Kapitel 11

Dynamisk Programmering

När man väl har förstått idén bakom *dynamisk programmering* så är tekniken ganska enkel för att lösa problem och skapa algoritmer. Men det hindrar dock inte att det hela från början ter sig närmast magiskt. Det bästa sättet att lära sig hantera *dynamisk programmering* torde vara att studera ett antal exempel och det ska vi göra här.

11.1 Fibonacci's talföljd

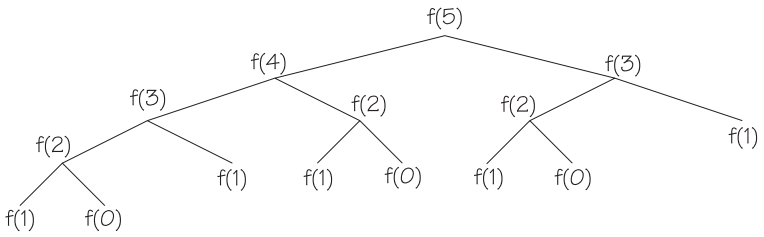
Vi återknyter här bekantskapen med *Fibonacci's talföljd* från första föreläsningen där vi studerade följande rekursiva funktion.

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

Vi konstaterade där att, för att bestämma f_n anropades funktionen `fib` total $2 \cdot f_n - 1$ gånger. För till exempel $f_{33} = 5702887$ kommer `fib` att anropas 11 405 773 gånger, vilket är oacceptabelt, speciellt med tanke på att vi känner till formeln

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

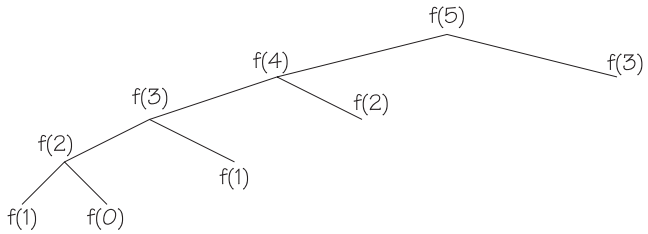
Tittar vi på ett träd över anropen så förstår vi varför anropen blir så många. Vi ser också att "samma jobb" görs flera gånger.



Figur 11.1: *För* f_5 *görs* 15 *anrop där bland annat* f_2 *beräknas tre gånger.*

Tre gånger i trädet i figur 11.1 beräknas f_2. En idé med *dynamisk programmering* är att lagra beräknade delresultat i en tabell. Vid varje anrop av funktionen kan vi då "slå upp" om värdet redan finns noterat. Om så är fallet har vi direkt resultatet.

På så sätt kan vi "klippa i trädet" och vårt anropsträd kommer i stället att se ut så här.



Figur 11.2: *Varje löv i trädet har nu ett unikt värde*

Vi hamnar nu på 9 anrop i stället för 15. Inte så stor skillnad kan tyckas men för f_{33} behövs nu bara 65 anrop i stället för tidigare 11 405 773.

Vad vi behöver är alltså en tabell t med lika många celler som det största n för vilket vi vill bestämma f_n. Vid uppstart initierar vi tabellen med $t[0] = 1$ och $t[1] = 1$ (de två kända värdena) och med -1 för resten.

Algorithm 11.1.1: FIBDYN(n)

if $\text{tab}[n] \geq 0$
  then return $(\text{tab}[n])$
  else
$f1 \leftarrow \text{FIBDYN}(n-1)$
$\text{tab}[n-1] \leftarrow f1$
$f2 \leftarrow \text{FIBDYN}(n-2)$
$\text{tab}[n-2] \leftarrow f2$
return $(f1 + f2)$;

Det finns två typer av dynamisk programmering DP:

- Vi bygger upp lösningar av delproblem från mindre till större, *bottom–up*.
- Eller så kan vi spara resultat i en tabell som vi slår upp i efter behov, *top–down* med *memoization*

Exemplet ovan med Fibonacci's talföljd är ett exempel på *top–down*. Detta exempel på samma talföljd kan klassificeras som *bottom–up*.

```
1 int fib3(int n){
2   int f1=1,f2=1,f3=1,i;
3   for(i=3;i<=n;i++){
4     f3=f1+f2;
5     f1=f2;
6     f2=f3;
7   }
8   return f3;
9 }
```

H Den enklaste och snabbaste av alla sätt att ta renda på f_n om vi bortser från formeln. Helt klart handlar det här om bottom–up.

11.2 Myntväxling

I samband med *glupska algoritmer*, bekantade vi oss med ett exempel, som gick ut på att minimera antalet sedlar och mynt för att räkna upp ett givet belopp.

Vi nämnde då att den glupska algoritmen vi där använde inte fungerade för alla uppsättningar av mynt och efterlyste då ett alternativt sätt att lösa problemet. Till exempel med följande *söndra och härska*-algoritm.

Algorithm 11.2.1: CHANGE(belopp)

```
min ← belopp
for i ← 1 to antal
  do { if mynt[i] = belopp
       then return (1)
for i ← 1 to belopp/2
  do { n ← CHANGE(i) + CHANGE(belopp − i)
       if n < min
         then min ← n
return (min)
```

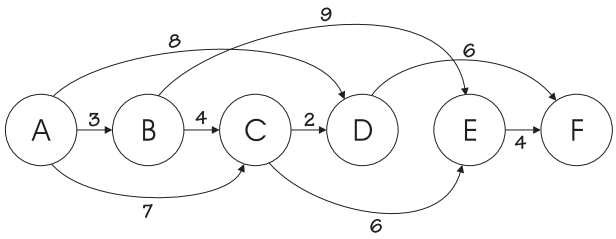
Indata är `mynt`, som är en array som håller `antal` valörer. Algoritmen fungerar, men den tar alldeles för lång tid på sig. Vi förstår att många anrop upprepas under arbetets gång och tankarna går därför till *dynamisk programmering*. Vi förstärker därför algoritmen:

Algorithm 11.2.2: CHANGE2(belopp)

if lista[belopp] $\neq 0$
  then return (lista[belopp])
min $\leftarrow$ belopp
for i $\leftarrow$ 1 **to** antal
  do { **if** mynt[i] = belopp
    then return (1)
for i $\leftarrow$ 1 **to** belopp/2
  do { n $\leftarrow$ CHANGE2(i) + CHANGE2(belopp $-$ i)
    if n $<$ min
      then min $\leftarrow$ n
lista[belopp] $\leftarrow$ min
return (min)

Här tillkommer en global array, `lista`, som har lika många element som det största belopp vi tänker räkna upp. Arrayen nollställer vi från början. Varje gång algoritmen returnerar ett funnet minimum uppdateras dessutom `lista`, för kommande behov. Vi räknar antalet anrop för de två algoritmerna med `belopp`=29 och `mynt`={1, 5, 10, 20, 50, 100, 500, 1000} För CHANGE görs 60 911 899 anrop och för CHANGE2 görs 387. Så små skillnader kan det vara mellan en oanvändbar och en snabb algoritm!

11.3 Riktad graf utan cykler



Figur 11.3:

I figur 11.3 ser vi en *riktad graf utan cykler*, en så kallad *DAG*. (directed acyclic graph). En graf av den typen kan ordnas utefter en linje så att alla bågar går från vänster till höger. Detta är ett viktigt sätt att betrakta problem som går att lösa med dynamisk programmering.

Om uppgiften kopplad till grafen består i att finna den kortaste vägen från A till F och till exempel betraktar nod E så kan det minsta avståndet till E skrivas

$$\text{mindist}(E) = \min(\text{mindist}(B), \text{mindist}(C))$$

Börjar vi i A, som har mindist = 0 kan vi successivt hitta minsta avståndet till B, C, D, E, F genom formeln ovan. Om vi har alla avstånden lagrade i `a[6][6]` och placerar ∞ i de celler i matrisen som saknar direkt förbindelse. Så löser vi detta problem med koden:

```
1  int a[6][6]={{1000,   3,   7,   8,1000,1000},
2              {1000,1000,   4,1000,   9,1000},
3              {1000,1000,1000,   2,   6,1000},
4              {1000,1000,1000,1000,1000,   6},
5              {1000,1000,1000,1000,1000,   4},
6              {1000,1000,1000,1000,1000,1000}};
7  int i,j;
8  int d[6]={0,1000,1000,1000,1000,1000};
9  for(i=0;i<6;i++)
10   for(j=0;j<i;j++)
11     d[i]=min(d[i],d[j]+a[j][i]);
```

1000 räcker gott och väl som ”oändligheten” i detta exempel. Man får akta sig för att använda `INT_MAX`, eftersom det kommer att adderas ett tal till detta och då uppstår overflow.

11.4 Myntväxling för sista gången

Med denna metod löser vi vårt, numera gamla, myntväxlingsproblem med följande idé. Vi tänker oss mynten 1, 3 och 5 i en okänd valuta och vill ta reda på det minsta antalet mynt för belopp från 1 till 18.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	2	1	2	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	2	1	2	0	2	0	0	0	0	0	0	0	0	0	0
1	2	1	2	1	2	0	2	0	2	0	0	0	0	0	0	0	0
1	2	1	2	1	2	3	2	3	2	3	0	0	0	0	0	0	0
1	2	1	2	1	2	3	2	3	2	3	0	3	0	0	0	0	0
1	2	1	2	1	2	3	2	3	2	3	0	3	0	3	0	0	0
1	2	1	2	1	2	3	2	3	2	3	4	3	4	3	4	0	0
1	2	1	2	1	2	3	2	3	2	3	4	3	4	3	4	0	4
1	2	1	2	1	2	3	2	3	2	3	4	3	4	3	4	5	4

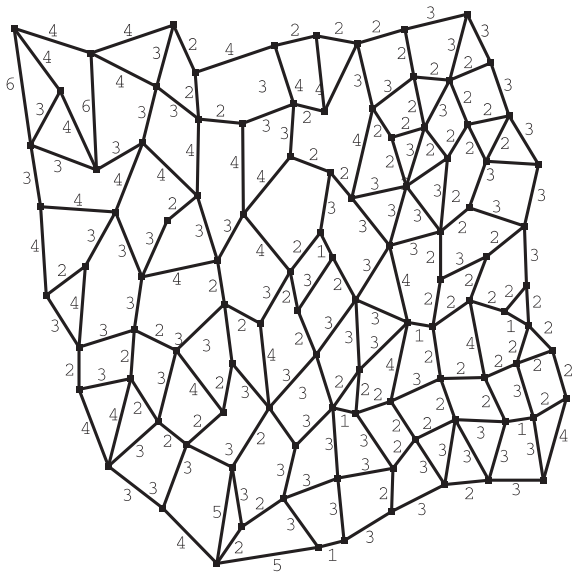
Vi startar i tredje raden med att placera ut ett mynt av varje valör. I fjärde raden lägger vi till valören 1 till alla belopp som kräver ett mynt och skriver in 2 på motsvarande plats. I

nästa rad lägger vi till valören 3 till alla belopp som kräver ett mynt. Om vi hamnar på 0 skriver vi in 2 där. Till sist, i den avdelningen, lägger vi till valören 5 till alla belopp som kräver ett mynt. I nästa avdelning lägger vi till valörerna $1, 3, 5$ till belopp som kräver två mynt och om vi då hamnar på 0 skriver vi in 3 i den cellen.

Till sist har alla 0:or försvunnit och problemet är löst.

UPPGIFT 11.1
Myntväxling för sista gången. Lös problemet från kapitel 10 om myntväxling av svenska mynt med den ovan diskuterade tekniken och jämför resultaten.

11.5 Floyd-Warshall

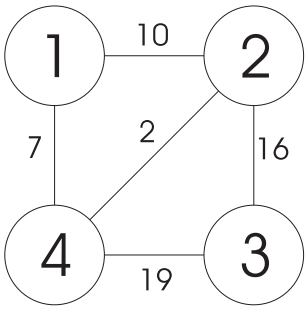


Figur 11.4:

Algoritmen *Floyd-Warshall* är en konkurrent till *Dijktstra's* algoritm. Den bestämmer kortaste avståndet till *alla* par av noder i en viktad graf med `n` noder. I `dist` har vi lagrat alla vikter (avstånd) mellan närliggande noder. Detta är all kod vi behöver:

```
1 void floyd(int dist[][101],int n){
2   int i,j,k;
3     for(k=1;k<=n;k++)
4       for(i=1;i<=n;i++ )
5         for(j=1;j<=n;j++)
6           dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]);
7 }
```

Vi genomför ett exempel med endast fyra noder efter figur 11.5, som leder till efterföljande avståndsmatris.



Figur 11.5:

	1	2	3	4
1	0	10	∞	7
2	10	0	16	2
3	∞	16	0	19
4	7	2	19	0

Vi förstår att denna algoritm arbetar efter bottom–up. Här några av värden hos i,j och k då distansmatrisen får ett nytt värde.

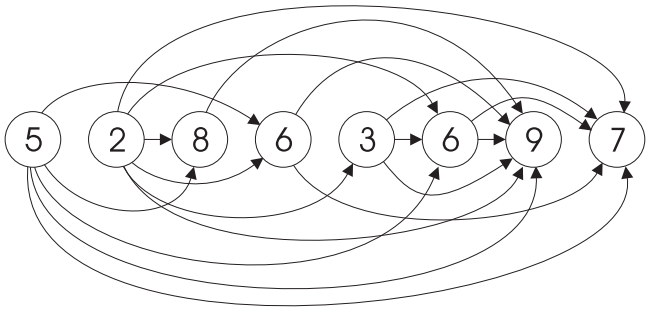
k	**i**	**j**	**resultat**
2	3	4	18
2	4	3	18
4	1	2	9
4	1	3	25
4	2	1	9
4	3	1	25

Resultatet blir till slut

	1	2	3	4
1	0	9	25	7
2	9	0	16	2
3	25	16	0	18
4	7	2	18	0

Vi förstår att detta är en bottom-up lösning.

11.6 Longest increasing subsequence



Figur 11.6:

Ur ett given följd av tal $a_1, a_2 \ldots a_n$, ska vi välja tal i följden, så att $a_{k_1} < a_{k_2} < \ldots < a_{k_m}$ och så att m blir så stort som möjligt. Om talen är $5, 2, 8, 6, 3, 6, 9, 7$, har vi i figur 11.6 markerat alla sådana följder. Den längsta innehåller de fyra talen $2, 3, 6, 9$.

Problemet kan lösas med följande kod:

```
1  int lcs(int a[],int N){
2    int *best,*prev,i,j,max=0;
3    best=(int*)malloc(sizeof(int)*N);
4    prev=(int*)malloc(sizeof(int)*N);
5    for(i=0;i<N;i++) {best[i] = 1; prev[i] = i;}
6    for(i=1;i<N;i++)
7      for(j=0;j<i;j++)
8        if(a[i]>a[j] && best[i]<best[j]+1)
9          best[i]=best[j]+1,prev[i]=j;
10   for(i=0;i<N;i++)
11     if(max<best[i])
12       max=best[i];
13   free(best);
14   free(prev);
15   return max;
16 }
```

H Vi använder oss av dynamiska arrayer, som vi under exekveringen kan skapa i lämplig storlek. Arrayen `best` används endast för att senare kunna få fram talen i den längsta sekvensen.

5 De två arrayerna initieras.

6-9 I yttre loopen går man igenom ett tal a_i i taget i sekvensen. I den inre tittar man på de tal a_j, som står till vänster om det aktuella talet. För alla tal $a_j < a_i$ tar man reda på, vilket av talen, som har störst värde i `best`[j], vilket är samma sak som den

längsta stigande sekvensen. `best[j] + 1` blir värdet för `best[i]` eftersom sekvensen har utökats med ett tal. I `prev[i]` lagrar vi `j` index till detta tal.

11.7 Knapsack II

Även i detta exempel återkommer vi till en gammal bekant, men nu i ny skepnad. Vi har som tidigare en ryggsäck med en given volym c, till detta n olika objekt $b_i, 1 \leq i \leq n$. Varje objekt har ett värde v_i och en volym w_i. Skillnaden är att

- det finns ett obegränsat antal av varje objekt b_i.
- man nu inte kan dela objekten, som man fick göra i förra exemplet.
- ryggsäcken inte behöver bli full.

Målet är fortfarande att ge ryggsäcken ett så dyrbart innehåll som möjligt. Vi studerar ett konkret exempel (se figur 11.7).

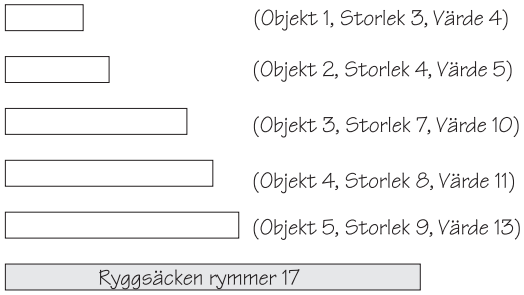
För detta exempel kan vi ställa upp en optimeringsfunktion

$$\begin{array}{ll} \text{maximera} & a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4 + a_5 \cdot v_5 \\ \text{så att} & a_1 \cdot w_1 + a_2 \cdot w_2 + a_3 \cdot w_3 + a_4 \cdot w_4 + a_5 \cdot w_5 \leq c \end{array}$$

Eller i klartext

$$\begin{array}{ll} \text{maximera} & a_1 \cdot 4 + a_2 \cdot 5 + a_3 \cdot 10 + a_4 \cdot 11 + a_5 \cdot 13 \\ \text{så att} & a_1 \cdot 3 + a_2 \cdot 4 + a_3 \cdot 7 + a_4 \cdot 8 + a_5 \cdot 9 \leq 17 \end{array}$$

$a_i, 1 \leq i \leq 5$ är antalet objekt av de fem olika typerna som vi ska välja.



Figur 11.7: *Av varje objekt finns det ett obegränsat antal. Hur värdefull kan innehållet i ryggsäcken bli?*

Ett enkelt och osofistikerat sätt att lösa det här problemet är en funktion med fem forloopar, där varje loop behandlar ett objekt. Loopvariabeln går från 0 till det *maximalt möjliga antalet* av detta objekt.

Minsta antal	0	0	0	0	0
Största antal	5	4	2	2	1

Den inre delen av dessa nästlade loopar, den där vi tar reda på om ”rekordet har slagits”, kommer att exekveras $6 \cdot 5 \cdot 3 \cdot 3 \cdot 2 = 540$ gånger.

För denna metod kan vi bestämma den övre gränsen för antalet varv till

$$T(n) = \left(\left\lfloor \frac{c}{\min(v)} \right\rfloor + 1 \right)^n$$

Det betyder att om vi har en ryggsäck med $c = 100$ och $n = 10$ objekt med minsta volymen $v_{\min} = 3$, så är $T(10) \approx 2 \cdot 10^{15}$. Då är inte längre den här metoden möjlig att använda.

Vad kan då *dynamisk programmering* göra för oss? Indata består av två arrayer `size` och `value`, data för objekten. Sedan behöver vi en tabell, `cost`. Tabellen ska ha plats för `c` värden och nollställs före start. I nedanstående algoritm kommer vi att bestämma den värdefullaste packningen för *alla* storlekar på ryggsäckar `1...c`. Tabellen `best` håller reda på vilka objekt som plockats ned i ryggsäcken.

Algorithm 11.7.1: KNAPSACK( )

```
for j ← 1 to n
do { for i ← 1 to c
     do { if i − size[j] ≥ 0
          then { if cost[i] < cost[i − size[j]] + value[j]
                 then { cost[i] ← cost[i − size[j]] + value[j]
                        best[i] = j
```

Vi följer exekveringen för vårt exempel och uppdaterar hela tiden `cost`.

volym	**värde**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
3	4	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
4	5	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
7	10	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
8	11	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
9	13	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24

- Den yttre loopen går igenom samtliga objekt. Under första varvet är inget annat objekt än b_1 inblandat. Vilket betyder att vi så långt, kommer att få en lösning för detta enda objekt. Hur många vi får ner i ryggsäckar med volymen från 1 till 17 ser vi i första raden i tabellen.

- Under andra varvet behandlas b_2. När den andra loopen nått fram till storleken 4 hos ryggsäcken ryms för första gången ett objekt b_2. Det bästa värdet vi tidigare hade för denna storlek var 4. Då måste det vara bättre att placera en b_2 i säcken och därmed få värdet 5. Detta gäller även för storleken 5. Däremot kan en b_2 inte konkurrera med 2 stycken b_1 för storleken 6. Andra raden visar de bästa värdena med två objekt inblandade.

- Tabellen fylls alltså på radvis. För att fylla i en ny ruta tänker man så här: ”Bästa värdet för den här storleken är just nu `cost`[i]. Värdet `cost[i-size[j]]` är det bästa värdet (hittills) för en ryggsäck med storleken `i-size[j]`. Om vi nu i stället lägger till ett objekt b_j, som ju kommer att få plats, får vi då ett bättre värde för `cost[i]`? Om svaret är ”ja” på den frågan uppdaterar vi `cost[i]` med detta nya värde.

Observera att det till vårt exempel finns två lösningar $\{1,3,3\}$ och $\{4,5\}$. Av allt att döma kommer denna algoritm att kunna exekveras på tiden $O(c \cdot n)$, vilket för vårt exempel blir $17 \cdot 5 = 85$. För exemplet där $c = 100$ och $n = 10$ kommer det hela att gå ungefär 10^{12} gånger snabbare om vi jämför med vårt naiva förslag i inledningen, som alltså skulle innehålla 10 for-loopar.

UPPGIFT 11.2
Implementera Knapsack och använd `best` för att skriva ut vad ryggsäcken innehåller. Ta också reda på om det är viktigt att objekten är sorterade efter stigande volym?

Låt programmet fråga efter kappsäckens storlek och antalet objekt vars värde och volym sedan slumpas fram.

11.8 Matrismultiplikation

Det är med glädje jag nu presenterar ett exempel hämtat från den *linjära algebran*. Det handlar om matrismultiplikation som vi först repeterar. För två matriser $A(n \times m)$ och $B(p \times q)$ kan produkten $C = AB$ endast bildas om $m = p$. Matrisen C får dimensionerna $C(n \times q)$. Om från $A(3 \times 2)$ och $B(2 \times 2)$ vi vill bilda AB går det i detalj till så här:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} \end{pmatrix}$$

Det behövs här 12 multiplikationer för att nå resultatet. Hur många multiplikationer behövs det då för att bilda $A(12 \times 18)B(18 \times 23)$ eller mer generellt $A(n \times m)B(m \times p)$? För den första multiplikationen behövs $12 \cdot 18 \cdot 23 = 4968$ multiplikationer och den generella formeln är nmp.

Eftersom den *associativa* lagen gäller vid matris multiplikation ger $(AB)C$ samma resultat som $A(BC)$ och $((AB)(CD))$ ger samma resultat som $(A(B(CD)))$. Men hur är det med antalet multiplikationer vid olika val av multiplikationsordning? Vi studerar följande

exempel.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} \begin{pmatrix} d_{11} & d_{12} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

Vi skulle lika gärna kunna skriva

$$(4 \times 2)(2 \times 3)(3 \times 1)(1 \times 2)(2 \times 2)(2 \times 3)$$

eller till och med $r = [4, 2, 3, 1, 2, 2, 3]$ utan att tappa någon information. På hur många sätt kan nu slutprodukten nås? På inte mindre är 42 olika sätt som vi får genom formeln för *Catalanska tal.*

$$C_n = \frac{1}{n}\binom{2(n-1)}{n-1}$$

där n står för antalet matriser i uttrycket. Bland dessa

$$(((((AB)C)D)E)F) \qquad ((((AB)(CD))E)F) \qquad ((AB)(((CD)E)F))$$

Hur många multiplikationer kräver dessa tre olika sätt? 84, 94 respektive 96. Med andra ord vill man spara in multiplikationer så är det väsentligt i vilken ordning matriserna multipliceras samman.

Vårt problem är nu förstås att hitta det mest ekonomiska schemat för arbetet. Det bästa för detta exempel är för övrigt $((A(BC))((DE)F))$ för vilket det endast krävs 36 multiplikationer.

Vi tillgriper *dynamisk programmering* och inför följande algoritm.

Algorithm 11.8.1: MATMULT( )

```
for i ← 1 to N
  do { for j ← i + 1 to N
       do cost[i, j] ← MAXINT
for i ← 1 to N
  do cost[i, i] ← 0
for j ← 1 to N − 1
  do { for i ← 1 to N − j
       do { for k ← i + 1 to i + j
            do { t ← cost[i, k − 1] + cost[k, i + j]+
                 +r[i] · r[k] · r[i + j + 1]
                 if t < cost[i, i + j]
                   then { cost[i, i + j] ← t
                          best[i, i + j] ← k
```

	B	C	D	E	F
A	24	14	22	26	36
B		6	10	14	22
C			6	10	19
D				4	10
E					12

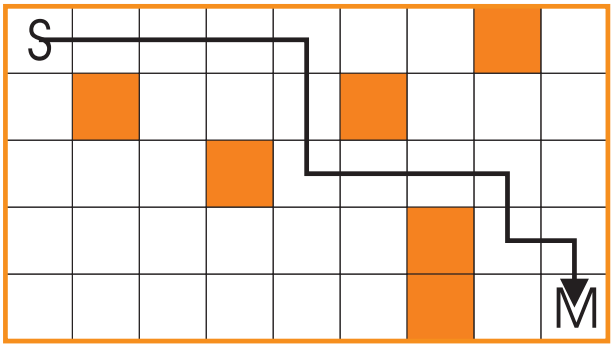
Algoritmen bestämmer först antalet multiplikationer för de fem kombinationerna AB, BC, CD, DE, EF. Dessa kan ju endast utföras på ett sätt. Antalet lagras i huvuddiagonalen i tabellen ovan.

Därefter bestäms alla möjliga kombinationer av tre matriser ABC, BCD, CDE, DEF. För att bestämma till exempel till ABC använder man sig av informationen från AB och BC. Algoritmen väljer den bästa av (AB)C och A(BC). Det vill säga att multiplicera det redan uträknade AB med C eller att multiplicera A med BC. Värdena från dessa beräkningar lagras i nästa diagonal.

I nästa steg bearbetas ABCD, BCDE och CDEF och så vidare till sista steget, ABCDEF. Slutresultatet hamnar högst upp i högra hörnet av tabellen, 36.

Med hjälp av denna lilla funktion får vi så till sist en trevlig utskrift av svaret:

```
1  void order(int i,int j){
2    if(i==j)
3      printf("%c",64+i);
4    else{
5      printf("(");
6      order(i,best[i][j]-1);
7      printf("*");
8      order(best[i][j],j);
9      printf(")");
10   }
11 }
```



Figur 11.8:

UPPGIFT 11.3

Kalles väg till skolan. När Kalle ska gå till skolan (rutan märkt S i figuren), startar han promenaden i rutan märkt H. Eftersom Kalle vill ha omväxling försöker han i det längsta att varje morgon ta en ny väg till skolan (se exempel i figuren på en av hans möjliga vägar). Han går aldrig längre än han behöver och väljer därför bara att gå i östlig eller sydlig riktning (sydöstlig riktning är omöjlig). På vägen kan finnas oframkomliga gator (rutor med mönster av tegel).

Frågan är nu: Hur många olika vägar det finns för Kalle att ta sig till skolan?

Skriv ett program som först frågar efter stadens storlek (antal rader och antal kolumner, dessa tal kan vara olika men är ≤ 100). I nästa fråga ska man ange hur många oframkomliga vägar staden innehåller och därefter var, var och en av dessa är belägna, genom att först ange rutans rad och därefter dess kolumn.

Programmet ska sedan bestämma antalet olika vägar mellan H och S där varje steg måste vara en förflyttning i antingen östlig eller sydlig riktning och där man inte får beträda oframkomliga rutor.

Indata: Den rektangulära stadens storlek genom antalet rader och kolumner. Antalet oframkomliga gator och var dessa är belägna genom att först ange raden och sedan kolumnen.

Utdata: Ett tal som anger antalet olika vägar från H till S.

UPPGIFT 11.4

Taltriangeln igen. Skriv ett program som löser taltriangeln från Kapitel 5 med dynamisk programmering.