

# Procedurer og funktioner - iteration og rekursion

## Procedurer

De første procedurer vi så på var knyttet til handlinger, der skulle udføres, fx at klikke på en knap for at lukke en form eller afslutte et program.

### Eksempel 1

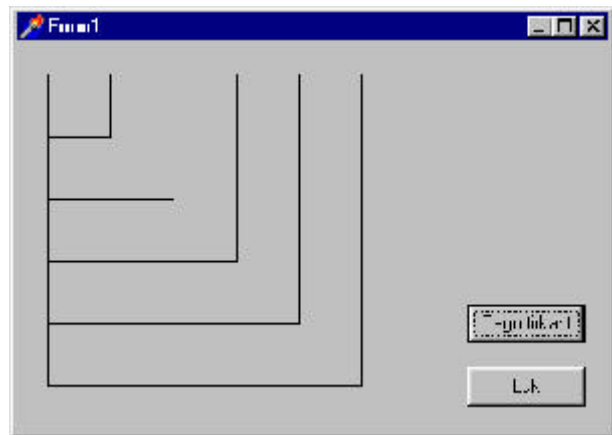
```
procedure TForm1.Button1Click(Sender: TObject);
begin
    close
end;
```



### Eksempel 2

Vi har også set på en klik-procedure, der kalder en anden procedure. I dette eksempel er det proceduren TForm1.Button2Click der kalder proceduren firkant:

```
procedure firkant(x,y,d: integer);
begin
    with form1.canvas do
    begin
        moveto(x,y);
        lineto(x+d,y);
        lineto(x+d,y+d);
        lineto(x,y+d);
        lineto(x,y);
    end;
end;
```



```
procedure TForm1.Button2Click(Sender: TObject);
var
    x,y,l: integer;
begin
    x:=20; y:=20; l:=200;
    while l>=100 do
    begin
        firkant(x,y,l);
        l:=l-40;
    end;
end;
```

Bemærk rækkefølgen af procedurerne. Den nederste procedure skal "bruge" proceduren firkant, derfor skal proceduren firkant stå først i kildeteksten til programmet.

Den første linje i en procedure kaldes procedurehovedet. I den står procedurens navn, og der kan være en liste med *formelle parametre*:

```
procedure firkant(x,y,d: integer);
```

I dette tilfælde er der tre formelle parametre, nemlig de tre integer-variable x, y og d.

Når proceduren firkant kaldes, sker det med en programlinje som denne

```
firkant(x,y,l);
```

hvor x, y og l kaldes *de aktuelle parametre*. Første gang proceduren firkant kaldes, har de tre aktuelle parametre værdierne 20, 20 og 200. Ved de efterfølgende kald er værdien af l mindsket med 40.

**Opgave 1** Forklar, at løkkestrukturen i programmet svarer til, at der udføres følgende procedurekald:

```
firkant(20,20,200);  
firkant(20,20,160);  
firkant(20,20,120);  
firkant(20,20,80);  
firkant(20,20,40);
```

Ved et procedurekald som `firkant(20,20,200)` bliver værdien 20 overført til parameteren x i proceduren `firkant`, værdien 20 bliver overført til parameteren y, og værdien 200 bliver overført til parameteren d. Herefter udføres programlinjerne i proceduren med disse værdier for x, y og d.

Når proceduren er afsluttet (fordi alle linjer i proceduren er gennemløbet), vender programmet tilbage til den klik-procedure, hvorfra `firkant` blev kaldt.

**Opgave 2** Følg med i processen ved brug af watch-vindue og Trace Into-knappen.

## Iteration og rekursion

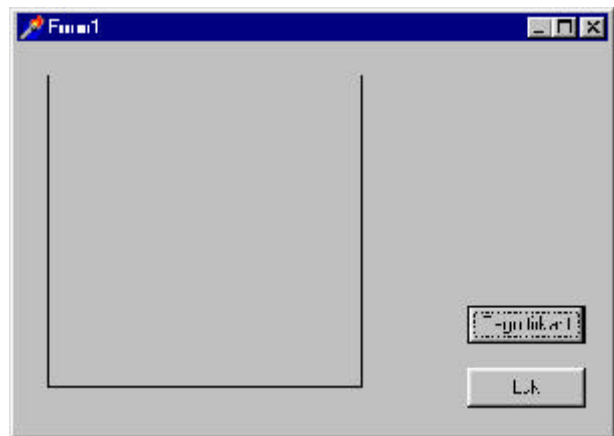
Klik-proceduren og `firkant`-proceduren på forrige side er eksempler på *iterative* procedurer, dvs. procedurer, der gennemføres ved at udføre programlinjerne i rækkefølge, evt. med brug af gentagelser og betingelser.

Der findes også en anden mulighed, nemlig *rekursive* procedurer. Herved forstår vi procedurer, der "kalder sig selv". I forbindelse med funktioner kommer vi ind på fordele og ulemper ved brug af rekursion.

### Eksempel 3, Iteration

Vi ser på `firkant`-proceduren igen, men ændrer i klik-proceduren, så der kun tegnes én firkant.

```
procedure firkant(x,y,d: integer);  
begin  
  with form1.canvas do  
    begin  
      moveto(x,y);  
      lineto(x+d,y);  
      lineto(x+d,y+d);  
      lineto(x,y+d);  
      lineto(x,y);  
    end;  
end;
```

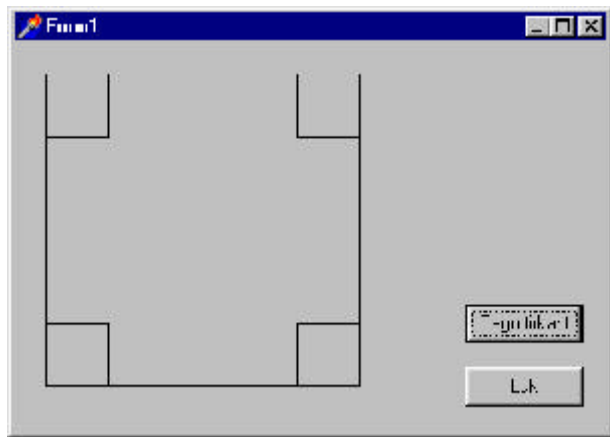


```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
  firkant(20,20,200);  
end;
```

De to procedurer er *iterative*. Alle programlinjer gennemløbes i rækkefølge, og hver linje kun én gang. Hvis man vil tegne flere firkanter indeni hinanden, fx således at der kommer "små" firkanter i hjørnerne på den store firkant, må man tilføje ekstra kald af proceduren `firkant` i klik-proceduren, fx

```
firkant(20,20,40);      {øverste venstre hjørne}  
firkant(180,20,40);    {øverste højre hjørne}  
firkant(180,180,40);   {nederste højre hjørne}  
firkant(20,180,40);    {nederste venstre hjørne}
```

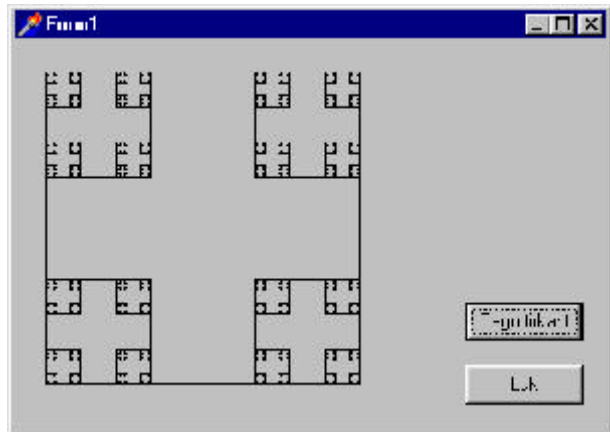
Hvis man også vil have tegnet små firkanter i hjørnerne af de 4 firkanter, skal man tilføje yderligere 16 linjer med kald af proceduren firkant. Det kan selvfølgelig lade sig gøre, men der findes en fikseret metode, nemlig brugen af *rekursion*.



#### Eksempel 4, Rekursion

Vi tilføjer 4 linjer med kald af firkant-proceduren, dvs. at *proceduren kalder sig selv*.

```
procedure firkant(x,y,d: integer);
var d2,d3: integer;
begin
  with form1.canvas do
  begin
    moveto(x,y);
    lineto(x+d,y);
    lineto(x+d,y+d);
    lineto(x,y+d);
    lineto(x,y);
    d2:=round(2*d/3);
    d3:=round(d/3);
    if d>4 then
    begin
      firkant(x,y,d3);
      firkant(x+d2,y,d3);
      firkant(x+d2,y+d2,d3);
      firkant(x,y+d2,d3);
    end;
  end;
end;
```



```
procedure TForm1.Button2Click(Sender: TObject);
begin
  firkant(20,20,200);
end;
```

De to variable  $d2$  og  $d3$  er tilføjet for at kunne nøjes med at udregne koordinater én gang. Det sparer lidt tid. Regneoperationen  $d/3$  og  $2*d/3$  giver som resultat reelle tal. De bliver konverteret til hele tal ved brug af funktionen `round`,  $d3:=\text{round}(d/3)$ .

De fire linjer med kald af proceduren firkant svarer til dem, vi satte ind i den iterative procedure i eksempel 3. Men det ser ud til, at spøgen med små firkanter i hjørnerne gentager sig. Hvordan går det til?

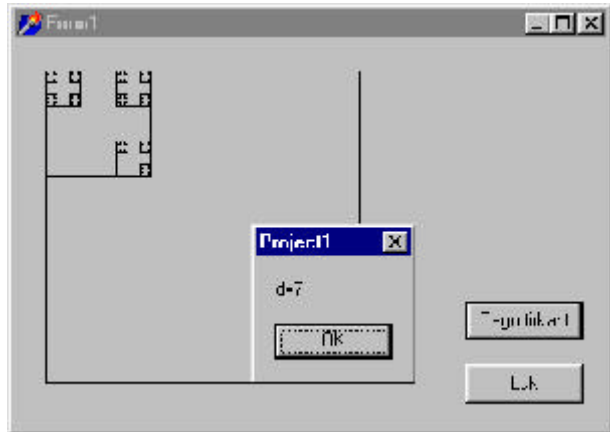
Forklaringen er, at når firkant-proceduren første gang kommer til linjen med `firkant(x,y,d3)`, så kaldes firkant-proceduren med de nye (mindre) kantlængder, og så bliver de tegnet, inden resten af den oprindelige firkant bliver gjort færdig. Systemet er så at sige, at "øverste venstre hjørne tegnes færdig først", så øverste højre hjørne, osv.

Det kan man se ved at indføje en "sladre-meddelelse" i form af en ShowMessage-boks inde i firkant-proceduren:

```
if d>20 then Showmessage('d='+inttostr(d));
```

Med  $d>20$  skal man kun svare OK 16 gange. Med  $d>6$  skal man svare OK 64 gange. Det betaler sig at begynde med at sammenligne  $d$  med et stort tal, så kan man altid nærme sig  $d>6$  eller noget i den retning...

Her er brugt  $d>6$ . Hver gang de fire firkanter er tegnet færdig, undersøger Delphi, om  $d>6$ , og hvis det er tilfældet udskrives meldingen om  $d$ -værdien.



## Funktioner

Vi kender bl.a. nogle funktioner, der konverterer text til integer eller integer til tekst, fx

```
Edit2.Text:=IntToStr(beløb)
```

og

```
beløb:=StrToInt(edit2.Text);
```

I eksempel 4 brugte vi også muligheden for at konvertere mellem real og integer,

```
d3:=round(d/3)
```

Fælles for funktionerne er, at de "arbejder på" en eller anden størrelse, og "leverer" en anden størrelse som resultat, nøjagtig som man i matematik kan bruge funktioner til at beregne fx  $\sin(47)$ .

## Eksempel 5

Man kan definere sine egne funktioner. Som eksempel definerer vi en funktion, der kan beregne Fibonacci-tal.

Fibonacci-tallene er den talrække der fremkommer, når man starter med 1 og 1, og derefter lader det næste tal være summen af de to forrige. Det giver følgende talrække:

1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 - 89 - 144 - 233 - 377 - 610 - 987 osv.

På de følgende sider skal vi se tre funktioner, der alle kan beregne det  $n$ 'te Fibonacci-tal.

## Eksempel 6 Iterativ funktion til beregning af fibonacci-tal

Beregnings-funktionen

```
function fibi(n: integer): longint;
var i,f,g,h: longint;
begin
  f:=1; g:=1;
  if n>2 then
  begin
    for i:=3 to n do
    begin
      h:=f+g; f:=g; g:=h;
    end;
    fibi:=g;
  end
else
  fibi:=1;
end;
```

Den procedure, der kalder beregningsfunktionen

```
procedure TForm1.Button1Click(Sender:
TObject);
var x: longint;
begin
  if edit1.text<>' ' then
  begin
    x:=strtoint(edit1.text);
    edit2.text:=inttostr(fibi(x));
  end;
end;
```

Hvis der er indtastet noget i edit1-boksen, omsættes tekststrengen til et tal (x). Det x'te fibonacci-tal beregnes af funktionen fibi(x), resultatet omsættes til en tekststreng, som skrives i edit2-boksen.

Det 4. fibonacci-tal findes ved at gennemføre følgende:

- 1) n har værdien 4
- 2) f:=1; g:=1 (det er værdierne af de to første fibonacci-tal)
- 3) i:=3
- 4) h:=f+g; f:=g; g:=h;      h:=1+1 (=2); f:=1; g:=2
- 5) i:=4
- 6) h:=f+g; f:=g; g:=h;      h:=1+2 (=3); f:=2; g:=3
- 7) nu er i=n, derfor ikke flere beregninger
- 8) Værdien af funktionen fås ved at sætte fibi:=g (=3)

n	4	4	4
i		3	4
h		2	3
f	1	1	2
g	1	2	3

Større fibonacci-tal findes ved at fortsætte med beregner som i pkt. 5) og 6). I skemaet herunder viser søjlen med n=7, n=10 osv. hvor langt man skal fortsætte for at beregne det 7. eller det 10. fibonacci-tal.

			n=4			n=7			n=10			n=13
i		3	4	5	6	7	8	9	10	11	12	13
h		2	3	5	8	13	21	34	55	89	144	233
f	1	1	2	3	5	8	13	21	34	55	89	144
g	1	2	3	5	8	13	21	34	55	89	144	233

## Eksempel 7 Rekursiv funktion til beregning af fibonacci-tal

Beregnings-funktionen

```
function fibr(n: integer): longint;
var res: longint;
begin
  if n>2 then
    res:=fibr(n-1)+fibr(n-2)
  else
    res:=1;
  fibr:=res;
end;
```

Den procedure, der kalder beregningsfunktionen

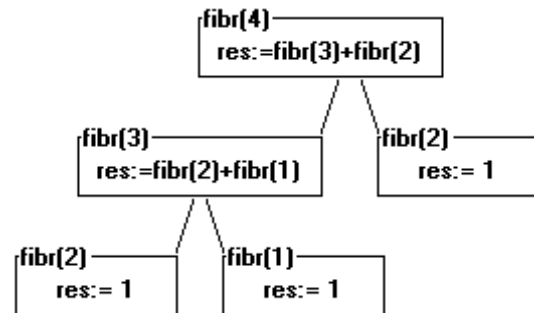
```
procedure TForm1.Button1Click(
  Sender: TObject);
var x: longint;
begin
  if edit1.text<>' ' then
  begin
    x:=strtoint(edit1.text);
    edit2.text:=inttostr(fibr(x));
  end;
end;
```

Hvis der er indtastet noget i edit1-boksen, omsættes tekststrengen til et tal (x). Det x'te fibonacci-tal beregnes af funktionen fibr(x), resultatet omsættes til en tekststreng, som skrives i edit2-boksen.

Det 4. fibonacci-tal findes ved at gennemføre følgende:

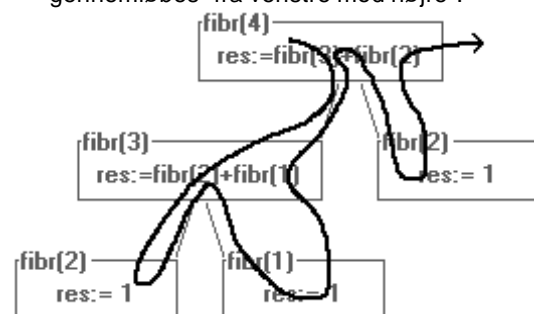
- 1) Resultatet er summen af de to foregående fibonacci-tal ( $fibr(n-1)+fibr(n-2)$ ), dvs.  $res:=fibr(3)+fibr(2)$
- 2) Så må fibr(3) findes. Vi starter forfra med "nye variable", og har  $n=3$ . Resultatet er igen summen af de to foregående fibonacci-tal, dvs.  $res:=fibr(2)+fibr(1)$ .
- 3) Så må fibr(2) findes. Vi starter forfra med "nye variable", og har  $n=2$ . Resultatet er 1.
- 4) Resultatet kan sættes ind i stedet for fibr(2) i beregningen af fibr(3), men vi mangler at finde fibr(1).
- 5) Så må fibr(1) findes. Vi starter forfra med "nye variable", og har  $n=1$ . Resultatet er 1.
- 6) Resultatet kan sættes ind i stedet for fibr(1) i beregningen af fibr(3). Alt i alt  $res:=1+1$ , som kan indsættes i stedet for fibr(3) i beregningen af fibr(4). Men i den beregning mangler vi stadig at finde fibr(2).
- 7) Så må fibr(2) findes. Vi starter forfra med "nye variable", og har  $n=2$ . Resultatet er 1.
- 8) Resultatet kan sættes ind i stedet for fibr(2) i beregningen af fibr(4). Alt i alt  $res:=2+1$ .
- 9) Resultatet?  $fibr(4)=3$ .

Bemærk, at vi i punkt 7) herover må beregne fibr(2) én gang til. Systemet kan altså ikke udnytte, at fibr(2) er beregnet én gang før (i punkt 3). Det er selvfølgelig med til at gøre denne metode langsommere end den iterative metode i eksempel 6.



Bemærkning:

- 1) Læg mærke til, hvordan "kasserne" er placeret i forhold til hinanden. Man siger, at de danner **et træ**. Det vender godt nok på hovedet, så **roden** er kassen med fibr(4), og de øvrige kasser udgør træets **grene**.
- 2) Læg også mærke til, hvordan træet gennemløbes "fra venstre mod højre".



### Eksempel 8 Rekursiv funktion (med hukommelse) til beregning af fibonacci-tal

Efter beregningen af fibr(4) så vi, at man kommer til at beregne den samme funktionsværdi flere gange, når man gennemløber "beregnings-træet".

I dette eksempel vil vi prøve at omgå dette problem ved at indbygge en slags hukommelse i den rekursive funktion.

Beregnings-funktionen

```
function fibrh(n: integer): longint;  
var res: longint;  
begin  
  if n>2 then  
    begin  
      if huk[n-2]=0 then huk[n-2]:=fibrh(n-2);  
      if huk[n-1]=0 then huk[n-1]:=fibrh(n-1);  
      res:=huk[n-1]+huk[n-2];  
    end  
  else  
    res:=1;  
  fibrh:=res;  
end;
```

Som hukommelse bruges et array, der naturligvis skal være erklæret på forhånd.

Array'et skal nulstilles ved programstart.

Den procedure, der kalder beregningsfunktionen

```
procedure TForm1.Button3Click(  
  Sender: TObject);  
var x: longint;  
begin  
  if edit1.text<>' ' then  
    begin  
      x:=strtoint(edit1.text);  
      edit4.text:=inttostr(fibrh(x));  
    end;  
end;
```

```
var  
  huk: array[1..100] of longint;
```

```
TForm1.OnActivate(Sender: TObject);  
var i: integer;  
begin  
  for i:=1 to antal do huk[i]:=0;  
end;
```

Det snedige ved denne rekursive procedure er, at hvert fibonacci-tal kun skal beregnes én gang. Så snart det er beregnet, bliver det placeret i hukommelsen, og derefter er det fibonacci-tallet i hukommelsen, der bliver brugt ved de efterfølgende kald af funktionen.

Omkostningen ved metoden er, at der skal afsættes plads til et array, og at man hele tiden skal spørge, om værdien nu er beregnet i forvejen. Men tidsmæssigt viser det sig, at der er en enorm gevinst at hente ved denne metode, især når man skal beregne store fibonacci-tal.

### 10 Sammenligning af de tre funktioner til beregning af fibonacci-tal

I programmet til højre kan man beregne fibonacci-tal ved de tre metoder, der er omtalt i eksemplerne 6-8.

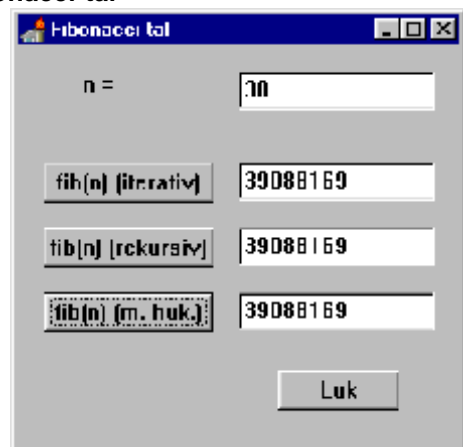
På en moderne Pentium-maskine ser man ikke, at det tager tid at få beregnet fibonacci-tal med den iterative funktion.

Med den rekursive funktion tager beregning af det 38. fibonacci-tal på en 500MHz PentiumIII-maskine ca. 3 sekunder.

Den rekursive funktion med hukommelse svarer tidsmæssigt til den iterative, man mærker ikke, at beregningen tager tid.

Man bør selvfølgelig tilføje en timer i programmet, så man kan måle den tid, beregningerne tager.

Når det er gjort, kan man efterfølgende undersøge sammenhængen mellem størrelsen af n og den tid beregningen tog, og lave en grafisk afbildning af tid som funktion af n. Den grafiske afbildning vil formodentlig vise en voldsom stigning i tidsforbruget ved den rekursive beregningsmetode, medens stigningen i tidsforbruget ved den iterative metode og den rekursive metode med hukommelse er moderat.



**Opgave 11** Vis teoretisk at beregningstiden vokser eksponentielt ved den rekursive metode.

### Eksempel 12 Tidsmåling i den rekursive fibonacci

I eksempel 7 så vi den rekursive beregningsfunktion og den klik-procedure, der kalder beregningsfunktionen. Her følger en ny udgave af den klik-procedure, der kalder beregningsfunktionen. Forskellen er, at den nye procedure har indbygget tidsmåling:

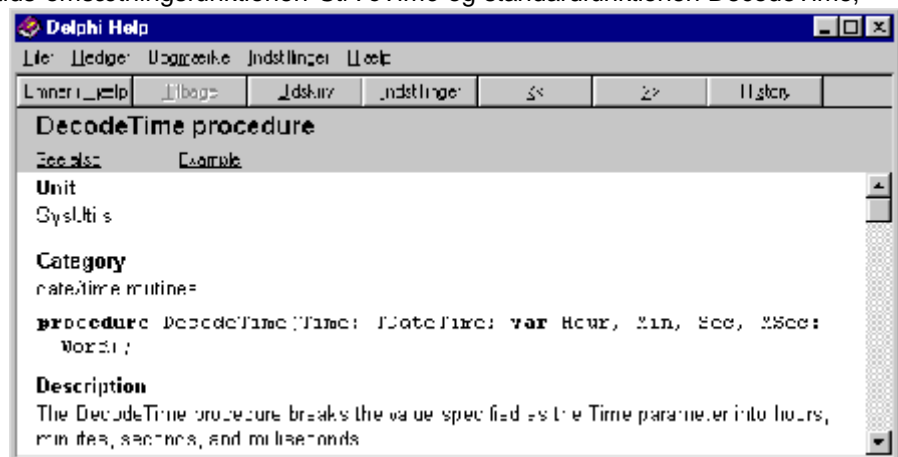
```
procedure TForm1.Button2Click(Sender: TObject);
var
  x, start_abs, slut_abs: longint;
  dtStart, starttid, sluttid : TDateTime;           {standard-tidsvariabel}
  Hour1, Min1, Sec1, MSec1, Hour2, Min2, Sec2, MSec2: Word; {hjælpevariable}

begin
  if edit1.text<>' ' then
  begin
    x:=strtoint(edit1.text);
    {tag tid og beregn}
    dtStart:= StrToTime('00:00:00');                {midnat}
    Starttid:= Now - dtStart;                       {starttid=tid fra midnat til nu}
    edit3.text:=inttostr(fibr(x));                  {kald fibonacci, skriv resultat}
    sluttid:= Now - dtStart;                        {sluttid=tid fra midnat til nu}
    {udskriv beregning og tider}
    DecodeTime( Starttid, Hour1, Min1, Sec1, MSec1);
    start_abs:=Msec1+1000*Sec1+1000*60*Min1+1000*3600*Hour1; {starttid i msec}
    edit8.text:=inttostr(start_abs);
    DecodeTime( Sluttid, Hour2, Min2, Sec2, MSec2 );
    slut_abs:=Msec2+1000*Sec2+1000*60*Min2+1000*3600*Hour2; {sluttid i msec}
    edit9.text:=inttostr(slut_abs);
    edit10.text:=inttostr(slut_abs-start_abs);      {regnetid i msec}
  end;
end;
```

Det nye er tidspunktet *Now*, tids-omsætningsfunktionen *StrToTime* og standardfunktionen *DecodeTime*, der opdeler et klokkeslæt i timer, minutter, sekunder og millisekunder.

Til højre ses hvad hjælpe-teksten skriver om standardfunktionen *DecodeTime*.

Fidusen er altså, at et klokkeslæt omsættes fra den form, som variabelen *Time* har (typen *TDateTime*), til timer, minutter, sekunder og millisekunder.



I vores klik-procedure indgår tre sådanne variable, *dtStart*, *starttid* og *sluttid*. *dtStart* får sin værdi ved omsætningen

```
dtStart:= StrToTime('00:00:00')
```

og *starttid* og *sluttid* ved at aflæse forskellen på det aktuelle tidspunkt (i standardvariabelen *Now*) og tidspunktet *dtStart*. Vi måler altså tiden fra midnat til lige før fibonacci-beregningen, og tiden fra midnat til lige efter fibonacci-beregningen.

Bagefter omsættes de to tider til timer, minutter, sekunder og millisekunder, og tiden fra midnat omregnes til et samlet antal millisekunder i linjerne

```
start_abs:=Msec1+1000*Sec1+1000*60*Min1+1000*3600*Hour1; {starttid i msec}
slut_abs:=Msec2+1000*Sec2+1000*60*Min2+1000*3600*Hour2; {sluttid i msec}
```

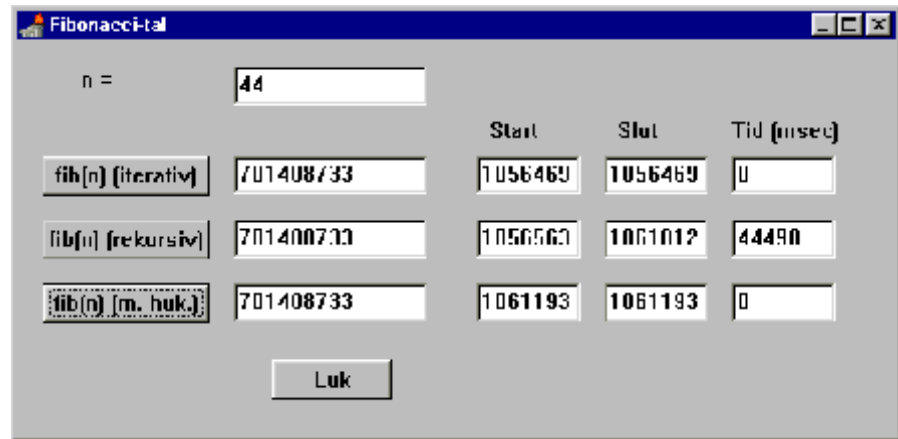
Herefter kan beregningstiden findes som forskellen på de to tider.

I stedet for heltal af typen *Integer* bruges heltal af typerne *Word* og *Longint*. Typen *Word*, fordi den indgår i DecodeTime-proceduren, typen *Longint* fordi *Integer* simpelt hen ikke "er store nok". Talområderne for de tre slags heltal er

```
Integer -32768..32767
Word 0..65535
Longint -2147483648..2147483647
```

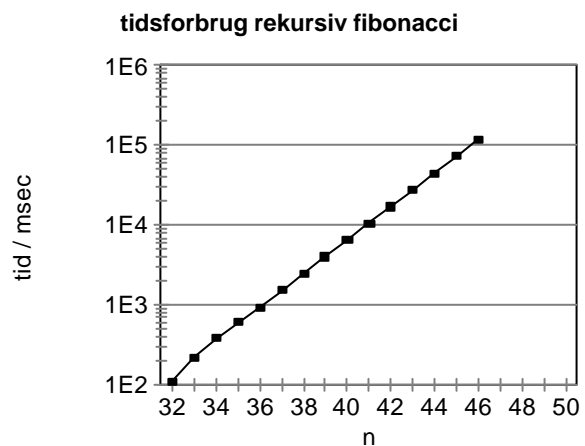
Man kan uden videre lade word-typer indgå på højresiden af beregninger, hvis resultat skal være af typen *Longint*. Se fx programlinjerne med beregningerne af start\_abs og slut\_abs.

Programmet kan fx præsenteres således:



Her følger resultatet af beregningerne for n=32 til n=45, udført på en 500MHz PentiumIII-maskine:

n	tid / msec	$t_n / t_{n-1}$
32	110	
33	220	2,000000
34	380	2,000000
35	600	1,727273
36	930	1,578947
37	1540	1,550000
38	2470	1,655914
39	4010	1,603896
40	6480	1,623482
41	10490	1,615960
42	16970	1,618827
43	27470	1,617731
44	44430	1,618739
45	72010	1,617401
46	116500	1,620752



**Opgave 13** Hvad viser grafen om beregningstiden ved den rekursive metode?