Bonyolultságelmélet gyakorlat – 01
Időigény

Recap: időigény

- ,,pszeudokódjaink" inputjai mindig (akárhány bites) egész számok, vagy azokból álló tömbök (kb `BigIntegerek` és `BigInteger` tömbök), ezt a típust röviden `int`nek fogjuk írni
- a futásidőt az input **méretéhez** képest adjuk meg
- ,,méret": **hány biten** tudjuk leírni az összes bejövő számot, általában n fogja jelölni
- a program/függvény **időigényét** n függvényében adjuk meg, pl. ,,az időigény $n^2 + 3n + 9$" azt jelenti, hogy ennyi elemi lépés után garantáltan megáll, ha n bites az input (,,legrosszabb eset analízis")
- pontos felső korlát helyett **nagyságrendi** korlátot elég adni, az előző pl. $O(n^2)$

Recap: ordó

Ha $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ az $f(n)$ és $g(n)$ nemnegatív, monoton növő függvényekre, akkor $f = O(g)$.

Ha $f = O(g)$ és $g = O(f)$, akkor $f = \Theta(g)$, ekkor **azonos a nagyságrendjük**.

Pl. $n^2 + 3n + 200 = O(n^2)$, hiszen $\lim\limits_{n\to\infty} \frac{n^2+3n+100}{n^2} = 1 < \infty$.

Recap: szorzás, összeadás

- ha ciklusnak elemezzük az időigényét, akkor **összeszorozzuk** a ciklusmag időigényét azzal, ahányszor lefuthat
- ha szekvenciálisan, egymás után vannak kódblokkok, azok időigényét pedig **összeadjuk**
- $O(f) \cdot O(g) = O(f \cdot g)$, $O(f) + O(g) = O(f + g)$, de ha pl. $g = O(f)$, akkor $O(f + g) = O(f)$ (így lesz pl. az $O(n) + O(1) = O(n)$)
- algából tanultunk ennél kifinomultabb becsléseket is (pl. amortizált időigény elemzésnél), de bonyából elég lesz így

1. feladat. Elemezzük az alábbi kód időigényét! Itt és a továbbiakban is az `int` típus a fenti, **akárhány bites egész** számot tárolni képes adattípus az összeadás, kivonás, értékadás elemi műveletekkel. Vegyük elemi műveletnek a `%` maradékos osztást és az `==`, `<` stb. összehasonlításokat is. Pszeudokódjainkban `:=` az értékadás, `==` az összehasonlítás, a sima `=` jelet nem használjuk. Most feltehetjük, hogy $A > 0$.

```
int isPrime(int A) {
  for (int B := 2; B < A; B++) {
    if (A % B == 0) return 0
  }
  return 1
}
```

2. feladat. Elemezzük az alábbi kód időigényét!

```
int topBit(int A) {
  int B := 1
  while (B + B <= A) {
    B := B + B
  }
  return B
}
```

3. feladat. Elemezzük az alábbi kód időigényét!

```
int mod(int A, int B) {
  while (B <= A) {
    A := A - B
  }
  return A
}
```

4. feladat. Elemezzük az alábbi kód időigényét!

```
int mod(int A, int B) {
  if (B <= 2) return 0
  while (B <= A) {
    A := A - B
  }
  return A
}
```

Recap

Ha egy programnak / függvénynek egy n biten tárolható T `int` tömb az inputja, akkor

- T.length-re mindig egy jó becslés az $O(n)$,
- T[i]-re, azaz **egy** tömbelem **értékére** pedig az $O(2^n)$.

5. feladat. Elemezzük az alábbi kód időigényét! Itt már egy **tömb** az input, ilyenkor n a tömbbeli összes szám reprezentálásához szükséges bitek száma lesz. Pszeudokódjainkban `T.length` adja meg a `T` tömb elemeinek számát, indexelni 0-tól `(T.length-1)`-ig fogjuk őket.

```
int tombMax(int[] T) {
  if (T.length == 0) return 0
  int max := T[0]
  for (int i := 1; i < T.length; i++) {
    if (max < T[i]) max := T[i]
  }
  return max
}
```

6. feladat. Elemezzük az alábbi kód időigényét!

```
int tombIncSum(int[] T) {
  int sum := 0
  for (int i := 0; i < T.length; i++) {
    while (T[i] > 0) {
      sum := sum + 1
      T[i] := T[i] - 1
    }
  }
  return sum
}
```

1. feladat megoldása.

A **ciklusmag** (egy maradékos osztás, egy összehasonlítás, és egy if, abban esetleg egy return) **konstans** időigényű. (amit $O(1)$-gyel tudunk jelölni)

A ciklus legrosszabb esetben (ha `A` prímszám) kb. `A`-szor (ha pontosak akarunk lenni, akkor $(\texttt{A}-2)$-szer) fut le, ezt írhatjuk eddig $O(A)$-val.

Az egész ciklus időigénye legrosszabb esetben tehát eddig $O(A) \cdot O(1) = O(A)$.

A végén a return az még egy lépés, tehát összesen $O(A) + O(1) = O(A)$ az időigény.

Ezt még az input n méretéhez képest kell kifejeznünk: az A szám $n = O(\log A)$ **biten** tárolható, ebből $A = O(2^n)$ és ezért az $O(A)$ időigény n függvényében $O(2^n)$, ez a programunk időigénye.

---

2. feladat megoldása.

A lokális változó deklarálása és a végén a return konstans sok elemi lépés, ahogy a ciklusmag is csak egy összeadás és egy értékadás, így már csak az a kérdés marad, hogy hányszor fut le legrosszabb esetben a ciklus.

A `B` változó értéke 1-ről indul és minden iterációban duplázódik, ez addig megy, amíg el nem éri (a duplája) az `A` értékét.

Tehát a k. iterációban B értéke 2^k, a kérdés, hogy mekkora k-ra lesz már $2^k \geq A \Rightarrow k \geq \log A$-ra, azaz a ciklus $O(\log A)$-szor fut le (ha A negatív, akkor egyszer sem, olyankor konstans idő alatt végez a program, ez nyilván nem a legrosszabb eset, így most ezzel nem kell foglalkoznunk).

Az egész kód tehát $O(1) + O(\log A) \cdot O(1) + O(1) = O(\log A)$ lépés után biztosan terminál.

Ismét, mivel A az egyetlen inputunk, így $n = O(\log A)$ és ezért ez a fenti $O(\log A)$ futásidő $O(n)$, azaz **lineáris** időkorlátot jelent.

---

3. feladat megoldása.

Ha $B = 0$ és $A > 0$, akkor ez a kód végtelen ciklusba esik. Ha egyetlen olyan input is van, amin a kód nem áll meg, akkor a kódunk **nem időkorlátos**.

---

4. feladat megoldása.

Az első sor (egy összehasonlítás, egy if, és esetleg egy return), a végső return és a ciklusmag (egy kivonás és egy értékadás) mind $O(1)$ időigényűek, így azt kell csak már kifejeznünk n függvényében, hogy hányszor futhat le a ciklusmag. Ha A negatív, akkor egyszer sem fog, nyilván nem ez lesz a legrosszabb eset, így ezzel megint nem kell foglalkoznunk.

Minden iterációban A értékét csökkentjük B-vel (ami legalább 3), ezt egész addig tesszük, amíg A értéke B alá nem csökken, így $O\left(\frac{A}{B}\right)$-szer fut le a ciklus, ezt kell még kifejeznünk, hogy az inputméret n függvényében mennyi lehet.

Maga az inputméret $n = O(\log A + \log B)$, ennyi bitre van szükség a két bejövő szám ábrázolásához. Az A/B tört értéke annál nagyobb, minél nagyobb A és minél kisebb B; mivel B legalább 3 lesz, ha a ciklus egyáltalán lefut, ami egy 2-bites szám, így ha összesen n bitet tudunk „szétosztani” a két input szám között, az A/B úgy lesz a legnagyobb, ha $B = 3$, az A pedig egy $(n-2)$-bites szám; ezek közül a legnagyobb a $2^{(n-2)} - 1$, így az A/B hányados értéke

$O\left(\frac{2^{(n-2)}-1}{3}\right) = O(2^n)$.

Vagyis, a teljes kód időigénye $O(1) \ + \ O(2^n) \cdot O(1) \ + \ O(1) = \mathbf{O(2^n)}$ lesz.

---

5. feladat megoldása.

A korábbi feladatokhoz hasonlóan az időigény

$$O(1) \ + \ O(1) \ + \ O(\text{T.length}) \cdot O(1) \ + \ O(1) = O(\text{T.length})$$

lesz, ezt kell n függvényében kifejeznünk.

Nyilván egy tömb minden egyes elemének tárolásához kell legalább 1 bit, így ha az egész tömb n biten ábrázolható, akkor legfeljebb n eleme lehet, ezért T.length $= O(n)$. Tehát az egész kód időigénye szintén $O(n)$ lesz.

---

6. feladat megoldása.

A kód elején a változó-deklarálás és inicializálás, és a végén a return $O(1)$ idő. Ezek között két egymásba ágyazott ciklust látunk, elemezzük belülről kifelé: a ciklusmagban van egy összeadás, egy kivonás és két értékadás, ez $O(1)$ lépés. A `while` ciklus teljes időigényéhez meg kéne határozni, hogy (mondjuk) n függvényében hányszor futhat le; mivel minden iterációban eggyel csökken egy tömbelem értéke, és addig fut, amíg nullára nem csökken, így az pl. egy jó becslés, hogy mekkora lehet a tömbelemek közül a legnagyobbnak az **értéke**. A tömb összes eleme összesen n biten reprezentálódik (hiszen az az input), így egy-egy tömbelem is el kell férjen n biten, tehát értékük maximum 2^n lehet, ezért a `while` ciklus legfeljebb 2^n-szer fog lefutni minden egyes elemre. Ezért a 4-7. sorok teljes időigénye $O(2^n) \cdot O(1) = O(2^n)$ lesz, ez pedig a 3. sorban kezdődő `for` ciklus magja, ami pedig legfeljebb n-szer fut le (ahogy az előző feladatban is, egy n biten tárolható tömbnek legfeljebb n eleme lehet, mert minden elem ,,elhasznál" legalább egy bitet), így a 3-8. sorok teljes időigénye $O(n) \cdot O(2^n) = O(n \cdot 2^n)$. Ehhez hozzáadva a 2. és 9. sor $O(1)$ időigényét, továbbra is $O(n \cdot 2^n)$-et kapunk.

Megjegyzés.

A bonyagyak szempontjából elég így kielemezni a ciklusokat, minden ciklusra ,,globálisan" kiszámolva a magra is és az iterációk számára is egy-egy felső korlátot, ebben a feladatban azért tudunk adni jobb korlátot is: a kód minden egyes tömbelemet egyesével lecsökkent nullára, így valójában a ,,teljes" időigénye $O\left(\sum_{i=0}^{\text{T.length}-1} T[i]\right)$. Ha a tömbelemek számát mondjuk k-val jelöljük, és azok rendre n_1, n_2, …, n_k biten reprezentálhatók, akkor egyrészt $n = n_1 + \ldots + n_k$ az össz méret, másrészt az i-edik tömbelem legfeljebb 2^{n_i} lesz (most 1-től k-ig indexelve azokat kivételesen), így a futásidő pedig legfeljebb $2^{n_1} + 2^{n_2} + \ldots + 2^{n_k}$. Mivel az összes n_i legalább 1, a tagok mindegyike legalább $2^1 = 2$, ezért az összegük legfeljebb annyi, mint a szorzatuk, azaz a futásidő legfeljebb $2^{n_1} \times 2^{n_2} \times \ldots \times 2^{n_k} = 2^{n_1+n_2+\ldots+n_k} = 2^n$, így az $O(n \cdot 2^n)$-nél ezzel a módszerrel egy jobb, $O(2^n)$-es felső korlátot tudunk megadni a futásidőre.

(Ilyen pontosságú elemzést nem várunk el gyakorlaton, elég a ,,Megoldás" részben leírt módszert alkalmazni.)