Alapvető algoritmusok

5. előadás

Dr. Pătcaș Csaba

BABEȘ-BOLYAI TUDOMÁNYEGYETEM
Matematika és Informatika Kar

Tartalom

A top-down és bottom-up vegyítése

Példa: Törzstényezőkre bontás

- A továbbiakban hatékonyabbá tesszük a fenti megoldást és egyben szemléltetjük is a vegyes módszert.
- Észrevesszük, hogy a `Felbont` alprogram különösen előnytelenül viselkedik, ha *n* egy adott ponton egy nagy prímszám lesz.
- Ebben az esetben ki is léphetnénk az első `Amíg` struktúrából:

```
...
AMÍG ((n != 1) ÉS (Prím(n) = HAMIS)) végezd el:
  ...
VÉGE(Amíg)
HA (n > 1) akkor
  m = m + 1
  a[m].tényező = n
  a[m].kitevő = 1
VÉGE(Ha)
```

A top-down és bottom-up vegyítése

Példa: Törzstényezőkre bontás

- Nem szerepel a feladat szövegében, hogy mekkora lehet az n (a megadott számok maximális értéke).
- A „megrendelővel” való egyeztetés után pontosítjuk, hogy ez legtöbb 1 000 000 lehet, ezt a továbbiakban `MAX_N`-el jelöljük.
- Ennek az információnak a tudatában és figyelembe véve, hogy `Prím(n)` hívás az `Amíg` ciklus minden egyes iterációjában meghívódik és a `Felbont` alprogramot k-szor hívjuk meg, úgy döntünk, hogy nagyon hatékonyan kell ellenőriznünk, hogy a megmaradt szám prím-e.
- Mivel nincs kikötésünk a felhasználható memóriára és az n-re adott határ kényelmesen megengedi, Eratoszthenész szitáját fogjuk használni.

A `Megold` alprogram így alakul:

```
ALGORITMUS Megold(k, számok)
  Eratoszthenész(MAX_N, prím) //felépíti a prím logikai tömböt
  MINDEN i = 1, k végezd el:
    Felbont(számok[i], m, felbontás, prím)
    Kiír(számok[i], m, felbontás)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

A top-down és bottom-up vegyítése

Példa: Törzstényezőkre bontás

A `Felbont()` alprogram most valahogy így néz ki:

```
...
AMÍG ((n != 1) ÉS (prím[n] = HAMIS)) végezd el:
  ...
VÉGE(Amíg)
HA (n > 1) akkor
  m = m + 1
  felbontás[m].tényező = n
  felbontás[m].kitevő = 1
VÉGE(Ha)
```

- Úgy döntünk, hogy tovább feszítjük a hatékonysági határokat.
- Észrevesszük, hogy a törzstényezőkre bontásban csak prímtényezők szerepelhetnek, ennek ellenére mi egyenként növeljük az osztót, így olyanokat is végigpróbálunk, amelyek biztosan nem lesznek a megoldásunk részei.
- Innen jön az ötlet, hogy felhasználva a `prím` logikai tömböt amit Eratoszthenész szitájával generáltunk, felépítünk egy `prímek` tömböt, amely az összes prímszámot fogja tartalmazni `MAX_N`-ig és az `osztó` változó csak innen fog értékeket felvenni.

A top-down és bottom-up vegyítése

Példa: Törzstényezőkre bontás

A `Megold` alprogram így alakul:

```
ALGORITMUS Megold(k, számok)
  Eratoszthenész(MAX_N, prím)
  FelépítPrímek(MAX_N, prím, prímek)
  MINDEN i = 1, k végezd el:
    Felbont(számok[i], m, felbontás, prím, prímek)
    Kiír(számok[i], m, felbontás)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

A bottom-up (lentről felfele) típusú programozás

Példa: Törzstényezőkre bontás

```
ALGORITMUS Felbont(n, m, felbontás, prím, prímek)
  m = 0
  i = 1
  AMÍG ((n != 1) ÉS (prím[n] = HAMIS)) végezd el:
    osztó = prímek[i]
    hatvány = 0
    AMÍG (n % osztó = 0) végezd el:
      hatvány = hatvány + 1
      n = n / osztó
    VÉGE(Amíg)
```

```
    HA (hatvány != 0)
      m = m + 1
      felbontás[m].tényező = osztó
      felbontás[m].kitevő = hatvány
    VÉGE(Ha)
    i = i + 1
  VÉGE(Amíg)
  HA (n > 1) akkor
    m = m + 1
    felbontás[m].tényező = n
    felbontás[m].kitevő = 1
  VÉGE(Ha)
VÉGE(Algoritmus)
```

A bottom-up (lentről felfele) típusú programozás

Példa: Törzstényezőkre bontás

A prímek tömb felépítése könnyen megy:

```
ALGORITMUS FelépítPrímek(n, prím, prímek)
  k = 0
  MINDEN i = 2, n végezd el
    HA (prím[i]) akkor
      k = k + 1
      prímek[k] = i
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Hogyan tudnánk még hatékonyabban megvalósítani?

A prímek tömb felépítése könnyen megy:

```
ALGORITMUS FelépítPrímek(n, prím, prímek)
  k = 0
  MINDEN i = 2, n végezd el
    HA (prím[i]) akkor
      k = k + 1
      prímek[k] = i
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Hogyan tudnánk még hatékonyabban megvalósítani?
Tulajdonképpen csak $\sqrt{n}$-ig van szükségünk a prímekre és mivel ismerjük `MAX_N` értékét, ezeket előre le is generálhatnánk, hogy ne kelljen a program minden alkalommal kiszámítsa.

A bottom-up (lentről felfele) típusú programozás

Eratoszthenész szitája

- Kezdetben minden számról n-ig feltételezzük, hogy prím.
- Elindulva a legkisebb prímszámtól, egyenként haladva, ha találunk egy számot amelyet nem húztunk még ki, akkor tudjuk, hogy az prím.
- Ennek kihúzzuk az összes többszörösét, a négyzetétől indulva.
- Mikor elértünk $\sqrt{n}$-ig, befejeztük és megtaláltunk az összes prímszámot n-ig.
- Az algoritmus egyszerűsége ellenére nagyon hatékony, időbonyolultsága $O(n \log \log n)$.
- Léteznek más szita algoritmusok is, melyek Eratoszthenész szitájának különböző gyenge pontjain javítanak, ilyen például a memóriaigény, amely $\Theta(n)$.

A bottom-up (lentről felfele) típusú programozás

Eratoszthenész szitája

```
ALGORITMUS Eratoszthenész(n, prím)
  MINDEN i = 2, n végezd el:
    prím[i] = IGAZ
  VÉGE(Minden)
```

A bottom-up (lentről felfele) típusú programozás
Eratoszthenész szitája

```
  gyök = sqrt(n)
  MINDEN i = 2, gyök végezd el:
    HA (prím[i]) akkor
      j = i * i
      AMÍG (j <= n) végezd el:
        prím[j] = HAMIS
        j = j + i
      VÉGE(Amíg)
    VÉGE(Ha)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Mennyi a bemutatott törzstényezőkre bontó algoritmus bonyolultsága?

Elég nehéz megállapítani a bemutatott algoritmus bonyolultságát a legrosszabb esetben, próbáljuk megsaccolni!
A helyes válasz:

Mennyi a bemutatott törzstényezőkre bontó algoritmus bonyolultsága?

Elég nehéz megállapítani a bemutatott algoritmus bonyolultságát a legrosszabb esetben, próbáljuk megsaccolni!
A helyes válasz:
A legrosszabb eset akkor áll fenn, ha mindegyik megadott szám két egymáshoz közel álló nagy prímszám szorzata. Ekkor végig kell próbálgatni az összes osztót a kisebbik prímszámig. A bonyolultság így alakul:
$O(MAX\_N \log\log MAX\_N + k \cdot p)$, ahol p a prímek száma $\sqrt{MAX\_N}$-ig, amit közelíthetünk $p \simeq \frac{\sqrt{MAX\_N}}{\log \sqrt{MAX\_N}}$-el.
Gondolhattuk volna, hogy a legrosszabb eset az, amikor mindegyik szám egy nagy kettő hatvány, viszont ekkor a bonyolultság csak
$\Theta(MAX\_N \log\log MAX\_N + k \log MAX\_N)$.

Tartalom

Lépések finomítása

- Megfigyelhettük az előző feladatok megoldása közben (főleg a top-down módszernél), hogy eleinte csak körvonalaztuk a megoldást és utána részleteztük.
- Ezt nevezzük a lépések finomításának, amely a kezdeti vázlattól a végleges kidolgozott algoritmusig vezet.
- Kiindulunk a feladat specifikációjából és fentről lefele megtervezzük az algoritmust.
- Újabb meg újabb változatokat dolgozunk ki, amelyek eleinte tartalmaznak természetes nyelven (magyarul, angolul stb.) leírt magyarázó sorokat is, ezeket idővel utasításokra írjuk át.
- Így az algoritmusnak több egymás utáni változata lesz, amelyek egyre bővülnek egyik változattól a másikig.

Optimalizálás

- Optimalizáláskor egy kész algoritmus hatékonyságát próbáljuk növelni.
- Ekkor már túl vagyunk a finomításon, az algoritmus teljesen kész, de elégedetlenek vagyunk a teljesítményével.
- Megpróbáljuk gyorsítani az algoritmust, vagy csökkenteni a memóriaigényt.
- Igazi optimalizálás akkor történik, ha anélkül sikerül jobb algoritmust találni, hogy változna a másik bonyolultság, vagy szerencsés esetben az is csökken.
- Ellenkező esetben idő-memória kompromisszumról beszélünk (time-memory tradeoff).
- Ezt a folyamatot megfigyelhettük a törzstényezőkre bontásnál, de a maximális összegű tömbszakasz feladatánál is.

- Ha a törzstényezőkre bontás feladatánál az n sokkal nagyobb lett volna, már nem használhattuk volna Eratoszthenész szitáját a memóriaigény miatt.
- Ha a prímszámellenőrzés nem egy cikluson belül lett volna, hanem csak párszor hívodott volna meg, szükségtelen lett volna n-ig az összes számról megállapítani, hogy prím-e.
- Eratoszthenész szitája ugyan nagyon hatékony, de ha csak kevés számról kell ellenőrizzük, hogy prímek-e, vannak sokkal hatékonyabb módszereink.

Prím-e?

Első változat

- Tudjuk, hogy egy szám akkor prím, ha pontosan két osztója van: 1 és önmaga.
- Ebből kiindulva, számoljuk meg a szám osztóit, végigpróbálva minden lehetőséget 1-től n-ig.
- Így $\Theta(n)$ időbonyolultságú és $\Theta(1)$ memóriabonyolultságú algoritmust kapunk, ami rögtön jobb mint Eratoszthenész szitája, pedig még csak az első változatnál tartunk.

Prím-e?
Első változat

```
ALGORITMUS Prím1(n, prím)
  osztók_száma = 0
  MINDEN osztó = 1, n végezd el:
    HA (n % osztó = 0) akkor
      osztók_száma = osztók_száma + 1
    VÉGE(Ha)
  VÉGE(Minden)
  prím = (osztók_száma = 2)
VÉGE(Algoritmus)
```

Prím-e?

Második változat

- Feleslegesen sok osztást végzünk.
- Ha 2 és $n/2$ között nincs egyetlen osztó sem, akkor biztos nincs $n/2$ és n között sem.
- Ezzel az észrevétellel megmaradna a $\Theta(n)$ időbonyolultság a legrosszabb esetben, de a jelölésben „elrejtett” konstans a felére csökkenne.
- Mivel az osztókat tudjuk párosítani, vagyis ha találtunk egy d osztót, egyben találtunk egy n/d osztót is, következik, hogy elég csak $\sqrt{n}$-ig keresni az osztókat.
- Ha addig nem találtunk, utána sem fogunk, mert megtaláltuk volna a párját korábban.
- Így az algoritmusunk bonyolultságát $O(\sqrt{n})$-re csökkentettük.

Prím-e?
Második változat

```
ALGORITMUS Prím2(n, prím)
  HA (n = 1) akkor
    prím = HAMIS
  KÜLÖNBEN
    prím = IGAZ
    ngy = sqrt(n)
    MINDEN osztó = 2, ngy végezd el:
      HA (n % osztó = 0) akkor
        prím = HAMIS
      VÉGE(Ha)
    VÉGE(Minden)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Prím-e?

Harmadik változat

- Ha találtunk egy osztót, a számunk biztosan nem prím.
- Ekkor leállíthatjuk az algoritmust.
- Ezt megvalósíthatjuk úgy, hogy `MINDEN` helyett `AMÍG` ciklust használunk.
- Egy másik lehetőség a `break` vagy `return` utasítások használata.

Prím-e?
Harmadik változat

```
ALGORITMUS Prím3(n, prím)
  HA (n = 1) akkor
    prím = HAMIS
  KÜLÖNBEN
```

Prím-e?
Harmadik változat

```
    prím = IGAZ
    osztó = 2
    ngy = sqrt(n)
    AMÍG (prím ÉS (osztó <= ngy)) végezd el:
      HA (n % osztó = 0) akkor
        prím = HAMIS
      KÜLÖNBEN
        osztó = osztó + 1
      VÉGE(Ha)
    VÉGE(Amíg)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Prím-e?

Negyedik változat

- A páros számok mind oszthatóak 2-vel, így a 2 kivételével nem prímek.
- Ha megszabadulunk a páros számok vizsgálatától, felesleges páros számokkal osztani, hiszen páratlan számnak csak páratlan osztója lehet.
- Az időbonyolultság $O(\sqrt{n})$ marad, de javítottunk a konstans szorzón legalább egy kétszeres faktorral.

Prím-e?

Negyedik változat

```
ALGORITMUS Prím4(n, prím)
  HA (n = 1) akkor
    prím = HAMIS
  KÜLÖNBEN
    HA (n % 2 = 0) akkor
      prím = (n = 2)
    KÜLÖNBEN
```

Prím-e?

Negyedik változat

```
      prím = IGAZ
      osztó = 3
      ngy = sqrt(n)
      AMÍG (prím ÉS (osztó <= ngy)) végezd el:
        HA (n % osztó = 0) akkor
          prím = HAMIS
        KÜLÖNBEN
          osztó = osztó + 2
        VÉGE(Ha)
    VÉGE(Amíg)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Prím-e?
Ötödik változat

- Tudjuk, hogy minden 3-nál nagyobb prímszám $6k \pm 1$ alakú.
- Figyelem, ez nem jelenti azt, hogy minden ilyen alakú szám prím!
- Viszont ha egy 3-nál nagyobb szám nem ilyen alakú, akkor biztosan nem prím!

Prím-e?

Ötödik változat

```
ALGORITMUS Prím5(n, prím)
  HA (n = 1) akkor
    prím = HAMIS
  KÜLÖNBEN
    HA (n % 2 = 0) akkor
      prím = (n = 2)
    KÜLÖNBEN
      HA (n <= 5)
        prím = IGAZ
      KÜLÖNBEN
        HA (((n - 1) % 6 != 0) ÉS ((n + 1) % 6 != 0))
          prím = HAMIS
        KÜLÖNBEN
          prím = IGAZ \\tovább ugyanaz mint az előző algoritmusban
```

Prím-e?
Hatodik változat

- Hogyan tudjuk az előző megoldás felhasználásával tovább gyorsítani az algoritmust?

Prím-e?

Hatodik változat

- Hogyan tudjuk az előző megoldás felhasználásával tovább gyorsítani az algoritmust?
- Mivel az összetett számok legkisebb osztója biztosan prím, elég csak a $6k \pm 1$ alakú osztókat végigpróbálni.
- Így ahelyett, hogy kettőnként ellenőriznénk egy osztót, hatonként ellenőrzünk kettőt, vagyis várhatóan egy másfélszeres faktorral gyorsabb lesz az algoritmusunk a legrosszabb esetben az előzőhöz képest.

Prím-e?

Megjegyzések az algoritmusok időbonyolultságával kapcsolatban

- Kicsit nehezebb belátni, hogy a bemutatott algoritmusok exponenciális időbonyolultságúaknak számítanak.
- Ennek oka, hogy a bemenet mérete nem n, hanem n számjegyeinek (vagy bitjeinek) a száma, vagyis $k = \log n$.
- Ekkor $\sqrt{n}$-t úgy kapjuk meg, hogy egy konstanst emelünk a $k/2$. hatványra.
- 2002-ben publikálták az AKS algoritmust, amely az első polinomiális idejű prímellenőrző algoritmus volt.

Tartalom

Algoritmusok helyességének ellenőrzése

- Egy algoritmust általában helyesnek tartunk, ha véges számú lépés után, megengedett bemeneti adatokból meghatározza a feladat megoldásának megfelelő kimeneti adatokat.
- Léteznek elméleti és gyakorlati eszközeink, melyek hibakeresésre szorítkoznak, ilyenek a fekete doboz és az átlátszó doboz módszerei, illetve az ellenőrző táblázat.
- Matematikai eszközökkel bizonyíthatjuk az algoritmus helyességét és végességét, ilyen a ciklusinvariáns kimutatása (lásd jegyzet).

Ellenőrző táblázat

- Az algoritmusban szereplő változók értékeinek módosulását követjük lépésenként.
- Az algoritmus lépéseit elvégezzük és a változók értékeit bevezetjük a táblázatba.
- A kezdeti értékeket választhatjuk véletlenszerűen, de szükséges bizonyos feltételeknek eleget tevő teszteseket is keresni.
- Ha konkrét kezdőértékeket választottunk, a végeredmény alapján eldönthetjük, hogy ennek a tesztnek az esetében helyesen működik-e az algoritmus.
- Egyszerűbb algoritmusok esetén használhatjuk az ellenőrző táblázatos módszert formális bizonyításra is (példa: két változó cseréje összeadással és kivonással).

A fekete doboz módszere

- Az algoritmust fekete doboznak tekintjük, amelynek tartalma nem érdekel bennünket.
- Csak a bemeneti adatokra figyelünk és azokra a kimeneti adatokra, amelyeket az algoritmust kódoló program futtatása eredményeként kapunk.
- A következő kategóriájú bemeneti adatokra érdemes tesztelni:
  1. Jellemző bemeneti adatok (gyakori, általános)
     Pl. `Prím(5)`, `Prím(24)`, `Prím(25)`
  2. Sajátos bemeneti adatok (szélsőséges tesztesetek)
     Pl. `Prím(1)`, `Prím(2)`, `Prím(1000000001)`
  3. Nem megengedett adatok (figyelmetlenségből, esetleg más program eredményeként kapott adatok)
     Pl. `Prím(-10)`, `Prím(abc)`
- Ezt a módszert használják programozási versenyeken (és a DOMjudge is) és bizonyos szinten a software termékek automatizált tesztelési rendszereiben is.

Az átlátszó doboz módszere

- Az adatokat úgy választjuk ki, hogy az algoritmus belső szerkezetére figyelünk és azt a célt követjük, hogy ezt valamennyi ágán lefuttatva ellenőrízhessük. Pl: `Prím5(1)`, `Prím5(2)`, `Prím5(4)`, `Prím5(3)`, `Prím5(20)`, `Prím5(19)`, `Prím5(35)`
- Minden érett gondolkodású programozó ezt a módszert kellene használja elsődlegesen, de a fekete doboz módszerét sem szabad teljesen figyelmen kívül hagyni.

Tartalom

LNKO és LKKT

Feladat

Határozzuk meg két szám legnagyobb közös osztóját és legkisebb közös többszörösét!

- Az LNKO kiszámításához Eukleidész algoritmusát használjuk.
- Ezt követően a legkisebb közös többszöröst a következő képlettel határozzuk meg: $lkkt(a,b) = \frac{ab}{lnko(a,b)}$
- Eukleidész algoritmusának ismételt kivonásokkal való implementálása nagyon rossz hatékonysággal rendelkezik, próbáljuk csak ki $lnko(2000000000, 1)$-re.
- Az osztási maradékra alapuló változat időbonyolultsága $O(\log \min(a,b))$, a legrosszabb eset két egymás utáni Fibonacci-számra való hívás esetén fordul elő.
- Az algoritmus kiterjesztett változatát használhatjuk arra is, hogy olyan x és y értékeket találjunk, amelyekre $ax + by = lnko(a,b)$. Ez a változat nem tartozik az előadás anyagához.

Eukleidész algoritmusa

```
ALGORITMUS Lnko(x, y)
  a = x
  b = y
  AMÍG (b != 0)
    r = a % b
    a = b
    b = r
  VÉGE(Amíg)
  VISSZATÉRÍT: a
VÉGE(Algoritmus)
```

Gyors hatványozás

Feladat

Adott x valós és n természetes szám. Számítsuk ki x^n-t!

- A feladat triviális megoldása $n-1$ darab szorzást végez, így $\Theta(n)$ bonyolultságú.
- Viszont megoldhatjuk a feladatot lényegesen gyorsabban, $\Theta(\log n)$ időben.
- Ehhez ismételt négyzetre emeléseket használunk, pl. $x^{11} = ((x^2)^2)^2 \cdot x^2 \cdot x$
- Gyakorlatilag felírjuk n-t kettes számrendszerben és x^n-t felírjuk x^{2^k} alakú számok szorzataként. Pl. $11 = (1011)_2$, vagyis $x^{11} = x^8 \cdot x^2 \cdot x$
- Számos gyors hatványozó algoritmus ismert, az itt bemutatott a klasszikus változat, melyet már Muhammad ibn Musa al-Khwarizmi is ismert.

Gyors hatványozás

```
ALGORITMUS Gyorshatvány(x, k, eredmény)
  eredmény = 1
  alap = x
  kitevő = k
  AMÍG (kitevő > 0) végezd el:
    HA (kitevő % 2 = 1) akkor
      eredmény = eredmény * alap
    VÉGE(Ha)
    alap = alap * alap
    kitevő = kitevő / 2
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Fibonacci

Feladat

Határozzuk meg az n. Fibonacci-számot!

- Ismerjük a Fibonacci-számok rekurzív definícióját: $F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$
- Ezt direkt módon alkalmazva felépíthetjük a Fibonacci-sorozatot, melynek elemei exponenciálisan nőnek.
- Az így kapott algoritmus $\Theta(n)$ bonyolultságú.

Fibonacci

```
ALGORITMUS Fibonacci(n)
  HA (n < 2) akkor
    VISSZATÉRÍT: n
  VÉGE(Ha)
  a = 0
  b = 1
  MINDEN i = 2, n végezd el
    c = a + b
    a = b
    b = c
  VÉGE(Minden)
  VISSZATÉRÍT: c
VÉGE(Algoritmus)
```

- Több lehetőség is létezik az n. Fibonacci-szám meghatározására $\Theta(\log n)$ időben.
- Az egyik a következő mátrix-egyenlőségre alapul:
$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$
- Mivel a gyors hatványozás algoritmusát mátrixokra is alkalmazhatjuk és 2×2-es mátrixokat konstans számú művelettel tudunk összeszorozni, a kezdeti mátrixot a megfelelő hatványra emelhetjük logaritmikus időben.

A fentiek felhasználásával tervezzünk hatékony algoritmust az alábbi feladat megoldására! Milyen időbonyolultsággal rendelkezik az algoritmus?
Határidő: 2023 november 5., 23:59
Hova: Canvas privát üzenet
Mit: forráskód és rövid magyarázat
Mennyi: legtöbb 2 pont 40-es skálán a H1 pontszámba (lineáris algoritmus nem ér pontot)
Hányszor: mindenki egyszer próbálkozhat

Feladat

Írjunk algoritmust, amely megadja a Fibonacci-sorozat egy adott számnál kisebb elemeinek számát!

Erősen összetett számok

Első és második változat

Feladat

Erősen összetett számnak nevezzük azt a természetes számot, amelynek több osztója van, mint bármely, nála kisebb természetes számnak. Írjunk programot, amely adott n-ig erősen összetett számokat keres!

- A legegyszerűbb megoldásban minden számra 1-től n-ig, az adott számig vagy a feléig megyünk és úgy számoljuk az osztókat, ennek a megoldásnak a bonyolultsága $O(n^2)$.
- Felhasználva a korábbi megjegyzést, megszámolhatjuk párosával is az osztókat, így csak az adott szám gyökéig kell mennünk és a bonyolultságot $O(n\sqrt{n})$-re javítottuk.

Erősen összetett számok

Harmadik változat

- Ismert a következő képlet.
- Ha egy szám prímtényezőkre bontása $n = p_1^{k_1} \cdot p_2^{k_2} \dots p_m^{k_m}$ alakú, akkor n osztóinak száma $(k_1 + 1)(k_2 + 1) \dots (k_m + 1)$.
- Ezt felhasználva, elég mindegyik számot törzstényezőkre bontani a tárgyalt algoritmussal és alkalmazni a képletet.
- A látottak alapján az algoritmusunk bonyolultsága $O(n \cdot \frac{\sqrt{n}}{\log \sqrt{n}})$ lesz, de gyakorlatban sokkal gyorsabban fog futni, mert csak kevés számra kell végigfutni az összes prím osztón.

Erősen összetett számok

Negyedik változat

Módosíthatjuk valahogy Eratoszthenész szitáját, úgy, hogy megoldja a feladatot még hatékonyabban?

Tökéletes számok

Feladat

Írjunk algoritmust, amely megkeresi és kiírja az első n tökéletes számot! Egy szám tökéletes, ha egyenlő a nála kisebb osztóinak összegével, például $6 = 1 + 2 + 3$.

- Ismét kereshetünk osztókat egyesével vagy párosával.
- Felhasználva a osztók számára vonatkozó képletet, a prímtényezőkre bontás után generálhatjuk direkt módon az osztókat.
- Minden egyes i. hatványkitevő felveheti az összes értéket 0-től k_i-ig és ezeket minden lehetséges módon kombinálva megkapjuk az összes osztót.
- Ehhez a backtracking módszert kell használnunk, amiről a későbbiekben fogunk tanulni.
- Ezzel a módszerrel például 2^{20} osztóinak összegét kb. 20 lépésből kapjuk meg kb. 1000 helyett.

Tökéletes számok
(avagy amikor én tanulok az elsőévesektől)

Viszont ha már megvan a prímtényezőkre bontás, akkor kis matekezéssel az osztók összege:

$$p_1^0 \cdot p_2^0 \cdot \ldots \cdot p_m^0 + \ldots + p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_m^{k_m} =$$
$$(p_1^0 + p_1^1 + \ldots + p_1^{k_1}) \cdot (p_2^0 + p_2^1 + \ldots + p_2^{k_2}) \cdot \ldots \cdot (p_m^0 + p_m^1 + \ldots + p_m^{k_m}) =$$
$$\frac{p_1^{k_1+1}-1}{p_1-1} \cdot \frac{p_2^{k_2+1}-1}{p_2-1} \cdot \ldots \cdot \frac{p_m^{k_m+1}-1}{p_m-1}$$

Példa:

$24 = 2^3 \cdot 3$

$1 + 2 + 3 + 4 + 6 + 8 + 12 + 24 = 60$

$\frac{p_1^{k_1+1}-1}{p_1-1} \cdot \frac{p_2^{k_2+1}-1}{p_2-1} = \frac{2^4-1}{2-1} \cdot \frac{3^2-1}{3-1} = 15 \cdot 4 = 60$

Tartalom

A rendezés feladata

Feladat

Adott egy n elemű a sorozat, amelynek keressük azt a permutációját, amelyre $a_1 \leq a_2 \leq \ldots \leq a_n$ (vagy $\leq$ helyett $\geq$).

- Knuth 33 rendező algoritmust írt le *A számítógép programozásának művészetében*, mi megelégszünk kevesebbel is :)
- Az összehasonlításon alapuló rendezésekről bizonyítható, hogy legkevesebb $\Theta(n \log n)$ összehasonlítást kell végezzenek.
- Az első részben pár egyszerűbb rendezést veszünk ebből a kategóriából, az Összefésüléses rendezést (Mergesort) és a Gyorsrendezést (Quicksort) az Oszd meg és uralkodj (Divide et impera) módszerről szóló fejezetre hagyjuk.

Stabil és helyben rendezés

Definíció

Egy rendezésről azt mondjuk, hogy helyben (in place) rendez, ha konstans méretű pluszmemóriát használ, vagyis memóriabonyolultsága $\Theta(1)$.

Definíció

Egy rendezésről azt mondjuk, hogy stabil, ha az egyenlő kulcsú elemek ugyanolyan sorrendben szerepelnek a rendezett sorozatban, mint az eredetiben. Ennek akkor van jelentősége, ha a nem csak a kulcsokat kell rendezni, hanem más adatokat is.

Buborékrendezés (Bubblesort)

Első változat

- Az alapötlet, hogy mindig két egymás utáni elemet ellenőrzünk, „buborékot rajzolunk” köréjük gondolatban.
- Ha a két elem rossz sorrendben van, akkor felcseréljük őket.
- Ezt folyamatot elvégezzük n-szer a teljes tömbre.

Buborékrendezés (Bubblesort)

Első változat

```
ALGORITMUS Buborékrendezés1(n, a)
  MINDEN i = 1, n végezd el:
    MINDEN j = 1, n - 1 végezd el:
      HA (a[j] > a[j + 1]) akkor
        Felcserél(a[j], a[j + 1])
      VÉGE(Ha)
    VÉGE(Minden)
  VÉGE(Minden)
VÉGE(Algoritmus)
```

Első változat
Példa, bonyolultság

Példa: `a = [10 9 2 3 5]`

Példa: `a = [10 9 2 3 5]`

- Minden esetben pontosan $n(n-1)$ összehasonlítást hajtunk végre, tehát legjobb, átlag és legrosszabb esetben is az időbonyolultság $\Theta(n^2)$.
- Konstans méretű plusz memóriát használunk (ciklusváltozók, esetleg a változócseréhez szükséges változó), így a memóriabonyolultság $\Theta(1)$, vagyis a buborékrendezés helyben (in place) rendez.
- Az egyenlő elemek sorrendje megmarad, vagyis stabil.

Második változat

Ötlet

- Észrevesszük, hogy felesleges n-szer végighaladni a sorozaton, hiszen miután rendezetté vált, már nem módosítunk rajta.
- A `rendben` logikai változóban fogjuk tárolni, hogy a sorozat rendezetté vált-e.
- Kezdetben mindig feltételezzük, hogy rendezett a sorozat, de ha egy cserét kellett végrehajtani, akkor a feltételezésünk hamis.

Buborékrendezés (Bubblesort)

Második változat

```
ALGORITMUS Buborékrendezés2(n, a)
  ISMÉTELD
    rendben = IGAZ
    MINDEN i = 1, n - 1 végezd el:
      HA (a[i] > a[i + 1]) akkor
        Felcserél(a[i], a[i + 1])
        rendben = HAMIS
      VÉGE(Ha)
    VÉGE(Minden)
  AMEDDIG (rendben)
VÉGE(Algoritmus)
```

Második változat

Példa, bonyolultság

Példa: `a = [2 3 4 5 1]`

Második változat

Példa, bonyolultság

Példa: `a = [2 3 4 5 1]`

- A legjobb eset, amikor a tömb már rendezett, ekkor csak egyszer járjuk be, tehát a bonyolultság $\Theta(n)$.
- A legrosszabb eset, amikor a legkisebb elem a tömb végén van, például ha a tömb ellentétesen rendezett ahhoz képest, mint amilyen sorrendbe szeretnénk rendezni. Ekkor $n-1$ bejárásra van szükségünk, vagyis a bonyolultság $\Theta(n^2)$.
- Az átlag eset szintén négyzetes.
- A második változat is helyben dolgozik és stabil.