Dinamikus programozás

Szlávi Péter
ELTE IK
szlavip@ elte.hu

2005

Vázlat:

0 Bevezetés

Feldolgozásom nagyban épít Horváth Gyula, az irodalomjegyzékben elsőként felsorolt füzetére [HGy]. Gondolatmenetét, amellyel messzemenően egyetértek módszertani erényei miatt, átveszem: indulás egy konkrét feladat elemzéséből, majd a dinamikus programozás lényegi jellemzői következnek, s végül valahány példa, mintegy a bemutatott módszer sulykolására.

A dinamikus programozás (DP) rövid lényegét emígyen határozzák meg Rónyai és társai az [RISz] irodalomban: „A dinamikus programozás módszerek gyakran használatos valamilyen numerikus paramétertől függő érték optimumának meghatározására. Lényege, hogy az optimumot (optimális megoldást) alkalmas kisebb részfeladatok optimumából (optimális megoldásából) próbáljuk előállítani. A dinamikus programozást használó algoritmusok vezérlési szerkezete gyakran emlékeztet egy *táblázat* szisztematikus (pl. sorról sorra haladó) kiszámítására. A táblázat kitöltését általában egy *rekurzív összefüggés* teszi lehetővé, ami alapján a bejárási sorrend szerint korábbi elemekből meghatározhatók a többiek. A kapott módszer költségét többnyire a kitöltendő táblázat mérete határozza meg. …”

1 Egy gondolatébresztő példa

1.1 Az „iskolapélda” – pénzfelválthatóság

Az alábbi feladat egyik „iskolapéldája”, mondhatni: „állatorvosi lova” a dinamikus programozással szokásosan megoldható feladatok világának. Nézzük a következő pénzfelválthatóságot vizsgáló feladatot, mindjárt formalizálva:

Bemenete:

- $P=\{p_1,...,p_N\}$ pozitív egészek – a pénzcímletek, és
- E pozitív egész – a felváltandó összeg

Kimenete:

- FelváltahatóE logikai érték – jelentése, hogy felváltható-e az E a P halmazban felsorolt címletekkel úgy, hogy minden címletet legfeljebb egyszer használunk fel

Példa:

P = {5,5,10,20,20,100}, E = 50
FelváltahatóE = Igaz,
hiszen 50 = 5+5+20+20.

P = {5,5,10,20,20,100}, E = 200
FelváltahatóE = Hamis,
hiszen 200 > 5+5+10+20+20+100.

1.2 A példa „ízlelgetése”

A felváltás „szerkezetének” elemzése

Megoldásként természetesnek látszik egy *rekurzív* alapú megközelítés. Tegyük föl, hogy az eredmény „igenlő”. Ekkor teljesül:

(1a) $E = p_{i_1}+\ldots+p_{i_k}$,

feltehető nyugodtan:

(1b) $i_1<\ldots<i_k$ is.

Ez esetben világos, hogy

(2) $E-p_{i_k} = p_{i_1}+\ldots+p_{i_{k-1}}$

leírása annak a redukált feladatnak, amelyben a felváltandó $E-p_{i_k}$, és a felváltásban felhasználhatók a $\{p_1,\ldots,p_{i_{k-1}}\}$ címletek.

Részproblémákra bontás

Látszik, hogy a feladatot jól jellemezhetjük két paraméterrel: az X összeggel, és a legnagyobb i indexszel, amely a címlethalmaz még felhasználható elemeinek részhalmazát meghatározza: (X,i). (X,i) jelentése: X; $\{p_1,\ldots,p_i\}$. E jelöléssel például (2)-t így írhatjuk: $(E-p_{i_k}, i_k-1)$ [1], a kiinduló feladatot pedig: (E, N).

Rekurzív összefüggések a részproblémák és megoldásaik között

Jelöljük FV-vel a ***felvált***hatóságot eldöntő függvényt! FV(X,i) igaz, ha az X felváltható az első i címlet segítségével, egyébként hamis.

Pontosítsuk: FV(X,i) igaz, ha

1. $p_i=X$; vagy
2. $p_i<X$ és FV(X–p_i,i–1) igaz, feltéve, hogy i–1. címlet egyáltalán létezik;
   azaz p_i-t a megoldásba bevéve, a maradék összeget az őt megelőzőkkel fel tudjuk váltani; vagy

[1] Csak i_k–1–et írhatunk, hiszen a feldolgozás közben fogalmunk sincs arról, hogy i_{k-1} gyanánt melyik elem választható.

3. FV(X,i–1) igaz, feltéve, hogy i–1. címlet egyáltalán létezik;
azaz ha ugyan a p_i-t nem használhatjuk, az őt megelőzőkkel elvégezhető a felváltás.

Ugyanez formalizálva:

$$FV(X,i) := \begin{cases} i>0 \wedge p_i = X & \vee \\ i>1 \wedge p_i < X \wedge FV(X-p_i, i-1) & \vee \\ i>1 \wedge FV(X, i-1) & \end{cases} \qquad \text{(RekFVDef)}$$

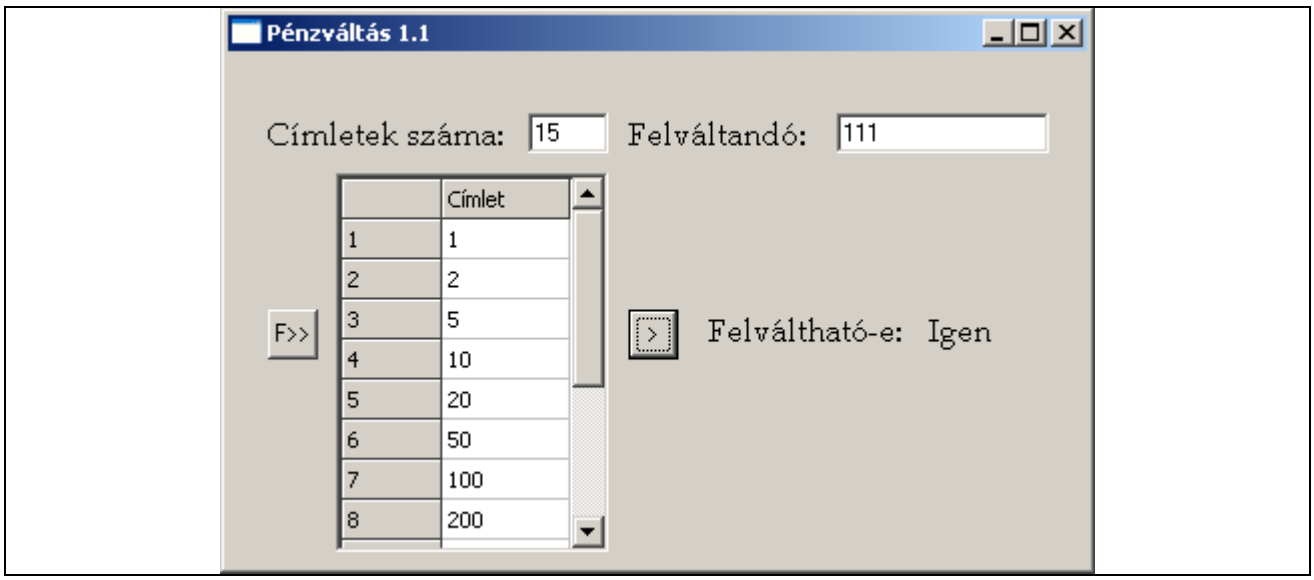
Az összefüggésből kiolvasható, hogy az (X,i) jellemezte probléma megoldása támaszkodik az (X,i–1), és az (X–p_i,i–1) részprobléma megoldására, ha ez egyáltalán lehetséges, vagyis ha i>1. i=1 esetén nincs rekurzív hivatkozás, az (X,1) megoldásának nincs „alapabb” összetevője.

A feladat megoldása: FV(E,N).

Rekurzív megoldás egy lehetséges algoritmusa

A lényegi eljárás Pascal-szerű algoritmusa az alábbi lehet[2]:

```
type TCimletek=record
                 db:integer; cimlet:array [1..MaxCimletDb] of integer
               end;
var Cimletek:TCimletek;
function FV(const X{felváltandó},i{max.index}:integer):boolean;
begin
  FV:=((i>0) and (Cimletek.cimlet[i]=X)) //i. éppen a kellő címlet
     or ((i>1) and (FV(X,i-1))) //nem az, de i-1.-ig felváltható
     or ((i>1) and (Cimletek.cimlet[i]<X) //i. felhasználható
         and (FV(X-Cimletek.cimlet[i],i-1))); //és a maradék i-1.-ig felváltható
end;//FV
```



1. ábra. Pénzváltás program futása: felváltható-e a 111 a {1,2,5,10,20,50,100,200,…} címletekkel?

Kipróbálhatjuk ennek Lazarus-os megvalósítását: zip, exe. Érdemes megfigyelnünk a következőket:

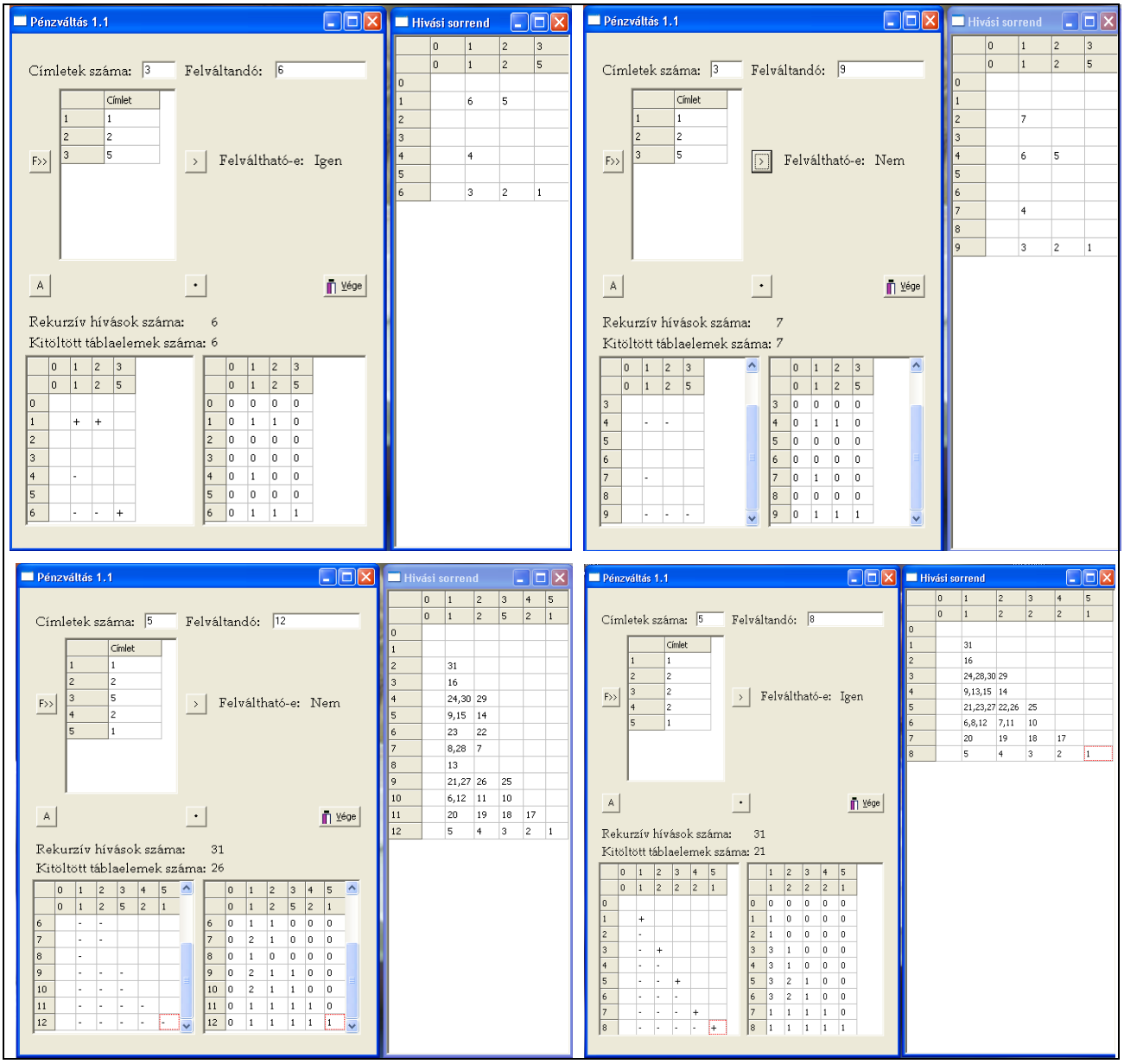
1. Kis címletszám (pl. {1,2,5,10,20}) mellett
   a. „igenlő” a megoldás (pl. 27) – hívás-sorrend
   b. „tagadó” a megoldás (pl. 9) – hívás-sorrend

[2] A rekurzív ágakat megcserélve; így előbbre véve az „olcsóbbat”, a kevesebb feltételt tartalmazót.

c. mindkét végeredményhez olyan adatok alapján, amelyben van ismétlődő címlet – hívási sorrend

2. Különféle címletszám (pl. 5, 9, 15, …) mellett olyan összeg, amely azért nem váltható fel, mert túl nagy – hívás-szám növekedése

A próba eredményeit foglaljuk össze a 2., 3. és 4. ábrán.



2. ábra. Pénzváltás program futási eredménye: felváltható-e a 6, ill. a 9 összeg az {1,2,5} címletekkel, továbbá a 12 az {1,2,5,2,1} címletekkel; és a 8 az {1,2,2,2,1} címletekkel?

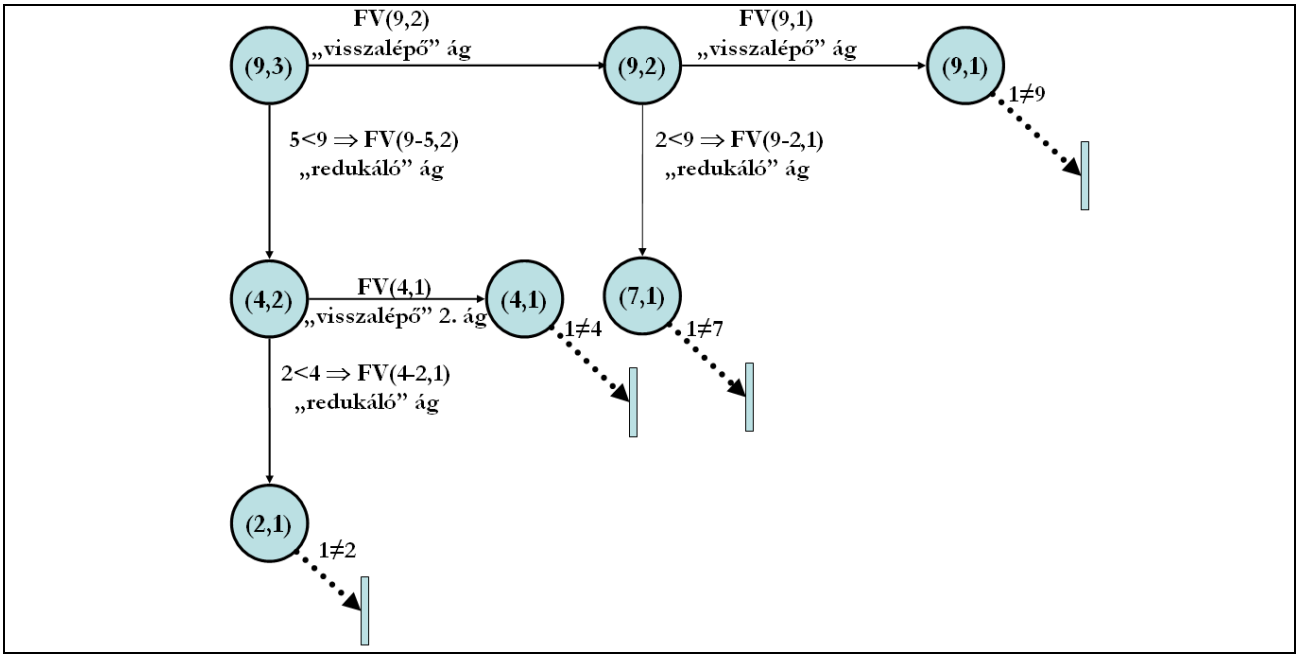
Egy kis kitérő a program használatához – kezelőfelületről

Az alkalmazás lehetővé teszi a címletek (számának és értékeinek) a bebillentyűzését, ill. fájlból beolvasását. Az ablakának alsó része lenyitható (a 2. ábrán látható screen-shot „Pénzváltás 1.1” ablak közepe táján, baloldalon található gomb segítségével, amin most „A” olvasható.) Itt kétféle, a végrehajtás mikéntjét magyarázó „adminisztrációs” táblázatot tekinthetjük át. A baloldali a rekurzió során kiszámolt részproblémák eredményét (az-az a FV(X,i) függvény értékét) tartalmazza. (Igaz : ’+’, Hamis : ’–’; üres : nem lett kiszámolva.) A jobboldali az egyes részproblémák újraszámolásának számát tartalmazza. Az utóbbiból derül majd fény a rekurzív módszer gyengeségére.

Egy különálló („Hívási sorrend” címkéjű) ablakban leshetjük meg, hogy milyen sorrendben történtek meg a hívások. (Ez az ablak a középső gombsor 2. gombjával nyitható-csukható.)

A 2. ábra alapos megfigyelése után a következőket fogalmazhatjuk meg:

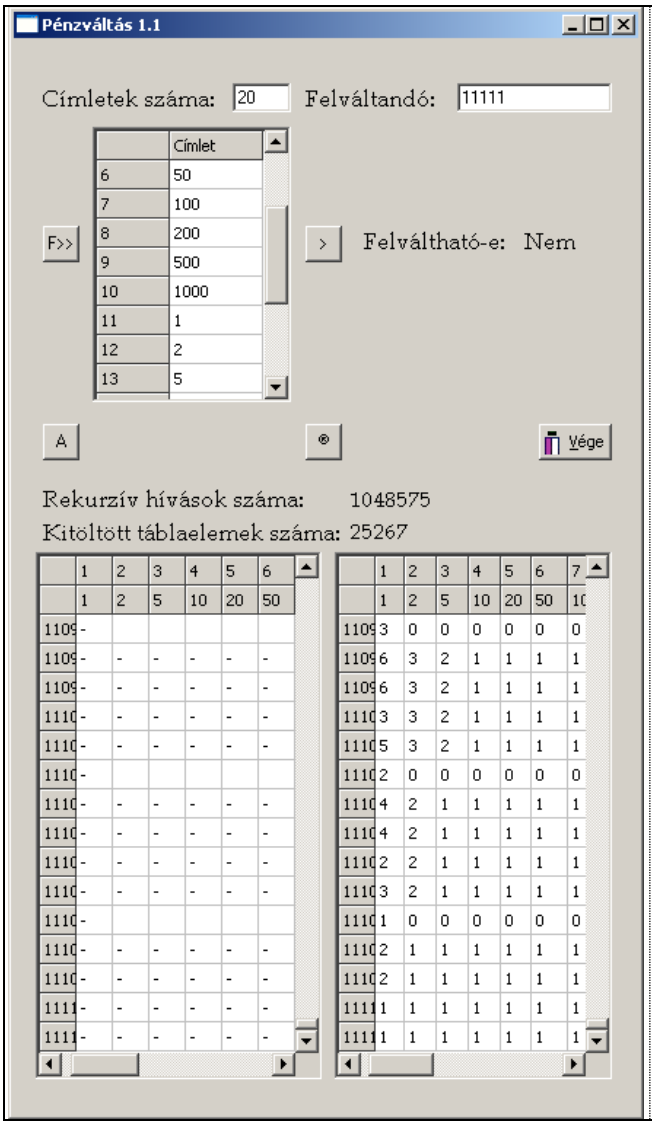
- A rekurzív hívásokban megfigyelhető szabályszerűség: először „vízszintesen” (az FV(X,i–1) ágon), majd ha így nem folytatható, „függőlegesen” (az FV(X–Cimletek.cimlet[i],i–1) ágon) lép a kisebb értékek felé.
- A hívások módszeresen következnek egymásután. Akár iterációval is „utánozhatnánk”. (Legalábbis az ismétlésmentes esetben.)
- Az ismétlésmentes esetekben a hívás-számok legfeljebb 1 értékűek, azaz legfeljebb egyszeres számolást jeleznek, ami a hatékonyság szempontjából megnyugtató. A 3. futásnál is még legfeljebb 2 az ismétlődő számítás, de a 4.-nél, ami alig különbözik a 3.-tól, már 4 helyen is „tripletet” látunk. S ez rosszat sejtet.
- A hívások során egy bináris fát „járunk be”. Példaként böngésszük át a 3. ábrát!



3. ábra. A „9 összeg {1,2,5} címletekkel” futáshoz tartozó bináris fa.

- Ha a 4. futáshoz tartozó bináris fát lerajzolnánk, érezhetnénk, hogy az újra meg újra „bejárandó” ágakkal még meggyűlik a bajunk.

Végezzük el ezek után a másik teszt-csoport futásait! Ennek eredményeit foglaltuk össze a 4. ábrán.
Az ábra tanúsága szerint durván növekszik a hívás-szám mindkét paraméter növekedtével. A legrosszabb (negatív végeredményű) esetekben a hívás-számokra „gyanús” értékeket kaptunk: valamely 2-hatványnál éppen eggyel kisebb érték! Ez elgondolkodtató. A növekedés tapasztalt ütemét okoskodással is beláthatjuk.



X	FV(X,20)	Hívás-szám
1	+	10
11	+	27
111	+	90
1111	+	601
11111	–	1 048 575
5	+	8
55	+	49
555	+	304
5555	–	1 048 575
12	+	26
123	+	99
1234	+	1 352
12345	–	1 048 575
A legrosszabb, negatív esetben:		
9999	FV(X,3)	7
9999	FV(X,5)	31
9999	FV(X,8)	255
9999	FV(X,10)	1 023
9999	FV(X,15)	8 195
9999	FV(X,20)	1 048 575

4. ábra. Pénzváltás program futása – hívás-szám növekedésüteme

Az FV függvényt 1 elemű címlethalmazzal meghívva további rekurzív hívás nélkül megkapjuk az eredményt. Tehát a hívás-szám: 1. (X,i)-re, *legkedvezőtlenebb* esetben a kiszámításhoz két rekurzív hívásra, az (X,i–1)-hez és az (X–p_i,i–1)-hez tartozóra van szükség, amelyek hívás-száma hozzáadódik az ő eredeti hívásához. Az a legkedvezőtlenebb eset, amelyben a „gyorsabban 0-ra fut(hat)ó” (X–p_i,i–1) is a lehető legtovább hívható újra meg újra, azaz X folytonosan csökkentve is pozitív marad: $X > \sum_{i=1}^{N} p_i$.

Formalizálva: RekHDb(X,i) jelentse az X összeg felbonthatóságához szükséges rekurzív hívások számát (i. címlettel bezárólag). Az elmondottak így formalizálhatók:

$RekHDb(X,1) = 1$ (RekHDb – R1)

$RekHDb(X,i) = RekHDb(X,i-1) + RekHDb(X-p_i,i-1) + 1$ (RekHDb – R2)

Ebből teljes indukcióval már könnyen belátható a tapasztalt szabályszerűség, azaz igaz az alábbi állítás:

Állítás:

Legkedvezőtlenebb esetben a szükséges rekurzív hívások száma:

$RekHDb(X,i) = 2^i - 1$ (RekHDb – I)

Bizonyítás:

… a fentiekből következik …

♦

Idekívánkozik egy további megjegyzés: ezt, a hívás-szám szempontjából negatív „eredményt” sajnos nemcsak az előre látható –s így akár ki is szűrhető– X értékek esetén produkálja az algoritmus, hanem igenlő válasz esetén is. Erre mutatott példát a 2. ábra 4. futása.

Mielőtt áttérnénk a gazdaságosabb megoldást jelentő DP-filozófiájúra, gondolja meg, majd próbálja ki azt a megoldást, amelyben az FV függvény lényegi részében a kiszámítási sorrendet a (RekFVDef) szerintire cseréli –szigorúan ragaszkodva hozzá–, azaz:

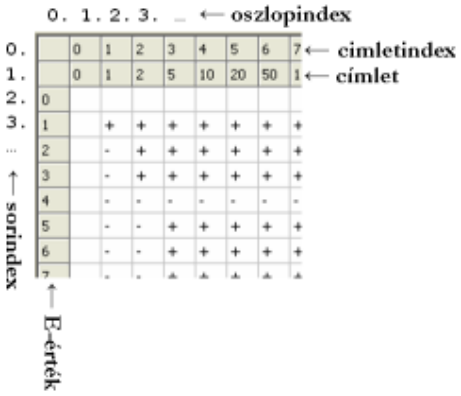
```
function FV(const X{felváltandó},i{max.index}:integer):boolean;
begin
  FV:=((i>0) and (Cimletek.cimlet[i]=X)) //i. éppen a kellő címlet
     or ((i>1) and (Cimletek.cimlet[i]<X) //i. felhasználható
         and (FV(X-Cimletek.cimlet[i],i-1))) //és a maradék i-1.-ig felváltható
     or ((i>1) and (FV(X,i-1))); //nem az, de i-1.-ig felváltható
end;//FV
```

A DP megoldás algoritmusa

A rekurzív megoldás hatékonysága tehát drasztikusan romlik a váltásban szereplő címletek számának növekedtével. A tesztprogram alapján támadhat az az ötletünk, hogy az ott tisztán adminisztratív célokat szolgáló táblázatot akár „tartalmi”, jobban mondva: hatékonyságot növelő, algoritmust érintő célra is bevethetnénk. Például úgy, hogy a táblázat kis indexű elemeinél kezdve[3] a kitöltést –ott, ahol nem rekurzív a fenti formula–, majd ezekre a kiszámolt értékekre támaszkodva haladunk a nagyobb tábla-indexű elemeken át, a meghatározandó táblaelem felé. Nyilvánvaló, hogy a táblázat elemeinek a száma (azaz méreteinek szorzata) lesz a hatékonyság mércéje. Így a növekedés üteme a címletszámtól csak *lineáris*, s nem exponenciális lesz. Igaz: az összegtől való függés is bekerül a képbe, amit eddig nem vettünk figyelembe.

Összevetésként nézzük meg, elöljáróban, a DP megoldást is! A DP-filozófia szerinti táblázat az előző megoldásban adminisztratív célokra szolgáló (ezért eddig „eltitkolt”) stringGrid lesz: a fmPenzValtas.sgRekTab nevű.

Ennek indexeléséhez számításba kell venni, hogy kétsoros fejléc és egyoszlopos oldalléc övezi –ahogy láttuk–, és „Delphi-s szokás” szerint 0-val kezdődik az indexelés.



A megoldást két részre vágjuk. Az első pusztán a táblázat előállításával foglalkozik, míg a második ez alapján megadja a keresett megoldást.

[3] Szokás mondani „alulról felfelé”…

```
procedure Tablafeltoltes (const E{felváltandó},N{max.index}:integer);
  var  i,x:integer;
begin
  with fmPenzValtas do
  begin
    for x:=1 to E do
    begin
      sgRekTab.Cells[2,x+2]:='-'; //első oszlop (=2 indexű) Hamis
    end;//for x
    //kivéve, ha az X éppen az 1. címlet:
    if Cimletek.cimlet[1]<=E then //ha van még cimlet[1]-dik sora a táblázatnak
    begin
      sgRekTab.Cells[2,Cimletek.cimlet[1]+2]:='+';
    end;//if
    for i:=2 to N do //az i. oszlop számítása
    begin
      for x:=1 to E do
      begin
        if   ({(i>0) and} (Cimletek.cimlet[i]=x)) //i. éppen a kellő címlet
          or ({(i>1) and} (sgRekTab.Cells[i,x+2])='+') //nem az, de i-1.-ig felváltható
          or ({(i>1) and} (Cimletek.cimlet[i]<x) //i. felhasználható
             and (sgRekTab.Cells[i,x-Cimletek.cimlet[i]+2]='+')) //és a maradék i-1.-ig
                                                                 //felváltható
        then
          sgRekTab.Cells[i+1,x+2]:='+'
        else
          sgRekTab.Cells[i+1,x+2]:='-'
        {endIf};
      end;//for x
    end;//for i
  end;//with
end;//Tablafeltoltes

begin//PenzValtas_DinProg
  //táblafeltöltés:
  Tablafeltoltes(Felvaltando,Cimletek.db);
  //megoldás megadás:
  PenzValtas_DinProg:=fmPenzValtas.sgRekTab.Cells[Cimletek.db+1,Felvaltando+2]='+';
end;//PenzValtas_DinProg
```

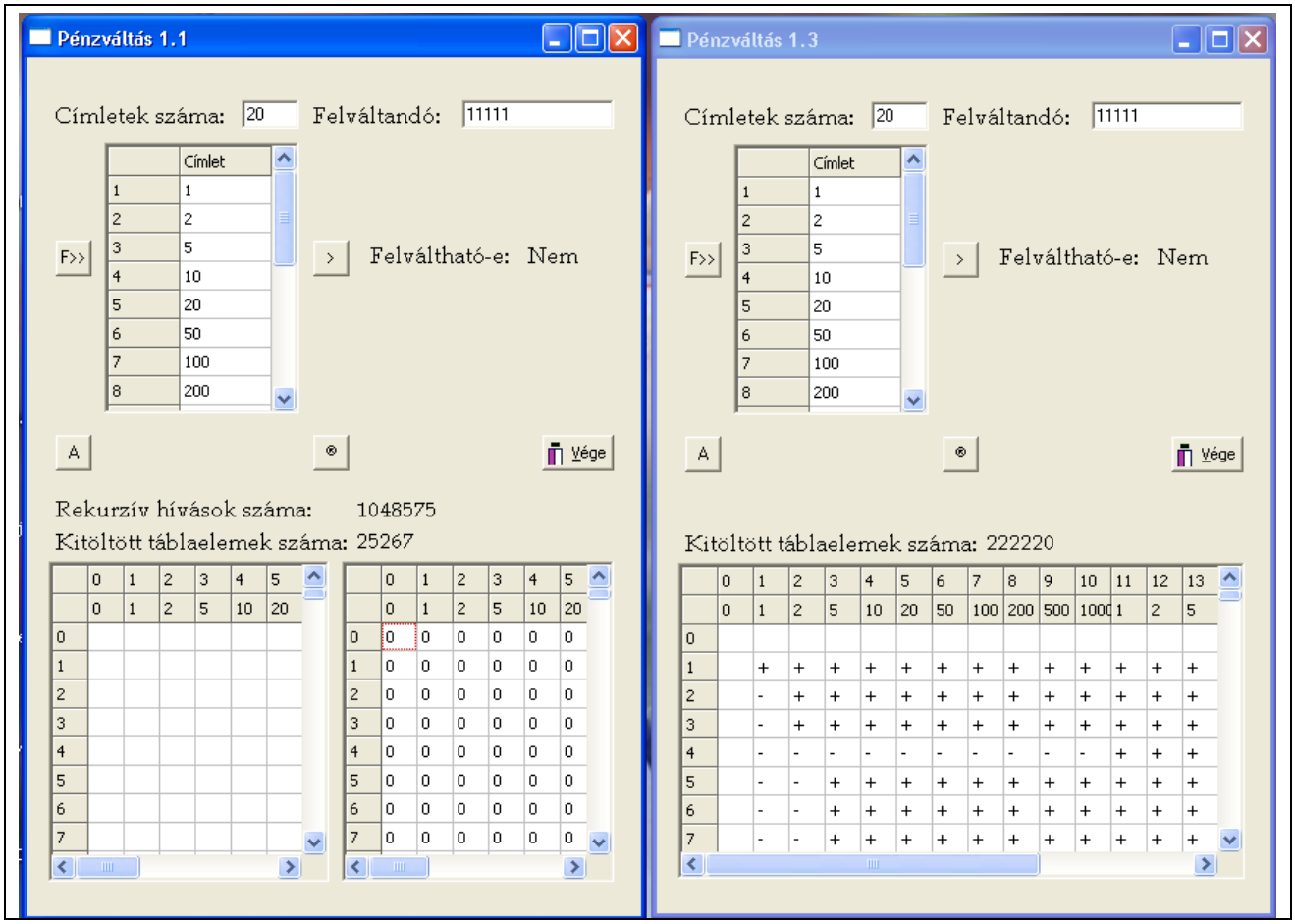
A következő, 5. ábrán összevethető a rekurzív és DP-gondolatmenet szerint működő programok futása abban a bizonyos „neuralgikus” esetben. A DP (az ábra jobboldali ablakában futó) program külleme is megváltozott, ui. semmi értelme most rekurzív hívásokat adminisztrálni. A sarkalatos különbséget az ablakpáron látható 3+1 szám jelenti. A +1 nem látható. ☺ Vessük össze ezeket!

Rekurzív hívások száma (azaz a kiszámolt táblaelemek száma az ismétlésekkel együtt: 1 048 575 ⇔ 222 220 (=a tábla mérete=20*11 111); a kitöltött táblaelemek száma: 25 267 ⇔ 222 220.

Egy kis elemzés következik. A rekurzív megoldás fajlagos ismétlési száma: 1 048 575 / 25 267 ≈ 41, ami borzasztó. Ennél természetesen elvileg és a futási idő tapasztalata szerint is jóval kedvezőbb a DP „222 220 darab egyszer kiszámolt táblaelem” stratégiája. A fajlagos sebesség növekedés: 1 048 575 / 222 220 ≈ 4,7. Mindazon által a „feleslegesen” kiszámolt táblaelem számát, a (222 220–25 267) = 196 953, ami hozzávetőlegesen 7,8 szorosa a szükségesnek, azt sokallanunk kell!

Egy „okosabb" DP-szerű megoldás algoritmusa

A kitöltött táblázatot összevetve a rekurzív hívás táblázatával, felvetődhet bennünk, hogy nem lehetne-e a rekurzív megoldás célratörését, azaz „*csak azt kiszámolni, amit muszáj*" elvét és a DP megoldás „*mindent csak egyszer kiszámolni*" elvével összeházasítani. Az üthet ugyanis szöget a fejünkbe, hogy a rekurzív megoldás táblázata meglehetősen ritkán kitöltött, hiszen csak a szükséges táblaelemeket számoltuk ki (sajnos esetleg többszöris), amíg a DP-ben sok-sok fölösleges táblaelem számításával is időt vesztegetünk.
Ez vezet egy újabb –a DP megengedőbb értelmezése szerint szintén dinamikus programozáshoz sorolt– megoldáshoz. Lényege dióhéjban: megmarad a rekurzív szemlélet, de csak akkor történik tényleges rekurzív hívás, amikor a paraméterekhez tartozó érték a táblázatban még kiszámolatlan. Világos, hogy a táblázatot előzetesen inicializálni kell (ahogy eddig is, csak éppen más ok miatt).



5. ábra. A rekurzív (baloldalon) és a DP (jobboldalon) megoldás programjának összevetése

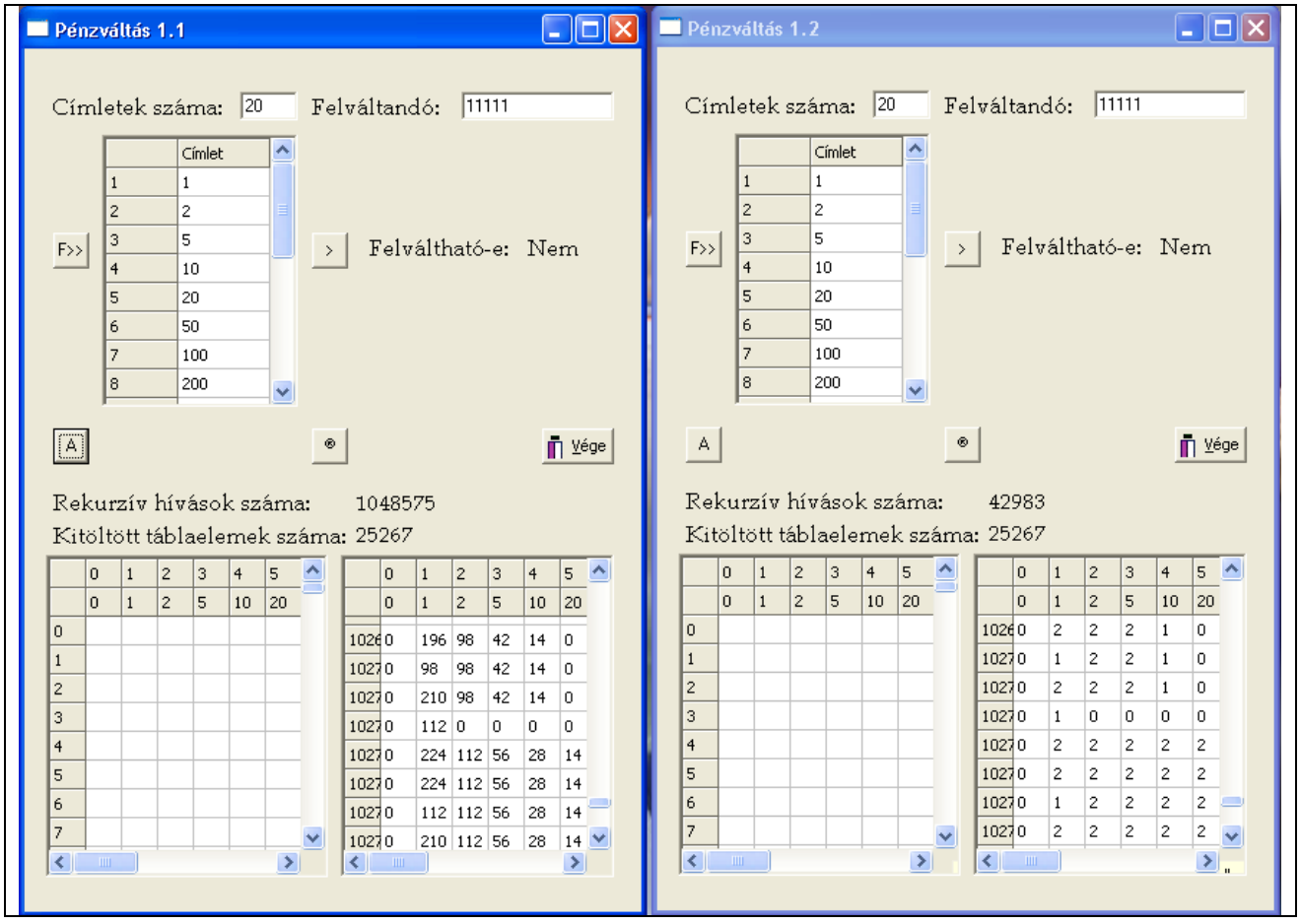
A lényegi függvény-eljárás algoritmusa az alábbi (ismét az sgRekTab stringGrid a DP táblázata):

```
Function FV(const X{felváltandó},i{max.index}:integer):boolean;
begin
  with fmPenzValtas do
  begin
    if
      (sgRekTab.Cells[i+1,X+2]='+') {már ismert a '+' válasz} then FV:=true
    else if
      (sgRekTab.Cells[i+1,X+2]='-') {már ismert a '-' válasz} then FV:=false
```

```
    else //még nem ismert a válasz és...
      begin
        if
           ((i>0) and (Cimletek.cimlet[i]=X)) //i. éppen a kellő címlet
           or ((i>1) and FV(X,i-1)) //nem az, de i-1.-ig felváltható
           or ((i>1) and (Cimletek.cimlet[i]<X) //i. felhasználható
               and FV(X-Cimletek.cimlet[i],i-1)) //és a maradék i-1.-ig felváltható
        then sgRekTab.Cells[i+1,X+2]='+' else sgRekTab.Cells[i+1,X+2]='-';
        FV:=sgRekTab.Cells[i+1,X+2]='+'; //pozitív-e a válasz?
      end;
  end;{with}
end;//FV
```

E megoldás jellemzéseként ismét egy futást demonstráló ábra következik.

Látható rajta, hogy a rekurzív hívások száma alig 42 983 / 1 048 575 ≈ 4%-a a „tisztán” rekurzívnak. Az átlagos ismétlés szám is viszonylag kicsi: 42 985 / 25 267 ≈ 1,7. Első blikkre meglepő, hogy nem egy. Ha azonban belegondolunk, ez a szám azt jelzi, hogy egy-egy elem kiszámolásához több úton is el lehet jutni, de onnan tovább már nem megy, hanem számolás nélkül visszaadja a korábban kiszámolt értéket.



6. ábra. A rekurzív (baloldalon) és a „okos” DP (jobboldalon) megoldás programjának összevetése

A fent elemzett algoritmusokat demonstráló alkalmazásokat próbálgathatjuk, ha letöltjük a következő linkeken található exe-ket: 1.1 (rekurzív), 1.2 (memorizálós), 1.3 (tiszta DP).
Ezzel lezárhatjuk a DP ízlelgetését. Foglaljuk össze az „absztrakt” tudnivalókat!

1.3 A dinamikus módszerek algoritmikus vázlata – összegezés

Mit kell tehát tennünk egy feladat dinamikus megoldásának tervezésekor?

1. A megoldás szerkezetének tanulmányozása – részproblémákra bontás megsejtése
2. Részproblémákra és összetevőkre bontás – részproblémák és paramétereik körvonalazása: a rekurzió előkészítése
   a. az összetevőkre bontás körmentes legyen
   b. minden részprobléma megoldása kifejezhető legyen rekurzívan az összetevők megoldásaival
3. Részproblémák megoldásának kifejezése rekurzívan az összetevők megoldásaiból – formalizálás: függvénydefiníció
4. Részproblémák megoldásának kiszámítása – a táblaszámítás algoritmizálása
   a. kiszámítási sorrend meghatározása: minden részprobléma minden összetevője előbb szerepeljen a felsorolásban
   b. az „alulról-felfelé" (helyesebben: a már kiszámoltak felől a még definiálatlanok felé) haladó számítás
5. A megoldás előállítása a 4. lépésben előállított táblázat segítségével – a megoldás algoritmizálása

Alkalmazzuk!

2 Első példázat – pénzváltás

2.1 A feladat

Az előzőleg taglalt pénzfelválthatóságot vizsgáló feladatot úgy módosítjuk, hogy meg kell adni a egy felváltásban szereplő címleteket. Formalizáljuk:

Bemenete:

- $P=\{p_1,...,p_N\}$ pozitív egészek – a pénzcímletek, és
- E pozitív egész – a felváltandó összeg

Kimenete:

- $S \subseteq P$ – a felváltásban szereplő pénzcímletek; az S halmazban felsorolt címleteket legfeljebb egyszer használhatjuk fel

2.2 A megoldás

2.2.1 A megoldás szerkezetének tanulmányozása

Ua. mint az 1. fejezetben az azonos című résznél

2.2.2 Részproblémákra és összetevőkre bontás

A részproblémák a korábbihoz hasonló: címlethalmaz még felhasználható elemeinek részhalmazát meghatározza az i index. Így a részprobléma most is azonosítható az (X,i)-vel. Az előzőhöz képest eltérés az elvárt eredmény jóval összetettebb volta: a logikai érték helyett itt egy *halmaz* (egy *sorozat*). A rekurzív gondolatmenetet és az eredmény összetett voltát úgy békítjük össze egymással, hogy leegyszerűsítjük a számítás lényegi, rekurzív részét. Ezért jelölje most FV(X,i) azt a *legnagyobb P-beli elem indexét*, amely még előfordul a X felváltásában. Ez ugyan még nem a végeredmény, de ebből kiindulva összeszedhetjük a kérdéses halmazt (sorozatot). Hogy hogyan? Nézzük az alábbi példát! Tegyük föl, hogy az FV függvény a fent elképzelt módon működik.

Pl.: $P=\{2,4,5\} \Rightarrow N=3$; $X=7$, akkor $S=\{2,5\}$

$FV(7,3)=\mathbf{3}$, mert $p_3=5$,

$FV(7-p_{FV(7,3)},FV(7,3)-1)=FV(7-p_3,\mathbf{3}-1)=FV(7-5,2)=FV(2,2)=\mathbf{1}$, mert $p_1=2$

és végül

$FV(2-p_{FV(2,2)},FV(2,2)-1)=FV(2-p_1,\mathbf{1}-1)=FV(2-2,0)=FV(0,0)=\ldots$

Azaz elértük szisztematikus rekurzív hívásokkal a felváltandó összeg 0-vá válását, s eközben a felváltásban szereplő p-ket előállítottuk.

Rekurzív megoldást keresünk nagyjából abból az ötletből kiindulva, hogy előállítunk egy P-beli elem indexét, s attól haladunk visszafelé.

Ha X nem váltható fel az első i címlettel, akkor FV(X,i) legyen N+1, azaz ezzel a „lehetetlen” indexszel jelezzük, hogy nincs megoldás. Mivel rekurzív megoldást keresünk és visszafelé lépdelünk a paraméterekkel, célszerű megengedni az X szerepében és az i-ében is a 0 értéket. Nézzük meg, mi a logikus érték egy ilyen paraméterhez: a FV(0,i) értéke legyen 0 minden i-re, hiszen készen vagyunk, továbbmenni nem kell. FV(X,0), ha X>0, akkor legyen N+1, hisz nincs címlet az összeg felváltásához, tehát vége kudarc-jelzéssel.

2.2.3 Részproblémák megoldásának kifejezése

A részproblémák összekapcsolása a fentiek után gyerekjáték. A fent elképzelt függvény lényegi részének a megalkotása maradt hátra, s jó csomó ága már kész.

$$FV(X,i) := \begin{cases} N+1 & , \text{ha } i = 0 \wedge X > 0 \\ 0 & , \text{ha } X = 0 \\ FV(X,i-1) & , \text{ha } FV(X,i-1) \le N \\ i & , \text{ha } X \ge p_i \wedge FV(X-p_i,i-1) \le N \end{cases} \quad \text{(FVDef–2)}$$

A „súlyponti gondolat” a függvény 3-4. ága.

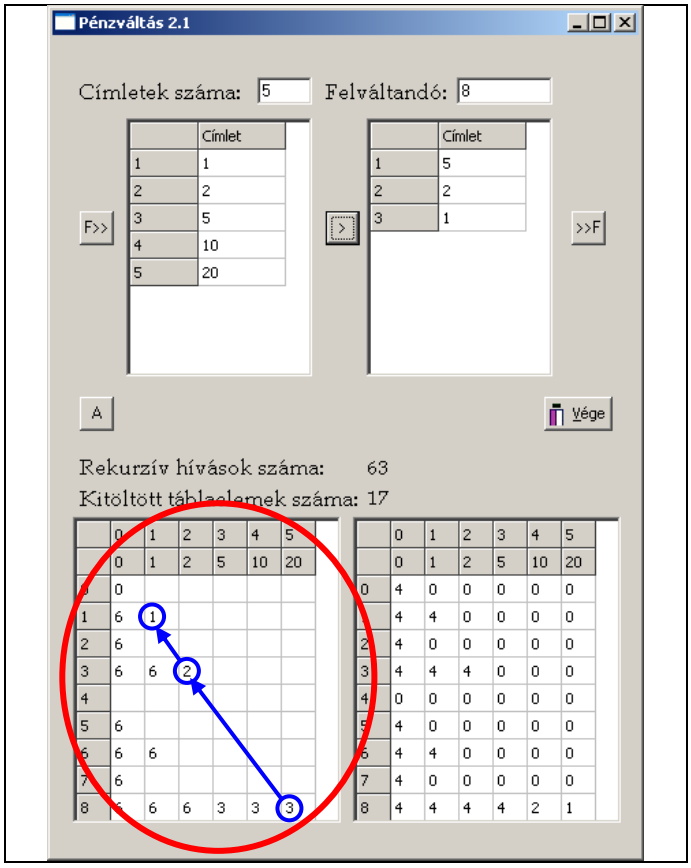
2.2.4 Részproblémák megoldásának kiszámítása

Könnyen belátható, hogy a 0 paraméterek felöl (alulról) az E és N felé haladva kell a táblázatot előállítani.

2.2.5 A megoldás előállítása

A megoldás meghatározása most bonyodalmasabb, mint volt korábban: hiszen most egy indexet kapunk a függvény értékeként (k_1=FV(E,N)), ami csak egy –a legnagyobb indexű elemé– a megoldásból. A táblázatból azonban kiolvasható a következőé (k_2= FV(E–p_{k_1}, k_1–1)), majd az azt követőé éít. Amíg a felváltandó összegparaméter 0-ra nem csökken.
(Lásd az ábrát és hátrébb a kódot.)

Megjegyzés:
A végső megoldást előállító algoritmusban iteratívan fogjuk összeszedni a kellő címleteket, felhasználva az amúgy is létező táblázatot. De tudnunk kell, hogy a tisztán rekurzív megoldásnál természetesen újra rekurzív hívásokkal kellene dolgoznunk.

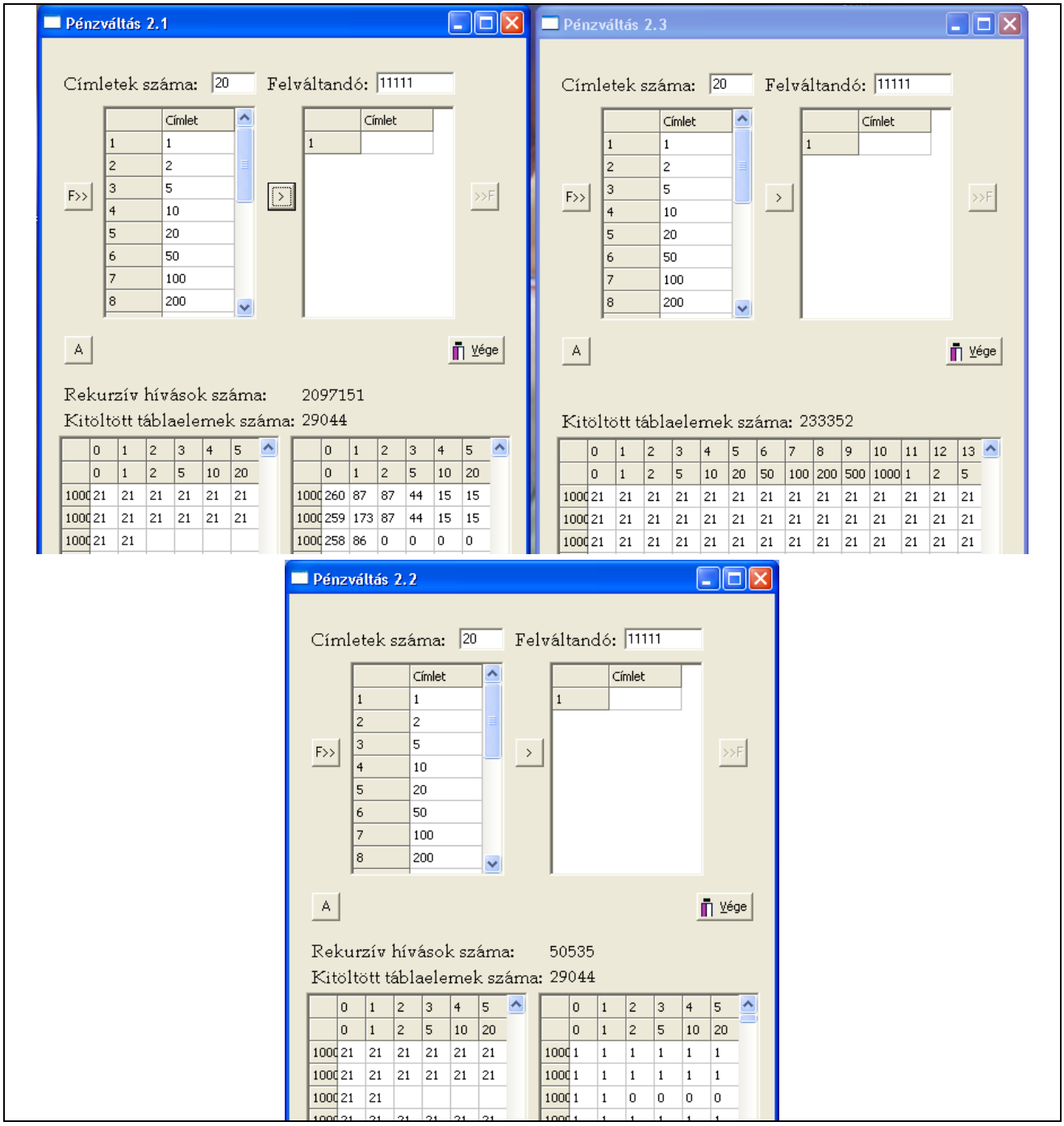


7. ábra. A megoldás előállítása.

Most álljon itt összehasonlításul a már „megszokott” 3 megoldásváltozat:
- a rekurzív,
- a tiszta dinamikus programozású megoldás és
- a kettejük közé, amolyan átmenetként, az „okos”, ún. *memorizálós* változat.

A 8. ábra konklúziói:
1. 2 097 152 rekurzív hívással szemben 233 352 (=(20+1)*(11111+1)) táblaelem-kitöltés áll, ez utóbbiból 204 308 (azaz cc. 87 %) „fölösleges”.
2. 2 097 152 rekurzív hívást memorizálással le lehet csökkenteni 50 535 (nem egészen 2,5 %-ára).
3. Visszagondolva az előző, hasonló feladatot megoldó rekurzív alkalmazás statisztikájára, el kell gondolkodjunk azon, hogy vajon miért növekedett több, mint kétszeresére a rekurzív hívások száma. A választ a megoldást specifikáló függvény 3. ágában találjuk: itt ugyanazon függvényérték kiszámolását kétszeresen írtuk elő; a feltételrészben, és annak igaz értékű esetében a függvényértéket defi-

niáló részben is. Természetesen ezt, mint amolyan lokális hatékonysági problémát egészen könnyen ki lehet küszöbölni.



8. ábra. A 11111 felváltása {1,2,5,10,20,50,100,200,500,1000, 1,2,5,10,20,50,100,200,500,1000} címletekkel rekurzívan, DP-módszerrel, és „memorizálós” DP-módszerrel

Végezetül következzék a „tisztán” DP-megoldást megtestesítő kód lényegi része:

```
procedure PenzValtas_DinProg;
  procedure Tablafeltoltes(const E{felváltandó},N{címletszám}:integer);
    var
      x,i:integer;
  begin
    with fmPenzValtas do
```

```
    begin
      //0. oszlop kitöltése
      for x:=1 to E do sgRekTab.Cells[1,x+2]:=intTostr(N+1);
      sgRekTab.Cells[1,2]:='0';
      //1.-N. oszlop kitöltése
      for i:=1 to N do
      begin
        sgRekTab.Cells[i+1,2]:='0'; //FVDef-2 2. ága
        for x:=1 to E do
        begin
          if strToInt(sgRekTab.Cells[i,x+2])<=N then
          begin
            sgRekTab.Cells[i+1,x+2]:=sgRekTab.Cells[i,x+2] //FVDef-2 3. ága
          end
            else
          begin
            if (x>=Cimletek.cimlet[i]) and
               (strToInt(sgRekTab.Cells[i,x-Cimletek.cimlet[i]+2])<=N) then
            begin
              sgRekTab.Cells[i+1,x+2]:=intToStr(i); //FVDef-2 4. ága
            end
              else
            begin
              sgRekTab.Cells[i+1,x+2]:=intToStr(N+1); //FVDef-2 5. ága
            end;//if x>=
          end;//if strToInt
        end;//for x
      end;//for i
    end;//with
  end;//Tablafeltoltes

  var
    k,kX:integer;

begin//PenzValtas_DinProg
  //táblafeltöltés:
  Tablafeltoltes(Felvaltando,Cimletek.db);
  //címletösszeszedés a táblázatból:
  KiCimletek.db:=0;
  if strToInt(fmPenzValtas.sgRekTab.Cells[Cimletek.db+1,Felvaltando+2])<=
       Cimletek.db then
  begin //van megoldás: legalább egy címlet
    kX:=Felvaltando; k:=Cimletek.db;
    while kX>0 do
    begin
      //kX felváltásához szükséges legnagyobb címlet indexe: k
      k:=strToInt(fmPenzValtas.sgRekTab.Cells[k+1,kX+2]);
      inc(KiCimletek.db); KiCimletek.cimlet[KiCimletek.db]:=Cimletek.cimlet[k];
      kX:=kX-Cimletek.cimlet[k]; //kX: a maradék összeg
      dec(k); //k: k. címlet már nem lehet, legfeljebb kisebb indexű
    end;//while
  end;
end;//PenzValtas_DinProg
```

Most is letölthetjük az elemzett alkalmazásokat: 2.1 (rekurzív), 2.2 (memorizálós), 2.3 (tiszta DP).

3 Második példázat – optimális pénzváltás

3.1 A feladat

Tovább bonyolítjuk a pénzfelválthatóságot vizsgáló feladatot. Most azzal, hogy meg kell adni egy optimális (azaz legkevesebb számú címletet tartalmazó) felbontást. Formalizáljuk:

Bemenete:

- $P=\{p_1,...,p_N\}$ pozitív egészek – a pénzcímletek, és
- E pozitív egész – a felváltandó összeg

Kimenete:

- $S \subseteq P$ – a felváltásban szereplő pénzcímletek; az S halmazban felsorolt címleteket legfeljebb egyszer használhatjuk fel; és S a lehető legkisebb elemszámú.

3.2 A megoldás

Egy kézenfekvő felvetéssel kezdjük. Vajon nem lehet, hogy ezt a problémát az úgynevezett *mohó stratégiá*val oldjuk meg? Vagyis nem lehet-e, hogy a globális optimumot az egyszerűen kiválasztható lokális optimumokon keresztül érhetjük el? A problémánkhoz természetesen illő mohó választás: a még választhatók közül a legnagyobb címlet. Ez esetben könnyű találni példát annak a kézenfekvő mohó választásnak az alkalmatlanságára, hogy „válassz úgy, hogy minél kisebb maradjon a továbbbontásra”: P={5,4,4,1,1,1} és E=8; a mohó megoldás: 8=5+1+1+1, aminél most ránézésre is találunk jobbat: 8=4+4.

3.2.1 A megoldás szerkezetének tanulmányozása

Ua. mint az 1. fejezetben az azonos című résznél.

3.2.2 Részproblémákra és összetevőkre bontás

Jelöljük Opt(X,i)-vel azt a függvényt, amely megadja az X felváltásához szükséges P-beli elemek *számát*, ha P elemeit –eddigi szokásunknak megfelelően– indexük sorrendjében vesszük figyelembe, és csak az első i darab használható föl a céljainkhoz. Ezt a függvényt használjuk majd föl a másik, előző részben megismert, hasonló funkciójúhoz.

Az Opt függvénnyel szembeni elvárásaink:

1. Ha a felhasználható P-elemek száma 0, de még maradt felváltandó, ez azt jelenti, hogy nincs optimális –sőt semmilyen– megoldása a feladatnak. Ezt jelezzük a címletek számánál nagyobb számmal: N+1. Ekkor nem építünk további összetevőre.
2. Ha a felváltandó pénz 0-ra csökkent, akkor célt értünk. További címletek felhasználása nélkül megoldáshoz jutottunk. Ezt a legkisebb darabszámmal, a 0-val fejezzük ki. Most sem építünk további összetevőre.
3. Ha az i. címlet a felbontásban –elvileg– szerepet kaphat, azaz értéke nem nagyobb az X-nél, akkor vagy az általa csökkentett érték felbontás-számánál eggyel nagyobb, vagy a nélküle kapható felbontás-szám lesz a függvény értéke. Nyilván a kisebb jelenti az optimumot. Ekkor visszavezetjük a problémát az $(X–p_i,i–1)$ és $(X,i–1)$ összetevőkre.
4. Ha az i. nem választható nagysága miatt, akkor a függvény értéke az legyen, amit ő nélküle ad. Ez esetben az (X,i–1) összetevőre építkezünk

Adjunk választ a feltett kérdésre! Nézzük, hogyan is definiálhatjuk a megoldást kiszámító FV(X,i) függvényt! A függvény értéke a 2. részben követett szerint a *legnagyobb indexű* felhasználandó indexe legyen.

Ne lepődjünk meg azon, hogy a fenti elvárásaink csak minimálisan módosulnak, a problémavilág ugyanaz, a függvényérték változik „néha”.

1. Ha a felhasználható P-elemek száma 0, de még maradt felváltandó, ez azt jelenti, hogy nincs optimális –sőt semmilyen– megoldása a feladatnak. Ezt jelezzük a címletek számánál nagyobb számmal: N+1. Ekkor nem építünk további összetevőre.
2. Ha a felváltandó pénz 0-ra csökkent, akkor célt értünk. További címletek felhasználása nélkül megoldáshoz jutottunk. Ezt a legkisebb darabszámmal, a 0-val fejezzük ki. Most sem építünk további összetevőre.
3. Ha az i. címlet a felbontásban –elvileg– szerepet kaphat, azaz értéke nem nagyobb az X-nél, és az általa csökkentett érték felbontás-száma plusz egy kisebb, mint a nélküle kapható *felbontás-szám*, akkor i lesz a függvény értéke. Így visszavezetjük a problémát az $(X–p_i,i–1)$ és $(X,i–1)$ összetevőkre, de vegyünk észre egy szokatlan körülményt: itt az összetevők felhasználása az Opt-on keresztül (annak hívásával) történik.
4. Ha az i. nem választható nagysága miatt, akkor a függvény értéke az legyen, amit ő nélküle ad. Ez esetben az $(X,i–1)$ összetevőre építünk.

A 3. pontban szerepel némi újdonság, hiszen itt bukkan föl csak az index az addigi darabszám helyett.

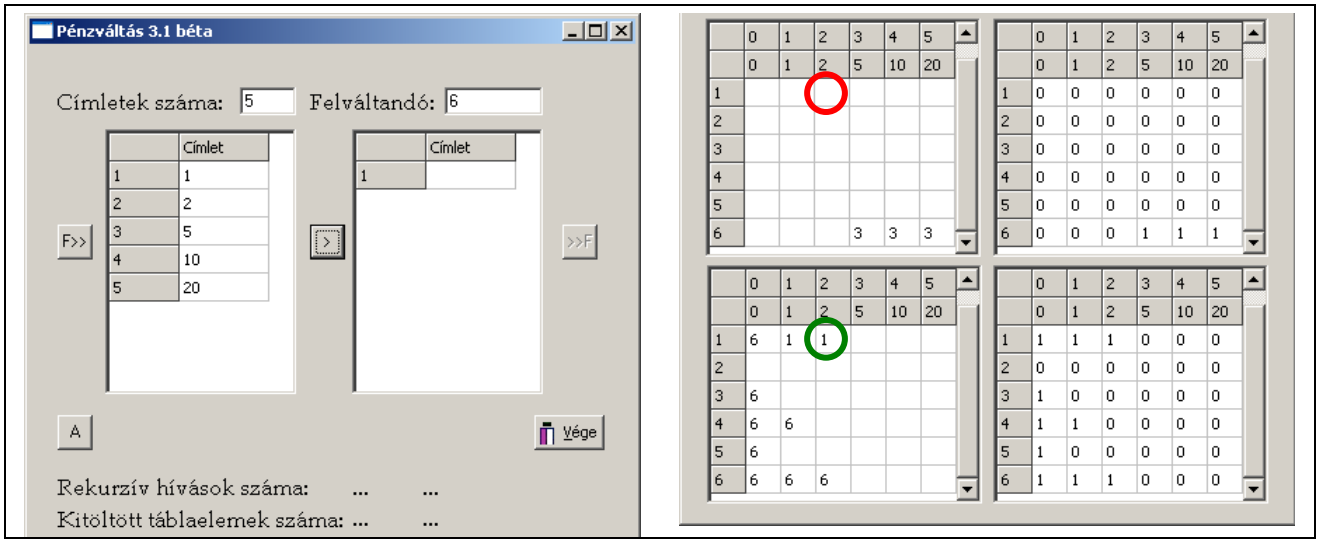
3.2.3 Részproblémák megoldásának kifejezése

A fentieket öntjük formulába. Először az Opt függvényt:

$$\mathrm{Opt}(X,i) := \begin{cases} N+1 & , \text{ha } i = 0 \wedge X > 0 \\ 0 & , \text{ha } X = 0 \\ \mathrm{Min}(1 + \mathrm{Opt}(X - p_i, i-1), \mathrm{Opt}(X, i-1)) & , \text{ha } p_i \le X \\ \mathrm{Opt}(X, i-1) & , \text{ha } p_i > X \end{cases} \qquad \text{(OptFVDef)}$$

majd az FV függvényt:

$$\mathrm{FV}(X,i) := \begin{cases} N+1 & , \text{ha } i = 0 \wedge X > 0 \\ 0 & , \text{ha } X = 0 \\ i & , \text{ha } p_i \le X \wedge 1 + \mathrm{Opt}(X - p_i, i-1) < \mathrm{Opt}(X, i-1) \\ \mathrm{FV}(X, i-1) & , \text{ha } p_i > X \end{cases} \qquad \text{(FVDef–3)}$$



9. ábra. A rekurzív megoldás – problémáját demonstráló screen-shot.
A futás adminisztrálását végző táblapár nincs szinkronban. Bajt okoz, hogy a bal-felső, az FV-hez tartozó tábla (1,2) eleme üres. Pedig az alkalmazás foglalkozott vele, hiszen a bal-alsó, az Opt-hoz tartozó tábla (1,2) cellája helyesen 1 értékű.

Mielőtt a dinamikus megoldás(ok)ra térnénk érdemes az alábbi észrevételt tennünk.

A rekurzív megoldást aprólékosabb vizsgálatnak alávetve megfigyelhetjük, hogy az FV-re fent adott definíció bármennyire is logikus, a ráépített, korábban alkalmazott, a címleteket összeszedni igyekvő, visszafelé haladó módszer elakad. Ugyanis, amikor az FV definíciójának optimumra hivatkozó nem rekurzív (3.) ágánál tartunk, az „készen kapja" a döntéséhez szükséges információt. Ezért aztán nem bontogatja le sem a $(X–p_i,i–1)$, sem a $(X,i–1)$ részproblémákra, azaz a megoldás alapjául szolgáló, adminisztratív táblázat megfelelő elemei kitöltetlenek maradnak. Ez a záró, a címleteket összeszedő részben katasztrófát okoz. (Lásd a 9. ábrát.)
Ezért az optimum kiszámításra olyan többlet, „testidegen" adminisztrációt kell bízni, amely során a másik (az FV-hez tartozó) táblázatba is belepiszkál: feljegyzi az általa e pillanatban még ismert információt, hogy hányadik címlet szerepel a felbontásban[4]. A megoldás középpontjában szereplő két függvényt alább részletezzük:

```
function Opt(const X{felváltandó},i{max.index}:integer):integer;
  var
    k,ki,ii:integer;
begin
  if
    (i=0) and (X>0) then //nincs megfelelő címlet: nem lehet felbontani
      begin
        k:=Cimletek.db+1;
        ii:=Cimletek.db+1;//a felbontásban szereplő címlet: nincs
      end else if
    X=0 then //a felbontás "csont nélkül" sikerült: nem kell folytatni
      begin
        k:=0;
        ii:=Cimletek.db+1;//a felbontásban szereplő címlet: nincs
      end else if
    (Cimletek.cimlet[i]>X) then //az i. címlet nem alkalmas
      begin
        k:=Opt(X,i-1);
        ii:=strToInt(fmPenzValtas.sgRekTab.Cells[i+1-1,X+2]);
                       //a felbontásban szereplő címlet: ugyanaz, mint előbb
      end
    else
      begin
        ii:=1+Opt(X-Cimletek.cimlet[i],i-1);
        ki:=Opt(X,i-1);
        if ii<ki then
        begin
          k:=ii;
          ii:=i;//a felbontásban szereplő címlet: i.
        end
          else
        begin
          k:=ki;
          ii:=strToInt(fmPenzValtas.sgRekTab.Cells[i+1-1,X+2]);
                         //a felbontásban szereplő címlet: ugyanaz, mint előbb
        end;//if
      end
  {If};
  //táblaelem kitöltés:
  fmPenzValtas.sgOptRekTab.Cells[i+1,X+2]:=intToStr(k);
  //a másik (címlet-)táblába írás:
  fmPenzValtas.sgRekTab.Cells[i+1,X+2]:=intToStr(ii{címlet-sorszám});
  Opt:=k
end;//Opt
```

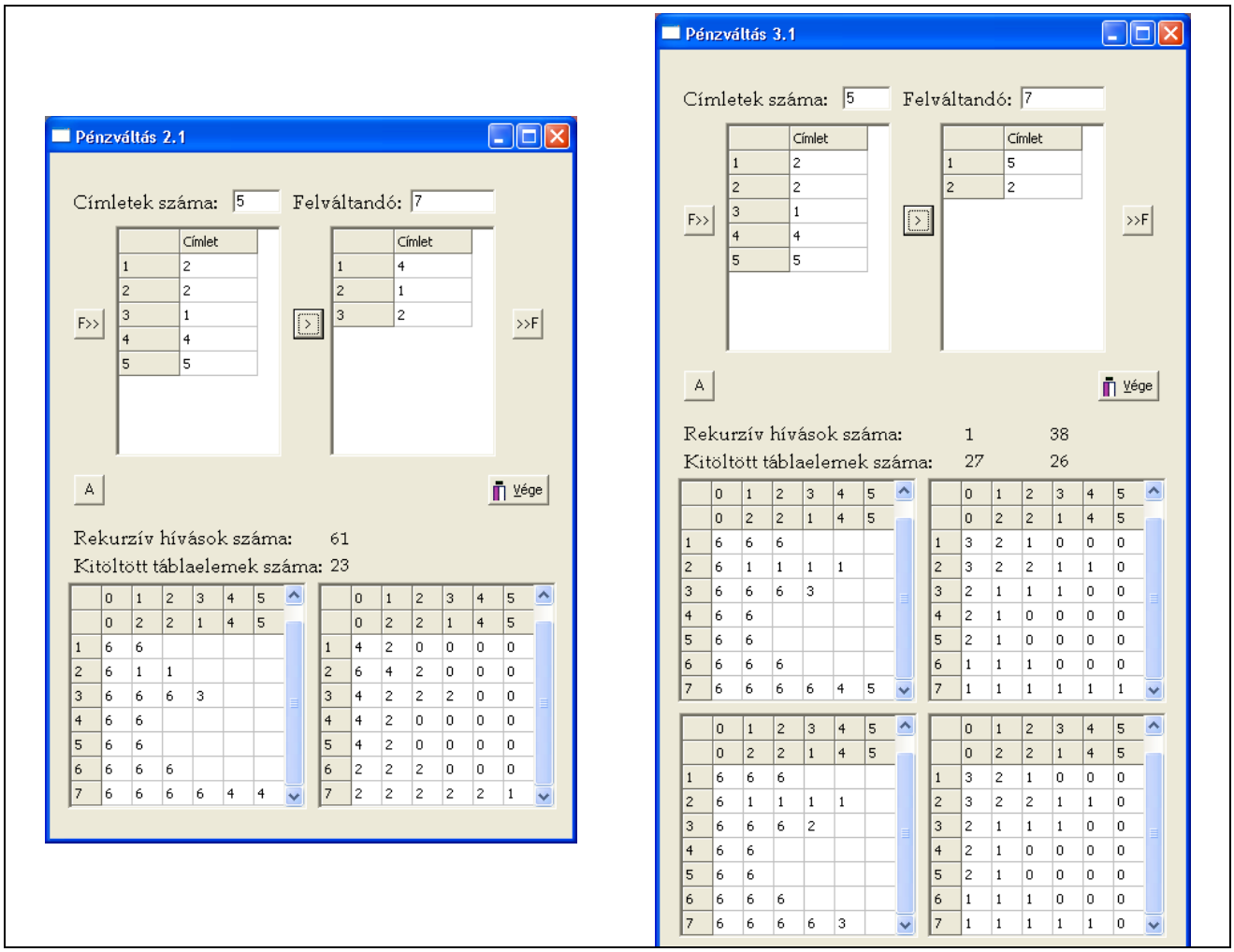
[4] A programban ii segédváltozó tartalmazza ezt a többlet információt.

```
function FV(const X{felváltandó},i{max.index}:integer):integer;
  var
    k:integer;
begin
  if
    (i=0) and (X>0) then //nincs megfelelő címlet: nem lehet felbontani
      begin
        k:=Cimletek.db+1
      end else if
    X=0 then //a felbontás "csont nélkül" sikerült: nem kell folytatni
      begin
        k:=0
      end else if
    (Cimletek.cimlet[i]<=X) and
      //az i. címlet alkalmas és a
    (1+Opt(X-Cimletek.cimlet[i],i-1)<Opt(X,i-1))
      //maradék optimális felbontása jobb, mint az i. nélkül
        then
      begin
        k:=i
      end
    else //eggyel kisebbig van -esetleg- optimális felbontás
      begin
        k:=FV(X,i-1)
      end
  {If};
  //táblaelem kitöltés:
  fmPenzValtas.sgRekTab.Cells[i+1,X+2]:=intToStr(k);
  V:=k
end;//FV
```

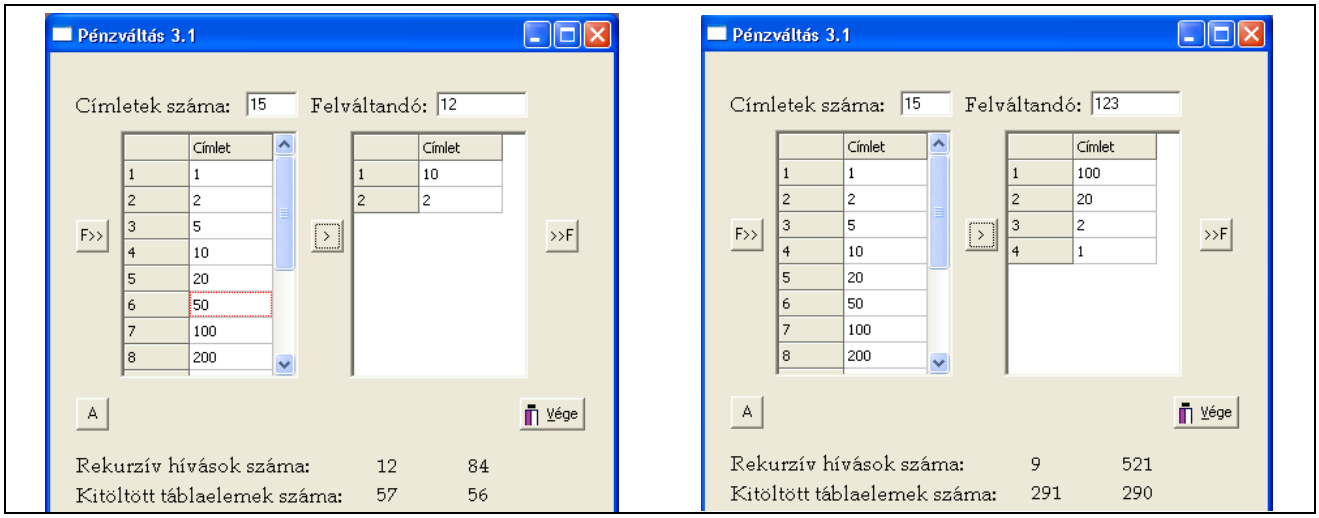
A futás eredményeit szemlélteti a 10-12. ábra.

Néhány észrevétel a futás(ok)hoz:

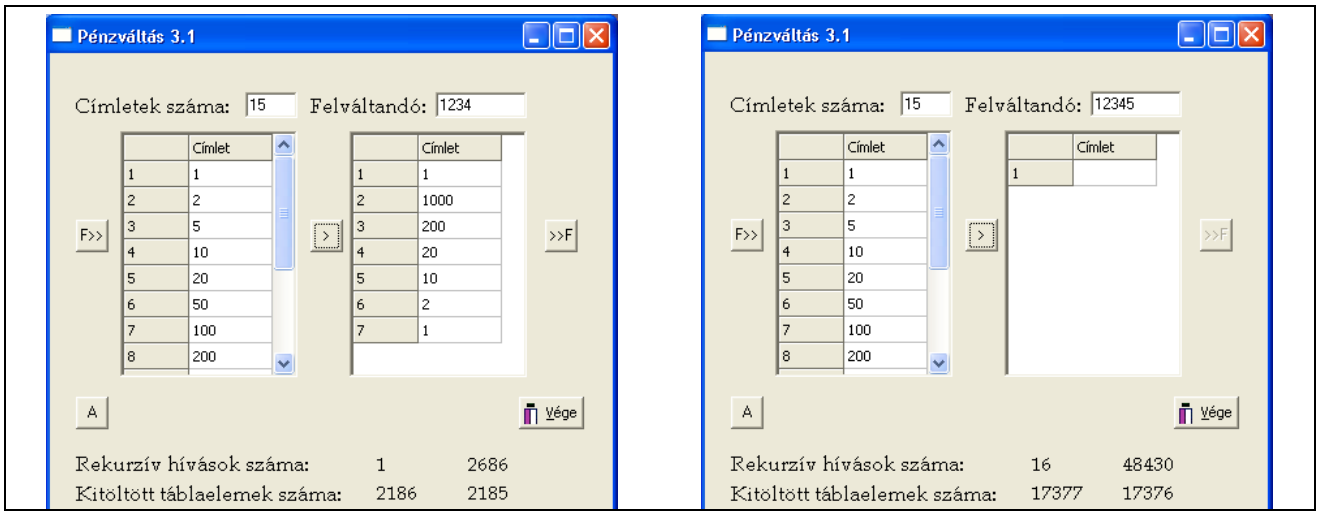
1. A kitöltött táblaelemek száma valamivel több, mint a nem optimális megoldást megtaláló párja esetében. Mindkét táblát figyelembe véve valamivel több, mint kétszerese.
2. Az FV rekurzív hívások száma minimális. A lényegi számítást (az FV-hez tartozó táblázat kitöltését) döntően az Opt függvény végzi.
3. Talán meglepő, de a rekurzív hívások száma kisebb az optimumra nem törekvőénél.
4. Az biztosan váratlan, hogy sokkal kevésbé drasztikusan nő az összes rekurzív hívások száma az előbbinél.
5. A kitöltött táblaelemek száma X növekedtével relatíve csökken. (10-11. ábra szerint e kitöltöttségi ráta: 57/(15*12)≈31%, 291/(15*123)≈15%, 2186/(15*1234) ≈ 11%,17377/(15*12345) ≈9%)



10. ábra. A nem optimális (2.1) és az optimális (3.1) megoldást előállító rekurzív alkalmazások (azonos bemenetre). A 3.1 alkalmazás alsó adminisztratív táblázata az Opt értékeit (baloldalon alul) és hívásainak számát (jobboldalon alul) tartalmazza.



11. ábra. A rekurzív hívások növekedése az optimum-kereső algoritmus esetén. – 1.



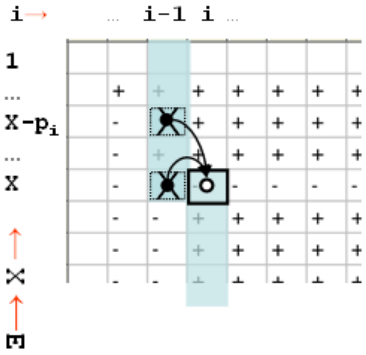
12. ábra. A rekurzív hívások növekedése az optimum-kereső algoritmus esetén. – 2.

3.2.4 Részproblémák megoldásának kiszámítása

Mind a „tiszta”, mind a memorizálós DP megoldás először a táblázat generálását végzi el (a (OptFVDef) és a (FVDef–3) képletek szerint), majd a kész táblázatból olvassa ki, szedi össze a megoldást a 3.2.5-ben leírtakhoz hasonlóan.

A részproblémák megoldásának kiszámítása, azaz a táblázat kitöltése a kétféle dinamikus programozású eljárásban kétféleképpen történik. A memorizálós a táblaelemeket a rekurzív hívás-sorrend által vezérelve –a fölösleges hívások elkerülésével–, ezzel szemben a tisztán iteratív szemléletű a címletek szerint *alulról fölfelé* haladva, de az érték szerint *fordítva*. A fordított irányt a következők indokolják:

- az optimumot kiszámoló függvényhez tartozó *táblázatra nincs szükség* a végeredmény –későbbi– generálásához;
- mivel *csak az előző* címlet oszlopára hivatkozik, elegendő azt és az éppen számítás alatt állót megtartani, sőt
- *egyetlen oszlop is elegendő*[5], ha a kiszámítás sorrendjét megfordítjuk, azaz az „előző” oszlop nem nagyobb indexű értékeire, „előre-hivatkozva” [(X–0,i–1),(X–p_i, i–1), l. (OptFVDef) 3. ágát] haladunk a kisebb E-értékek felé.



Így alakult ki az alábbi kód a dinamikus programozás lényegét jelentő táblázatfeltöltő eljáráshoz:

```
procedure Tablafeltoltes(const E{felváltandó},N{címletszám}:integer);
  var
    x,i,Rop:integer;
begin
  with fmPenzValtas do
  begin
    //0. oszlop kitöltése
    for x:=1 to E do
    begin
      sgOptRekTab.Cells[1,x+2]:=intTostr(N+1); //OptFVDef 1. ága
      sgRekTab.Cells[1,x+2]:=intTostr(N+1); //FVDef-3 1. ága
    end;//for x
    sgOptRekTab.Cells[1,2]:='0'; //OptFVDef 2. ága
```

[5] … lenne, de mi megtartjuk a változatok összevethetősége érdekében …

```
      //1.-N. oszlop kitöltése
      for i:=1 to N do
      begin
        for x:=E downto 1 do
        begin
          if x>=Cimletek.cimlet[i] then
            Rop:=strToInt(sgOptRekTab.Cells[1,x-Cimletek.cimlet[i]+2])+1
          else
            Rop:=N+1
          {EndIf x};
          if Rop<strToInt(sgOptRekTab.Cells[1,x+2]) then
          begin
            sgOptRekTab.Cells[1,x+2]:=intToStr(Rop);
            sgRekTab.Cells[i+1,x+2]:=intToStr(i)
          end
            else
          begin
            sgRekTab.Cells[i+1,x+2]:=sgRekTab.Cells[i,x+2]
          end{if Rop};
        end;//for x
      end;//for i
    end;//with
  end;//Tablafeltoltes
```

3.2.5 A megoldás előállítása

Ahogy a 2. feladat esetében láttuk (2.2.5), a végső válasz előállítása úgy történik, hogy a FV táblázat (E,N)-beli értékéből visszafelé haladva lépésről, lépésre olvassuk ki a címletek sorszámát, ebből az értékét… (Lásd a korábbi kóddarabban.)

Néhány futási eredménnyel zárul a fejezet.

Az alábbi táblázat összefoglalja a rekurzív és a memorizálós DP alkalmazás rekurzív hívásainak számát a {1,2,5,10,20,50,100,200,500,1000,1,2,5,10,20,50,100,200,500,1000} címletek esetén. A számpárok elsője a *felváltás* (az FV), a második az *optimális elemszám* (az Opt) számára szükséges rekurzív hívások számát jelöli.

E	Rekurzív	Memorizálós DP	E	Rekurzív	Memorizálós DP
1	20 + 15	11 + 13	111	14 + 21 615	5 + 1 449
6	18 + 196	9 + 74	1111	11 + 1 309 367	2 + 15 567
11	17 + 513	8 + 139	11111	21 + 4 194 260	2 + 50 534

Most is letölthetjük az elemzett alkalmazásokat: 3.1 (rekurzív), 3.2 (memorizálós), 3.3 (tiszta DP).

4 Harmadik példázat – tükörszavak

4.1 A feladat

A tükörszó (palindrom) egy szimmetrikus karaktersorozat, azaz balról jobbra és jobbról balra olvasva azonos. Adjuk meg, hogy minimum hány jel beillesztésével lehet egy szót tükörszóvá alakítani. Formálisabban:

Bemenete:

- szó=(b_1,...,b_N) szó – jelek sorozata

Kimenete:

- bj természetes szám – minimálisan ennyi jel beillesztésével tehető a szó palindrommá.

4.2 A megoldás

4.2.1 A megoldás szerkezetének tanulmányozása

Jelöljük TSz(S)-sel azt a szöveget, amely a minimális számú jel beillesztésével képződik az S-hez. Ilyen biztosan létezik, hiszen S&S' tükörszó, ahol S' az S megfordítását jelöli (csak legfeljebb nem a legrövidebb, ami S-hez hozzá rendelhető). Vizsgáljuk meg a következő eseteket!

a) Ha S (legfeljebb) egy jelből áll, akkor maga is tükörszó, azaz TSz(S)=S.
b) Legyen S=x&R&y, ahol x, y az S első és utolsó jele, és R akár üres is lehet.
   (i) Ha x=y, akkor TSz(S)=x&TSz(R)&y.
   (ii) Ha x≠y, akkor TSz(S) első és utolsó jele vagy x vagy y.

   Ui. tegyük föl indirekt: TSz(S)=z&U&z miközben z≠x és z≠y (ilyennek kell lennie, hiszen tükörszó). Ekkor U is tükörszó, továbbá a TSz(S) legalább a két z U-hoz vételével keletkezett. Vagyis U is megoldás lett volna az S-hez. Tehát a z&U&z nem minimális. Ellentmondásra jutottunk.

   α. Ha TSz(S)=x&U&x, ekkor x-et szúrtunk be a végére, azaz U=TSz(R&y).
   β. Ha TSz(S)=y&U&y, ekkor y-t szúrtunk be az elejére, azaz U=TSz(x&R).

   Vagyis U (ami eggyel rövidebb S-nél) gyanánt a TSz(R&y) és a TSz(x&R) közül azt kell választani, amely kevesebb beszúrással kapható meg.

4.2.2 Részproblémákra és összetevőkre bontás

A szerkezeti vizsgálatból látszik módszerünk madártávlati vázlata: hol a szó elejéről, hol a végéről hagyunk el jelet (a fenti feltételeknek megfelelően), s így folytatjuk az ily módon redukált szóval a vizsgálatot.

Jelöljük MSz(i,j)-vel a minimális jel-beillesztések számát és szó(i..j)-vel a szó szöveg i. és j. jele közötti részét, és szó(i) legyen a szó(i..i) rövidítése! Ekkor az MSz(i,j) részproblémák és a szó(i..j) összetevő jelsorozatok között a következő kapcsolatokat körvonalazhatjuk:

a) Ha i≥j, akkor üres szöveg, vele nincs mit tenni.
b) Ha i<j és
   (i) szó(i)=szó(j), akkor jel beillesztések számát az i. és j. elhagyásával kapható MSz(i+1,j–1) részprobléma határozza meg.
   (ii) szó(i)≠szó(j), akkor az MSz(i+1,j) és az MSz(i,j–1) közül a kisebb értékűt kell választanunk.

A kiinduló probléma e jelölésekkel: MSz(1,Hossz(szó)).

4.2.3 Részproblémák megoldásának kifejezése

A fentiek formalizálása:

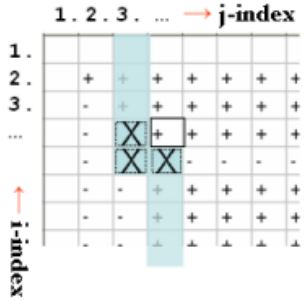
$$MSz(i,\ j) := \begin{cases} 0 & , \text{ha } i \geq j \\ MSz(i+1, j-1) & , \text{ha } i < j \wedge szó(i) = sz(j) \\ 1 + Min(MSz(i+1, j), MSz(i, j-1)) & , \text{ha } i < j \wedge szó(i) \neq sz(j) \end{cases} \quad \text{(MSzDef)}$$

4.2.4 Részproblémák megoldásának kiszámítása

Az (i,j)-vel azonosított részprobléma az (i+1,j–1), az (i+1,j) és az (i,j–1) részproblémáktól függ. Tehát a kiszámítási sorrend i-értelemben fogyó, j-értelemben növekvő lehet. Ennek figyelembe vételével lehet a táblázat kitöltéséhez fogni.

Ezzel a dinamikus programozású megoldást elkészítettük. Érdemes azonban továbbgondolnunk. Mivel a végeredményt egy szám, a táblázat (1,Hossz(szó)) index-párral kijelölt eleme, és a rekurzív összefüggés csak a közvetlen szomszédságban {(i+1,j–1),(i+1,j),(i,j–1)} „kotorász”, ezért a megfelelő sorrendben történő számítás esetén *egyetlen egy oszlop*ának tárolása is elegendő.

Így:

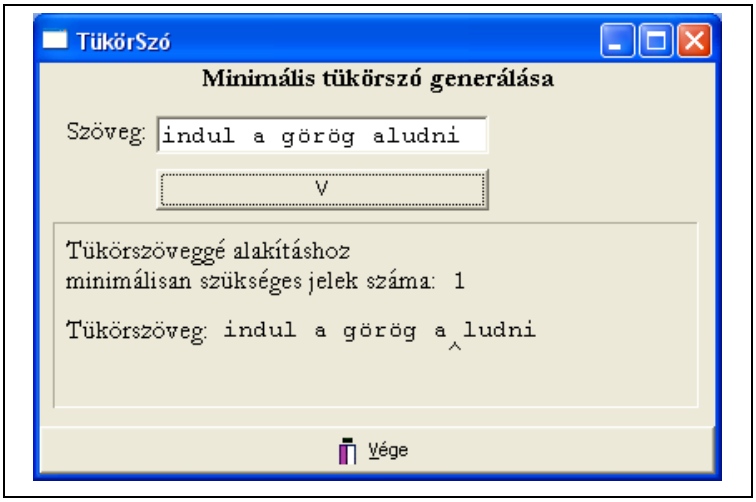



```
...
Function TukorSzo(const s:string):integer;
  var T:array [1..MaxN] of integer;
      i,j,ment,menti:integer;
begin
  T[1]:=0;//MSz(1,1)<-0
  for j:=2 to N do
  begin
    T[j]:=0; menti:=0;//MSz(j,j)<-0
    for i:=j-1 downto 1 do
    begin
      //T(i)<-MSz(i,j) számítása
      ment:=T[i]
      if s[i]=s[j] then
        T[i]:=menti //MSz(i,j)<-MSz(i+1,j-1)
      else
        T[i]:=1+Min(T[i],T[i+1]) //MSz(i,j)<-1+Min(MSz(i,j-1),MSz(i+1,j))
      {EndIf};
      menti:=ment;
    end;//for i
  end;//for j
  TukorSzo:=T[1] //TukorSzo<-Msz(1,N)
end;//TukorSzo
```

4.2.5 A megoldás előállítása

Itt nem nagy kaland: a tábla (1,Hossz(szó)) eleme, amelyet az algoritmusbeli T vektor első eleme tartalmaz.

A feladat teljes megoldását (kiegészítve a tükörszóval, sőt azon helyek megjelölésével, ahol a beillesztés megtörtént) letölthetjük: rekurzívat (zip), memorizálósat (zip).



13. ábra. A minimális jel-beillesztéssel kapható tükörszó.

5 Negyedik példázat – a persely minimális tartalma

5.1 A feladat

Mohó Marci malacperselyben gyűjti a pénzét. Csak fémpénzeket rakott a perselybe, de nem jegyezte föl, hogy milyeneket. Felírta azonban az üres persely súlyát, így meg tudja állapítani a perselyben levő pénzek összsúlyát. Ismeri továbbá az egyes pénzérmék egyedi súlyát és értékét. Szeretné kiszámítani, hogy mennyi az a legkisebb összeg, amelyet a perselye biztosan tartalmaz. Egy adott típusú pénzérméből –természetesen– több is lehet a perselyben. Adjuk meg a címletezést is!

Bemenete:

- pSúly pozitív egész – a perselyben lévő pénz össz-súlya
- érmék={p_1,...,p_N} a pénzcímletek – a p_i=(s_i,$é_i$) súly és érték kettőse

Kimenete:

- minÖsszeg pozitív egész – a minimális összeg, és
- kiÉrmék={kp_1,...,kp_{Db}} a pénzcímletek – kp_i=(db_i,$é_i$) darab és érték kettőse.

5.2 A megoldás

5.2.1 A megoldás szerkezetének tanulmányozása

Tételezzük fel, hogy van megoldás, akkor

(1a) pSúly = s_{i_1}+…+ s_{i_k} (i_1≤…≤i_k megengedve *ugyanazon címlet többszörös* előfordulását), és

(1b) minÖsszeg = $é_{i_1}$+…+ $é_{i_k}$.

Ugyanez rövidebben, a végeredményhez jobban közelítően:

pSúly = n_{i_1}*s_{i_1}+…+ n_{i_k}*s_{i_k} (i_1<…<i_k; n_{i_1},… n_{i_k}>0 egészek), és

minÖsszeg = n_{i_1}*$é_{i_1}$+…+ n_{i_k}*$é_{i_k}$.

Az a-esetben világos, hogy

(2a) pSúly – s_{i_k} = s_{i_1}+…+ s_{i_m} (m=k vagy m=k–1) és

(2b) minÖsszeg – $é_{i_k}$ = $é_{i_1}$+…+ $é_{i_m}$

annak a redukált feladat megoldása, amelyben az össz-súlyt az s_{i_k} rögzített címlet súlyával csökkentettük, de a továbbiakban választható címletek vagy változatlanul i_k-ig, vagy az azt megelőzőig terjednek.

5.2.2 Részproblémákra és összetevőkre bontás

A részfeladatokat jellemezhetjük az X össz-súllyal és a felhasználható utolsó címlet i indexével: (X,i). A kiinduló feladat a (pSúly,N)-nel jellemzett.

Figyelemmel az 5.2.1-ben írtakra így lehetővé tesszük, hogy egy részfeladatot csökkentett jellemzőjű részfeladatokra bontsuk. Vagyis előkészítettük a rekurzív definiálást.

5.2.3 Részproblémák megoldásának kifejezése

Jelölje PMO a perselyben lévő minimális összeget kiszámító függvényt, amely paramétere egy (X,i)-vel jellemzett részfeladat! A kiszámítása nagyjából a következő gondolatmenetű lesz. A hangsúlyt különösen a rekurzív részproblémákra helyezzük.

1. Ha az X össz-súly 0, akkor az eredmény is az.: PMO(X,i)←0.
2. Ha az össz-súly nem fogyott el miközben a címletsorszám 0-ra csökkent, ez biztos jele a probléma megoldhatatlanságának. Ekkor e tényt olyan értékkel jelezzük, amely összetéveszthetetlen bármilyen összeggel.: PMO(X,i)←∞.

3. Ha az aktuális (a felhasználhatók utolsója) címlet felhasználható –súlya szerint–, akkor vagy a csökkentet súlyértékből kapott minimumhoz kell számítani az értékét, és lehetővé téve az újra felhasználást (PMO(X,i)←PMO(X–s_i,i)+e_i), vagy elhagyva őt, az előzőt tekintve utolsónak vezetjük elemibb részfeladatra vissza (PMO(X,i)←PMO(X,i–1)).
4. Utolsó eset az, hogy az aktuális címlet súlya nem teszi lehetővé a felhasználást, ekkor egyszerűen kihagyjuk, s így vezetjük vissza: PMO(X,i)←PMO(X,i–1).

A PMO függvény tömörebb, formális leírása a következő (figyelembe véve az algoritmizálás sorrendi szempontjait):

$$PMO(X,i) := \begin{cases} \infty & , \text{ha } X > 0 \wedge i = 0 \\ 0 & , \text{ha } X = 0 \\ Min(PMO(X - s_i, i) + é_i, PMO(X, i-1)) & , \text{ha } X \geq s_i \\ PMO(X, i-1) & , \text{egyébként} \end{cases} \quad \text{(PMODef)}$$

Az első lépésen vagyunk csak túl: a minimum összeg kiszámításán. A következő feladat: a kellő címletek előállítása. Azt kell feljegyeznünk, hogy milyen címletet választ a függvény az egyes rekurzív hívás esetében. Egyetlen ponton, a definíció 3. ágán választ címletet (s_i,$é_i$). Ha itt feljegyezzük az adott X-hez választott címlet indexét, akkor megtartható a címletekre vonatkozó, egyébként elvesző információ. Arra azonban ügyelni kell, hogy adott X-hez több úton (több i-vel is) eljuthat a számítás, így csak a minimális értékhez tartozó választást szabad megőrizni. Megoldható ez, ha minden X-hez nemcsak az aktuális i indexet, hanem a hozzá tartozó minimumértéket is könyveljük. Természetesen könyvelni csak a „jobbat” szabad!

A megoldó *rekurzív* függvény kódja a következő lehet:

```
function PMO(x,i:integer):integer;//rekurzív fv.
  var
    min1,min2:integer;
begin
  if
    (i=0) and (x>0) then
      begin
        PMO:=MaxErt;
      end else if
    x=0 then
      begin
        PMO:=0;
      end else if
    x>=Cimletek.cimlet[i].suly then
      begin
        min1:=PMO(x-Cimletek.cimlet[i].suly,i)+Cimletek.cimlet[i].ertek;
        min2:=PMO(x,i-1);
        if min1<min2 then
        begin
          PMO:=min1;
          //a kiválasztott címlet feljegyzése:
          if Valaszt[x].min>min1 then//az eddiginél jobb
          begin
            Valaszt[x].i:=i; Valaszt[x].min:=min1;
          end;//if
        end
          else
        begin
          PMO:=min2;
        end;//if min1
      end
```

```
      else
        begin
          PMO:=PMO(x,i-1);
        end
    {endIf};
end;//PMO
```

5.2.4 Részproblémák megoldásának kiszámítása

A *dinamikus programozás*os megoldás azon változatát mellékeljük, amely a rekurzívból könnyedén származtatható: a *memorizálós*at. Lássuk mindenfajta kommentár nélkül a kódját, megjelölve (piros betűszínnel) az új részeket:

```
function PMO(x,i:integer):integer;//memorizálós rekurzív fv.
  var
    min1,min2:integer;
begin
  if PMOT[x,i]<>-1 then//már számolt érték
  begin
    PMO:=PMOT[x,i];
  end
    else//még nem számolt érték
  begin
    if
      (i=0) and (x>0) then
        begin
          PMOT[x,i]:=MaxErt; PMO:=MaxErt;
        end else if
      x=0 then
        begin
          PMOT[x,i]:=0; PMO:=0;
        end else if
      x>=Cimletek.cimlet[i].suly then
        begin
          min1:=PMO(x-Cimletek.cimlet[i].suly,i)+Cimletek.cimlet[i].ertek;
          min2:=PMO(x,i-1);
          if min1<min2 then
          begin
            PMOT[x,i]:=min1; PMO:=min1;
            //a kiválasztott címlet feljegyzése:
            if Valaszt[x].min>min1 then//az eddiginél jobb
            begin
              Valaszt[x].i:=i; Valaszt[x].min:=min1;
            end;//if
          end
            else
          begin
            PMOT[x,i]:=min2; PMO:=min2;
          end;//if min1
        end
      else
        begin
          PMOT[x,i]:=PMO(x,i-1); PMO:=PMOT[x,i];
        end
    {endIf};
  end;//if PMOT
end;//PMO
```

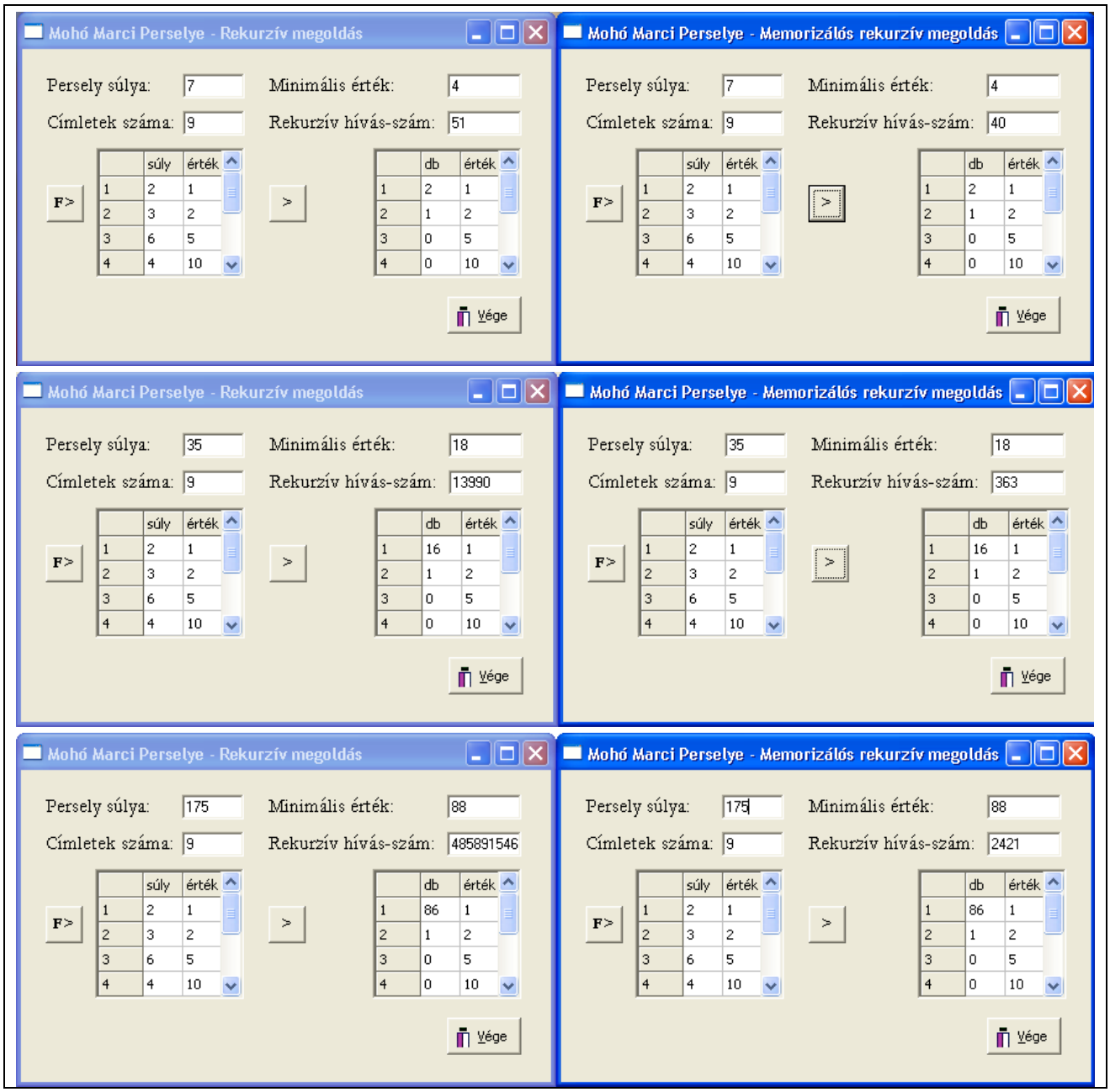
5.2.5 A megoldás előállítása

A Valaszt tömb felhasználásával generálható a megoldásként szolgáló címletsorozat. Az alábbiakban közöljük az egész megoldást jelentő kódrészletet, amely tartalmazza a címletek összeszedésének részleteit is:

```
…
begin//PerselybeliMinimalisOsszeg_MemDin
  //Valaszt tömb létrehozása, inicializálása:
  SetLength(Valaszt,perselySuly+1);
  for i:=0 to perselySuly do
  begin
    Valaszt[i].i:=0; Valaszt[i].min:=MaxInt;
  end;//for
  //MPOT tömb létrehozása, inicializálása:
  //dinamikus deklarálás: sorok számának megadása:
  SetLength(PMOT,perselySuly+1);
  //dinamikus deklarálás: oszlopszámának megadása:
  for i:=0 to perselySuly do
  begin
    SetLength(PMOT[i],cimletek.db+1);
    for j:=0 to cimletek.db do
      PMOT[i,j]:=-1
  end;//for i
  MaxErt:=MaxInt-MaxOsszeg(Cimletek);//a kudarchoz tartozó (max) érték
  rekH:=0;//rekurzív hívások száma: 0
  minOsszeg:=PMO(perselySuly,cimletek.db);
  if minOsszeg=MaxErt then//nincs megoldás
  begin
    minOsszeg:=-1
  end
    else//van megoldás
  begin
    //KiCimletek.cimlet tömb létrehozása, inicializálása:
    SetLength(KiCimletek.cimlet,perselySuly{div Min...}+1);
    for i:=0 to perselySuly do
    begin
      KiCimletek.cimlet[i]:=0;
    end;//for
    //KiCimletek kitöltése:
    j:=perselySuly; KiCimletek.db:=0;
    while j>0 do
    begin
      i:=Valaszt[j].i;
      inc(KiCimletek.db);
      KiCimletek.cimlet[KiCimletek.db]:=Cimletek.cimlet[i].ertek;
      dec(j,Cimletek.cimlet[i].suly)
    end;//while
  end;//if minOsszeg
end;//PerselybeliMinimalisOsszeg_MemDin
```

Most is letölthetjük a hivatkozott alkalmazásokat: a rekurzívat (zip), memorizálóst (zip).

Néhány párhuzamos futási eredménnyel zárjuk a feladat 1., „előírásos” megoldását, amelyeket a következő ábrán találunk.



14. ábra. A perselybeli minimális pénzt meghatározó rekurzív és memorizálós alkalmazás összevetése.

5.3 Egy másik megoldás

5.3.1 Cél

Ha a dinamikus megoldást kiszolgáló tömb méretét sokalljuk, a következő –az eddigiektől elvében eltérő– megoldással segíthetünk ezen. Most nagyvonalakban vázoljuk ezt a megoldást, s a szokásos, akkurátus „levezetést” elhagyjuk.

5.3.2 A kiinduló ötlet

Az ötlet alapja: olyan rekurzív összefüggést kell találnunk, amely csak egy paraméteres. Nem lehet kétségünk afelől, hogy az X összegparamétert nem küszöbölhetjük ki. A felhasználandó címleteket azonosító indexparaméter azzal a halovány tervvel esetleg eliminálhatjuk, hogy az összefüggésben az *összes címletet*, a maguk *egészében* vesszük figyelembe.

5.3.3 A rekurzív megoldó függvény

Segítségünkre lehet, ha megfogalmazzuk a feladatot kicsit formálisabban. Így:

$$PMO(X) := Min\left\{\sum_i é_i \mid \sum_i s_i = X\right\}$$

Átfogalmazzuk most rekurzívra. Az alábbi összefüggés kellően áttekinthető ahhoz, hogy különösebb bevezetést ne igényeljen. Jól látszik rajta, hogy mi módon veszi figyelembe –tervünkhöz híven– *globálisan* az összes címletet.

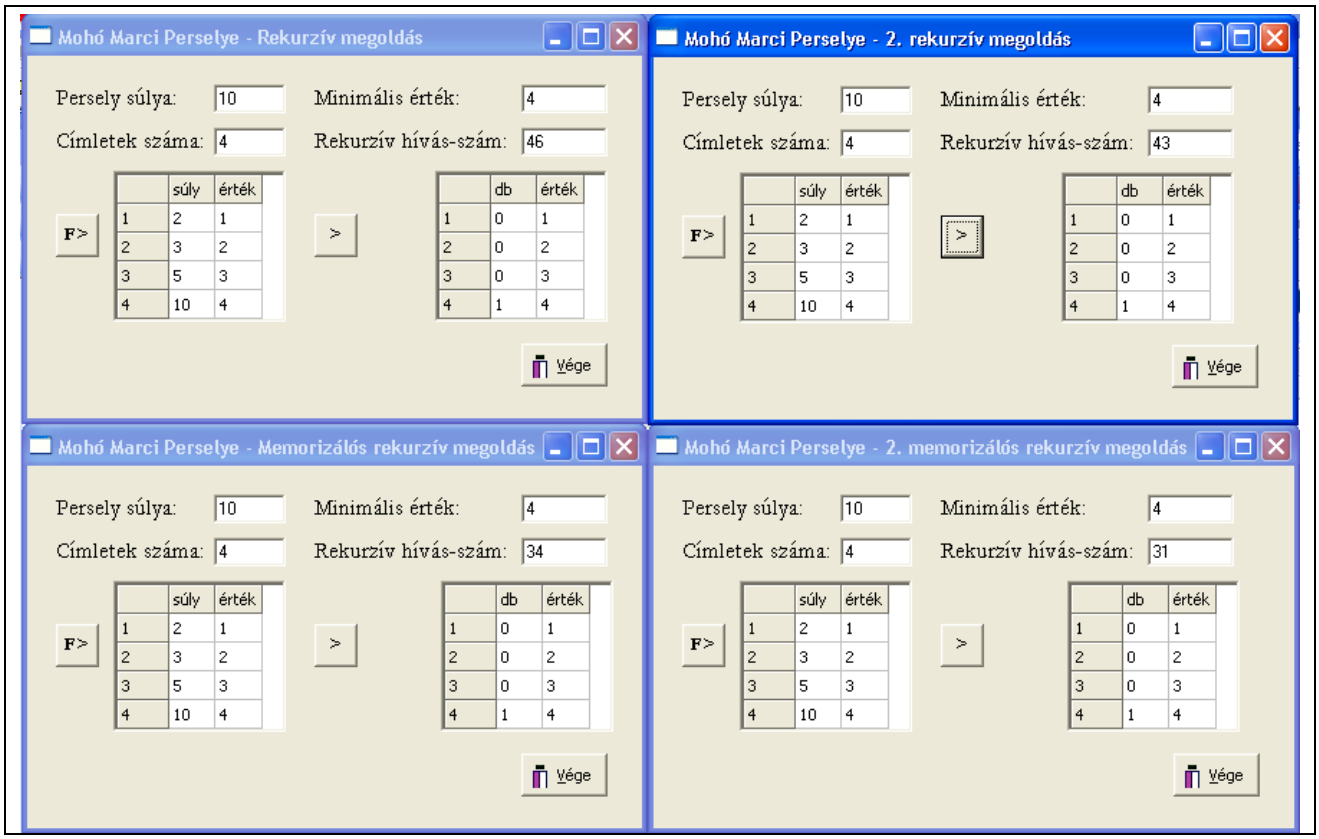
$$PMO(X) := \begin{cases} 0 & , \text{ha } X = 0 \\ Min_{i=1}^{N}\{PMO(X - s_i) + é_i \mid s_i \le X\} & , \text{egyébként} \end{cases} \qquad \text{(PMODef-2)}$$

Ennek kódmegfelelője a következő:

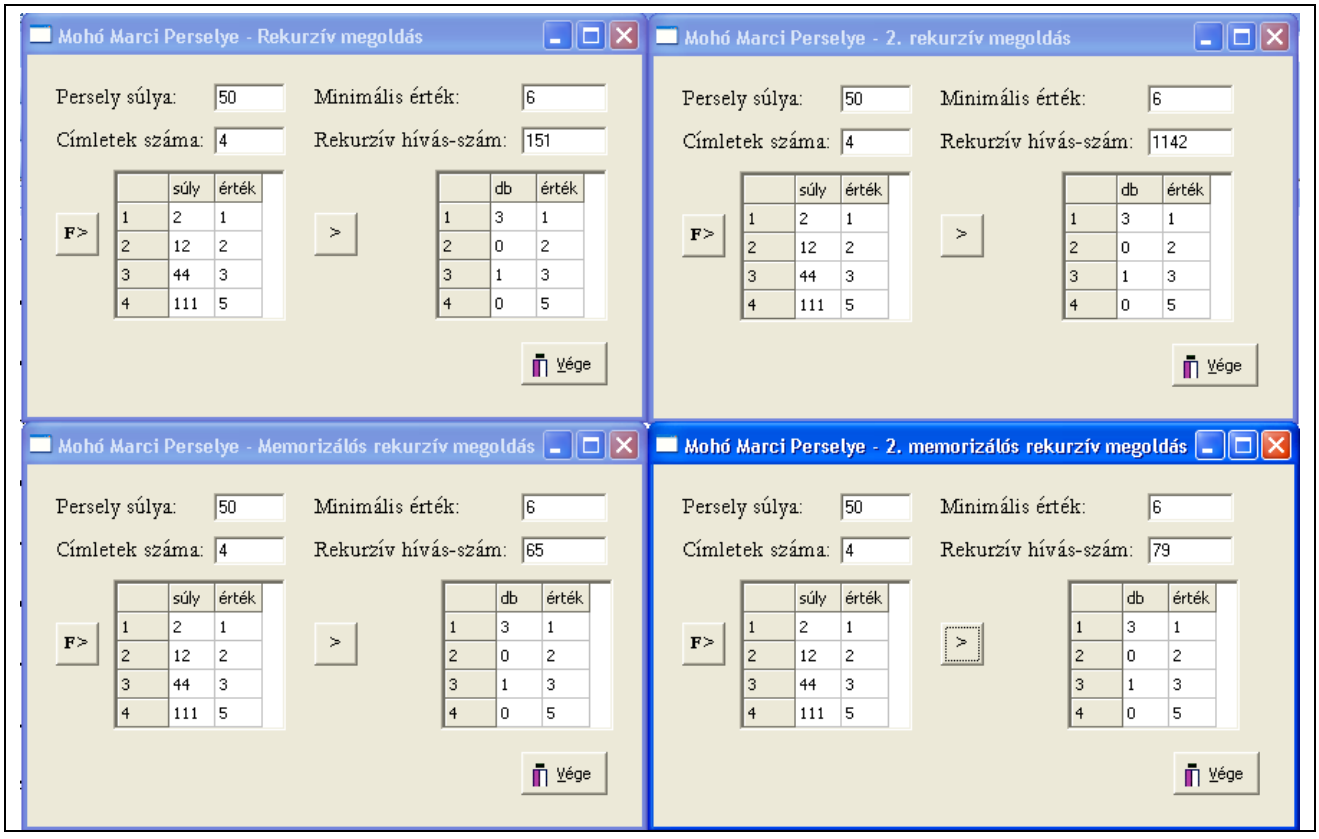
```
function PMO(x:integer):integer;//rekurzív fv. -- 2. megoldás
  var
    m,m2,i:integer;
begin
  inc(rekH);
  If
    x=0 then
    Begin
      PMO:=0
    End
      Else
    Begin
      m:=MaxErt;
      For i:=1 to Cimletek.db do
      Begin
        If Cimletek.cimlet[i].suly<=x then
        begin
          m2:=PMO(x-Cimletek.cimlet[i].suly)+Cimletek.cimlet[i].ertek;
          If m2<m then
          Begin
             m:=m2;
             //a kiválasztott címlet feljegyzése:
             if Valaszt[x].min>m then//az eddiginél jobb
             begin
               Valaszt[x].i:=i; Valaszt[x].min:=m;
             end;//if
          End;
        end;//if Cimletek
      End;
      PMO:=m
    End
  {EndIf};
end;//PMO
```

5.3.4 Egy DP megoldás

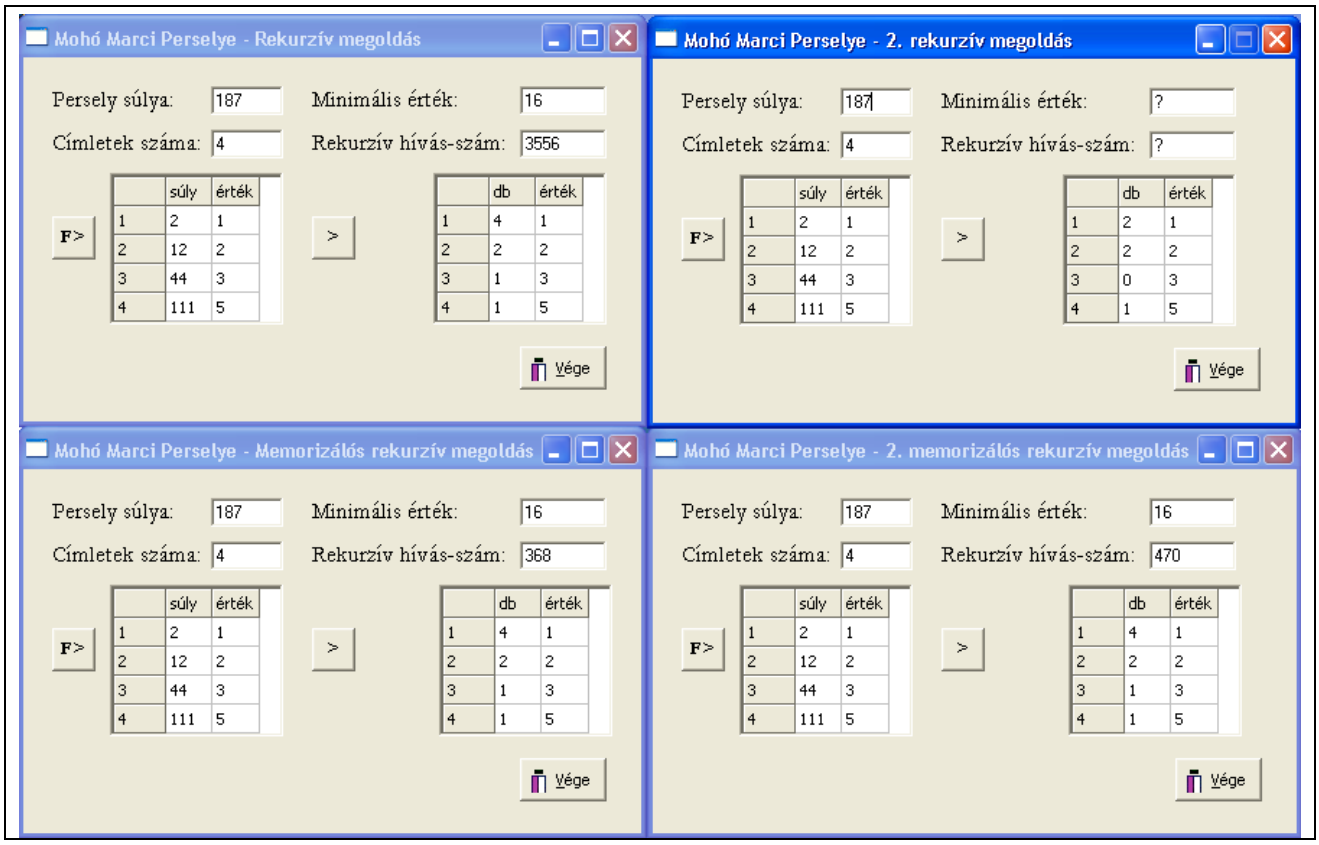
A memorizálós elvű DP változat ebből szinte magától értődik, ezért nem részletezzük. Az így kapott alkalmazások néhány futási eredményét viszont alábbi ábrasoron közöljük.



15. ábra. A perselybeli minimális pénzt meghatározó, globális címletezésen alapuló rekurzív és memorizálós alkalmazás összevetése – kis összeg esetén.



16. ábra. A perselybeli minimális pénzt meghatározó, globális címletezésen alapuló rekurzív és memorizálós alkalmazás összevetése – közepes összeg esetén.



17. ábra. A perselybeli minimális pénzt meghatározó, globális címletezésen alapuló rekurzív és memorizálós alkalmazás összevetése – „nagy” összeg esetén. A 2. rekurzív megoldás kivárhatatlanul lassú!

Érdemes elgondolkodni, hogy egyáltalán meglepő-e lesújtó eredmény! A helyért lehet, hogy ez esetben is az idővel kell fizetni?

Befejezésül letölthetjük e más elvű változatokat: a rekurzívat (zip), memorizálóst (zip).

6 Irodalom

[HGy] Horváth Gyula: „*Tehetséggondozó Program – Dinamikus programozás*”, NJSzT, 2005
[RISz] Rónyai L.,Ivanyos G.,Szabó R.: „*Algoritmusok*”, TYPOTEX, 1999
[SzP2] Szlávi Péter et al.: „*További példák a dinamikus programozáshoz*”, e-anyag
http://people.inf.elte.hu/szlavi/DP/