# Kapitel 9

# Glupska algoritmer

Det vi här kallar *glupsk algoritm* heter på engelska *greedy algorithm*. Algoritmerna konstruerade efter denna idé är alltså snarare glupska än giriga!

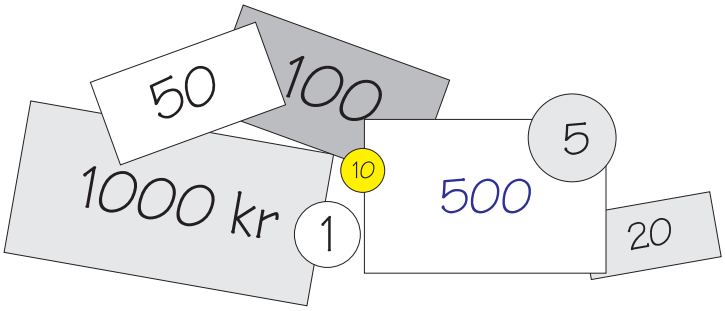
En glupsk algoritm startar med att söka lösningen till ett litet delproblem av hela problemet. På ett ”glupskt” sätt gör algoritmen sitt val, enbart efter lokala omständigheter och utan att tänka på vad detta val kommer att betyda för den totala, slutliga lösningen. Detta arbetssätt kan leda till en optimal, en ganska bra eller en urusel lösning.

Glupska algoritmer är oftast enkla att utveckla, enkla att implementera, enkla att analysera och är dessutom mycket snabba. Problemet är dock att det oftast är mycket svårt att bevisa att de är korrekta.

Som tur är finns det några välkända algoritmer av detta slag, som är bevisade vara korrekta och som vi ska studera här. Förhoppningsvis ska dessa vara tillräckliga för att påvisa de glupska algoritmernas karaktär.

## 9.1 Mynt och sedlar

Som första exempel på en glupsk algoritm ska vi diskutera följande problem:




Figur 9.1: *Alla valörer av svenska mynt och sedlar, utom 50-öringen ska användas i detta exempel*

I figuren ser du de svenska mynt och sedlar som används idag (utom 50-öringen). Hur många mynt och sedlar behöver man *högst* för att räkna upp ett givet belopp på $x \leq 1000$ kr?

Den metod som vi först kommer att tänka på fungerar alldeles utmärkt. *Starta med den högsta valör som är $\leq x$. Använd den valören så länge det är möjligt. Välj sedan den högsta valör som är $\leq$ det som återstår. Använd den så många gånger som möjligt. Och så vidare tills hela beloppet är uppräknat*

Viktigt är att myntet 1 kr finns med, annars skulle det finnas belopp som inte kunde uppnås. Alla andra mynt skulle man kunna avstå ifrån även om det skulle innebära att antalet mynt ökar.

**Algorithm 9.1.1:** GREEDYVÄXEL(valörer, antal, belopp)

```
n ← 0
for i ← antal downto 1
  do { while belopp ≥ valörer[i]
       do { belopp ← belopp − valörer[i]
            n ← n + 1
return (n)
```

`Valörer` är en array som förstås innehåller de valörer som finns, sorterade i stigande värde. `antal` anger antalet valörer och `belopp` det belopp man med så få mynt som möjligt vill räkna upp.

Kommer den här algoritmen alltid att fungera? Svaret är ja. För dessa valörer ger oss GREEDYVÄXEL alltid den optimala lösningen. Men hur är det då med andra uppsättningar av valörer? Svaret är: *Nej vår glupska algoritm fungerar inte alltid för andra uppsättningar.* Vad sägs till exempel om valörerna $[1, 10, 25]$ och beloppet 30. Algoritmen ovan föreslår $25, 1, 1, 1, 1, 1$, alltså 6 mynt men vi kan se att 3 mynt räcker – $10, 10, 10$ är den optimala lösningen.

Hur som helst är detta en mycket bra introduktion till glupska algoritmer. Men två frågor kvarstår: Hur löser vi problemet då GREEDYVÄXEL inte fungerar? Hur kan vi se när GREEDYVÄXEL inte fungerar? Den första frågan återkommer vi till i samband med *dynamisk programmering* och den andra frågan är för stor för vår kurs!

UPPGIFT 9.1
**Med svenska mynt och sedlar**. Implementera GREEDYVÄXEL för de svenska valörerna och ta med programmet reda på vilka belopp $b \leq 1000$, som kräver flest mynt och sedlar. Hur många dessa belopp är och hur många mynt och sedlar som behövs.

## 9.2 Schemaläggning 1

Vi har en mängd $A$ av $n$ *aktiviteter* $a_1, a_2 \ldots a_n$, som vi vill schemalägga i en sal. Till varje aktivitet finns bestämd *starttid* $s_i$ och *sluttid* $t_i$. Vårt mål är att placera in *så många aktiviteter som möjligt*. Självklart får inga aktiviteter tidsmässigt överlappa varandra.

Här följer fyra tänkbara algoritmer för att lösa problemet. Endast en av dem ger en optimal lösning för alla tänkbara uppsättningar av indata.

**Algoritm 1**

a) $S$ mängden av schemalagda aktiviteter, som är tom från start

b) Flytta till $S$, den aktivitet $i$ i $A$, som har tidigaste starttiden $s_i$.

c) Stryk sedan ur $A$, alla aktiviteter ur $S$, som överlappar den just valda.

d) Om $A$ är tom finns resultatet i $S$ annars gå till b)

**Algoritm 2**

a) $S$ mängden av schemalagda aktiviteter, som är tom från start

b) Flytta till $S$, den aktivitet $i$ i $A$, med kortaste tiden $t_i - s_i$.

c) Stryk sedan ur $A$, alla aktiviteter ur $S$, som överlappar den just valda.

d) Om $A$ är tom finns resultatet i $S$ annars gå till b)

**Algoritm 3**

a) $S$ mängden av schemalagda aktiviteter, som är tom från start

b) Flytta till $S$, den aktivitet $i$ i $A$, som står i konflikt med (överlappas av) minst antal andra aktiviteter.

c) Stryk sedan ur $A$, alla aktiviteter ur $S$, som överlappar den just valda.

d) Om $A$ är tom finns resultatet i $S$ annars gå till b)

**Algoritm 4**

a) $S$ mängden av schemalagda aktiviteter, som är tom från start

b) Flytta till $S$, den aktivitet $i$ i $A$, som slutar först, har minsta $t_i$

c) Stryk sedan ur $A$, alla aktiviteter ur $S$, som överlappar den just valda.

d) Om $A$ är tom finns resultatet i $S$ annars gå till b)

Eftersom vi påstår att den bästa algoritmen ovan alltid ger det optimala resultatet räcker det med ett motexempel för tre av de givna algoritmerna för att komma fram till rätt svar.

I figur 9.2 ser vi ett motexempel, som får oss att förkasta algoritm 1. Längst ned har vi tidsaxeln. Direkt ovanför har vi den aktivitet vi enligt algoritmen ska välja. Detta leder till att de två andra aktiviteterna slås ut. Resultatet blir alltså 1 aktivitet i stället för det uppenbarliga 2.

I figur 9.3 visas ett exempel där algoritm 2 inte ger det optimala resultatet. Vi väljer den kortaste aktiviteten, som slår ut den de två andra!

Figur 9.2:

Figur 9.3:

Algoritm 3 är det bästa förslaget hittills och det är ganska knepigt att hitta motexempel, men figur 9.4 visar ett. Väljer vi aktiviteten närmast tidsaxeln, som ju endast har två överlappningar, får vi sedan plats med endast två ytterligare aktiviteter.

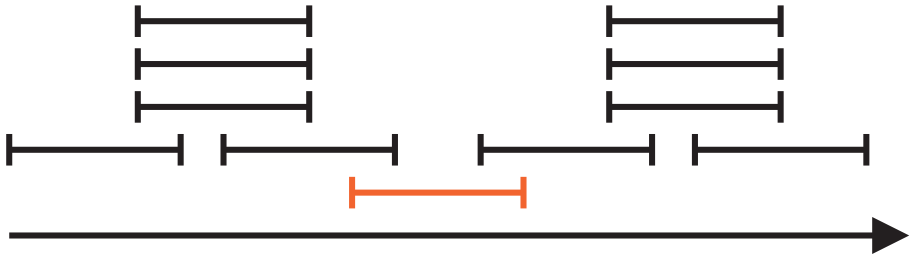
Det är alltså Algoritm 4 som är bäst och som alltid ger optimalt resultat. Vi bryr oss inte om att bevisa detta, men kan enkelt se att den fungerar för våra tre exempel ovan.

## 9.3 Schemaläggning 2

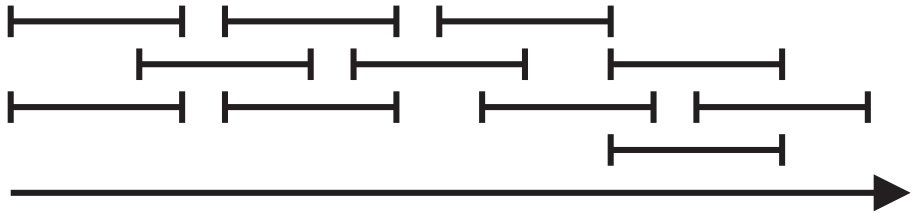
Ett nytt schemaläggningsproblem med en mängd $A$ av $n$ *aktiviteter* $a_1, a_2 \ldots a_n$, som vi *alla* vill schemalägga. Men denna gång har flera salar till vårt förfogande. Till varje aktivitet finns som förut bestämda *starttider* $s_i$ och *sluttider* $t_i$. Vårt mål är minimera antalet salar vi behöver.

Detta betyder förstås att två aktiviteter som överlappar aldrig kan hamna i samma sal. Vi studerar exemplet i figur 9.5.

Vi förstår direkt att man måste ta till minst 3 salar eftersom det på flera ställen finns tre aktiviteter som överlappar varandra. Man klarar sig alltid med $d$ salar, där $d$ står för den största antalet aktiviteter som överlappar varandra om man följer denna algoritm.



Figur 9.4:



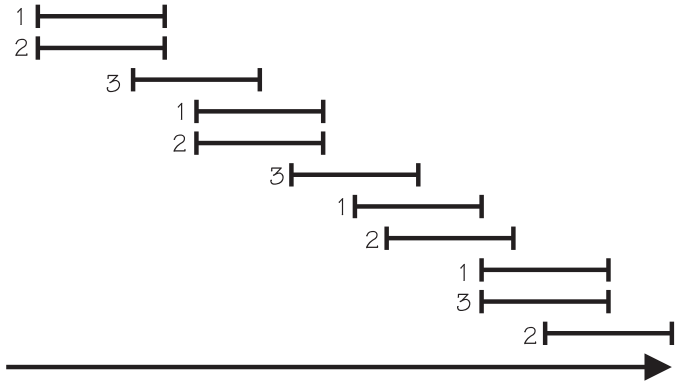
Figur 9.5:

A är en matris med tre kolumner. Den första innehåller $s_i$, den andra $t_i$ och den sista kommer att innehålla tilldelad sal.

**Algorithm 9.3.1:** SCHEMALÄGGNING$(A, n, d)$

```
Sortera de n aktiviteterna i A efter stigande s_i
for i ← 1 to n
do { for j ← 1 to d
     do sal[j] ← 1
   for j ← 1 to i − 1
     do { if A[j][sluttid] > A[i][starttid]
          then sal[A[j][sal] ← 0
   j ← 1
   while sal[j] = 0
     do j ← j + 1
   A[i][sal] ← j
```

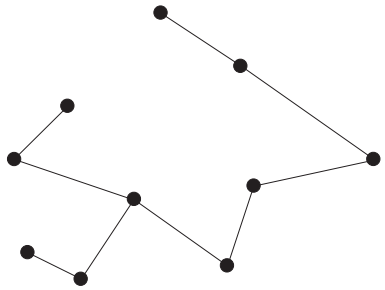
I figur 9.6 kan vi se hur de 11 aktiviteterna hamnar i 3 olika salar.




Figur 9.6:

## 9.4 MCSP, Minimum cost spanning tree

Vi presenterar MCSP genom ett exempel. I figur 9.7 ser vi en karta över 10 städer. Mellan en del städer finns en rät linje, en väg. Hela systemet av vägar sammanbinder alla städer. Man önskar konstruera vägnätet på ett så ekonomiskt sätt som möjligt. Man vill minimera den sammanlagda vägsträckan. En väg dras alltid rätlinjigt mellan två städer. Lösningen till detta problem är en graf. Om den innehåller $n$ städer finns det $n-1$ bågar. Lösningen kallas *det minsta uppspännande trädet*

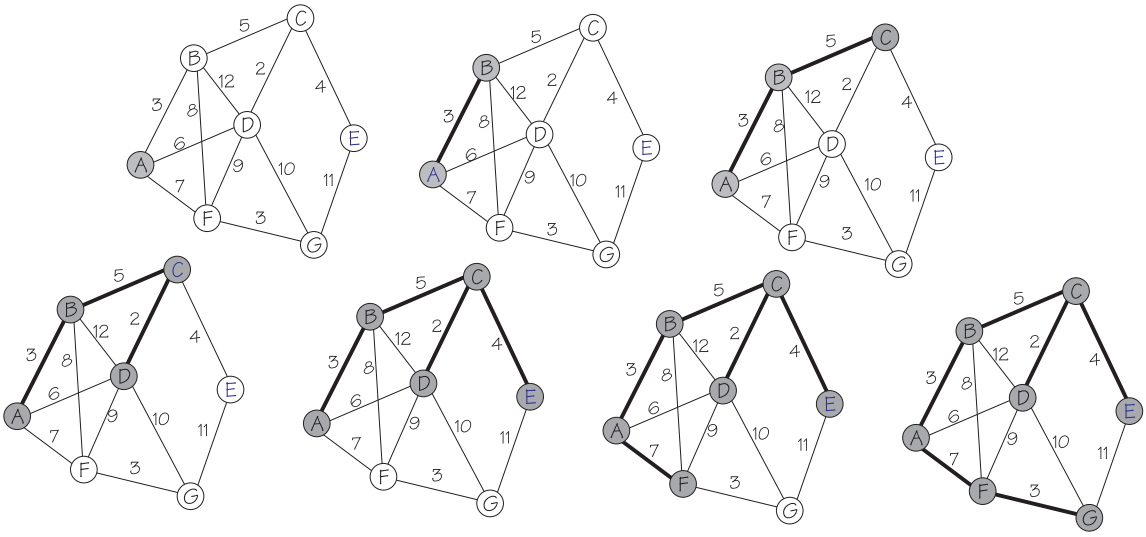


Figur 9.7: *En karta över* 10 *städer*

Det program vi vill ha, ska ta emot uppgift om antalet städer och deras koordinater. Vi kan för enkelhet skull begränsa oss till heltalskoordinater, både för $x$ och $y$, i intervallet $[0, 100]$. Programmet ska sedan finna det kortaste vägnätet genom följande algoritm:

- Välj en stad A, vilken som helst.
- Sök reda på den stad B, som ligger närmast A.
- Förbind A och B med en väg.
- Sök upp den stad C, som ligger närmast någon av städerna A och B.
- Förbind C med en väg till närmaste stad, A eller B.
- Nu består vägnätet av tre städer och två vägar. Nästa stad som ska förbindas är den stad som ligger närmast någon av de tre städerna A, B eller C.
- Fortsätt denna procedur tills alla städer är anslutna.

I verkligheten skulle man kunna finna ännu bättre lösningar än vad denna algoritm ger, genom att lägga in *vägkorsningar* eller *knutpunkter* på lämpliga platser på kartan. Detta problem kallas *Steiner tree problem* och är ett helt annat problem. Det är alltså inte tillåtet, att i denna uppgift, lägga till nya punkter på kartan!




Figur 9.8: *Ett exempel som vill visa hur algoritmen fungerar*

UPPGIFT 9.2

**Avståndstabell** I ett område finns 10 tätorter. A till J. Man ska nu förbinda dem med genom ett vägnät. Givet är en avståndstabell kan man kan utläsa avståndet mellana alla par av orter. Bestäm med hjälp av penna och papper längden hos det kortaste vägnätet.

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 29 | 77 | 45 | 3 | 33 | 82 | 107 | 57 | 21 |
| B | 29 | 0 | 74 | 68 | 30 | 50 | 85 | 114 | 40 | 30 |
| C | 77 | 74 | 0 | 63 | 80 | 52 | 18 | 46 | 41 | 56 |
| D | 45 | 68 | 63 | 0 | 47 | 18 | 58 | 73 | 71 | 38 |
| E | 3 | 30 | 80 | 47 | 0 | 36 | 86 | 110 | 60 | 24 |
| F | 33 | 50 | 52 | 18 | 36 | 0 | 52 | 74 | 53 | 20 |
| G | 82 | 85 | 18 | 58 | 86 | 52 | 0 | 29 | 57 | 62 |
| H | 107 | 114 | 46 | 73 | 110 | 74 | 29 | 0 | 86 | 88 |
| I | 57 | 40 | 41 | 71 | 60 | 53 | 57 | 86 | 0 | 40 |
| J | 21 | 30 | 56 | 38 | 24 | 20 | 62 | 88 | 40 | 0 |

UPPGIFT 9.3

**Program för MCSP**. Skriv ett program i java som implementerar MCSP. Programmet ska demonstrera MCSP grafiskt. Från en koordinat-fil, som man ska kunna välja från programmet, visas först punkterna på skärmen. För varje knapptyckning visas så en ny väg tills hela trädet finns på skärmen.

På hemsidan finns ett fult program `MCSP` som vill visa idén.

## 9.5 Kortaste vägen

Även nästa problem hör till grafteorin. Denna gång handlar det om att bestämma *kortaste vägen* mellan två städer (noder) i ett land där vissa städer är förbundna med vägar. Till varje väg (båge) mellan två städer är längden given.

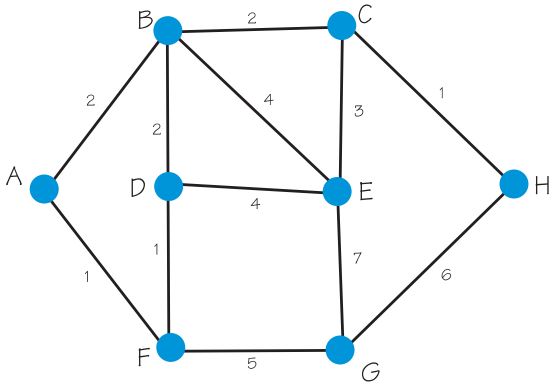
Algoritmen som den beskrivs i matematikboken

**Algorithm 9.5.1:** DIJKSTRA$(w, a, z, L)$

$L[a] \leftarrow 0$
**for** alla noder $x \neq a$
  **do** $L[x] \leftarrow \infty$
$T \leftarrow$ mängd av alla noder
**while** $z \in T$
  **do** välj $v \in T$ med minsta $L[v]$
    $T \leftarrow T - \{v\}$
    **for** varje $x \in T$ förbunden med $v$
      **do** $L[x] \leftarrow \min(L[x], L[v] + w(v, x))$

$w$ innehåller *vikten* (längden) för en båge mellan två städer. $a$ är *startnod* och $z$ är *slutnod*. I L kommer resultatet att lagras.

Ett exempel förklarar hur algoritmen fungerar.




Figur 9.9: *Att hitta den kortaste vägen från* ***A*** *till* ***H*** *utan algoritm är inte svårt. Men hur hittar man den med hjälp av Dijkstra's algoritm?*

Här ser vi huvudprogrammet som anropar Dijkstras algoritm. Grafen som beräknas är samma som i figuren.

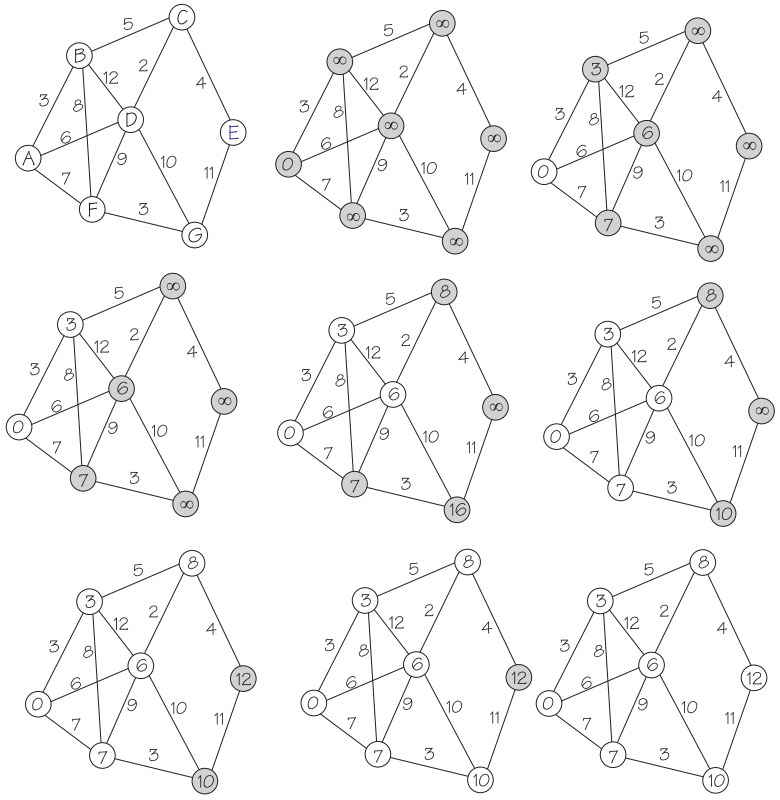
```
1  int main(void){
2    char graph[][3]={{'A','B',2},{'A','F',1},{'B','D',2},{'D','F',1},
3                     {'B','C',2},{'B','E',4},{'C','E',3},{'C','H',1},
4                     {'D','E',4},{'E','G',7},{'G','H',6},{'F','G',5}};
5    int path[256],i;
6    dijkstra(graph,12,8,'A','H',path);
7    printf("%d\n",path['H']);
8  }
9
10 void dijkstra(char graph[][3],int ne,int nv,char start,char stop,int path[]){
11   int t[256],i,j,v,x,min;
12   for(i='A';i<='A'+nv;i++){
13     path[i]=INT_MAX;
14     t[i]=1;
15   }
16   path[start]=0;
17
18   while(t[stop]){
19     min=INT_MAX;
20     for(i='A';i<='A'+nv;i++)
21       if(t[i] && path[i]<min){
22         v=i;
23         min=path[i];
24       }
25     t[v]=0;
26
27     for(j=0;j<ne;j++){
28       x=-1;
29       if(graph[j][0]==v) x=graph[j][1];
30       if(graph[j][1]==v) x=graph[j][0];
31       if(x>=0 && path[x]>path[v]+graph[j][2])
32          path[x]=path[v]+graph[j][2];
33     }
34   }
35 }
```

Parametrarna är i tur och ordning själva grafen (graph) som innehåller nodnummer och vägsträckans längd. Antalet vägar (ne) och antalet städer (nv). *start* anger numret på staden varifrån resan startar och *stop* anger var resan tar slut. I *path* rapporteras resultatet.

UPPGIFT 9.4

**Program för Dijkstra's algoritm**. Skriv ett program i java som demonstrerar Dijkstra's algoritm grafiskt. Programmet ska slumpa fram ett antal noder (dess skärmkoordinater) och ett lämpligt antal bågar. Användaren väljer mellan vilka två noder man önskar kortaste avståndet. Som sedan visas grafiskt.

På hemsidan finns `Kortaste vägen` som demo. Öppna en av filerna. Klicka på två noder och välj `Djupet först`. Man ska få samma svar om man väljer `Closest Path`.




Figur 9.10: *Här följer vi Dijkstra's algoritm steg för steg för att hitta den kortaste vägen från* A *till* E. *Som ett resultat på vägen har vi fått den kortaste vägen från* A *till alla andra noder.*

## 9.6 Knapsack Problem

Till detta problem hör $n$ objekt av något slag $b_1, b_2, \ldots, b_n$ och ryggsäck eller kont med *volymen* $C$. Varje objekt $b_i$ har en *volym* $w_i$ och ett *värde* $v_i$ för $i = 1, \ldots, n$.

Målet är nu att fylla ryggsäcken med en så värdefull last som möjligt. Observera att det är möjligt att lägga endast en del av ett objekt i säcken. Då $f_i b_i$ läggs till lasten för $0 \leq f_i \leq 1$ innebär det att det totala värdet ökar med $f_i v_i$ samtidigt som lastens totala volym ökar med $f_i w_i$.

Vi söker en lösning till problemet

$$\begin{aligned} \text{maximera } & \textstyle\sum_{i=1}^{n} f_i v_i \\ \text{under villkoret } & \textstyle\sum_{i=1}^{n} f_i w_i \leq C \\ & 0 \leq f_i \leq 1, \quad i = 1, \ldots, n \end{aligned}$$

Här har vi tre olika idéer:

- Vi lägger så mycket vi kan av det *dyraste objektet* i kappsäcken, därefter så mycket

vi kan av det *näst dyraste* och så vidare tills kappsäcken är fylld.

- Vi lägger till så mycket vi kan av det *minsta objektet*. Därefter så mycket vi kan, av det *näst minsta* och så vidare tills kappsäcken är fylld.
- Vi räknar ut och sorterar objekten efter *volymspris* (till exempel *kr/l*) $v_i/w_i$. I tur och ordning lägger vi sedan till lasten så mycket vi kan av det objekt som har *högst volympris*. Därefter adderas så mycket som möjligt av objektet med *näst högsta volympriset* och så vidare tills ryggsäcken är full.

Endast en av dessa ger *alltid* den korrekta lösningen. För de båda första förslagen kan vi finna exempel som visar att idéerna inte leder till den optimala lösningen.

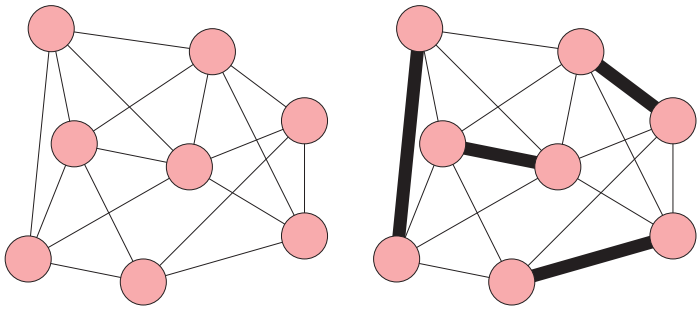
**Algorithm 9.6.1:** KNAPSACK($V, W, C, F$)

```
for i ← 1 to n
  do F[i] ← 0
återstår ← C
i ← 1
if W[1] ≤ C
  then fårplats ← true
  else fårplats ← false
while fårplats and i ≤ n
      ⎧ F[i] ← 1
      ⎪ återstår ← återstår − W[i]
      ⎪ i ← i + 1
  do  ⎪ if W[i] ≤ återstår
      ⎪   then fårplats ← true
      ⎩   else fårplats ← false
if i ≤ n
  then F[i] ← återstår/W[i]
```

$V[1 \ldots n]$ är en array som innehåller objektens *värden*. $W[1 \ldots n]$ är en array som innehåller objektens *volymer*. Både $V$ och $W$ är sorterade så att $V[1]/W[1] \geq V[2]/W[2] \geq \ldots \geq V[n]/W[n]$. $C$ är ryggsäckens volym. $F[1 \ldots n]$ är algoritmens utdata, som innehåller hur stora andelar $0 \leq F[i] \leq 1$ av varje objekt som tagits med.

## 9.7 Perfekt matchning

Till vänster i figur 9.11 ser vi en oriktad graf med 8 noder. Till höger ser vi samma graf där noderna parats samman två och två. För att genomföra detta har vi valt ut lämpliga bågar. Detta arrangemang kallas för en *matchning*. Eftersom samtliga noder ingår i matchningen



Figur 9.11:

kallas den *perfekt matchning*. För en godtycklig graf är det inte alltid möjligt att finna en perfekt matchning. Däremot finns det alltid en *maximal matchning*. Det vill säga en matchning där det ingår så många noder som möjligt.

Normalt är det, för en godtycklig graf, ett svårt problem att finna en maximal matchning (som likväl kan vara en perfekt). Här ska vi bestämma oss för grafer där antalet noder $|V| = 2n$ (alltså ett jämnt antal noder) och där varje nod $v$ har ett gradtal $deg(v) \geq n$ . (Gradtalet hos en nod $v$ är det antal bågar som utgår från den noden). Vi ser att grafen i figur 9.11 uppfyller dessa villkor och påstår, utan att bevisa, att det alltid finns en perfekt matchning för denna typ av grafer och att det finns en glupsk algoritm som finner denna matchning.

Så här kan vi beskriva den: Vi startar med att välja ut bågar, sådana att de binder ihop noder som tidigare inte varit inblandade. De berörda noderna och bågarna bildar en ny, från början tom, graf $M$. Valet av bågar sker i godtycklig ordning. Har vi riktig tur löser vi problemet redan i denna fas.

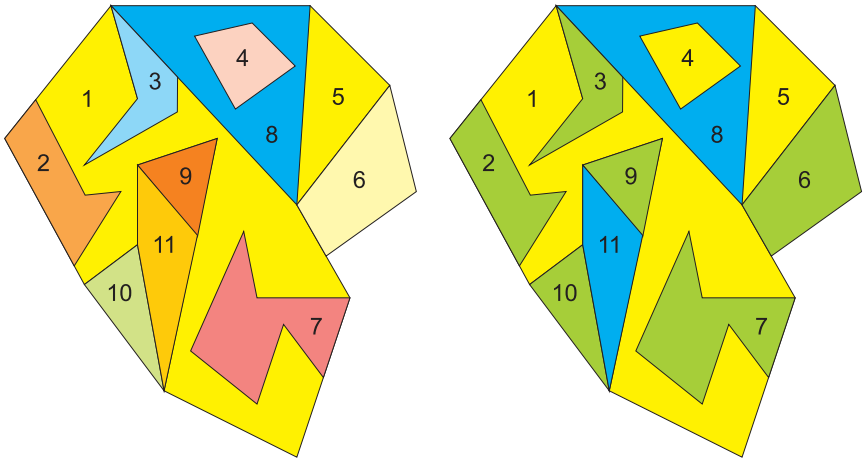
Men anta nu att vi kommit en bit på vägen och inte kan lägga till någon mer båge. Då finns det åtminstone två noder $v_1$ och $v_2$, som inte tillhör $M$. Dessa två noder har tillsammans $\geq 2n$ bågar, som alla går till en nod som finns i $M$. $(v_1, v_2)$ finns ju inte. Eftersom antalet bågar i $M$ är $< n$ och antalet bågar som utgår från $v_1$ och $v_2$ är $\geq 2n$, så måste det finnas en båge $(u_1, u_2)$ i $M$, sådan att det i $G$ finns $(u_1, v_1), (u_1, v_2)$ och $(u_2, v_1)$. Om vi då tar bort bågen $(u_1, u_2)$ och ersätter den med $(u_1, v_2)$ och $(u_2, v_1)$ har vi kommit närmare lösningen.

UPPGIFT 9.5

**Perfekt matchning** Översätt resonemanget i avsnittet om perfekt matchning, till ett program som löser problemet med en glupsk algoritm. På filerna `matchning1.txt` och `matchning2.txt` finns data som beskriver grafer av den speciella typ som krävs för att algoritmen ska fungera. Första raden anger antalet noder. Andra raden antalet bågar. På efterföljande rader finns två tal som beskriver en båge mellan två noder.

### Att färglägga en karta

Att färglägga en karta med $n$ länder så att inga läder med gemensam gräns får samma färg, kan klaras av med hjälp av en glupsk algoritm, utan att för den skull finna det minsta antalet färger som behövs för att utföra arbetet. Här har vi två kartor




I figuren ser vi till vänster en karta med överdrivet många färger. Till höger har vi klarat av målandet med endast tre färger, vilket man enkelt kan se, är det minsta antal som behövs.

Hur ska då datastrukturen se ut? Givet är en fil `kartan.txt` som innehåller uppgifter om alla gränder. Filen inleds med ett tal som anger antalet länder $n$, därefter ett tal som anger antalet gränser $m$. På efterföljande $m$ rader innehåller alla två tal. Numren på två länder mellan vilken det går en gräns

Här följer algoritmen:

```
#include <stdio.h>
#define MAX 20
void mala(int antal,int granser[][MAX],int karta[],int lander[]){
  int farger[MAX]={0},i,j;
  for(i=1;i<=antal;i++){
    for(j=1;j<=antal;j++)
      farger[j]=0;
    for(j=1;j<=antal;j++)
      if(granser[i][j] && lander[j]!=0)
        farger[lander[j]]=1;
    j=1;
    while(farger[j])
      j++;
    lander[i]=j;
  }
}
```

```
int main(void){
  int granser[MAX][MAX]={0},karta[MAX]={0};
  int antal,i,f,t,lander[MAX]={0},nlander,ngranser;
  FILE *fil;
  fil=fopen("kartan.txt","rt");
  fscanf(fil,"%d",&nlander);
  fscanf(fil,"%d",&ngranser);
  for(i=0;i<ngranser;i++){
    fscanf(fil,"%d %d",&f,&t);
    granser[f][t]=1;
    granser[t][f]=1;
  }
  fclose(fil);
  mala(nlander,granser,karta,lander);
  for(i=1;i<=nlander;i++)
    printf("Land %d Farg %d\n",i,lander[i]);
}
```

### Egyptiska bråk

Alla rationella tal kan skrivas $\frac{a}{b}$, där $a, b \in Z$. Vi koncentrerar oss här på rationella tal $0 < \frac{a}{b} < 1$ och påstår att de alla kan skrivas som en summa av bråk med täljaren 1. Till exempel:

$$\frac{7}{11} = \frac{1}{2} + \frac{1}{8} + \frac{1}{88}$$

Utvecklingen kan skapas med hjälp av en *greedy* algoritm. Observera dock att denna algoritm varken ger det minsta antalet termer eller minimerar nämnarens storlek.

Ett problem är att nämnaren kan bli så stor att den inte ryms i ett `int` eller ens `long long int` även då nämnre och täljare är relativt små. Till exempel kan

$$\frac{5}{121} = \frac{1}{25} + \frac{1}{757} + \frac{1}{763309} + \frac{1}{873960180913} + \frac{1}{1527612795642093418846225}$$

som är resultatet från vår greedy algoritm, också skrivas

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

```
#include <stdio.h>
#define T 7
#define N 11

void findnext(long long t,long long n,long long *n2){
  *n2=n/t;
  while(*n2*t<n)
    (*n2)++;
}

void swap(long long *a,long long *b){
  int tmp=*a;
  *a=*b;
  *b=tmp;
}

long long gcd(long long a,long long b){
  if(b>a)
    swap(&a,&b);
  if (b==0){
    return a;
  }
  else{
    return gcd(b,a%b);
  }
}
```

```
void sub(long long *t,long long *n,long long n2){
  long long t3,n3,g;
  t3=*t*n2-*n;
  n3=*n*n2;
  g=gcd(t3,n3);
  *t=t3/g;
  *n=n3/g;
}

int main(void){
  long long t=T,n=N,n2;
  while(t!=0){
    findnext(t,n,&n2);
    printf("1/%lld+",n2);
    sub(&t,&n,n2);
  }
}
```

`long long` klarar max 9 223 372 036 854 775 807, `unsigned`, 18 446 744 073 709 551 615, som skulle kunna hjälpa oss en liten bit till. Uttalas 18 triljoner 446 biljarder 744 biljoner 73 miljarder 709 miljoner 551 tusen 615

UPPGIFT 9.6

**Program för Knapsack**. Skriv ett program som implementerar *Knapsack* efter algoritmen ovan. Programmets indata ska läsas från fil. Filen ska inledas med $C$, ryggsäckens volym. På nästa rad anges antal objekt $n \leq 50$. Därefter följer $n$ rader där varje rad innehåller två positiva heltal $v_i$ och $w_i$

Utdata är en lista, som innehåller ordningsnumret för de objekt, som ingår i den fullpackade ryggsäcken. För det sista objektet anges dessutom vilken andel som ingår, ett reellt tal.

UPPGIFT 9.7

**Optimal tape-storage problem**. Givet i denna uppgift är $n$ filer $f_1 \ldots f_n$. Till var och en av filerna är storleken (i byte) $s_i \ldots s_n$ given. Dessa filer ska läsas till ett magnetband på ett tidsmässigt så ekonomiskt sätt som möjligt.

För att läsa en viss fil $f_i$ från bandet (vi antar att bandet är återspolat från start) måste man först läsa förbi eventuella filer som ligger före. Den totala ”kostnaden” blir därför summan av alla filstorlekar före och inklusive den fil som ska läsas.

Konstruera en *glupsk algoritm* som föreslår i vilken ordning filerna ska skrivas på bandet för att framtida läsningar ska bli optimal.

Föreslå också en algoritm för att dela upp filerna på två band. Tiden mäts från det att rätt band sitter i bandstationen.

UPPGIFT 9.8

**Färjan** En färja som kan ta $n$ bilar behöver $t$ minuter för att korsa floden, låta bilarna köra av och återvända till färjläget för ny pålastning. Filen `bilar.txt` inleds tre tal $n, t, b$, där $b$ anger hur många bilar som kommer att anlända. Därefter följer $b$ tider, heltal $\leq 1000$, i stigande ordning, angivna i hela minuter, som anger då bilarna anländer.

Kan du komma på en glupsk algoritm som beräknar den den tid, då arbetet tidigast kan vara avklarat?

UPPGIFT 9.9

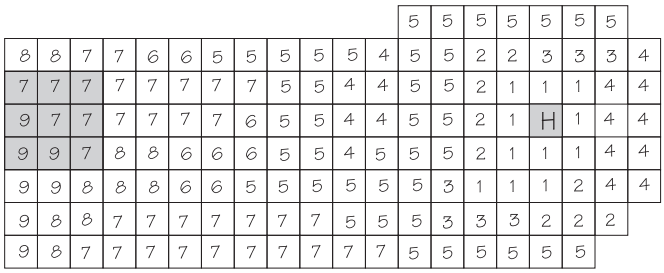
**Schemaläggning**. På ett universitet ska ett antal *föreläsningar* hållas. Varje föreläsning har en *starttid* och en *sluttid*. Man vill nu placera så många som möjligt i Stora Hörsalen. Konstruera och implementera en glupsk algoritm som löser problemet.

Indata finns på en fil, som inleds med ett tal $n \leq 50$ som anger *antalet föreläsningar*. Därefter följer $n$ rader där varje rad innehåller två heltal *starttid* och *sluttid*, där sluttiden alltid är större än starttiden. Ingen speciell ordning mellan raderna kan förutsättas.

Utdata är ordningsnumren (motsvarande i indatafilen) av de utvalda föreläsningarna i den ordning de ska hållas med angivande av start- och sluttid.

Observera alltså att det är *antalet föreläsningar* som ska maximeras – inte den totala beläggningstiden. Använd datafilen *schema.dat* för test.

UPPGIFT 9.10
**Golfhålet**




Figur 9.12:

Figuren visar en del av en golfbana! Utslaget sker från någon av de nio rutorna i den grå kvadraten till vänster (tee). Från den ruta man väljer kan bollen sedan slås i åtta olika riktningar (N, NÖ, Ö, SÖ, S, SV, V, NV), så många rutor som talet i utgångsrutan anger. Detta under förutsättning att bollen hamnar inom banans geometri.

Från den ruta i vilken bollen stannar går spelet sedan vidare i en av de åtta riktningarna med en slaglängd som motsvarar talet i den rutan. Målet är att med så få slag som möjligt nå *hålet* märkt H i figuren.

Skriv ett program som tar emot data om banans geometri och som enligt reglerna ovan bestämmer det minsta antal slag som krävs för att komma från *tee* till *hål*

**Indata:** Programmet startar med att fråga efter indatafilens namn:

```
Filnamn: uppg6.dat
```

Filen inleds med en rad innehållande 6 tal: Banans *bredd* (antal rader $b, 3 \leq b \leq 40$), banans *längd* (antal kolumner $l, 4 \leq l \leq 60$), *rad* och *kolumn* övre vänstra hörnet hos tee (alltid $3 \times 3$) och *rad* och *kolumn* för hålets placering. Följande $b$ rader innehåller $l$ siffror 0...9 som anger slaglängden från denna ruta. För vårt exempel:

```
8 20 3 1 4 17
00000000000055555550
88776655555455223334
77777777554455211144
97777776554455210144
99788666554555211144
99888665555553111244
98877777775553332220
98777777777755555500
```

**Utdata:** Programmet skriver ut det minsta antalet slag som behövs för att nå hålet

```
Hålet kan klaras av på 3 slag
```

UPPGIFT 9.11

**Sammanslagning av valdistrikt** I landet finns $n$ valdistrikt. Kartan visar hur många röster det finns i varje distrikt. Man vill nu slå samman dem till $m < n$ distrikt, så att antalet röster blir så jämnt fördelat som möjligt. Ett krav är dock att var och en av de $m$ större distrikten är sammanhängande. Föreslå en glupsk algoritm!