
Programmering 1

Del 3: Sökning, sortering och samlingar

Martin Larsson

19 juni 2019

Innehåll

1	Sortering	3
1.1	<i>Insertion sort</i>	3
1.2	<i>Bubble sort</i>	7
1.3	Tidskomplexitet	10
1.3.1	Exempel: Sortera Facebook-användare	11
1.4	Att göra	12
1.4.1	Övningar	12
2	Sökning	15
2.1	Binär sökning	18
2.2	Effektivitet	22
2.3	Att göra	22
2.3.1	Övningar	22
2.3.2	Fördjupning: Sortering/sökning med strängar	23
3	Samlingar	24
3.1	List	24
3.1.1	List-metoder	25
3.2	Stack	27
3.3	Queue	30
3.4	Andra samlingar	31
3.5	Att göra	31
3.5.1	Övningar	31

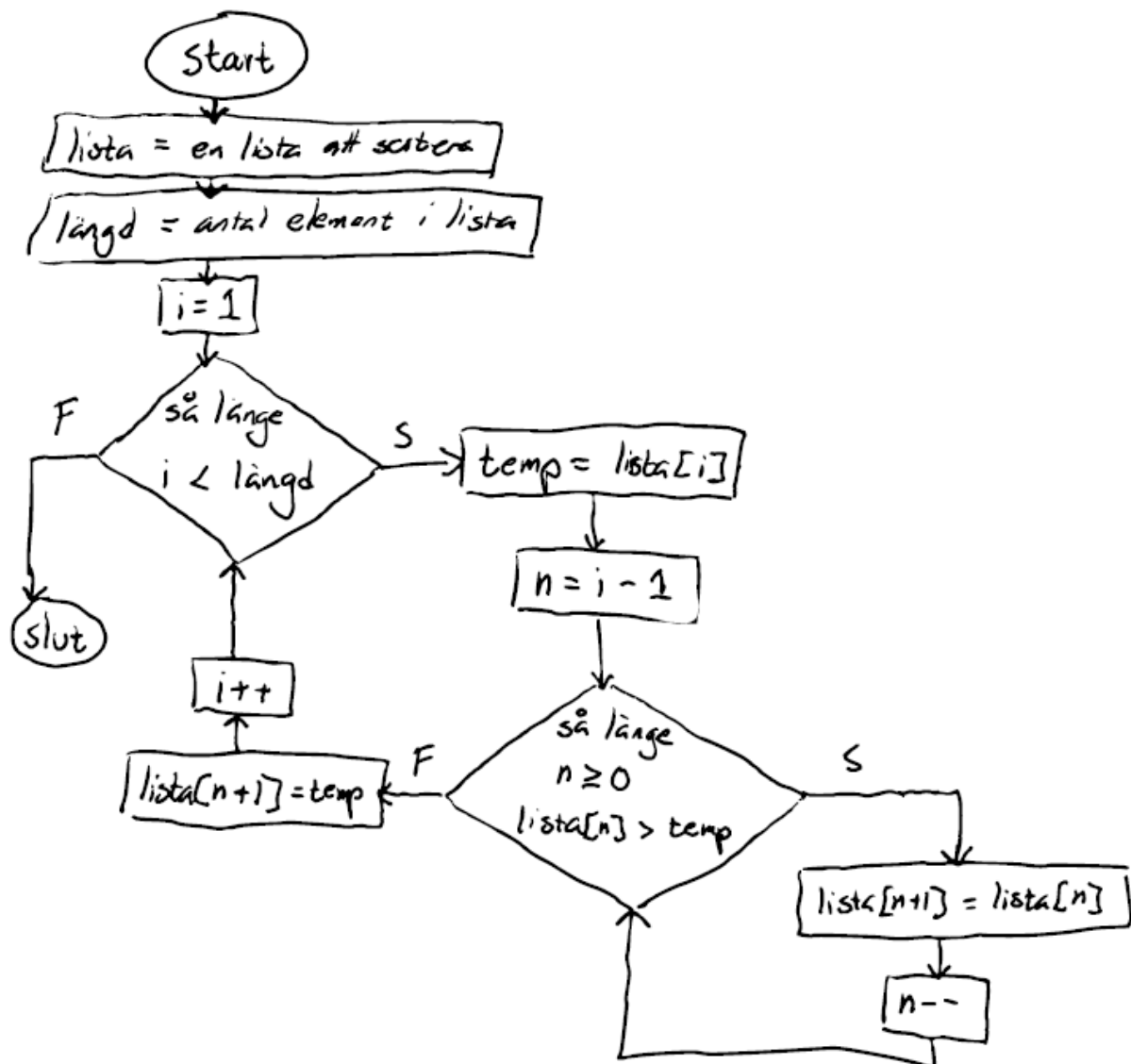
1 Sortering

Väldigt ofta måste data sorteras för att kunna vara hanterbart. Det kan röra sig om att sortera t.ex. tal, datum eller ord i storleksordning, stigande/fallande ordning eller bokstavsordning. Syftena med att programmera sortering kan vara många men inte sällan handlar det om att enklare kunna söka (nästa kapitel) efter information, oavsett om det rör sig om en människa eller en dator.

Eftersom att sortering är ett så pass vanligt förekommande problem inom programmeringen finns massor av algoritmer utvecklade för att lösa det. De är olika effektiva i olika situationer (storlek och typ av data) och det är svårt, åtminstone inom ramen för denna kurs att lära sig allihop. Däremot bör man lära sig *minst* en algoritm för att kunna förstå principen bakom hur en dator sorterar. För att göra det hela begripligt kommer vi till en början att nöja oss med att sortera tal som är organiserade i arrayer.

1.1 *Insertion sort*

En av alla sorteringsalgoritmer som finns heter *insertion sort* (eller infogande sortering på svenska). Vi studerar just denna algoritm eftersom att den har en bra avvägning mellan att vara enkel att förstå och effektiv väldigt många fall. Strukturdiagrammet för algoritmen ser ut såhär:



Figur 1.1: Strukturdiagram för *insertion sort*.

Ovanstående diagram ger oss pseudokoden för *insertion sort*:

```

1 start
2   lista = en lista att sortera
3   längd = antal element att sortera i lista
4   i = 1
5   så länge i < längd
6     temp = lista[i]
7     n = i - 1

```

```

8      så länge n >= 0 & lista[n] > temp
9          lista[n + 1] = lista[n]
10         n--
11     lista[n + 1] = temp
12     i++
13 slut

```

Kortfattat kan man säga att algoritmens teknik för att sortera går ut på att plocka ut ett tal i taget och stoppa in det (infoga) på rätt plats. På sidan 191 i boken finns en enkel steg för steg-beskrivning som gör det lite klarare. En ännu bättre förklaring finner du dock på visualgo.net/en/sorting om du sedan klickar på *INS* i menyn längst upp. Där kan du starta en animering som visar vad som sker i varje steg. Du kan även pausa och manuellt stega både framåt och bakåt.

Om vi istället tittar på källkod så kan *insertion sort* skrivas såhär i C#:

```

1  int[] lista = { 3, -5, 7, 38, 1, 0, 23, 15 };
2  for (int i = 1; i < lista.Length; i++)
3  {
4      int temp = lista[i];
5      int n = i - 1;
6      while (n >= 0 && lista[n] > temp)
7      {
8          lista[n + 1] = lista[n];
9          n--;
10     }
11     lista[n + 1] = temp;
12 }

```

Det är ofta en bra idé att göra en utskrift av listan både före och efter sortering så att man enkelt kan kontrollera sin algoritm. Detta görs smidigast med en utskrifts-metod.

```

1  public static void Main()
2  {
3      int[] lista = { 3, -5, 7, 38, 1, 0, 23, 15 };
4
5      SkrivUtLista(lista);
6
7      for (int i = 1; i < lista.Length; i++)
8      {
9          int temp = lista[i];
10         int n = i - 1;
11         while (n >= 0 && lista[n] > temp)
12         {

```

```

13         lista[n + 1] = lista[n];
14         n--;
15     }
16     lista[n + 1] = temp;
17 }
18
19 SkrivUtLista(lista);
20 }
21
22 static void SkrivUtLista(int[] lista)
23 {
24     for (int i = 0; i < lista.Length; i++)
25     {
26         Console.Write(lista[i] + " ");
27     }
28     Console.WriteLine();
29 }
    
```

Självklart är det också en god idé att placera sin sorteringsalgoritm i en egen metod. Det kan ju mycket väl vara så att den behöver användas flera gånger eller på olika arrayer. Kom ihåg att array som parameter hanteras som referens och du behöver därmed ej returnera något - förändringarna i metoden sker även i originalarrayen.

```

1 public static void Main()
2 {
3     int[] lista = { 3, -5, 7, 38, 1, 0, 23, 15 };
4
5     SkrivUtLista(lista);
6     SorteraInfogade(lista);
7     SkrivUtLista(lista);
8 }
9
10 static void SorteraInfogade(int[] lista)
11 {
12     for (int i = 1; i < lista.Length; i++)
13     {
14         int temp = lista[i];
15         int n = i - 1;
16         while (n >= 0 && lista[n] > temp)
17         {
18             lista[n + 1] = lista[n];
19             n--;
20         }
    
```

```

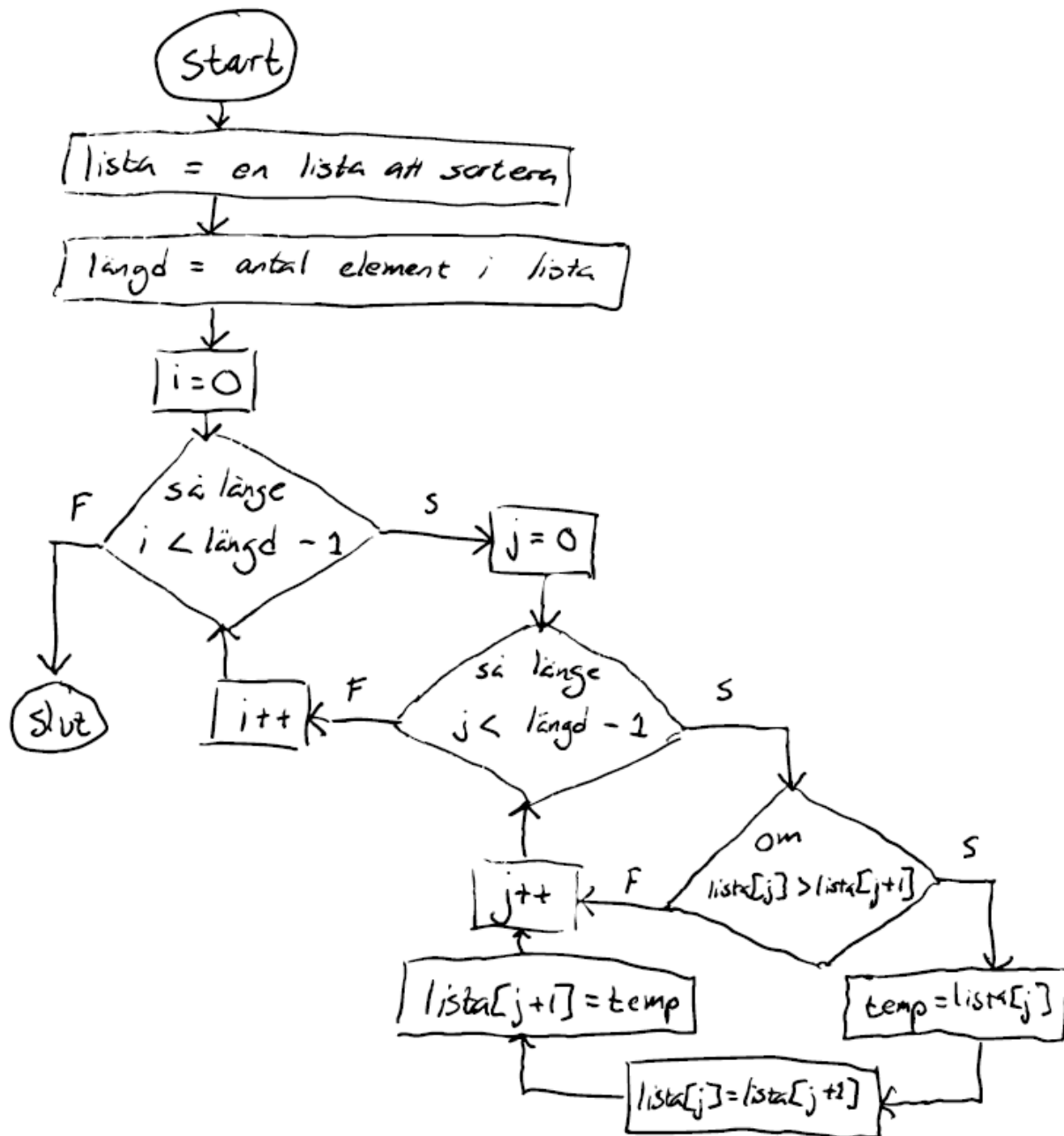
21         lista[n + 1] = temp;
22     }
23 }
24
25 static void SkrivUtLista(int[] lista)
26 {
27     for (int i = 0; i < lista.Length; i++)
28     {
29         Console.Write(lista[i] + " ");
30     }
31     Console.WriteLine();
32 }

```

1.2 Bubble sort

Den sorteringsalgoritm som många elever/studenter ofta får lära sig allra först är *bubble sort* (eller bubbelsortering). Den må vara relativt enkel att förstå men den är i väldigt många fall ineffektiv och därav sällan användbar. Det är dock inte så dumt att bekanta sig med *bubble sort* för att i nästa del kunna resonera om algoritmers effektivitet.

Den generella principen bakom *bubble sort* är att jämföra talen i de två första elementen, byta plats om de sitter i fel ordning, jämföra nästa två tal, byta plats om de sitter i fel ordning och sedan upprepa denna procedur tills hela listan har gått igenom en gång (ett tal kommer då finnas på rätt plats). Därefter behöver upprepningar av hela proceduren ske igen tills dess att hela listan är sorterad, dvs till vi har fått *alla* tal på rätt plats (du kan även se en animering för *bubble sort* på visualgo.net/en/sorting, välj BUB i menyn längst upp). Vi illustrerar i vanlig ordning med ett strukturdiagram:



Figur 1.2: Strukturdiagram för *bubble sort*.

Pseudeokoden för detta diagram ser ut såhär:

```

1 start
2   lista = en lista att sortera
3   längd = antal element i lista
  
```



```

4     i = 0
5     så länge i < längd - 1
6         j = 0
7         så länge j < längd - 1
8             om lista[j] > lista[j + 1]
9                 temp = lista[j]
10                lista[j] = lista[j + 1]
11                lista[j + 1] = temp
12            j++
13        i++
14 slut

```

Om vi implementerar algoritmen i C# ser den något enklare ut eftersom att vi då kan dra nytta av **for**-satsen vilket gör att koden för våra loopar kan skrivas något enklare:

```

1  int[] lista = { 3, -5, 7, 38, 1, 0, 23, 15 }
2  for (int i = 0; i < lista.Length - 1; i++)
3  {
4      for (int j = 0; j < lista.Length - 1; j++)
5      {
6          if (lista[j] > lista[j + 1])
7          {
8              double temp = lista[j];
9              lista[j] = lista[j + 1];
10             lista[j + 1] = temp;
11         }
12     }
13 }

```

Givetvis är det god idé även för denna algoritm (och för de allra flesta) att göra utskrifter före och efter samt plocka ut algoritmen till en egen metod, men det kan du nu så vi tar inte med det steget en gång till!

Den algoritmbeskrivning av *bubble sort* som finns ovan kan optimeras på två olika sätt. Dels vet vi att efter ett varv i den yttre loopen kommer vi inte behöva gå igen *hela* lista i den inre loopen. Dessutom kan vi vara säkra på att om *ingen* byte alls sker under ett varv så måste hela listan redan vara sorterad och vi kan avbryta. Mer om detta i en kommande övning.

1.3 Tidskomplexitet

En algoritms tidsåtgång (eller egentligen antalet operationer datorn behöver göra) som funktion av indata's storlek kan växa på olika sätt för olika algoritmer. Vid små datamängder ser vi ingen skillnad men för stora indata kan en snabbt växande tidsåtgång bli ett stort problem.

Ju större datamängder vi tittar på desto mer dominant blir bidraget till resultatet från *en* viss term i ett uttryck. Om vi t.ex. tittar på

$$f(x) = x^2 + 3x$$

och undersöker $x = 100000$ så får vi

$$f(100000) = 100000^2 + 3 * 100000 = 10^{10} + 3 * 10^5 = 1.00003 * 10^{10}$$

Bidraget till funktionens värde då $x = 100000$ från termen x^2 är alltså mycket större än det från termen $3x$ och detta växer givetvis med större värden på x .

Om vi ska resonera kring funktionsvärdet är det alltså den starkast bidragande termen som är mest intressant. Till och med i sådan utsträckning att vi nöjer oss med att *endast* titta på den termen. Dessutom förenklar vi bort eventuella koefficienter. Vi närmar oss då (något omatematiskt) det matematiska begreppet *ordo*, O (eng: *Big O-notation*). När vi pratar om indata så håller vi oss till heltal (vi kan ju t.ex. inte ha en halv extra position i en array) och därför brukar man använda n som variabel snarare än x . Om den starkast bidragande termen är kvadratisk (som i exemplet ovan) skriver man då

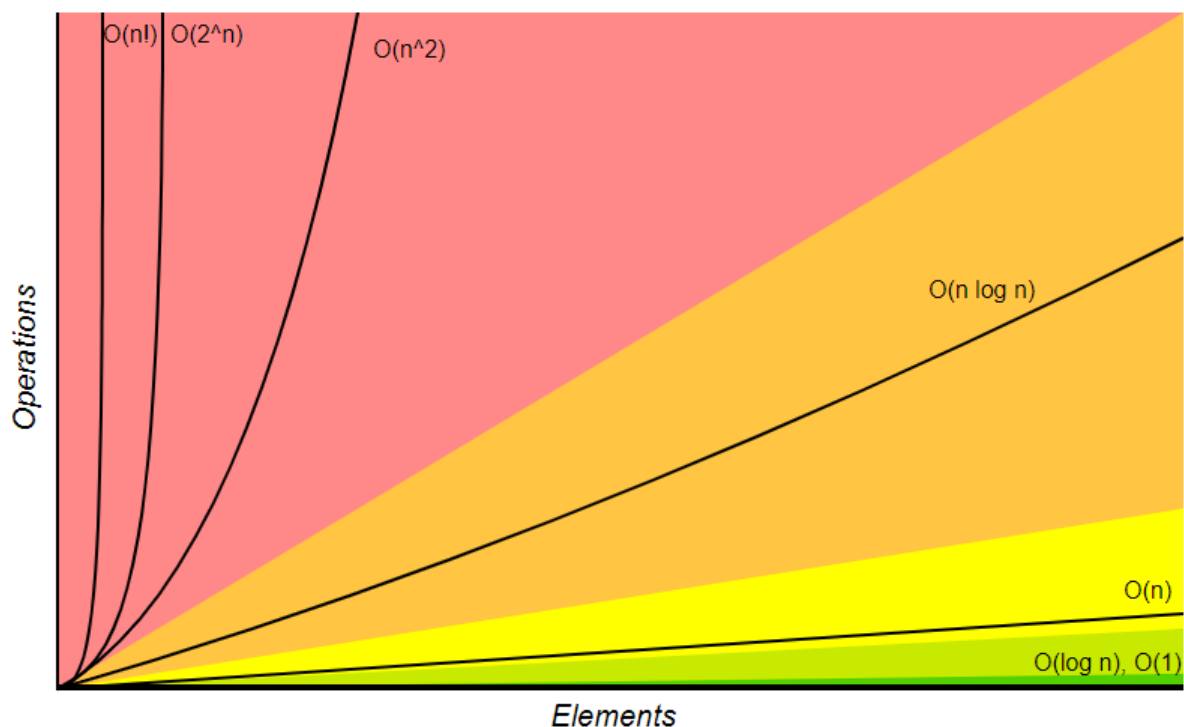
$$O(n^2)$$

och detta utläses *ordo n-kvadrat* och inte som *O av n-kvadrat* som man kanske är van vid då det liknar skrivsättet $f(x)$. Man säger alltså att funktionen $f(n) = n^2 + 3n$ har ordo n^2 . Några fler exempel:

Funktion	Ordo
$3n^2$	$O(n^2)$
$3n^2 - 7$	$O(n^2)$
$5n$	$O(n)$
$13n^3 + 2n^2$	$O(n^3)$
$2^n + n^2$	$O(2^n)$

Funktion	Ordo
5 (konst.)	$O(1)$

Om vi ritar flera olika typer av kurvor i ett och samma diagram och fortfarande minns att ordo beskriver tidsåtgång (tidskomplexitet) så kan vi ganska lätt inse att algoritmer som har olika ordo kan ha väldigt stor skillnad i effektivitet när vi pratar om tillräckligt stora värden på indata.



Figur 1.3: Graf över olika ordo.

En bra sammanfattning över ordo för olika algoritmer finns på bigocheatsheet.com.

1.3.1 Exempel: Sortera Facebook-användare

Enligt svenska Wikipedia hade Facebook två miljarder ($2.9 \cdot 10^9$) aktiva användare 2017. Om det skulle vara så att vi behöver sortera alla användares ID skulle det bli rejäla skillnader i tidsåtgång beroende på vilken sorteringsalgoritm vi använder. Om vi först provar med *bubble sort* som har $O(n^2)$. Antalet operationer datorn behöver göra blir:

$$O(n^2) \Rightarrow 2 * 10^9^2 = 4 * 10^{18}$$

En dator som inte är allt för trött idag kan utföra ungefär 150000 MIPS (IPS = instruktioner per sekund), dvs $150000 * 10^6 = 1.5 * 10^{11}$ IPS. Tiden för sortering med en sådan dator blir då:

$$\frac{4 * 10^{18}}{1.5 * 10^{11}} = 2.67 * 10^7 s = 7400h = 310dygn$$

Om vi istället skulle använda t.ex. *merge sort* (som har $O(n * \log(n))$) skulle beräkningen istället bli:

$$O(n * \log_2(n)) \Rightarrow 2 * 10^9 * \log_2(2 * 10^9) = 6.18 * 10^{10}$$

$$\frac{6.18 * 10^{10}}{1.5 * 10^{11}} = 0.4s$$

Från 310 dygn till 0,4 s - valet av algoritm kan alltså spela väldigt stor roll!

1.4 Att göra

- 📖 Läs mer om sortering i boken på s. 190-193.
- 🌐 Läs mer om sorteringsalgoritmer och effektivitet på csharpskolan.se/article/sorteringsalgoritmer.
- ✅ Genomför självtestet *Sortering* på Moodle: moodle.teed.se.
- ✍ Genomför övningarna 1-4 här nedan.

1.4.1 Övningar

1. Mätningar på *Bubble sort*:

- Utgå från den *Bubble sort*-kod som finns tidigare i kapitlet och utöka med/genomför följande.
- Lägg utskrift av arrayen i egen metod. Anropa metoden före och efter sortering för att kontrollera att sorteringsalgoritmen beter sig korrekt.
- Lägg *Bubble sort*-algoritmen i egen metod. Anropa metoden från huvudprogrammet. Testkör igen och kontrollera så att ingen funktionalitet har förlorats.
- Lägg till direktivet `using System.Diagnostics`; längst upp i källkoden.
- Skapa en ny variabel av typen `Stopwatch` med kodraden `Stopwatch timer = new Stopwatch();`

- Kör `timer.Start()` ; precis före sorteringsanropet och `timer.Stop()` ; precis efter.
 - Skriv ut tiden för sortering med hjälp av `timer.ElapsedMilliseconds`.
 - Ändra storleken på arrayen till 1000 och låt talen som slumpas fram vara inom spannet `int.MinValue` och `int.MaxValue`.
 - Kör programmet och notera tiden det tar att sortera samt storleken på arrayen. Upprepa samma körning tre gånger och beräkna medelvärdet av tiden. Fyll i storleken och medelvärdet i ett kalkylblad i *Excel*.
 - Upprepa proceduren (med tre körningar per storlek och sedan ta medelvärdet) för storlekarna: 2500, 5000, 7500, 10000, 20000, 30000, 40000 och 50000. Notera samtliga i ditt kalkylblad.
 - Använd *Excel* för att göra ett punktdiagram och anpassa en kurva för dina värden. Vilken typ av kurva ger bäst matchning (störst R^2 -värde)?
 - Försök att göra följande optimeringar av algoritmen för *Bubble sort*:
 - När ett varv har genomförts vet vi att det sista elementet är på rätt plats. Det behöver inte kontrolleras igen vid nästa varv. Med andra ord behöver vi inte gå igenom hela arrayen vid varje varv. Redigera din kod så att dessa onödiga kontroller ej behöver göras.
 - Om ett varv, vilket som helst, inte behöver byta plats på några element alls så måste hela arrayen vara sorterad. Vi kan då avbryta algoritmen i förtid och slippa köra resten av den. Redigera din kod så att den minns om ett byte har gjorts eller inte, om inte så avbryts algoritmen.
 - Testa din sortering (med hjälp av utskrift) igen för att försäkra dig om att den fortfarande fungerar som tänkt.
 - Gör om några av mätningarna av exekveringstid, märks någon skillnad?
 - Redigera din *Bubble sort*-kod så att den sorterar fallande istället för stigande. Testa med hjälp av utskrift så att den gör rätt.
2. Genomför övningen *Save Patients* som finns på ett separat blad (laddas ned på lärplattformen).
 3. Skriv ett program som slumpar fram 20 heltal i en array. Låt användaren bestämma om talen i arrayen ska sorteras i stigande eller fallande ordning. Implementera *insertion sort* och sortera utifrån det val användaren har gjort.

```

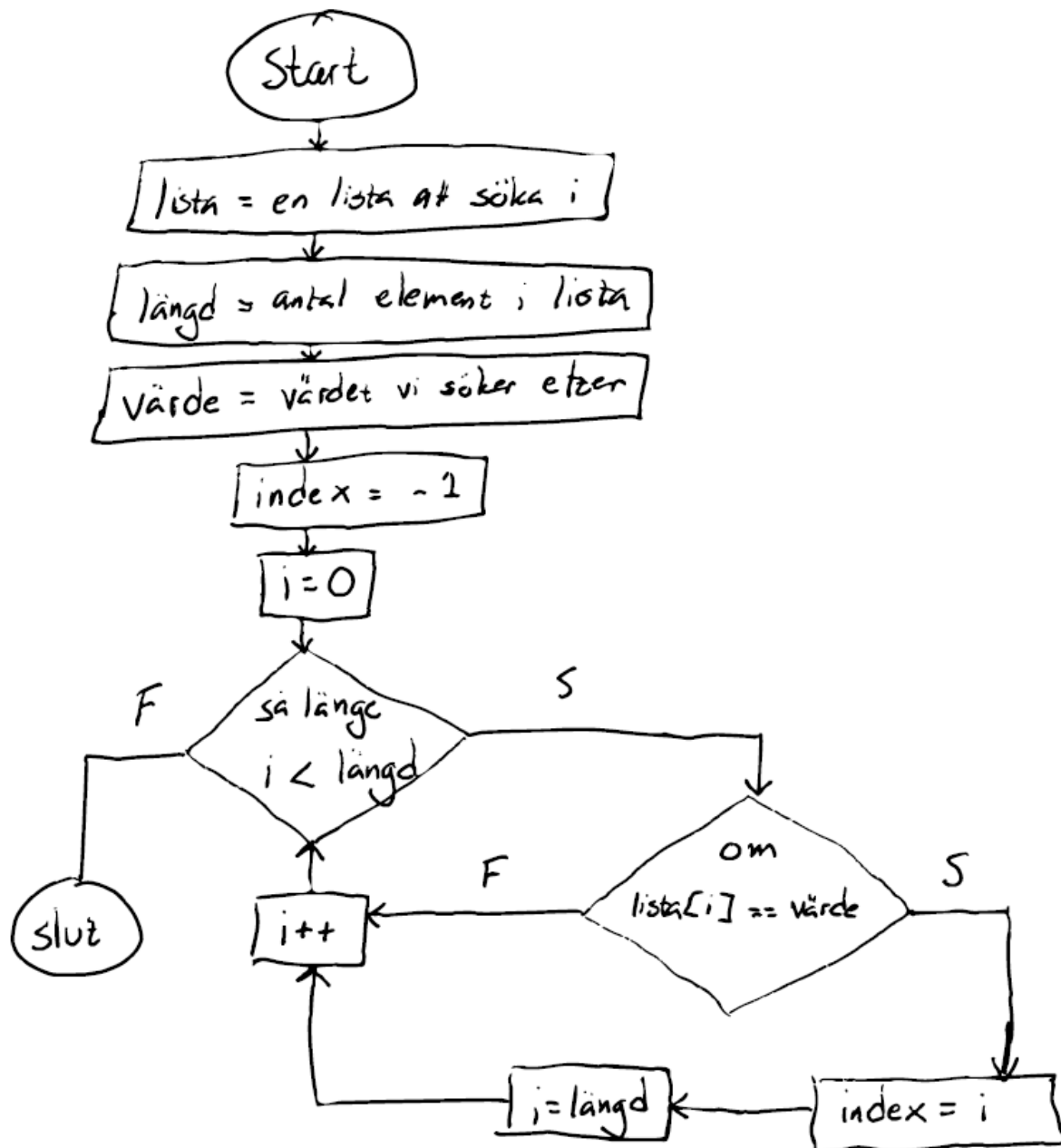
C:\WINDOWS\system32\cmd.exe
Vill du sortera stigande eller fallande? (s/f): f
100 99 91 81 79 78 74 70 60 48 38 29 26 17 17 15 8 6 5 2
Press any key to continue . . .
    
```

Figur 1.4: Sortering i fallande ordning med *insertion sort*.

4. Skriv ett program där 20 decimaltal slumpas till en array. Sortera första halvan av arrayen i stigande ordning och andra halvan i fallande ordning. Du bestämmer själv vilken sorteringsalgoritm du utgår ifrån.

2 Sökning

I förra kapitlet nämndes att en av de vanligaste anledningarna till att sortera något är att enklare kunna göra en sökning. Givetvis är det möjligt att söka i en datamängd som inte är sorterad - det är ju helt enkelt bara att titta på varje element, från början till slut tills dess att vi funnit det vi söker efter. En sådan enkel algoritm kan vi beskriva med följande strukturdiagram och pseudokod:



Figur 2.1: Strukturdiagram för sekventiell sökning.

```

1 start
2 lista = en lista att söka i
3 längd = antal element i lista
4 värde = värdet vi söker efter
5 index = -1
    
```



```

6     i = 0
7     så länge i < längd
8         om lista[i] == värde
9             index = i
10            i = längd
11        i++
12    slut
    
```

Gör vi en implementation i C# ser det ut såhär (att sätta `i = längd` innebär att vi vill avbryta, därav **break** i källkoden nedan):

```

1  int[] lista = { 3, -5, 7, 38, 1, 0, 23, 15 };
2  int varde = 23;
3  int index = -1;
4  for (int i = 0; i < lista.Length; i++)
5  {
6      if (lista[i] == varde)
7      {
8          index = i;
9          break;
10     }
11 }
    
```

Att vi sätter `index = -1` från början innebär att vi själva har definierat att `-1` betyder att vi inte hittat det sökta talet i arrayen. Om vi ska snygga till strukturen på vår algoritm lite så lägger vi den i en egen metod och låter den returnera det index som vi finner det sökta talet på, dvs vår `index`-variabel. Då ser i så fall ovanstående kod ut såhär:

```

1  static int SekventiellSokning(int[] lista, int varde)
2  {
3      int index = -1;
4      for (int i = 0; i < lista.Length; i++)
5      {
6          if (lista[i] == varde)
7          {
8              index = i;
9              break;
10         }
11     }
12     return index;
13 }
    
```

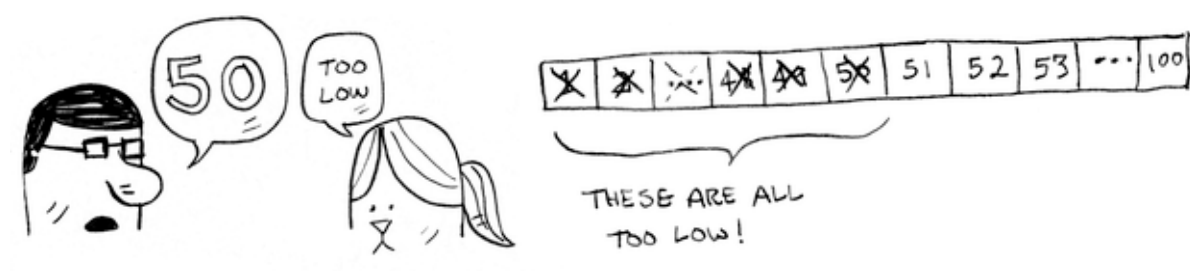
Denna algoritm kallar vi för **sekventiell sökning** (eller ibland enkel sökning) och kan alltid användas för att hitta vad som helst i vilken datamängd som helst. Däremot är den inte särskilt effektiv jämfört med nästa sökalgoritm - **binär sökning**!



Figur 2.2: Sekventiell sökning kan bli outhärdlig.

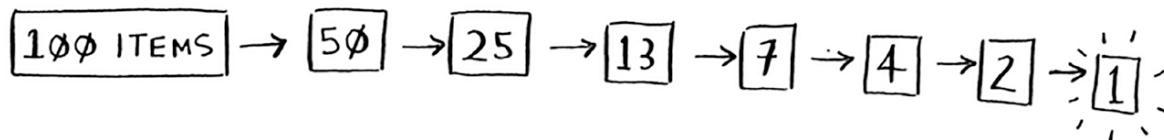
2.1 Binär sökning

Du kanske minns leken där en person tänker på ett tal (t.ex. mellan 1 och 100) och en annan skall gissa vilket tal man tänker på. Vid varje felgissning får man reda på om man gissat på ett för stort eller för litet tal. Givetvis kan man ha tur och gissa rätt med en gång om man tar ett tal på måfå men tricket för att i det generella fallet minimera antalet gissningar är att hela tiden gissa på talet i mitten. Eftersom att vi får reda på om vi gissat för stort eller för litet kan vi redan efter första gissningen utesluta hälften av alla möjliga tal.



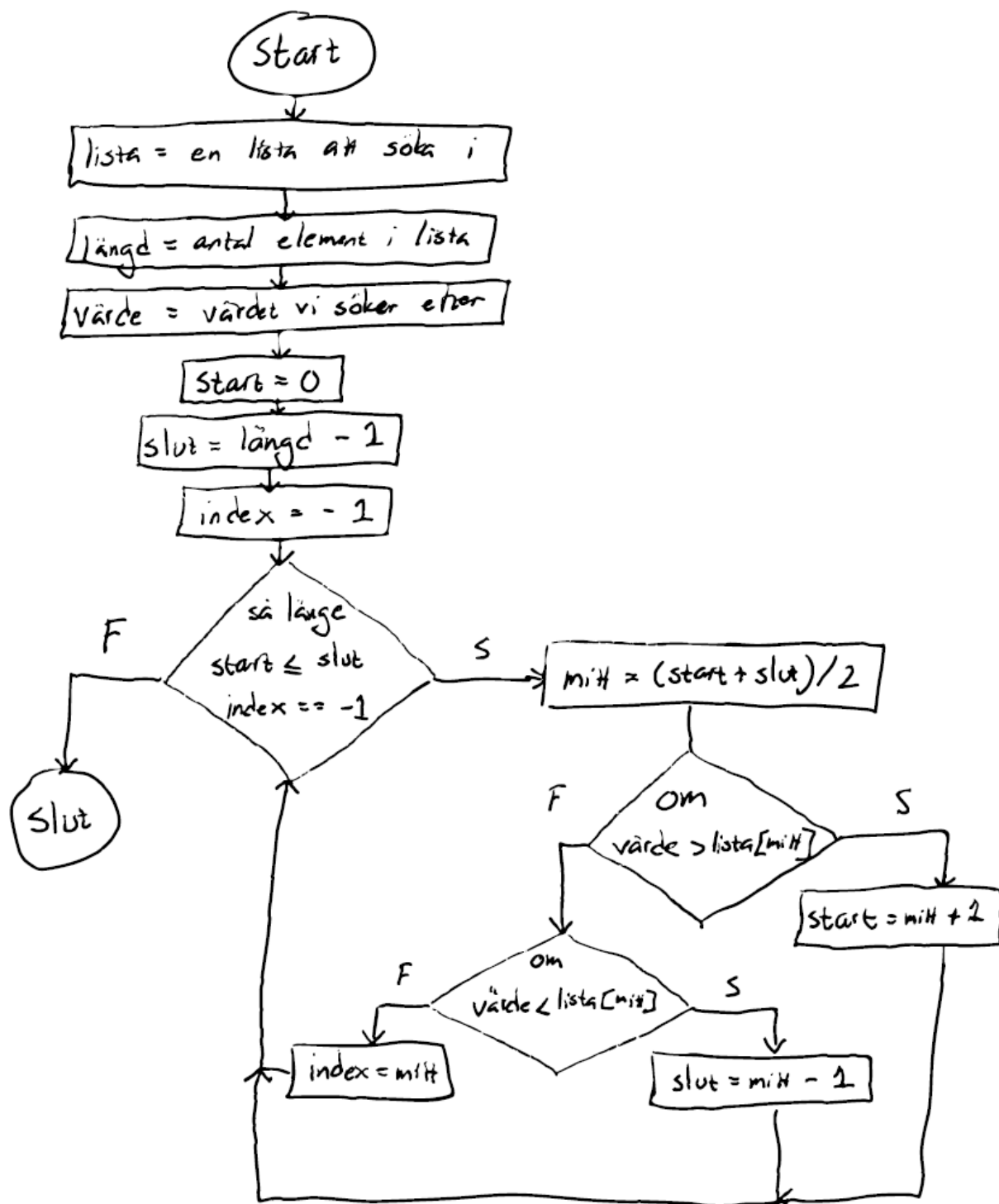
Figur 2.3: Binär sökning utesluter hälften av möjligheterna direkt.

Om man sedan upprepar denna procedur för den del av talen som är kvar (gissa på nya mitten) så har vi en konceptuell beskrivning för en algoritm som hittar det vi söker efter. Vi halverar sökmängden vid varje gissning (varje varv i vår algoritm):



Figur 2.4: Relativt få upprepningar med binär sökning behövs för att hitta rätt svar.

Som strukturdiagram och pseudokod ser algoritmen ut som följer.



Figur 2.5: Strukturdiagram för binär sökning.

1 start

```
2 lista = en lista att söka i
3 längd = antal element i lista
4 värde = värdet vi söker efter
5 start = 0
6 slut = längd - 1
7 index = -1
8 så länge start <= slut & index == -1
9     mitt = (start + slut) / 2
10     om värde > lista[mitt]
11         start = mitt + 1
12     annars om värde < lista[mitt]
13         slut = mitt - 1
14     annars
15         index = mitt
16 slut
```

Gör vi en implementation av denna i C# (som egen metod) får vi följande källkod:

```
1 static int BinarSokning(int[] lista, int varde)
2 {
3     int start = 0;
4     int slut = lista.Length - 1;
5     int index = -1;
6     while (start <= slut && index == -1)
7     {
8         int mitt = (start + slut) / 2;
9         if (varde > lista[mitt])
10         {
11             start = mitt + 1;
12         }
13         else if (varde < lista[mitt])
14         {
15             slut = mitt - 1;
16         }
17         else
18         {
19             index = mitt;
20         }
21     }
22     return index;
23 }
```

Var dock noga med att lägga på minnet att denna typ av sökning *kräver* en sorterad datamängd. Du kan

se en animering över hur de två olika sökalgoritmerna jobbar på cs.usfca.edu/~galles/visualization/Search.html.

2.2 Effektivitet

Givetvis skall man ha i åtanke att det tar en viss tid att sortera en datamängd innan man blint hoppar på binär sökning. Men om vi gör några enkla exempelberäkningar med hjälp av ordo för sekventiell och binär sökning inser vi att det lönar sig ganska snabbt med binär sökning om mer än sökning skall genomföras i samma datamängd (den behöver ju bara sorteras *en* gång).

Sekventiell sökning har $O(n)$ och binär sökningar har $O(\log_2(n))$. Vi tittar återigen på *worst case* (och avrundar uppåt) får nedanstående, enorma skillnad.

Antal element	Antal operationer: \ Sekventiell sökning	Antal operationer: \ Binär sökning
100	$1 * 10^2$	7
10 000	$1 * 10^4$	14
1 000 000 000	$1 * 10^9$	30

2.3 Att göra

- 📖 Läs mer om sökning i boken på s. 195-198.
- 🌐 Läs mer om sökalgoritmer och effektivitet på csharpskolan.se/article/sokalgoritmer.
- ✅ Genomför självtestet *Sökning* på Moodle: moodle.teed.se.
- ✍ Genomför övningarna 5-9 här nedan.
- ✍ Genomför övningarna 10-11 som lite fördjupning för att prova på sökning/sortering med strängar istället för siffror.

2.3.1 Övningar

OBS! Om du har en bok med upplaga från 2012 så är de fel som nämns i uppgift 5 och 6 tillrättade. Du kan därmed hoppa över dessa. Om din bok är äldre än 2012 är det bara köra på som vanligt.

5. Studera källkoden till sekventiell sökning på s. 195 i boken. Ett fel har letat sig in där. Kan du hitta det?
6. Studera källkoden till binär sökning på s. 197 i boken. Tre fel (utöver den märkliga indentering av if-satserna) har letat sig in där. Kan du hitta alla tre?

7. Hur många varv behöver en binär sökalgoritm gå för att garanterat hitta ett tal i en sorterad mängd med 150 element?
8. Implementera en sekventiell sökning genom att endast titta på strukturdiagrammet eller pseudokoden i kapitlet (inte C#-koden).
9. Implementera en binär sökning genom att endast titta på strukturdiagrammet eller pseudokoden i kapitlet (inte C#-koden). Återanvänd dig av någon av sorteringsalgoritmerna från förra kapitlet.

2.3.2 Fördjupning: Sortering/sökning med strängar

Hittills har vi nöjt oss med att sortera/söka efter heltal och vi kan göra likadant med decimaltal. Däremot behöver vi lite nya knep om vi ska sortera strängar. Studera dokumentationen för `String.CompareTo()` på docs.microsoft.com/en-us/dotnet/api/system.string.compareto?view=netframework-4.7.2#System_String_CompareTo_System_String_ och genomför sedan nedanstående uppgifter.

10. Skriv ett program där användaren får mata in flera ord på en och samma rad. Placera varje ord i varsitt element i en strängarray (använd `Split`). Sortera sedan denna array i bokstavsordning och skriv ut till användaren. Använd valfri sorteringsalgoritm.
11. Skriv ett program som hanterar en highscore-lista. Användaren får skriva in tio olika rader med namn och poäng (separerade med mellanslag) och dessa placeras i varsin array. Sortera arrayen med poäng i fallande ordning och gör samtidigt samma förändringar i namn-arrayen (så att den blir sorterad på samma sätt som poängarrayen).
 - Utöka programmet så att man kan söka upp vem som har en viss poängsumma. Använd binär sökning.
 - Går det göra en binär sökning på namn istället (så att man får reda på hur många poäng den eftersökte spelaren har)? Varför/varför inte?
 - Genomför de eventuellt nödvändiga förändringarna för att man som användare ska kunna söka efter namn istället för poäng.
 - Går det skriva programmet så att användaren har möjlighet att söka både på namn och på poäng? Vad krävs då i så fall?

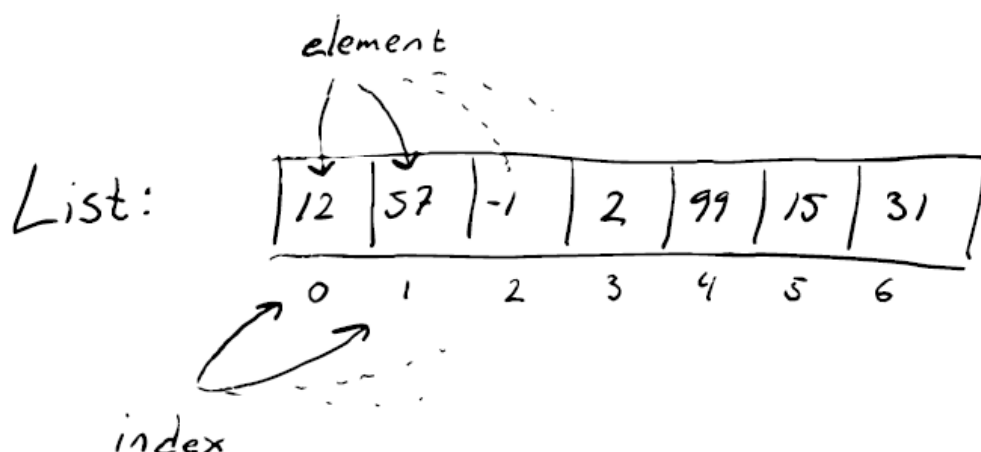
3 Samlingar

För att strukturera upp flera variabler av samma typ har vi hittills uteslutande använt arrayer (fält). Detta är långt ifrån det enda sättet att strukturera sina data i ett program, det finns massvis av så kallade datastrukturer. I C# finns många av dessa tillgängliga i ett paket som heter `Collections` - därav namnet samlingar på svenska. Vill man prata i mer generella termer använder man dock ofast datastrukturer.

Det finns alldeles för många samlingar att lära oss för att vi ska hinna inom ramen för Programmering 1 men tre av de vanligaste/enklast eller mest grundläggande är några som väldigt mycket liknar en array - nämligen `List`, `Stack` och `Queue`.

3.1 List

En lista (`List`) fungerar ungefär som en array men den har en del tillhörande, färdiga metoder som underlättar vårt arbete en hel del. Det är dock fortfarande fråga om att organisera flera data i led efter varandra och vi indexerar fortfarande med hjälp av heltal (räknat från 0). Vi använder fortfarande samma terminologi för innehållet på respektive plats - element. Den data vi väljer att organisera kan fortfarande vara av flera olika typer: `int`, `char`, `string` osv. Med andra ord har vi en `List` av olika typer (vi säger att vi specificerar en *typparameter*). Är det fråga om heltal kan vi illustrera listan som nedan - dvs. precis som en array.



Figur 3.1: En `List` av typen heltal.

En egenskap som skiljer `List` och `array` åt är dock att en listas storlek är dynamisk. Vi specificerar inte vilken storlek vi behöver på en lista när vi skapar den utan storleken kan växa och krympa under tiden programmet körs. Med andra ord är `List` väldigt användbar när vi ska lagra okänt antal data i en lista.

När vi skapar en `List` så ser den generella syntaxen ut såhär

```
1 List<datatype> variabelnamn = new List<datatype>();
```

där `datatype` givetvis ersätts med den typ av data vi vill lagra och `variabelnamn` med ett beskrivande namn för det vi lagrar. Om vi exempelvis ska lagra poäng som heltal använder vi följande:

```
1 List<int> poang = new List<int>();
```

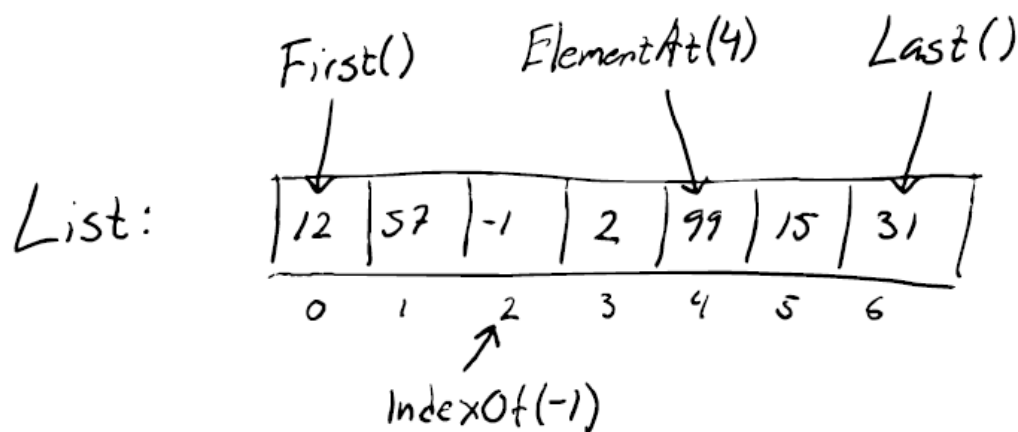
Till denna variabeln `poang` följer sedan en hel del metoder vi direkt kan använda för att bland annat lägga till, ta bort, hämta ut och rensa i listan.

3.1.1 List-metoder

Det finns som sagt en mängd olika metoder att använda när man arbetar med en `List` (komplett dokumentation på docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netframework-4.7.2#methods) men vi väljer här att ta en titt på några av dessa.

- `Add(element)`: Lägger till ett element längst bak i listan (storleken ökar).
- `Clear()`: Rensar/tömmer hela listan (alla element försvinner).
- `Contains(element)`: Returnerar `true` om `element` existerar i listan, annars `false`.
- `ElementAt(position)`: Hämtar elementet på index `position`.

- `First()`: Hämtar det första elementet i listan.
- `IndexOf(element)`: Returnerar ett heltal som är index för den position där `element` befinner sig i listan. Om elementet inte finns alls returneras `-1`.
- `Insert(position, element)`: Stoppar in `element` på index `position`. Alla efterföljande element flyttas ett steg i ordningen, dvs. inga element skrivs över (listans storlek ökar).
- `Last()`: Hämtar det sista elementet i listan.
- `Remove(element)`: Tar bort `element` från listan. Om flera element kan matchas tas det första i ordningen bort. Efter att ett element plockats bort flyttas alla efterföljande element framåt ett steg (listans storlek krymper).
- `RemoveAt(position)`: Tar bort det element som finns på index `position` och flyttar/krymper listan sedan enligt ovanstående punkt.



Figur 3.2: Kom ihåg att skilja på element och index.

I samtliga ovanstående punkter är `position` ett heltal och `element` ett element av den datatyp man använde när man skapade listan. Dessutom finns en egenskap `Count` som innehåller antalet element listan har just nu. Med andra ord är detta `List`-motsvarigheten till `Length` i en array.

Många av dessa metoder gör det väldigt mycket enklare för oss som programmerare men man ska alltid ha effektivitet i åtanke när man använder någon datastruktur. Läs igenom *Listoperationer och effektivitet* på sidan 206 i boken för att få en bättre uppfattning om detta i fallet `List`.

Ett exempelprogram som nyttjar några av ovanstående metoder kan se ut såhär:

```
1 static void Main(string[] args)
2 {
3     List<string> kurser = new List<string>();
4     kurser.Add("Programmering 1");
5     kurser.Add("Programmering 2");
```

```

6     kurser.Add("Webbutveckling 1");
7     kurser.Add("Webbutveckling 2");
8
9     SkrivUt(kurser);
10
11    Console.Write("Ange ny kurs att lägga till: ");
12    string nyKurs = Console.ReadLine();
13    kurser.Add(nyKurs);
14
15    SkrivUt(kurser);
16
17    Console.Write("Vilken kurs i ordningen vill du ta bort? ");
18    int index = int.Parse(Console.ReadLine());
19    kurser.RemoveAt(index);
20
21    Console.WriteLine("Det finns nu " + kurser.Count + " kurser i
        listan.");
22    Console.WriteLine("Först i ordningen ligger '" + kurser.First() + "
        '.");
23    Console.WriteLine("Sist i ordningen ligger '" + kurser.Last() + "'."
        );
24 }
25
26 static void SkrivUt(List<string> kurser)
27 {
28     for (int i = 0; i < kurser.Count; i++)
29     {
30         Console.Write(i + ": " + kurser.ElementAt(i) + " ");
31         Console.WriteLine();
32     }
33 }

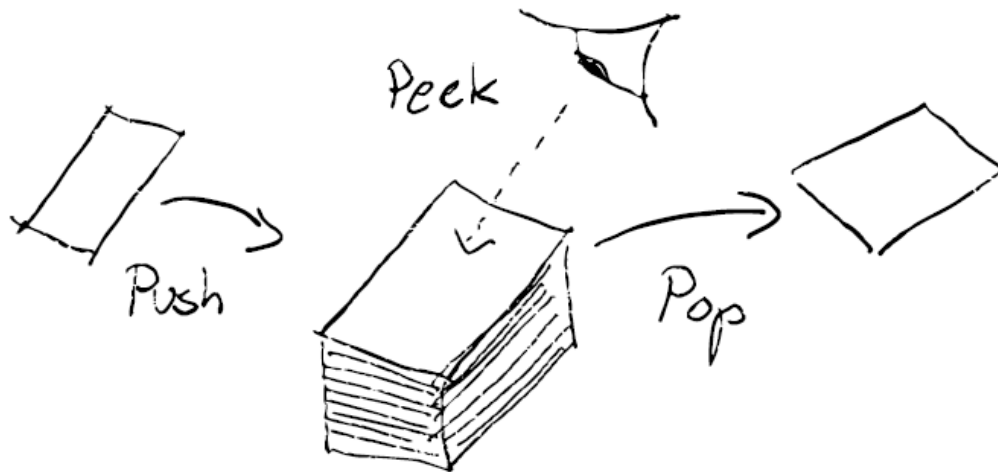
```

3.2 Stack

Den andra samligen vi tar en titt på heter **Stack**. En **Stack** liknar en **List** i ganska stor utsträckning, den kan lagra olika datatyper och gör det en och en efter varandra (så kallad diskret linjär ordning). Men i samband med en **Stack** nämner man sällan index eftersom att samtliga operationer på en **Stack** görs i en och samma ände.

Man kan likna en **Stack** med en trave papper på ett bord. Om vi ska lägga till ett papper till traven gör vi det längst upp. Pappret längst upp är också de enda vars innehåll vi kan se. Vill vi se innehåll på

pappret under måste vi lyfta bort det översta pappret. Dessa tre operationer kallar man för **Push**, **Peek** och **Pop** när man talar om en **Stack**.



Figur 3.3: Push, peek och pop på en Stack.

Lite mer formellt kan vi beskriva dessa tre **Stack**-metoder såhär:

- **Push(element)**: Lägger till **element** på toppen av stacken.
- **Peek()**: Läser det översta elementet på stacken.
- **Pop()**: Läser och plockar bort det översta elementet på stacken.

Utöver dessa finns även metoder **Clear()** som (precis som med **List**) rensar hela datastrukturen och egenskapen **Count** som anger hur många element som finns på stacken.

För att skapa en **Stack** skriver vi generellt följande kod:

```
1 Stack<datatyp> variabelnamn = new Stack<datatyp>();
```

Om vi t.ex. vill skapa en stack som lagrar strängar med namn ser koden ut såhär:

```
1 Stack<string> namn = new Stack<string>();
```

Med variabelnamnet **namn** kan vi sedan alltså använda **Stack**-metoderna. Exempelvis såhär:

```
1 Stack<string> namn = new Stack<string>();
2 namn.Push("Martin");
3 namn.Push("Andreas");
```

```

4  namn.Peek(); // --> "Andreas"
5  namn.Pop(); // --> "Andreas"
6  namn.Peek(); // --> "Martin"
7  namn.Push("Andrew");
8  namn.Count; // --> 2
9  namn.Clear();
10 namn.Count; // --> 0 (tom stack)

```

Märk väl att om vi vill behålla det element vi plockar ut från en *Stack* så får vi se till att lagra detta i en egen variabel. Se nedanstående exempel:

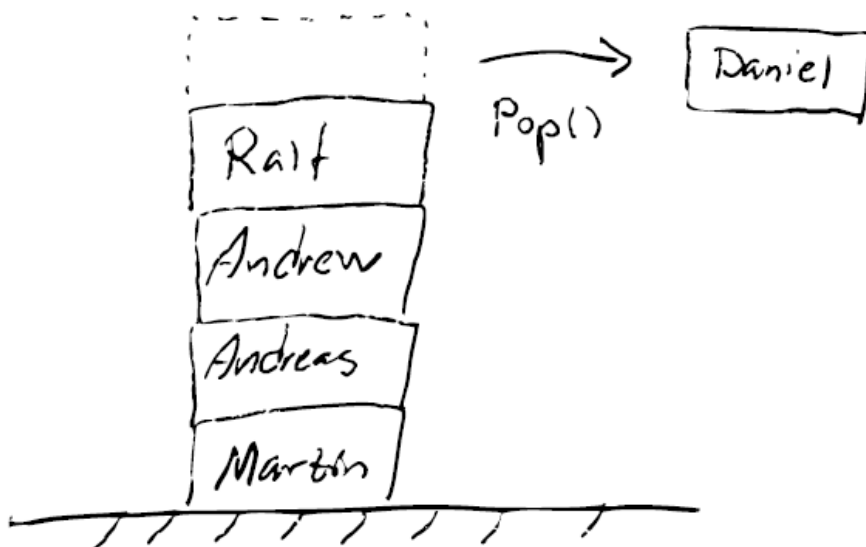
```

1  Stack<string> namn = new Stack<string>();
2  namn.Push("Martin");
3  string martin = namn.Pop();

```

Efter att ovanstående kod har körts är stacken tom men elementet "Martin" finns lagrat i variabeln *martin*.

När man illustrerar en *Stack* är det vanligt att man ritar den på höjden (och inte liggande som *List* och array) för att behålla analogin med en trave. Och precis som nämnts tidigare så utelämnar man index eftersom det ändå bara är det översta elementet man kan arbeta med. Datastrukturer som betar sig på detta vis (sist in, först ut) brukar man benämna som **LIFO** från engelskans *last in first out*.



Figur 3.4: En Stack ritar vi på höjden, utan index.

Vanliga användningsområden för *Stack* är olika former av att stega tillbaka eller när man vill åt något i

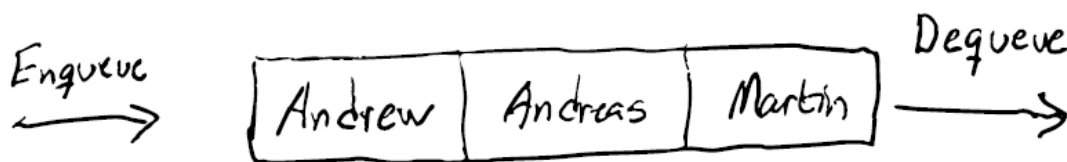
omvänd ordning. Vi kan t.ex. skriva ett ord baklänges genom att sätta in alla bokstäver i en stack och sedan plocka ut dom igen. Ett annat användningsområde man kan tänka sig är att vandra tillbaka samma väg man gått på en spelplan. Detta genom att placera varje position man besöker på stacken och sedan plocka ut dom och därmed besöka dom i omvänd ordning.

3.3 Queue

Den tredje datastrukturen vi tittar på är **Queue**. När vi nu känner till **Stack** som en LIFO-lista är det väldigt enkelt att beskriva **Queue** som en **FIFO**-lista, *first in first out*. Här handlar det alltså om att det element som först lades till är det som först kommer plockas ut.

En **Queue** har linjärt ordnade element men vi kan bara lägga till element i ena änden och plocka bort från den andra. Med andra ord precis som en vanlig kö i en matbutik eller liknande. **Peek()**, **Clear()** och **Count** för **Queue** är identiska med metoderna med samma namn för **Stack**. För att lägga till och ta bort använder vi följande:

- **Enqueue(element)**: Lägger till ett element till kön (i slutet).
- **Dequeue()**: Läser och plockar bort det element som ligger först i kön.



Figur 3.5: De unika metoderna för Queue.

Precis som med **List** och **Stack** kan **Queue** lagra vilken datatyp som helst och syntaxen för att skapa en kö är likadan som för de tidigare datastrukturerna:

```
1 Queue<datatyp> variabelnamn = new Queue<datatyp>();
```

För personer i en kö skulle det alltså bli:

```
1 Queue<string> personer = new Queue<string>();
```

Med variabeln **personer** kan jag sedan använda metoderna vi beskrev ovan.

```
1 Queue<string> personer = new Queue<string>();
```





```
2 personer.Enqueue("Martin");
3 personer.Enqueue("Andreas");
4 personer.Enqueue("Andrew");
5 string forstIKon = personer.Dequeue(); // "Martin" försvinner från kön
   men lagras i variabeln.
```

3.4 Andra samlingar

För den som vill fördjupa sig inom samlingar/*collections*/datastrukturer finns massor att lära sig. En bra ingång är Microsofts egna dokumentation på docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netcore-4.7.2#classes bara för att få en överblick över vad som finns.

Letar man efter mer genomarbetad litteratur finns en uppsjö av C#-böcker på engelska, men vill man åt något på svenska finns boken *Datatyper och algoritmer* av Janlert och Wiberg. Den handlar inte om C# i sig utan tar upp datastrukturer (och algoritmer) på ett generellt, något mer abstrakt plan.

3.5 Att göra

-  Läs mer om samlingar i boken på s. 200-215.
-  Läs mer om olika samlingsklasser på csharpskolan.se/article/samlingsklasser/.
-  Genomför självtestet *Samlingar* på Moodle: moodle.teed.se.
-  Genomför övningarna 12-16 här nedan.

3.5.1 Övningar

12. I denna, lite större övning ska du skapa ett program som hanterar en att göra-lista. Programmet ska kunna visa, lägga till och ta bort inlägg i listan. Dessutom ska man kunna ta bort och lägga till inlägg på en viss position. Till hjälp får du en grundstruktur med bland annat en **do-while**-sats att utgå ifrån. Detta upplägg är ofta lämpligt när man skriver någon form av ”menyprogram” i konsollmiljö. Den **List** som hela programmet skall jobba mot finns i form av variabeln **todo** och tanken är att du delar upp programmet i metoder enligt kommentarerna i skelettkoden.

```
1 static List<string> todo = new List<string>();
2
3 static void Main(string[] args)
4 {
5     int option;
```

```

6
7     do
8     {
9         Console.Clear();
10        PrintMenu();
11        Console.Write("Enter option: ");
12        option = int.Parse(Console.ReadLine());
13
14        if (option == 0)
15        {
16            //Anropa metod som skriver ut listan
17        }
18        else if (option == 1)
19        {
20            //Anropa en metod som lägger till i lista
21        }
22        else if (option == 2)
23        {
24            //Anropa metod som tar bort översta i lista
25        }
26        else if (option == 3)
27        {
28            //Anropa metod som tar bort på viss position
29        }
30        else if (option == 4)
31        {
32            //Anropa metod som lägger till på viss position
33        }
34
35    } while (option != -1);
36 }
37
38 static void PrintMenu()
39 {
40     Console.WriteLine("***** Menu *****");
41     Console.WriteLine("* 0. View ToDo-list  *");
42     Console.WriteLine("* 1. Add item        *");
43     Console.WriteLine("* 2. Remove item     *");
44     Console.WriteLine("* 3. Remove item #n  *");
45     Console.WriteLine("* 4. Insert at #n    *");
46     Console.WriteLine("* -1. Exit          *");
47     Console.WriteLine("*****");
48 }
    
```


13. Skapa ett program som har två listor som innehåller följande heltal: 5, 6, 20, 19, 5, 16, 20, 13, 1, 11 och 16, 19, 6, 1, 8, 17, 11, 1, 20, 13. Du kan skapa variabler av typen `list` och sätta in värdet direkt vid deklaration såhär:

```
1 int[] tal = { 6, 19, 6, 1, 8, 17, 11, 1, 20, 13 };
2 List<int> lista = new List<int>(tal);
```

En array skapas visserligen i onödan men ditt kodande blir något enklare. Din uppgift är sedan att ta bort alla tal ur respektive lista som inte finns i den andra - dvs. endast unika tal skall finnas kvar. Dessa skall sedan kopieras in till en ny, gemensam lista som skrivs ut till användaren.

14. Skapa ett program som läser in en sträng och sedan kontrollerar om detta är en palindrom. Uppgiften skall lösas med hjälp av en `Stack`.
15. Skriv ett program som kontrollerar att en uppsättning paranteser är korrekt. Användaren skall kunna skriva in paranteserna och programmet svarar sedan med "OK" eller "FEL". Du skall lösa programmet med hjälp av en `Stack` och kan till hjälp ta denna pseudokod (ej färdigt program):

```
1 så länge tecken kan läsas in och fel saknas i parantesföljden
2   om det inlästa tecknet är en vänsterparantes
3     lägg det på stacken
4   om det inlästa tecknet är en högerparantes
5     om stacken är tom
6       parantesföljden är felaktig
7     annars
8       ta bort topp-elementet från stacken
9 om stacken inte är tom
10  parantesföljden är felaktig
```

Du kan testköra med följande korrekta följder `()()`, `(())` och dessa två felaktiga `))()`, `((()`.

16. Skriv ett program som hanterar en enkel kö med personnamn. Återanvänd tänket/upplägget från uppgiften med `ToDo`-listan.
- Programmet skall ha en meny med alternativen "Lägg till i kön", "Ta bort från kön" och "Avsluta".
 - Programmet ska fortsätta köra kontinuerligt tills dess att användaren avslutar.
 - När man lägger till i kön får man skriva in namnet på den person man vill lägga till.
 - När man tar bort någon ifrån kön skrivs dennes namn ut.
 - I menyn skrivs även alltid antalet personer i kön ut.
 - Använd datastrukturen `Queue` och dess metoder för att lösa kösystemet.