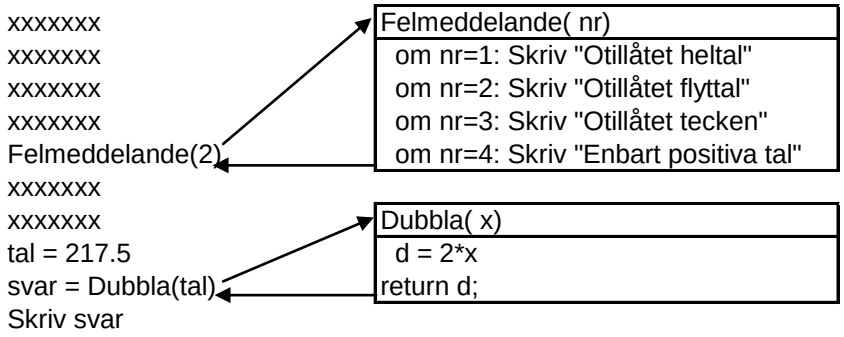
Innehållsförteckning

4 Rekursion

I C++ kan funktioner anropa sig själva. Detta kallas rekursion.

Först en repetition av hur funktionsanrop görs. Funktionsanrop kan illustreras så här:



I det första fallet anropas funktionen Felmeddelande() med argumentet 2. Detta innebär att parametern nr sätts lika med 2 och satsen i anslutning till nr=2 utförs. Strängen "Otillåtet flyttal" skrivs ut på skärmen.

I det andra fallet anropas funktionen Dubbla() med argumentet tal = 217,5. Funktionen multiplicerar talet med faktorn 2 och returnerar värdet 535, vilket sedan skrivs ut av det anropande programmet.

4.1 Fakultet

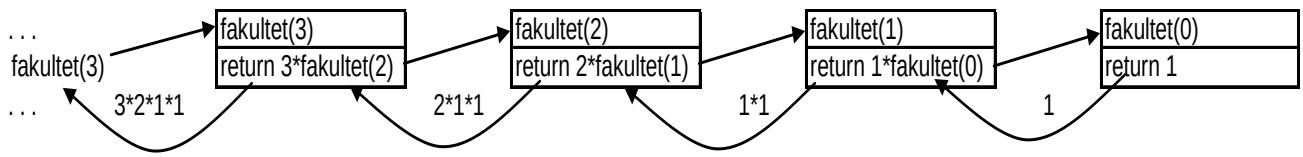
Fakultet är ett exempel då man kan använda rekursion. Antag att man vill räkna ut 3! (uttalas 3 – fakultet) då gör man så här: 3! = 3*2*1, vilket uträknat blir = 6.

Man kan då konstatera att 3!=3* 2! och 2! = 2*1! och 1! = 1*0! Observera att 0! =1.

Här har man ett rekursivt förlopp vilket kan utnyttjas när man skriver en funktion för beräkning av n! En sådan funktion kan se ut så här:

```
int fakultet(int n)
{
  if(n>0)
    return n*Fakultet(n-1);     // Rekursivt anrop om n > 0
  else
    return 1;                   // Returnera 1 om n <= 0
}
```

Anropet int f = fakultet(3); ger följande funktionsanropskedja
f = 3*fakultet(2)
f = 3*2*fakultet(1)
f = 3*2*1*fakultet(0)
f = 3*2*1*1



Observera att beräkningen görs vid återhoppen från funktion till funktion

Ett program som använder funktionen för beräkning av fakultet:

```
// rek_010 Version 10
// Rekursion: Beräkna fakultet
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
using namespace std;
//-----------------------------------------------------------------------------
// Funktionsprototyp
//-----------------------------------------------------------------------------
unsigned int fakultet(int n);

//-----------------------------------------------------------------------------
// Huvudprogram
//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Beräkna och skriv ut 3-fakultet 3!= 3*2*1
//-----------------------------------------------------------------------------
  cout << "6! = " <<  fakultet(6) << endl << endl;

  return 0;
}
//-----------------------------------------------------------------------------
// Funktionsdefinition
//-----------------------------------------------------------------------------
unsigned int fakultet(int n)
{
  if(n > 1)
    return n * fakultet(n-1);    // Rekursivt anrop om n > 0
  else
    return 1;                  // Returnera 1 om n <= 0
}
```

4.2 Linjalen

Linjalen är ytterligare ett exempel på rekursion. För att rita strecken på en linjal gör vi så att en sträcka delas i två lika delar som sedan delas i två lika delar. Detta fortsätter tills att vi har lagom många streck på linjalen. Det kan t.ex. se ut så här:

Vi har först en sträcka som är markerad där den börjar och sluta

```
|                                                               |
```

Sedan sätter vi ut ett streck på mitten av denna sträcka

```
|                               |                               |
```

Vi har nu två sträckor vilka delas på mitten

```
|               |               |               |               |
```

De nu skapade delsträckorna delas på mitten i sin tur

```
|       |       |       |       |       |       |       |       |
```

Och så fortsätter vi att dela på mitten tills vi kommer till

```
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
0       4       8       12      16      20      24      28      32
```

Detta är ett utmärkt exempel på problem som kan lösas med rekursion. Här visas en rekursiv funktion som jobbar på detta sätt:

```
void subdivide(string &r, int low, int high, int level)
{
  if(level > 0)
  {
    int mid = (high + low) / 2;
    r[mid] = '|';
    cout << r << endl;
    subdivide(r,low,mid,level-1);
    subdivide(r,mid, high, level-1);
  }
  // Om level ==0 sker återhopp till anropande funktion
}
```

Kommentarer:

- Funktionen returnerar inget värde. Den anropar sig själv två gånger, en gång för vänstra halvan och en gång för högra halvan.
- En sträng *r* används för att simulera den sträcka som ska delas.
- Strängen, r, skickas med som en referensdeklarerad parameter. Detta innebär att det är samma sträng som hanteras i alla anrop i rekursionskedjan.
- | används som mittenmarkering.
- *low* = index för sträckans vänstra position.
- *high* = index för sträckans högra position.
- *level* = antal gånger som sträckan ska delas. Det blir 2 upphöjt till *level* stycken delar.

Hela programmet:

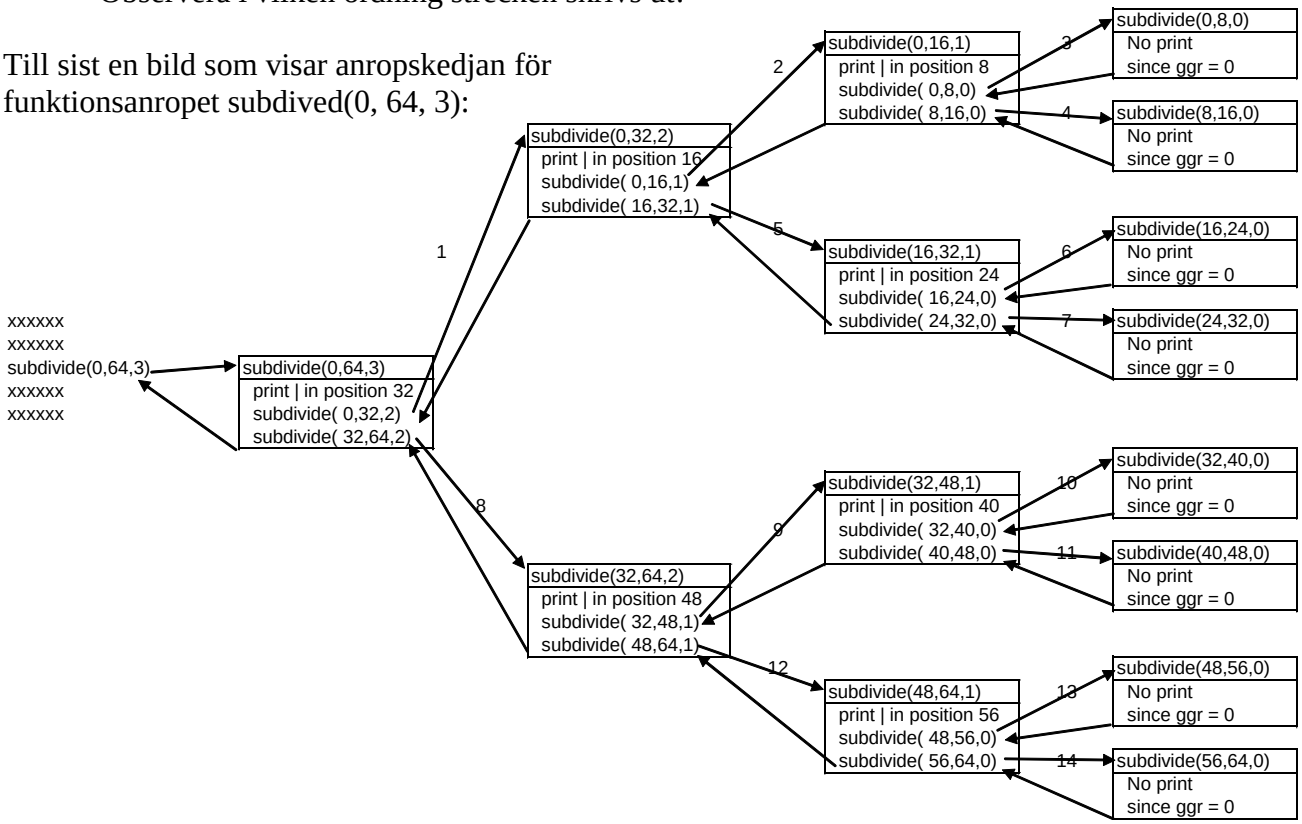
```
// rek_020 Version 10
// Rekursion: Rita en "linjal"
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>
#include <string>
using namespace std;
//-----------------------------------------------------------------------------
void subdivide(string &r, int low, int high, int level);
//-----------------------------------------------------------------------------
int main()
{
  const int LEN = 65;      // Linjalens längd
  const int SPLITNUM = 6;  // Linjalen delas i 2^SPLITNUM delar
//-----------------------------------------------------------------------------
// Variabler. Deklaration och initiering
//-----------------------------------------------------------------------------
  int max = LEN;
  int min = 0;
  string ruler(LEN+1,' ');       // Initiera strängen ruler med LEN+1 st ' '
  ruler[min] = ruler[max] = '|'; // Sätt in ett | först och sist i ruler
//-----------------------------------------------------------------------------
// Anrop av Subdivide() .
//-----------------------------------------------------------------------------
  cout << ruler << endl;         // Skriv ändlägena
  subdivide(ruler,min,max,SPLITNUM);  // för demo av utskrift av subdivide
  return 0;
}
```

Kommentarer:

- `string ruler(LEN, ' ')` initierar `ruler` med LEN stycken mellanslag (' ')!

- Observera i vilken ordning strecken skrivs ut!

Till sist en bild som visar anropskedjan för funktionsanropet subdived(0, 64, 3):



Kommentarer:

- Siffrorna vid pilarna anger i viken ordning anropen görs
- Observera att funktionen anropas 8 gånger utan att något skrivs!!
- Strängen r är borttagen ur denna figur.
- "ggr" i denna figur = "level" i funktionen

5 Sortering

Man vill ofta sortera en lista med t.ex. namn. Detta kan göras på många olika sätt. Vi ska här gå igenom några olika sorteringsalgoritmer sorteringsalgoritmerna Bubblesort, Selectionsort och quicksort. För Bubblesort och quicksort visas två varianter.

5.1 Bubblesort I

Elementen jämförs parvis n-1 st gånger. n är antalet element Antag att nedanstående array med heltal ska sorteras så att minsta tal placeras längst till vänster. Bubblesort gör så här:

[6][2][1][3][4][5][8][7][9][0]	Startarray
[2][6][1][3][4][5][8][7][9][0]	2 och 6 har bytt plats
[2]**[1][6]**[3][4][5][8][7][9][0]	1 och 6 har bytt plats
[2][1]**[3][6]**[4][5][8][7][9][0]	3 och 6 har bytt plats
[2][1][3]**[4][6]**[5][8][7][9][0]	4 och 6 har bytt plats
[2][1][3][4]**[5][6]**[8][7][9][0]	5 och 6 har bytt plats
[2][1][3][4][5][6]**[7][8]**[9][0]	7 och 8 har bytt plats
[2][1][3][4][5][6][7][8]**[0][9]**	0 och 9 har bytt plats
[1][2][3][4][5][6][7][8][0][9]	1 och 2 har bytt plats
[1][2][3][4][5][6][7]**[0][8]**[9]	0 och 8 har bytt plats
[1][2][3][4][5][6]**[0][7]**[8][9]	0 och 7 har bytt plats
[1][2][3][4][5]**[0][6]**[7][8][9]	0 och 6 har bytt plats
[1][2][3][4]**[0][5]**[6][7][8][9]	0 och 5 har bytt plats
[1][2][3]**[0][4]**[5][6][7][8][9]	0 och 4 har bytt plats
[1][2]**[0][3]**[4][5][6][7][8][9]	0 och 3 har bytt plats
[1]**[0][2]**[3][4][5][6][7][8][9]	0 och 2 har bytt plats
[0][1][2][3][4][5][6][7][8][9]	0 och 1 har bytt plats

Vi har nu stegat igenom arrayen n-1 = 9 gånger och gjort 81 jämförelser. Vi måste stega genom arrayen n – 1 (här 9) gånger för att vi ska vara säkra på att den är riktigt sorterad. I detta fall krävs det 9 genomstegningar för att 0:an ska komma längst till vänster. Här visas koden för en algoritm som utför denna sortering:

```
for(int pass=0; pass < n-1; pass++)
  for(int i=0; i < n-1; i++)
    if(a[i] > a[i+1])           // Jämför elementen i och i+1
    {
      swap(a[i], a[i+1]);        // Byt plats om element i > element i+1
    }

//Där funktionen swap() byter plats på två element och definieras så här:
void swap(int &a, int &b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

Kommentarer:

- a är arrayen som ska sorteras
- n är antal element i arrayen
- variabeln *pass* håller reda på vilken gång (passering) som arrayen stegas igenom
- variabeln *i* är index för det vänstra av två intilliggande element som jämförs

- algoritmen kallas för Bubblesort pga att elementen förflyttar sig som en bubbla genom arrayen, se t.ex. på 6:an eller 0:an ovan
- Bubblesort är enkel men den är **väldigt** ineffektiv!! För små arrayer fungerar den **hjälpligt**.

Denna bubblesort-variant och kommande sorteringsexempel finns i sort_010.
Källkod för sorteringsalgoritmerna hittar du i definitionsfilen sort.cpp

```
// sort_010.cpp Version 10
// Per Ekeroot
// 2014-01-13
//-----------------------------------------------------------------------------
#include "sort.h"
#include <ctime>
#include <random>
#include <iostream>
using namespace std;

//-----------------------------------------------------------------------------
int main()
{
  const size_t SIZE = 450;
  int *array = new int [SIZE];

  // Slumpa tal
  default_random_engine generator(static_cast<unsigned>(time(0)));
  uniform_int_distribution<int> random(0,999);

  for(int i=0; i < SIZE; i++)
    array[i] = random(generator);

  // Skriv ut osorterad array
  cout << "*** Random array ***" << endl;
  printArray(array, SIZE, 15, 4);
  cout << endl;
  cout << "Press ENTER to continue!";
  cin.get();

  //Sortera
  cout << "Sorting . . . ." << endl;
  // Olika sorteringsrutiner
  // Ta bort kommenteringen för den sorteringsrutin du vill testa
  bubbleSort1(array, SIZE);  // Bubble sort 1
  //bubbleSort2(array, SIZE);  // Bubble sort 2
  //selectSort(array, SIZE);   // Select sort (Urvalssortering)
  insertSort(array, SIZE);   // Insertion sort
  //q1(array, SIZE);           // Quicksort 1
  //q2(array, SIZE);           // Quicksort 2

          // Skriv ut sorterad arrayresultat
  cout << endl << endl << "Sorted array" << endl << endl;

  printArray(array, SIZE, 15, 4);
  cout << "Press ENTER to continue!";
  cin.get();

  delete [] array;
  return 0;
}
```

Kommentarer:

- Skapa en array för 450 heltal.
- Slumpa tal till arrayen och skriv ut den.
- Sortera sedan arrayen med någon av sorteringsalgoritmerna.
- printArray() skriver ut arrayen med ett angivet antal element på varje rad. Du hittar den i sort.cpp

5.2 Bubblesort II

I denna variant av Bubblesort tar man hänsyn till om arrayen är sorterad eller inte. Man kontrollerar efter varje genomgång av arrayen om några element bytt plats. Om inga byten är gjorda kan man dra slutsatsen att arrayen måste vara sorterad och sorteringen avbryts. En algoritm för detta:

```
bool sorted =false;
for(int pass=0; pass < n-1 && !sorted ; pass++)
{
  sorted = true;
  for(int i=0; i < n-1; i++)
    if(a[i] > a[i+1])            // Jämför elementen i och i+1
    {
      swap(a[i],a[i+1]);         // Byt plats om element i > element i+1
      sorted = false;
    }
}
```

Kommentarer:

- a är arrayen som ska sorteras
- n är antal element i arrayen
- variabeln sorted håller reda på om arrayen är sorterad
- om sorted är **true** när en genomgång av arrayen gjorts är den sorterad och sorteringen avbryts
- om två element byter plats sätts sorted till **false** och sorteringen fortsätter minst en gång till
- om arrayen är sorterad redan innan sorteringen påbörjas räcker det att gå igenom arrayen en gång.
- Bubblesort II är ofta något effektivare är Bubblesort I. Den är dock fortfarande mycket ineffektiv.

5.3 Selectionssort

Selectionsort (urvalssortering) är nästan lika enkel att koda som bubblesort men är mycket effektivare. Den fungerar så här:

- Jämför första elementet med alla övriga. Byt så att det minsta kommer först
- Jämför andra elementet med övriga till höger. Byt så att det minsta av dessa kommer till andra elementet
- Jämför tredje elementet med övriga till höger. Byt så att det minsta av dessa kommer till tredje elementet
- Fortsätt på detta sätt tills att det bara är ett element kvar till höger om det man jämför med.

Om vi tar samma exempel som för Bubblesort I får vi:

[6][2][1][3][4][5][8][7][9][0]	Startarray
[0][2][1][3][4][5][8][7][9]**[6]**	Fås nu genom att 0 och 6 byter plats
[0]**[1][2]**[3][4][5][8][7][9][6]	1 och 2 byter plats
[0][1][2][3][4][5]**[6]**[7][9]**[8]**	6 och 8 byter plats
[0][1][2][3][4][5][6][7]**[8][9]**	8 och 9 byter plats

Koden för denna algoritm ser ut så här:

```
int smallIdx = 0;
for(int i=0; i < n - 1; i++)
{
  smallIdx = i;           //Index för det minsta elementet till höger om pos i
  for( int j= i+1; j < n; j++) // Sök det minsta "osorterade" elementet
    if(a[j] < a[smallIdx])
      smallIdx = j;            // Spara index för det minsta elementet

  if(smallIdx != i)            // Byt plats om det fanns något mindre än a[i]
    swap(a[i], a[smallIdx]);
}
```

Kommentarer:

- a är arrayen som ska sorteras
- n är antal element i arrayen
- n håller rätt på det vänstra elementet
- j håller rätt på det högra elementet
- observera att j börjar ett steg till höger om i `(int j = i+1)`
- smallIdx är index till det element som innehåller det minsta talet för tillfället
- fördelen med denna algoritm är att den är relativt effektiv och samtidigt ganska enkel att skriva kod för
- en nackdel är att den inte tar hänsyn till om arrayen redan är sorterad

5.4 Quicksort

Den sorteringsalgoritm som anses vara mest effektiv för stora arrayer är Quicksort. Den är en rekursiv sorteringsalgoritm. Det är inte helt enkelt att förklara hur den jobbar, här kommer ett försök:

1. Utgå från en osorterad array
2. Kalla elementet som ligger längst till vänster i arrayen för pivotelement.
3. Gå från vänster och börja med andra elementet för att leta rätt på första talet som är större eller lika med pivotelementet
4. Gå från höger och leta rätt på första talet som är mindre eller lika med pivotelementet
5. Låt dessa tal byta plats.
6. Fortsätt med punkterna 3 - 5 tills alla tal mindre än pivotelementet finns i vänstra delen och de som är större än pivotelementet i högra delen.
7. Flytta pivotelementet så att det kommer mellan den vänstra och den högra delen.
8. Arrayen består nu av tre delar: en vänsterdel, ett element "i mitten" som har rätt position och en högerdel. Elementen i vänsterdelen är mindre än "mittelementet" och elementen i högerdelen är större än "mittelementet". Se fig. nedan!
9. Gör 2 – 7 för den del av arrayen som ligger till vänster om pivotelementet om delarrayen innehåller två eller fler element (rekursion!)
10. Gör 2 – 7 för den del av arrayen som ligger till höger om pivotelementet om delarrayen innehåller två eller fler element (rekursion!)

Så här ser arrayen ut efter att första steget i QuickSort (punkt 8 ovan) är utförd:

a[0], a[1], . . .,a[j-1], a[j], a[j+1], a[j+2], . . ., a[n-1],

- a[j] är det element som är placerad i sorterad position
- a[0], a[1], . . ., a[j-1] är alla mindre än a[j], men inte sorterade
- a[j+1], a[j+2], . . ., a[n-1] är alla större än a[j] , men inte sorterade

Vi tar samma array som ovan och visar hur quicksort sorterar den

[3][9][6][2][5][8][1][7][4]	Startarray
[3][1][6][2][5][8][9][7][4]	1 och 9 har bytt plats
[3][1][2][6][5][8][9][7][4]	2 och 6 har bytt plats
[2][1][3][6][5][8][9][7][4]	2 och 3 har bytt plats

Här har 3:an placerats i rätt position.
Vi har nu två delarrayer som ska sorteras:
[2][1] och [6][5][8][9][7][4]

Fortsättningsvis arbetar denna variant av quicksort så här:

[1][2]_____	1 och 2 har bytt plats
[1][2]_____	1 och 1 har bytt plats
_____[4][5][8][9][7][6]	4 och 6 har bytt plats
_____[4][5][8][9][7][6]	4 och 4 har bytt plats
_____[5][8][9][7][6]	5 och 5 har bytt plats
_____[6][9][7][8]	6 och 8 har bytt plats
_____[6][9][7][8]	6 och 6 har bytt plats
_____[8][7][9]	8 och 9 har bytt plats
_____[7][8][9]	7 och 8 har bytt plats

Kommentarer: Enbart den delarray som sorteras för tillfället skrivs ut. Övriga element är markerade med _.

Koden för denna variant av quicksort ser ut så här:

```
void quick1(int a[], int first, int last)
{
  if(first < last)
  {
    int low  = first;
    int high = last;
    if(a[first] > a[last])     // Place a sentinel in the position a[last ] !!??
     swap(a[first], a[last]);

    do
    {
      // Gå från från början och sök första värdet som är större än a[first]
      do{ low++; }while(a[low] < a[first]);

      // Gå från slutet och sök första värdet som är mindre än a[first]
      do{ high--;}while(a[high] > a[first]);

      // Byt plats på a[low] och a[high] om low < high
      if(low < high)
        swap(a[low], a[high]);
    }while(low <= high);            // Fortsätt tills low > high

    swap(a[first], a[high]);        // Placera a[first] i sorterad position

    quick1(a, first, high-1);               // Sortera vänster dellista
    quick1(a, high+1, last);                // Sortera höger dellista
  }
}
```

Eftersom quicksort är en rekursiv algoritm måste den placeras i en funktion för att enkelt och snyggt kunna anropa sig själv. Det brukar vara snyggt att lägga in anropet till själva quicksort i en wrapper-funktion, så här:

```
void q1(int a[], int n)
{
  quick1(a,0,n-1);
}
```

Kommentar:

- a är arrayen som ska sorteras
- n är antal element i arrayen
- Quicksort är mycket effektiv för mycket stora arrayer
- Quicksort är ineffektiv för en redan sorterad array
- Quicksort är ineffektiv för små arrayer, 0 – 100 element.

5.5 Quicksort II

Det finns några varianter på quicksort. De skiljer sig främst åt genom hur pivotelementet väljs. Här visas en variant där pivotelementet väljs som det element som ligger mitt i arrayen. Denna variant jobbar så här:

1. Utgå från en osorterad array
2. Kalla elementet som ligger mitt i arrayen för pivotelement.
3. Gå från vänster, börja med andra elementet och leta rätt på första talet som är större eller lika med pivotelementet
4. Gå från höger och leta rätt på första talet som är mindre eller lika med pivotelementet
5. Låt dessa tal byta plats.
6. Fortsätt med punkterna 3 - 5 tills alla tal mindre än pivotelementet finns i vänstra delen och de större än pivotelementet i högra delen.
7. Arrayen består nu av tre delar: en vänsterdel, ett element "i mitten" som har rätt position och en högerdel. Elementen i vänsterdelen är mindre än "mittelementet" och elementen i högerdelen är större än "mittelementet". Se fig. nedan!
8. Gör 2 – 7 för den del av arrayen som ligger till vänster om pivotelementet om delarrayen innehåller två eller fler element (rekursion!)
9. Gör 2 – 7 för den del av arrayen som ligger till höger om pivotelementet om delarrayen innehåller två eller fler element (rekursion!)

Så här kan det se ut:
[3][9][6][2][5][8][1][7][4] Startarray

[3][4][6][2][5][8][1][7][9] 4 och 9 har bytt plats
[3][4][1][2][5][8][6][7][9] 1 och 6 har bytt plats
[3][4][1][2][5][8][6][7][9] 5 och 5 har bytt plats

- 5:an är på rätt plats från början
- Vi har nu fått delarrayerna:

[3][4][1][2], [5] , [8][6][7][9]

Fortsättningen ser ut så här:

[3][2][1][4]_____	2 och 4 har bytt plats
[1][2][3]_____	1 och 3 har bytt plats
[1][2][3]_____	2 och 2 har bytt plats
_____[6][8][7][9]	6 och 8 har bytt plats
_____[7][8][9]	7 och 8 har bytt plats
_____[8][9]	8 och 8 har bytt plats

Koden för denna variant kan se ut så här, se nästa sida:

```
void quick2(int a[], int first, int last)
{
  int low = first;
  int high = last;
  int x = a[(first+last)/2];      // Välj mittvärdet som pivotelement

  do
  {
    // Gå från början och sök första värdet som är större än x
    while(a[low] < x){low++;}

    // Gå från slutet och sök första värdet som är mindre än x
    while(a[high] > x){high--;}

    if(low<=high)
    {
       swap(a[low],a[high]);
       low++;
       high--;
    }
  }while(low <= high); // Loopa genom vektorn medan low <= high

  // Rekursion
  // "Sortera" vänster halva om high inte nått vektorns första element
  if(first < high) quick2(a,first,high);

  // "Sortera" höger halva om low inte nått vektorns sista element
  if(low < last) quick2(a,low,last);
}
```

Kommentarer:

- QuickSort blir effektiv eftersom de tal som byts ”kommer på rätt halva” i samma byte!
- QuickSort jobbar hela tiden med ”orginalarrayen” a (pekaranrop!)

5.6 Jämförelse mellan de olika sorteringsalgoritmerna.

För att ge en fingervisning om sorteringsalgoritmernas effektivitet har jag sorterat arrayen

```
int a[10] = {6, 2, 1, 3, 4, 5, 8, 7, 9, 0};
```

och räknat antalet jämförelser och antalet byten som görs då arrayen a sorteras i stigande ordning. Observera att 0:an är placerad i sämsta möjliga position. Efter sortering får arrayen innehållet a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

Resultat:

Algoritm	**Antal jämförelser**	**Antal byten**
BubbleSort I	81	16
BubbleSort II	81	16
SelectionSort	45	4
QuickSort1	46	11
QuickSort2	14	9

Vi byter till arrayen: int a[10] = {3, 9, 6, 2, 0, 5, 8, 1, 7, 4 };
Resultatet blir då:

Algoritm	**Antal jämförelser**	**Antal byten**
BubbleSort I	81	24
BubbleSort II	63	24
SelectionSort	45	8
QuickSort1	33	13
QuickSort2	19	10

Om vi istället slumpar en array med 1000 heltal får vi:

Algoritm	**Antal jämförelser**	**Antal byten**
BubbleSort I	998 001	253 384
BubbleSort II	960 039	253 384
SelectionSort	499 500	995
QuickSort1	11 842	2 653
QuickSort2	7 653	2 662

Kommentarer:

- Det är samma array som jämförs i fallet då 1000 heltal slumpas.
- Visuella demonstrationer av olika sorteringsalgoritmer finns bl. a. på webbsidan: http://fy.chalmers.se/~f96hajo/old/sortdemo/sd_swe.htm

6 Sökning

I det här avsnittet ska vi utföra sökning i en array. Vi ska se på två olika sätt att söka:

- Linjär sökning
- Binär sökning

6.1 Linjär sökning

Vid linjär sökning i en array söker man igenom arrayen från första elementet, element för element, tills man hittar den data man söker eller tills listan är slut. Här följer ett exempel på en funktion som söker efter värdet *value* i arrayen *a* som har *n* element (se search.cpp):

```
int linSearch(const int a[], int n, int value )
{
   for( int i=0; i < n; i++)
     if(a[i] == value)
       return i;

   return -1;
}
```

- funktionen returnerar index för **första** förekomsten av *value* i arrayen a
- om sökningen misslyckas returneras -1

Exempel på användning: sök efter talet 35 i arrayen a som har SIZE element

```
  int idx = linSearch(a, SIZE, 35);
  if(idx>=0)
  {
    cout << endl << "Index för eftersökt tal = " << idx << endl;
    cout << "Talet = " << a[idx] << endl;
  }
  else
    cout << endl << "Talet 35 saknas i listan!";
```

Kommentarer:

- Om sökningen lyckas returneras index för elementet i vilket eftersökt data finns. Detta index används i exemplet ovan för att skriva ut sökt data tillsammans med lämplig text.
- Om sökningen misslyckas returnerar sökfunktionen -1. Detta används tillsammans med en if-funktion för utskrift av lämpligt meddelande.
- I detta fall är "sökvärde" och data i arrayen samma sak. Men om vi istället haft en array med objekt där varje objekt innehåller namn, adress, personnummer mm och sökningen görs på personnummer, så blir det vettigt/logiskt att låta sökfunktionen returnera index för elementet där eftersökt data hittats.
- Det sökta talet (35) är här hårdkodat, normalt låter man användaren ange tal att söka efter.

I följande program testas både linjärsökning och binärsökning. Binärsökningen beskrivs längre fram.

```
// search_010 Version 10
// Demo av linjärsökning och bijnärsökning
// Per Ekeroot 2014-01-13
#include <iostream>
#include <iomanip>
#include <random>
#include <ctime>
#include "search.h"
#include "sort.h"
using namespace std;

int main()
{
// Deklarera och initiera konstanter och variabler
   const size_t SIZE = 100;
   int a[SIZE];

// Slumpa heltal till arrayen
  default_random_engine generator(static_cast<unsigned>(time(0)));
  uniform_int_distribution<int> random(0,99);
  for(int i=0; i<SIZE; i++)
    a[i] = random(generator);

// Skriv ut den slumpade arrayen
  cout << "Random array" << endl;
  printArray(a,SIZE,15,4);

//-----------------------------------------------------------------------------
// Sök efter talet 35
//-----------------------------------------------------------------------------
// Använd linjärsökning
//-----------------------------------------------------------------------------
   int idx = linSearch(a, SIZE, 35);

//-----------------------------------------------------------------------------
// Använd binärsökning
//-----------------------------------------------------------------------------
//  q2(a,SIZE);                        // Arrayen måste sorteras innan binärsökning
  cout << endl << "Sorted array" << endl;
  printArray(a,SIZE,15,4);
//  int idx = binSearch(a,SIZE,35);

  if(idx>=0)
  {
    cout << endl << "Index of the search number = " << idx << endl;
    cout << "The number = " << a[idx] << endl;
  }
  else
    cout << endl << "The number 35 is not in the list!"<< endl;

  return 0;
}
```

Kommentarer:

- printArray() finns i sort.cpp
- För att testa binärsökning kommenterar du raden med linSearch() och avkommentera raderna med q2() och binSearch()

6.2 Binärsökning

Vid sökning i stora arrayer kan linjärsökning vara tidsödande. Förutsatt att arrayen är **sorterad** är binärsökning mycket effektivare.

Principen för binärsökning:

1. Se till att arrayen är sorterad
2. Leta reda på elementet som är mitt i arrayen.
3. Om detta är det sökta talet är sökningen färdig. Index för hittat tal returneras.
4. Om mittalet inte är det sökta fortsätter man sökningen i vänster halva om *sökt tal* < *mittalet* annars i höger halva
5. Upprepa 2 – 4 tills att bara ett element återstår
6. Om talet inte hittas i 3 (när bara ett element återstår), så finns inte det sökta talet i arrayen. Returnera -1!

Ett exempel på hur detta kan kodas (se också search.cpp). Vi söker efter värdet *value* i arrayen *a* som har *n* element:

```
int binSearch(int a[], int n, int value)
{
   bool found = false;
   int high = n;
   int low  = 0;

   int mid = (low + high) /2;
   while(!found && high >= low)
   {
     if(value==a[mid])
       found = true;
     else if(value < a[mid])
       high = mid - 1;
     else
       low = mid + 1;
     mid = (low + high) / 2;
   }
   return (found?mid:-1);
}
```

Kommentarer:

- Funktionen returnerar index i arrayen om sökningen lyckas
- Funktionen returnerar -1 om sökningen misslyckas

Anrop av funktionen visas i följande exempel:

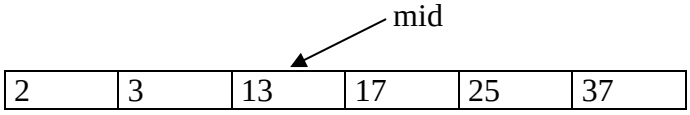
```
  q2(a,SIZE);                        // Arrayen måste sorteras innan binärsökning
  int idx = binSearch(a,SIZE,35);

  if(idx>=0)
  {
    cout << endl << "Index för eftersökt tal = " << idx << endl;
    cout << "Talet = " << a[idx] << endl;
  }
  else
    cout << endl << "Talet 35 saknas i listan!"<< endl;
```

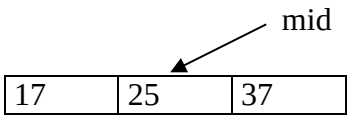
Kommentarer:
Observera att q2() (se sort.cpp) anropas innan den binära sökningen utförs.

6.2.1 Ett förklarande exempel

Sök talet 25 i arrayen 2, 3, 13, 17, 25, 37.



high = 5
low = 0 ger mid=(5+0)/2= 2 (heltalsdivision!)
25 > a[2] ger low = mid +1 = 3 (high behåller värdet 5)



high=5
low = 3 ger mid = (5+3)/2=4
25 == a[4] ger found =true