ПРВА КРАГУЈЕВАЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

ПРОБЛЕМИ КОЈИ СЕ МОГУ РЕШИТИ ДИНАМИЧКИ

Ученик:
Тишић Никола IVсм

Професор:
Васиљевић Зоран

Јун, 2010. год.

САДРЖАЈ:

УВОД

Динамичко програмирање најчешће се користи у проблемима оптимизације. Другим речима, проблем може имати више решења, а тражи се решење са оптималном вредношћу (најмања или највећа вредност), а ако проблем има више оптималних решења, тражи се било које од њих. У једној широкој класи проблема оптимизације, једно од оптималних решења можемо наћи користећи динамичко програмирање.

Сама реч „програмирање“ овде се односи на попуњавање једне или више табела при решавању проблема, а не на употребу компјутера и програмских језика. Неке технике које у себи имају елементе динамичког програмирања биле су познате и раније, али творцем метода данас се сматра професор Ричард Е. Белман. Педесетих година, прецизније 1953. године, Белман је проучавао динамилко програмирање и дао чврсту математичку основу за овај начин решавања проблема. Уопштено говорећи, проблем се решава тако што се уочи хијерархија проблема истог типа, садржаних у главном проблему, и решавање се почне од најједноставнијих проблема. Вредности и делови решења свих подпроблема памте се у табели, а свако наредно решење добија се генерисањем предходни вредности. На овај начин се генеришу решења све док се не добије оно које представља решење главног проблема, што и јесте крајњи циљ.

Када се говори о предностима динамичког програмирања, важно је и напоменути да се проблеми који се решавају овом методом могу решити и на друге начине. Неки од њих би били рецимо коришћење рекурзије или „backtracking“-а, ови начини осим што заузимају више меморијског простора, поједине подпроблеме решавају више од једног пута што се одражава на време извршавања таквих програма.

Када се говори о динамичком програмирању потребно је поменути и меморизацију. Иако се рекурзивна решења не препоручују због преклапања подпроблема, меморизацијом рекурзија може бити врло корисна. Меморизација, односно рекурзија са памћењем, представља једну од модификација динамичког програмирања и остварује се тако што се табеле најпре попуњавају бесмисленим вредностима које се никада не могу добити (0, -1 или $\pm\infty$). На овај начин означавамо да вредност решења није дефинисана, односно да проблем није решаван. Рекурзивна функција или процедура која решава задатак, треба најпре да потражи решење у табели. Ако нађе решење, враћа то решење (ако је у питању функција) и завршава са радом, а ако га не нађе, тражи рекурзивно, уписује у табелу, враћа га (ако је у питању функција) и завршава са радом.

О свему овоме биће више објашњења у наредним редовима.

ПРОБЛЕМИ ОПТИМИЗАЦИЈЕ

Проблем ранца

Проблем ранца са неограниченим бројем елемената представља један он најпознатијих и најилустративнијих примера методе динамичког програмирања.

Провалник са ранцем запремина N упао је у просторију у којој се чувају вредни предмети. У просторији има укупно M предмета. За сваки предмет позната је и његова вредност $v[k]$ и његова запремина $z[k]$, $k \in [1,M]$. Све наведене вредности су целобројне. Провалник жели да напуну ранац највреднијим садржајем. Потребно је одредити максималну вредност предмета који се могу ставити у ранац као и који су то предмети.

Решење:

Из задатка се види да треба изабрати оне предмете за које је сума запремина мања или једнака N, а чија је сума вредности максимална. На почетку прокоментаришимо неколико карактеристика проблема:

- Ранац који је оптимално попуњен не мора бити попуњен до врха (у неким случајевима то и неће бити могуће). На пример, посматрајмо случај у коме је $N=7$, $v=\{3,4,8\}$ и $z=\{3,4,5\}$. Овде је оптимално решење узети само последњи предмет чиме би вредност ранца била 8, док ранац не би био у потпуности попуњен.
- Највреднији предмет не мора ући у решење. Идеја да предмете треба сортирати по „вредности по јединици запремине“, тј. По $v[k]/z[k]$ није коректна. Овај приступ (грабљиви метод) не даје увек оптимално решење, што показује пример: $N=7$, $v=\{3,4,6\}$ и $z=\{3,4,5\}$. Овде би грабљивим алгоритмом као решење узели трећи предмет, док оптимално решење представљају предмети 1 и 2.

Из ове дискусије се види да проблем није једноставан и да се до решења не може доћи директно. Анализирајући проблем, можемо приметити следеће: ако је при оптималном попуњавању ранца последњи изабрани предмет κ, онда преостали предмети представљају оптимално попуњавање ранца запремина $N - z[k]$. Ова констатација се лако доказује свођењем на контрадикцију (као и већина ових проблема). Према томе оптимално попуњавање ранца садржи оптимално попуњавање мањег ранца, што нас наводи на динамичко програмирање.

Ако формирамо низ *vrednost* у коме би κ-ти елемент низа представљао оптимално попуњавање ранца запремине κ и низ *poslednji* у којем би κ-ти елемент представљао последњи стављени елемент у ранац запремине κ, онда би N-ти члан низа *vrednost*

представљао решење нашег задатка, а реконструкцијом низа *poslednji* добили бисмо предмете који су су једно или једино решење задатка.

```
program ranac;

var
 zap,vred:array[1..10000] of integer;
 vrednost,poslednji: array[1..100] of integer;
 k,i,V,br_predmeta,max_vred,max_zap;

begin
 readln(V,br_predmeta);
 for k:=1 to m do readln(zap[i],vred[i]);

 for k:=1 to V do
  begin
   vrednost[k]:=0;
   poslednji[k]:=0;
   for i:=1 to br_predmeta do
    if (zap[i]<=k) then
     if ((vrednost[k-zap[i]]+vred[i])>vrednost[i]) then
      begin
       vrednost[k]:=vrednost[k - zap[i]]+ vred[i];
       poslednji[k]:=i;
      end;
  end;

 max_vred:=0;
 for i:=1 to V do
  if ( max_vred > vrednost[i] ) then
   begin
    max_vred:=vrednost[i];
    max_zap:=i;
   end;
 writeln('Maksimalna vrednost ranca je: ',max_vred);
 writeln('Sadrzaj ranca je: ');
 k:=max_zap;
 while ( c[q] > 0 ) do
  begin
   write(poslednji[k],'.   ');
   k:=k-zap[poslednji[k]];
  end;
end.
```

```
#include<stdio.h>
main()
{
int zap[10000],vred[10000];
int vrednost[100],poslednji[100];
int k,i,V,br_predmeta,max_vred,max_zap;
scanf("%d%d",&V,&br_predmeta);
for (k=1;k<=br_predmeta;k++)
 scanf("%d%d",&zap[i],&vred[i]);
for (k=1;k<=V;k++)
 {
  vrednost[k]=0;
  poslednji[k]=0;
  for (i=1;i<=br_predmeta;i++)
   if (zap[i]<=k)
    if ( ( vrednost[k - zap[i]]+vred[i] ) >
          vrednost[i] )
     {
     vrednost[k]=vrednost[k - zap[i]]+vred[i];
     poslednji[k]=i;
     }
 }
max_vred=0;
for (i=1;i<=V;i++)
if ( max_vred > vrednost[i] )
 {
 max_vred=vrednost[i];
 max_zap=i;
 }
printf("Maksimalna vrednost ranca je: %d",
        max_vred);
printf("Sadrzaj ranca je: ");
k=max_zap;
while ( poslednji[k] > 0 )
 {
 printf("%d. ",poslednji[k]);
 k=k-zap[poslednji[k]];
 }
}
```

Максимални збир у матрици

Дата је матрица димензија *MxN*, попуњена целим бројевима. Са сваког поља дозвољено је прећи само на поље изнад, десно или по дијагонали (горе-десно) од тог поља. Потребно је изабрати пут од од доњег левог до горњег десног поља матрице тако да сума вредности обиђњних поља буде максимална. Одредити и штампати ту вредност.

Решење:

Као што се може приметити, генерисање свих могућих путева и памћење оног који има максималну суму није паметна идеја, јер са порастом димензија матрице, број путева се екпоненцијално повећава.

Али, ако се мало боље погледа, лако се може закључити да је последње укупан пут уствари вредност последњег поља сабрана са максималном вредношћу поља са којег се може доћи. Ако се ово рекурентно примени на предпоследње поље задатак се може разбити на више подпроблема, односно може се доћи до почетног поља, што ће нам представљати почетак.

Ако формирамо матрицу *dp* тако да *dp[i,j]* представља максималан збир од почетног поља до поља *dp[i,j]*, онда је јасно да је поље *dp[1,n]* уствари решење задатка, а да се матрица *dp* попуњава по следећем закону.

$$dp[i,j]=max\{dp[i,j-1], dp[i-1,j], dp[i-1,j-1]\} + vrednost[i,j]$$

```
program maximalni_zbir;

var
 dp : array [ 1..100 , 1..100 ] of integer;
 i,j,n,m:integer;

function max(a,b,c:real):integer;
 var
  temp:integer;

 begin
  temp := a;
  if ( b > temp ) then temp := b;
  if ( c > temp ) then temp := c;
  max := temp;
 end;

begin
 read(n,m);
 for i:= 1 to n do
  for j:= 1 to m do
   read(dp[i,j]);

 for i := 2 to m do
  dp[n,i] := dp[n,i] + dp[n,i-1];
 for i := n-1 downto 1 do
  dp[i,1] := dp[i,1] + dp[i+1,1];

 for i := n-1 downto 1 do
  for j := 2 to m do
   dp[i,j] := max(dp[i,j-1],
                  dp[i+1,j],
                  dp[i+1,j-1]) + dp[i,j];

 writeln('Maksimalan zbir koji se moze postici je: ',
          dp[1,m]:3:2);

end.
```

```
#include<stdio.h>

float max(float a,float b,float c)
{
 float temp;
 temp = a;
 if ( b > temp )  temp = b;
 if ( c > temp )  temp = c;
 return(temp);
}

main()
{
float dp[100][100];
int i,j,n,m;

scanf("%d%d",&n,&m);
for (i=1;i<=n;i++)
 for (j=1;j<=m;j++)
  scanf("%f",&dp[i][j]);

for (i=2;i<=m;i++)
 dp[n][i] = dp[n][i] + dp[n][i-1];
for (i=n-1;i>=1;i--)
 dp[i][1] = dp[i][1] + dp[i+1][1];

for (i=n-1;i>=1;i--)
 for (j=2;j<=m;j++)
  dp[i][j] = max( dp[i][j-1] ,
                  dp[i+1][j] ,
                  dp[i+1][j-1] ) + dp[i][j];

printf("Maksimalan zbir koji se moze postici je: %f",
        dp[1][m]);

}
```

Највећи квадрат

Дата је матрица димензија *MxN*. Поједина поља матрице попуњена су тачком. Наћи највећи квадрат састављен од поља матрице тако да не садржи ни једну тачку.

Решење:

Најпре ћемо за модел задате матрице узети логичку матрицу и поља на којима се налази тачко обележићемо са *False*, а она која је не садрже са *True*.

Задатак ћемо решити слично као и предходни с том разликом да ће свако поље матрице *dp[i,j]* имати за вредност максималну површину квадрата чији је доњи-десни угао баш поље са координатама *i,j*. Јасно је да ће поље матрице *dp* са максималном вредношћу бити решење задатка, а да се матрица, с обзиром да је потребно наћи квадрат, попуњава по следећем правилу:

$$dp[i,j] = min\{dp[i,j-1], dp[i-1,j], dp[i-1,j-1]\} + 1$$

```
program najveci_kvadrat;

var
 dp : array [1..200,1..200] of integer;
 log : array [1..200,1..200] of boolean;
 x,y,i,n,p,max : integer;

function min(a,b,c:integer):integer;

 var
  temp : integer;

 begin
  temp := a;
  if ( temp > b ) then temp := b;
  if ( temp > c ) then temp := c;
  min := temp;
 end;

begin
 readln(n);
 readln(p);
 fillchar(log,sizeof(log),true);
 fillchar(dp,sizeof(dp),0);
 for i := 1 to p do
  begin
   read(x,y);
   log[x,y] := false;
  end;

 for i := 1 to n do
  begin
   if ( log[1,i] ) then dp[1,i] := 1;
   if ( log[i,1] ) then dp[i,1] := 1;
  end;

 for y := 2 to n do
  for x := 2 to n do
   if ( log[x,y] ) then
    dp[x,y] := min(dp[x-1,y],
                   dp[x,y-1],
                   dp[x-1,y-1])+1;

 max := 0;
 for y := 1 to n do
  for x := 1 to n do
   if ( dp[x,y] > max ) then
    max := dp[x,y];

 writeln(max);
end.
```

```
Include<stdio.h>
int min (int a,int b, int c)

{
 int temp = a;
 if ( temp > b ) temp = b;
 if ( temp > c ) temp = c;
 return(temp);
}

main()
{
 int dp[200][200], log[200][200];
 int x,y,i,n,p,max;

 scanf("%d",&n);
 scanf("%d",&p);
 for (x=1;x<=n;x++)
  for (y=1;y<=n;y++)
  {
   log[x][y]=1;
   dp[x][y]=0;
  }

 for(i=1;i<=p;i++)
 {
  scanf("%d%d",&x,&y);
  log[x][y]=0;
 }

 for(i=1;i<=n;i++)
 {
  if (log[1][i] ) dp[1][i]= 1;
  if (log[i][1] ) dp[i][1]= 1;
 }

 for(y=2;y<=n;y++)
  for(x=2;x<=n;x++)
   if ( log[x][y] )
    dp[x][y] = min(dp[x-1][y] ,
                   dp[x][y-1] ,
                   dp[x-1][y-1])+1;

 max=0;
 for(y=1;y<=n;y++)
  for(x=1;x<=n;x++)
   if (dp[x][y] > max)
    max=dp[x][y];
 printf("%d\n",max);
}
```

Максимална сума несуседних

Дат је низ A од n чланова. Одредити подниз задатог низа чији је збор елемената максималан, а у коме нема суседних елемената задатог низа. Сматрати да празан подниз има суму 0.

Решење:

Нека је за задати низ A познат један такав подниз P. Ако елемент a_N не припада поднизу P, онда је P оптималан подниз и за низ A_{N-1}. Ако елемент a_N припада поднизу P, онда је подниз P без последњег елемента (тј. без a_N) оптималан за низ A_{N-2}. У противном би било могуће побољшати оптимални подниз P низа A, тако што се на бољи подниз низа A_{N-2} допише број a_N. Установили смо оптималност подструктуре проблема.

Решења за низове A_0 и A_1 су тривијална. За X од 2 до N, $P(A_X)$ је онај од низова $P(A_{X-1})$ и $P(A_{X-2}) \cup \{a_X\}$, који има већи збир.

Из ове дискусије јасно је да се низ dp формира по следећем правилу:

$$dp(0)=0;$$

$$dp(1)=max\{0,a_1\}$$

$$dp(X)=max\{B(X-1), B(X-2)+a_X\}, X=2,N$$

На основу формираног низа dp лако се формира подниз P који је решење задатка.

```
program podniz_nesusedni;

var
 niz: array [1..100] of real;
 dp: array [-1..100] of real;
 n,i:integer;

function max(a,b:real):real;
 begin
  if a<b then
   max := b
  else
   max := a;
  end;

procedure ispis(k:integer);
 begin
  if ( k > 0 ) then
   if ( dp[k] = dp[k-1] ) then
    ispis(k-1)
   else
    begin
     ispis(k-2);
     write(niz[k]:3:1,'  ');
    end;
 end;

begin
 read(n);
 for i := 1 to n do
  read(niz[i]);

 dp[-1] := 0;
 dp[0] := 0;
 for i := 1 to n do
  dp[i] := max( dp[i-1] , dp[i-2] + niz[i] );

 writeln('Trazeni podniz je:');
 ispis(n);
end.
```

```
#include<stdio.h>

float max(float a,float b);
 {
  return((a<b)?b:a);
 }

void ispis(int k);
 {
  if (k>0)
   if (dp[k]=dp[k-1])
    ispis(k-1);
    {
     ispis(k-2);
     printf("%d ",niz[k]);
    }
 }

main()
{
 float niz[100],dp[100];
 int temp=0,n,i;
 scanf("%d",&n);

 for (i=1;i<=n;i++)
  scanf("%d",&niz[i]);
 dp[0] = 0;

 dp[1]= max (dp[0], temp +niz[1]);
 for(i=2;i<=n;i++)
  dp[i] = max( dp[i-1] , dp[i-2] + niz[i] );

 printf("Trazeni podniz je:");
 ispis(n);
}
```

Најдужи заједнички подниз

Дата су два низа: *P* од *M* чланова и *Q* од *N* чланова. Наћи низ највеће могуће дужине који је подниз и низа *P* и низа *Q*.

Решење:

Нека је *НЗП(X,Y)* најдужи заједнички подниз низова P_X і Q_Y. У задатку се тражи *НЗП(M,N)*. Ако је $p_M = q_N$, тада је *НЗП(M,N) = НЗП(M-1,N-1) U* $\{p_M\}$, док је за $p_M \neq q_N$, *НЗП(M,N)* једнак дужем од *НЗП(M-1,N)* и *НЗП(M,N-1)*.

Ова релација нам омогућава да израчунамо све *НЗП(X,Y)* редом (по врстама или колонама), знајући да је *НЗП(0,Y)=НЗП(X,0)=Ø* (празан низ). Довољно је памтити дужине најдужих заједничких поднизова у матрици *dp*, а *НЗП(X,Y)* се лако реконструише на основу матрице *dp*.

На основу предходне дискусије јасно је да се матрица *dp* попуњава по следећем правилу:

$$dp[X,O] = dp[O,Y] = 0$$

$$dp[X,Y] = \begin{cases} dp[X-1,Y-1]+1, a[X] = b[Y] \\ \max\{dp[X-1,Y], dp[X,Y-1], a[X] \neq b[Y]\} \end{cases}$$

```
var
 a,b:array[1..100] of integer;
 dp:array[0..100,0..100] of integer;
 i,j,n,m:integer;

function max(a,b:integer):integer;
 begin
  if ( a >= b ) then
   max:=a
  else
   max:=b;
 end;

procedure ispis(x,y:integer);
 begin
  if ( ( x > 0 ) and ( y > 0  ) ) then
   if ( a[x] = b[y] ) then
    begin
     ispis(x-1,y-1);
     write(a[x]:5);
    end
   else
    if ( dp[x-1,y] > dp[x,y-1] ) then
     ispis(x-1,y)
    else
     ispis(x,y-1);
  end;

begin
 write('Unesi broj clanova niza A: ');
 readln(n);
 writeln('Pocni sa unosom niza A!');
 for i:=1 to n do
  read(a[i]);

 write('Unesi broj clanova niza B: ');
 readln(m);
 writeln('Pocni sa unosom niza B!');
 for i:=1 to m do
  read(b[i]);

 for i:=0 to n do
  dp[i,0]:=0;
 for i:=1 to m do
  dp[0,i]:=0;

 for j:=1 to m do
  for i:=1 to n do
   if ( a[i] = b[j] )  then
    dp[i,j]:=dp[i-1,j-1] + 1
   else
    dp[i,j]:=max(dp[i-1,j],dp[i,j-1]);

 writeln('Najduzi zajednicki podniz
          je duzine: ',dp[n,m]);
 writeln('Clanovi trazenog podniza su:');
 ispis(n,m);
end.
```

```
#include<stdio.h>
int max (int a,int b)
 {
 if ( a >= b )
  return(a);
 else
  return(b);
 }

void ispis(int x,int y)
 {
 if ( ( x > 0 ) && ( y > 0  ) )
  if ( a[x] == b[y] )
   {
   ispis(x-1,y-1);
   printf("%d ",a[x]);
   }
  else
   if ( dp[x-1][y] > dp[x][y-1] )
    ispis(x-1,y);
   else
    ispis(x,y-1);
  }

main()
{
 int    a[100],b[100];
 int    dp[100][100];
 int    i,j,n,m;

printf("Unesi broj clanova niza A: ");
scanf("%d",&n);
printf("Pocni sa unosom niza A!");
for (i=1;i<=n;i++)
 scanf("%d",&a[i]);

printf("Unesi broj clanova niza B: ");
scanf("%d",&m);
printf("Pocni sa unosom niza B!");
for (i=1;i<=m;i++)
 scanf("%d",&b[i]);

for (i=0;i<=n;i++)
 dp[i][0]=0;
for (i=1;i<=m;i++)
 dp[0][i]=0;

for (j=1;j<=m;j++)
 for (i=1;i<=n;i++)
  if ( a[i] == b[j] )
   dp[i][j]=dp[i-1][j-1] + 1;
  else
   dp[i][j]=max(dp[i-1][j],dp[i][j-1]);

printf("Najduzi zajednicki podniz je duzine: %d",
        dp[n][m]);
printf("Clanovi trazenog podniza su:");
ispis(n,m);
}
```

Најдужи неопадајући подниз

Дат је низ од N целих (или реалних) бројева. Наћи његов најдужи неопадајући подниз. Суседни чланови подниза не морају бити и суседни чланови у почетном низу.

Решење:

Нека је познат један најдужи неопадајући подниз (*ННП*) за задати низ. Ако се тај *ННП* не завршава елементом x_N, онда је тај *ННП* уједно и *ННП* за низ X_{N-1}. Међутим, уколико *ННП* низа X садржи и елемент x_N, није јасно шта добијамо када изоставимо x_N из *ННП*, односно за који низ је овај остатак оптималан низ.

До истог проблема долази се и ако покушамо да комбинујемо решења подпроблема. Предпоставимо да је задатак решен за сваки од низова $X_1, X_2, X_3, \ldots, X_K$. Да бисмо од *ННП* за X_K дошли до *ННП* за X_{K-1} размотримо следеће случајеве: ако је X_{K+1} већи од последњег елемента у *ННП* за X_K, онда га дописујемо на крај *ННП* и добијамо нови, дужи *ННП*. Међутим, ако се X_{K+1} не може дописати на крај *ННП* који смо до сада формирали, можда може да се допише на неки други *ННП* исте дужине.

```
program podniz;

var
 niz : array [1..100] of real;
 dp : array [1..100] of integer;
 i,j,n,max : integer;

begin
 readln(n);
 for i := 1 to n do read(niz[i]);
 dp[n] := 1;
 max := 1;
 for i:= n-1 downto 1 do
  begin
   dp[i] := 1;
   for j := i+1 to n do
   if ( ( niz[i] <= niz[j] )
   and ( dp[i] < ( dp[j]+1 ) ) ) then
    dp[i] := dp[j] + 1;
    if ( dp[i] > max ) then
    max := dp[i];
  end;

writeln('Najduzi podniz u zadatom nizu je:');
j:=1;
for i:= max downto 1 do
 begin
  while ( dp[j] <> i ) do
  inc(j);
  write(niz[j]:3:1,'  ');
  inc(j);
 end;
end.
```

```
#include<stdio.h>
main()
{
float niz[100];
int dp[100];
int i,j,n,max;

scanf("%d",&n);
for (i=1;i<=n;i++) scanf("%f",&niz[i]);
dp[n] = 1;
max = 1;
for (i=n-1;i>=1;i--)
 {
 dp[i] = 1;
 for (j=i+1;j<=n;j++)
  if ( ( niz[i] <= niz[j] ) &&
       ( dp[i] < ( dp[j]+1 ) ) )
   dp[i] = dp[j] + 1;
 if ( dp[i] > max )
  max = dp[i];
 }

printf("Najduzi podniz u zadatom nizu je:");
j=1;
for (i=max;i>=1;i--)
 {
 while ( dp[j] != i )  j++;
 printf("%f ",niz[j]);
 j++;
 }
}
```

Партиције

Одредити на колико се различитих начина задати природни број $n \leq 100$ може представити као збир природних бројева при чему поредак сабирака није битан.

Решење:

Задатак се поред осталих метода може решити и динамички, што и јесте тема овог рада. Формираћемо динамичку матрицу, а поље *dp[i,j]* означаваће тражени број начина (партиција) да се број *i* формира као збир, али уз помоћ сабирака који су мањи или једнаки од *j*. Када смо овако дефинисали матрицу *dp* онда је јасно да се прва врста и прва колона ове матрице попуњавају јединицама јер постоји само једна партиција сваког броја помоћу бројева мањих или једнаких од један (збир јединица), а исто тако постоји само једна партиција броја 1. Што се тиче осталих поља, прича није много компликованија. Замислимо да имамо донекле попуњену матрицу и да је на реду поље *dp[i,j]*, то би значило да тражимо број партиција броја *i* које чине само бројеви мањи или једнаки од *j*. При решавању овог подпроблема јављају се два случаја. Први је када је $i>j$,а други када је $i \leq j$. У првом случају решење се може наћи као збир броја партиција броја *i* када у њима не учествује *j* и партиција броја *i* када *j* учествује. У другом случају на број партиција *dp[i,i-1]* додајемо 1, односно партицију броја *i* која је сам тај број. У овом случају *j* које је веће од *i* не мења број партиција. Из ове дискусије јасно је да се динамичка матрица *dp* попуњава по следећем правилу:

$$dp[1,i]=dp[i,1]=1,\ 1 \leq i \leq n$$

$$dp[i,j] = \begin{cases} dp[i,i-1]+1, i \leq j \\ dp[i,j-1]+dp[i-j,j], i > j \end{cases}$$

```
program particije;

var
 dp:array[1..50,1..50] of integer;
 i,j,n:integer;

begin
 read(n);
 for i:=1 to n do
  begin
   dp[1,i] := 1;
   dp[i,1] := 1;
  end;

 for i:=2 to n do
  for j:=2 to n do
   if ( i <= j ) then
    dp[i,j] := dp[i,i-1] + 1
   else
    dp[i,j] :=dp [i-j,j] + dp[i,j-1];
 writeln(dp[n,n]);
end.
```

```
#include <stdio.h>

main()
{
 int dp[50][50];
 int i,j,n;
 scanf("%d",&n);
 for(i=1;i<=n;i++)
  {
   dp[1][i] = 1;
   dp[i][1] = 1;
  }

 for(i=2;i<=n;i++)
  for(j=2;j<=n;j++)
   if ( i <= j )
    dp[i][j] = dp[i][i-1] + 1
   else
    dp[i][j] =dp [i-j][j] + dp[i][j-1];
 printf("%d\n",dp[n][n]);
}
```

Шаховски бројеви

Ради додељивања телефонских бројева члановима шаховског клуба, телефонска компанија жели да зна колико има различитих „шаховских бројева“. Шаховски број се може откуцати потезима скакача, састоји се од *N* цифара и не може почети са 0 или 8. Написати програм који израчунава број различитих шаховских бројева.

Решење:

За решавање овог проблема потребно је формирати матрицу чије ће вредности поља *dp[i,j]* бити једнаке броју шаховских бројева дужине *i*, а који почињу цифром *j*. За дужину *1*, јасно је да је број таквих бројева по *1* за сваку почетну цифру. Сви бројеви дужине *i*, који почињу са *j* су уствари бројеви дужине *i-1* који почињу бројем на који се може „скочити“ са броја *j*. Када имамо овако попуњену матрицу решење задатка је збир свих вредности *N*-те врсте, искључујући поља *dp[n,0]* и dp*[n,8]* јер су то бројеви који почињу нулом или осмицом, а њих не узимамо у обзир (услов задатка). Јасно је да се прва врста може попунити тривијално, а да се свака следећа попуњава коришћењем вредности из предходне, те се уместо матрице могу користити два низа од којих би један био тренутна, а други предходна врста.

```
program sahovskibrojevi;

var
 dp_1,dp_2:array [0..9] of integer;
 i,n,sol: integer;

begin
 read(n);

 for i:=0 to 9 do
  dp_1 [i] := 1;

 for i:=2 to n do
  begin
   dp_2[0] := dp_1[4]+dp_1[6];
   dp_2[1] := dp_1[6]+dp_1[8];
   dp_2[2] := dp_1[7]+dp_1[9];
   dp_2[3] := dp_1[4]+dp_1[8];
   dp_2[4] := dp_1[3]+dp_1[9]+dp_1[0];
   dp_2[5] := 0;
   dp_2[6] := dp_1[1]+dp_1[7]+dp_1[0];
   dp_2[7] := dp_1[2]+dp_1[6];
   dp_2[8] := dp_1[1]+dp_1[3];
   dp_2[9] := dp_1[2]+dp_1[4];
   dp_1 := dp_2;
  end;

 for i := 1 to 7 do
  inc(sol , dp_1[i]);
 inc(sol , dp_1[9]);

writeln('Broj razlicitih brojeva telrefona za
         sahiste duzine ',n,' je: ',sol);
end.
```

```
#include <stdio.h>

main()
{
int prvi[10], drugi[10], brojac, brojac2, n, zbir;

scanf("%d", &n);

for (brojac = 0; brojac <= 9; prvi[brojac] = 1,
brojac++);

for (brojac = 2; brojac <= n; brojac++)
 {
 drugi[0] = prvi[4] + prvi[6];
 drugi[1] = prvi[6] + prvi[8];
 drugi[2] = prvi[7] + prvi[9];
 drugi[3] = prvi[4] + prvi[8];
 drugi[4] = prvi[3] + prvi[9] + prvi[0];
 drugi[5] = 0;
 drugi[6] = prvi[1] + prvi[7] + prvi[0];
 drugi[7] = prvi[2] + prvi[6];
 drugi[8] = prvi[1] + prvi[3];
 drugi[9] = prvi[2] + prvi[4];

 for( brojac2=0; brojac2 <= 9;
      prvi[brojac2] = drugi[brojac2], brojac2++ );
 }

for (brojac = 1, zbir = 0; brojac <= 7;
     brojac++, zbir += prvi[brojac]);

zbir += prvi[9];
printf("Broj razlicitih brojeva telefona za
sahiste duzine %d je: %d.\n",n,zbir);
```

Најјефтинија исправка речи

Дозвољене операције над стрингом су уметање слова, брисање једног слова и брисање свих слова до краја стринга. Свака од ових операција има задату цену. Потребно је одредити најмању укупну цену операција којима се од датог стринга *A* добија стринг *B*.

Решење:

Како се кроз цео рад бавимо динамичким програмирањем, јасно је да је и за решавање овог проблема потребно попуњавање динамичких таблица, али како? У овом задатку формираћемо динамичку матрицу тако да *i*-тој врсти одговара *i*-то слово једног стринга, а *j*-тој колони *j*-то слово другог стринга. У динамичкој матрици поље *dp[x,y]* представља оптималну цену да се стринг *A* од нулте до *x*-те позиције претвори у стринг *B* од нулте до *y*-те позиције. Задатак ћемо решавати тако што ћемо најпре решити тривијалне проблеме. Најпре ћемо израчунати колико би коштало да од стринга *A* направимо стринг дужине 0, потом ћемо се бавити проблемом колико кошта да из стринга дужине 0 направимо нови стринг дужине *i*, а цена тога је уствари *i* цена уметања једног карактера. Када смо решили тривијалне проблеме, можемо почети са попуњавањем осталих поља. Наиме, за поље *dp[x,y]* најпре ћемо проверити да ли су карактери на местима *A[x]* и *B[y]* једнаки и ако јесу то значи да не морамо ништа плаћати већ да је оптимално решење тог подпроблема једнако пољу *dp[x-1,y-1]*. Када ова два карактера нису једнака бирамо између три опције: да ли обрисати један карактер, изменити један карактер или уметнути један карактер. На основу овога што је продискутовано следи формула по којој се матрица *dp* попуњава:

$$dp[0,0]=0;$$

$$dp[x,0] = min\{dp[x-1]+cBrJ, cBrKr\}, x\in [1,m] \wedge x\in N$$

$$dp[0,y] = dp[0,y-1]+cUmet, y\in [1,n] \wedge y\in N$$

$$dp[x,y] = \begin{cases} dp[x-1,y-1], A[x]=B[y] \\ min\{dp[x-1,y-1]+cZam, dp[x,y-1]+cUmet, dp[x-1,y]+cBrJ\} \end{cases}, x\in [2,m] \wedge y\in [2,n]$$

$$\wedge x,y\in N$$

```
program najjeftinija_ispravka;
var
 dp: array [0..100,0..100] of real;
 a,b:string;
 cUmet, cBrJ, cZam, cBrKr:real;
 i,j,k,m,n:integer;
function min(a,b,c:real):real;
 var
  temp:real;
 begin
  temp:=a;
  if ( b < temp ) then
   temp:=b;
  if ( c < temp ) then
   temp:=c;
  min:=temp;
 end;
begin
 write('Unesi cenu umetanja: '); readln(cUmet);
 write('Unesi cenu brisanja jednog simbola: ');
 readln(cBrJ);
 write('Cena zamene: '); readln(cZam);
 write('Cena brisanja do kraja stringa: ');
 readln(cBrkr);
 write('Unesi string A: '); readln(a);
   m := length(a);
 write('Unesi string B: '); readln(b);
   n := length(b);
 dp[0,0] := 0;
 for i:=1 to m do
  begin
   dp[i,0] := dp[i-1,0] + cBrJ;
   if dp[i,0] > cBrKr then
    dp[i,0] := cBrKr;
  end;
 for i := 1 to n do
  dp[0,i] := dp[0,i-1] + cUmet;
 for i := 1 to m do
  for j := 1 to n do
   begin
    if a[i] = b[j] then
     dp[i,j] := dp [i-1,j-1]
    else
     dp[i,j] := min (dp[i-1,j-1] + cZam,
                     dp[i,j-1] + cUmet,
                     dp[i-1,j] + cBrJ);
    for k := 1 to i-1 do
     if dp[i,j] > dp[k,j] + cBrKr then
      dp[i,j]:=dp[k,j] + cBrKr;
   end;
 writeln('Najjeftinija ispravka kosta:
',dp[m,n]:5:2);
end.
```

```
include<stdio.h>

float min(float a,float b, float c)
{
 float temp;

temp=a;
 if(b<temp) temp=b;
 if(c<temp) temp=c;
 return(temp);
}

main()
{
 float dp[100][100],cUmet,cBrJ,cZam,cBrKr;
 char a[255],b[255];
 int i,j,k,m,n;

 printf("Unesi cenu umetanja: ");scanf("%f",&cUmet);
 printf("Unesi cenu brisanja jednog simbola: ");
 scanf("%f",&cBrJ);
 printf("Cena zamene: "); scanf("%f",&cZam);
 printf("Cena brisanja do kraja stringa: ");
 scanf("%f",&cBrkr);
 printf("Unesi string A: "); gets(a); m=strlen(a);
 printf("Unesi string B: "); gets(b); n=strlen(b);

dp[0][0] = 0;
 for(i=1;i<=m;i++)
  {
   dp[i][0]=dp[i-1][0]+cBrJ;
   if(dp[i][0]>cBrKr)dp[i][0]=cBrKr;
  }

 for(i=1;i<=n;i++)
  dp[0][i]=dp[0][i-1]+cUmet;

 for(i=1;i<=m;i++)
  for(j=1;j<=n;j++)
   {
    if(a[i]==b[j]) dp[i][j]=dp[i-1][j-1];
              else dp[i][j]=min(dp[i-1][j-1]+cZam,
                                dp[i][j-1]+cUmet,
                                dp[i-1][j]+cBrJ);
    for(k=1;i<=i-1;i++)
     if(dp[i][j]>dp[k][j]+cBrKr)
      dp[i][j]=dp[k][j]+cBrKr;
   }

 printf("Najjeftinija ispravka kosta: %g\n",dp[m][n]);

}
```

Сви најјефтинији путеви

У једној земљи има N градова. Сви градови су повезани путевима, мада не обавезно директним. Нема градова који су спојени са више од једног директног пута. За свако путовања од града до града позната је (позитивна) цена, која не мора бити сразмерна дужини пута. Цене су дате матрицом D од N врста и N колона. Ако између градова i и j не постоји пут, цена таквог пута је бесконачна. Потребно је одредити најниже цене путовања за сваки пар градова.

Решење:

Овај проблем решава се врло познатим алгоритмом Флојда, који на први поглед нема везе са динамичким програмирањем. Констатујмо најпре пар чињеница. На најјефтинијем путу од града u до града v градови могу да се појаве највише једном. У противном би било могуће појефтинити пут избацивањем петље, што је у контрадикцији са претпоставком да је пут најјефтинији. Ако је пут P најјефтинији пут од u до v, и пут P садржи чвор w, онда је део пута P од w до v најјефтинији, што се једноставно доказује свођењем на контрадикцију. У овом тврђењу се огледа оптималност подструктуре проблема. Формулишимо исто запажање на нешто погоднији начин: нека је одређен оптималан (најјефтинији) пут за сваки пар градова. Нека оптималан пут P од u до v не садржи град N. Тада је пут од u до v, оптималан пут који користи само (неке од) првих $N-1$ градова за преседање. Ако оптимални пут P садржи град N, онда су путеви од u до N и од N до v оптимални путеви који користе само првих $N-1$ градова за преседање (јер се на путу P град N може појавити највише једном; и јер је пут P оптималан, па су оптимални и његови делови). Сада је јасно да величина подпроблема треба да се зада бројем градова преко којих се сме преседати. Подпроблем величине K је налажење (за сваки пар градова) оптималних путева, користећи по потреби преседање само у првих K градова.

Даље решавање је једноставно. Нека је dp_κ матрица дужина оптималних путева који за међучворове користе само првих K чворова. Тада је:

$$dp_0[i,j] = dp[i,j]$$

$$dp_k[i,j] = min\{\ dp_{k-1}[i,j],\ dp_{k-1}[i,K]\ dp_{k-1}[K,j]\ \}$$

Тражена матрица најјефтинијих путева је dp_N. Да бисмо могли да реконструишемо и саме потеве, користићемо и матрицу *rek*. На почетку *rek[i,j]=0* за свако i и j. Када се при неком K промени *dp[i,j]* ставимо *rek[i,j]=K*, чиме смо запамтили да оптималан пут (у K-том подпроблему) иде преко K. Рекурзивна процедура *ispis(i,j)* исписује бројеве свих градова на путу од i до j (осим самих i и j).

```
var
 dp,rek : array [1..100,1..100] of integer;
 i,j,k,n : integer;
procedure ispis(a,b:integer);
 begin
  if rek[a,b]>0 then
   begin
    ispis(a,rek[a,b]);
    write(rek[a,b]:5);
    ispis(rek[a,b],b);
   end;
  end;
begin
 write('Unesi broj gradova: ');readln(n);
 for i:=1 to n do
  for j:=1 to n do
   begin
    write('Unesi cenu puta od ',i,'. do ',j,'.
          grada: '); readln(dp[i,j]);
    if dp[i,j]<0 then
     dp[i,j] := maxint;
     rek[i,j] := 0;
   end;
 for k := 1 to n do
  for i := 1 to n do
   for j := 1 to n do
    if dp[i,j] > dp[i,k] + dp[k,j] then
     begin
      dp[i,j] := dp[i,k] + dp[k,j];
      rek[i,j] := k;
     end;
 writeln('Optimalne cene puteva: ');
 for i := 1 to n do
  begin
   for j:=1 to n do
   write(dp[i,j]:5);
   writeln;
  end;
 write('Unesi put koji zelis da rekonstruises: ');
 readln(i,j);
 write(i:5); ispis(i,j); write(j:5); readln;
end.
```

```
#include<stdio.h>
void ispis(int a,int b)
 {
 if (rek[a,b]>0)
  {
  ispis(a,rek[a][b]);
  printf("%d ",rek[a][b]);
  ispis(rek[a][b],b);
  }
 }

main()
{
int dp[100][100],rek[100][100];
int i,j,k,n;

printf("Unesi broj gradova: ");scanf("%d",&n);
for (i=1;i<=n;i++)
 for (j=1;j<=n;j++)
  {
  printf("Unesi cenu puta od %d. do %d. grada: ",
          i,j);scanf("%d",&dp[i][j]);
  if (dp[i][j]<0)
   dp[i][j] = 32000;
  rek[i][j] = 0;
  }

for (k=1;k<=n;k++)
 for (i=1;i<=n;i++)
  for (j=1;j<=n;j++)
   if (dp[i][j] > dp[i][k] + dp[k][j])
    {
    dp[i][j] = dp[i][k] + dp[k][j];
    rek[i][j] = k;
    }

printf("Optimalne cene puteva: ");
for (i=1;i<=n;i++)
 {
 for (j=1;j<=n;j++)
 printf("%d ",dp[i][j]);
 }

printf("Unesi put koji zelis da rekonstruises: ");
scanf("%d%d",&i,&j);
printf("%d ",i);ispis(i,j);printf("%d ",j);
}
```

Најбрже степеновање

Дат је природан број N и променљива K. Користећи операције множења и степеновања, мале заграде и променљиву K, написати израз који је једнак K^N, а у коме учествује минималан број операција. Множење се сматра једном операцијом, а рачунање Q-тог степена за $Q-1$ операцију. Приликом испитивања израза степеновање изразити са две звездице.

Решење:

Нека се у оптималном изразу једнаком K^N ($N>1$) последња извршава операција множења. Тада је израз облика T_1*T_2, где су T_1 и T_2 оптимални изрази једнаки K^P и K^{N-P} за неко P. Број операција у изразу једнаком K^N је тада $dp[N]=dp[P]+dp[N-P]+1$.

Ако се у оптималном изразу једнаком K^N ($N>1$) последња извршава операција степеновања, израз је облика T^R, где је N дељиво са R, а T је оптималан израз једнак $K^{N/R}$, па важи да је тада $dp[N]=dp[N/R]+R-1$.

Видели смо у чему се огледа оптималност подструктуре проблема. Приликом решавања свих подпроблема редом (за X од 1 до N), оптималан израз једнак K^X наћи ћемо тако што сваку могућу операцију испробамо као последњу, а за подизразе које треба уврстити у изразе користимо раније израчуната решења. За добијање првог степена K^1 потребно је 0 операција, па имамо следеће правило по којем попуњавамо динамичку матрицу:

$$dp[1]=0$$

$$dp[X]=\min\begin{cases}\min\limits_{1\le P\le X}\{dp[P]+dp[X-P]+1\}\\ \min\limits_{R\neq 1,R|X}\{dp[X/R]+R-1\}\end{cases},1<X\le N$$

Да би се реконструисао израз са најмање операција, који је једнак K^N, довољно је при решавању сваког подпроблема запамтити последњу извршену операцију. Поред низа dp, који памти најмањи број операција, памтићемо и низ rek исте дужине. Ставићемо $rek[X]=P$, ако се K^X најјефтиније добија као производ оптималних израза једнаких K^P и K^{X-P} редом, или $rek[X]=-R$ ако K^X најјефтиније добијамо као R-ти степен оптималног израза једнаког $K^{X/R}$.

```
program dp_stepenovanje;

var
 dp,rek : array [1..100] of integer;
 i,j,n:integer;

procedure ispis(n:integer);

 begin
  if rek[n]=0 then
   write('K')
  else
   if rek[n] < 0 then
    begin
     write('(');
     ispis(n div (-rek[n]));
     write(')**',-rek[n]);
    end
   else
    begin
     ispis(rek[n]);
     write('*');
     ispis(n-rek[n]);
    end;
 end;

begin
 dp[1] := 0;
 rek[1] := 0;
 write('Unesi broj N: ');readln(n);
  for i :=  2 to n do
   begin
    dp[i] := dp[1] + dp[i-1] + 1;
    rek[i]:=1;
    for j := 1 to i-1 do
    if dp[i] > dp[j] + dp[i-j] + 1 then
     begin
      dp[i] := dp[j] + dp[i-j] + 1;
      rek[i] := j;
     end;

    for j := 2 to ( i div 2 ) do
     if ( i mod j ) = 0 then
      if dp[i]>dp[i div j] + j-1 then
       begin
        dp[i] := dp[i div j] + j - 1;
        rek[i] := -j;
       end;
   end;

 writeln('Potreban broj operacija je: ',dp[n]);
 write('Izraz za to je: '); ispis(n); writeln;
end.
```

```
#include<stdio.h>

void ispis(int n)
{
 if (rek[n]==0)
  printf("K");
 else
  if (rek[n] < 0)
  {
   printf("(");
   ispis(n / (-rek[n]));
   printf(")**-%d",rek[n]);
  }
  else
  {
   ispis(rek[n]);
   printf("*");
   ispis(n-rek[n]);
  }
}

main()
{
 int dp[100],rek[100];
 int i,j,n;

 dp[1] = 0;
 rek[1] = 0;

 printf("Unesi broj N: ");scanf("%d",&n);
 for (i=2;i<=n;i++)
 {
  dp[i] = dp[1] + dp[i-1] + 1;
  rek[i]=1;
  for (j=1;j<=i-1;j++)
  if (dp[i] > dp[j] + dp[i-j] + 1)
  {
   dp[i] = dp[j] + dp[i-j] + 1;
   rek[i] = j;
  }

 for (j=2;j<=( i / 2 );j++)
  if (( i / j ) == 0)
   if (dp[i]>dp[i / j] + j-1)
   {
    dp[i] = dp[i / j] + j - 1;
    rek[i] = -j;
   }
 }

printf("Potreban broj operacija je: %d",dp[n]);
printf("Izraz za to je: ");ispis(n);
}
```

Број палиндрома

Написати програм којим се одређује број начина на који се могу обрисати неки знакови из стринга тако да он буде палиндром.

Решење:

Нека је *a* унети стринг, *n* његова дужина, а *palin* дуинамичка матрица. Динамичка матрица се попуњава тако што се упоређују знакови *a[i]* и *a[i+k]* (при чему је *k* удаљеност знакова $(k \in [1,n] \wedge k \in N))$ и ако су једнаки онда је *palin[i,i+k]=palin[i,i+k-1]+ palin[i+1,i+k]+1*, у супротном је *palin[i,i+k]=palin[i,i+k-1]+palin[i+1,i+k]-palin[i+1,i+k-1]*. Поље *palin[i,j]* представља број начина да се избришу неки знакови тако да део стринга од индекса *i* до индекса *j* буде палиндром. Ако су знакови са индексима *i* и *j* различити сабирају се број начина за део стринга од *i* до *j-1* и број начина за део стринга од *i+1* до *j*. Међутим ту је два пута урачунат део стринга од *i+1* до *j-1*, па се тај део одузима. У случају када упоређени карактери исти поново се сабирају два поменута дела стринга, али иако је део од *i+1* до *j-1* два пута урачунат овог пута се не одузима, већ се на овај збир дода 1. То се ради из разлога што се од свих палиндрома од *i+1* до *j-1* могу формирати нови када се знакови који су једнаки ставе на почетак и крај, а *1* се додаје јер знакови који су једнаки и сами формирају палиндром. На крају се испише *palin[1,n]*, јер се на овај начин попуњавају прво главна дијагонала, а затим све њој паралелне изнад ње у смеру од горњег левог угла ка доњем десном.

Пример:

Улаз: патка

	п	а	т	к	а
п	*1*	*2*	*3*	*4*	*8*
а		*1*	*2*	*3*	*7**
т			*1*	*2*	*3*
к				*1*	*2*
а					*1*

* - ово је једино поље где су знакови који се упоређују једнаки. Прво се сабирају палиндроми од *а* до *к* и од *т* до *а* и то су; *а,т, к, т, к,* а и двапут су урачунати палиндроми *т* и *к*. Међутим од њих се формирају палиндроми *ата* и *ака* и додаје се још један палиндром – *аа*. На крају се испише поље *palin[1,5]*, а то је *8*.

```
program palindromi;
var
 i,n,k:integer;
 a:ansistring;
 palin:array[1..5000,1..5000] of longint;

begin
 readln(n);
 readln(a);

 for i:=1 to n-1 do
  begin
   palin[i,i]:=1;
   if a[i]=a[i+1] then
    palin[i,i+1]:=3
   else
    palin[i,i+1]:=2;
  end;

 palin[n,n]:=1;
 for k:=1 to n-1 do
  for i:=1 to n-k do
   begin
    if a[i]=a[i+k] then
          palin[i,i+k]:=palin[i,i+k-1]+
                        palin[i+1,i+k]+1
    else
          palin[i,i+k]:=palin[i,i+k-1]+
                         palin[i+1,i+k]-
                         palin[i+1,i+k-1];
   end;

 writeln(palin[1,n]);
end.
```

```
include<stdio.h>

main()
{
 int i,n,k;
 char a[5000];
 longint palin[5000];

 scanf("%d",&n);
 scanf("%d",&a);

 for(i=1;i<=n-1;i++)
 {
  palin[i][i]=1;
  if(a[i]=a[i+1]) palin[i][i+1]=3;
  else palin[i][i+1]=2;
 }

 palin[n][n]=1;
 for(k=1;k<=n-1;k++)
  for(i=1;i<=n-k;i++)
   {
    if(a[i]=a[i+k])
     palin[i][i+k]=palin[i][i+k-1]+
                   palin[i+1][i+k]+1;
    else
     palin[i][i+k]=palin[i][i+k-1]+
                   palin[i+1][i+k]-
                   palin[i+1][i+k-1];
   }

 printf("%d",palin[1][n]);
}
```

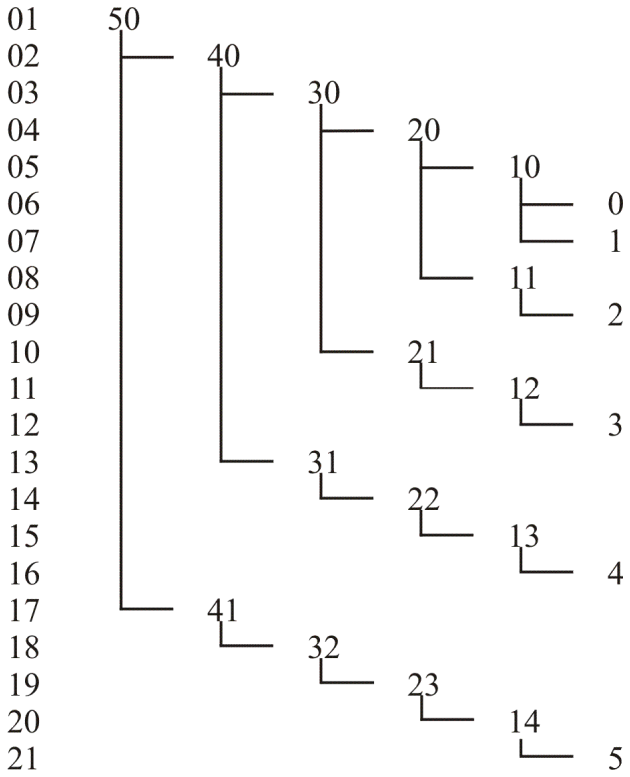
МЕМОРИЗАЦИЈА

У свим описаним проблемима попуњавана је једна табела (низ или матирца) решења подпроблема „одоздо на горе“, што значи да су проблеми решавани редом по сложености, од најједноставнијег до главног проблема. Већ је поменуто да у оваквим проблемима рекурзивна решења нису добра, зато што се подпроблеми делимични преклапају, па се поједини подпроблеми постављају и решавају више пута. Међутим, рекурзивно решење може бити и корисно, ако се паметно употреби и ако се избегну преклапања и вишеструка решавања подпроблема. То се може остварити на следећи начин: формира се табела решења подпроблема, као у решавању динамичким програмирањем. На почетку се сви елементи табеле поставе на неку бесмислену вредност (најчешће 0,-1 или ∞), чиме означавамо да је вредност решења недефинисанна, односно да проблем није решаван. Рекурзивна функција или процедура која решава задатак треба најпре да потражи решење у табели. Ако нађе решење враћа то решење (ако је у питању функција) и завршава са радом, а ако га не нађе, тражи га рекурзивно, уписује у табелу, враћа га (ако је функција) и завршава са радом.

Овакав начин решавања назива се меморизација, односно рекурзија са памћењем. Меморизација се може сматрати модификацијом динамичког програмирања, јер се и даље користи табела за памћење вредности решења решених подпроблема. Разлика је у томе што се проблеми меморизацијом (као и сваком другом рекурзијом) решавају „одозго на доле“. Ова разлика и није битна. Најважније је да се подпроблеми не решавају више пута. По ефикасности меморизација је врло блиска динамичком програмирању, а најчешће је незнатно спорија јер се извесно време губи на пренос параметара и друге активности које захтева сама рекурзија.

Постоје, међутим, проблеми у којима меморизација може бити ефикаснија од динамичког програмирања. То су проблеми у којима није потребно решити све подпроблеме да би се дошло до решења (што и јесте најчешћи случај), већ само неке од њих. Динамичко програмирање не може унапред знати да ли ће неки подпроблем бити касније употребљен или не, док меморизација тај недостатак нема из разлога што она поставља само њој неопходне подпроблеме.

Предпоставимо да је у проблему ранца задата величина 50, а да предмети (разних вредности) имају величине само 9 и 10. Динамичко програмирање попуњава ранчеве свих величина од 1 до 50, док меморизација попуњава само ранчеве оних величина које се појављују у дрвету рекурзивних позива (слика). Види се да у овом примеру меморизација решава 21 подпроблем уместо да решава 50 као што би то радило динамичко програмирање.



Стабло позива рекурзије у проблему ранца

Победничка стратегија

Постоји много игара за два играча, које имају следеће заједничке особине: дат је коначан скуп легалних позиција S и коначан скуп дозвољних потеза M, којима се из једне ллегалне позиције прелази у другу. Играчи вуку потезе наизменично. Један подскуп F скупа свих позиција S чине завршне позиције. То су позиције у којима није могуће одиграти ни један легалан потез. Када се стигне до неке завршне позиције, игра је завршена, а играч који је одиграо последњи потез је победник. Ни једна позиција се не може поновити током игре, тако да се свака игра неминовно завршава у коначном броју потеза и нерешен исход не постоји. Циљ сваког играча је да победи кад год је то могуће. Проблем који се у оваквим играма обично јавља је да за дату позицију установимо да ли играч који је на потезу може да победи, и који потез треба да одигра.

Ево једног поступка који у оваквим играма увек можемо применити. Према условима задатка, игра се може представити коначним ацикличним графом, тако да су чворови графа позиције, а гране графа су потези. Обојимо све чворове који одговарају завршним чворовима у црно. У тим позицијама, играч који је на потезу је изгубио. Након тога, необојене чворове из којих се неким потезом може прећи на црни чвор обојимо у бело (у таквој позицији играч који је на потезу може да победи тако што пређе на позицију којој одговара црни чвор), а необојене чворове из којих се сваким могућим потезом прелази у бели чвор обојимо у црно. Поступак бојења је коначан и сваки чвор на описани начин добија црну или белу боју. Победничка стратегија се (као што смо већ рекли) састоји у томе да играч који је на потезу пређе на позицију којој одговара црни чвор. На тај начин противник затиче или завршну позицију или је одмах изгубио, или позицију из које мора прећи у неку позицију којој одговара бели чвор. Према томе, први играч може да настави да спроводи исту стратегију, па ће на тај начин последњи потез и победити. Ево и примера једне овакве игре.

На столу је N куглица. Сваки играч наизменично узима по известан број куглица. У првом потезу играч може узети највише P куглица ($P<N$), а у сваком следећем потезу не више од P и не више од за Q више од онога колико је узео противник у претходном потезу ($Q<P$). На пример, за $N=100$, $P=10$, $Q=5$ ако би први играч узео 3 куглице, други може узети од 1 до 8 куглица, а ако би први узео 6 куглица, легални потези другог били би да узме од 1 до 10 куглица. Циљ је узети последњу куглицу. Одредити победничку стратегију.

Решење:

Позиција се може задати бројем преосталих куглица на столу и максималним бројем куглица које играч може узети. Тако је полазна позиција за задати пример ($100,10$), а из ње настају позиције ($97, 8$) и ($94, 10$). Можемо попунити матрицу dp величине NxP, тако да је

B[x,y] број куглица које према победничкој стратегији треба треба узети да би се победило, или *0* ако победу није могуће гарантовати. Подпроблеми се решавају по врстама са лева на десно, дакле за свако *x* од *0* до *N*, а при фиксираном *x* за свако *y* од *Q+1* do *P* (други број у уређеном пару који описује текућу позицију, представља горње ограничење броја куглица које се могу узети, а оно се у зависности од претходног израза креће од *Q+1* до *P*).

Меморизација је овде нешто боље решење јер се ни овде не морају решавати сви подпроблеми. Чим установимо да је позиција (*91,10*) губитничка, следи да је позиција (*100,10*) победничка, па нећемо имати потребе да решавамо проблем за позиције са *92, 93, 94, 95, 96, 97, 98* и *99* куглица. Таквих уштеда има много током игре, а не само на првом кораку. При неким вредностима *N*, *P* и *Q* (на пример *35,20,15*) може се десити да се стварно мање од *30%* укупног броја проблема, што значи да у тим случајевима меморизација ради и око три пута брже од класичног приступа динамичким програмирањем.

```
var
 dp:array[0..100,0..100] of integer;
 n,p,q,x,y:integer;

function min(a,b:integer):integer;
 begin
  if a>b then min:=a
  else min:=b;
 end;

procedure resi(x,y:integer);
 var
  k:integer;
 begin
  if b[x,y]=-1 then
   begin
    if x<=y then b[x,y]:=0;
     k:=y;
     while (k>0) and (b[x,y]=-1) do
      begin
       resi(x-k, min(k+q,p));
       if b[x-k, min(k+q,p)]=0 then
        b[x,y]:=k;
       k:=k-1;
      end;
     if b[x,y]=-1 then
      b[x,y]:=0;
   end
 end;

begin
 readln(n,p,q);
 for x:=0 to n do
  for y:=0 to p do
   b[x,y]:=-1;
 resi(n,p);
 writeln(b[n,p]);
end.
```

```
include<stdio.h>

int dp[100][100];

int min(int a,int b)
{
 return((a<b)? a:b);
}

void resi(int x,int y)
{
 int k;
 if(dp[x][y]==-1)
  {
   if(x<=y) dp[x][y]=0;
   k=y;
   while((k>0)&&(dp[x][y]==-1))
    {
     resi(x-k,min(k+q,p));
     if(dp[x-k][min(k+q,p)]==0)
      dp[x][y]=k;
     k--;
    }
   if(dp[x][y]==-1) dp[x][y]=0;
  }
}

main()
{
 int n,p,q,x,y;

 scanf("%d%d%d",&n,&p,&q);
 for(x=0;x<=n;x++)
  for(y=0;y<=p;y++)
   b[x][y]=-1;
 resi(n,p);
 printf("%d",b[n][p]);
}
```

ЗАКЉУЧАК

Из наведених примера може се видети када је згодно користити методу динамичког програмирања. То су сви они проблеми који у себи могу садржати мање подпроблеме који су хијерархијски уређени по сложености. Сада када смо се упознали и са динамичким програмирањем и са меморизацијом, јасно је да проблем који нам се зада можемо радити и индуктивно и дедуктивно, избор ћемо направити у складу са захтевима задатка.

На крају је остало још само да констатујемо да динамичко програмирање, као и метод меморизације треба користити када год је то могуће, јер решења најчешће буду једна од најбржих.

ЛИТЕРАТУРА

1. М. Вугделија: *Динамичко програмирање*, Друштво математичара Србије, Beograd, 1999.
2. М. Вугделија: *Програмирање и програмирање,* Сова, Beograd, 2010.
3. М. Чабаркапа: *Основи програмирања у Паскал-у са екстензијом у турбо паскал-а*, Иро „Грађевинска књига“, 1989.
4. М. Чабаркапа: *Основи програмирања у Ц-у,* Круг, Београд,1996.
5. Интернет: http://www.wikipedia.org
6. Интернет: http://www.z-trening.com