin sida om mittlinjen. De två linjerna på var sida om mittlinjen bestäms av kortaste avståndet* d. *Det finns fyra punkter som måste undersökas om de ligger på ett avstånd* $< d$ *från varandra.*

H Vi deklarerar en matris `m` med 16 punkter sorterade efter stigande x-koordinater.

6-8 Vi behöver en funktion som kan bestämma avståndet mellan två givna punkter (x_1, y_1) och (x_2, y_2)

```
1  float brutalkraft(void){
2    int i,j;
3    float min=1E10,d;
4    for(i=0;i<15;i++)
5      for(j=i+1;j<16;j++){
6        d=avst(m[i][0],m[i][1],m[j][0],m[j][1]);
7        if(d<min)
8          min=d;
9      }
10   return min;
11 }
```

H En funktion som bestämmer avståndet mellan de två punkter som ligger närmast varandra med hjälp av *exhausted search*. Med så här få punkter kommer det inte att märkas någon skillnad mellan den funktion och den nedan.

```
1  float closestpair(int s,int t){
2    float d,d1,d2,d3,mitt,v,h;
3    int i,j;
4    if(t-s==1)
5      return avst(m[t][0],m[t][1],m[s][0],m[s][1]);
6    else{
7      d1=closestpair(s,(s+t-1)/2);
8      d2=closestpair((s+t-1)/2+1,t);
9      mitt=(m[(s+t-1)/2][0]+m[(s+t-1)/2+1][0])/2;
10     if(d1<d2) d=d1; else d=d2;
11     v=mitt-d;
12     h=mitt+d;
13     for(i=(s+t-1)/2;i>=s;i--)
14       if(m[i][0]>v)
15         for(j=(s+t-1)/2+1;j<=t;j++)
16           if(m[j][0]<h){
17             d3=avst(m[i][0],m[i][1],m[j][0],m[j][1]);
18             if(d3<d) d=d3;
19           }
20     return d;
21   }
22 }
```

7-8 Här sker söndringen. Vi halverar antalet punkter hela tiden. och anropar funktionen två gånger med varsin halva.

4-5 När det bara är två punkter kvar kan man bestämma avståndet mellan dessa och returnera resultatet.

9-19 Här slår man ska man slå samman två punktmängder. Man vet att i den ena är minsta avståndet `d1` och i den andra `d2`.

9 `mitt` är ett x-värde som ligger mitt emellan punkten längst till höger i den vänstra mängden och punkten längst till vänster i den högra mängden.

10-12 När man nu ska slå samman de två mängderna kan det finnas två punkter som ligger närmare varandra är `d1` och `d2`. Då måste punkterna ligga i olika mängder och inte för långt från `mitt`, närmare bestämt $\leq$ 2`d`. Där `d` är minimum av `d1` och `d2`. Tänk efter, så att du förstår varför!

13-19 Så bestämmer vi avståndet mellan alla punkter som ligger innanför detta område (se figur 10.5). Denna rutin kan göras snabbare genom att sortera punkterna även i `y`-led.

20 Det värde som `d` har returneras nu.

```
1 int main(void){
2   printf("%.2f\n",brutalkraft());
3   printf("%.2f\n",closestpair(1,16));
4 }
```

10.6.1 Fibonacci med Söndra och Härska

Vi har tidigare studerat olika sätt att bestämma fibonacci-tal.

$$f_n = f_{n-1} + f_{n-2} \qquad f_0 = 1 \qquad f_1 = 1$$

Här följer ett program som löser problemet med hjälp av *divide-and-conquer*.

```
1 #include <stdio.h>
2 int anrop=0;
3 long long fibonacci(long long n){
4   long long a,b;
5   anrop++;
6   if(n==1||n==0)
7     return n;
8   else{
9     a=fibonacci((n+1)/2);
10    b=fibonacci((n+1)/2-1);
11    if(n%2==0)
12      return a*(a+2*b);
13    else
14      return a*a+b*b;
15  }
16 }
17
18 long long fibonacci2(long long n){
19   anrop++;
20   if(n==1||n==0)
21     return n;
22   else
23     return fibonacci2(n-1)+fibonacci2(n-2);
24 }
25
26 int main(void){
27   printf("Efter %d anrop: %lld\n",anrop,fibonacci(25));
28   anrop=0;
29   printf("Efter %d anrop: %lld\n",anrop,fibonacci2(25));
30 }
```

fibonacci(25) = 75025. Detta klaras av med 41 anrop med `fibonacci()`. Samma resultat erhålles med `fibonacci2()`, men denna gång efter 242785 anrop.

`fibonacci()` beräknar sitt resultat genom

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ (F_{\lceil n/2 \rceil})^2 + F_{\lceil n/2 \rceil - 1})^2 & n \geq 2 \quad n\text{udda} \\ (F_{\lceil n/2 \rceil})^2 + 2F_{\lceil n/2 \rceil} F_{\lceil n/2 \rceil - 1} & n \geq 2 \quad n\text{jämn} \end{cases}$$

För $n = 1 \ldots 25$ utförs följande antal anrop

n	1	2	3	4	5	6	7	8	9	10	11	12	13
anrop	1	3	5	5	9	9	11	11	15	15	19	19	21
n	14	15	16	17	18	19	20	21	22	23	24	25	
anrop	21	23	23	27	27	31	31	35	35	39	39	41	

Vi önskar nu en formel som anger antalet anrop för givet n. Även om man inte helt kommer att finna denna formel kan vi med hjälp av funktionen

$$T(n) = 2T(n/2) + 1 \qquad T(1) = 1$$

på ett ungefär att kunna bestämma n. Vi har dålig träning på att lösa *rekursionsekvationer* för hand och tar därför till Mathematica

```
RSolve[{T[1]==1, T[n]==2 T[n/2]+1}, T[n], n]
  T[n]=2n-1
```

Det viktiga här är att vi får komplexitetsfunktionen $O(n)$. Problemet går alltså att med denna metod att lösa på linjär tid.

Tittar vi på det inledande exemplet, *största värdet i en array* som lyder under samma rekursionsekvation, så ger den ett mer exakt resultat. Komplexitetsfunktionen är här, som väntat, $O(n)$

När det gäller problemet med *Exponentiering* får vi följande resultat för $b = 1 \ldots 25$

n	1	2	3	4	5	6	7	8	9	10	11	12	13
mult	0	1	2	2	3	3	4	3	4	4	5	4	5
n	14	15	16	17	18	19	20	21	22	23	24	25	
mult	5	6	4	5	5	6	5	6	6	7	5	6	

Rekursionsekvationen blir

```
RSolve[{T[1] == 1, T[n] == T[n/2] + 1}, T[n], n]
  T[n] -> 1 + Log[n]/Log[2]
```

som ger

$$T_n = \frac{\ln 2 + 2 \ln n}{\ln 2}$$

med $O(\log n)$

För *Mergesort* får vi följande rekursionsekvation

$$T(n) = 2T(n/2) + n$$

som när vi löser den med Mathematica ger

```
RSolve[{T[n]==2T[n/2]+n, T[1]==1}, T[n], n]
  T[n] -> n (1 + Log[n]/Log[2])
```

som ger

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

som i sin tur ger $O(n \log n)$

10.6.2 Övningstentamen

Som en försmak till vad som komma skall får du här fem uppgifter. Du får 2 poäng för varje uppgift du löser under förutsättning att du löser dem under detta föreläsningstillfälle.

UPPGIFT 10.2
Delvis sant Vilket är heltalet x om precis 3 av dessa påstående är sanna?

- $3x > 35$
- $7x \geq 43$
- $2x \leq 99$
- $x \geq 21$
- $5x \geq 51$

UPPGIFT 10.3
Talormen Till vänster i figuren en kvadrat med några tal givna. Skriv ett program som

	11	10	
	7		

12	11	10	1
13	8	9	2
14	7	6	3
15	16	5	4

Figur 10.6:

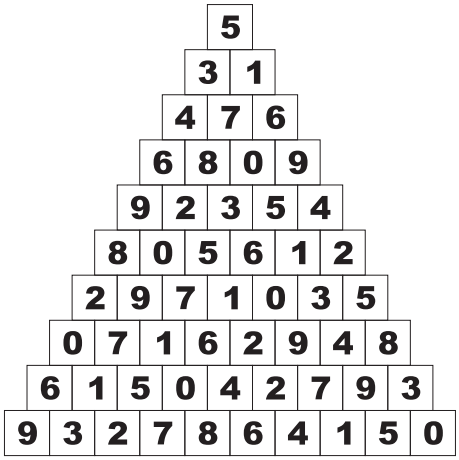
fyller i de tomma rutorna, så att det bildas en väg från 1 till $n \times n, n$ (i det här fallet 16). Det är tillåtet att förflytta sig en ruta åt gången *uppåt, nedåt, åt vänster* eller *åt höger*.

Data finns på filerna `talorm1.txt` och `talorm2.txt`. På första raden finns ett tal $n \leq 6$ som anger kvadratens storlek. Därefter n rader med n tal på varje. 0 betecknar tom ruta. Körningsexempel:

```
12 11 10  1
13  8  9  2
14  7  6  3
15 16  5  4
```

UPPGIFT 10.4

Pyramidvandring Här gäller det att ta sig från toppen på pyramiden till basen, varje gång



Figur 10.7:

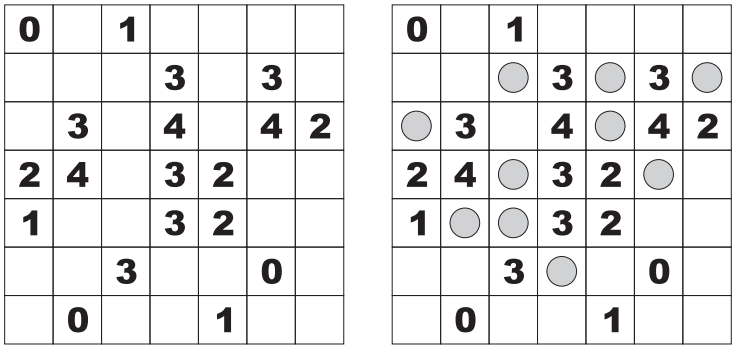
genom att välja *snett ned åt vänster* (V) eller *snett ned åt höger* (H). När man nått basen ska man ha 'besökt' en ruta med alla talen 0 till 9 precis en gång.

Data finns på filen `pyramid.txt`. Filen består av 10 rader. På första raden finns ett tal, pyramidens topptal. På andra raden två tal och så vidare till sista raden med sina 10 tal.

Resultatet skrivs ut som en sträng av 10 tecken – H och V

UPPGIFT 10.5

MineSweeper Skriv ett program som placerar ut 10 bomber i tomma rutor så att det finns



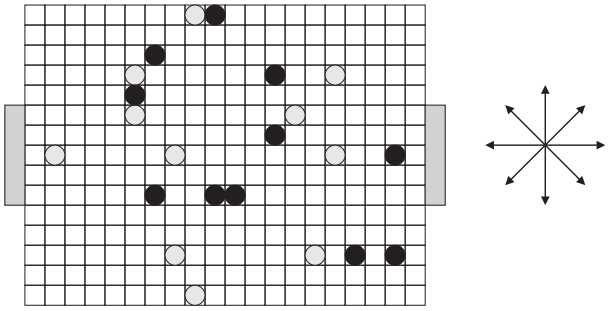
Figur 10.8:

så många bomber runt en ruta med tal, som talet anger. Se figur 10.8.

Data finns på filerna `bomber1.txt` till `bomber4.txt`. Filerna innehåller alla 7 rader med 7 tal på varje. Talet −1 anger *tom ruta*. Valfri utskrift, här ett exempel:

```
0 . 1 . . . .
. . B 3 B 3 B
B 3 . 4 B 4 2
2 4 B 3 2 B .
1 B B 3 2 . .
. . 3 B . 0 .
. 0 . . 1 . .
```

UPPGIFT 10.6
Fotbollsmatchen



Figur 10.9:

Detta är en situation i en fotbollsmatch. Utspel sker från målvakten (spelaren närmast målet). Passning sker till någon av de egna spelarna i någon av de åtta riktningarna som visas i kompassrosen till höger. Ingen annan spelare, vare sig med eller motspelare får då stå i vägen. Innan den elfte mannen i laget förpassar bollen över mållinjen har alla elva spelarna vidrört bollen exakt en gång.

Så här ser filen `match.txt` ut

```
======================
=........VS..........=
=....................=
=......S.............=
=.....V......S..V....=
=.....S..............=
#.....V.......V......#
#............S.......#
#.V.....V.......V..S.#
#....................#
#......S..SS.........#
=....................=
=....................=
=.......V......V.S.S.=
=....................=
=........V...........=
======================
```

Båda lagen får var sin chans. Hur slutade matchen?

10.6.3 Att tävla i programmering

Här listar jag några länkar till hemsidor för den som är intresserad av att tävla i programmering, eller hitta bra problem att öva sig på. Sidorna innehåller bland annat tusentals programmeringsproblem

- ProgrammeringsOlympiaden `http://www.csc.kth.se/contest/ioi/`
- Tävlingsprogrammering `http://sv.wikibooks.org/wiki/T%C3%A4vlingsprogrammering`
- UVa Online Judge `http://uva.onlinejudge.org/`
- UVA toolkit `http://www.uvatoolkit.com/problemssolve.php`
- TopCoder `http://community.topcoder.com/tc`
- International Olympiad in Informatics `www.ioinformatics.org`