Kapitel 12

Sortering

12.1 Sortering

Sortering är en väl analyserad disciplin inom datalogin. Speciellt i datorernas barndom var sortering nästan huvudaktiviteten, men även idag är sortering viktig. Ordet *sortering* har två betydelser – antingen *att ordna i följd* eller *att dela upp efter typ*. För oss gäller här den första definitionen.

Ett antal *poster* ska sorteras efter en för varje post given *nyckel*. Sorteringen kan ske i antingen *stigande* eller *fallande* ordning.

Det finns ett stort antal *sorteringsalgoritmer*. En del är bättre (effektivare) än andra. I stället för att rekommendera en – den bästa – och alltid använda den, kommer vi här att titta på flera olika och försöka utvärdera dem. Om inte annat, kommer vi genom dessa diskussioner, att förstå att valet av algoritm i vissa situationer kan vara avgörande för om programmet kommer att bli användbart eller inte.

Beroende av storleken hos den datamängd som ska sorteras avgör vi (eller rättare sagt programspråket och hårdvaran) om sorteringen i sin helhet kan ske i primärminnet – *intern-sortering* eller om sekundärminnet (hårddisken) måste användas – *extern-sortering*.

12.1.1 Testdata

För att kunna testa de olika algoritmerna behöver vi data att testa med. Normalt utgör *nyckeln*, efter vilken sorteringen ska ske, bara en liten del av hela *posten*. Vi ska dock i de inledande avsnitten ägna oss åt poster som endast består av en nyckel – ett heltal av typen `int`.

För att få fram dessa nycklar kan vi, i brist på verkliga data, inne i programmet, med slumpens hjälp generera testdata, som varierar mellan körningarna. Här har vi i stället låtit skapa en *binärfil* med talen `1...1 000 000` i en slumpmässig ordning. Filen heter `T.DAT`.

För några av de algoritmer vi här kommer att presentera är det inte lönt att ens försöka sortera en liten del av dessa, på grund av att dessa är så ineffektiva.

Eftersom talen är lagrade binärt på filen kan de läsas in i ett svep till en array med hjälp av

```
fread(&a,100000,sizeof(int),fil);
```

100 000 binära heltal läses in till arrayen `a`.

12.2 Enkel sortering

Denna algoritm, som kanske är den enklaste av alla sorteringsalgoritmer att minnas, är också den klart mest ineffektiva. ”Algoritmen” jämför alla par av nycklar och om, under en jämförelse, nyckeln med *lägst index* är större än den andra nyckeln, så byter posterna plats.

Är det n nycklar som ska sorteras så kommer det att göras $n(n-1)/2$ jämförelser. Hur många byten som kommer att ske beror förstås på den ursprungliga ordningen i arrayen. Är allting redan sorterat kommer inga byten alls att ske.

Alla sorteringsalgoritmer har ett ”värsta fall” – den ursprungliga ordning som tar längst tid att ordna. Vilken är den sämsta array denna algoritm kan få från start?

```
1  void enkel_sort(int t[],int n){
2    int i,j,tmp;
3    for(i=0;i<=n-2;i++){
4      for(j=i+1;j<=n-1;j++)
5        if(t[i]>t[j]){
6          tmp=t[i];
7          t[i]=t[j];
8          t[j]=tmp;
9        }
10   }
11 }
```

12.2.1 Tidtagning

Av antalet jämförelser, som görs för n tal, att döma verkar arbetet växa efter $O(n^2)$ för algoritmen ovan. För att mer exakt ta reda på detta polynom vill vi på något sätt kunna mäta mängden av arbete.

Genom följande programkod kan man mäta exekveringstiden för den kod, som står på punkternas plats.

```
1  #include <stdio.h>
2  #include <time.h>
3  void main(void){
4     clock_t start;
5     double tid;
6
7     start=clock();
8     ...
9     tid=(clock()-start)/CLK_TCK;
10    printf("Tid %6.3f\n",tid);
11 }
```

Datorn läser av klockan cirka 18 gånger/sekund, vilket betyder att man inte kan mäta tiden bättre än med 0.055 sekunder när.

Låter vi den enkla algoritmen ovan sortera från 2 000 till 20 000 tal från filen `T.dat` får vi följande tider:

Antal	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Tid (s)	0.11	0.49	1.10	1.98	3.08	4.45	6.04	7.91	10.00	12.36

Med hjälp av *Mathematica* och funktionen `Fit` kan vi nu bestämma det *andragradspolynom* som (i minsta kvadratmetodens mening) bäst beskriver de uppmätta punkterna.

```
data={{2,0.11},{4,0.49},{6,1.10},{8,1.98},{10,3.08},
      {12,4.45},{14,6.04},{16,7.91},{18,10.00},{20,12.36}};
Fit[data, {1, n, n^2}, n]

-0.0111667 + 0.000685606 n + 0.0308807 n^2
```

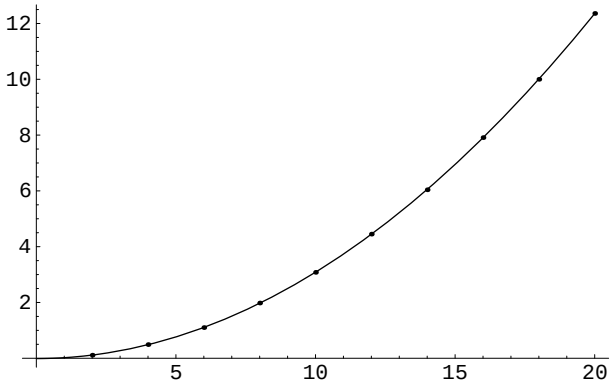
När vi sedan plottar punkterna tillsammans med den erhållna funktionen

$$f(x = \frac{n}{1000}) = 0.0308x^2 + 0.000686x - 0.0111$$

ser det hela väldigt bra ut!

Kan man då använda denna sorteringsalgoritm i seriösa sammanhang? Allt beror på hur man tänker använda den. Ska man sortera 1000 tal vid morgonens uppstart av systemet, så kan det väl inte betyda något om kafferasten blir 0.1 sekunder längre. Ska däremot sortering av 6000 tal ske var 30:e sekund kanske man inte har råd att ”slösa bort” en hel sekund på en dålig algoritm. Ska till sist sortering av 12000 tal ske varje sekund – ja då är förstås denna algoritm omöjlig att använda.

Funktionen som den bestämts ovan är relaterad till en viss dator och till ett visst programspråk och till och med till en viss indatafil, så därför är koefficienterna inte speciellt viktiga.



Figur 12.1: *Den tid som krävs för att sortera* n *tal är proportionell mot* n^2

Däremot är det viktigt att känna till att *exekveringstiden växer med kvadraten på antalet tal som ska sorteras*. I följande avsnitt ska vi se på sorteringsalgoritmer med andra egenskaper.

UPPGIFT 12.1
Handräkning. Beräkna med penna och papper

- Hur lång tid tar det att sortera 1 000 000 tal?
- Hur många tal kan man sortera på en minut?
- Kör programmet med 30 000 tal och jämför den erhållna tiden med den från formeln.

UPPGIFT 12.2
Andra datorer. Datorn som använts för att mäta tiden för sorteringarna ovan (och i resten av kapitlet) arbetar med frekvensen 400 Mhz. För tabellen nedan har två andra datorer, D1 och D2 exekverats med samma program. Bestäm med vilka frekvenser dessa arbetar.

Antal	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Tid (s) D1	0.27	0.99	2.20	3.84	5.99	8.68	11.75	15.32	19.22	23.68
Tid (s) D2	0.06	0.17	0.39	0.77	1.15	1.65	2.25	2.97	3.74	4.62

12.3 Sök minsta och byt

Denna algoritms arbetssätt förstår vi direkt genom att titta på koden. Skillnaden mellan *sök minsta och byt* och den tidigare är att inga byten sker förrän man funnit det minsta talet. Detta sparar förstås tid.

```
1  void sok_byt(int t[],int n){
2     int i,j,minp,min;
3     for(i=0;i<n;i++){
4        min=t[i]; minp=i;
5        for(j=i+1;j<n;j++)
6           if(t[j]<min){
7              min=t[j];
8              minp=j;
9           }
10       t[minp]=t[i]; t[i]=min;
11    }
12 }
```

UPPGIFT 12.3
Mät effektiviteten. Tyvärr är även denna algoritm kvadratisk, men den borde vara effektivare än den inledande.

Bestäm med en teknik liknade den för *enkel sortering* den funktion som beskriver exekveringstiden för *sök minsta och byt*. Plotta sedan punkterna tillsammans med funktionen med hjälp av *Mathematica*.

Utgå ifrån filen `t.dat`, läs in och sortera i tur och ordning 2000, 4000,...20000 tal och mät upp tiden.

För att resultaten ska bli jämförbara med varandra, krävs förstås att du mäter upp tiderna för *enkel sortering* igen i den miljö du arbetar (kompilator och dator).

12.4 Insticksortering

De tredje och sista algoritmen med ungefär samma effektivitet.

3	5	7	2	8	1	4
2	3	5	7	8	1	4
1	2	3	5	7	8	4
1	2	3	4	5	7	8

Figur 12.2:

De 7 talen som ska sorteras återfinns på första raden i figuren ovan. Målet är att sortera dem i stigande ordning. Vi söker, från vänster, upp det första talet som inte ligger i ordning – talet 2, söker tillbaka till den plats på raden där det ska ”stickas in”. Nu råkade det bli på

första platsen. Nästa tal som inte ligger i ordning är 1, som även det kommer att ”stickas in” först i raden. Sorteringen fullbordas sedan genom att, söka upp, leta bakåt och placera in, talet 4 på rätt plats.

```
1  void instick(int t[],int n){
2     int i,j,v;
3     for(i=1;i<n;i++){
4        v=t[i];
5        j=i;
6        while(j>0 && t[j-1]>v){
7           t[j]=t[j-1];
8           j--;
9        }
10       t[j]=v;
11    }
12 }
```

Förutsättningen för att det ska hända något under ett varv i for-loopen är att `t[i-1]<t[i]`

Detta inträffar första gången då `i=3` i vårt exempel. Talet 2 rutschar nu fram till cell nr 0 och de framförvarande talen får backa ett steg.

Nästa gång byte kommer att ske är då `i=5`. Även denna gång kommer talet 1 att stegvis flyttas ända fram till cell nr 0.

När så `i` kommer till slutmålet 6 återfinns talet 4 där, som sedan transporteras till sin rätta plats genom while-loopen och arrayen är sorterad!

Som för de två andra rutinerna mäter vi så upp tiderna för att bestämma funktionen.

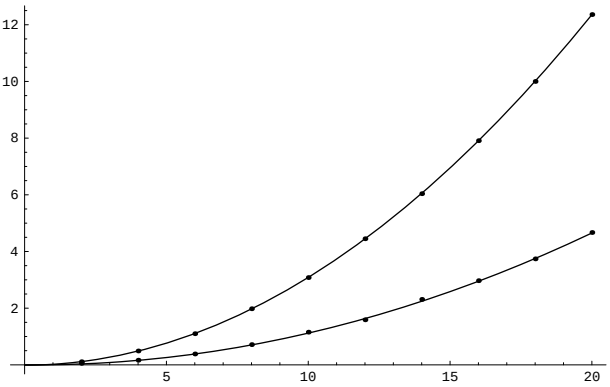
Antal	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Tid (s)	0.05	0.16	0.38	0.71	1.15	1.59	2.31	2.97	3.74	4.67

Funktionen, som vi bestämmer precis som tidigare med hjälp av Mathematica, blir

$$f(x = \frac{n}{1000}) = 0.0120x^2 + 0.00906x + 0.0145$$

12.5 Shellsort

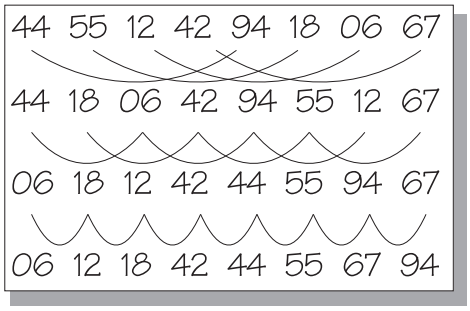
Efter de tre inledande algoritmerna inställer sig nu frågan om det finns någon sorteringsalgoritm, som klart överglänser dessa? De tre nästkommande, lite mer komplicerade metoderna kommer att visa detta.



Figur 12.3: *Instickssortering är närmare tre gånger så snabb som den enkla rutinen från inledningen, men fortfarande är det så att exekveringstiden växer med kvadraten på antalet tal som skall sorteras.*

Namnet *ShellSort* leder tankarna till snäckor, men liknelsen finns inte, eftersom namnet har sitt ursprung från uppfinnaren *D. L. Shell*, som konstruerade algoritmen 1959.

Nedan följer ett försök att förklara arbetssättet, som påminner om *instickssortering*.



Figur 12.4: *En illustarion, som försöker visa hur ShellSort fungerar*

- Först sorteras de 4 paren förbundna av bågarna i översta raden.
- Bågarnas längder blir sedan 2 och eftersom det är 8 tal som ska sorteras kommer varje kedja att innehålla 4 tal. Dessa kedjor sorteras sedan med hjälp av instickssortering.
- I sista steget är bågarnas längd 1 och genom en instickssortering som i förra avsnittet kommer så hela arrayen att bli sorterad.

Eftersom allt avslutas med en *instickssortering*, så måste förarbetet – sorteringen av de inledande kedjorna – vara ordentligt arbetsbesparande. Det är precis vad som är fallet. Genom att i grova drag samla de små talen i början och de stora i slutet av arrayen, så tar det inte så lång tid för instickssorteringen att finna den rätta platsen för ett visst tal.

Ingen har lyckats analysera *ShellSort*. Det betyder att man inte riktigt vet hur arbetet växer

då n, antalet tal, ökar. Ett förslag är att funktionen skulle ha utseendet $f(n) = kn^a$, där $1.2 \leq a \leq 1.3$ På vägen mot att bestämma k och a tar vi som vanligt upp tiderna:

Antal	**Tid** (s)
50000	0.11
100000	0.27
150000	0.44
200000	0.60
250000	0.82
300000	0.99
350000	1.21
400000	1.43
450000	1.65
500000	1.87

Vilken enorm förbättring! Att sortera 500 000 tal på mindre än 2 sekunder – kan det göras snabbare på samma dator?

```
1  void shellsort(int t[],int n){
2     int i,j,h,v;
3
4     h=1;
5     do{
6        h=3*h+1;
7     }while(h<=n);
8     do{
9        h=h/3;
10       for(i=h;i<n;i++){
11          v=t[i];
12          j=i;
13          while(t[j-h]>v){
14             t[j]=t[j-h];
15             j=j-h;
16             if(j<h) break;
17          }
18          t[j]=v;
19       }
20    }while(h!=1);
21 }
```

Funktionen ovan skiljer sig lite från exemplet i figuren. I figuren används talen $\ldots, 8, 4, 2, 1$ för bågarnas längd. I algoritmen ovan används $\ldots, 40, 13, 4, 1$, som lär ska var en bättre serie. Ingen vet dock vilken som är den bästa!

4–7 Rutinen inleds med att bestämma startvärdet för `h`.

8–20 För varje varv i do-loopen används den ”båglängd”som bestäms av `h`.

9–18 Resten av rutinen är mycket lik instickssortering. Enda skillnaden är alltså att steget, ”bågen”, inte alltid är 1

Den följd som har visat sig fungera bäst i praktiken är en följd av *Sedgewick*. Följden ger en $O(N^{4/3})$ exekveringstid i sämsta tänkbara situation (*worst-case*-running time). I genomsnitt (*average-case* running time) har bågvalet en uppskattat komplexitet på $O(N^{7/6})$. Följden ser ut som $\{1, 5, 19, 41, 109, ...\}$, elementen i följden är antingen på formen $9 \cdot 4^i - 9 \cdot 2^i + 1$ eller $4^i - 3 \cdot 2^i + 1$.

12.5.1 Profiler

Vi återknyter bekantskapen med *profiler* för att studera vilken del av koden som tar största tiden.

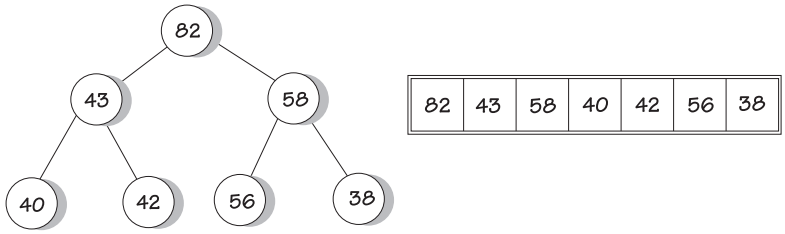
Talen till vänster om koden anger *tid i sekunder* och *antalet gånger raden exekverats*. Observera att denna körning gjorts på en ”slöare burk” och att summan av tiderna därför inte överensstämmer med tiden i den tidigare tabellen. Lägg märke till att nästan precis 50% av exekveringstiden har gått åt för att exekvera rad 13, 14, 15 och 16. Raderna 11 och 12 innehåller båda vanliga tilldelningar. Varför drar rad 11 mer tid?

```
1  0.0000 1     void shellsort(int t[],int n){
2                 int i,j,h,v;
3
4  0.0000 1         h=1;
5                 do{
6  0.0000 9            h=3*h+1;
7  0.0000 9          }while(h<=n);
8                 do{
9  0.0000 9            h=h/3;
10 0.0000 9            for(i=h;i<n;i++){
11 0.0910 75243           v=t[i];
12 0.0666 75243           j=i;
13 0.2659 232250          while(t[j-h]>v){
14 0.2832 161346             t[j]=t[j-h];
15 0.1467 161346             j=j-h;
16 0.1499 161346             if(j<h) break;
17                        }
18 0.0900 75243           t[j]=v;
19                     }
20 0.0000 9         }while(h!=1);
21 0.0000 1      }
```

12.6 Heapsort

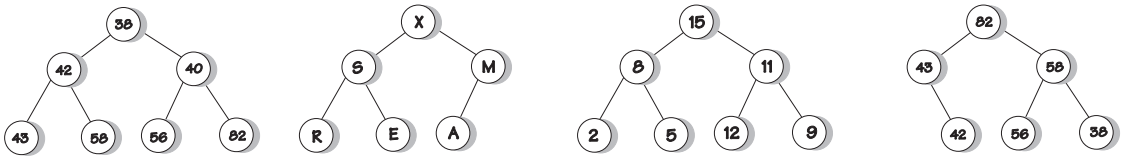
En *heap* är ett *komplett binärt träd* – ett binärt träd där alla löv finns på nivå `n` eller `n – 1` och där raden av löv på nivå `n` är fylld från vänster till höger.

För en heap gäller att varje *nod* har en *nyckel* (ett värde), som är *extremare* (större eller mindre) än dess *förälder*. Det betyder att den största eller den minsta nyckeln finns i trädets rot. En nyckel hos ett barn kan vara lika stor som nyckeln hos föräldern.



Figur 12.5: *En heap, vilket kan avgöras genom att betrakta barnen till varje förälder. Barnen är här mindre än föräldern*

Tack vare att en heap är så regelbunden, till sin karaktär, kan den enkelt lagras i en array. För en godtycklig *förälder* i position `i` i arrayen finns det *vänstra barnet* i position `2i` och det *högra barnet* i position `2i + 1`. Jämför arrayen med trädet i figur 12.5. Detta ger en mycket enkel datastruktur, med enkla och snabba operationer för att traversera trädet.



Figur 12.6: *Från vänster: En heap där en förälder alltid är mindre än sina barn. En heap där nycklarna är bokstäver och där föräldern är störst. Ingen korrekt heap därför att nycklarna inte ligger i rätt ordning. En till formen felaktig heap.*

Enda nackdelen med denna implementation kan vara att den maximala storleken på heapen måste bestämmas på förhand, men detta är oftast inget större problem.

I **heapsort** använder man sig förstås (av namnet att döma) av en heap. I tabell 12.7 ska vi följa hur *heapsort* fungerar. Första raden i tabellen anger index för arrayen. I vårt exempel ska vi sortera de `15` heltal, som ursprungligen ligger i den ordning, som tabellens andra rad visar.

Från programkoden ser vi att funktionen `heap_sort` består av två `for`-loopar. Den första skapar under sina 7 varv, med hjälp av funktionen `sift` en heap av arrayen `a`.

`l`, som är en av parametrarna till `sift` väljs som fader. Under första varvet är `l=6`. De två barnen finns då i index 13 och 14. `sift` ordnar så att dessa tre tal bildar en korrekt heap. Fadern flyttas nu vartefter mot lägre index. Nya ”småheapar” betraktas och ordnas

av `sift`. När index når till fadern i index 2 bildas den betraktade heapen av både barn och barnbarn. Till sist, i första steget, har index nått 0 och hela arrayen med alla 15 talen ses nu som en heap och den sista justeringen kan göras.

Den andra `for`-loopen sorterar till sist talen i den ordning vi önskar. Det största talet finns nu i roten och kan byta plats med det sista. Sedan ska den nu 14 tal stora heapen justeras. Efter justeringen ligger det näst största talet i roten. På samma sätt kan detta byta plats med det näst sista och vi har kommit ytterligare ett steg mot målet...

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
3	10	12	5	7	13	**2**	9	15	1	6	14	8	**4**	**11**
3	10	12	5	7	**13**	11	9	15	1	6	**14**	**8**	4	2
3	10	12	5	**7**	14	11	9	15	**1**	**6**	13	8	4	2
3	10	12	**5**	7	14	11	**9**	**15**	1	6	13	8	4	2
3	10	**12**	15	7	**14**	**11**	9	5	1	6	**13**	**8**	**4**	**2**
3	**10**	14	**15**	**7**	13	11	**9**	**5**	**1**	**6**	12	8	4	2
3	**15**	**14**	**11**	**7**	**13**	**10**	**9**	**5**	**1**	**6**	**12**	**8**	**4**	**2**
15	14	13	11	7	12	10	9	5	1	6	3	8	4	2
15	14	13	11	7	12	10	9	5	1	6	3	8	4	2
14	13	12	11	7	6	10	9	5	1	2	3	8	4	**15**
13	12	7	11	5	6	10	9	4	1	2	3	8	**14**	**15**
12	11	7	10	5	6	8	9	4	1	2	3	**13**	**14**	**15**
11	10	7	9	5	6	8	3	4	1	2	**12**	**13**	**14**	**15**
10	9	7	8	5	6	2	3	4	1	**11**	**12**	**13**	**14**	**15**
9	8	7	3	5	6	2	1	4	**10**	**11**	**12**	**13**	**14**	**15**
8	7	6	3	5	4	2	1	**9**	**10**	**11**	**12**	**13**	**14**	**15**
7	6	5	3	1	4	2	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
6	5	4	3	1	2	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
5	4	2	3	1	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
4	3	2	1	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
3	2	1	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
2	1	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

Figur 12.7: *Genom att följa denna illustration är det meningen att du ska komma närmare en förståelse av hur HeapSort fungerar*

```
1  void sift(int a[],int l, int r){
2     int i=l,j=2*l+1,x=a[l];
3     while(j<=r){
4        if(j<r)
5           if(a[j]<a[j+1])
6              j++;
7        if(x>=a[j]) break;
8        a[i]=a[j];
9        i=j;
10       j=2*i;
11    }
12    a[i]=x;
13 }
14
15 void heap_sort(int a[],int n){
16    int l,r,x;
17    for(l=(n-1)/2-1;l>=0;l--)
18       sift(a,l,n-1);
19    for(r=n-2;r>=0;r--){
20       x=a[0];
21       a[0]=a[r+1];
22       a[r+1]=x;
23       sift(a,0,r);
24    }
25 }
```

Algoritmen är uppdelad i två funktioner, `heapsort` och `sift`.

Under den första for-loopen 17–18 byggs en heap upp med det *största elementet* i roten, i cell `0`.

I nästa for-loop, 19-24, byts talet i roten, det största återstående, med ett längre bak i arrayen. Därefter justeras heapen inför nästa varv i loopen.

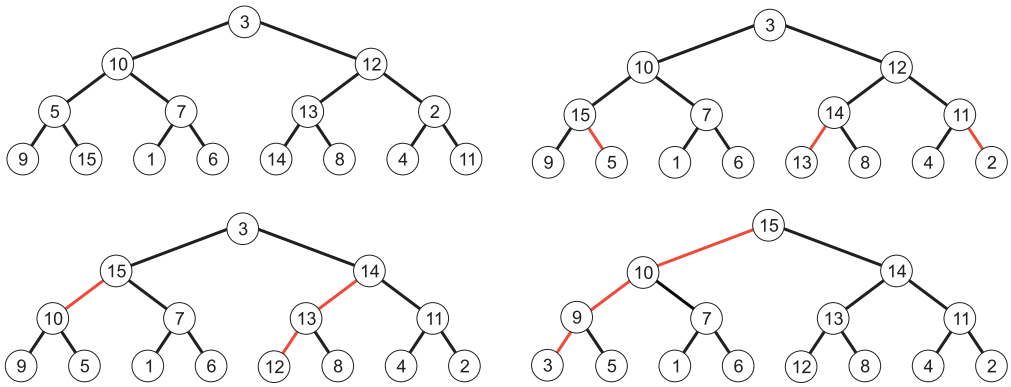
Nu är vi förstås intresserade av att få veta om *HeapSort* är snabbare än *ShellSort*. Här följer resultatet:

Antal	**Tid** (s)
50000	0.05
100000	0.16
150000	0.33
200000	0.49
250000	0.60
300000	0.77
350000	0.88
400000	1.04
450000	1.26
500000	1.37

Även om dessa mätningar visar att *HeapSort* är snabbare än *Shellsort* för området $n \leq 500000$ och detta val av båglängd, så är det inte givet att det alltid är så. *HeapSort* har komplexitet $O(n \log n)$ som är den bästa komplexiteten vi har sett hittills, men i praktiken är den inte så snabb som *Shellsort* för *Sedgewick's* val av båglängd.

12.7 Heapsort en gång till

HeapSort en gång till



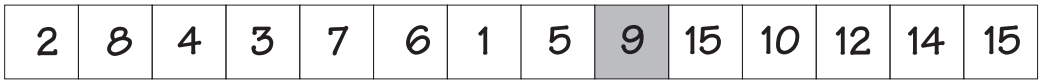
Figur 12.8:

```
#include <stdio.h>
void swap(int *a,int *b){
  int t;
  t=*a;
  *a=*b;
  *b=t;
}
void fixDown(int a[],int k,int N){
  int j;
  while(2*k<=N){
    j=2*k;
    if(j<N && a[j]<a[j+1])
      j++;
    if(a[k]>=a[j])
      break;
    swap(&a[k],&a[j]);
      k=j;
  }
}
int main(void){
  int a[]={0,3,10,12,5,7,13,2,9,15,1,6,14,8,4,11};
  int k,N=15;
  for(k=N/2;k>=1;k--)
    fixDown(a,k,N);
  while(N>1){
    swap(&a[1],&a[N]);
    fixDown(a,1,--N);
  }
}
```

12.8 QuickSort

Den berömda algoritmen *QuickSort* är ett nytt exempel på strategin *divide-and-conquer*, för att ordna en lista. Den har visat sig vara den snabbaste sorteringsalgoritmen i praxis av alla kända sorteringsalgoritmer. Den komplexitesfunktionen är, som för flera andra metoder. Den sämsta tänkbara exekveringstiden är $O(N^2)$, men den situationen kan göras väldig osannolik med rätt implementation.

Algoritmen, konstruerad av *C A R Hoare*, utför en serie rekursiva anrop där en lista delas i mindre och mindre dellistor. Delningen sker kring ett så kallat pivot-element. I varje steg väljs, till exempel, det värde som ligger i dellistans mitt som pivot-element. Dellistan delas nu upp i två nya listor så att pivot-elementet till sist hamnar på sin rätta plats i listan. I den 'lägre' dellistan finns bara element som är mindre än pivot-elementet och i den 'högre' dellistan hamnar de element som är större än (eller lika med) pivot-elemenetet.



Figur 12.9: *Vektorn ovan har sorterats en gång och har pivotelementet 9.*

Har man tur ligger pivotelementet i mitten av de tal som ska sorteras men det är långtifrån alltid fallet.

Här ett idealiserat exempel med $15, 2^4 - 1$, element utan dubbletter:

Figur 12.10:

Ovan i figur 12.10 har vi den ursprungliga listan. Av den 'goda fen' får vi reda på att det 8:e största elementet i listan är 8 och vi väljer därför detta element som pivot och placerar in det mitt i listan.

Figur 12.11:

Vi går nu igenom alla de övriga elementen och om det är < 8 placerar vi det till vänster om pivot-elementet, annars till höger. I figur 12.11 ser vi resultatet – första steget.

Nu är det dags att finna två nya pivot-element, ett till den 'låga' listan och ett till den 'höga'. Talen är 4 respektive 12. De båda listorna hanteras sedan på samma sätt. Den 'låga' faller, precis som den 'höga' sönder i vardera två listor, där talen är sorterade efter samma princip som i det första fallet. Figur 12.12 visar resultatet efter andra steget.

Figur 12.12:

Figur 12.13:

När vi nu väljer ut fyra nya pivot-element $2, 6, 10, 14$ får vi fyra nya listor att dela upp. När det är gjort är arrayen sorterad (figur 12.13).

Tyvärr är situationen inte alltid denna, det vill säga, antalet element som ska sorteras är inte lika med $2^n - 1$ och, framför allt, vi vet inte vilket element som fungerar bäst som pivot och vi har inte tid eller resurser att ta reda på det heller.

Här följer ett mer realistiskt exempel, som samtidigt visar på en teknik som direkt kan kodas:

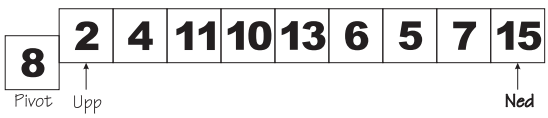
Figur 12.14:

Till skillnad från *MergeSort*, som kopierar element från och till temporära arrayer, så utförs allt jobb i QuickSort i den ursprungliga arrayen.

Steg 1 Listan (figur 12.14, som vi kallar `v` har index `0...9`, `first` är just nu `0` och `last` sätter vi till `10`. Med formeln

$$\text{mid} = \frac{\text{last} + \text{first}}{2}$$

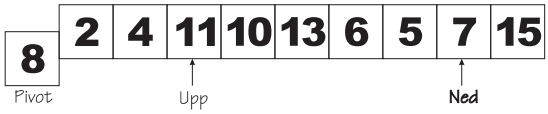
som ger `mid`$=(0+10)/2=5$ får vi `v[mid]=8`, som blir vårt första pivot-element. Detta element ska nu användas för att separera talen i två listor S_1 (de lägre talen) och S_2 (de högre). Här sker nu en teknisk finurlighet: Vi byter plats på pivot-elementet och talet `v[first]`. Det är talen till höger om pivot-elementet, som ska delas upp i S_1 och S_2. Två index blir aktuella `upp` och `ned`. De ges ursprungligen värden `upp`$=1$ och `ned`$=9$. Se figur 12.15



Figur 12.15:

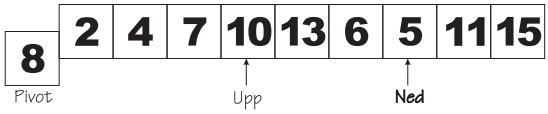
Index `upp` kommer under arbetets gång att flytta sig mot högre index index `ned` flyttar sig i andra riktningen. Arbetet börjar med att `upp` ökas tills `v[upp]>pivot`. Detta inträffar i

vårt exempel för `v[3]=11`. Nu har turen kommit till index `ned`, som söker efter tal mindre än `pivot`. I figur 12.16 har vi nått fram till detta läge.



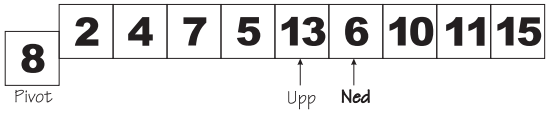
Figur 12.16:

Ingenting känns nu mer naturligt än att låta dessa element byta plats. Därefter bör `upp` och `ned` uppdateras, så att de pekar ut nya element. Se figur 12.17



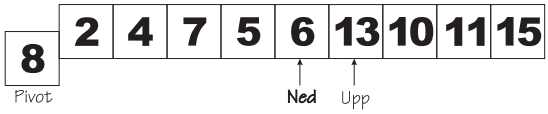
Figur 12.17:

Sedan är det `upp` tur igen, att söka uppåt i listan och kommer fram till att talet `10` i `v[4]` ligger fel. När `ned` kommit fram till att talet `5` i `v[7]` ligger fel, är det dags för byte igen och uppdatering av `upp` och `ned`. Se figur 12.18



Figur 12.18:

Vi ser att det omedelbart är dags för byte igen och efter ny uppdatering av `upp` och `ned` har vi nått fram till följande situation i figur 12.19

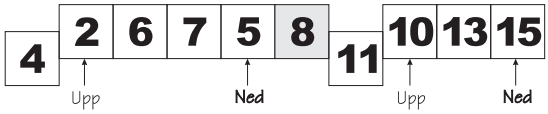


Figur 12.19:

`upp` och `ned` har nu passerat varandra och det är dags att avbryta steg 1, med att låta det element som `ned` pekar på `v[5]` byta plats med pivot-elementet `v[0]`! (observera att steget också kan brytas genom att `upp`=`ned`). Efter steg 1 har listan fått det utseende som visas i figur 12.20

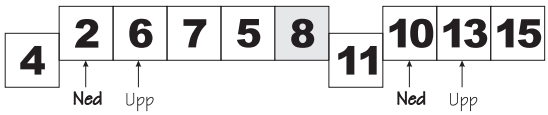
Steg 2 och **steg 3** innebär nu att med samma teknik ta sig an listorna `v[0]... v[4]` och `v[6]...v[9]`. `v[5]` ligger ju fast. De två pivot-elementen i dessa listor blir nu `v[2]=4` och `v[8]=11`. Laddat för nya sökningar i figur 12.21

Figur 12.20:



Figur 12.21:

Genom att behandla en lista i taget når vi efter steg 2 och 3 fram till läget i figur 12.22. Inga ändringar görs innan pivot-elementen flyttas in

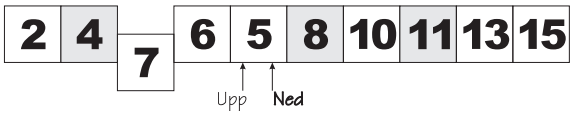


Figur 12.22:

Vi har nu följande situation (figur 12.23), där 3 tal nu är på sin rätta plats, återstår 4 listor, varav två har längden ett, att sortera med samma metod. I praktiken avbryter man tidigare och övergår till att sortera med till exempel *instickssortering*.

Figur 12.23:

Det enda intressanta som återstår är listan `v[2]...v[4]`. `v[3]=7` blir pivot-element – `v[2]` byter plats med `v[3]`. Därefter kommer `upp` att leta efter ett index `i` där `v[i]>v[2]=7`. Det finns inget sådant utan båda index kommer att stanna som i figur 12.24 innan ens `ned` har hunnit testa.



Figur 12.24:

Till slut är hela arrayen sorterad

Så här skriver vi den första funktionen

Figur 12.25:

```
1 void qsort(int a[],int l,int h){
2   int pl;
3   if(l<h){
4     dela(a,l,h,pl);
5     qsort(a,l,pl-1);
6     qsort(a,pl+1,h);
7   }
8 }
```

Funktionen anropas med adressen till arrayen `a`. Det lägsta `l` respektive det högsta `h` index. `qsort(t,0,n-1)` innebär alltså att arrayen `t` ska sorteras för index `0` till $n-1$.

3 Endast då lägsta index är mindre än högsta finns något att sortera.

4 Funktionen `dela` delar upp arrayen i två delar där `pl` anger den plats i arrayen där det så kallade *pivot*-elementet finns.

5–6 Två anrop av funktionen `qsort`, rekursion alltså. Det enda element som inte får vara med i fortsättningen är det som är placerat i `pl`.

Intuitivt förstår vi att skillnaden mellan `l` och `h` minskar vart efter och att detta leder till att villkoret i if-satsen i 4 så småningom blir falskt.

```
1  void dela(int a[],int l,int h,int *pl){
2     int i,senast,pivot;
3     swap(a[l],a[(l+h)/2]);
4     pivot=a[l];
5     senast=l;
6     for(i=l+1;i<=h;i++){
7        if(a[i]<pivot){
8           senast++;
9           swap(a[i],a[senast]);
10       }
11    }
12    swap(a[l],a[senast]);
13    *pl=senast;
14 }
```

Det är denna funktion som gör jobbet. Först väljer den ut ett *pivot*-element, som förhoppningsvis är det *mittersta talet* bland nycklarna. I denna implementation av *QuickSort* anstränger vi oss inte speciellt för att finna ett bra *pivot*-element utan väljer bara det mittersta talet.

Sedan görs ett antal byten för att få alla tal *mindre än* vårt *pivot*-element till vänster och resten till höger.

3–5 Det mittersta talet, det i `(l+h)/2` flyttas ut till `l`, så länge och kallas alltså för *pivot*-element.

6–11 Resten av talen jämförs nu med *pivot*-elementet och flyttas om talet är mindre till en plats som variabeln `senast` håller reda på.

12 Till sist flyttas *pivot*-elementet in i arrayen igen och nu är alla tal till vänster om detta element mindre och alla till höger större.

13 Den plats där *pivot*-elementet finns rapporteras på detta sätt till den anropande funktionen `qsort`.

Funktionen `swap` är bara en funktion som byter värden mellan två variabler.

```
1 void swap(int *a,int *b){
2    int tmp;
3    tmp=*a; *a=*b; *b=*tmp;}
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
	3	11	6	9	5	12	8	14	1	13	4	15	7	10	2
0 14															
	14	11	6	9	5	12	8	**3**	1	13	4	15	7	10	2
	14	11	6	9	5	12	8	3	1	13	4	**7**	**15**	10	2
	14	11	6	9	5	12	8	3	1	13	4	7	**10**	**15**	2
	14	11	6	9	5	12	8	3	1	13	4	7	10	**2**	**15**
	2	11	6	9	5	12	8	3	1	13	4	7	10	**14**	15
0 12															
	8	11	6	9	5	12	**2**	3	1	13	4	7	10	14	15
	8	**6**	**11**	9	5	12	2	3	1	13	4	7	10	14	15
	8	6	**5**	9	**11**	12	2	3	1	13	4	7	10	14	15
	8	6	5	**2**	11	12	**9**	3	1	13	4	7	10	14	15
	8	6	5	2	**3**	12	9	**11**	1	13	4	7	10	14	15
	8	6	5	2	3	**1**	9	11	**12**	13	4	7	10	14	15
	8	6	5	2	3	1	**4**	11	12	13	**9**	7	10	14	15
	8	6	5	2	3	1	4	**7**	12	13	9	**11**	10	14	15
	7	6	5	2	3	1	4	**8**	12	13	9	11	10	14	15
0 6															
	2	6	5	**7**	3	1	4	8	12	13	9	11	10	14	15
	2	**1**	5	7	3	**6**	4	8	12	13	9	11	10	14	15
	1	**2**	5	7	3	6	4	8	12	13	9	11	10	14	15
2 6															
	1	2	**3**	7	**5**	6	4	8	12	13	9	11	10	14	15
3 6															
	1	2	3	**5**	**7**	6	4	8	12	13	9	11	10	14	15
	1	2	3	5	**4**	6	**7**	8	12	13	9	11	10	14	15
	1	2	3	**4**	**5**	6	7	8	12	13	9	11	10	14	15
8 12															
	1	2	3	4	5	6	7	8	**9**	13	**12**	11	10	14	15
9 12															
	1	2	3	4	5	6	7	8	9	**12**	**13**	11	10	14	15
	1	2	3	4	5	6	7	8	9	12	**11**	**13**	10	14	15
	1	2	3	4	5	6	7	8	9	12	11	**10**	**13**	14	15
	1	2	3	4	5	6	7	8	9	**10**	11	**12**	13	14	15
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Så här arbetar Quicksort. Till vänster i tabellen finns talen som anger mellan vilka index sorteringen av arrayen ska ske. Elementet i mitten i intervallet väljs som *pivotelement* och det är alltid det som flyttas först. Alla byten som sker är markerade med fet stil.

12.8.1 Inbyggd qsort i C

QuickSort finns inbyggd i C och heter då `qsort`. Här följer ett enkelt exempel på hur den används.

```
1 int ordning(const void  *a,const void *b) {
2    if (*(int *)a<*(int *)b) return -1;
3    if (*(int *)a==*(int *)b) return 0;
4    if (*(int *)a>*(int *)b) return 1;
5  }
```

```
1 void main(void){
2 ...
3 qsort(t,n,sizeof(int),ordning);
4 ...
```

Den första parametern är adressen till den array som ska sorteras. Därefter följer parametrar som anger *antalet element* och *elementens storlek*. `ordning` är namnet på en funktion som avgör relationen *störst – minst* mellan två element.

Funktionen `ordning` skrivs av programmeraren. Normalt brukar det var tryggast att ha en förebild när den ska skapas.

12.8.2 Kommentarer

Programmet ovan visar *QuickSorts* grundidé. Till detta kommer funderingar om förbättringar.

- Det finns anledning att ”jobba bort” rekursionen, eftersom vi vet i att rekursion tar tid. I funktionen nedan har rekursionen ersatts med en egen stack.
- Ju mer i mitten pivotelementet är desto bättre. I tabellen ovan ser vi att 8 gör mer nytta än 14. Detta skulle kunna betyda att vi bör lägga ner mer tid på att finna ett bra pivotelement än att bara ta det som råkar stå mitt i intervallet.
- När antalet tal i listan som ska sorteras blir färre, är det troligtvis bättre att använda *insticksortering* i stället för att använda *QuickSort* ända ”ner i botten”. Man kan faktiskt spara ca 15% i exekveringstid på att använda *insticksortering* när antalet element är 10 eller mindre.
- Partisioneringen av arrayen i två halvor kan också göras på många olika sätt, med olik effektivitet. Metoden som har använts i implementationen under är känd metod som man vet ger goda resultat.
- Hur man hanterar dubbletter kan också vara avgörande för effektiviteten.

```
1  void quicksort(int a[],int antal){
2     int i,j,l,r,x,w,s;
3     struct stacktyp {int l,r;} stack[101];
4
5     s=1;
6     stack[1].l=0;
7     stack[1].r=antal-1;
8     do{
9        l=stack[s].l;
10       r=stack[s].r;
11       s--;
12       do{
13          i=l; j=r;
14          x=a[(l+r)/2];
15          do{
16             while (a[i]<x) i++;
17             while (x<a[j]) j--;
18             if (i<=j){
19                w=a[i];
20                a[i]=a[j];a[j]=w;
21                i++; j--;
22             }
23          } while (i<=j);
24          if (j-l<r-i){
25             if (i<r){
26                s++;
27                stack[s].l=i;
28                stack[s].r=r;
29             }
30             r=j;
31          }
32          else {
33             if (l<j){
34                s++;
35                stack[s].l=l;
36                stack[s].r=j;
37             };
38             l=i;
39          }
40       } while (l<r);
41    }while (s!=0);
42 }
```

12.9 BucketSort

Den enklaste och dessutom effektivaste av alla sorteringsrutiner heter *BucketSort*.

Ett okänt antal tal på en binärfil ska sorteras. Talen `t` ligger i intervallet $0 \leq t \leq 32767$. En array, `t`, av typ `int` med storleken 32768 skapas på heapen och nollställs. Från filen läses ett tal `x` i taget. Cellen med index `x` ökas med ett och efter ett par tre tiondels sekunder kan den sorterade filen med över 1 000 000 tal skapas på hårddisken!

Nämn två situationer när denna sorteringsalgoritm inte kan användas.

```
1   t=(char *)calloc(sizeof(char),32768U);
2   while(!feof(infil)){
3      fread(&k,sizeof(int),1,infil);
4      if(!feof(infil))
5         t[k]++;
6   }
```

1 En array skapas på heapen. Genom `calloc` blir den samtidigt nollställd.

4–6 Vi läser in ett tal `k` i taget från filen. Den plats i `t`, som pekas ut av talet `k`, ökas med `1`.

När filen har lästs vet vi att summan av talen i `t` är lika med antalet tal i filen

```
1   for(k=0;k<=32767;k++){
2      if(t[k]>0)
3         for(l=1;l<=t[k];l++)
4            fprintf(utfil,"%u\n",k);
5   }
```

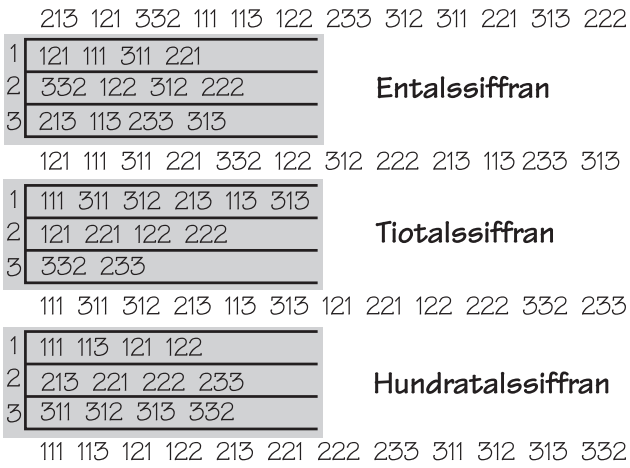
Den yttre loopen går igenom alla cellerna i `t`. Om talet i cellen är > 0, så skriver vi ut lika många tal till filen.

12.10 Radixsort

RadixSort efterliknar den metod man ofta använde för så där en $30-40$ år sedan, när man sorterade hålkort. Illustrationen nedan visar hur det hela gick till.

Vi inleder med ett antal hålkort alla med en tresiffrig nyckel efter vilken vi vill sortera korten. I detta speciella fall (speciellt endast för att hålla nere storleken av figuren) ingår endast tre olika siffror i nycklarna.

- Korten får i tur och ordning ramla ned i en av de tre behållarna, allt beroende på vilken entals-siffra de har.



Figur 12.26: *En sorteringsmetod som användes på den gamla hålkortstiden*

- Korthögarna tas nu ut ur behållarna, med början i behållare 1, och läggs efter varandra för att bilda en ny rad.
- Denna gång får korten falla ner i de tre behållarna beroende på siffran i mitten.
- På samma sätt tar nu korten ut ur behållarna och placeras i en ny rad.
- Sista gången är det så hundratals-siffrorna som bestämmer i vilken behållare korten ska hamna. När vi nu plockar ut korten för sista gången har vi dem sorterade i ordning!

Vi ska nu använda denna teknik för att sortera 50 000 personnummer i filen `elevreg.dat`. En post i filen har utseendet

Fält	**ant tkn**
Personnummer	12
Förnamn	10
Efternamn	12

- Först läser vi in posterna från filen och ”skalar av” personnumret som vi lägger in i en länkad lista på heapen.
- När listan finns på plats anropar vi funktionen `sortera`, som återges här nedan.
- I denna funktion skapar vi tio köer, läser igenom personnumren i listan och lägger in dem i rätt kö, beroende på vilken siffra personnumret har sista positionen.
- När listan är slut finns tio köer, som länkas samman till en lista. Denna nybildade lista genomlöper sedan samma procedur, men nu för en annan position i personnumret.
- Efter 10 genomgångar är de 50 000 personnumren sorterade, vilket på min dator tar 24 sekunder.

```
1  void sortera(element **start,int k){
2     element *s,*g;
3     int siffra,m1,m2;
4
5     s=*start;
6     while(s!=NULL){
7        siffra=s->persnr[k]-'0';
8        laegg_till(*s,siffra);
9        g=s;
10       s=s->naesta;
11       free(g);
12    }
13    m1=0;
14    while(foerst[m1]==NULL)
15       m1++;
16    *start=foerst[m1];
17
18    m2=m1+1;
19    do{
20       while(m2<=9 && foerst[m2]==NULL)
21          m2++;
22       if(m2<=9){
23          sist[m1]->naesta=foerst[m2];
24          m1=m2;
25          m2++;
26       }
27    }while(m2<10);
28    for(m2=0;m2<10;m2++){
29       foerst[m2]=NULL;
30       sist[m2]=NULL;
31    }
32 }
```

1 Listans start kommer troligtvis att ändras efter sorteringen och därför är parametern `**start` en adress till adressen för listan.

`k` anger vilken för position i personnumret efter vilken sortering ska ske i denna omgång.

6–12 Så länge listan inte är slut bestämmer vi siffran på plats i nästa personnummer, lagras i `siffra`.

8 Personnumret köas i en av de 10 köerna.

9–11 Innan vi går vidare till nästa personnummer plockar vi bort det aktuella personnumret från listan och heapen.

13–27 När vi nu ska länka samma de 10 köerna ill en länkad lista måste vi ta hänsyn till att en kö kan vara tom! Detta gör denna rutin lite mer omständlig än vad man först tänker sig.

Till exempel får listans start adressen som pekas ut av det första elementet i den första kön, som inte är tom!

28-31 När nu alla köer nu är borta så NULL-ställer vi köpekarna, som är globala, inför nästa varv.

12.11 Merge

Då två *arrayer, filer* eller *länkade listor* innehåller var sina delar av ett datamaterial, båda sorterade efter samma nyckel, kan vi enkelt skapa en enda sorterad datamängd genom *samsortering* (merge).

Denna teknik används ofta då en stor datafil ska sorteras. Vi läser in en lagom stor del av filen. Efter sortering läses data till en fil. Denna procedur upprepas tills hela filen är uppstyckad i sorterade delfiler.

I sista steget utför vi så samsorteringen. Vårt studieobjekt blir det nu så bekanta elevregistret.

- Öppna filen som ska sorteras och ta reda på hur lång den är. Räkna ut hur många små filer som behöver skapas. Filens storlek begränsas till 64 Kb och därmed kan man också ta reda på hur många poster som får plats i varje delfil.
- Fyll en array med poster och sortera den. Skriv ut resultatet till en fil, vars namn enkelt kan räknas fram.
- Fortsätt tills alla poster är sorterade och placerade i ”småfiler”.
- Samsortera filerna till den sorterade slutprodukten.

```
1    oppna(&infil,&nfil,&npost,&antpost);
2    v=(persontyp *)calloc(sizeof(persontyp),npost);
3    for(k=1;k<=nfil;k++){
4       if(k<nfil){
5          laesin(infil,npost,v);
6          u=npost;
7          tot+=u;
8       }
9       else{
10         u=int(antpost-tot);
11         laesin(infil,u,v);
12      }
13      qsort(v,u,sizeof(persontyp),ordning);
14      skrivut(v,u,k);
15   }
16   fclose(infil);
17   samsortera("FIL",1,13,"DEL1");
18   samsortera("FIL",14,nfil,"DEL2");
19   samsortera("DEL",1,2,"ELEVREG.DAT");
20 }
```

1 Funktionen `oppna` returnerar

- `nfil`, som anger hur många småfiler som kommer att behövas.
- `npost`, som anger hur många poster som ryms i varje liten fil. Utom för den sista filen som kan innehålla färre filer.
- `antpost`, som anger det totala antalet poster som ska sorteras.

2 I vektorn `V` ska sorteringen ske. Dess storlek bestäms av värdena på parametrarna ovan.

3–15 Ett varv i loopen för varje liten fil, som ska skapas. För alla varv utom det sista exekveras den övre delen av if-satsen.

13 Med hjälp av den inbyggda sorteringsfunktionen `qsort` sorteras vektorn `v`, innehållande `u` poster, med storleken `sizeof(persontyp)`. Sorteringen sker med hjälp av funktionen `ordning`.

14 Den sorterade arrayen `v` skrivs ut till en fil vars filnamn innehåller ordningsnumret `k`.

17–19 Nu när alla poster är insorterade i småfiler kan *samsorteringen* starta. Ett problem dyker dock upp. Registret `elevreg.dat` kräver hela 26 småfiler. Så många filer kan man inte ha öppna samtidigt i den miljö, som programmet utvecklats i.

Därför måste samsorteringen ske i två steg. I första steget sorteras filerna `FIL1, FIL2 ... FIL13` samman till en fil `DEL1`. På samma sätt sammanställs `FIL14, FIL15 ... FIL26` till `DEL2`. Filer `DEL1` och `DEL2` samsorteras till `ELEVREG.DAT`.

```
1 int ordning(const void *a,const void *b) {
2   return strcmp(((persontyp *)a)->persnr,((persontyp *)b)->persnr);
3 }
```

I detta program använder vi den inbyggda *QuickSort – qsort* för att på det sättet göra framställningen mer komplett.

För att kunna använda `qsort`, måste man definiera en funktion som avgör ordningen mellan två nycklar. Två stycken `void`-pekare pekar ut den plats där posterna finns. Genom typomvandling ges pekarna rätt typ och därmed kan jämförelsen göras.

```
1  void samsortera(char fnamn[],int fran,int till,char unamn[]){
2    int k,klar=0,slut[30],minfil,antal=0;
3    FILE *infil[30],*utfil;
4    char namn[9],enamn[3],minst[12];
5    persontyp p[30];
6
7    utfil=fopen(unamn,"wb");
8    for(k=fran;k<=till;k++){
9      strcpy(namn,fnamn);
10     itoa(k,enamn,10);
11     strcat(namn,enamn);
12     infil[k]=fopen(namn,"rb");
13     fread(&p[k],sizeof(persontyp),1,infil[k]);
14     slut[k]=0;
15   }
16   while (!klar){
17     strcpy(minst,"99999999999");
18     for(k=fran;k<=till;k++){
19       if(!slut[k] && strcmp(p[k].persnr,minst)<0){
20         strcpy(minst,p[k].persnr);
21         minfil=k;
22       }
23     }
24     fwrite(&p[minfil],sizeof(persontyp),1,utfil);
25     fread(&p[minfil],sizeof(persontyp),1,infil[minfil]);
26     if(feof(infil[minfil])){
27       slut[minfil]=1;
28       antal++;
29       fclose(infil[minfil]);
30     }
31     if(antal==till-fran+1) klar=1;
32   }
```

```
33    fclose(utfil);
34    for(k=fran;k<=till;k++){
35       strcpy(namn,fnamn);
36       itoa(k,enamn,10);
37       strcat(namn,enamn);
38       remove(namn);
39    }
40 }
```

1–5 Här deklareras en array `*infil[30]`, som kan hålla reda på hela 30 filer.

Arrayen `slut[30]` ska hålla reda på om en fil har tagit slut.

I `post[30]` lagras de senast inlästa posterna från de upp till 30 filerna.

Att det sedan bara går att ha 14 filer öppna samtidigt är en helt annan historia.

8–15 Filerna ges ett namn och öppnas. Den första posten från varje fil läses in och lagras i `p`. `slut` nollställs vilket betyder att filerna inte har lästs till slutet.

16–32 Så länge `klar` inte är 1 finns det poster kvar att sortera in.

17–23 Det lägsta personnumret leta upp. När loopen är klar vet vi att det minsta personnumret finns i fil `minfil`

24–25 Denna post skrivs till utfilen och en ny post läses in

26–31 Om filen tog slut måste detta noteras i `slut` och `antal` växer för att till sist nå upp till `till-fran+1`, vilket betyder att alla filer är slut.

34–40 Funktionen avslutas med att filerna stängs och tas bort med den inbyggda funktionen `remove`. Detta betyder att inga tillfälliga filer finns kvar när programmet har kört färdigt.

UPPGIFT 12.4

Ändra funktionen för heapsort så att den i stället sorterar i *fallande* ordning. Testa ditt program med hjälp av de befintliga filerna `T?.DAT`.

UPPGIFT 12.5

Sortera personregister. På filen `elevreg.dat` finns som bekant uppgifter om 50 000 elever, *personnummer*, *förnamn* och *efternamn*.

Skriv ett program, som skapar en textfil med namnen, på registrets personer, sorterade i bokstavsordning. Namnen skrivs i ordningen *efternamn*, *förnamn* med ett mellanslag. Om efternamnen är lika sker sorteringen efter förnamnet.

Sorteringen ska utföras med hjälp av *shell-*, *heap-* eller *quicksort*, som måste modifieras med utgångspunkt från algoritmerna ovan.

UPPGIFT 12.6
Vilken Quicksort år bäst?. I avsnittet om *Quicksort* presenterades tre olika varianter, *den grundläggande*, *den rekursionsfria* och *den inbyggda*. Klassificera dessa genom att med hjälp av *Mathematica* plotta deras komplexitetsfunktioner.
Använd `t.dat` med $100\,000, 200\,000, \ldots, 1\,000\,000$ tal.

12.11.1 Lite till om sortering

Även om *mergesort* har samma komplexitet som *heapsort*, behöver mergesort $O(n)$ extra minne till skillnad från heapsort, som endast behöver $O(1)$ extra minne. Heapsort är i praktiken snabbare än mergesort. *quicksort* å andra sidan, betraktas av många som den i praktiken snabbaste sorteringsmetoden. Alla dessa metoder med samma komplexitet, $O(n \log n)$, men där quicksort har en mycket mindre koefficient, vide en bra implementation.

Mergesort är ofta bästa valet för att sortera länkade listor, då det är relativt enkelt att implementera mergesort, till och med att behöva utnyttja extra minne. Den relativt höga accesstiden för en länkad lista gör quicksort till ett dåligt val och heapsort till ett omöjligt. I det värsta fallet gör mergesort omkring 39% färre jämförelser än quicksort i ett normalfall. Mergesort gör alltid färre jämförelser än quicksort utom i extremfallet då alla objekt är lika, då samtidigt quicksort har sitt bästa fall.

Den inbyggda sorteringsrutinen i Java (`Arrays.sort()`) använder mergesort eller en modifierad quicksort beroende på datatyp. I Perl används idag mergesort mot tidigare quicksort.

Bogosort

Algoritm

Algorithm 12.11.1: BOGOSORT(deck)

while not inOrder(deck)
{blanda(deck)

Detta är en så kallad *randomized algorithm* med komplexitetsfunktionen $O((n-1)n!)$, som bestäms av *vätevärdet*

CombSort

Är det här en seriös algoritm? (Rätt svar ger 3 poäng)

```
void combsort(int *arr, int size) {
    float shrink_factor = 1.247330950103979;
    int gap = size, swapped = 1, swap, i;
    while ((gap > 1) || swapped) {
        if (gap > 1)
           gap = gap / shrink_factor;
        swapped = 0;
        i = 0;
        while ((gap + i) < size) {
            if (arr[i] - arr[i + gap] > 0) {
                swap = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = swap;
                swapped = 1;
            }
            ++i;
        }
    }
}
```

PROBLEM 12.1
Hur bra är CombSort (3 poäng). Sortera in denna sorteringsalgoritm, efter effektivitet, i listan över algoritmer.

Är detta verkligen en sorteringsalgoritm?

Följande algoritm, kallad *Sleep sort*, kom på posten här om dagen. Insänd av Richard Tjernberg, tidigare student på utbildningen. Vad krävs för att kunna exekveras och hur fungerar den?

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int c, char **v){
  while (--c > 1 && !fork());
  sleep(c = atoi(v[c]));
  printf("%d\n", c);
  wait(0);
  return 0;
}
```

12.11.2 En gammal tenta

Uppgift 1 – Omvänt tal

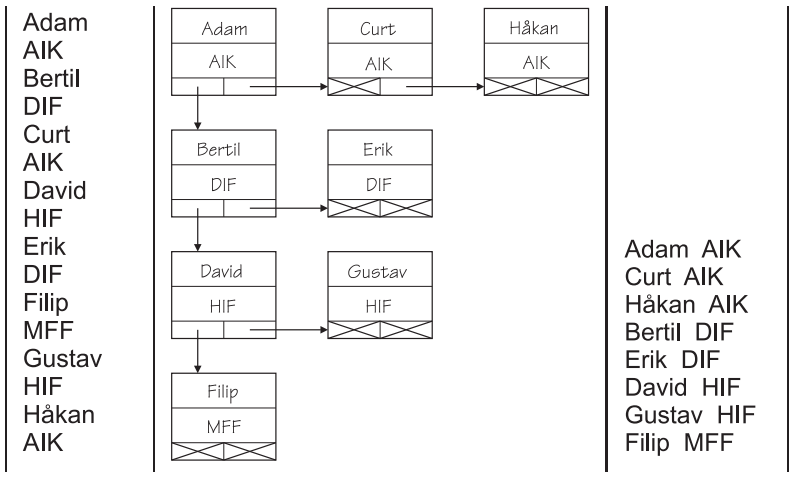
Låt s_0 vara summan av siffrorna hos ett positivt heltal t_0. Låt s_1 vara summan av siffrorna hos t_1, där $t_1 = s_0 + t_0$. Låt s_n vara summan av siffrorna hos t_n, där $t_n = s_{n-1} + t_{n-1}$. Vi undrar nu för vilka startvärden $1 \leq t_0 \leq 10000$, förr eller senare t_n får ett värde som är lika med t_0 läst bakifrån, från höger till vänster.

Ett exempel: Vi startar med $t_0 = 15$ och får i tur och ordning $t_1 = 21$, $t_2 = 24$, $t_3 = 30$, $t_4 = 33$, $t_5 = 39$, $t_6 = 51$. Målet är uppnått efter 6 steg, 51 är 15 läst bakifrån.

Skriv ett program som tar emot *undre* och övre *gräns* för vilka tal som ska undersökas (inklusive gränserna). Programmet ska skriva ut de tal som uppfyller villkoret och efter hur många steg eftersökt tal har uppnåtts. Ett körningsexempel:

```
Undre gräns ? 1000
Övre gräns  ? 1020
1011 efter 10 steg
1017 efter 388 steg
```

Uppgift 2 – Fotbollsklubbarna



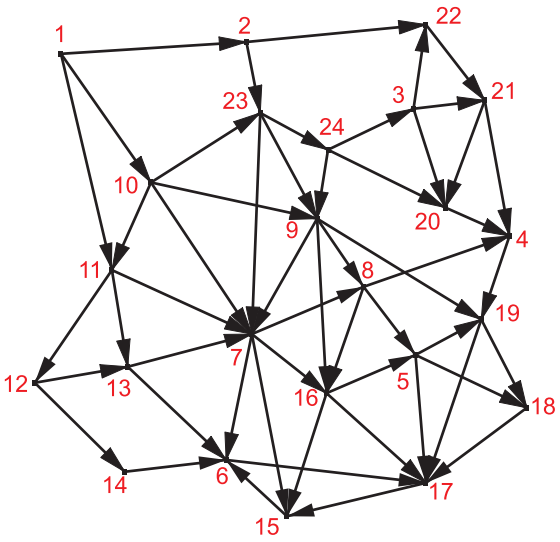
Figur 12.27:

Från filen `spelare.txt`, som innehåller *namn* och *klubbtillhörighet*, till exempel som till vänster i figur 12.27, vill man skapa den struktur som visas i mitten i figuren, för att till slut få den utskrift som visas till höger. En ny nod bildas i den vänstra kolumnen då en ny klubb påträffas. Spelare som tillhör redan registrerad klubb adderas i en länkad lista åt höger

Till ditt förfogande har du programkod som läser in data från filen och anropar `addspelare`, en funktion som du ska skriva. Programmet avslutas med att anropa `skrivut` som ska generera önskad utskrift.

Förutom funktionen `addspelare` får du inte ändra något i den givna koden.

Uppgift 3 – Den enkelriktade staden



Figur 12.28:

Kartan i figur 12.28 visar en stad med idel *enkelriktade* gator. Gatukontoret vill nu ha reda på till hur många vägkorsningar (noder) man kan ta sig från en given vägkorsning utan att bryta mot trafikreglerna. Pilarna visar i vilken riktning man kan åka.

Skriv ett program som frågar efter en vägkorsning, ett heltal mellan 1 och n, där $n \leq 50$ och bestämmer hur många korsningar man kan ta sig till utan att för den skull återvända till den korsning där färden började.

Filen `korsningar.txt` inleds med en rad som anger antalet korsningar n. På nästa rad finns uppgift om antalet `g` gator. Därefter följer `g` rader med två tal på varje. Som, i tur och ordning, anger *från* vilken korsning gatan går *till* vilken korsning. Ett körningsexempel:

```
Var startar färden ? 13
Man kan nå andra 10 korsningar
```

Uppgift 4 – Antal besök på WWW.PRIME.SE

Hemsidan `www.prime.se` besöks varje dag mellan n_1 och n_2 gånger (inklusive gränser). Förutom en räknare som håller reda på antalet besök per dag, finns en räknare r, som håller reda på det totala antalet besök under en längre tid. I samband med att r nollställdes härom dagen, ställde man sig frågan, i hur många dagar i rad det var möjligt att r skulle kunna uppvisa ett primtal, klockan 00 : 00 vid dagens slut.

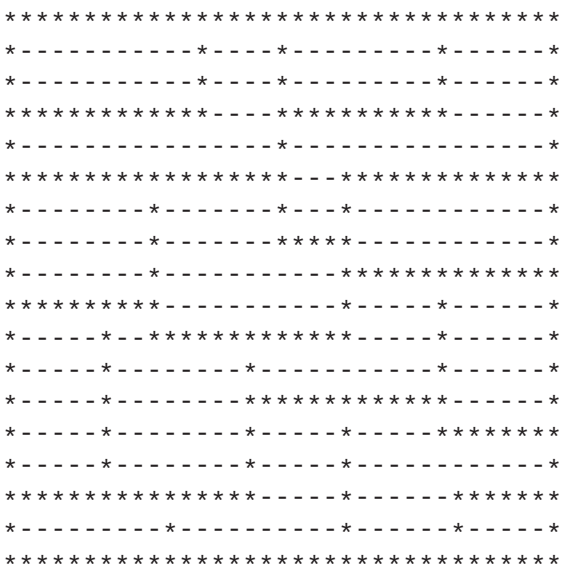
Om $n_1 = 2$ och $n_2 = 5$ får vi följande resultat

Dag nr	1	2	3	4	5	6	7	8
Antal besök denna dag	2	3	2	4	2	4	2	4
Antal besök totalt (r)	2	5	7	11	13	17	19	23

r kan anta primtal 8 dagar i rad. Skriv ett program som frågar efter $1 \leq n_1 < n_2 \leq 30$ och som bestämmer hur många dagar i rad r kan landa på ett primtal.

```
n1 ? 10
n2 ? 20
"Primtalstotalen" kan upprätthållas i 84 dagar
```

Uppgift 5 – Slottet



Figur 12.29:

Figur 12.29 ovan förställer ritningen över ett slott, med alla dess rum. *Asterisker*, (`*`), utgör väggar och *bindestreck*, (`-`), utgör golv. Man har nu tänkt sig att slå ut en vägg, ersätta **en** asterisk med ett bindestreck, och genom detta bilda det största möjliga rummet i slottet.

Skriv ett program som från filen `slott.txt` läser in slottets ritning, på filen återgiven som i figuren ovan. Filen inleds med ett tal som anger hur många rader, $3 < \mathtt{h} < 100$, den innehåller. På nästa rad anges antal tecken, $3 < \mathtt{b} < 100$, samtliga rader består av. Därefter följer `h` rader. Första och sista raden innehåller enbart asterisker. Raderna däremellan inleds och avslutas alltid med en asterisk. Programmet ska bestämma antalet areaenheter hos det största rum som kan skapas. En areaenhet är en position på filen. Ett körningsexempel:

```
Det största rummet har arean 77 a.e.
```

Uppgift 6 – Eftersökta ord

På filen `ord.txt` finns inte mindre än 119796 ord. Inget längre än 30 bokstäver, `a...ö`.

a) Det finns, på filen, en del ord, sådana att då man avlägsnar *första* bokstaven, uppstår ett nytt ord som också finns på filen. Till exempel ordet `katt` ger `att`.

b) Det finns även ord som, då *sista* bokstaven avlägsnas, ger ett nytt ord som också finns på filen. Till exempel ordet `oxel` ger `oxe`.

c) Det finns ett mindre antal ord som tillhör både kategori `a`) och `b`). Till exempel ordet `blogga` ger `logga` och `blogg`.

Skriv ett program som tar reda på hur många ord av kategori `c`) det finns på filen `ord.txt`. Filen inleds med ett tal $n, n < 120000$ som anger antalet ord filen innehåller. På följande n rader finns ett ord med ≤ 30 tecken.

Mitt program klarar jobbet på cirka 0.3 sekunder. För att få poäng på uppgiften ska du skriva ett program som klarar jobbet på mindre 20 sekunder!

Filen `ord2.txt`, en betydligt mindre fil, kan du använda för att testa att logiken i ditt program fungerar. Ett körningsexempel med filen `ord2.txt`

```
Det finns 8 eftersökta ord
```