Kőrösi Gábor

Algoritmusok és adatszerkezetek a gyakorlatban

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

Dinamikus programozás

Összefoglaló

Az eddigiekben egy-egy összetett feladat megoldására oszd-meg-és-uralkodj megoldást használtunk. A rekurzív megoldás legnagyobb hátránya azonban az, hogy legtöbb esetben, ahogy növekszik az elemszámunk, vele együtt hatványozottan növekszik a megoldáshoz szükséges idő hossza és a lépések száma is. Ennek a problémának a kiküszöbölésére ad hatékony megoldást a dinamikus programozás. Ebben a tananyagban egy másik, szintén részproblémára bontáson alapuló, algoritmuscsaláddal, a dinamikus programozással ismerkedünk meg.

Lecke fejezetei:

- Mi is az a dinamikus programozás? – Olvasó (5 perc)
- A táblázatkitöltés módszere – Olvasó (17 perc)
- Az alulról felfelé építkezés módszere – Olvasó (18 perc)
- Mikor kell a dinamikus programozás technikáját használni? – Olvasó (5 perc)
- Gyakorló feladatok – Gyakorlati (45 perc)

Téma típusa: Gyakorlati
Olvasási és gyakorlási idő: 90 perc

Mi is az a dinamikus programozás? (5 perc)

A rekurzív algoritmusok sokszor azért lassúak, mert bizonyos részproblémákat többször is kiszámolnak. Tipikusan ez történik, amikor optimalizálási feladatunk van és a megoldáskezdeményeket lepésről lépésre építjük fel. Ezt úgy tudjuk elkerülni, ha az egyszer már kiszámolt részmegoldásokat eltároljuk, és később újra felhasználjuk azokat. A több időt valójában most több tárra „cseréljük", melynek eredményeként a rekurzív programok időkomplexitása drasztikusan csökkenthető.

A dinamikus programozás a részproblémákra bontás gyengeségeit két fő módszerrel próbálja orvosolni:

- Az egyik módszer lényege, hogy programunk futása alatt a megoldott részproblémák (optimális) megoldását eltároljuk, majd ha ugyanezen részprobléma megoldására szükségünk lenne később, csak kiolvassuk a megoldást, nem oldjuk meg új a részfeladatot. (TÁBLÁZATKITÖLTÉS módszere)
- Általában amikor a részproblémákat többször is szükséges meg oldani, akkor az összes lehetséges „kis" részproblémát ki kell számolni, hogy az eredeti nagy probléma megoldása kiszámítható legyen. Ezért a dinamikus programozásban tipikusan a legkisebb problémától indulunk, minden egyes nagyobb részprobléma megoldását kiszámítjuk a kisebb részfeladatok megoldásainak felhasználásával, egészen addig míg az eredeti/legnagyobb feladatig el nem jutunk (LENTRŐL FELFELÉ ÉPÍTKEZÉS módszere). Vegyük észre, hogy ez ellentétes az oszd-meg-és-uralkodj algoritmusok filozófiájával, ahol mindig az eredeti problémát bontjuk kisebb és kisebb problémákra (FELÖLRŐL LEFELÉ ÉPÍTKEZÉS módszere)

A táblázatkitöltés módszere (17 perc)

A részeredmények (TÁBLÁZATKITÖLTÉS) eltárolásának módszere:

- A részproblémára bontási algoritmusunkat kibővítjük egy- vagy kétdimenziós tömbbel, melyet a részeredmények tárolására használunk fel.
- Amikor egy részproblémát kell megoldanunk először megvizsgáljuk, hogy a keresett érték megtalálható-e a részeredmények között.
- Ha megtalálható, akkor felhasználjuk, egyébként kiszámoljuk, majd eltároljuk.

Nézzünk egy egyszerű példát, melyben a Fibonacci számokat számoljuk ki rekurzív módszerrel majd táblázatkitöltős módszerrel!

```
def fib(n):
    if (n==2) or (n==1):   #alapeset
        return 1
    else:                  #rekurziv eset
        return fib(n-1) + fib(n-2)
```

Ez a megoldás (rekurzió) a kívánt Fibonacci szám értékével tér vissza, ám hatásfoka nagyon rossz, hiszen minden egyes számot újra és újra kiszámol, függetlenül attól, hogy egyszer (vagy többször) már kiszámolta ezt az értéket, ahogy ezt már láthattuk az „oszd meg és uralkodj" anyagrésznél.

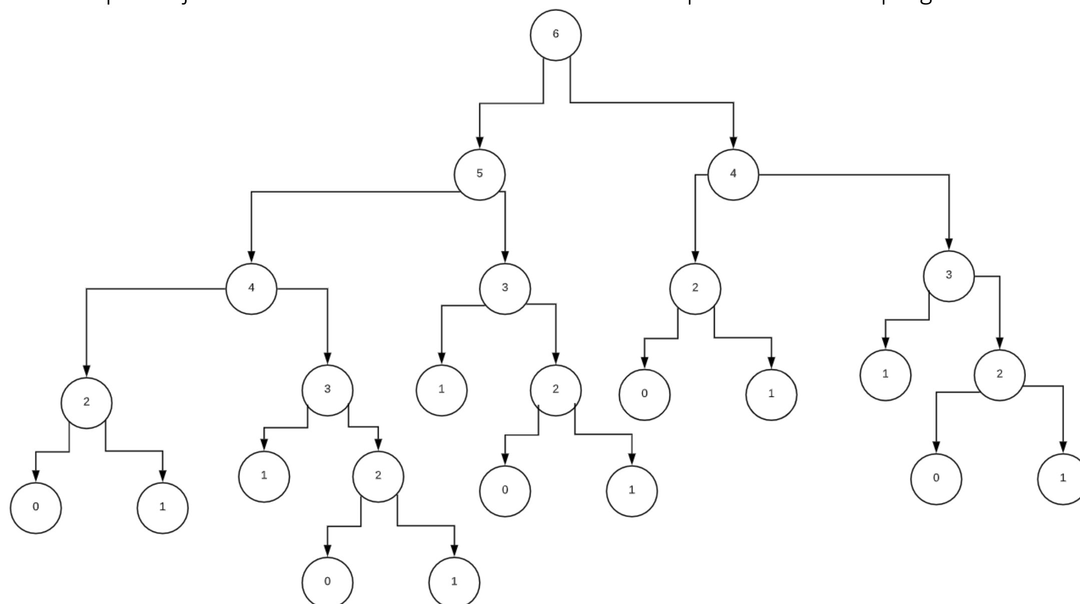
Sok számot többször ki kell számolnunk, és emiatt sok az ismételt lépés. Ez öt szám esetén nem tűnhet nagy problémának, de ahogy azt észrevehettük a megoldásnak az időkomplexitása exponenciális $O(n^2)$. A dinamikus programozás az ilyen problémák kiküszöbölésére lett megalkotva. A megoldás azt „kérdezi": ha egyszer valamit kiszámoltunk, akkor azt miért nem tároljuk el, és használjuk fel később, ha szükséges.

Módosítsuk úgy a fenti példánkat úgy, hogy algoritmusunkat kibővítjük egy MEMO nevű tömbbel, mely képes eltárolni az egyes lépések eredményét! Amennyiben egy lépés eredménye még nem ismert (==0) úgy eltároljuk azt, egyébként (!=0) használjuk fel az eltárolt eredményt. Ennek a megoldásnak köszönhetően az ismétlődő lépéseket nem kell újra és újra kiszámolnunk.

Nézzük meg, hogyan nézne ki a példánk, ha eltárolnánk az értékeket!

```
def fib(memo,n):
    if (memo[n]!= 0):
        return memo[n]
    if (n==2) or (n==1):
        memo[n] = 1
    else:
        memo[n] = fib(memo,n-1) + fib(memo,n-2)
    return memo[n]
```

A fenti példa jól szemlélteti a TÁBLÁZATKITÖLTÉSEN alapuló dinamikus programozás hatékonyságát.



Az alulról felfelé építkezés módszere (18 perc)

A TÁBLÁZATKITÖLTÉSEN alapuló dinamikus programozási megoldásunk (azoknál a feladatoknál, ahol minden részproblémát ki kell számolnunk) még gyorsabb lehet, ha nem FELÜLRŐL LEFELÉ, rekurzív módon töltjük ki a táblázatot, hanem az ALULRÓL FELFELÉ ÉPÍTKEZÉS módszerével:

- Hozzuk létre a megfelelő méretű táblázatot, amiben minden cella egy részproblémának felel meg.
- Töltsük az alapesetek megoldásait.
- „alulról felfelé” az alapesetekből kiindulva, majd a teljes megoldásig egy jó sorrendben számolja ki a részproblémák megoldásait a már korábban kiszámolt megoldások (táblázat cellák alapján)
- A végeredmény az eredeti, legnagyobb problémának megfelelő cella (általában jobb alsó sarok) szerepel

Az iménti példában láthattuk, hogyan működik a táblázatkitöltés módszere. Vizsgáljuk meg most az ALULRÓL FELFELÉ ÉPÍTKEZÉS módszerét egy példán keresztül!

Jelölje P(n) azt a számot, ahányféleképpen mehetünk fel egy n lépcsőfokból álló lépcsőn, ha egyszerre csak 1 vagy 2 lépcsőfokot léphetünk. Adjunk egy dinamikus programozási eljárást a P(n) érték kiszámítására!

P(6) = ?

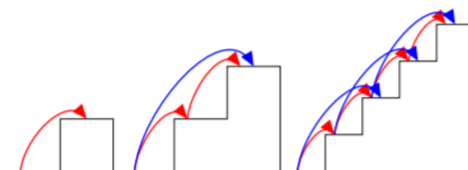
Az előzőkben a rekurzív algoritmushoz megadtuk az összefüggéseket:

Alapeset:

- P(1) = 1 (ha egy lépcső van, egyféleképpen mehetünk)
- P(2) = 2 (ha két lépcső van, vagy kétszer egyet lépünk, vagy egyszer kettőt)

Rekurzív eset:

- $P(n) = P(n-1) + P(n-2)$, ha $n \geq 3$ (utolsó lépésként egyet vagy kettőt léphetünk)



```
def lepcso(n):
    if (n==1):     #alapeset1
        return 1
    if (n==2):     #alapeset2
        return 2
    else:          #rekurziv eset
        return lepcso(n-1) + lepcso(n-2)
```

Az előző példához hasonlóan, ismét azzal szembesülünk, hogy a rekurzív módszerünk a megfelelő eredményt adja vissza, viszont ez idő alatt feleslegesen újraszámol minden eshetőséget. Lássuk, hogyan nézne ez ki ALULRÓL FELFELÉ ÉPÍTKEZÉSes TÁBLÁZATKITÖLTÉSsel!

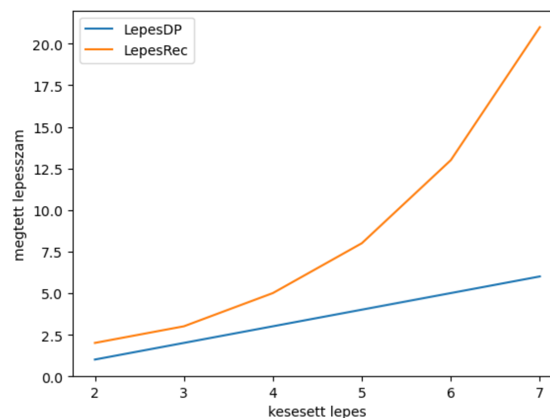
```
def lepcsoDP(n):
    T = np.array([0]*(n+1))
    T[1]=1                      #alapeset1
    T[2]=2                      #alapeset2
    for i in range (3,n+1):     #rekurziv eset
        T[i]=T[i-1]+T[i-2]
    return T[n]
```

A következő példában egy nyolc lépcsőfokból álló lépcsőre szeretnénk felmenni.

i	0	1	2	3	4	5	6	7	8
$T[i]$	–	1	2	3	5	8	13	21	34

Láthatóan sok lépést spórolunk meg az előző rekurzív megoldáshoz képest!
Vizsgáljuk meg ezt a következő mintakód segítségével.

```
def lepcsoDP(n):
    global count
    T = np.array([0]*(n+1))
    T[1]=1                          #alapeset1
    T[2]=2                          #alapeset2
    for i in range (3,n+1):         #rekurziv eset
        count = count + 1
        T[i]=T[i-1]+T[i-2]
    return count


def lepcsoRec(n):
    global count
    count = count + 1
    if (n==1):     #alapeset1
        return 1
    if (n==2):     #alapeset2
        return 2
    else:          #rekurziv eset
        return lepcsoRec(n-1) + lepcsoRec(n-2)

#keressuk meg az eredmenyeket DP-vel 2-8 kozott
#kozben szamoljuk meg a lepeseket
count = 1
lepesDP = []
for i in range(2,8):
    count = 1
    lepesDP.append(lepcsoDP(i))

#keressuk meg az eredmenyeket REC-al 2-8 kozott
#kozben szamoljuk meg a lepeseket
lepesRec = []
for i in range(2,8):
    count = 1
    lepesRec.append(lepcsoRec(i))

#rajzoljuk ki a kapott eredmenyt
fig, ax = plt.subplots()
ax.plot(range(2,8),lepesDP)
ax.plot(range(2,8),lepesRec)
ax.legend(['LepesDP','LepesRec'])
```

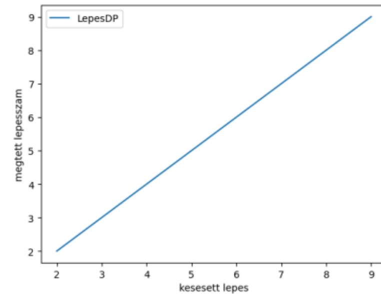


Mikor kell a dinamikus programozás technikáját használni? *(5 perc)*

Felvetődik a kérdés, hogy oszd-meg-és-uralkodj vagy dinamikus programozás technikáját érdemes-e használni egy feladat megoldása során? A dinamikus programok legnagyobb előnye, hogy minden részfeladatot csak egyszer old meg, így elsősorban olyan probléma megoldásánál kell ehhez a technikához nyúlnunk, ahol a részproblémák többször is előfordulnak. Optimalizálási problémáknál egyből gondoljunk arra, hogy kis részproblémákat kell majd többször megoldani, azaz dinamikus programozásra van szükségünk! Annak érdekében, hogy megértsük mit is jelent ez a valóságban, tekintsük meg a következő példát!

Készítsünk egy programot az n-dik Fibonacci szám kiírására! Komplexitás szemléltetése táblázatkitöltéssel:

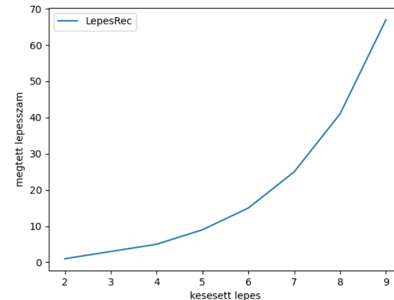
```
# n-dik Fibonacci szam meghatarozasa
# szamoljuk meg a lepeseket a count valtozo segitsegevel
count=0
Fib =[0,0,0,0,0,0,0,0,0,0]
Fib[0]=1
Fib[1]=1
for i in range(0,10):
    count = count + 1
    if i>1:
        Fib[i]=Fib[i-1]+Fib[i-2]
    else:
        Fib[i]=1

print(count)
# n=10 eseten 10 alkalommal futott le a ciklusunk
```




```
10
```

Komplexitás szemléltetése rekurzióval

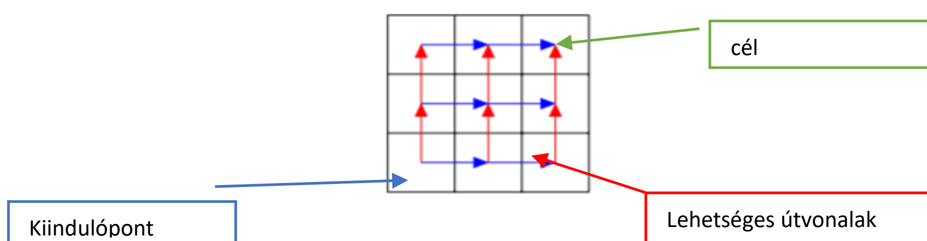
```
# n. Fibonacci szam meghatarozasa
# szamoljuk meg a lepeseket a count valtozo segitsegevel
count=0
def Fib(i):
    global count
    count = count + 1
    if i>2:
        return Fib(i-1)+Fib(i-2)
    else:
        return 1
print(Fib(10))
#n=10 eseten a fugvenyunk 109 alkalommal hivodott meg
```




```
109
```

Gyakorló feladatok (45 perc)

1. Vizsgáljuk meg, és szemeltessük egy diagrammal, hogy a Fibonacci számok kiíratására szolgáló programok rekurzív módszerrel, és a táblázatkitöltés módszerével milyen időkomplexitással futnak majd le! A megoldáshoz használjuk az első óra mintafeladatait!
2. Vizsgáljuk meg, és szemeltessük egy diagrammal, hogy a lépcsőn való lépegetés megoldására szolgáló programok rekurzív módszerrel, és a táblázatkitöltés módszerével milyen időkomplexitással futnak majd le! A megoldáshoz használjuk az első óra mintafeladatait!
3. Jelölje R(k; n) azt a számot, ahányféleképpen eljuthatunk egy k x n méretű sakktábla bal alsó sarkából a jobb felső sarkába, ha csak a jobbra vagy a felfelé szomszédos mezőre léphetünk. Határozzuk meg a dinamikus programozási eljárást az R(k; n) érték kiszámítására!
   R(3,4) = ?



Megoldás:

A következő összefüggések állnak fent:

Alapeset:

R(k, 1) = 1 (csak felfelé mehetünk)

R(1, n) = 1 (csak jobbra mehetünk)

Rekurzív eset:

R(k, n) = R(k, n−1)+R(k−1, n), ha n, k ≥ 2 (az első lépés jobbra vagy felfelé történhet)

Ezek alapján a következő dinamikus programozási eljárást adhatjuk meg:

```
def sakk(k,n):
    T = np.zeros(shape=(k,n))
    for i in range(0,n):
        T[0,i]=1
    for j in range(0,k):
        T[j,0]=1
    for i in range(1,k):
        for j in range(1,n):
            T[i,j]=T[i-1,j]+T[i,j-1]
            print(T)
    return T[k-1,n-1]
```

5	1	5	15	35
4	1	4	10	20
3	1	3	6	10
2	1	2	3	4
1	1	1	1	1
j / i	1	2	3	4

A program futtatása után keletkező táblánk így nézne ki. (5x4 táblázat esetén)

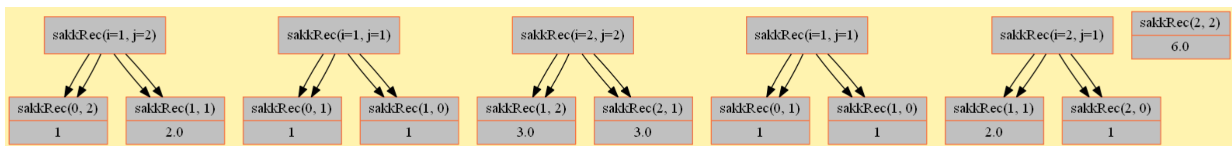
Szemléltessük a táblázatkitöltő programot lépésről lépésre dinamikus és rekurzív programmal is, úgy, hogy közben megszámoljuk a ciklus/rekurzív hívások számát is.

```
def sakk(k,n):
    global count
    T = np.zeros(shape=(k,n))
    for i in range(0,n):
        T[0,i]=1
    for j in range(0,k):
        T[j,0]=1
    for i in range(1,k):
        for j in range(1,n):
            count = count + 1
            T[i,j]=T[i-1,j]+T[i,j-1]
            print(T)
    return T[k-1,n-1]

def sakkRec(i,j):
    global countr
    countr = countr + 1
    global T
    if j == 0:
        T[i,j] = 1
        return 1
    if i == 0:
        T[i,j] = 1
        return 1
    T[i,j] = sakkRec(i-1,j) + sakkRec(i,j-1)
    print(T)
    return T[i,j]

T = np.zeros(shape=(3,3))
```

```
[[1. 1. 1.]
 [1. 2. 0.]
 [1. 0. 0.]]
[[1. 1. 1.]
 [1. 2. 3.]
 [1. 0. 0.]]
[[1. 1. 1.]
 [1. 2. 3.]
 [1. 3. 0.]]
[[1. 1. 1.]
 [1. 2. 3.]
 [1. 3. 6.]]
```

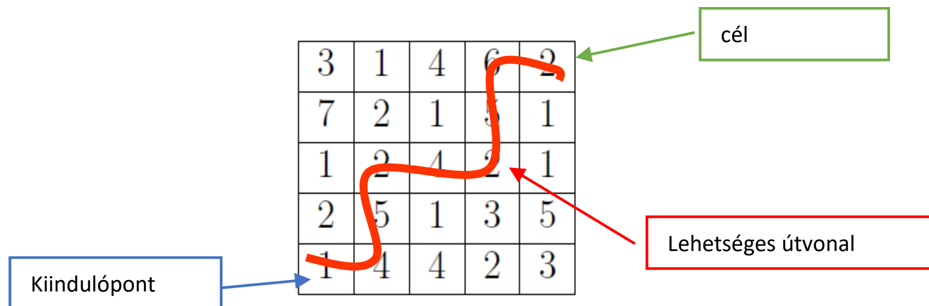
```
T[0,0]=1
countr=0
sakkRec(2,2)
count=0
sakk(3,3)
print('A rekuriziv prog lepesszama')
print(countr)
print('A dp lepesszama')
print(count)
```

```
A rekuriv prog lepesszama
11
A dp lepesszama
4
```



1.Ábra: a rekurzív hívásaink ábrája

4. Adott egy **k×n** méretű tábla. Minden mező esetén adott egy C_{ij} pozitív szám, ami a mezőről begyűjthető érték. Egy játékos a bal alsó sarokból szeretne eljutni a jobb felső sarokba úgy, hogy csak jobbra és felfelé léphet a szomszédos mezőre. Az útja során összegyűjtheti a mezőkről az értékeket (pontokat).
   Mennyi pontot tudunk összegyűjteni maximálisan?
   Hogyan határoznánk meg azt az utat, amin a maximális pontszámot tudjuk összegyűjteni?



Az előző feladatot annyiban kell módosítanunk, hogy nem a lépések számát, hanem az addig összegyűjthető maximum értéket kell, hogy eltároljuk.

```
def sakk(k,n,C):
    T = np.zeros(shape=(k,n))
    T[0,0] = C[0,0]
    for i in range(1,n):
        T[0,i]= T[0,i-1]+C[0,i]
    for j in range(1,k):
        T[j,0]= T[i-1,0]+C[i,0]
    for i in range(1,k):
        for j in range(1,n):
            T[i,j]=np.max([T[i-1,j],T[i,j-1]])+C[i,j]
    return T[k-1,n-1]

k = 2 #harom oszlop
n = 3 #negy sor
C = np.array(
```

```
    [[1,2,3],
     [4,5,6],
     [4,5,6],
     [4,5,6]
     ])

print('A megoldas')
sakk(k,n,C)
```
```
A megoldas
15.0
```

Teszteljük a programot különböző méretekre és ábrázoljuk az így kapott lépések számát.

Ajánlott kitekintő anyag (209 perc)

Pénzváltási feladat **(9 perc)** – Videó magyar nyelven

Optimalizálási feladatok egyszerű megoldásai **(5 perc)** – Videó magyar nyelven

Pénzváltási feladat rekurzív megoldása **(10 perc)** – Videó magyar nyelven

Dinamikus Programozás és Pénzváltási feladat DP megoldása **(24 perc)** – Videó magyar nyelven

Hátizsák problémák **(7 perc)** – Videó magyar nyelven

Ismétléses hátizsák feladat **(16 perc)** – Videó magyar nyelven

Ismétlés nélküli hátizsák feladat és egy optimális megoldás megszerkesztése **(32 perc)** – Videó magyar nyelven

Dinamikus Programozás helyessége **(11 perc)** – Videó magyar nyelven

Dinamikus programozás **(14 perc)** – Youtube link angol nyelven

Dinamikus programozás **(51 perc)** – MIT Video link angol nyelven

Dinamikus programozás **(30 perc)** - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein ÚJ ALGORITMUSOK link 288-294 oldal

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE