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

Oszd meg és uralkodj

Összefoglaló

A következőkben egyszerű példákon keresztül fogunk megismerkedni az oszd meg és uralkodj algoritmus működésével, valamint gyakorlati példákkal fogjuk összehasonlítani az oszd meg és uralkodj módszert az egyéb rekurzív problémamegoldó eljárásokkal.

Lecke fejezetei:

- Mi is az „oszd meg és uralkodj” módszer? – Olvasó (2 perc)
- Rekurzió – Olvasó (2 perc)
- Az „oszd meg és uralkodj” módszer működése – Olvasó (5 perc)
- Példák az „oszd meg és uralkodj” módszer működésére – Olvasó, gyakorló (20 perc)
- Gyakorló feladatok – Gyakorlati (45 perc)

Téma típusa: Gyakorlati
Olvasási és gyakorlási idő: 90 perc

Mi is az „oszd meg és uralkodj” módszer? *(2 perc)*

Bonyolult matematikai képletek helyett vizsgáljuk meg egy egyszerű példával.

3+6+2+4

3+6 2+4

9+6

15

A fent látható „általános iskolás” példában egy összetett matematikai műveletet bontottunk két egyszerűbb feladattá, majd a részösszegeket felhasználva megkapjuk a végeredményt. Amennyiben nagyon röviden szeretnénk elmagyarázni, akkor ennyi lenne az „oszd meg és uralkodj” módszer lényege. Ezt elmondani igencsak egyszerű, viszont ezt algoritmusossal (és programmal) megoldani már jóval bonyolultabb, és nagy odafigyelést igényel. Ahogyan azt láthattuk, a módszer valódi lényege az, hogy egy bonyolult összefüggést kis, kezelhetőbb darabokra vágjunk fel.

	3+6+2+4
Oszd meg	3+6 2+4
Uralkodj	(3+6) + (2+4)
Egyesítsd	9+6=15

Rekurzió *(8 perc)*

Mielőtt nagyon belemelegednénk a „oszd meg és uralkodj” módszerbe, szükségesnek érzem, hogy néhány szót ejtsünk a rekurzióról. Rekurzióról akkor beszélünk, ha egy függvény önmagát hívja meg, vagy függvények kölcsönösen egymást hívják meg.

```
def rek(a):
        if a<0:
           return
        a=a+1
        rek(a)
```

A példa jól szemelteti a rekurzív programhívást, és egyben bemutatja annak veszélyeit is, ugyanis egy rekurzív programnak két igen fontos elemmel kell rendelkeznie: elsősorban önmagát kell valamilyen formában meghívnia (rekurzív eset), és mindemellett a végtelen futás kivédésére rendelkeznie kell egy kilépési feltétellel (alapesettel).
Hogy ezt jobban megértsük, vegyük szemügyre a rekurzió egyik legjobb példáját, az orosz vidékeken jól ismert Matrjoska babák (1. kép) egymásba pakolásának módszerét: amikor kinyitunk egy (a legnagyobb) babát akkor egy kisebb babát találunk benne (rekurzív eset). Majd ismét kinyitunk egy babát és ismét egy kisebbet találunk. A baba nyitogatás közben rekurzívan ismételjük a feladatot, azzal, hogy minden egyes nyitás előtt leellenőrizzük, hogy a következő baba nyitható-e még. Ha elértük az utolsó babát (alapeset), akkor abbahagyjuk a műveletet.

1. Kép - Matrjoska babák (forrás: cinnamon4x.cafeblog.hu)

Mutassuk be ezt egy program segítségével! A Matrjoska bábukat egy többdimenziós tömbbel szemléltetjük.

```
a=np.array([[[[[1]]]]])
print('Matryoshka babuk szama:' + str(len(a.shape)))
print('Rekurziv nyitogatas')
def rek(a):
    if len(a.shape) == 0:
        return
    print('nyit')
    print(a)
    rek(a[0])
rek(a)
```

```
Matryoshka babuk szama:5
Rekurziv nyitogatas
nyit
[[[[[1]]]]]
nyit
[[[[1]]]]
nyit
[[[1]]]
nyit
[[1]]
nyit
[1]
```

A rekurzív függvényünk addig fog futni, amíg van mit kinyitni (rekurzív eset), azonban, ha már elérte az utolsó bábút (alapeset), akkor visszakerül a kiindulási állapotba.
A matematikában is találkozunk rekurzív képletekkel. Egy ilyen matematikai rekurzív példa a Fibonacci számok kiszámolása, melyet a következő képpen tudunk implementálni.

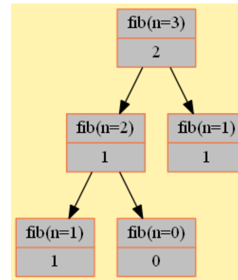
$$F(n) = \begin{cases} n, \text{If n} = 0 \text{ or } 1 \\ F(n-1) + F(n-2), \text{ if n} > 1 \end{cases}$$

```
def F(n):
        if n == 0 or n == 1:              #alapeset
                return n
        else:                             #rekurziv eset
                return F(n-1)+F(n-2))
```

Rekurzió animálása

A rekurzió megértése sokszor a gyakorlott programozóknak is fejtörést okoz. Éppen ezért, szinte minden programozási nyelv fejlesztő felülete kínál valamilyen lehetőséget a rekurzió vizualizálására. Pythonban az egyik ilyen a **recursion-visualiser** csomag (!pip install recursion-visualiser), mely png-vel és gif-fel el segíti a megértés folyamtát. Nézzük meg ennek használatát egy példán keresztül!

```
from visualiser.visualiser import Visualiser as vs
@vs(node_properties_kwargs={"shape":"record", "color":"#f57542", "style":"filled", "fillcolor":"grey"})

def fib(n):
    if n <= 1:
        return n
    return fib(n=n - 1) + fib(n=n - 2)

def main():
    # hivjuk meg a fuggvenyt
    print(fib(n=3))
    # mentsuk le a kapott eredmenyt
    vs.make_animation("fibonacci.gif", delay=2)

main()
```



Az „oszd meg és uralkodj" módszer működése (5 perc)

Bár a módszer neve azt sugallja, hogy két lépésből (megoszt, uralkodik) kell megoldanunk egy-egy problémát, viszont már az első példában is feltűnhetett, hogy egy harmadik dologra, az eredmények kiszámítására, összevonására is szükségünk van.

Felosztja a feladatot több részfeladatra.
Uralkodik a részfeladatokon, rekurzív módon történő megoldásukkal. Ha a részfeladatok mérete elég kicsi, akkor közvetlenül megoldja a részfeladatokat.
Összevonja a részfeladatok megoldásait az eredeti feladat megoldásává.

Annak érdekében, hogy érthetőbbé váljon az elgondolás, nézzünk meg egy másik igen egyszerű matematikai példát, a faktoriális számítását. A faktoriális számítása elve rekurzív, hiszen pl. 4! az valójában 4*3!, és 3! pedig 3*2!.... stb. Tehát akaratunkon kívül is rekurziót használtunk, és felosztottuk a nagy problémát kisebb részfeladatokra (*return n * recur_factorial(n-1).),* amíg elértünk az 1!-ig, ami elég kicsi részfeladat a megoldáshoz (*return n*). Ezek után a kapott értékünket visszafelé görgetve (*return n * recur_factorial(n-1)*) megkapjuk az eredményt.

```
n = 6
# rekurzioval
def recur_factorial(n):
   if n == 1:
```

```
        return n
    else:
        return n * recur_factorial(n-1)

print(recur_factorial(n))
```

Példák az „oszd meg és uralkodj” módszer működésére (30 perc)

Bináris keresés *(8 perc)*

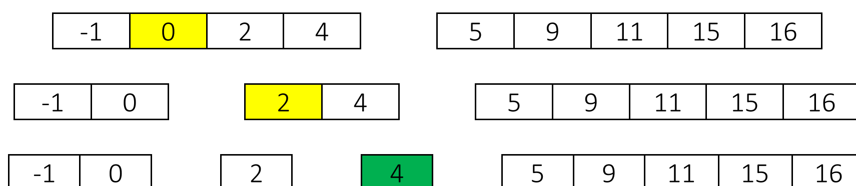
A bináris keresés egy módszer, amellyel egy rendezett tömbben tudunk megkeresni egy elemet. Nézzünk erre egy példát! Keressük meg a 4-es szám helyét az alábbi tömbben!

-1	0	2	4	5	9	11	15	16

Amennyiben nem ismerjük ezt a módszett, úgy számtalan más megoldással is kereshetnénk egy adott értéket a tömbben. Például megtehetnénk ezt ciklus segítségével, ám ennek futási ideje legjobb esetben O(1) (első helyen van a keresett elem) vagy legrosszabb esetben O(n) (utolsó helyen van a keresett elem). Épp ezért, valami hatékonyabb megoldást kell alkalmaznunk. Ebben nyújt segítséget az „oszd meg és uralkodj” módszeren alapuló bináris keresés.
A bináris keresés kihasználja azt, hogy a tömbünk eleve rendezett, és a tömb eleje helyett a közepén próbálja megkeresni az elemet.

-1	0	2	4	5	9	11	15	16

Amennyiben nem találja meg a keresett számot, úgy megnézi, hogy a középső elemhez képest, hol található a keresett elem, és ezek után, vagy a jobb- vagy a baloldali tömbben folytatódik a keresés. Az új résztömbön ismételten elvégezzük az iménti lépést. Ezt a műveletet addig ismételjük, amíg meg nem találjuk a keresett elem helyét.



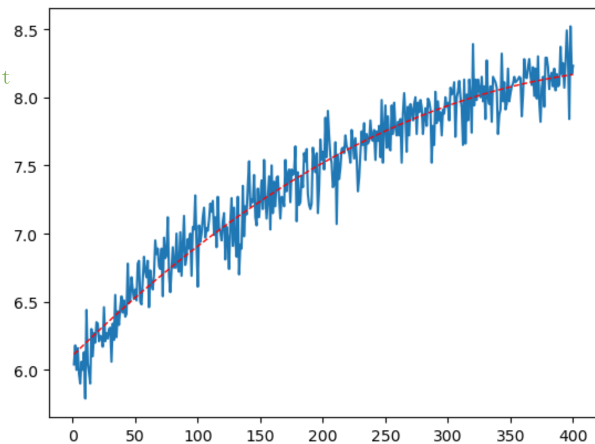
A módszer mindamellett, hogy elegáns, képes O(log n) időben megtalálni a keresett elemünket egy n elemű tömbben.
Nézzük, hogyan festene ez kóddal:

```
def binary_search(arr, ertek):
    n = len(arr)
    bal = 0
    jobb = n - 1
    # ellenorizzuk az alap esetet
    while bal <= jobb:
        kozep = (bal + jobb) // 2
        # ha a keresett elem kisebb mint e kozepso elem
        # akkor csak a baloldali resztomben lehet
        if ertek < arr[kozep]:
            jobb = kozep - 1
        # kulonban csak a jobb oldali resztomben lehet
        elif ertek > arr[kozep]:
            bal = kozep + 1
        else:
```

```
            return kozep
    # nincs ilyen elem a tomben
    raise ValueError('az ertek nincs a tomben')

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(binary_search(arr, 8))
```

Vizsgáljuk meg a program időkomplexitását (átlagos esetre):

```
def binary_search(arr, ertek):
    n = len(arr)
    bal = 0
    jobb = n - 1
    # szimulaljuk az ido mullasat egy szamlaloval
    count = 0
    # ellenorizzuk az alap esetet
    while bal <= jobb:
        count = count +1
        kozep = (bal + jobb) // 2
        # ha a keresett elem kisebb mint e kozepso elem
        # akkor csak a baloldali resztomben lehet
        if ertek < arr[kozep]:
            jobb = kozep - 1
        # kulonban csak a jobb oldali resztomben lehet
        elif ertek > arr[kozep]:
            bal = kozep + 1
        else:
            return count
    # nincs ilyen elem a tomben
    raise ValueError('az ertek nincs a tomben')


def demo():

    ido = []
    # keszitsunk 100-500 hosszu veleten szambol allo
    # tomboket es keressunk benne meg egy veletlen elemet
    for n in range(100,500):
        temp_ido = []
        # teszteljuk ezt a meretet 100x
        for i in range(0,100):
            # keszitsunk egy veletlen tombot(listat)
            a = np.random.randint(0,n,size=(n+1))
            a = list(np.sort(a))
            # veletleszeruen valsszunk ki egy elemet a listabol
            keresett = np.random.randint(0,n)
            keresett = a[keresett]
            # nezzuk meg, hogy hany lepesbol futott le
            count = binary_search(a,keresett)
            temp_ido.append(count)
        # vegyuk 100 futas atlagos idejet
        ido.append(np.mean(temp_ido))
    # rajzoljuk ki a kapott idogerbet
    plt.plot(range(1,len(ido)+1),ido)
    # keszitsunk egy trend vonalat es rajzoljuk ki
    z = np.polyfit(range(1,len(ido)+1),ido, 2)
    y_hat = np.poly1d(z)(range(1,len(ido)+1))
    plt.plot(range(1,len(ido)+1), y_hat, "r--", lw=1)

demo()
```

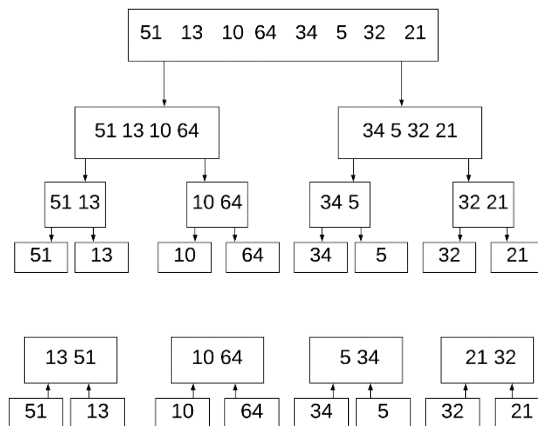


Merge Sort - összefésülő rendezés (12 perc)

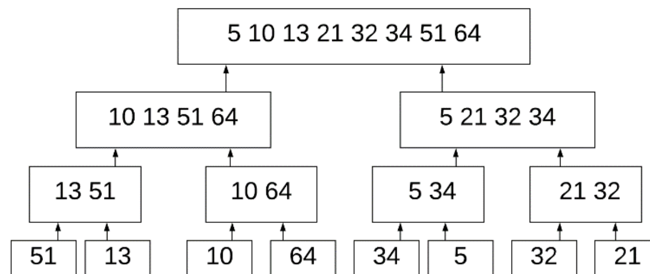
Az összefésülő rendezés lényegében egy olyan algoritmus, mely előbb darabokra szedi a rendezni kívánt tömbünket, majd az így kapott eredményt rendezett résztömbök segítségével fésüli össze. Ehhez a következő lépések szükségesek:

- Felezzük meg az n elemű tömbünket n alkalommal
- Rekurzívan rendezzük a „felezett részhalmazainkat”
- Vonjuk össze a két rendezett részhalmazt

Nézzünk erre egy egyszerű példát! Az első ábrán latható, hogy a tömbünket rekurzióval n részhalmazzá daraboltuk, majd ezek után sorrendbe raktuk a kis részhalmazpárokat. Az így kapott résztömbökből rekurzióval új részhalmazokat vontunk össze.



A következő lépésben az újonnan kialakított résztömbünket az előzőkben ismert módszerrel ismét rendeztük, majd összevontunk.



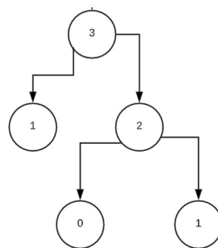
Addig ismételtük ezt a módszert, amíg végül ismét egy, de most már rendezett tömbünk marad.

```
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2 # Keressuk meg a tomb kozepet
        L = arr[:mid] # Felezzuk meg a tomb elemeit
        R = arr[mid:] # ket resztombre

        mergeSort(L) # rendezzuk az eslo felet
        mergeSort(R) # rendezzuk a masik felet

        i = j = k = 0
        # Masoljuk az elemeket egy ideiglenes L[] es R[] tombbe
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+= 1
            else:
                arr[k] = R[j]
                j+= 1
```

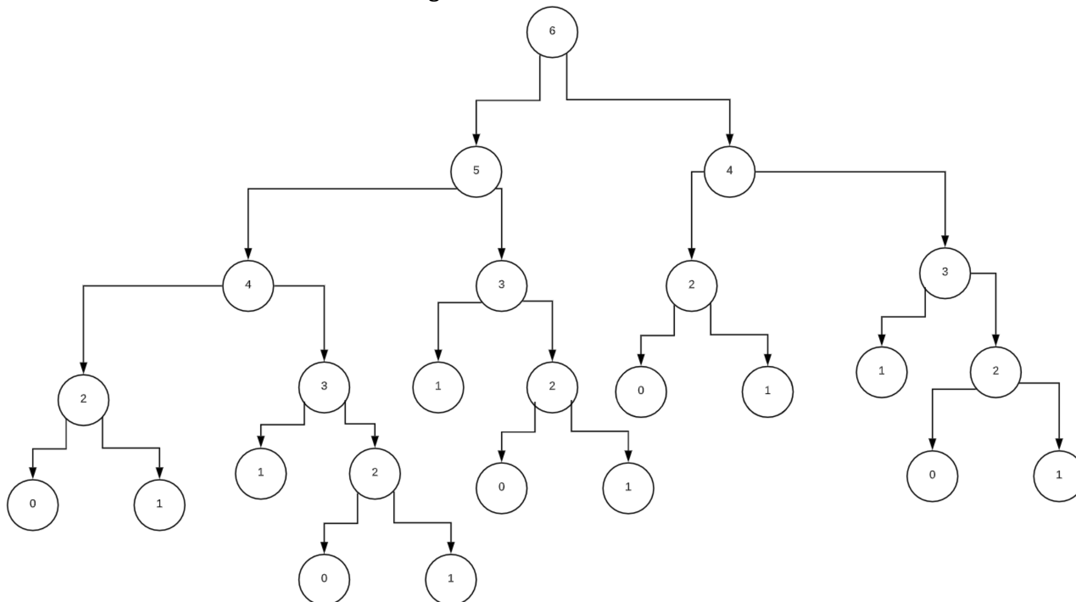
```
            k+= 1

        # viszgaljuk meg, hogy maradt-e meg elem a tombben
        while i < len(L):
            arr[k] = L[i]
            i+= 1
            k+= 1

        while j < len(R):
            arr[k] = R[j]
            j+= 1
            k+= 1

# Forras: https://www.geeksforgeeks.org/merge-sort/
```

Fibonacci számok (10 perc)

Fibonacci számokkal bárhol találkozhatunk a természetben. A nyulak szaporodása, a napraforgó magjainak rendeződése, és az írisz virágzása is Fibonacci számszerűséget mutat. A Fibonacci számsorozatban az első két szám a 0 és az 1, az összes többi szám pedig az előző két szám összege. (0,1,1,2,3,5,8,13,21,.....)
Magából a megfogalmazásból is jól látható, hogy ezen probléma megoldására is célszerű az „oszd meg és uralkodj" módszert alkalmazni. Amennyiben a harmadik Fibonacci számra vagyunk kíváncsiak, felosztjuk két kisebb egységre, vagyis megnézzük mennyi a két előző Fibonacci szám értéke (1. és 2.), majd, ha szükséges, akkor ezt is tovább bontjuk, és megnézzük, hogy mennyi a második elemet megelőző két szám értéke.



Ugyanezt a logikát követve, akármelyik Fibonacci számot ki tudjuk számítani. Nézzünk meg a példát a hatodik Fibonacci szám értékének meghatározására!



Vegyük most ezt szemügyre kód segítségével is!

```
def f(n):
        if n == 0 or n == 1:
        return n
        else:
        f(n-1) + f(n-2)
```

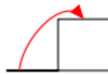
Mikor kell az „oszd meg és uralkodj” módszert használni?

A legegyszerűbb megfogalmazás szerint, akkor kell ezt a technikát alkalmaznunk, ha a problémánk kísértetiesen hasonlít az Összefésüléses rendezés problémájára. Amennyiben listával vagy tömbbel dolgozunk, és az ismételt felosztással könnyebb megoldáshoz jutunk, úgy érdemes ebben a módszerben gondolkodnunk.

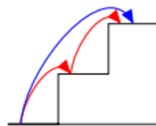
Gyakorló feladatok *(45 perc)*

1. Készítsünk rekurzív algoritmust, ami meghatározza, hogy hányféleképpen mehetünk fel egy n lépcsőfokból álló lépcsőn, ha egyszerre csak 1 vagy 2 lépcsőfokot léphetünk!

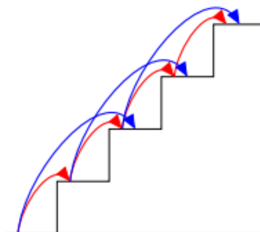
   Jelölje P(n) azt, hogy n lépcsőfokon hányféleképpen mehetünk fel.
   Ekkor a következő összefüggések állnak fent:
   P(1) = 1 (ha egy lépcső van, egyféleképpen mehetünk)



   P(2) = 2 (ha két lépcső van, vagy kétszer egyet lépünk, vagy egyszer kettőt)



   P(n) = P(n − 1) + P(n − 2), ha n ≥ 3 (utolsó lépésként egyet vagy kettőt léphetünk)



```
def P( n ) :
        if ( n == 1 ):
                return 1
        elseif ( n == 2 ):
                return 2
        else:
                return P( n-1) + P( n-2)
```

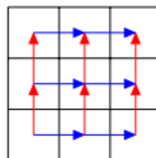
2. Készítsünk rekurzív algoritmust, amely meghatározza, hogy hányféleképpen juthatunk el egy k sorból és n oszlopból álló sakktábla bal alsó sarkából a jobb felső sarkába, ha csak a jobbra vagy a felfelé szomszédos mezőre léphetünk!

Jelölje R(n, k) azt a számot, ahány módon eljuthatunk a bal alsó sarokból a jobb felső sarokba.

R(1, k) = 1 (csak felfelé mehetünk)

R(n, 1) = 1 (csak jobbra mehetünk)

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



```
def R(  n ,   k ):
        if ( ( n == 1 ) || ( k == 1 ) ):
                return 1
        else
        return R( n−1, k ) + R( n , k−1)
```

3. Módosítsuk úgy a bináris keresőalgoritmust, hogy vizualizálja a függvények részeredményeit.

```
import numpy as np
from visualiser.visualiser import Visualiser as vs
@vs()
def binary_search(arr, bal, jobb, x):
        if jobb >= bal: # alapeset
             kozep = (jobb + bal) // 2
             if arr[kozep] == x: # ha a keresett elem a kozepso elem
                   return kozep
                   # ha keresett elem kisebb, mint a kozepso elem
                   # lepjunk a bal reszhalmazra
             elif arr[kozep] > x:
                   return binary_search(arr, bal, kozep - 1, x)
                   # kulonben csak a jobboldali resztombben lehet
             else:
                   return binary_search(arr, kozep + 1, jobb, x)
         else:
                   # az elem nem talalhato meg a tombben
             return -1
def main():
   arr=np.array([1,4,5,6,9,13,15,76])
   bal = 0
   jobb = len(arr)
   x=6
   print(binary_search(arr, bal, jobb, x))
   # mentsuk le az eredmenyt
   vs.make_animation("binkeres.gif", delay=2)
main()
```

4. Vizualizáljuk a kódot a http://pythontutor.com/visualize.html segítségével.

Ajánlott kitekintő anyag *(185 perc)*

Oszd meg és uralkodj **(12 perc)** – Videó magyar nyelven

A rekurzió fogalma(i) **(14 perc)** – Videó magyar nyelven

A rendezési feladat és a beszúró rendezés **(6 perc)** – Videó magyar nyelven

Az összefésülő rendezés **(12 perc)** – Videó magyar nyelven

Az összefésülő rendezés helyessége **(9 perc)** – Videó magyar nyelven

Rekurzív futásidők feloldása **(18 perc)** – Videó magyar nyelven

Oszd meg és uralkodj **(7 perc)** – YouTube link angol nyelven

Oszd meg és uralkodj **(22 perc)** – YouTube link angol nyelven

Oszd meg és uralkodj **(65 perc)** – MIT video link angol nyelven

Algoritmusok és adatszerkezetek gyakorlat - Oszd meg és uralkodj **(20 perc)** - Gelle Kitti - link

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE