Kapitel 1

Algoritmer och Datastrukturer

1.1 Om kursen

I den här kursen ska vi studera *algoritmer* och i första hand de *datastrukturer* vi behöver för att implementera dessa algoritmer.

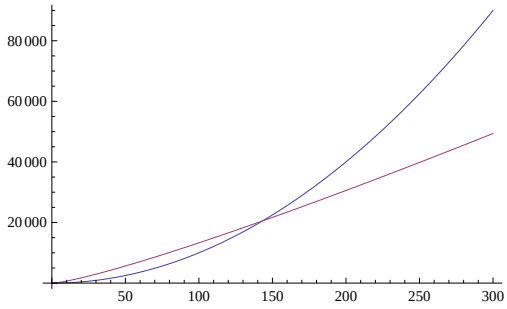
Ett problem (ofta optimeringsproblem) kan alltid lösas med hjälp av olika algoritmer. Några kan vara ineffektiva, kräver stora datorresurser, andra kan var effektiva. Skillnaden i ett programs effektivitet beror oftare på algoritmen än, skillnader i hårdvara och val av programspråk.

I kursen kommer vi att titta på flera effektiva algoritmer, som är kopplade till en viss typ av problem. Dessa algoritmer är ofta välkända och har ett namn och en konstruktör. Vi använder dem som ett recept ur en kokbok.

Dessutom kommer du att få konstruera egna algoritmer. Kravet på att dessa är effektiva är inte alltid så stort. Om problemet går att lösa i överskådlig tid är vi nöjda.

Om vi håller oss till sortering av n stycken heltal, kan *BubbelSort* fungera väl så länge n är relativt litet. Väljer vi i stället sorteringsalgoritmen *MergeSort*, kommer den att ta längre tid att exekvera än BubbelSort, för små värden på n. Men för ett visst värde n kommer kurvorna att skära varandra, eftersom den tid som krävs för att sortera n tal med BubbelSort är $c_1 \cdot n^2$ och den tid som krävs för att sortera samma tal med MergeSort är $c_2 \cdot n \log_2 n$. c_1 och c_2 är konstanter som inte beror av n och $c_1 < c_2$.

För värden på c_1 och c_2, tagna ur luften och där x-axeln visar n och y-axeln *tid* i en obestämd storhet får vi en graf där vi kan se att för $n > 150$, så är MergSort effektivare.



Det kan vara så att ett systems totala prestanda mer beror på valet av algoritmer än valet av hårdvara.

I takt med att man konstruerar snabbare och snabbare datorer blir algoritmerna också effektivare. Du kanske undrar om algoritmerna är lika viktiga, i ett system, som

- datorer med hög clock rate, pipelining och superscalar arkitektur
- lättanvända, intuitiva grafiska gränssnitt
- objektorienterade system
- lokala och globala nätverk

Svaret är ja! Även hos web-baserade applikationer, som till exempel Stockholms lokaltrafik, behöver man algoritmer för att

- finna *kortaste vägen* mellan två hållplatser
- rendera kartor
- tyda inskrivna, eventuellt felstavade, namn på hållplatser.

1.2 Algoritm

Vad är en algoritm?

> En **bra** algoritm är som en skarp kniv – den gör exakt vad den är konstruerad för att göra, med minsta möjliga ansträngning. Att försöka lösa ett problem med en **dålig** algoritm, är som att försöka tranchera en stek med mejsel: Du kommer kanske att få ett ätbart resultat, men kommer att behöva lägga ner betydligt mer kraft än nödvändigt och resultatet kommer troligtvis inte att bli lika estetiskt tilltalande.

En mer strikt definition

- **Indata.** Algoritmen tar emot indata
- **Utdata.** Algoritmen producerar ett resultat
- **Precision.** Varje steg är noggrant bestämt
- **Determinism.** Resultatet av varje steg är entydigt och bestäms endast av indata och resultatet från tidigare steg
- **Ändlig.** Algoritmen når sitt mål efter ett ändligt antal steg
- **Korrekt.** Algoritmen producerar ett korrekt resultat
- **Generell.** Algoritmen kan tillämpas på en given mängd indata

1.2.1 Ett enkelt exempel

Givet: Tre tal a, b och c.
Sökt: Det största av de tre talen

Algorithm 1.2.1: TRETALMAX(a,b,c)

```
local x
x ← a
if b > x
  then x ← b
if c > x
  then x ← c
return (x)
```

[1] Vi ser direkt att algoritmen TRETALMAX har tre tal, a, b, c, som *indata* och att dess *utdata* är ett tal x. Varje steg är klart definierat. Resultatet i varje steg är bestämt av indata och

[1] Vi kommer ibland att använda denna layout för att beskriva algoritmer

resultatet i tidigare steg. Algoritmen når sitt mål efter fyra steg. Vi tror att algoritmen är korrekt och vi förstår att den fungerar för alla uppsättningar av tre reella tal.

Nu över till ett mer komplicerat problem

1.2.2 Sudoku

		8		6				7
	3				5	4		
				2		1		
	9			8		2		
7								6
		1			3		5	
		4		9				
		6	1				8	
2				4		9		

Figur 1.1:

I figur 1.1 ser du en fluga på utdöende(?). I nästan alla dagstidningar finns fortfarande, varje dag, en *sudokuruta*, tänkt att lösa endast med en penna som hjälpmedel, men här ska du använda datorn!

Ett korrekt löst sudokuproblem har siffrorna `1...9` precis en gång i var och en av de 9 raderna, de 9 kolumnerna och de 9, 3×3 boxarna (begränsade av de något grövre strecken i figuren)

Skriv ett program som läser in problemet från filen `sudoku.txt`. Filen består av 9 rader med 9 tal ur mängden {`0...9`} på varje. `0` betecknar *tom ruta*. Programmet ska sedan ersätta alla `0`:or med tal efter reglerna ovan och skriva ut lösningen Ett körningsexempel:

```
1 2 8 3 6 4 5 9 7
6 3 9 7 1 5 4 2 8
5 4 7 8 2 9 1 6 3
3 9 5 4 8 6 2 7 1
7 8 2 9 5 1 3 4 6
4 6 1 2 7 3 8 5 9
8 1 4 5 9 7 6 3 2
9 5 6 1 3 2 7 8 4
2 7 3 6 4 8 9 1 5
```

Ett riktigt sudokuproblem har endast en lösning, detta gäller även detta problem och våra

testexempel. Just detta sudokuproblem är klassificerat som *svårt*, men när man använder en dator är alla problem lika lätta! – eller?

1.2.3 Lösning

Backtracking löser problemet med hjälp av följande algoritm. Sök upp första rutan som inte är ifylld. Testa med att fylla i rutan med talen `1...9` i tur och ordning. En aning knepigt är det att testa om ett tal finns i en viss *box*.

Algorithm 1.2.2: SUDOKU(void)

```
local fyllda
local problem[9][9]
LAESINDATA(problem, fyllda)
SOLVE(problem, fyllda)
```

Variabeln `fyllda` håller reda på hur många rutor som redan är ifyllda. Då senare, `fyllda` får värdet 81, är problemet löst.

Algorithm 1.2.3: SOLVE(problem, fyllda)

```
local rad, kol, siffra
if fyllda =81
  then SKRIVUT(problem)
  else { for rad ← 1 to 9
       { do { for kol ← 1 to 9
            { do { if problem[rad][kol] = 0
                 { then go to ut
ut:
for siffra ← 1 to 9
{ do { if TESTA(rad, kol, siffra, problem)
     { then { problem[rad][kol] ← k
            { SOLVE(problem, fyllda+1)
            { problem[rad][kol] ← 0
```

Om inte 81 rutor är fyllda börjar vi leta efter första tomma rutan. När vi funnit den hoppar vi ut ur dubbelloopen. `rad` och `kol` pekar nu ut den tomma rutan, som försöker förses med alla talen `1...9`. Allt beroende på funktionen `test`s resultat

Algorithm 1.2.4: TESTA(rad, kol, siffra, problem)

```
local r, k
for r ← 1 to 9
  do { if problem[r][kol] = siffra
       then return (0)
for k ← 1 to 9
  do { if problem[rad][k] = siffra
       then return (0)
for r ← 3 · (rad − 1)/3 to 3 · (rad − 1)/3 + 2
  do { for k ← 3 · (kol − 1)/3 to 3 · (kol − 1)/3 + 2
       do { if problem[r][k] = siffra
            then return (0)
return (1)
```

Här testas i tur och ordning aktuell kolumn, rad och box. Om inget av de 27 testade villkoren är sant returnerar vi 1 – siffran kan placeras här. Om den verkligen ska vara i rutan kommer att visa sig senare.

```
1  #include <stdio.h>
2
3  int testa(int rad,int kol,int tal,int tab[9][9]){
4    int i,j;
5    for(i=0;i<9;i++)
6      if(tab[rad][i]==tal)
7        return 0;
8    for(i=0;i<9;i++)
9      if(tab[i][kol]==tal)
10       return 0;
11   for(i=rad/3*3;i<=rad/3*3+2;i++)
12     for(j=kol/3*3;j<=kol/3*3+2;j++)
13       if(tab[i][j]==tal){
14         return 0;
15       }
16   return 1;
17 }
18
19 void solve(int n,int tab[9][9]){
20   int i,j,k;
21   if(n==81)
22     skrivut(tab);
23   else{
24     for(i=0;i<9;i++)
25       for(j=0;j<9;j++)
26         if(tab[i][j]==0)
27           goto ut;
28   ut:
29     for(k=1;k<=9;k++)
30       if(testa(i,j,k,tab)){
31         tab[i][j]=k;
32         solve(n+1,tab);
33         tab[i][j]=0;
34       }
35   }
36 }
37 int main(void){
38   FILE *fil;
39   int i,j,n=0,tab[9][9];
40   fil=fopen("sudoku3.txt","rt");
41   for(i=0;i<9;i++)
42     for(j=0;j<9;j++){
43       fscanf(fil,"%d",&tab[i][j]);
44       if(tab[i][j]!=0)
45         n++;
46     }
47   fclose(fil);
48   solve(n,tab);
49 }
```

Frågor man nu kan ställa sig om denna algoritm är

- Terminerar algoritmen?
- Producerar den korrekt resultat?
- Är den tillräckligt effektiv? Finns det en snabbare algoritm?

Rekursion

1.3 Inledning till rekursion

Vi inleder kursen med begreppet *rekursion*, en programmeringsteknik som kommer att användas flitigt under kursen. Det är därför viktigt att du tillägnar dig denna teknik inför fortsättningen. Från *Bra Böckers Lexikon* klipper vi följande:

> **Rekursiv:** återkommande, som kan upprepas (bl.a. om sjukdomar); inom matematik och datateknik, något som implicit upprepas på ny nivå.

Rekursivt formulerade lösningar är ofta eleganta och betraktas inte sällan som ”geniala”. Det är lite av programmerarens adelsmärke, att kunna använda sig av rekursion. Likväl kan rekursion, för dig som ser det för första gången, te sig både svår och mystisk.

Det *rekursiva sättet* att lösa problem, bygger på att reducera det ursprungliga problemet till ett antal delproblem, som har samma struktur som ursprungsproblemet men som är enklare att lösa. Resultatet från delproblemen kan sedan användas till lösandet av det ursprungliga problemet.

1.3.1 Ett första exempel

Vårt första exempel får bli en *rekursionsformel*, en formel där man kan bestämma till exempel, ett tal i en talföljd med hjälp av föregående tal.

Dessa formler är speciellt enkla att översätta till datorprogram. *Fibonacci-följden* är kanske den mest kända rekursionsformeln. Exemplet utgör en ”nära släkting”

Givet är följande formel:

$$h_n = h_{n-1} + h_{n-2} + 1$$
$$\text{och att } h_1 = h_2 = 1$$

Antag att man vill bestämma h_4. Allt vi kan säga i första steget, är att $h_4 = h_3 + h_2 + 1$, där vet vi att $h_2 = 1$. Därför återstår bara att bestämma h_3, som ju är $h_3 = h_2 + h_1 + 1$. h_4 kan nu skrivas $h_4 = h_2 + h_1 + 1 + h_2 + 1 = 1 + 1 + 1 + 1 + 1 = 5$.

Vi förstår att vi kan använda denna teknik för att bestämma h_n för vilket naturligt tal n som helst, bara vi har tillräckligt med tålamod.

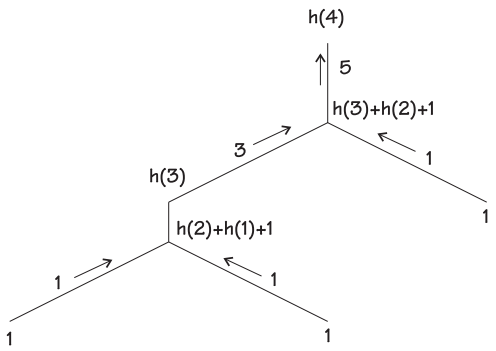
Återstår att visa hur en rekursiv funktion i C kan lösa detta problem. Indata är n för vilket vi vill bestämma h_n.

```
1 int h(int n){
2   if(n==1||n==2)
3         return 1;
4   else
5         return h(n-1)+h(n-2)+1;
6 }
```

Funktionen h består av en `if`-sats. Om $n = 1$ eller $n = 2$ är svaret omedelbart klart – det blir 1. I annat fall ska vi bestämma $h_{n-1} + h_{n-2} + 1$. Detta innebär att vi ska anropa funktionen h två gånger men med ett mindre n.

Det ovanliga eller det nya (för oss) är att funktionen *anropar sig själv*, vilket kännetecknar rekursiva funktioner. Detta kan leda till nya anrop som i sin tur kan leda till ytterligare anrop och så vidare ...

Vi följer anropet $h(4)$. Villkoret i `if`-satsen är inte sant, vilket leder till *två* nya anrop $h(3)$ och $h(2)$. Det första av dessa leder till två nya anrop $h(2)$ och $h(1)$. Båda dessa returnerar 1. Genom trädet i *figur* 1.10 ser vi hur funktionen till sist får värdet 5.



Figur 1.2: *Genom ett träd ser vi hur h(4) exekveras*

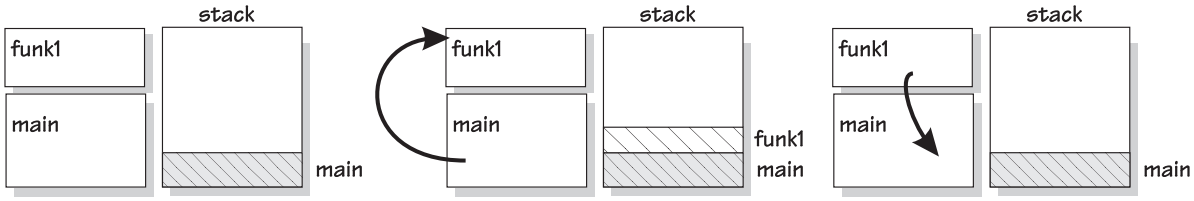
UPPGIFT 1.1
Några enkla frågor (0 poäng) Några frågor kring det inledande exemplet.

- Hur många gånger anropas funktionen $h(n)$ för $n = 4$
- Vilket värde har $h(5)$?
- Hur många anrop av $h(n)$ görs då $h(5)$, ska beräknas?

1.3.2 Detta händer vid funktionsanrop

Från tidigare känner vi till att varje gång en funktion anropas i C, så skapas utrymme på *stacken* där de *lokala variablerna*, *formella parametrarna* och *återhoppsadressen* lagras. När funktionen

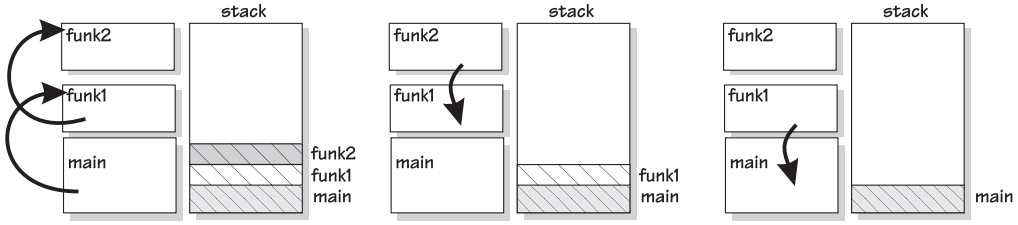
exekverats, försvinner dessa uppgifter från stacken och hopp sker till den funktion från vilket anropet skedde. Figurerna nedan vill visa detta.



Figur 1.3: *Ett vanligt anrop av en funktion*

För att vara riktigt tydliga tar vi ytterligare ett *icke rekursivt* exempel.

- På stacken finns från början de variabler som är deklarerade i *main*.
- När anrop av *funk1* görs från main allokeras alltså plats för de lokala variabler som finns i *funk1* på stacken.
- Dessutom finns adressen till den plats i *main* varifrån anropet skedde, återhoppsadressen.
- Någonstans från *funk1* görs nu ett anrop av *funk2*. På samma sätt bereds plats på stacken för lokala variabler i *funk2*
- Samt för återhoppsadressen, till någonstans i *funk1*
- När *funk2* har exekverats klart fortsätter exekveringen i *funk1*. På den plats som pekas ut av återhoppsadressen.
- Det utrymme på stacken som upptogs av lokala variabler i *funk2* under dess exekvering återlämnas nu.
- På samma sätt når så småningom *funk1* sitt slut – lokala variabler tas bort från stacken och återhopp sker till *main*.

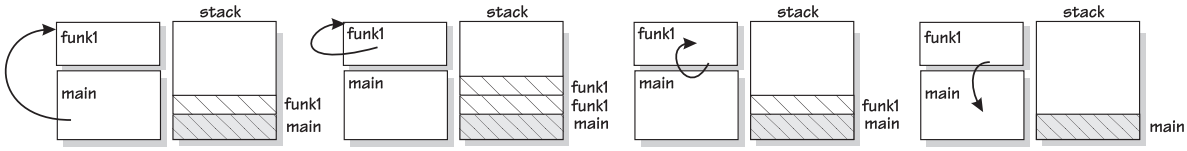


Figur 1.4: *Huvudfunktionen anropar en funktion som i sin tur anropar en funktion*

I figur 1.5, ser vi så till sist ett rekursivt anrop och förstår att det inte är någon principiell skillnad mellan detta och förra exemplet. Då återhopp sker rensas allt från just det anropet bort från stacken och exekveringen fortsätter vid återhoppsadressen.

1.3.3 Avbrott

Av detta och det inledande exemplet förstår vi hur viktigt det är att rekursionen har ett slut. Om det inte funnits en gren i if-satsen som sagt att då $n = 1$ eller $n = 2$, så är $h = 1$, ja då hade anropandet



Figur 1.5: *Ett rekursivt anrop: Huvudfunktionen anropar en funktion som anropar sig själv*

av nästa nivå pågått i all evighet (i själva verket bryts exekveringen av att stacken blir full).

Teoretiskt är det alltså möjligt att bestämma $h(n)$ för vilket naturligt tal n som helst med hjälp av vår funktion, men i praktiken kommer inte stacken att rymma all information som behövs för stora värden på n.

1.3.4 Effektivitet

Motsatsen till *rekursion* heter *iteration* – att upprepa samma rutin flera gånger till exempel genom en `while` eller `for`-loop. Skillnaden mellan dessa tekniker är bland annat att man hos iterationen slipper att betala för den administration som krävs vid funktionsanrop. Detta talar *för* iteration.

I övrigt kan man inte säga vilken teknik som är effektivast. Det hela beror på algoritmen som används. Det är en utbredd missuppfattning ”att rekursiva algoritmer är snabbare än iterativa”. Det kan vara och är oftast precis tvärt om! Om vi åter tittar på vårt inledande exempel, så har vi alltså från början $1, 1\ldots$. Nästa tal, det för $n = 3$ får vi genom att addera de två föregående och till detta lägga 1, alltså 3. Plötsligt blir det enkel huvudräkning för att få fram $1, 1, 3, 5, 9, 15, 25\ldots$. Denna iterativa algoritm är betydligt effektivare.

En annan aspekt på effektivitet är den tid och möda som krävs för att skriva koden. Det kan i vissa situationer gå snabbare att skriva programmet för en rekursiv algoritm än för en iterativ. Ofta blir detta kort och elegant. Men först måste man lära sig att tänka rekursivt – och det är inte alltid så lätt.

1.4 Några exempel med rekursion

1.4.1 Fakultet

Vi börjar med det mest traditionella av alla exemplen på rekursion. Funktionen *fact* bestämmer $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

```
1 int fact(int n){
2   if (n==0)
3         return 1;
4   else
5         return n*fact(n-1);
6 }
```

En tabell får visa hur resultatet växer fram. När vi nått ner till avbrottet *fact(0)* är allt uppradat och det är bara att multiplicera.

$$
\begin{gathered}
\texttt{fact}(5) \\
5 \cdot \texttt{fact}(4) \\
5 \cdot 4 \cdot \texttt{fact}(3) \\
5 \cdot 4 \cdot 3 \cdot \texttt{fact}(2) \\
5 \cdot 4 \cdot 3 \cdot 2 \cdot \texttt{fact}(1) \\
5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot \texttt{fact}(0) \\
5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\
120
\end{gathered}
$$

Att funktionen inte bryter förrän vid $n = 0$ beror på att den ska fungera även för $0! = 1$. Alternativt skulle vi kunna ha avslutat det hela genom denna sats och därmed tjänat in ett anrop

```
if (n==0 || n==1) return 1
```

Man ska nu inte ledas till att tro att detta är det enda sätt på vilket $n!$ kan beräknas, inte heller till att det är det mest effektiva sättet. Programuppgiften skulle mycket väl ha kunnat förekomma i början av C-kursen och då sett ut som nedan. Det ger samma resultat (förstås), som programmet ovan och är något effektivare.

```
1 int fact1(int n){
2   int i,p;
3   p=1;
4   for(i=1;i<=n;i++)
5         p*=i;
6   return p;
7 }
```

Algoritmerna (om man nu kan säga att det är två olika) är däremot jämförbara ur effektivitetssynpunkt. Lika många multiplikationer, additioner (subtraktioner) och jämförelser krävs för att bestämma $n!$.

Om vi bortser (vilket vi kan göra) från rekursionens extra kostnad så kan vi säga att algoritmerna är helt likvärdiga.

Skillnaden i effektivitet ligger i det vi nämnde ovan. För varje gång man anropar en funktion i C ska parametrar, lokala variabler och återhoppsadress lagras på stacken. När anropet är färdigexekverat ska motsvarande minne som dessa data tog upp friställas.

Det är storleken på n som avgör hur mycket arbete datorn ska utföra. Ökar vi n med 1 ökar antalet anrop av den rekursiva funktionen med ett och antalet varv i den iterativa rutinen med ett. Mängden arbete växer *linjärt* med n.

Den här typen av resonemang ska vi genomföra mer seriöst i kommande föreläsning.

UPPGIFT 1.2
Största värdet(1 poäng). Både *fact* och *fact1* har en och samma klara begränsning – vilket är det största värde på n för vilka det fungerar med den kompilator du använder?

1.4.2 Fibonacci

Den italienske matematikern *Leonardo från Pisa* eller *Fibonacci*, som han oftare kallas, betraktas som Europas förste matematiker. Han har bland annat givit sitt namn till talföljden 1, 1, 2, 3, 5, 8, 13, ...

– definierad som

$$f_n = f_{n-1} + f_{n-2}$$
$$\text{där } f_0 = 1 \text{ och } f_1 = 1$$

Ett problem i kapitel tre av Fibonaccit's *Liber abbaci* ledde till introduktionen av denna talföljd:

> En man lät placera ett par kaniner i en inhägnad trädgård. Hur många par av kaniner kan under ett år produceras, med utgång från detta par, om vi antar att varje par nedkommer med ett nytt par varje månad från och med att paret uppnått två månaders ålder?

Första och andra månaden finns bara det ursprungliga paret, $1, 1$. Månad tre och fyra föds ett nytt par. Vi har nu $1, 1, 2, 3$. Den femte månaden föds två par och månad sex tre par ... Eftersom alla kaninerna överlever kommer det att finnas $f_{11} = 144$ par med kaniner efter ett år

Funktionen nedan tar emot ett ordningsnummer n och bestämmer det n:te fibonaccitalet.

```
1 int fib(int n){
2   if (n==0 || n==1)
3         return 1;
4   else
5         return fib(n-1)+fib(n-2);
6 }
```

En iterativ metod. Med samma teknik som för vårt inledande exempel kan vi här använda en effektivare algoritm.

Algorithm 1.4.1: FIB2(n)

local $f_0 = 1, f_1 = 1, f_2$
for $I \leftarrow 2$ **to** n
do $\begin{cases} f_2 \leftarrow f_0 + f_1 \\ f_0 \leftarrow f_1 \\ f_1 \leftarrow f_2 \end{cases}$
return (f_1)

[2] Loopen kommer inte ens att exekveras n varv för att bestämma f_n. Ökar vi n med 1 kommer loopen att exekveras en extra gång. Arbetet växer linjärt med n.

Hur är det då med vår första funktion *fib* – hur många gånger kommer den att exekveras för talet f_n? Ett sätt att få reda på det utan att egentligen anstränga sig är att placera en räknare (globalt

[2] Vi inför här ett speciellt språk för att uttrycka algoritmer. Tanken är att språket ska vara lätt att förstå och friare, samtidigt som vi ska slippa se onödiga detaljer som motsvarande C-program skulle innehålla. Normalt kommer det att bli ganska enkelt att översätta algoritmen till ett fungerande program.

deklarerad) i funktionen. För $f_5 = 8$ anropas funktionen 15 gånger och för $f_{13} = 377753$ gånger. Av detta sluter vi oss till att funktionen för f_n anropas $2 \cdot f_n - 1$. Den här algoritmen tvingar datorn att utföra ett betydligt större arbete än dess iterativa kusin. Algoritmen *fib* arbetar linjärt med f_n.

UPPGIFT 1.3
Skillnad mellan talföljder(0 poäng) Vilken är skillnaden mellan Fibonaccis talföljd och den vi definierade i det inledande exemplet? Om man känner f_n vad blir då h_n?

En explicit formel. Det är inte slut här heller. Med hjälp av formeln

$$f_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{(n+1)} - \left(\frac{1-\sqrt{5}}{2}\right)^{(n+1)}}{\sqrt{5}}$$

kan man bestämma f_n i ett enda slag. Effektivare än så här kan det inte göras. Tidsåtgången är konstant, oberoende av storleken på n.

Här har vi sett tre olika sätt att bestämma fibonaccitalet f_n. Med början från det mest resurskrävande till det effektivaste Vi återkommer till Fibonacci och hans talföljd i samband med *dynamisk programmering* lite längre fram i kursen.

1.4.3 Gissa mitt tal

Vi återknyter till en programmeringsuppgift som förekom i grundkursen.

Vi önskar ett program, där datorn ”tänker på” ett heltal mellan 1 och 100. Programmet ska sedan själv (!) gissa det hemliga talet (utan att fuska), till skillnad från tidigare version där du skulle gissa.

Men precis som tidigare finns tre svar på gissningen RÄTT, FÖR STORT, FÖR LITET. Programmet fortsätter att gissa tills den får rätt svar.

När problemet gavs kom de flesta på följande algoritm:

Algorithm 1.4.2: GISSA(min, max, antal)

```
local x
x ← (min + max)/2
if x = hemlig
  then return (antal)
       { if x < hemlig
  else   { then GISSA(x + 1, max, antal + 1)
         { else GISSA(min, x − 1, antal + 1)
```

Alla gissningar görs med ett tal så nära mitten som möjligt i det möjliga intervallet.

Låter vi denna funktion verka på alla tal 1...100 så kommer det att i genomsnitt behövas 4.8 gissningar. Utökar vi intervallet till att omfatta 1...1000 blir medelvärdet 7.99. Vad vi visat här är inget annat än vad vi senare ska studera under rubriken *binärsökning*.

Den här algoritmen kan förstås mycket enkelt uttryckas iterativt. Men nu är det ju i första hand rekursion som gäller.

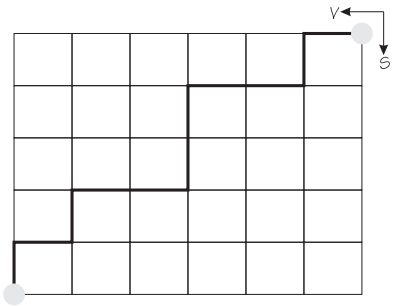
1.4.4 Att skriva ut ett tal

För att visa att nästan allt går att utföra rekursivt visar vi här en funktion som skriver ut ett tal `n` på bildskärmen.

```
1 void printnum(int n){
2   if(n<10)
3         printf("%d",n);
4   else {
5         printnum(n/10);
6         printf("%d",n%10);
7   }
8 }
```

Anropar vi funktionen genom `printnum(1234)`, så kommer den att ”bryta ned” talet n tills det blir < 10 och då skrivs detta n ut. Men vad händer sedan, eller rättare sagt vad har hänt innan?

- Eftersom det i 5 är fråga om en heltalsdivision $1234/10 = 123$ kommer `printnum` att anropas nästa gång genom `printnum(123)`
- Därefter sker anrop i tur och ordning med `printnum(12)` och `printnum(1)`.
- Genom det sista anropet skrivs talet 1 ut.
- Efter återhoppet är vi tillbaka till det anrop då $n = 12$ och sista satsen i funktionen skriver ut 2.
- Återhopp sker nu till den plats då $n = 123$ och $123 \mod 10 = 3$, som skrivs ut.
- Därefter kommer vi tillbaka till det första anropet. Det då $n = 1234$ och eftersom $1234 \mod 10 = 4$ så blir det 4 som till sist skrivs ut.



Figur 1.6: *Vi ska ta oss från övre högra hörnet till det nedre vänstra. Ett av de möjliga vägarna är markerad*

1.4.5 Hur många vägar?

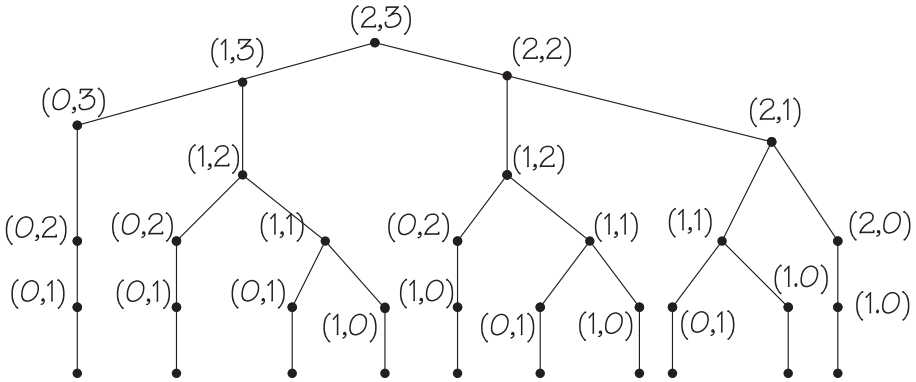
I *figur* 1.6 ser vi ett nätverk av gator. Vi tänker oss att vi startar längst upp i högra hörnet och vill ta oss ned till vänstra hörnet.

Vi får bara gå åt *väster* eller *söder*, men kan byta riktning i varje gatukorsning. Frågan är nu hur många olika promenader vi kan göra. I figur 1.6 ser du en av flera inritad.

Algorithm 1.4.3: ANTALVÄGAR(m, n)

```
local v = 0
if m = 0 and n = 0
  then return (1)
  else { if m > 0
           then v ← v + ANTALVÄGAR(m − 1, n)
       { if n > 0
           then v ← v + ANTALVÄGAR(m, n − 1)
       { return (v)
```

Indata är *kartans höjd* (m) och *kartans bredd*, (n) i kvarter räknat. Vi startar i (m, n) och är på väg till $(0, 0)$. Varje gång vi når målet har vi hittat en ny väg och returnerar 1. Antalet funna vägar v ackumuleras till det sökta antalet.



Figur 1.7: *Genom detta träd kan man följa exekveringen av* ANTALVÄGAR(2,3)

Anropet ANTALVÄGAR(2,3) kan åskådliggöras med ett träd som i figur 1.7. I varje nod ger vi koordinaterna för gatukorsningen.

Att bevisa att antalet vägar är $\binom{m+n}{n}$ överlämnar vi till matematiken. Även här har vi alltså skapat en algoritm som söker ett resultat på ett mer omständligt sätt än den explicita formel som matematiken ger oss. Nu är det inte alltid så – matematiken kan inte alltid ge så stor hjälp – det kommer vi att se längre fram i kursen.

UPPGIFT 1.4
Vilka vägar? (0 poäng) Utgå från kartan i figuren och svara på följande frågor.

- Vilken är den *första* vägen algoritmen hittar?
- Vilken är den *sista* vägen algoritmen hittar?
- Vilken är den *näst sista* vägen algoritmen hittar?
- Hur ska algoritmen ändras för att vägarna ska hittas i omvänd ordning?

UPPGIFT 1.5
Hur många anrop?(2 poäng) Implementera algoritmen i C och tar reda på hur många anrop den gör för olika värden på m och n

1.4.6 Euklides algoritm

Euklides (Euclid of Alexandria) är den mest prominenta av antikens matematiker. Han är känd för sin matematiska avhandling, Elementa, som näst Bibeln är den mest spridda skriften i Västerlandet. Eftersom hans böcker användes långt in på 1900-talet, så måste han vara alla tiders största matemtiklärare.

Han har bland annat givit namn åt en algoritm för att bestämma *största gemensamma delaren* till heltal (SGD(a,b), engelska gcd(a,b)).

Denna algoritm kallas följaktligen *Euklides algoritm* och ger till exempel $\text{SGD}(78, 21) = 3$ eller att $\text{SGD}(102, 31) = 1$. I $\text{SGD}(a, b) = c$, där $a \geq b > 0$, är c det största heltal som delar både a och b, $a \bmod c = 0$ och $b \bmod c = 0$, vilket inte är sant för något annat heltal $> c$.

Nedan följer ett exempel på hur algoritmen fungerar för att ta reda på *största gemensamma delaren* till $\text{SGD}(78, 21)$

$$\begin{aligned} 78 &= 3 \cdot 21 + 15 \\ 21 &= 1 \cdot 15 + 6 \\ 15 &= 2 \cdot 6 + 3 \\ 6 &= 2 \cdot 3 + 0 \end{aligned}$$

- 21 går i 78, 3 gånger med resten 15.
- 15 går i 21, 1 gång med resten 6.
- 6 går i 15, 2 gånger med resten 3
- 3 går i 6, 2 gånger, resten blir 0, och beräkningarna avbryts. *Den största gemensamma delaren* är den sista **resten** som inte är 0 – alltså 3.

Vi beskriver algoritmen rekursivt (och väljer namnet GCD, *greatest common divisor*) på följande sätt:

Algorithm 1.4.4: GCD1(a, b)

if $b = 0$
  then return (a)
  else return $(\text{GCD1}(b, a \bmod b))$

Den ser förföriskt enkel ut, speciellt om vi jämför den med den iterativa implementationen.

Algorithm 1.4.5: GCD2(a, b)

local $r = b, q$
while $r > 0$
do {
  $q \leftarrow \lfloor a/b \rfloor$
  $r \leftarrow a - q \cdot b$
  if $r = 0$
    then return (b)
    else { $a \leftarrow b$; $b \leftarrow r$ }
}

$y = \lfloor x \rfloor$ (engelska *floor*), y största heltal $\leq x$. $\lfloor a/b \rfloor$ är detsamma som heltalsdivision i C, så länge $a/b \geq 0$.

Algoritmen GCD3 är den man kanske själv skulle komma på. Hur står den sig i konkurrensen?

Algorithm 1.4.6: GCD3(a, b)

local i
for $i \leftarrow b$ **downto** 1
do {
  if $b \mod i = 0$ **and** $a \mod i = 0$
    then return (i)
}

Man kan inte säga något definitivt om det arbete som behöver utföras efter anropet GCD3(a, b). För GCD3$(120, 40) = 40$ når algoritmen sitt mål omedelbart, men efter
GCD3$(171, 170) = 1$ utför den 170 varv med 340 divisioner. Det är alltså inte storleken hos a och b som bestämmer arbetsåtgången, utan snarare dessa tals inbördes förhållande. Vi kan fastslå en övre gräns för GCD3, som motsvarar $2 \cdot b$ divisioner.

Inte vid något tillfälle kräver GCD1 eller GCD2 mer arbete än GCD3. Den övre gränsen för dessa algoritmers arbete är svårare att bestämma och vi återkommer därför till detta i en senare föreläsning.

Observera att vi här har tittat på *två* olika algoritmer. Den första, Euklides, har implementerats på två olika sätt.

UPPGIFT 1.6
Euler's funktion (2 poäng). Om SGD$(a, b) = 1$, säger vi att a och b är *relativt prima*. För Euler's ϕ-funktion är $\phi(n)$ antalet tal $1 \ldots n$ som är relativt prima med n. Skriv ett program som bestämmer $\phi(n)$ för önskat n.

1.5 Programmeringsuppgifter

UPPGIFT 1.7
Siffersumman (2 poäng). Om $t = 1765$ så är $ss = 19$ och om $t = 128956$ så är $ss = 31$, eller hur? Efter en stunds funderande hade du förstås kommit på att ss är *siffersumman* till t (summan av de i talet ingående siffrorna).

Skriv nu en rekursiv funktion SIFFERSUMMA(T) som bestämmer ss för ett heltal $t > 0$.

UPPGIFT 1.8
Binomialkoefficienten (2 poäng) Så här ser toppen av *Pascals triangel ut*

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & & 1 & & & & \\ & & & 1 & & 2 & & 1 & & & \\ & & 1 & & 3 & & 3 & & 1 & & \\ & 1 & & 4 & & 6 & & 4 & & 1 & \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \end{array}$$

Du kommer säkert ihåg hur man bildar nästa rad i triangeln. Om inte, så har du svaret här

$$\binom{n+1}{m} = \binom{n}{m} + \binom{n}{m-1}$$

Det vill säga, man får ett nytt tal med ordningsnummer m på rad $n+1$ genom att addera de två tal som står snett ovanför, till höger respektive vänster. Båda på rad n med ordningsnumren m och $m-1$. Varje rad inleds och avslutas med talet 1

Du ska nu skriva en rekursiv funktion *binomkoeff(n,m)* som beräknar *binomialkoefficienten* för givna n och m. Det är bra att känna till att

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \binom{n}{1} = n$$

Vad blir $\binom{13}{4}$ och hur många gånger anropar funktionen sig själv för att beräkna detta tal?

UPPGIFT 1.9
Myntmaskin (3 poäng) En maskin har en *display* på vilken den visar uppnådd poäng. Från början visar displayen 1 poäng. Genom att stoppa mynt i maskinen kan poängen förändras.

- Genom en 10-öring multipliceras poängen på displayen med 3
- Genom en 5-öring adderas 4 till poängen på displayen.

Skriv ett program som tar emot den poäng som ska uppnås. Programmet ska sedan beräkna och skriva ut det *lägsta* belopp som krävs för att uppnå poängen. Observera att man inte kan nå målet för alla slutpoäng.

En testkörning

```
Vilken poäng ska uppnås: 109
Poängen kan nås med 45 öre
```

UPPGIFT 1.10
MergeSort. (2 poäng) Antalet *jämförelser* i sorteringsalgoritmen MERGESORT kan beräknas med hjälp av följande rekursiva funktion

$$f(n) = f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + n - 1$$

där $f(1) = 0$. $\lceil a/b \rceil$ står för det minsta heltal c sådant att $c \geq a/b$. $\lfloor a/b \rfloor$ står för det största heltal c sådant att $c \leq a/b$.

Skriv ett program som för inmatat n bestämmer $f(n)$.

```
Hur många tal ska sorteras? 1000
För detta krävs 8977 jämförelser
```

UPPGIFT 1.11
Samma summa. (2 poäng) Följande rekursiva formel kan användas för att bestämma på hur många sätt talet n kan bildas som en summan av positiva heltal.

$$\text{part}(n, m) = \begin{cases} 1 & n = 1 \text{ och } m \geq 0 \text{ eller } m = 0 \text{ eller } m = 1 \\ 0 & m < 0 \\ \text{part}(n-1, m) + \text{part}(n, m-n) & \text{annars} \end{cases}$$

Testexempel $\text{part}(5, 5) = 7$,
$5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1$

UPPGIFT 1.12
Kvadratroten. (2 poäng) Med hjälp av följande rekursiva formel kan man bestämma roten ur $1 \leq n \leq 300$. Resultatet levereras i a, som ursprungligen har värdet $a = 1$. Talet $0.0000001 \leq e \leq 0.001$ anger med vilken noggrannhet resultatet ska beräknas

$$\text{ROT}(n, a, e) = \begin{cases} a & |a^2 - n| < e \\ \text{ROT}\left(n, \frac{a^2+n}{2a}, e\right) & \text{annars} \end{cases}$$

Testexempel $\text{ROT}(2, 1, 0.001) = 1.41421$. e garanterar att tre decimaler är korrekta.

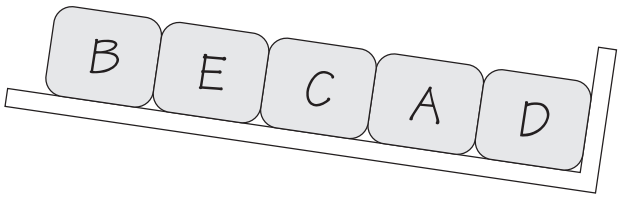
UPPGIFT 1.13
Bitstring. (2 poäng) Vi önskar här en rekursiv funktion, *BitStr*, som skriver ut alla 2^n ”binärsträngarna” med den givna längden n.

Funktionen anropas inledningsvis med till exempel `BitStr("",0,3)`. Det betyder att vi i utskriften vill ha $000, 001, 010, 011, 100, 101, 110, 111$, helst i den ordningen. Man kan till exempel använda `strcat`, med vars hjälp man kan slå samman två strängar.

Tips: Någonstans i funktionen kan följande kod finnas `BitStr(strcat(s,"0"),k+1,3)`, som alltså betyder att man lägger till en 0:a sist i strängen. Med $k+1$ noteras antalet tecken i strängen.

UPPGIFT 1.14

Roboten. (2 poäng) Med hjälp av en robot vill man sortera de fem paketen, som ligger på hyllan



Figur 1.8:

i figur 1.8, så att de ligger i bokstavsordning, från vänster till höger. Roboten klarar två olika funktioner.

- **b** Byta plats på de första två paketen, de längst till vänster
- **s** Flytta det sista paketet längst fram, paketet som ligger längst till höger, läggs längst till vänster.

Genom 'dragen' `bsssbsb` förvandlas ordningen BECAD till ABCDE. Skriv ett program som tar emot en sträng med de fem bokstäverna ABCDE i godtycklig ordning och som sedan bestämmer det minsta antalet steg roboten behöver utföra, för att ordningen ABCDE ska uppnås. Ett körningsexempel:

```
Hur ligger paketen ? BECAD
Det krävs 7 steg
```

UPPGIFT 1.15

Hissen i lustiga huset. (3 poäng) I *Lustiga Huset* finns $v, 1 \leq v \leq 100$ våningar. I den märkliga hissen finns bara två knappar. Dels en som förflyttar hissen u våningar *uppåt* och dels en som förflyttar hissen n våningar *nedåt*. Men med hjälp av en kombination av *resor*, *uppåt* och *nedåt* kan man ta sig till önskad våning m.

Du ska skriva ett program som tar emot uppgifter om v, u, n och m och som sedan beräknar *det minsta antalet resor* som behövs för att nå våning m, målet. En *resa* är alltså en knapptryckning som för hissen från en våning till en annan. Den första resan startar alltid på våning 1, som ligger i husets bottenplan. Huset saknar källare och vind, vilket betyder att hissen alltid måste befinna sig någonstans mellan våning 1 och v.

Indata: Programmet inleds med att fråga efter v, u, n och m

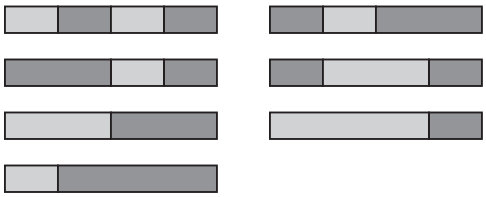
```
Hur många våningar har huset: 78
Förflyttning uppåt: 15
Förflyttning nedåt: 8
Till vilken våning ska du: 35
```

Utdata: En rad som talar om det minsta antalet resor som behövs för att nå målet:

```
Det behövs minst 13 resor för att nå målet
```

Endast testexempel där det finns en lösning kommer att användas.

UPPGIFT 1.16
Plankan. (2 poäng)



Figur 1.9:

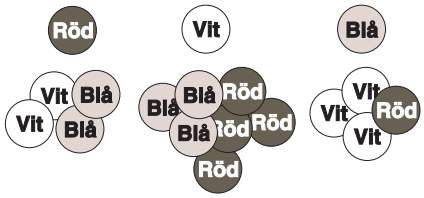
Man vill skapa en längre *planka* med hjälp av ett antal mindre *brädor*. Det finns tre olika typer av brädor, som har längden $1, 2$ respektive 3 meter. Det finns ett obegränsat antal av varje typ.

Skriv ett program som bestämmer på hur många olika sätt man kan åstadkomma en planka av längden $n, 1 \leq n \leq 24$.

I figuren ser du de 7 olika sätten att skapa en planka med längden 4 meter. Dialogen från en programkörning:

```
Plankans längd: 4
Det finns 7 olika sätt
```

UPPGIFT 1.17
Lek med kulor (3 poäng)



Figur 1.10: *En **röd** kan växlas till två **vita** och två **blå**. En **vit** kan växlas till tre **blå** och fyra **röda**. En **blå** kan växlas till tre **vita** och en **röd**.*

En lek med kulor för barn som kan räkna, står det på kartongen, som innehåller tillräckligt med kulor i tre olika färger: *blå, vita* och *röda.*

Leken går ut på att till slut ha *lika många kulor av varje färg*. Hur många kulor man har från början varierar från omgång till omgång. Genom att lämna en kula till ”banken”, kan de växlas mot ett antal kulor av de två andra färgerna enligt följande:

En **blå** kan växlas mot	3 **vita**	1 **röd**
En **vit** kan växlas mot	3 **blå**	4 **röda**
En **röd** kan växlas mot	2 **vita**	2 **blå**

Man får högst göra 15 växlingar för att nå målet och man ska nå det med så få växlingar som möjligt. Skriv ett program som frågar efter antalet kulor i de olika färgerna från start och som bestämmer det minsta antalet växlingar som behövs för att till slut *få lika många kulor av varje färg*.

Indata: Programmet frågar efter antalet kulor i de tre färgerna.

```
Hur många blåa? 22
Hur många vita? 22
Hur många röda? 13
```

Utdata: En rad, som talar om det minsta antalet växlingar som behöver göras för att målet

```
Det behövs minst 6 växlingar för att nå målet
```

Endast testexempel där det finns en lösning kommer att användas.

UPPGIFT 1.18
Norska vargar. (3 poäng) [3] När den norska regeringen väl bestämt sig för att minska vargstammen, beslöt man att 42 var det antal vargar man ville stanna vid efter avskjutningen.

Jägarna fick nu följande regler att rätta sig efter, för att i olika etapper, från de ursprungliga n vargarna försöka nå fram till de 42 önskade.

- Om n är ett jämnt tal får i denna etapp n halveras.
- Om n är delbart med 3 och/eller 4, så får, i denna etapp, lika många vargar som produkten av de två sista siffrorna i n visar, skjutas bort.
- Om n är delbart med 5 får, om det är möjligt, 42 vargar skjutas bort i denna etapp.

Skriv ett program som tar emot ett tal $42 < n < 10000$, som anger hur många vargar det finns just nu, och som tar reda på om det är möjligt att nå det önskade antalet 42.

Ett exempel förklarar. Vi börjar med 250 vargar

- 250 är delbart med 5 och > 42 så vi kan skjuta bort 42 vargar och få 208 kvar.
- 208 är ett jämnt tal så vi kan skjuta bort hälften och får 104 kvar.
- 104 är ett jämnt tal och vi kan åter skjuta bort hälften. Återstår nu 52 vargar.
- 52 är delbart med 4 och då kan vi skjuta bort $5 \cdot 2 = 10$ vargar och vi har nått fram till 42!

De olika reglerna kan alltså användas i vilken ordning som helst. Observera dock att i exemplet ovan är visserligen 208 delbart med 4, men regeln kan inte tillämpas eftersom $0 \cdot 8 = 0$. Två körningsexempel:

```
Hur många vargar finns nu?  7762
Målet kan INTE nås

Hur många vargar finns nu?  7461
Målet kan nås
```

[3] Denna uppgift har tidigare använts i kursen *Algoritmer og Datastrukturer* vid Universitet i Oslo men då handlade den om teddybjörnar.