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

SZÉCHENYI 2020
Magyarország Kormánya
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

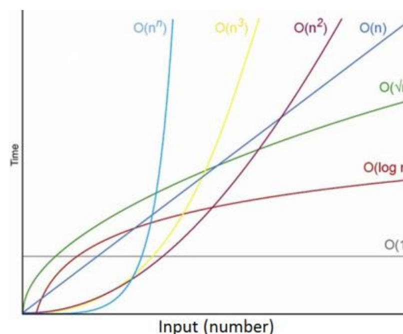
Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Algoritmusok futásidő elemzése

Összefoglaló

Egy számítógépes programnak vagy algoritmusnak sok tulajdonságát vizsgálhatjuk gyakorlati szempontból, mint például a megbízhatóságát, karbantarthatóságát, vagy azt, hogy mennyire felhasználóbarát. Azonban adja magát a kérdés, hogy hogyan zajlik egy algoritmus vizsgálata?

Képzeljük el, hogy a programozó megír egy feladatot, majd teszteli azt. Először kis értékekre, majd nagyobbra, és végül még nagyobbra. Ezek után, a kopott eredmények alapján módosít a programon, és megismétli az első lépést. Ezeket a lépéseket egészen addig ismétli, amíg pontos képet nem kap a program futásának eredményéről. Ez az elgondolás azonban helytelen, hiszen nincs szükségünk pontos futási értékekre, csak az idő- vagy memóriaigény növekedésének a megbecslésére – esetleg komplexitás meghatározására. A lépésszám pontos meghatározása helyett általában elegendő a lépésszám nagyságrendjének meghatározása, és ebből már (kis óvatossággal) következtetni lehet arra, hogy az algoritmus mennyire hatékony, avagy hogyan fog viselkedni nagyobb értékekre.



Gyakorlatban pedig egy programnak a következő komplexitása lehet:

$$1<\log n<\sqrt{n}<n<n\log n<n^2<n^3<\ldots\ldots<2^n<3^n<\ldots\ldots<n^n$$

Ebben a tananyagrészben, a gyakorlati feladatokon keresztül, megismerkedünk az egyes algoritmusok időkomplexitásával, valamint megvizsgáljuk a konstans, logaritmikus, négyzetes és egyéb algoritmusok működésének időigényét.

Lecke fejezetei:

- Konstans időkomplexitás – Olvasó (10 perc)
- Lineáris időkomplexitás – Olvasó (10 perc)
- Négyzetes időkomplexitás – Olvasó (10 perc)
- Összetett időkomplexitás – Olvasó (10 perc)
- Gyakorló feladatok – Gyakorlati (50 perc)

Téma típusa: Gyakorlati
Olvasási és gyakorlási idő: 90 perc

Konstans időkomplexitás *(10 perc)*

Absztrakt fogalmak helyett vizsgáljuk meg egy egyszerű példával, hogy mit is jelentenek az alábbi fogalmak: komplex, állandó komplexitás. Adott egy algoritmusunk, amely egy N elemű tömb két tetszőleges elemének összegével tér vissza. Elemezzük ki a kapott kód működését:

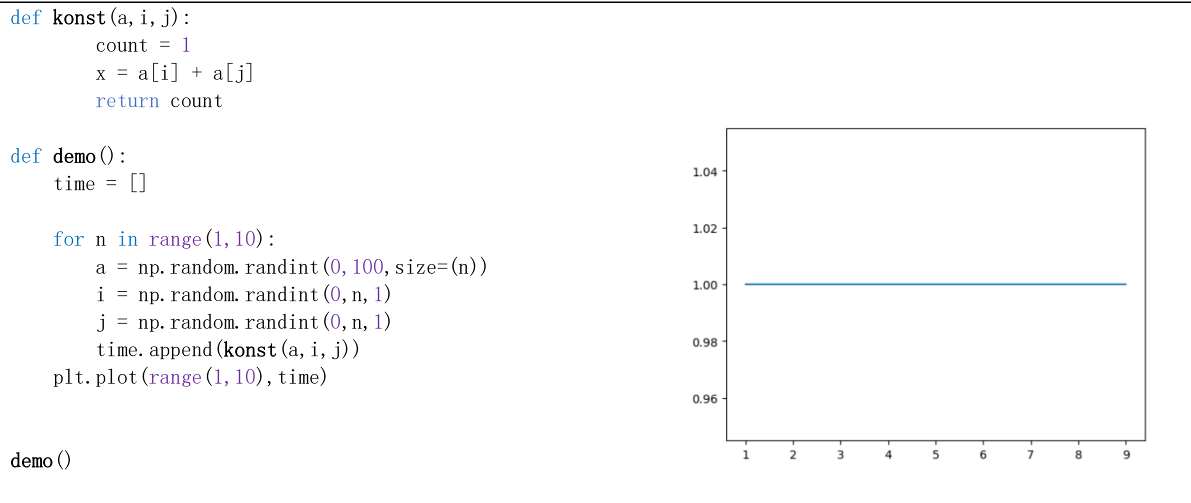
	költség	végrehajtási szám
def demo(a ,i, j):		
x = a[i] + a[j]	C1	1
return x	C2	1

A példában a C1 ideje a tömb méretétől függetlenül mindössze 1 műveletet vesz igénybe, és ez igaz a második C2 műveltünkre is. Jól látható tehát, hogy a programunk futási időigénye KONSTANS, melyet

gyakorlatilag semmilyen külső változó nem befolyásol, még a tömb n mérete sem. Tehát az algoritmusunk futási ideje valójában C1+C2 avagy *O*(1) + *O*(1) = 2* *O*(1), mely képletből számunkra csak az *O*(1) tag számottevő.

Fontos, hogy a futásidőnek az *O*(1)-el történő jelölése nem azt jelenti, hogy csupán egy lépésből áll az algoritmus, hanem azt, hogy egy futtatás megközelítőleg mindig ugyanannyi időt vesz igénybe.

Igazoljuk ezt az állításunkat egy program segítségével! Az idő múlását a COUNT változó értéke szemlélteti.



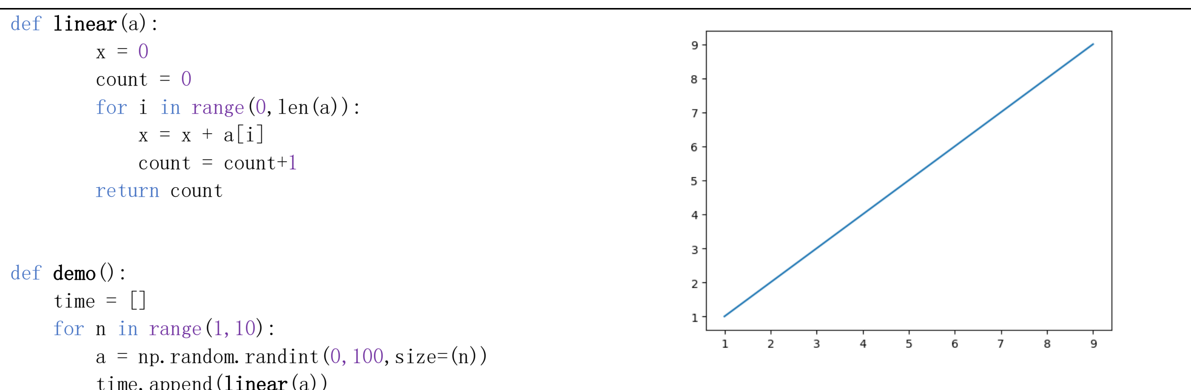
```
def konst(a, i, j):
        count = 1
        x = a[i] + a[j]
        return count

def demo():
    time = []

    for n in range(1, 10):
        a = np.random.randint(0, 100, size=(n))
        i = np.random.randint(0, n, 1)
        j = np.random.randint(0, n, 1)
        time.append(konst(a, i, j))
    plt.plot(range(1, 10), time)


demo()
```


Lineáris időkomplexitás(10 perc)

Komplex algoritmusok idő- vagy memóriaigénye nagyon ritkán KONSTANS. Egy-egy ilyen program gyakori alkotó eleme a ciklus, amely LINEÁRIS komplexitású. Nézzünk most egy algoritmust, melyben egy N méretű tömb elemeinek értékét kívánjuk összeadni.

	költség	végrehajtási szám
def demo(a):		
for i in range(0,len(a))	C1	N
x = x + a[i]	C2	N
return x	C3	1

Az algoritmusunkban a C1 ciklusunk nem kevesebbszer, mint n + 1-szer fut le, míg a maradék két sorunk időigénye továbbra is konstans marad. Az így létrehozott algoritmus (n+1) * (C1 + C2) +1 ideig fut, azonban számunkra elegendő tudni azt, hogy melyik az a paraméter, amely a leggyorsabban növekszik, ez pedig az n, tehát a futási időnk *O*(n).
Igazoljuk ezt az állításunkat egy program segítségével! Az idő múlását a COUNT változó értéke szemlélteti.



```
def linear(a):
        x = 0
        count = 0
        for i in range(0, len(a)):
            x = x + a[i]
            count = count+1
        return count


def demo():
    time = []
    for n in range(1, 10):
        a = np.random.randint(0, 100, size=(n))
        time.append(linear(a))
```

```
    plt.plot(range(1,10), time)

demo()
```

Négyzetes időkomplexitás *(10 perc)*

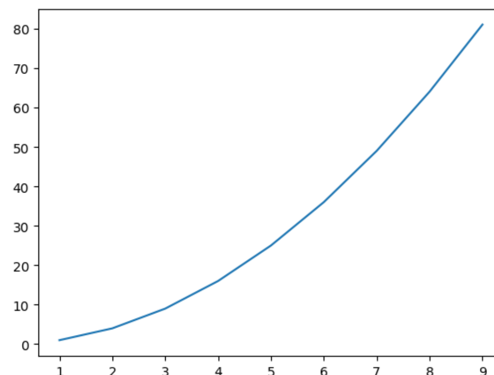
Egy algoritmus megalkotása közben szükségszerű, hogy a meglévő függvényeinket, algoritmusainkat egymásba ágyazzuk, amely gyakran négyzetes viselkedést eredményez. Nézzünk meg egy példát, ahol az előzőekben használt algoritmust egy újabb ciklusba ágyazzuk!

	költség	végrehajtási szám
def demo(a):		
for i in range(0,len(a))	C1	N
for j in range (0,len(a))	C2	N
x = a[i] + a[j]	C3	N*N
return x	C4	1

Az algoritmusunkban a C1 ciklusunk, ahogy az előző példánkban, most is n+1-szer fut le. A belső ciklusunk is n+1 szer fut le, valamint a két utolsó sorunk időigénye pedig továbbra is konstans marad. Tudjuk, hogy az előzőleg használt algoritmusunk időigénye O(n), így nem nehéz kitalálni, hogy ez a program (n+1) * O(n), azaz $O(n^2)$ komplexitású.

Igazoljuk ezt az állításunkat egy program segítségével! Az idő múlását a COUNT változó értéke szemlélteti.

```
def negyz(a):
        x = 0
        count = 0
        for i in range(0,len(a)):
            for j in range(0,len(a)):
                x = a[i] + a[j]
                count = count+1
        return count

def demo():
    time = []
    for n in range(1,10):
        a = np.random.randint(0,100,size=(n))
        time.append(negyz(a))
    plt.plot(range(1,10),time)

demo()
```



Logaritmikus időkomplexitás *(10 perc)*

Az előző példákban láthattuk, hogy az ideális esetben konstans, rosszabb esetben lineáris tulajdonság még elfogatható, azonban az exponenciális komplexitású algoritmusok nagy elemszám esetén szinte megoldhatatlanul nehézzé teszik a feladtok végrehajtását. Épp ezért számtalan megoldás született a feladatok optimalizálására. Ilyenek a különböző rendezési és elemtárolási algoritmusok és adatszerkezetek, melyek a célja, hogy nagy elemszám esetén $O(n^2)$ helyett O(log N) időkomplexitású megoldást kapjunk. A tananyag következő fejezeteiben ilyen megoldásokkal fogunk majd megismerkedni. Ahhoz, hogy megértsük, mit is jelent O(log N) időben megoldani egy feladatot, vegyünk ismét egy N elemű vektort, melynek elemeit a következő algoritmussal adjuk össze:

	költség	végrehajtási szám
def demo(a):		
i = len(a)	C1	1
while(i>0)	C2	LogN

x = a[i] + a[i]	C3	LogN
i = int(i/2)	C4	LogN
return x	C5	1

A C1 parancsunk konstans időben fut le, míg a C2 ciklusunk előbb a tömb feléig, majd annak feléig... fut, mindaddig, amíg felezni tudja a tömbünk méretét. Ez az idő O(log n) komplexitású, melyből kiszámítható, hogy az algoritmusunk 1+ log n * (1+1) + 1 ideig fut, melyből számunkra csak a leggyorsabban növekvő paraméter érdekes, így a kapott eredmény O(log n).

Igazoljuk ezt az állításunkat egy program segítségével! Az idő múlását a COUNT változó értéke szemlélteti.

```
def logN(a):
        count = 0
        x = 0
        i = len(a)-1
        while (i>0):
            count = count +1
            x = x + a[i]
            i= int(i/2)
        return count

def demo():
    time = []
    for n in range(1,1000):
        a = np.random.randint(0,100,size=(n))
        time.append(logN(a))
    plt.plot(range(1,1000),time)

demo()
```


Összetett időkomplexitás (10 perc)

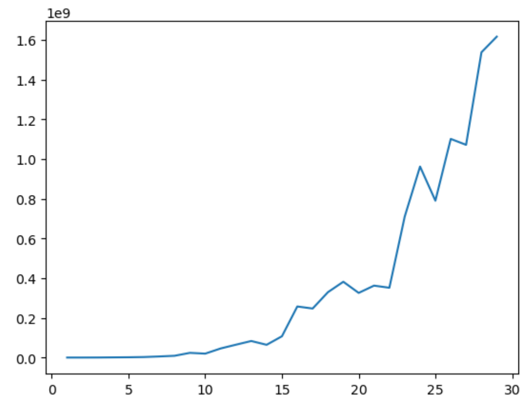
Természetesen csekély az esély arra, hogy egy-egy algoritmus csupán ilyen egyszerű számítással kifejezhető legyen. Gyakran előfordul ugyanis, hogy a programunk polinom időben fut le, melyet a következő algoritmus szemléltet:

	költség	végrehajtási szám
def demo(a):		
x = 1	C1	1
for i in range(0,2):	C2	N
x = x*2	C3	N
for i in range(0,len(a)):	C4	N
for j in range(0,len(a)):	C5	N
x = x + a[i]+ a[j]	C6	N
x = x**2	C7	1
return x	C8	1

Az algoritmus futási ideje ebből könnyen kiszámítható: 1+(n+1)*1 + (n+1)* (n+1)*1 + 1 +1, melyből következik, hogy O(n) + O(n^2) + 3. Azonban a futásidő elemzésénél már megtanultuk, hogy a leggyorsabban növekvő elemet kell figyelni, így ennek az algoritmusnak a futási ideje O(n^2).

```
def fug(a):
    x = 1
    for i in range(0,2):
        x = x*2
    for i in range(0,len(a)):
        for j in range(0,len(a)):
            x = x + a[i]+ a[j]
    x = x**2
```

```
    print(x)
    return x

def demo():
    time = []
    for n in range(1,30):
        a = np.random.randint(0,50,size=(n))
        time.append(fug(a))

    plt.plot(range(1,30),time)

demo()
```



Gyakorló feladatok *(50 perc)*

Készíts programot az alábbi algoritmusok segítségével, és határozd meg, hogy melyiknek mennyi az futási ideje (Ordó)! Melyik eljárás futási idejének legmagasabb a nagyságrendje?

```
def demo1(n)
        if (n<=0):
            return 1
        else:
                return 1 + demo1(n-1)

def demo2(n)
        if (n<=0):
            return 1
        else:
                return 1 + demo2(n-5)

def demo3(n)
        if (n<=0):
            return 1
        else:
                return 1 + demo3(n/5)


def demo4(n,m,o)
        if (n<=0):
            return 1
        else:
                demo4(n-1,m+1,o)
                demo4(n-1,m,o+1)

def demo5(n)
             for i in range(0,n):
             print(i)
        if (n<=0):
            return 1
        else:
                return 1 + demo3(n-5)
```

Ajánlott kitekintő anyag *(235 perc)*

Motivációs teaser **(32 perc)** – Videó magyar nyelven

Algoritmusok elemzése **(79 perc)** – Videó magyar nyelven

Az ordó jelölés **(18 perc)** – Videó magyar nyelven

Időkomplexitás fogalma **(36 perc)** – YouTube link angol nyelven

Log, Lineáris, Négyzetes, Exponenciális Algoritmusok **(50 perc)** – MIT video link angol nyelven

Algoritmusok és adatszerkezetek gyakorlat - Algoritmusok futásidő elemzése **(20 perc)** - Gelle Kitti - link

SZÉCHENYI

MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap

BEFEKTETÉS A JÖVŐBE