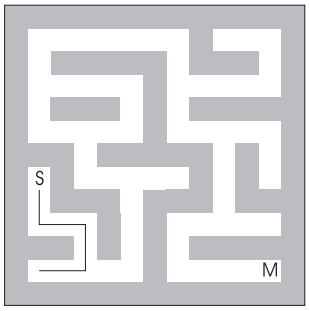
Kapitel 8

Backtracking



Figur 8.1: *Även om man går lite fel i början kommer man ändå fram till slut,*

8.1 Att hitta rätt i labyrinten

Vi inleder kapitlet om *backtracking* med ett exempel där man ska ta sig genom en labyrint. I figur 8.1 är utmärkt var starten går och var målet finns. Det är inte alls svårt att med blotta ögat finna den rätta vägen, men hur ska vi kunna ”lära” en dator att söka sig fram i en labyrint?

Eftersom labyrinten i detta exempel är ganska liten borde det inte ta så lång stund för ett program att hitta fram genom en slumpmässig vandring. Denna metod räknar vi inte som ett alternativ, speciellt inte då labyrinten kan vara betydligt större.

På något sätt måste vi beskriva labyrinten för datorn. En matris 13×13 fungerar bra här. Vi beskriver väggarna med ett värde och golvet med ett annat.

En systematisk promenad där programmet hela tiden håller reda på var man varit, så att inga försök upprepas beskrivs här

Algorithm 8.1.1: LUBBARUNTEN(labyrint, arad, akol, nr)

```
labyrint[arad][akol] ← vägg
if arad = mrad and akol = mkol
  then Framme efter nr steg
  else { if Ledigt uppåt
           then LUBBARUNTEN(labyrint, arad − 1, akol, nr + 1)
         if Ledigt nedåt
           then LUBBARUNTEN(labyrint, arad + 1, akol, nr + 1)
         if Ledigt vänster
           then LUBBARUNTEN(labyrint, arad, akol − 1, nr + 1)
         if Ledigt höger
           then LUBBARUNTEN(labyrint, arad, akol + 1, nr + 1)
labyrint[arad][akol] ← golv
```

[1] Indata är matrisen `labyrint`, som håller reda på labyrintens utseende. `vägg` och `golv` är de två möjliga värdena. `arad` och `akol` håller reda på var vi befinner oss. Dessa variabler får ett värde vid starten som sedan uppdateras för varje steg. `nr` håller reda på hur många steg vi tagit.

När steget är taget murar vi igen bakom oss genom att placera en `vägg` i labyrinten. `mrad` och `mkol` håller reda på var målet är beläget. Om dessa är lika med motsvarande `arad` och `akol` betyder det att vi nått målet och kan skiva ut hur många steg som behövdes genom `nr`.

I annat fall är det dags att ta ett steg till. Med hjälp av fyra if-satser tittar vi oss om i de fyra riktningarna. Så fort vi finner ett steg som är möjligt att ta, så tar vi det. Vi anropar alltså `LubbaRunten` med delvis nya parametrar.

Denna rekursiva process fortgår tills vi, *antingen når målet* eller *inte kan ta något mer steg*. I det senare fallet betyder det att ingen av de fyra villkoren är sant – det blir inget anrop från denna nivå och `LubbaRunten` kommer till sista satsen, vilken innebär att man tar bort `vägg` och placerar ett `golv` här igen. (Det är bara att hoppas att murbruket inte hunnit torka).

Men när vi lämnar en nivå så kommer vi till anropet strax innan. Här finns kanske fler villkor i if-satserna att testa och skulle något av dem vara sant så sticker vi iväg åt det hållet istället. Finns det en lösning så kommer vi att hitta den.

Hur många gånger kommer villkoret ”framme” att bli sant för vårt exempel på labyrint? Jo, två gånger det finns två olika vägar till målet. Den ena – snabbaste – kräver 46 steg. Den andra 4 steg ytterligare – 50.

Nu är det kanske inte så spännande att bara få reda på hur många steg som behövs för att

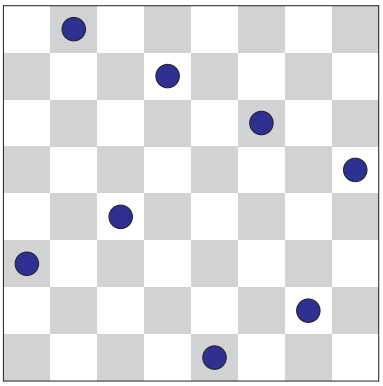
[1] Algoritmens namn kommer från den kända gåtan: Vad gjorde *laboranten* i *labyrinten*? – Jo, han *lubbarunt'en*

nå målet. Genom att lagra `arad` och `akol` i en lista, med `nr` som index, kommer vi att ha hela vandringen bokförd när vi når målet. Ännu trevligare är förstås att visa vägen grafiskt.

8.2 Åtta damer på schackbrädet

Att placera ut åtta damer på ett schackbräde så att inga damer står i slag för varandra är ett klassiskt problem som till och med Gauss – matematikens konung – roade sig med.

I figur 8.4 ser Du en av flera möjligheter att åstadkomma detta. En dam står i slag för en annan om de båda står på samma rad, kolumn eller diagonal.



Figur 8.2: *I de vackra damernas frånvaro får de svarta markerna träda in.*

Vårt mål är nu att skriva ett program som söker upp resten av lösningarna. Vi ska använda oss av backtracking igen. Vi placerar ut en dam i taget på brädet tills vi finner en lösning eller tills det inte finns någon ny plats. När vi funnit en lösning (och noterat det) eller ”kört fast” börjar vi plocka bort damer och placerar i stället in dem på andra platser.

Så här ser en typisk algoritm för backtracking ut.

Algorithm 8.2.1: ADDQUEEN(rad)

for kolumn ← 1 **to** 8
do
- **if** kolumn möjlig
- **then**
  - boka platsen
  - **if** rad = 8
    - **then** SKRIVUT( )
    - **else** ADDQUEEN(rad+1)
  - avboka platsen

Vad som är typiskt är att man, i en lista, markerar vad som utförts. Blockerar eller noterar för att sedan går vidare, egentligen med ett nytt, mindre, problem, med färre damer. När man så småningom kommer tillbaka efter det rekursiva anropet återställer man, tar bort blockeringen eller noteringen.

Det enda egentliga problem som återstår är att bestämma sig för hur bokningen ska ske, vilken datastruktur man vill ha. Det första man kommer att tänka på för detta speciella problem är en 8×8 matris där man sätter in, till exempel 1:or, för att markera damer.

Arbetet med att ta reda på om en utvald plats är möjlig blir då att ”titta” efter damer i alla 8 riktningarna från aktuell ruta. Vi ska dock tänka lite till för att slippa en mängd dubbla for-loopar.

Det kommer att hamna precis en dam i varje rad och en i varje kolumn. Vår algoritm stegar fram en rad i taget och placerar vid varje anrop in en dam (om det nu går) på en rad som tidigare var utan. När programmet lyckats placera en dam på rad 8 vet vi därför att vi funnit en lösning.

För att veta i vilken kolumn, damen på en viss rad, hamnar behöver vi en array med 8 celler – en för varje rad. Vi kallar denna array `kolumn` och `kolumn[i]` anger då i vilken kolumn damen på `i`:te raden finns.

En array, `kolfri` får hålla reda på om en viss kolumn är ledig. `kolfri[3]=1` innebär att det inte finns någon dam i kolumn 3, men `kolfri[6]=0` talar om att det redan finns en dam i kolumn 6.

För att ange en viss ruta behöver vi två index, *radindex* och *kolumnindex*.

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)
(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)

Först ska vi skilja på två olika typer av diagonaler. Det finns dels 15 diagonaler som börjar längst till vänster eller på sista raden med riktning nordost. På samma sätt finns det nordvästliga diagonaler.

Gemensamt för de nordostliga är att summan `rad + kol` är konstant inom samma diagonal! Två rutor med samma summa ligger på samma diagonal. Gemensamt för de nordvästliga är differensen `rad-kol` är konstant, även om den kan vara negativ. Detta ska vi utnyttja!

Vi inrättar efter det ytterligare två *boolska* arrayer, `uppfri` och `nerfri`. Alla celler med värdet 1, ledig, från start. Vi kan nu skriva det villkor, som måste vara sant för att rutan `[rad][kol]` ska vara ledig för en dam.

```
kolfri[kol] && uppfri[rad+kol] && nerfri[rad-kol+8]
```

”Platsbokningen” består förstås sedan i att uppdatera de fyra arrayerna (även `kolumn`).

UPPGIFT 8.1
Utskriftsrutin till damerna. Programmet `damer.c` saknar funktionen `SkrivUt`. Skriv den funktionen och passa samtidigt på att låta räkna antalet olika lösningar till problemet.

8.2.1 Hur bra är den här algoritmen?

En annan metod, än den vi här använder, är att placera ut de 8 damerna på brädet och sedan ta reda på om placeringen innebar en lösning. Från kombinatoriken kan vi härleda följande uttryck

$$\binom{64}{8} = 4\ 426\ 165\ 368$$

Inte bra! Lite bättre blir det om vi bestämmer oss för att endast ha en dam i varje rad.

$$8^8 = 16\ 777\ 216$$

Vår algoritm ligger snarare i närheten av $8! = 40\ 320$, vilket är en munsbit för våra snabba datorer.

UPPGIFT 8.2
Komplexitet för damproblemet. Även den senaste uppskattningen är en överskattning. Hur många anrop av funktionen `AddQueen` görs egentligen? Om man vill skaffa sig en uppfattning om $T(n)$, så måste man lösa problemet för olika brädstorlekar. Ta reda på om $T(n)$ ligger nära $n!$ genom att anpassa programmet till olika värden för n och bestämma antalet anrop av `AddQueen` för $1 \leq n \leq 12$. Använd sedan `Fit` i Maple för att bestämma en uppskattning av $T(n)$

Vi avslutar detta avsnitt med ett program jag funnit på internet

```
1  #include <stdio.h>
2  int count;
3
4  void try(int row, int ld, int rd){
5    if (row == 0xFF)
6      count++;
7    else{
8      int poss = 0xFF & ~(row | ld | rd);
9      while (poss){
10       int p = poss& -poss;
11       poss = poss -p;
12       try(row+p, (ld+p)<<1, (rd+p)>>1);
13     }
14   }
15 }
16 int main(int argc, char *argv[]){
17   printf("Eight Queens\n");
18   count = 0;
19   try(0,0,0);
20   printf("Number of solutions is %d\n", count);
21   exit(0);
22 }
```

UPPGIFT 8.3
Åtta damer på brädet. Förklara detta program för mig och hela klassen, så att vi alla begriper hur det fungerar.

8.3 Solitär

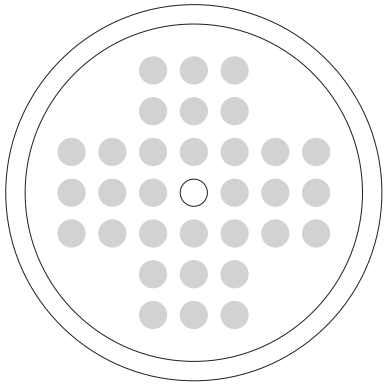
Lika känt som damerna på schackbrädet är spelet *Solitär*. Av namnet förstår vi att det är frågan om ett spel för *en* person – ett slags pussel.

I figur 8.3 ser vi 32 kulor i ett kors med platsen i mitten tom. Kulor kan hoppa ortogonalt, (alltså ej diagonalt) över en annan kula, under förutsättning att det är tomt direkt bakom. Den överhoppade kulan tas bort. Efter 31 drag ska bara en kula finnas kvar.

Inledningsvis finns fyra drag, som på grund av symmetrin kan betraktas som likvärdiga.

Vi önskar nu ett program som kan ge oss åtminstone en lösning. Finns det en lösning har vi anledning att tro att det finns många, på grund av så kallade ”dragomkastningar”.

Algoritmen, i bästa *backtracking*-stil



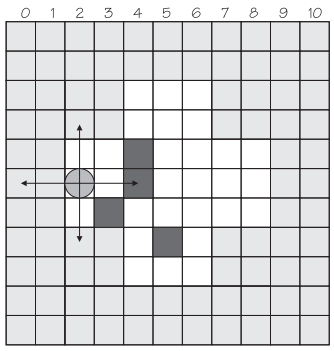
Figur 8.3: *Många av er har säkert sett detta soltärspel och kanske försökt lösa problemet. Här ska vi nu finna en lösning med hjälp av datorn*

Algorithm 8.3.1: NÄSTADRAG(nr)

```
if nr = 31
  then LÖSNINGFUNNEN
  else { DRAGGENERERING(lista, n)
         for k ← 1 to n
           do { UTFÖRDRAG(k)
                NÄSTADRAG(nr + 1)
                ÅTERSTÄLLDRAG(k)
```

Det är funktionen `nästadrag` som är programmets hjärta, men det återstår en hel del administration innan man kan sätta igång med kodningen.

Datastruktur Lämplig som datastruktur är en matris. Att den bör vara av storleken 11×11 och inte 7×7 förklaras här.



Figur 8.4: *En situation i spelet*

De *ljusgrå* rutorna är platser utanför spelplanen, de *vita* är kulor och de *mörkgrå* är tomma platser. När man ska bestämma till vilka platser den utmärkta kulan i $(5, 2)$ kan flytta finns det fyra riktningar att studera. Vi vill då ha reda på om den närmaste platsen innehåller en kula och om den längre bort är tom.

Om matrisen har 7 rader och 7 kolumner kommer man att hamna utanför den, när man ”tittar” åt vänster. För att hela tiden slippa testa, om det är tillåtet att ”titta” i en viss riktning, utökar vi spelplanen med en ram, som gör att vi hela tiden håller oss på spelplanen.

Lika viktigt är det att från början fylla matrisen med lämplig information så att dragen *uppåt* och *nedåt* i denna situation inte bokförs som möjliga.

Vid initieringen bör vi fylla matrisen med tre olika värden:

- Tom plats
- Kula
- Otillåten plats

Denna matris kan vi placera globalt. De som är motståndare till *globala* variabler får i stället öka antalet parametrar i funktionen `nästadrag`.

Till datastrukturerna lägger vi dessutom en matris av storleken 32×6 som ska innehålla information om tillåtna drag. 32 är förstås en grov överdrift eftersom vi aldrig når en situation där alla 32 kulorna kan flyttas (kom gärna på ett bättre värde).

För varje drag behöver vi sex olika tal.

- *rad* och *kolumn* för platsen där kulan just nu finns
- *rad* och *kolumn* för platsen där kulan som ska hoppas över finns
- *rad* och *kolumn* för platsen dit kulan ska flyttas.

Funktioner Förutom funktionen `nästadrag` behöver vi en funktion som initierar spelplanen, enligt ovan, innan spelet börjar.

Vi behöver dessutom funktionen `draggenerering` som fyller på ”dragmatrisen” med alla i ställningen möjliga drag. Eftersom denna matris är lokal, (det behövs en för varje ny ställning) behövs den som parameter tillsammans med `n` som håller reda på hur många drag som finns.

Funktionerna `utfördrag` och `återställdrag` blir så korta att koden med fördel kan skrivas in i funktionen `nästadrag`. Här ska man alltså omsätta informationen i dragmatrisen till en förändring på spelplanen som senare skall återställas.

UPPGIFT 8.4
Spara dragen till Solitär. När funktionen `lösningfunnen` anropas finns inget annat att göra än att skriva ”Ja, det fanns en lösning”. Ingenstans har vi lagrat dragen fram till målet och kan därför inte skriva ut lösningen.

Utöka därför programmet `solitar.c` så att dragen bokförs vartefter de utförs. När man når målet har man så hela dragföljden och kan skriva ut den till en fil.

Om du formaterar filen så att den innehåller ett drag per rad med samma data i samma ordning som dragmatrisen, kan du använda programmet `SpelarUppSolitär` för att ”spela upp” din lösning. Programmet använder $(5,5)$ för spelplanens mitt Använd höger musknapp för att få fram programmenyn.

UPPGIFT 8.5
Var kan sista kulan hamna?. Det finns ett tillägg till *solitär*. Man vill bestämma var sista kulan ska hamna. Kanske är det vackrast om den hamnar i mitten? Om man tar hänsyn till symmetrin finns det *sju* olika platser där den sista kulan kan hamna.

Finns det en lösning för alla dessa platser? Utöka programmet så att vi kan få svar på frågan. Observera att frågeställaren inte känner till svaret!

8.4 Att måla

Att fylla en area på skärmen i grafisk mode, det som normalt `floodfill` i C's DOS-grafik gör kan vi härma med följande rekursiva funktion. Funktionen `draw` ritar först, med vita streck, en triangel på den svarta bildskärmen. Funktionen `fill` tar emot en x- och y-koordinat. Det är viktigt att denna pixel ligger inuti triangeln.

```
1  void draw(void){
2    int gd=9,gm=2;
3    initgraph(&gd,&gm,"d:\\program\\bc45\\bgi");
4    moveto(100,100); lineto(150,150); lineto(150,100);
5    lineto(100,100);
6  }
7  void fill(int x,int y){
8    int n,x1,y1;
9    putpixel(x,y,WHITE);
10   for(n=1;n<=4;n++){
11         switch (n) {
12               case 1:{x1=x-1; y1=y;} break;
13               case 2:{y1=y-1; x1=x;} break;
14               case 3:{x1=x+1; y1=y;} break;
15               case 4:{y1=y+1; x1=x;} break;
16         }
17         if (getpixel(x1,y1)==BLACK) fill(x1,y1);
18   }
19 }
```

- Direkt målar funktionen pixeln (x, y) vit.
- For-loopen påbörjar nu en ”snurra” på fyra varv. Ett för varje riktning *vänster, uppåt, höger, nedåt*
- Funktionen tar i tur och ordning ett steg i varje riktning.
- Om denna pixel är svart anropar funktionen `fill` sig själv, med dessa koordinater som parametrar, vars motsvarande pixel inledningsvis målas vit. Och så vidare ...
- Så småningom måste funktionen hamna i ett läge då ingen av de fyra riktningarna ger en svart pixel – då sker heller inget nytt anrop utan återhopp sker.
- När exekveringen kommer tillbaka till ett ”gammalt” anrop finns där förstås uppgifter om vilket värde `n` har och for-loopen kan snurra vidare.

Tyvärr fungerar denna funktion bra endast för relativt små ytor som ska målas. Då ytan blir större kommer antalet anrop att växa och risk finns för att stacken ska bli full.

Tekniken kommer ofta till användning i andra, än grafiska, tillämpningar.

8.5 Permutationer

Bokstäverna ABC kan ordnas om på ytterligare fem sätt: ACB, BAC, BCA, CAB, CBA. Varje sådan ordning kallas en *permutation* av ABC. Det finns 6 permutationer för tre objekt, $3! = 6$. På samma sätt kan 4 bokstäver ordnas på $4! = 24$ sätt.

Det finns många problem där man har anledning att generera alla permutationer av ett antal objekt. Tyvärr växer antal permutationer som bekant ganska raskt. Det är sällsynt att man, i en sådan undersökning, tror sig klara av fler än 13 objekt med sina $13! = 6\ 227\ 020\ 800$ permutationer.

```
1  #include <stdio.h>
2  void permute(int a[],int n,int v[],int p){
3    int i;
4    if(n==p){
5      for(i=0;i<n;i++)
6        printf("%d ",a[i]);
7      printf("\n");
8    }
9    else
10     for(i=0;i<n;i++)
11       if(!v[i]){
12         v[i]=1;
13         a[p]=i;
14         permute(a,n,v,p+1);
15         v[i]=0;
16       }
17 }
```

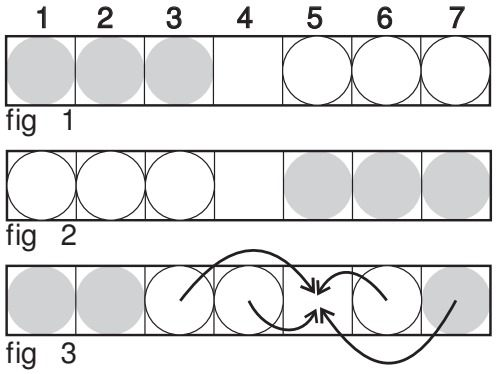
```
18 int main(void){
19   int a[5],v[5]={0};
20   permute(a,5,v,0);
21 }
```

H I `a` hamnar talen `0` till $n-1$ i alla tänkbara $n!$ ordningar. `v` använder vi för att boka av använda tal, där `0` betyder *ej använt*. `p` anger hur många tal vi tal vi placerat i `a`. När $p = n$ har vi nått ett löv i rekursionsträdet och kan skriva permutationen.

13 Här flyttar vi in talet på rätt plats i `a`. Detta värde behöver inte återställas utan kommer bara att skrivas över senare.

15 Som vanligt är det viktigt att återställa bokningen.

5-7 Just den här gången skriver vi bara ut permutationen. En annan gång anropar vi en funktion där `a` är en parameter och utför någon beräkning på just den permutationen.



Figur 8.5:

UPPGIFT 8.6

Brickspel. I figur 8.5 visas utgångsläget i ett enkelt spel med sju rutor och sex brickor, tre svarta och tre vita. Uppgiften består i ett antal drag byta plats på de svarta och de vita brickorna och nå ställningen i delfigur 2. Delfigur 3 visar fyra tillåtna drag i en viss situation. Man får alltså flytta en bricka till den intilliggande rutan eller hoppa över en bricka och hamna på den tomma rutan. Man behöver inte alternera brickornas färg mellan dragen, utan kan till exempel flytta två vita brickor i följd.

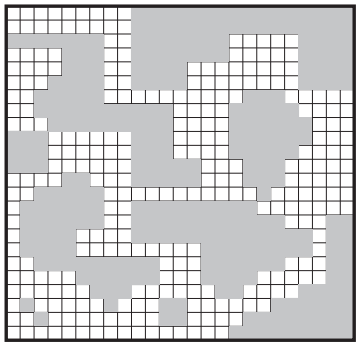
Skriv ett program som listar den unika dragföljden som når målet efter 15 drag då första draget görs med en svart bricka.

Indata: -

Utdata: Den eftersökta dragföljden på formen $3-4$, $5-3$, ... där siffrorna anger *frånruta* och *tillruta* enligt numrering i figur 8.5. Från hemsidan kan du ladda ner BRICKSOLITÄR. Klarar du det på 15 drag?

UPPGIFT 8.7

Öar i skärgården. I figur 8.6 ser du en karta över en skärgård. De grå fälten utgör *land* och de vita är *vatten*. Du ska nu skriva ett program som utifrån en given karta bestämmer hur många öar skärgården har. Eftersom två landmassor, som *endast* har ett diagonalt förhållande till varandra, räknas som skilda öar, finns det i figuren 10 öar.



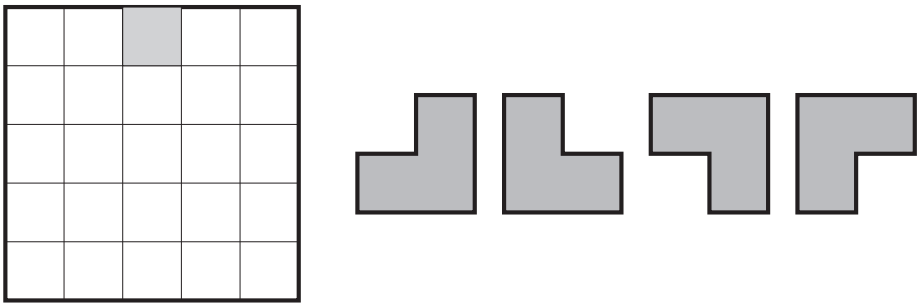
Figur 8.6: *Kartan visar 10 öar. Nere i vänstra hörnet ser vi två små öar*

Det finns ett antal färdiga kartor på datafiler, alla med namnet `oar?.dat`. Datafilen inleds med två heltal $n < 20$, för antalet rader (av rutor) och $m < 20$ för antalet kolumner. Därefter följer n rader med m tecken i varje, där `x`, står för land och `o` står för vatten.

Extrauppgift: Du har *fyllnadsmaterial i form av jord*, motsvarande 4 rutor, *var ska du placera den för att minimera antalet öar på kartan?*

UPPGIFT 8.8

Pussel med vinklar. I figur 8.7 nedan ser du ett bräde, 5×5 rutor stort. Till vänster ser du pusselbitar, som alla är av samma typ. De har bara vridits 90°. Målet är nu att skriva ett program som täcker brädet med hjälp av 8 pusselbitar (vridna och vända). Observera, att den grå rutan dock inte får täckas.



Figur 8.7: *Kan pusslet läggas med hjälp av* 8 *L-formade pusselbitar, där den grå rutan inte får övertäckas?*

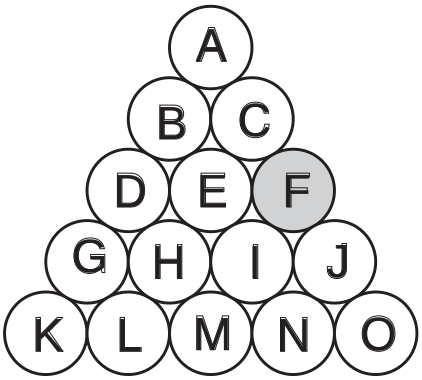
Hur många olika lösningar har pusslet för skilda placeringar av den grå rutan?

UPPGIFT 8.9

Nytt solitärspel. Figur 8.8 visar ett solitärspel, enmansspel, med 14 kulor och ett hål (F). Målet är att efter *tretton drag* ha endast en kula kvar på brädet. Ett *tillåtet drag* är att med en kula hoppa över en intilliggande och hamna i ett tomt hål på andra sidan, varefter den överhoppade kulan avlägsnas. I figuren finns fyra möjliga drag AF, OF, MF och DF. Efter dessa drag avlägsnas motsvarande kulor i C, J, I och E. Skriv ett program som efterfrågar vilket hål som ska vara tomt vid spelets början och som sedan presenterar en lösning. För varje utgångsläge finns flera lösningar, men vi efterfrågar alltså endast en, som ska beräknas av programmet.

Indata: Bokstaven som motsvarar det hål som ska vara tomt.

Utdata: De tretton dragen som beskriver en lösning. Ett drag anges på formen `<Bokstav Bokstav>`, till exempel AD, där A anger från vilken position kulan flyttas och D till vilken position.



Figur 8.8: *Figur till uppgift 4.6*

UPPGIFT 8.10

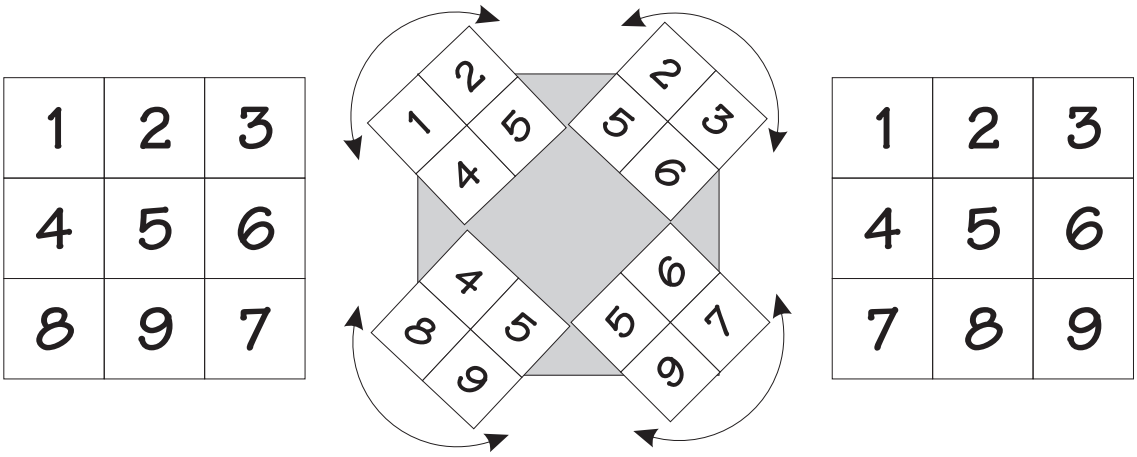
Nokiaspel

I flera av Nokias mobiltelefoner finns detta förströelsespel eller pussel. Pusslet går ut på att från ett givet, slumpmässigt utgångsläge, som till exempel till vänster i figur 8.9 nå fram till ordning och reda som till höger. I varje ställning finns åtta möjliga drag. Varje liten 2×2-kvadrat kan vridas $90°$ *medurs* eller *moturs* i taget. Efter fyra drag i följd och i samma riktning med samma kvadrat är utgångsställningen återuppnådd.

Vi har undersökt att det aldrig behövs fler än 11 drag för att nå målet från vilken utgångsställning som helst. Din uppgift är att skriva ett program som tar emot en ställning och som bestämmer hur många drag den minst kräver för att nå målet.

Ställningen ska skrivas in, rad för rad, till programmet som en sträng innehållande en permutation av siffrorna 1...9. Utdata är en enda mening.

```
Vilken ställning: 123456897
```



Figur 8.9: *Till vänster ser vi en av många startmöjligheter. I mitten visas hur de fyra* 2×2*-kvadraterna kan roteras. Till höger visas målet.*

```
Denna ställning kräver 6 drag
```

Vi ger här ytterligare några testexempel alla med olika ”svårighetsgrad” observera att vid ren backtracking kan det ta ganska lång tid för datorn att hitta lösningar till problem som kräver 10 och 11 drag.

Ställning	**Kräver**	**Ställning**	**Kräver**
123456789	0	123469758	1
123498765	2	123458976	3
123457698	4	123456798	5
123456897	6	123456987	7
123584679	8	123654789	9
147268359	10	387654921	11

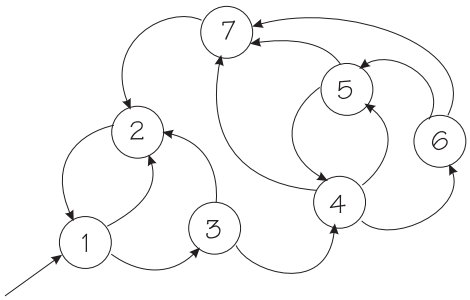
UPPGIFT 8.11

Resan

I figur 8.10 ser Du en karta bestående av sju städer. Mellan städerna finns *enkelriktade* vägar, där pilen anger riktningen.

Din uppgift blir att skriva en eller flera rutiner till det bifogade programmet `uppg5.cpp`, som finner en väg med start i *stad 1* som går genom alla de $n (n \leq 20)$ städerna.

Data läses från filen `uppg5.dat` av funktionen `init`. Denna funktion som lämnar ifrån sig en pekare till *stad 1* och *antalet städer på kartan* ska du använda men inte ändra.



Figur 8.10:

```
struct stadtyp{
  int nr;
  struct stadtyp *adr[4];
};
```

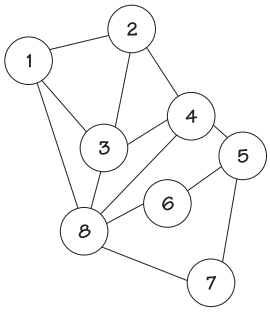
Varje stad består av ett dynamiskt allokerat objekt (struct stadtyp), som innehåller *stadens nummer* samt en array med tre pekare, index 1 till 3. (index 0 används inte). Om en av dessa pekare inte är NULL pekar den på en annan stad till vilket det är möjligt att färdas.

Resultatet skrivs ut på en rad med städernas nummer i den ordning de besöks. Från exemplet ovan skrivs resultatet

```
Resan: 1 3 4 6 5 7 2
```

Resan börjar alltså alltid i *stad 1* och får avslutas var som helst när samtliga städer har besökts. Varje test har minst en lösning. Om det finns flera olika resor som uppfyller villkoren så räcker det med att redovisa en av dessa.

UPPGIFT 8.12
Vägar i en graf



Figur 8.11:

I figur 8.11 ser vi en *graf*. I denna uppgift ska Du skriva ett program som bestämmer *antalet turer*, som startar i nod **1** och slutar i nod **n**. Under resans gång ska alla noder i grafen besökas exakt en gång. Nod **n** är det högsta nodnumret i grafen. I vårt exempel är $n = 8$ och i testexemplen alltid $n < 20$.

På filen UPPG6.DAT finns uppgifter om hur noderna är förbundna. Filen inleds med ett tal som anger n. Därefter kommer ett tal som anger hur många *bågar* (kanter), $m, m < 100$, grafen har. Filen avslutas med m rader där varje rad innehåller två tal – nodnumren för de noder som förbinds av bågen.

Programmet skriver endast ut en rad. Med vårt exempel får vi:

```
Det finns 4 turer
```

8.5.1 Vandringen – revisited

Vi upprepar frågeställningen:

UPPGIFT 8.13
Vandringen Vi befinner oss i koordinaten (x, y) och vill hem till $(0, 0)$. Vår promenad är uppdelad i små promenader i rakt nordlig (8 steg), sydlig (3 steg), östlig (5 steg) och västlig (6 steg) riktning. Vilket är det minsta antalet små promenader vi behöver ta för att nå målet?

Skriv ett program som använder sig av tekniken *bredden först* och med följande dialog

```
Start x : 36
Start y : 27
Det krävs minst 15 små promenader
```

Eftersom *heapen* är begränsad kan den bli full! Detta kan man hålla kontroll på genom:

```
ny=(struct queuebox *) malloc(sizeof(struct queuebox));
  if(ny==NULL){
  printf("Heapen full\n");
  getch();
  abort();
}
```

Den post vi vill köa kan möjligtvis komprimeras till

```
struct box{
  char x,y,antal;
};
struct queuebox{
  struct box data;
  struct queuebox *next;
};
```

`box` tar upp 3 byte tillsammans med pekaren, som tar upp 8 byte, får vi 11 byte/post. Vi förväntar oss att finna resultatet på nivå 15. Här finns $4^{15} = 1073741824$ noder på totalt 11 811 160 064 byte, alldeles för stort för normala persondatorer. Sorry! Även om vi lyckas hitta fram 'tidigt' på denna nivå finns 4^{14} noder i kön från tidigare nivå.

Djupet först

Vet vi att resultatet för $(36, 27)$ ska bli 15 är det lätt att verifiera med hjälp djupet först:

```
1  #include <stdio.h>
2  int dragmin=100000, maxdrag=15;
3  void solve(int dragnr,int x, int y){
4    if(x==0 && y==0){
5      if(dragnr<=dragmin){
6        dragmin=dragnr;
7      }
8    }
9    else
10     if(dragnr<maxdrag){
11       solve(dragnr+1,x,y+8);
12       solve(dragnr+1,x,y-3);
13       solve(dragnr+1,x+5,y);
14       solve(dragnr+1,x-6,y);
15     }
16 }
17 int main(void){
18   solve(0,36,27);
19   printf("Minimum: %d\n",dragmin);
20 }
```

Efter att ha undersökt 357 913 942 noder hittar vi svaret 15. Man kan nå målet på 5005 olika sätt. Ersätter vi rad 10 med

```
if(dragnr<maxdrag && dragnr<dragmin){
```

kan vi tjäna en del för problem som kräver färre än 15 promenader.

Retrograde analysis

Vi ger inte upp. Här följer en algoritm, som vi kan kalla *bakåtanalys*, som löser problemet på mindre än en sekund. Inte bara för efterfrågade $x = 36$ och $y = 27$ utan för nästan alla startkoordinater i $[-50 \leq x \leq 50, -50 \leq y \leq 50]$.

```
1  #include <stdio.h>
2  int main(void){
3    int plan[101][101],ok=1,y,x,level=0;
4    for(y=0;y<=100;y++)
5      for(x=0;x<=100;x++)
6        plan[y][x]=-1;
7    plan[50][50]=0;
8    while(ok){
9      ok=0;
10     for(y=0;y<100;y++)
11       for(x=0;x<100;x++)
12         if(plan[y][x]==level){
13           if(y-8>=0 && plan[y-8][x]==-1){
14             plan[y-8][x]=level+1;
15             ok=1;
16           }
17           if(y+3<=100 && plan[y+3][x]==-1){
18             plan[y+3][x]=level+1;
19             ok=1;
20           }
21           if(x-5<=100 && plan[y][x-5]==-1){
22             plan[y][x-5]=level+1;
23             ok=1;
24           }
25           if(x+6>=0 && plan[y][x+6]==-1){
26             plan[y][x+6]=level+1;
27             ok=1;
28           }
29         }
30     level++;
31   }
32   printf("(36,27) efter %d\n",plan[77][86]);
33 }
```

3 Vi deklarerar en array `plan`, där varje cell motsvarar en startkoordinat.

4-7 Arrayen -1-ställs. I [50][50] bestämmer vi oss för att origo $(0,0)$, vårt mål ska ligga. Om man startar i $(0,0)$ är man redan framme därför placerar vi 0 i denna cell.

8-31 Tanken är nu att placera talet 1 i de fyra punkter från vilka man kan ta sig till mål i ett drag. Vi går igenom hela matrisen och letar efter talet 0, det värde `level` har från början. När vi funnit det i [50][50] kan vi placera talet 1 i [42][50], [53][50], [50][45] och [50][56]. Vi hittar inga fler celler med talet 0. Ökar `level` till 1 och går igenom hela matrisen igen och söker efter celler med 1:or. Varje gång vi hittar en, ”expanderar” vi

den genom att titta i fyra riktningar. Vi aktar oss för att hamna utanför matrisen och sätter inte in några 2:or i de celler som redan har 1:or.
Om flaggan `ok` inte slagit om under en hel genomsökning av matrisen betyder det att inga fler celler kan expanderas och rutinen avbryts.
I `plan` finns nu tal som anger hur många drag det behövs för att nå målet. Några celler kan fortfarande innehålla talet -1. För dessa gäller att man måste gå utanför matrisen i några drag.
Detta är ett exempel på en lösning där man börjar från målet och går baklänges. Har man bestämt sig för en viss koordinat redan innan man fyller i matrisen kan man sluta, så fort man når fram till aktuell koordinat.

UPPGIFT 8.14
Hissen i lustiga huset för tredje gången. Lös problemet med denna teknik och med hjälp av penna och papper, då antalet våningar är 25. *Upp-Knappen* ger 5 våningar, *Ned-Knappen* ger 7 våningar och vi ska till våning 23. Hur många knapptryckningar blir det?

En heuristisk algoritm

I valet mellan de fyra möjliga promenaderna kan man tycka att man först bör utföra den promenad, som närmar sig målet mest. Vi kan inte bevisa att detta i allmänhet är ett bra val och med säkerhet är det inte *alltid* det bästa. Denna typ av intuitiva idéer kallas *heuristiska*. Tanken är att man relativt snabbt ska få en ”hyfsad lösning”, som i fortsättningen gör det möjligt att utesluta stora delar av rekursionsträdet.

```
1  int dragmin=100000;
2  void genereradrag(int x,int y,int d[][3]){
3    int steg[4][2]={{0,8},{5,0},{0,-3},{-6,0}};
4    int i,j,k,tmp;
5    for(i=0;i<4;i++){
6      d[i][0]=x+steg[i][0];
7      d[i][1]=y+steg[i][1];
8      d[i][2]=abs(d[i][0])+abs(d[i][1]);
9    }
10   for(i=0;i<3;i++)
11     for(j=i+1;j<4;j++)
12       if(d[i][2]>d[j][2])
13         for(k=0;k<3;k++){
14           tmp=d[i][k];
15           d[i][k]=d[j][k];
16           d[j][k]=tmp;
17         }
18 }
```

5-9 Vi beräknar koordinaterna för de fyra dragen, samt den summa vi sedan ska sortera efter. Ju närmare målet vi hamnar genom draget desto högre upp i listan.

10-17 En vanlig bubbelsort.

```
1  void solve(int dragnr,int x, int y){
2    int drag[4][3],i;
3    if(x==0 && y==0){
4      if(dragnr<=dragmin)
5        dragmin=dragnr;
6    }
7    else
8      if(dragnr<dragmin){
9        genereradrag(x,y,drag);
10       for(i=0;i<4;i++)
11         solve(dragnr+1,drag[i][0],drag[i][1]);
12     }
13 }
14 int main(void){
15   solve(0,36,27);
16   printf("Minimum: %d\n",dragmin);
17 }
```

H Observera att vi nu släppt begränsningen på trädets djup. Vi räknar med att programmet ska hitta en bra lösning relativt snabbt, som därefter begränsar sökdjupet. Efter att ha genererat 357 913 941 noder finner programmet lösningen 15 promenader.

8 Denna sats är viktig och innebär att man aldrig söker vidare när det inte finns någon chans att slå rekordet. Självklart kan tyckas.

9 Vi genererar de fyra dragen, som levereras i en sorterad array, med det drag som ligger närmast målet först.

10-11 Precis som i tidigare program anropar vi nu `solve` fyra gånger, men nu i en vad vi tror bättre ordning.

Jämför vi detta program med djupet först för andra startkoordinater får vi för starten $(0,-1)$ resultatet 7 promenader efter att ha undersökt 5461 noder (7 454 606 noder för djupet först och med utökat villkor). Med starten $(-1,0)$ undersöker vi 92841 noder för att nå resultatet 9 promenader (att jämföra med 238 753 536 för djupet först och utökat villkor). Av dessa tester bör vi kunna sluta oss att den implementerade heuristiska metoden snabbar upp programmet.

Diofantisk ekvation

Om vi antar att antalet promenader i nordlig riktning är n, sydlig riktning är s, västlig är w och östlig e kan vi ställa upp följande diofantiska ekvationssystem

$$\begin{cases} 6w - 5e = 36 \\ 3s - 8n = 27 \end{cases}$$

Med hjälp av Maple får vi för `isolve(6w-5e=36)`

$$\begin{cases} e = -6 + 6t \\ w = 1 + 5t \end{cases}$$

Det finns som bekant oändligt många lösningar, för varje t får vi en. Vilket värde på t ska vi välja så att $e \geq 0, w \geq 0$ och så att $e + w$ blir så litet som möjligt. $t = 1$ ger $e = 0$ och $w = 6$

På samma sätt för `isolve(3s+8n=27)` får vi

$$\begin{cases} n = 3t \\ s = 9 + 8t \end{cases}$$

För $t = 0$ får vi $n = 0$ och $s = 9$. Totalt ger detta som väntat 15 promenader.

$$\frac{15!}{9! \cdot 6!} = 5005$$

visar att målet kan nås på 5005 olika sätt genom att kasta om ordningen hos promenaderna.