



Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

---

# Prímszita algoritmusok összehasonlítása

Vatai Emil  
Adjunktus

Nagy Péter  
Programtervező Informatikus BSc

Budapest, 2018

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>2</b>
<b>2. Felhasználói dokumentáció</b>	<b>3</b>
2.1. A megoldott feladat . . . . .	3
2.2. Felhasznált módszerek . . . . .	3
2.2.1. Init és generator program . . . . .	5
2.2.2. Gui . . . . .	5
2.3. A program telepítése és futtatása . . . . .	7
2.4. Prímek keresése . . . . .	7
2.5. Mintaadatbázis karbantartása . . . . .	8
2.5.1. Adatbázis műveletei . . . . .	10
2.5.2. Adatbázis információk . . . . .	10
2.5.3. Szegmensek összesítése . . . . .	11
2.5.4. Összesítőfájlok összefésülése . . . . .	12
2.6. Szitatáblák ellenőrzése . . . . .	12
2.7. Sziták ellenőrzése . . . . .	13
2.8. Minta megjelenítése . . . . .	14
2.8.1. Minta hozzáadása . . . . .	14
2.8.2. Minta közelítése . . . . .	15
2.9. Sziták mérése szkriptekkel . . . . .	17
<b>3. Fejlesztői dokumentáció</b>	<b>19</b>
3.1. A feladat leírása . . . . .	19
3.2. A program komponensei . . . . .	21
3.3. Az adatbázis . . . . .	22
3.4. A megvalósítás . . . . .	24
3.4.1. Init és generator . . . . .	25
3.4.2. Gui . . . . .	27
3.5. Sziták . . . . .	37
3.5.1. Prioritásos sorok . . . . .	39

3.5.2. Edénysor . . . . .	39
3.6. A program ellenőrzése . . . . .	46
3.6.1. Numerikus pontosság . . . . .	46
<b>4. Irodalomjegyzék</b>	<b>49</b>

# 1. fejezet

## Bevezetés

Sziták összehasonlítása azonos környezetben. Hatékonyság összehasonlítása az elmélet szerint várható értékekkel. Ellenőrzésként a prímek néhány statisztikájának összevetése az elméleti értékekkel, és az ismert eredményekkel. Hatékonyság és implementálhatóság.

## 2. fejezet

# Felhasználói dokumentáció

### A megoldott feladat

A program lehetővé teszi a prímek néhány statisztikájának ábrázolását grafikonon, és összevetését elméleti becsült értékekkel, valamint néhány különböző szita-algoritmus futási idejének megmérését, és grafikus megjelenítését. A programmal egyéb forrásból származó minták megjelenítése is lehetséges.

A prímszámok statisztikáinak előállításához a programhoz tartozik egy optimalizált szita-implementáció, amivel  $2^{64}$ -ig lehet szitálni, és az eredményt rögzíteni. Az elmentett prímlistákat a program összesíti több statisztika szerint, és a megjelenítést az összesítések alapján végzi el.

A különböző szita-algoritmusok futási idejének összehasonlításához a program tartalmazza Atkin[1] szitájának egy implementációját, és Eratoszthenész szitájának néhány variációját. Ezek a sziták a prímek statisztikáinak előállításában nem vesznek részt.

### Felhasznált módszerek

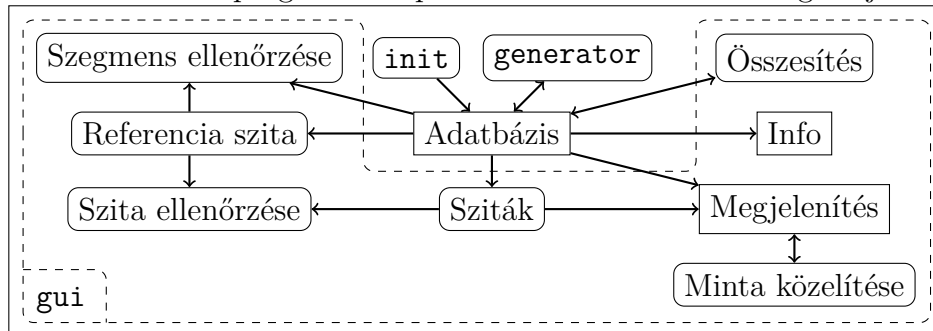
A program két elkülönülő részből áll. A prímszámok statisztikáinak előállításához először szükséges a prímek listájának előállítása. Ehhez egy optimalizált, gyors szita C nyelven készült el. A könnyebb implementálhatóság miatt ez is két külön program, a kis számokat szitáló `init`, és a nagyobb számokkal dolgozó `generator`. A két program a működési elvében hasonló.

A program többi része Java nyelven van megírva, ez a `gui` nevű program. Ez összesíti a prímek listáját, tartalmazza a futási idők összehasonlításához írt szitákat, és jeleníti meg a prímek statisztikáit, és a futási idők mintáit.

Ez a három program az "adatbázis könyvtárba" mentett bináris fájlokon keresztül cserél adatokat. Ide kerülnek az elkészült szitatáblák, és a prímek statisztikái.

Az adatbázis könyvtár nem adatbázis a szokásos értelemben, a klasszikus adatbázis funkciókból csak a megbízható adattárolást végzi el.

2.1. ábra. A program komponensei és adatáramlás diagramja



A programok több prímszitát tartalmaznak. A prímsziták számok egy intervallumáról döntenek el, hogy melyik szám prím, és melyik összetett. A legtöbb implementált szita Eratoszthenész szitájának variációja. Eratoszthenész szitája az egész számok egy adott  $[2, n]$  intervallumába eső számokról dönti el, hogy melyik prím. Ezt a  $pq$  szorzatra bontható számok megjelölésével teszi, ahol a  $p < n$  prím, és  $1 < q \in \mathbb{N}$ . Az algoritmus sorban veszi a számokat a listában kettőtől. Ha egy addig még meg nem jelölt számot talál, akkor az prím, és a listában megjelöli annak többszöröseit, de a prímet nem. Az algoritmus futása után a meg nem jelölt számok a prímek.

```

1:  $n$ : a szitálás felső határa
2: a számok legyenek nem megjelölve, 2-től  $n$ -ig
3: for  $i \leftarrow 2, i \leq n, i \leftarrow i + 1$  do
4:   if  $i$  nincs megjelölve then
5:     for  $j \leftarrow i^2, j \leq n, j \leftarrow j + i$  do
6:       legyen  $j$  megjelölve
7:     end for
8:   end if
9: end for

```

Ennek az egyszerű algoritmusnak a egyik hátránya, hogy  $n$ -ig szitálva  $n - 1$  bit memóriára van szüksége, ez a bitvektor a szitatábla. Erre megoldás a szegmentált szitálás, ahol egy rövidebb, nem feltétlenül 2-től kezdődő intervallumban szitálnak. Ehhez szükség van a prímek listájára, legalább az intervallum végének négyzetgyökéig.

A legegyszerűbb szegmentált szita egy szegmens szitáláshoz minden prímet sorban vesz az intervallum végének négyzetgyökéig, osztással meghatározza, hogy melyik a legkisebb többszöröse a szegmensben, ha van ilyen szám, majd onnantól összeadással szitál a szegmens végéig, ahogy Eratoszthenész szitája is teszi.

Több, egymás utáni szegmens szitálásával az osztás költsége több szegmensre

szétesztható, ha a szita a minden prímről minden szegmens szitálása után megjegyzi, hogy melyik számot szitálna, ami már nagyobb, mint a szegmens vége. A következő szegmens feldolgozásakor a prímmel innen lehet folytatni a szitálást. A módszer hátránya, hogy ehhez el kell tárolni ezeket a prím-pozíció párokat, ennek tárigénye nagyságrendileg  $\mathcal{O}(\sqrt{n})$ , ha legnagyobb szitált szám  $n$ . A hatékony eratoszthenészi sziták ennek a prím-pozíció listának csak azon elemeit veszik figyelembe egy szegmens szitálásakor, amik ténylegesen szitálnak is a szegmensben.

## Init és generator program

A prímszámok listájának generátora C nyelven készült el. A program csak parancsorból futtatható, és a sztenderd könyvtári függvényekből is csak néhányat használ a memóriájában előállított eredmények fájlba írásához. A C nyelv választását a hatékony végrehajtás és memóriafelhasználás, valamint a hordozhatóság indokolja. A generátor elkülönítését a `gui` programtól az automatizálhatóság indokolja.

A prímlisták generátora Eratoszthenész szitájának szegmentált változata, a szitatábla egy  $2^{30}$  hosszú darabját szitálja ki minden iterációban. Ez a generátor önmaga is két részből áll. Az egyszerűbb implementációjú `init` a prímek listáját  $2^{32}$ -ig állítja elő. Ezekre a prímekre nem csak a statisztikák előállítása miatt van szükséges, hanem a `generator`, és a `gui` program szitái is használják, amikor a szitálás intervalluma 1-nél nagyobb számtól kezdődik.

A gyorsabb, de összetettebb `generator`  $2^{32}$ -től  $2^{64} - 2^{34}$ -ig szitál, és a futásához szükséges a prímek listája  $2^{32}$ -ig. A szitatáblát a memóriában a gyorsítótár hatékonyabb a kihasználásához kisebb részszegegmentekre osztja, és a prímeket a nagyságuk és a következő szitálási pozíciójuk szerint csoportosítja. Továbbá nagyobb prímeknek csak harminccal relatív prím többszörőseit veszi figyelembe.

## Gui

A program többi része a Java környezethez készült, ez a `gui` program. Ez végzi a `generator` által előállított szegmensfájlok összesítését, és az elkészült statisztikák megjelenítését.

A `generator` és a `gui` a szegmensfájlokat az adatbázis könyvtáron keresztül adja át egymásnak, és az összesítések is ebbe a könyvtárba kerülnek. A szegmensfájlokra az ellenőrzés és összesítés után már nincs szükség, ezek eltávolíthatóak. Az adatbázisában hosszabb távon egyedül a prímek szegmensenkénti statisztikáit kell tárolni. A `gui` program a statisztikákat csak sorban dolgozza fel, és a statisztikák összmérete sem indokol egy egyszerű bináris fájlnál bonyolultabb megoldást. Az adatvesztés elkerüléséhez elég a statisztikafájl csere alapú felülírása.

A gui képes egyváltozós mintákat megjeleníteni Descartes-féle koordináta rendszerben. A minták értékkészlete a természetes számok, az értelmezési tartomány a valós számok. Ezek a minták a szitálással gyűjtött statisztikákból származhatnak, vagy külső forrásból is betölthető minta az x-y párok felsorolásával.

Lehetséges a mintákat alapfüggvények lineáris kombinációjával közelíteni a legkisebb négyzetek módszerével. A prímstatisztikák elemeinek nagy száma, és a mintaértékek nagy terjedelme miatt szükséges volt nagyobb pontosságú numerikus algoritmusokat választani a közelítéshez, ami a számítási idő növekedésével jár. A Gauss-elimináció és az összegzés algoritmusának kiválasztása a program tesztelésével történt, ismert függvényekből származó minták közelítésével. A választás a tesztközelítések kontrollfüggvényéhez képesti négyzetes eltérést minimalizálja.

A gui program tartalmaz több prím-szita implementációt is, ezeknek az algoritmikus bonyolultsága különböző, de a futási idejük összehasonlíthatóságához a közös részek közös implementációt kaptak, és az eltérő részek hasonló szinten optimalizáltak. A sziták helyességének ellenőrzéséhez a sziták eredménye egymással összehasonlítható.

Elkészült Eratoszthenész szitájának több variációja, mindegyik szegmentáltan működik, a különbség a prímek listájának kezelésében van. A legegyszerűbb implementált eratoszthenészi szita a prímek listáját minden szegmensben végig bejárja. A prioritásos soron alapuló sziták a prímeket részlegesen rendezik az alapján, hogy hány szegmensnyire van az éppen szitált szegmenstől a prím következő többszöröse. Ilyen a bináris kupacot, és az edénysort használó szita.

A Cache Optimalizált Lineáris Szita[2] a prímeket teljesen rendezi az alapján, hogy legközelebb melyik szegmensben fognak szitálni, és a prímek nagyság szerinti csoportosításával a rendezés fenntartásához az elemek memóriabeli mozgását is kerüli.

Atkin szitája[1] az előbbi algoritmusoktól jelentősen eltér, a prímek többszörösei helyett  $n = ax^2 + by^2$  egyenlet megoldásait keresi egy  $n$  szám vizsgálatához,  $x$  és  $y$  pozitív egészek befutásával,  $a$  és  $b$  egész konstansok. A pontos kvadratikusan formát, és  $x$  és  $y$  lehetséges értékeit Atkin az egészek algebrai bővítésével határozza meg,  $n \in 4\mathbb{Z} + 1$  esetén a Gauss-egészek,  $n \in 6\mathbb{Z} + 1$  esetén az Eisenstein-egészek, és  $n \in 12\mathbb{Z} + 11$  esetén a  $\mathbb{Z}[\sqrt{3}]$  segítségével.

A sziták között kapott helyet a próbaosztás, és az erős pszeudoprím teszt[5] is, de ezek a lassúságuk miatt csak igen rövid tartományokon használhatóak. A próbaosztás egy faktorizációs eljárás, ami egyetlen számot nem-triviális szorzatra próbál bontani. Az (erős) pszeudoprím teszt egy prímteszt, egyetlen számról próbálja eldönteni, hogy az prím-e. A prímteszteket általában több nagyságrenddel nagyobb számokra alkalmazzák, mint a szitákat, és teljes módszer helyett sokszor megelégednek egy valószínűségi válasszal.



## A program telepítése és futtatása

A program futtatásához Windows vagy Linux operációs rendszer szükséges, legalább 4GB szabad memóriával. A Java program futtatásához JRE8 szükséges, a fordításához JDK8. A C program fordításához C99 szabványú fordítóprogram kell. A program fejlesztése és tesztelése OpenJDK8-cal és GCC 7.3-mal történt Linux operációs rendszeren.

A CD mellékleten található tarball a Java programot lefordítva is tartalmazza, kicsomagolás után azonnal futtatható. A C programot a "generator" könyvtárban található Makefile segítségével lehet lefordítani:

```
generator$ make clean build
```

A Java program a NetBeans 8-as verziójával készült, parancssorból a project ant fájljának segítségével lehet újrafordítani:

```
gui$ ant clean jar
```

A program futtatásához több előkészített szkript is rendelkezésre áll, ezeket mind a "scripts" könyvtárból lehet indítani. Az előző két fordítást egyben is el lehet végezni a "recompile" szkripttel:

```
scripts$ ./recompile
```

## Prímek keresése

A prímek keresését két program végzi, az `init` program  $2^{32}$ -ig keresi meg és tárolja el a prímeket, a `generator`  $2^{32}$ -től  $2^{64} - 2^{43}$ -ig. Az `init` készítette szitattáblák a `generator` futásához szükségesek.

Mindkét program a számokat  $2^{30}$  hosszú táblánként szitálja, minden tábla első száma  $2^{30}k + 1$  alakú. A szitattáblában csak a páratlan számok vannak nyilvántartva. A szitattáblákat a programok a fájlrendszerben bitvektorként tárolják, egy tábla kb. 64Mb.

Az első négymilliárd szám szitálása a következőképpen indítható el:

```
generator$ ./init.bin ../db
szegmens 0 - kezdet          1 - felkészülés 159 906 063 ns - szitálás 1 122 878 403 ns
szegmens 1 - kezdet 1 073 741 825 - felkészülés 2 353 698 ns - szitálás 1 171 101 336 ns
szegmens 2 - kezdet 2 147 483 649 - felkészülés 2 344 515 ns - szitálás 1 188 330 478 ns
szegmens 3 - kezdet 3 221 225 473 - felkészülés 2 391 660 ns - szitálás 1 199 900 603 ns
```

Ennek eredmény 4 bitvektort tartalmazó bináris fájl, amit a db mappába ment.

A `generator` programot többféleképpen is lehet indítani. Mindegyik esetben két paramétert kell megadni az indításhoz, a számot, ahol szitálást kezdje, és a szegmensek számát, amit ebben a futásban végig kell szitálni. A kezdő számot meg lehet adni decimálisan is, vagy hexadecimálisan is, "0x" prefixszel. A szegmensek

számát is kétféleképpen lehet szabályozni. Az egyik lehetőség fix számú szegmens megadása, decimálisan, vagy hexadecimálisan.

```
generator$ ./generator.bin ../db start 0x100000001 segments 3
szegmens kezdet: 4 294 967 297
szegmensek száma: 3
kis szegmensek mérete: 22
elágazások bitjei: 8
felkészülés 1/4
felkészülés 2/4
felkészülés 3/4
felkészülés 4/4
felkészülés vége
szegmens 4 - kezdet 4 294 967 297 - felkészülés 8 162 031 487 ns - szitálás 729 914 370 ns
szegmens 5 - kezdet 5 368 709 121 - felkészülés 418 ns - szitálás 736 412 117 ns
szegmens 6 - kezdet 6 442 450 945 - felkészülés 385 ns - szitálás 750 022 409 ns
összes szitálás 2 216 348 896 ns
```

A másik mód a háttértáron fenntartandó szabad hely átadása, decimálisan, vagy "0x" prefixszel hexadecimálisan. A szabad helyet bájtokban kell megadni. Ekkor következő szegmens szitálása előtt megvizsgálja, hogy van-e még annyi szabad hely az adatbáziskönyvtárban, mint amennyi meg van adva, ha igen, akkor szitálja a szegmenst, és lép a következőre, és ha nincs már elég szabad hely, akkor a program megáll.

```
generator$ ./generator.bin ../db start 0x100000001 reserve-space 1000000000
szegmens kezdet: 4 294 967 297
fenntartandó hely: 1 000 000 000 byte
kis szegmensek mérete: 22
elágazások bitjei: 8
felkészülés 1/4
felkészülés 2/4
felkészülés 3/4
felkészülés 4/4
felkészülés vége
szegmens 4 - kezdet 4 294 967 297 - felkészülés 8 162 031 487 ns - szitálás 729 914 370 ns
szegmens 5 - kezdet 5 368 709 121 - felkészülés 418 ns - szitálás 736 412 117 ns
szegmens 6 - kezdet 6 442 450 945 - felkészülés 385 ns - szitálás 750 022 409 ns
összes szitálás 2 216 348 896 ns
```

Mindkét program kimenetében a "start" a szegmens kezdőszáma, a "szegmens" az indexe,  $start = szegmens \cdot 2^{30} + 1$ , a "felkészülés" a szita inicializálási ideje a szitálás előtt, és a 'szitálás' a szegmens szitálásának ideje. Ezek az információk az elmentett szegmensfájlokból is kiolvashatóak.

## Mintaadatbázis karbantartása

A mintaadatbázis a szitatáblák összesített statisztikáit tárolja. A programok az adatbázis könyvtárában kétfajta fájl vesznek figyelembe. A szegmensek szitatáblái a "primes\[0-9a-f]{16}" reguláris kifejezésnek megfelelő nevű fájlok. A fájlnev második fele a szegmens kezdőszáma hexadecimálisan. A fájl a páratlan számok bitvektora mellett tartalmazza a szegmens szitálásának megkezdésére, és a szitálására fordított időt, valamint ellenőrzésképpen a szegmens kezdőszámát is.

A szitatáblák mellett az adatbázis tartalmazhatja szegmensek összesített statisztikáit.

tikáit is, **aggregates** néven. Ez a fájl az eddig összesített szegmensekről a következő információkat tartja nyilván, minden szegmensről külön-külön:

- a szegmens kezdőszáma
- a szegmens szitálására való felkészülés idejét
- a szegmens szitálásának idejét
- az összesítésre fordított időt
- a szegmensfájl utolsó módosítási idejét
- a szegmensbe eső legkisebb, és legnagyobb prímet
- a szegmensbe eső prímek számát,  $12\mathbb{Z} + 11$ ,  $4\mathbb{Z} + 1$ ,  $4\mathbb{Z} + 3$ ,  $6\mathbb{Z} + 1$  alakok szerinti bontásban
- a szegmensben előforduló prímhézagok első előfordulásának helyét, és az előfordulások számát.

Példaként ha a szegmensek mérete 16 szám lenne, és 17-től szitálnánk egy szegmensnyit, akkor a szegmensfájl leírná, hogy a prímek a 17, 19, 23, 29, 31, és az összetett számok a 15, 21, 25, 27. A szegmens összesítése azt tartalmazná, hogy

- a szegmens 17-tel kezdődik
- a legkisebb prím 17, a legnagyobb 31
- a szegmensben 5 darab prím van, ebből
  - 2db  $4\mathbb{Z} + 1$
  - 3db  $4\mathbb{Z} + 3$
  - 2db  $6\mathbb{Z} + 1$
  - 1db  $12\mathbb{Z} + 11$  alakú
- a szegmensben előforduló prímhézagok
  - a 2 (ikerprímek) kétszer fordul elő, először 17-nél
  - a 4 egyszer fordul elő, először 19-nél
  - a 6 egyszer fordul elő, először 23-nál.

## Adatbázis műveletei

Az adatbázison három művelet végezhető:

- Le lehet kérni
  - a szegmensfájlok számát
  - az első és utolsó szegmensfájlt
  - az legkisebb hiányzó szegmensfájlt
  - a szegmensstatisztikák számát
  - az első és utolsó szegmensstatisztikát
  - az legkisebb hiányzó szegmensstatisztikát
  - az összesítést igénylő új szegmensfájlok számát.
- Összesíteni lehet az új szegmensfájlokat.
- Össze lehet fésülni két összesítőfájlt.

Mindhárom művelet elvégezhető a grafikus felületen, és automatizáláshoz parancssorból is. A grafikus felületet a gui szkripttel lehet indítani, a parancssorból a database szkript segítségével lehet elérni a műveleteket.

Ezek a szkriptek meghatározzák, hogy a program melyik könyvtárat használja adatbázisként, ezt nem kell külön megadni. Szükség esetén ez a könyvtár a szkriptek egyszerű módosításával megváltoztatható.

## Adatbázis információk

A adatbázis információkat a grafikus felületen a "DB info" gomb megnyomásával lehet lekérdezni, parancssorból a database info kiadásával.

```
scripts$ ./database info
100.0%
szegmensfájlok: szegmensek száma: 7
szegmensfájlok: első szegmens kezdete: 1
szegmensfájlok: utolsó szegmens kezdete: 6,442,450,945
szegmensfájlok: első hiányzó szegmens kezdete: 7,516,192,769
szegmensfájlok: hiányzó szegmensek száma: 18,446,744,073,709,551,593

összesítések: szegmensek száma: 62,881
összesítések: első szegmens kezdete: 1
összesítések: utolsó szegmens kezdete: 67,516,885,893,121
összesítések: első hiányzó szegmens kezdete: 67,517,959,634,945
összesítések: hiányzó szegmensek száma: 18,446,744,073,709,488,719

új szegmensfájlok: 7

scripts$ ./database info crunch 1024
100.0%
szegmensfájlok: szegmensek száma: 4
szegmensfájlok: első szegmens kezdete: 1
szegmensfájlok: utolsó szegmens kezdete: 3,221,225,473
```

```

szegmensfájlok: első hiányzó szegmens kezdete: 4,294,967,297
szegmensfájlok: hiányzó szegmensek száma: 18,446,744,073,709,551,596

összesítések: szegmensek száma: 140,292
összesítések: első szegmens kezdete: 1
összesítések: utolsó szegmens kezdete: 150,636,314,230,785
összesítések: első hiányzó szegmens kezdete: 150,637,387,972,609
összesítések: hiányzó szegmensek száma: 18,446,744,073,709,411,308

crunch: ../generator/generator.bin ../db start 0x890100000001 segments 0x400

```

A **crunch** utótag és a szitálni kívánt szegmensek számának megadásával az utolsó sorban a következő futtatásra ajánlott parancsot is megjeleníti a program. Ezeknek a parancsoknak a követésével a teljes szitálható tartományt fel lehetne dolgozni. A **crunch-numbers** szkript ezt a feladatot kísérli meg elvégezni.

A **database info** megjeleníti az új szegmensfájlok számát is, ezek azok, amikhez vagy nem tartozik összesített adat, vagy tartozik, de az régebbi szegmensfájl alapján készült, mint ami éppen az adatbáziskönyvtárban található.

2.2. ábra. Adatbázis információk

szegmensfájlok	
szegmensek száma	4
első szegmens kezdete	1
utolsó szegmens kezdete	3,221,225,473
első hiányzó szegmens kezdete	4,294,967,297
hiányzó szegmensek száma	18,446,744,073,709,551,596
összesítések	
szegmensek száma	140,292
első szegmens kezdete	1
utolsó szegmens kezdete	150,636,314,230,785
első hiányzó szegmens kezdete	150,637,387,972,609
hiányzó szegmensek száma	18,446,744,073,709,411,308
Ok	

## Szegmensek összesítése

Az adatbázis könyvtárban található új szegmensfájlokat a "DB összesítés" gomb megnyomásával, vagy a **database reaggregate** paranccsal lehet összesíteni. Minden olyan szegmensfájlról új összesítés készül, amihez még nem tartozott összesítés, vagy a szegmensfájl újabb, mint ami alapján a már meglévő összesítés készült. Az adatbázisban már meglévő, többi szegmenshez tartozó statisztika nem vész el.

```

scripts$ ./database reaggregate
100.0%

```

## Összesítőfájlok összefésülése

Lehetőség van egy másik adatbázisból statisztikák átvételére is. Erre több gépen párhuzamosan szítálva lehet szükség. Az összefésülést a grafikus felületen a "DB import" megnyomása után egy file kiválasztásával lehet indítani. Parancssorból a `database import aggregates fájlnev` parancs kiadásával. A megadott fájlból azokat a szegmenseket veszi át a program, amik vagy nincsenek még meg az adatbázisban, vagy megvannak, de az importált statisztikák utolsó módosítási ideje nagyobb az adatbázisban lévőnél.

```
scripts$ ./database import aggregates ../db/aggregates.large
100.0%
```

## Szitatáblák ellenőrzése

A `generator` program teszteléséhez a grafikus program képes a szegmensfájlok ellenőrzésére. Ehhez a `gui` újra előállítja az ellenőrzött szegmensfájlokat, és azokat összeveti. Az ellenőrzőszegmensek előállításához az egyszerű szegmentált eratoszthenészi szitát, vagy ez erős pszeudoprím tesztet lehet választani.

A pszeudoprím teszt a szegmens minden számáról egyesével dönti el, hogy az prím-e. A pszeudoprím teszt rendkívül lassú, a célja, hogy a szitákhoz képest egy alapvetően más módszer is rendelkezésre álljon az ellenőrzéshez.

A grafikus felületen a "Szegmensfájlok ellenőrzése" gomb megnyomása után lehet kiválasztani a szitálni kívánt szegmenseket, és a referencia előállításának módszerét, majd az "Ellenőrzés" gombbal lehet megkezdeni a tényleges folyamatot.

Az ellenőrzést parancssorból is lehet futtatni a `check-segments` szkripttel. A szkript első paraméterében a referencia módszerét várja, ami lehet `sieve`, vagy `test`, a paraméterek maradéka az ellenőrizendő szegmensek kezdőszáma kell legyen. Ha egy kezdet sincs megadva, akkor a program az adatbázis könyvtárban található összes szegmensfájlt ellenőrzi.

```
scripts$ ./check-segments sieve
100.0%: A szegmensek helyesek.

scripts$ ./check-segments sieve 0x1 0xc0000001
100.0%: A szegmensek helyesek.

scripts$ ./check-segments test 0x1
100.0%: A szegmensek helyesek.
```

Az ellenőrzőszegmens szitával történő előállításához szükséges, hogy az `init` program által előállított négy darab szegmensfájlt tartalmazza az adatbáziskönyvtár. Az 1-től kezdődő szegmens kivételével az ellenőrző szegmens előállításához a szita az inicializálásához a prímszámokat innen olvassa fel.

## Sziták ellenőrzése

Az sebességek összehasonlításához írt sziták eredményét is lehet ellenőrizni. Az összehasonlításhoz a szegmensek ellenőrzésénél használt egyszerű szegmentált szita a referencia.

A program grafikus felületén a "Sziták ellenőrzése" lehetőséget választva ki kell választani az ellenőrizendő szitát, és az ellenőrizni kívánt intervallum kezdő és végző számát. A folyamat az "Ellenőrzés" gomb megnyomásával indítható.

Parancssorból is ellenőrizhetők a sziták a **check-sieve** szkripttel. A szkript 3 paramétere a szita neve, és az intervallum kezdő-, és végszáma. Az ellenőrizhető sziták nevei:

- atkin: Atkin szitája[1]
- cols: Cache Optimalizált Lineáris Szita[2]
- eratosthenes-segmented: Eratoszthenész szitájának szegmentált változata
- bin-heap és bin-heap-inplace: Eratoszthenész szitája bináris kupaccal, az bin-heap-inplace a kupac javítását helyben végzi
- buckets, buckets- $n$ , buckets-simple: Eratoszthenész szitája, edénysorral  
a buckets-simple nem szegmentált, a buckets, és a buckets- $n$  szegmentált. a buckets- $n$ -ben az  $n$  helyére 1-től 8-ig a számrendszer bitjeit lehet megadni, pl. buckets-3 8as számrendszerben számol
- trial-division: próbaosztás

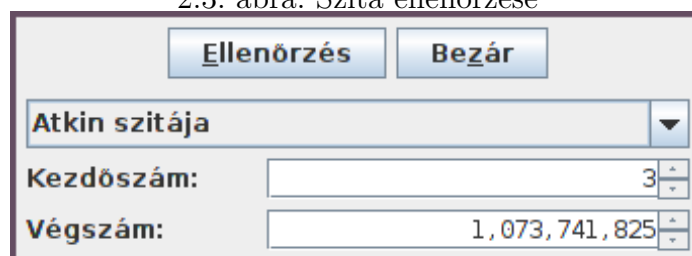
A "Sziták" fejezetben található ezeknek a szitáknak a részletesebb leírása.

A sziták nevei listázhatóak a **list-sieves** szkripttel, a **describe-sieves** szkript rövid leírást is ad.

```
scripts$ ./check-sieve buckets-3 30000000000001 30100000000001
100.0%: A szita helyes.
```

Ha a kezdőszám nagyobb, mint 1, az ellenőrzéshez ilyenkor is szükséges, hogy a négy darab kezdő szegmensfájl tartalmazza az adatbáziskönyvtár.

2.3. ábra. Szita ellenőrzése



## Minta megjelenítése

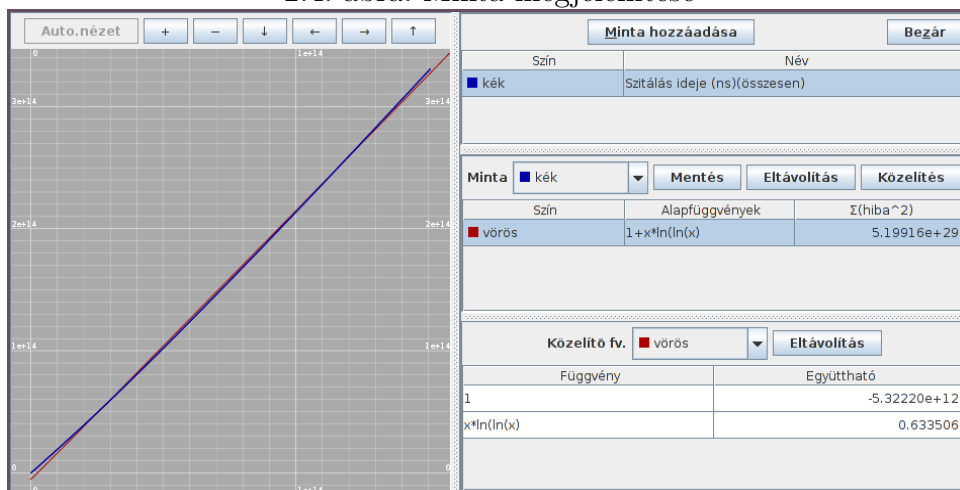
A gui program a grafikus felületén képes mintákat Descartes-féle koordináta rendszerben ábrázolni, a mintákat függvényekkel közelíteni, és ezeket a függvényeket a mintával együtt ábrázolni. A program a közelítések négyzetes hibáját is kiszámítja a mintához képest. A program egyváltozós, egyértékű mintákat kezel, azok között is a természetes számokhoz valós számokat rendelőket. A minta akár külső forrásból is származhat.

A grafikonábrázoló részt a **gui-graph** szkripttel lehet indítani. A gui szkripttel indítva, majd a "Grafikon" gomb megnyomásával is meg lehet nyitni ezt az ablakot.

Az ablak bal oldalán jelennek meg a betöltött minták és közelítő függvények grafikonjai. Minta és függvény hozzáadása után a grafikon megjelenített részét a program úgy választja meg, hogy minden minta összes pontja ebbe a nézetbe essen. A nézet a grafikon feletti gombokkal, vagy egérrel mozgatható, nagyítható, vagy kicsinyíthető. Az "Auto.nézet" gombbal a nézet a minták teljes terjedelmére visszaállítható.

Az ablak jobb oldala a betöltött minták kezelésére szolgál. A felső harmada az összes betöltött mintát mutatja. Az ablak jobb oldalának alsó két harmadában az itt kiválasztott minta részletei látszódnak.

2.4. ábra. Minta megjelenítése



## Minta hozzáadása

A "Minta hozzáadásánál" három adatforrásból lehet választani. A "Fájl betöltésével" CSV fájl olvasható be. A fájl formátuma megengedő, az első oszlopot  $x$ , a második oszlopot  $y$  értékeknek próbálja értelmezni, a fel nem ismert sorokat eldobja. Ha az első sor első oszlopa "BARS", akkor vonaldiagram helyett a mintát oszlopdiagramként fogja ábrázolni.



A "Prímstatisztikák" a **generator** kimeneteléből összesített statisztikák, amiket az adatbáziskönyvtárban keres. A program a szegmensstatisztikák hiánytalan kezdőszeletét veszi csak figyelembe. A statisztikáknak három csoportja van. A prímek száma, és néhány kiemelt alakú prím száma a szegmensek végén. Ide tartozik ugyanezek a helyeken a prímszámtétel becslése, és a becslés abszolút és relatív hibája is. A prímhézagok statisztikái a szegmensek teljes kezdőszeletére vonatkozik, az értékek a kezdőszelet végéig összesített statisztikából származnak. A prímhézagok statisztikái az előfordult hézagok gyakorisága, első előfordulása, jósága, és az legnagyobb prímhézag adott számig. Ezeken kívül a szegmensek szitálásának ideje is betölthető. A szegmensenkénti idő minden szegmens végéhez annak a szegmensnek a szitálásának idejét rendeli. Az összesített idő a szegmensek végéig az összes megelőző szegmens szitálásának idejét adja össze.

A minták harmadik forrása a programba épített sziták futási idejének mérése. Ehhez meg kell adni a szitát, a szitált intervallum elejét és végét, a szita belső szegmensméretét, a generálandó minták számát, és a mérések számát, a mértéket, és hogy az eredményt összesítse-e. A sziták listája ugyanaz, mint a sziták ellenőrzésénél. A mintákat a program a szitált intervallumon egyenletesen osztja el a megadott méretű szegmensek határainál. A teljes szitálást annyiszor végzi el, amennyi a mérések számában meg volt adva, és az eredményt átlagolja. Összesítést kiválasztva a minta értéke a szitált intervallum elejétől az adott számig szitálás teljes ideje, különben az előző minta helyétől. A mérték szabályozza, hogy a tényleges eltelt időt mérje a program nanoszekundumokban, vagy az elvégzett műveletek számát.

## Minta közelítése

Az ablak jobb oldalának középső harmadában a kiválasztott minta részletei látszódnak. Itt változtatható meg a minta ábrázolásához használt szín, távolítható el a minta, vagy menthető CSV fájlba.

A táblázat a minta közelítéseit mutatja, itt olvashatóak le a közelítés komponensfüggvényei, és a közelítés hibája. A "Közelítés" gombbal lehet új közelítést hozzáadni az ábrázoláshoz. A közelítéshez a lineáris legkisebb négyzetek módszerét használja a program, ehhez a lineáris kombináció elemi függvényeit meg kell adni. A függvények kiválaszthatóak az előre megadott listából, vagy JavaScript nyelven is megadhatóak. A közelítéshez csak olyan függvények használhatóak, amik a minta minden pontjában értelmezve vannak.

2.5. ábra. A közelítés elemi függvényeinek megadása

JavaScript függvényhez meg kell adni a függvény nevét és forráskódját. A forráskódban a függvény argumentumára az  $x$  változóval lehet hivatkozni, és a szkript utolsó kifejezésének értéke lesz a függvény értéke  $x$ -ben. Pl. az  $x \mapsto \frac{\ln x}{x}$  hozzárendelés forráskódja:

```
Math.log(x)/x
```

A szkriptben használható változó, és a szokásos vezérlőutasítások. Pl. az iterált logaritmus egy megadása:

```
var y=0;
while (1<x) {
  x=Math.log(x);
  ++y;
}
y
```

Az így megadott függvények visszatérési értéke mindig `double` típusú kell legyen, és a függvény azokban a pontokban értelmezett, ahol ez az érték véges.

Egy kiválasztott közelítés részletei az ablak jobb alsó sarkában látszódnak, itt megváltoztatható a közelítéshez rendelt szín, és eltávolítható a közelítés. A táblázat a közelítéshez kiválasztott függvények együtthatóit mutatja a lineáris kombinációban.

A legkisebb négyzetek módszere röviden leírható. Ha az  $n$  elemű minta

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) (x_i \in \mathbb{N}, y_i \in \mathbb{R})$$

és a közelítéshez kiválasztott  $m$  függvény

$$f_1, f_2, \dots, f_m (f_i : \mathbb{R} \mapsto \mathbb{R})$$

akkor a legkisebb négyzetek módszere keresi azokat a

$$c_1, c_2, \dots, c_m (c_i \in \mathbb{R})$$

skalárokat, hogy az

$$f(x) = \sum_{i=1}^m c_i f_i(x) (x \in \mathbb{R})$$

közelítő függvény és a minta eltérése minimális legyen a

$$\sum_{i=1}^n (f(x_i) - y_i)^2$$

négyzetes eltérés összeg szerint.

A program ezt az eltérés összeget jeleníti meg a közelítés hibájaként, valamint a  $c_i$  skalárok a közelítés elemi függvényeinek együtthatói.

## Sziták mérése szkriptekkel

A sziták futási idejének mintáit parancssorból is elő lehet állítani. Ilyenkor az eredményt, megjelenítés helyett, a megadott fájlba írja a `gui` program, ami később a grafikus felületen betölthető. A méréseket a `measure-sieve` szkripttel lehet elvégezni, paraméterként meg kell adni ebben a sorrendben:

- a szita nevét, ezek ugyan azok lehetnek, mint a `check-sieve` szkriptnél
- a számot, ahol a szitálás kezdődik, ez páratlan szám kell legyen
- a szitálás végét, ez páratlan szám kell legyen
- a szegmensek méretét
- a mérések számát
- a minták számát
- hogy időt (`nanosecs`), vagy műveleteket (`operations`) mérjen
- összesítse az időket (`sum`), vagy szegmensenként külön számolja (`segment`)
- a fájlt, ahova az eredményt mentse CSV-ben

A számokat meg lehet adni decimálisan, és hexadecimálisan is "0x" előtaggal. Ha a kezdőszám nagyobb, mint 1, akkor az adatbáziskönyvtárnak tartalmaznia kell az init program generálta első négy szegmenst.

Lehetőség van a mérték és az összesítés mind a négy kombinációját egy méréssel előállítani, ehhez mind a négy kimeneti fájl meg kell adni külön.

```
scripts$ ./measure-sieve buckets-3 1 0x10000001 0x10000 3 1000 nanosecs sum out1.csv
100.0%
scripts$ ./measure-sieve buckets-3 1 0x10000001 0x10000 3 1000 operations segment out2.csv
100.0%
scripts$ ./measure-sieve buckets-3 1 0x10000001 0x10000 3 1000
seg-ns.csv seg-ops.csv sum-ns.csv sum-ops.csv
100.0%
```

A mérések megkönnyítéséhez a **measure-sieves** szkript az összes szitát megméri, és az eredményt a **samples** könyvtárba menti. A **samples** könyvtárban a mérési eredmények 4 külön könyvtárban szegmensenként/összesítve, és idő/művelet szerinti bontásban vannak. Ezekben a könyvtárakban 2 fajta mérés eredményei vannak. Az atkin könyvtárban Atkin szitájának sebessége van megmérve különböző szegmensméretek esetén. Azokban a könyvtárakban, amiknek a neve **speed**-del kezdődik, az összes szita sebességének mérése található,  $2^{20}$  szegmensmérettel. A könyvtárak nevében a szám a szitalás kezdetét adja meg, a **speed $n$**  nevű könyvtárban a mérések  $2^n + 1$ -től kezdődnek. A lassabb szitákat csak kisebb nagyságrendekre méri meg a szkript, ezek futási ideje hamar túl nagyra válik.

A **measure-generator** szkript a **generator** program sebességét méri meg. A mérés 3 paramétert variál, a szitalás kezdetének nagyságrendjét, a szegmensméretet, és az edénysor számrendszerét. Az eredményt a **samples/measure-generator.csv** fájlba menti. Ez nem használható a program mintamegjelenítésével, mert az egy változós mintákkal dolgozik. Ennek a mérésnek a célja az, hogy a **generator** program paramétereit egy konkrét számítógéphez lehessen igazítani. A szkript 16 egymás utáni szegmensfájlt szital,

- a szitalás kezdetét  $2^{33}$ -tól  $2^{63}$ -ig változtatja, a kitevőt hármassal növelve
- a szegmensek méretét  $2^{16}$ -tól  $2^{24}$ -ig, a kitevőt egyesével növelve
- és a számrendszert  $2^1$ -től  $2^8$ -ig, szintén a kitevőt egyesével növelve.

## 3. fejezet

# Fejlesztői dokumentáció

### A feladat leírása

A program feladata, hogy lehetővé tegye prímszíták futási idejének összehasonlítását. A prímszíták az egész számok egy intervallumán minden számról eldöntik, hogy a szám prím, vagy összetett. A program keretet ad, hogy azonos környezetben lehessen szíták futási idejét mérni, és egyszerű, egymáshoz hasonlóan optimalizált szítákat implementál, hogy a futási időket a fő algoritmusok határozzák meg. A futási időt eltelt időben, és összeszámolt műveletekben is méri.

A szítákat szegmentáltan valósítja meg, a szegmensek mérete, és a szítálandó intervallum kezdete és vége szabályozható. A megvalósított szíták:

- Atkin szítája
- Eratoszthenész szítája
  - egyszerű tömbbel
  - bináris kupaccal
  - edényekkel
  - monoton edénysorral

A programnak grafikusán ábrázolja a futási időket, és lehetőséget ad ezek közelítésére függvényekkel. A közelítéshez a legkisebb négyzetek módszerét használja, és a közelítés elemi függvényei megválaszthatóak. A program megjeleníti a közelítések négyzetes eltérését is a mintához képest. Elemi függvényt meg lehet adni JavaScript nyelven is.

A grafikus ábrázolás képes külső forrásokból származó minták megjelenítésére és közelítésére is.

A minták és függvények ábrázolása interaktív, az éppen megjelenített intervallum elhelyezkedése és mérete megválasztható. Egyszerre több mintha is megjeleníthető, és minden mintához egyszerre több közelítő függvény is tartozhat.

A program a prímszámok statisztikáinak megjelenítéséhez elő tudja állítani a következő statisztikákat nagyobb intervallumokon is. Ehhez ad egy szitát, ami hatékonyan képes  $2^{64}$ -ig szitálni, és ennek az eredményét összesíteni. Az előállítható statisztikák adott  $n$ -ig szitálva:

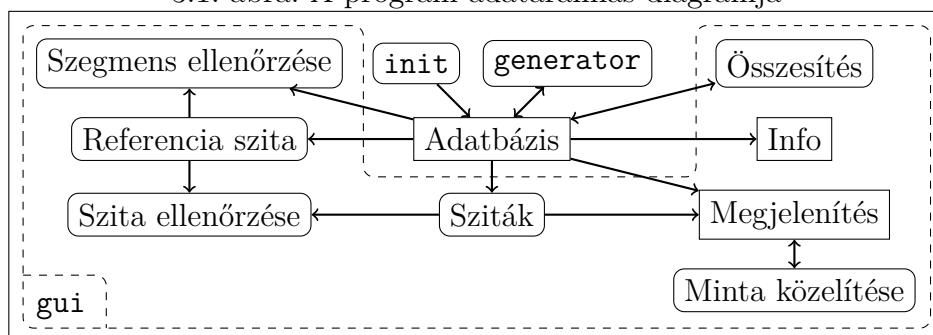
- a prímek száma
- a  $4\mathbb{Z} + 1$ ,  $4\mathbb{Z} + 3$ ,  $6\mathbb{Z} + 1$ ,  $12\mathbb{Z} + 11$  alakú prímek száma
- ikerprímek számát
- az előforduló prímhézagok első előfordulásának helye
- a prímhézagok előfordulásának száma
- a szitálásra fordított idő

A prímek számát, és a szitálás idejét az intervallum belsejében több ponton is elkészíti.

A sziták sebességének mérését, és a statisztikák előállítását parancssorból is el tudja végezni, hogy szkriptekkel ez automatizálható legyen. A statisztikák előállításának folyamata bármikor adatvesztés nélkül leállítható, és később folytatható.

## A program komponensei

3.1. ábra. A program adatáramlás diagramja



A program az adatbáziskönyvtárban tárolja az átszitált szegmensek tábláit, és az ezekből készült statisztikákat. A szegmensfájlokban egy  $2^{30}$  hosszú intervallum minden páratlan számához hozzá van rendelve, hogy az az szám összetett vagy prím, és minden szegmens kezdő száma kongruens 1 (mod  $2^{30}$ ).

Az első négy szegmenst az `init` program készíti el, így ezek  $2^{32}$ -ig tartalmazzák a prímszámokat. A program többi szitája ezek alapján inicializálja magát, ha 3-nál nagyobb számtól kell a szitálást elkezdni. Ez a négy fájl rövid idő alatt újragenerálható, de ezeket érdemes a futások között megtartani.

A `generator` egy optimalizált szita,  $2^{32}$ -tól  $2^{64} - 2^{34}$ -ig képes szitálni, mindig egész szegmenseket, és az eredményt az adatbáziskönyvtárba menti. Ezekre a szegmensfájlokra összesítés és ellenőrzés után többet nincs már szükség. A szitálást tetszőleges szegmenstől lehet kezdeni a megengedett tartományon belül, nem szükséges sorban végigmenni a számokon. Az utolsó  $2^{34}$  szám kihagyása egyszerű optimalizáció, így a programban az éppen szitált szám változója sosem csordul túl.

Az `init` és a `generator` csak konzol módban futtatható. A program többi része a `gui` nevű Java program része, de az automatizáláshoz az adatbázissal kapcsolatos műveletei parancssorból is elérhetőek, ezek az adatbázis információ lekérése, az új szegmensfájlok ellenőrzése, és összesítése, és az összesítőfájlok összefésülése. Az időigényessége miatt a futási idő összehasonlításához implementált sziták ellenőrzése is futtatható grafikus felületen, és parancssorból is.

Az adatbázis információ kilistázza a szegmensfájlok számát, az összesített szegmensfájlok számát, ezen szegmensek intervallumának kezdetét és végét, a még nem összesített szegmensfájlok számát. Ezek alapján eldönthető, szitálást vagy az összesítést hogyan kell folytatni, hogy a teljes  $[1, 2^{64} - 2^{34}[$  intervallum statisztikai elkészüljenek.

A szegmensfájlok ellenőrizhetőek, ehhez egy, a generator-tól különböző eratoszthenészi szita, vagy az erős pszeudo-prímteszt[5][6] előállítja a szegmens ellenőrzött

részét, és a program a kettőt összehasonlítja. A szita lényegesen lassabb, mint a **generator**, és a pszeudoprímteszt időigényessége miatt intervallumok szitálásra nem alkalmas. A szegmensfájlok ellenőrzése alapvetően a **generator** helyességének tesztelésére való, minden összesített szegmens ellenőrzése többszörözné a szitáláshoz szükséges időt.

Az összesítő az adatbáziskönyvtárban található szegmensfájlok statisztikáit készíti el, és ezeket a szegmensstatisztikákat szintén az adatbázisban tárolja. A szegmensstatisztikák alapján a végső statisztikákat közvetlenül a megjelenítés előtt készíti el az összesítő. Az összesítéshez sem szükséges, hogy a szegmensek sorban kerüljenek összesítésre, így akár több gépen párhuzamosan is lehetne szitálni, összeíteni, majd az eredményt összefésülni.

A program része több, különböző szita algoritmus megvalósítása. Ezeknek az eredménye ellenőrizhető a szegmensfájlok ellenőrzéséhez használt szitával, szintén a sziták helyességének teszteléséhez. A feladat része, hogy ezeknek a szitáknak a futási idejét össze lehessen hasonlítani, ezért mindegyik szita néhány jól meghatározott ötlet egyszerű megvalósítása optimalizálások nélkül. Mindegyik szita szegmentált, a szitálást fix hosszúságú intervallumokon végzi. A sziták bármelyik szegmenstől képesek kezdeni a szitálást, és egy szegmens szitálása után további előkészület nélkül képesek a következő szegmessel folytatni a szitálást.

A minták közelítése alapfüggvények lineáris kombinációjával történik, a legkisebb négyzetek módszerét használva.

A program a mintákat és közelítő függvényeket Descartes-féle koordináta rendszerben jeleníti meg, ehhez a mintákat és a függvényeket mintavételezi az  $x$  tengely kiválasztott szakaszának pixelenkénti felosztása szerint.

## Az adatbázis

A program adatbázisa egy egyszerű könyvtár, amin keresztül a különböző programrészek adatokat cserélnek. Ezek az adatok mindig egy-egy teljes bináris fájljt jelentenek. Az adatbázis két fajta fájljt tárol, a szitált szegmenseket, azaz a szitatábla bitvektorait, és a szegmensek összesítéseit. Mindkettő a szegmens első számával van azonosítva, a szegmens hossza mindig  $2^{30}$  szám, és az  $n \in \mathbb{N}$  megengedett kezdő szám, ha  $n < 2^{64} - 2^{34}$  és  $n \equiv 1 \pmod{2^{30}}$ . Minden szegmens egy külön fájl az adatbázis könyvtárában, és az összes szegmensstatisztika egy fájlban van.

Az **init** program az első 4 szegmensfájlt írja. A **generator** az első 4 szegmensfájlt olvassa, és az összes többi szegmensfájlt írja. A **gui** az összes szegmensfájlt olvassa, és egyiket se írja, és szegmensstatisztika fájljt csak a **gui** használja.

Egy szegmensfájlok nevei **primes**. előtaggal kel kezdődjön, és a pontosan 16 hexadecimális számjegy kell kövesse, ami a szegmens által leírt első szám. A fájl bináris



fájl, a legkisebb helyiértékű bájt elöl, azaz little endian sorrendben. A tartalma:

- a szegmens elejével kezdve  $2^{29}$  egymást követő páratlan szám bitvektora.  
Ez  $2^{26}$  byte, legkisebb helyiértékű byte elöl sorrendben, minden bit egy páratlan szám összetettségét írja le. Ha a bit 1, akkor a szám összetett, ha 0, akkor a szám prím. Ha  $s$  a szegmens intervallumának kezdete, akkor az 0. bájt 0. bitje  $s-t$  írja le, a 0. bájt 1. bitje  $s + 2-t$ , a  $k$ . bájt  $l$ . bitje  $s + 16k + 2l-t$ .
- a szegmens kezdete, 8 bájt, egész
- a szegmens szitálására való felkészülés ideje, 8 bájt, egész, nanoszekundumokban
- a szegmens szitálásának ideje, 8 bájt, egész, nanoszekundumokban

Ahhoz, hogy az `init` és a `generator` program bármikor leállítható legyen szegmensfájl írása közben, a szegmenst először egy ideiglenes fájlba írja, és ha ez hiba nélkül megtörtént, akkor az ideiglenes fájlt átnevezi a megfelelő névre. Mivel az ideiglenes fájl ugyanabba a könyvtárba kerül, mint ahova a szegmensfájl is, így feltételezhető, hogy helyi meghajtón az átnevezés már atomi művelet.

Ez az eljárás az átnevezés előtt nem győződik meg arról, hogy az ideiglenes fájl minden része kiíródott a háttértárra, így ez csak a program leállíthatóságát garantálja, a számítógép váratlan leállása esetén érdemes azokat a szegmensfájlokat letörölni, amikről feltételezhető, hogy nem volt ideje a számítógépnek minden részét kiírnia a tényleges tárolóra.

A szegmensstatisztikák az `aggregates` fájlban vannak. A szegmensek a fájlban a kezdőszámok sorrendjében követik egymást, egy szegmensnek legfeljebb egy statisztikája lehet. A fájl bináris, a legnagyobb helyiértékű bájt elöl, azaz big endian sorrendű. A fájl fejléce:

- formátum azonosító: 0xd1d1b0b0, 4 bájt
- verziószám: 1, 4 bájt.

Ezután minden statisztika egy változó méretű blokkban van tárolva, a fájl végéig. Minden block két részből áll, az első 4 bájt a második rész mérete bájtokban. Ez a második rész a szegmensstatisztika. A statisztika gzip-pel tömörített, kitömörítve bináris adat, a legnagyobb helyiértékű bájt elöl, aminek a formátuma:

- a szegmens összesítésének futási ideje, 8 bájt, nanoszekundumokban
- a szegmens szitálásának inicializálásának ideje, 8 bájt, nanoszekundumokban
- az összesített szegmensfájl utolsó módosításának ideje, 8 bájt, milliszekundumokban, 1970.1.1. 00:00:00-tól

- a szegmensben található legnagyobb prím, 8 bájt
- a szegmensben található legkisebb prím, 8 bájt
- a  $12\mathbb{Z} + 11$  alakú prímek száma a szegmensben, 8 bájt
- a  $4\mathbb{Z} + 1$  alakú prímek száma a szegmensben, 8 bájt
- a  $4\mathbb{Z} + 3$  alakú prímek száma a szegmensben, 8 bájt
- a  $6\mathbb{Z} + 1$  alakú prímek száma a szegmensben, 8 bájt
- a szegmensben előforduló prímhézagok számát, 4 bájt. ahány prímhézag előfordult, annyszor ismétlődik a következő három mező
  - a prímhézag nagysága, 8 bájt
  - a prímhézag előfordulásainak száma, 8 bájt
  - a prímhézag első előfordulása, 8 bájt
- a szegmens sorszáma, 8 bájt
- a szegmens vége, 8 bájt
- a szegmens kezdete, 8 bájt
- a szegmens szítálására fordított idő, 8 bájt, nanoszekundumokban.

Az összesítésfájl írása a szegmensfájl írásához hasonló. Egy ideiglenes fájlba írása után a program megvárja, hogy a fájl tartalma a háttértárolóra kerüljön, majd az ideiglenes fájl átnevezi a megfelelő névre. Az ideiglenes fájl kiírásának megvárásával a statisztikafájl váratlan számítógép leállás esetén is visszaállítható.

A fájl formátuma és az új fájl kiírásának algoritmusai lehetővé teszi, hogy a statisztikafájlt egyszerre lehessen sorban felolvasni, és az új eredményt sorban kiírni, úgy, hogy egyszerre csak egy statisztikát kelljen a memóriában tartani.

Az adatbázis nincs védve a párhuzamos írásoktól, és az összesítőfájl egyszerű sorban feldolgozhatósága feltételezi, hogy egy fájl kevés szegmensstatisztikát fog tartalmazni, legfeljebb kb.  $10^6$  szegmensét.

## A megvalósítás

A megoldás 3 programból áll, az `init` és a `generator` C nyelven készült el, az összes többi részfeladat a `gui` nevű Java nyelven írt programban lett megoldva.

## Init és generator

Az `init` és a `generator` forráskódja a `tarball` generator könyvtárában van. A két program egymáshoz nagyon hasonló feladatot lát el, hasonló megoldásokkal. Mindkettő  $2^{30}$  hosszú intervallumokat szítál, és az eredményt szegmensfájlokba írja. Ehhez mindkettő Eratoszthenész szitáját használja. Mindkét program ezt az egyetlen feladatot oldja meg, és szkriptekkel automatizált, kötegelt futtatásra van tervezve.

### Közös rész

A két program közös kódja a `common.h` fájl. Ez a fájl tartalmazza a konstansokat, és az eljárásokat, amik megkönnyítik a POSIX hívások használatát. A legtöbb ilyen könnyítés a hibák kezelésének elfedése, ami a kötegelt futtatáshoz illeszkedően a hiba kijelzése utáni azonnali programleállás. Az I/O műveleteket könnyítik meg a `readFully()/writeFully()` eljárások.

A sziták belső, kis szegmensmérete a `SEGMENT_SMALL_SIZE_BITS_LOG2` konstanssal szabályozható, és a `measure-generator` szkript kimenete alapján egy adott géphez igazítható.

A szegmensfájlok olvasására és írására a `readSegment()` és `writeSegment()` függvények szolgálnak.

### Init

Az `init` program előállítja az első 4 szegmens fájlt, azaz  $2^{32}$ -ig az összes prímet megkeresi. Ezzel a többi részprogram már a  $2^{32} - 2^{64}$  tartomány tetszőleges részén tud szitálni, anélkül, hogy a szitált szegmenseket a szitálás után végig kéne nézze új prímekért.

A szita egy egyszerű szegmentált eratoszthenészi szita. A  $2^{30}$  hosszú szegmenseket több kisebb, egyenlő szegmensre osztja, és ezeket a kis szegmenseket egymás után szitálja a prímek listáján végigmenve. A prímek mellett a következő szitálás helyét is eltárolja, így nem kell minden szegmens szitálásának elején minden prímmel osztással meghatározni, hogy melyik a legkisebb szám a szegmensben, amit az oszt. Így egy kis szegmensben egy prímmel a következő algoritmus fut le. Ez az algoritmus visszatérő minta a szegmentált szitákban.

```
1: p: a prímmel
2: q: a következő szám, amit p oszt
3: e: a szegmens vége
4: while q < e do
5:     MEGJELÖL(q)
6:     q = q + 2p
```

## 7: end while

A prímek listáját elég  $2^{16}$ -ig előállítani, ezt a program próbaosztással végzi el.

### Generator

A **generator** fő részei hasonlóak az **init**-hez. Indulása után felolvassa az összes prímet az első 4 szegmensfájlból, és mindegyik prímmel osztással meghatározza, hogy melyik a legkisebb szám, amit oszt, és legalább akkora, mint legelső szítálandó szegmens kezdete. A prímek, és a hozzájuk tartozó következő szítálandó pozíciót tároló adatstruktúra szítálás közben nem érzékeny azokra az elemekre, amik az éppen szítált szegmensnél nagyobb helyeken szítálnak, így a **generator** az összes prímet eltárolja  $2^{32}$ -ig, nem csak a szítálandó tartomány végének négyzetgyökéig, és a prímek első többszörösét is a prím négyzetétől szítálja.

A prímek listájának inicializálása után a nagy szegmenseket az **init**-hez hasonlóan kisebbekre osztja, és ezeket sorban szítálja a prímek listája alapján.

A prímek listája a **generator**-ban több külön adatszerkezet összessége. A prímeket a szita inicializálásánál 3 részre osztja a nagyságuk szerint. A 64-nél kisebb prímeket 64-bit széles bitmintákban tárolja, és ezekkel a mintákkal 64-bitesével szítálja végig a kis szegmenseket a bitenkénti vagy művelettel. A 3 és a 11, valamint az 5 és a 7 szorzata kisebb, mint 64, ezeket a párokat egyetlen 64 bites mintába fejt ki, ezeknek a mintáknak a periódusa 33, és 35. A többi prím mintájának periódusa a legnagyobb olyan szám, ami 64-nél kisebb, és a prím osztja.

A prímek, amik 64-nél nagyobbak, de a kis szegmensek hosszánál kisebbek, az az minden kis szegmensben szítálnak, az **init**-hez hasonlóan egy egyszerű tömbbe kerülnek, és ugyanúgy szítál ezekkel a program, mint az **init**-ben.

A kis szegmensek hosszánál nagyobb prímek egy kis szegmensben legfeljebb egyszer szítálnak, és van olyan szegmens, ahol egyszer sem. A **generator** ezeket a prímeket egy prioritásos sorban tárolja, amiben az elemek sorrendjét a következő szegmens száma határozza meg, amiben az a prím szítálni fog. A szegmensben belüli sorrendet a sor nem veszi figyelembe. A kis szegmensekben legfeljebb egyszer szítáló prímeknek nem csak a páros többszöröseit ugorja át a szita, hanem a 3-mal és 5-tel osztható többszöröseit is kihagyja. Ezeket a többszörösöket a bitmintával szítáló kis prímek jelölik meg.

Az edénysor algoritmusának részletes leírására egy későbbi fejezetben kerül sor. Az ott leírt távolság függvényt implementálja a **bucketIndex()** függvény a forráskódban. A számrendszert a **BUCKET\_BITS** konstans szabályozza. A sorhoz tartozó edények listája a **buckets** tömb, ennek minden eleme egy láncolt lista, ahol a lista-elemek fix méretű edények.

A program futása alatt a sor elemeinek száma nem változik, de az elemek az

edények között mozognak. A sor legkisebb elemeinek, azaz az aktuális szegmensben szitáló prímek megkereséséhez a lista egyik edényének minden elemét megvizsgálja. Ha egy elem az aktuális szegmensben nem szitál, akkor az elemet a sorba visszahelyezi, a távolság függvény szerint már közelebbi edénybe. Ha az elem szitál az aktuális szegmensben, akkor elvégzi a szitálást, és az elemet visszateszi a sorba, azon pozíció szerint, ahol a prím legközelebb szitálni fog.

A 2, 3, és 5 többszöröseinek hatékony kihagyásához a prioritásos sor elemeiben a pozíció mellett el van tárolva, hogy az a pozíció melyik modulo 30 maradékosztály eleme. A nyolc szóba jöhető lehetőség közül a `PrimePosition.prime` mező felső 3 bitje választ. Ezt a 3 bit információt biztonsággal lehet itt tárolni, a 64 bit széles változóból a feladat határai miatt mindig csak az alsó 32 tartalmaz értékes számjegyet. Ez a 2-3-5 kerék alapú gyorsítás csak a prioritásos sorban tárolt prímekre van alkalmazva.

## Gui

A program Javában írt része tartalmazza az összes többi funkció megvalósítását, a szegmensfájlok feldolgozását, a különböző szitákat, és a minták megjelenítését. Ezeknek a funkcióknak a többsége nem interaktív, a felhasználó a paraméterek megadása után a végrehajtást már nem tudja befolyásolni.

A forráskód ennek megfelelően három egymástól jól elkülöníthető részre osztható. A legalsó szint a program adatszerkezeteit, algoritmusait, és fájlformátumait írja le. Erre építenek a felhasználói műveletek megvalósításai. Mindegyik művelethez tartozik egy eljárásként megvalósított belépési pont, a programkód legfelső szintje ezeket teszi a felhasználó számára meghívhatóvá, parancssorból és grafikusán is.

A grafikonmegjelenítés az egyetlen ténylegesen interaktív feladat, ennek megoldása a modell-nézet-vezérlő felosztást alkalmazza.

### gui.util

A csomag néhány közismert algoritmust és adatszerkezetet tartalmaz, valamint az ezekhez szükséges funkcionális interfészek definícióit. A legtöbbnek létezik könnyen használható sztenderd könyvtári megvalósítása, de ezek referenciatípusokat használnak. A primitív és referencia típusú értékek közötti rengeteg konverzió három problémát okoz. Feleslegesen növeli a futási időt, miközben egyetlen számítógépen szitálva eleve  $2^{64}$ -ig reménytelenül hosszú feladat. Ezen túl megnöveli a program memóriaigényét.  $2^{32}$ -ig kb.  $2 \cdot 10^8$  darab prím van, ha egy szita csak a prím-pozíció párokat egy egyszerű tömbben tárolná, prímenként 4 bájton, pozíciónként 8 bájton, akkor ez a tömb kb. 2,4Gb memóriát foglalna el. A program a primitív típusokon alapuló konténerekkel képes 3,5Gb memória használatával futni. A harmadik

probléma az, hogy primitív érték objektummá alakítása szinte mindig új objektum allokálásával jár, ami legtöbbször egy objektum elérhetetlenségéhez is vezet. A szemetgyűjtő ezzel járó folyamatos háttér munkája nem csak feleslegesen növeli a futási időt, hanem kiszámíthatatlanságával megnöveli a sziták mért futási idejének zaját.

A `PrimitiveList` osztály és leszármazottai az `ArrayList` mintájára szükség szerint növekvő vektorok. Konkrét megvalósítások a `double`, `int`, és `long` primitív típusokra vannak.

A `BinarySearch` és a `QuickSort` a bináris keresés és a gyorsrendezés algoritmusát valósítják meg absztrakt módon elért tömb felett. Az elemeket közvetlenül sosem érik el, az összehasonlításokat és a cseréket interfészeken keresztül végzik index alapján.

A `BinaryMinHeap` egy bináris kupac, mint absztrakt konténer osztály, az elemek típusától függő műveleteket absztrakt metódusként határozza meg. A műveletek felteszik, hogy az elemek indexelhetőek, mint egy tömbben. A kupac egy elsőbbségi sor, a sor eleje az absztrakt rendezés szerinti legkisebb elem.

## **gui.math**

Ebben a csomagban az osztályok két részre oszthatóak. Az `UnsignedLong` a Java `long` típus előjel nélküli műveleteit valósítja meg. A Java a 8-as verziótól támogatja a primitív egész típusok előjel nélküli összehasonlítását és osztását, ez az osztály ezekre építve ad néhány segédeljárást. A pszeudoprím-teszthez a maradékos hatványozás, Java támogatás hiányában, a kitevő bitjei alapján az ismételt szorzást és négyzetre emelést használja, míg a szorzás ehhez hasonlóan összeadást és kétszerezést. Ezeknek az algoritmusoknak a futási ideje egy-egy szám esetén elfogadható, de intervallumok szitálására nem alkalmas.

A csomag többi osztálya a legkisebb négyzetek módszerének implementációja, és annak segédosztályai. A `Sum double` típusú értékek véges összegeit számolja ki. A `double` véges pontosságának következménye, hogy nem minden szám reprezentálható `double` értéként, és többtagú összeg eredménye függhet a tagok összeadásának sorrendjétől, azaz az összeadások sorrendje befolyásolhatja az eredmény hibáját.

A `Sum` három stratégiát ad az összeadások sorrendjéből származó hiba minimalizálására. A program használata közben a felhasználó ezek közül nem tud választani, a tesztek alapján legstabilabbnak ítélt elsőbbségi soron alapuló összegzést használja a közelítés. A legegyszerűbb `Sum.Simple` egyetlen változót használva a tagokat felsorolásuk sorrendjében adja össze.

A bináris kupacot használó `Sum.Priority` a tagokat az elsőbbségi sorban gyűjti, majd a végső összeg kiszámításához a sorból, amíg lehet, ismételten kiveszi a két legkisebb abszolút értékű elemet, azokat összeadja, és az eredményt a sorba visszahelyezi. A végeredmény a sorban maradó érték lesz.

A `Sum.Array` nagyságrend szerinti részösszegeket tart nyilván, új tagot a nagyságrendjéhez tartozó részösszeghez adja hozzá. A nagyságrend a `double` érték IEEE 754 szabvány szerinti reprezentációjának kitevője. Ebben a reprezentációban a kitevő 11 bit hosszú, a `Sum.Array` így mindig 2048 részösszeget tárol.

A `RealFunction` interfész valóshoz valóst rendelő függvények reprezentációi. Egy függvény a helyettesítési értékein kívül az értelmezési tartományáról is információt kell adjon. A grafikonmegjelenítő az  $x$ -tengely egy pixel által lefedett intervallumán akkor jelenít meg értéket, ha az az intervallum az értelmezési tartomány része. A `Functions` osztály a beépített elemi függvényeket tartalmazza. JavaScript-ben függvény a `CustomFunction`-nel adható meg. A `LinearCombinationFunction` már létező függvények lineáris kombinációja. A mintaközelítés eredménye ilyen típusú.

A `Matrix` osztályban vannak megadva a valós mátrixok műveletei, mindegyik kétdimenziós `double` tömbökre. Az implementált műveletek a legkisebb négyzetek módszeréhez szükségesekre korlátozódnak. A Gauss-eliminációnál kiválasztható, hogy részleges sorcserére, vagy teljes sor és oszlopcserére kerüljön sor. A tesztek alapján a teljes csere lényegesen csökkenti a számítás hibáját, a felhasználó részleges sorcserét nem tud választani. A `Matrix` osztály Householder-féle QR-felbontást is el tudja készíteni, de a tesztek alapján a felhasználó ezt nem tudja kiválasztani lineáris egyenletrendszerek megoldására.

A `LeastSquares` osztályban lett megvalósítva a legkisebb négyzetek módszere. A csomag eddig felsorolt osztályaival itt lehet megadni a közelítéshez az elemi függvényeket, a pontosságot szabályozó paramétereket, és a közelítendő mintát. A közelítés hibáját is ki lehet ezzel az osztállyal számolni.

## gui.io

A `gui.io` csomagban az adatbázis kezelését leíró osztályok vannak. Az adatbázis funkciók a `Database` egy objektumán keresztül érhetőek el, ez a `common.h`-val párhuzamosan definiálja a szegmensfájlok neveit. A felhasználói műveletek közül az adatbázis információk lekérését, a szegmensfájlok összesítését, és összesítés importálását `Database` osztály `info()`, `reaggregate()`, és `importAggreagtes()` metódusai valósítják meg. A sziták az inicializálásukkor a `Database.largePrimes()` metódust használva olvassák be az első 4 szegmensfájlból a prímekeket.

A `Segment` osztály példányai szegmensfájlok a memóriában. Ezek tényleges fájlból olvasott adatokat is tartalmazhatnak, vagy szegmensek és sziták ellenőrzésekor a Java program is generálja. Ezzel az osztállyal szegmensfájlt csak olvasni lehet, írni nem.

Egy szegmens összesítését a `Segment.aggregate()` függvény végzi. Az eredményül kapott `Aggregate` példány az adatbázis formátumánál leírt szegmensstatistika tükre. Az összesítőfájlt az `AggregatesReader` és `AggregatesWriter` osztállyal le-

het olvasni és írni, mindkét esetben a szegmensek szigorúan növekedő kezdőszámú sorrendben dolgozhatóak fel. A írás és az olvasás **AggregateBlock** példányokkal dolgozik, ezek az objektumok a statisztikához való hozzáférés mellett a betömörített bináris adatot jegyzi meg, hogy a statisztika változatlan kiírása esetén ki lehessen hagyni az időigényes betömörítést.

A megjelenítésre szánt mintákat az **Aggregates** osztály készíti el a szegmensenkénti statisztikák alapján. A szegmenseket az összesítésfájl sorrendjében dolgozza fel, az első hiányzó szegmensig. A megjelenített statisztikák legtöbbje egyszerűen összesíthető a szegmensenkénti adatokból. A prímhézagok esetében a szegmenseken átívelő hézagok külön odafigyelést igényelnek.

Ez a csomag tartalmazza a CSV fájlok olvasásához és írásához a **CSVReader** és **CSVWriter** osztályokat. Mindkettő adatfolyamként teszi lehetővé a szöveges cellák soronkénti feldolgozását.

## **gui.sieve**

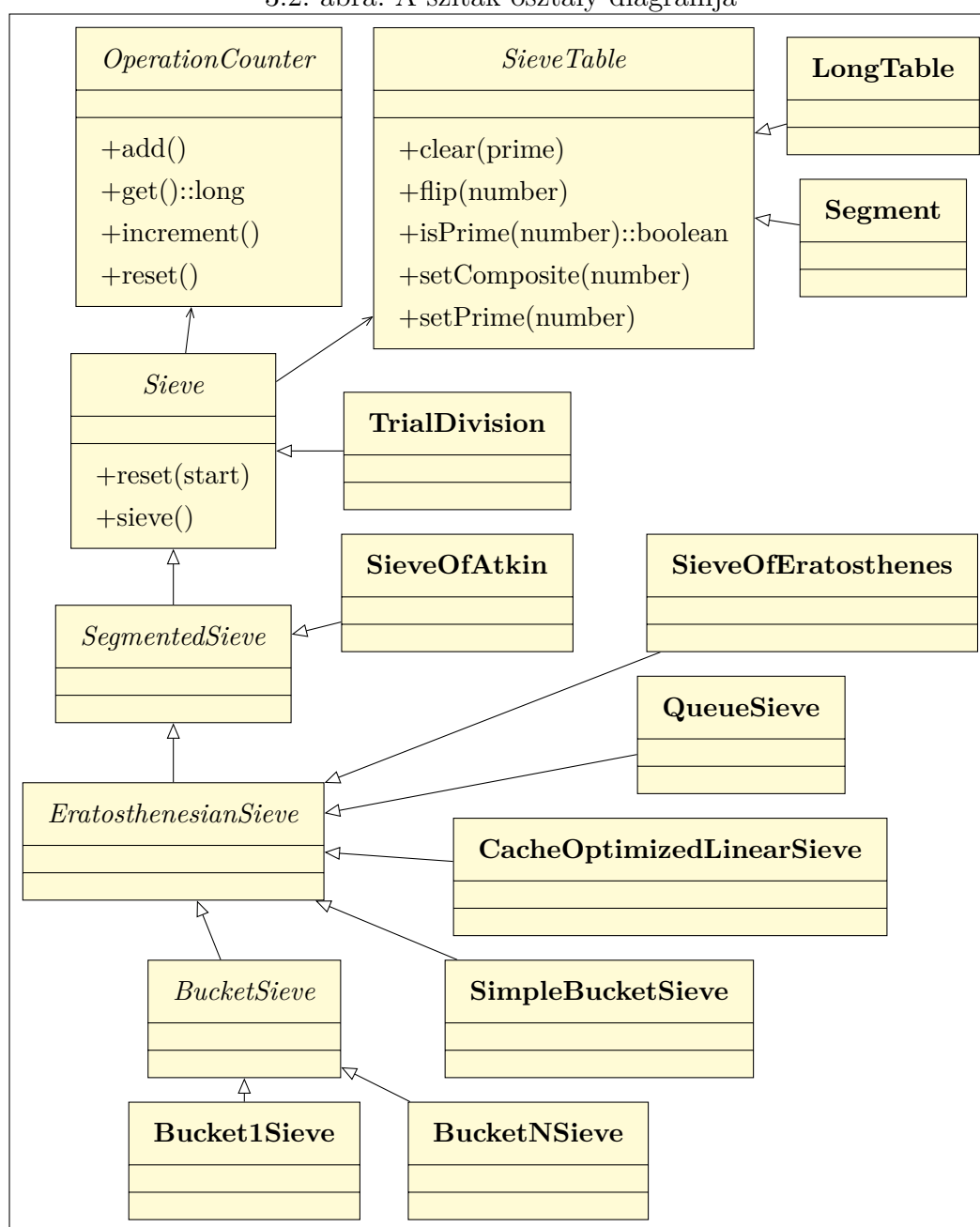
Ebben a csomagban, és a **gui.sieve.eratosthenes** alcsomagjában, a futási idők méréséhez implementált sziták vannak.

A sziták a **Sieve** osztály leszármazottai. A példányaitól elvárás, hogy

- a szitálást szegmensenként végezzék
- a szitatábla bitvektorát csak korlátozottan éri el
  - csak az éppen szitált szegmensbe eső számokat éri el
  - páros számot sose ér el
  - legfeljebb  $2^{32} - 1$ -ig olvassa vissza a szitalás eredményét
- a szegmens mérete szabályozható, a lehetséges értékek a kettő hatványai  $2^8$ -tól  $2^{30}$ -ig
- bármikor újra lehessen inicializálni a szitát egy szegmens kezdetéhez úgy, hogy onnantól már további felkészülési idő nélkül több egymás utáni szegmenst tud szitalni
- az **OperationCounter** osztály segítségével számon tartja a szitalás művelet-igényét.



3.2. ábra. A sziták osztály diagramja



A `SieveTable` osztály `Segment` implementációja az adatbázisnál is használt szegmensfájl, szita ellenőrzésekor ez a szitatábla tárolja a bitvektort, és az ellenőrzés a végeredményt hasonlítja össze a referenciaszita eredményével. A `LongTable` megvalósítást a program a futási idő mérésénél használja, ez a szitatábla mindig  $2^{32}$ -ig tartja nyilván az eredményt, annál nagyobb számoknál a művelet elvégzi a memóriában, de különböző számok ugyanazt a bitet módosíthatják, így egy számra vonatkozó eredmény nem olvasható vissza.

Mindegyik szita megvalósítása valamilyen formában eltárolja a már megtalált prímeket, de csak a szitált szegmens végének négyzetgyökéig veszik azokat figyelembe. A sziták a prímelek és pozíciójuk tárolását, egy kivétellel, egy vagy két

`PrimitiveList` használatával oldják meg, az elemek csoportosítását a listán belüli helyük adja. Az edénysoron alapuló sziták belső edényeinek pillanatnyi mérete tág határok között mozog, azért a memóriaveszteség csökkentésére ezek az edények kisebb, fix méretű edények láncolt listái. Az edénysor ezeket a listaelemeket feldolgozás után újrahasznosítja a szemétgyűjtés elkerüléséhez.

A `CacheOptimizedLinearSieve` futás közben nehezebben bővíthető új prímmel, mint a listás vagy elsőbbségi soros szitaké. Ezért a COLS szita  $2^{32}$ -ig egy egyszerű listás eratoszthenészi szitaként viselkedik, és az azutáni első szegmens szitalásakor alakítja ki a prímek csoportosítását.

A sziták egyszerű hierarchiát alkotnak. A `SegmentedSieve` leszármazottai minden prímmel csak egyszer foglalkoznak egy szegmensben, bár lehet, hogy az több helyen szital. Az `EratosthenesianSieve` típusú sziták a prímekkel úgy szitalnak, hogy a prím eltárolt utolsó többszörösét a prím kétszeresével növelik, amíg a többszörös a szegmensbe esik.

A sziták algoritmusai a "Sziták" fejezetben vannak részletesebben kifejtve. Az implementációk főbb jellemzői:

- `TrialDivision`: próbaosztás
- `SieveOfAtkin`: Atkin szitája
- Eratoszthenész szitájának implementációi, mindegyik a prím-pozíció párok eltárolásával próbálja gyorsítani a futását
  - `SieveOfEratosthenes`: a prímekeket egy tömbben tárolja, és minden szegmens szitalásához minden prímmel próbál szitalni a szegmensben
  - `QueueSieve`: a prímekeket bináris kupacban tárolja, szitaláskor a kupac elejét dolgozza fel, amíg az a szegmensbe esik.
  - `CacheOptimizedLinearSieve`: a prímekeket kétszintű edényrendszerben tárolja, szitaláskor a szegmensbe eső elemek már edényekben vannak válogatva, azokat csak be kell járni
  - `SimpleBucketSieve`: a prímekeket a `QueueSieve`-hez hasonlóan egy elsőbbségi sorban tartja nyilván, az elsőbbségi sor egyre nagyobb intervallumokat lefedő edények listája
  - `Bucket1Sieve`: az edénysoron alapuló szita felgyorsítása a COLS 0. "körével", a szegmensek méreténél kisebb prímekeket a sortól külön, egy tömbben tárolja. Ezek a kis prímek minden szegmensben szitalnak, felesleges lenne a sorból minden szegmensnél kivenni, majd visszahelyezni.
  - `BucketNSieve`: a `Bucket1Sieve` további gyorsítása, az edénysor számrendszerének alapját megnövelve a prímek részleges rendezését finomítja.

## **gui.ui**

Ebben a csomagban a Swing használatát megkönnyítő osztályok vannak.

A program legtöbb művelete hosszú időt vesz igénybe, és a felhasználó az elindítása után már nem tudja befolyásolni. A `gui.ui.progress` alcsomag az ilyen lassú folyamatok visszajelzését segíti. A `Progress` interfészen keresztül egy folyamat százalékban visszaadhatja, hogy hol tart. A `Progress.subProgress()` függvénnyel részjeljárások előtt elfedhető a teljes megoldásban elfoglalt helyük, az új `Progress` példány a teljes  $[0, 1]$  intervallumon vett készenléti aránnyal hívható. Az interfésznek két implementációja adott, parancssorhoz a `PrintStreamProgress`, és a grafikus `GuiProgress`. Mindkét megvalósítás ügyel arra, hogy gyakori státuszfrissítés esetén a megjelenítés ne tartsa fel a ténylegesen végzendő műveletet.

A lassú folyamatok megvalósításának másik segédosztálya a `GuiProcess`. Ez az absztrakt osztály megszervezi a grafikus felületen a háttérfolyamatok futtatását. A Swing szál feltartása helyett a lassú művelet háttérben futtatható részét egy külön szálban futtatja, a műveletet a `background()` metódusában kell megvalósítani. A `background()` függvény lefutása után a `foreground()` metódust a Swing szálban hívja meg, az eredmény itt jeleníthető meg. Az eredmény átadása a két szál között az osztály megvalósítására van bízva, a szálak szinkronizálásának megkönnyítésére a `background()` metódusból visszatérés garantáltan *happens-before* relációban van a `foreground()` meghívásával. A `GuiProcess` a `background()` futásához egy `GuiProgress` példányt is létrehoz.

## **Grafikus megjelenítés**

A grafikus program fő ablakát a `gui.Gui` osztály írja le. A program ablakai egy közös `Session` objektumon keresztül fenntartanak szálakat a háttérfolyamatok futtatására. Ez az objektum számon tartja a nyitott ablakok számát, és az utolsó becsukásával a háttérszálakat szabályosan leállítja.

## **Parancssor**

A `gui` program legtöbb művelete parancssorból is indítható. A parancssori műveletek írják le a `gui.Command.Descriptor` osztály példányai. A `gui.Main.main()` a program indítási paramétereit ezekkel a leírókkal összevetve választja ki a futtatandó parancsot. A leíró a parancs belépési pontja mellett a megengedett argumentumok listáját tartalmazza. A parancs kiválasztásához a tényleges és elvárt argumentumok meg kell egyezzenek. Az argumentum leírása reguláris kifejezéssel szintaktikailag ellenőrzi a tényleges argumentumokat, a szemantikai helyesség ellenőrzése minden parancs saját feladata.

## Adatbázis műveletek

Az adatbázison három művelet végezhető, ezek implementációja a **Database** osztály **info()** metódusa, ami összegyűjti az adatbázisban tárolt szegmensek információit, az **importAggregates()** a megadott és az adatbázis összesítőfájlát összefésüli, és a **reaggregate()** az új szegmensfájlokat összesíti. Ez a két művelet a szegmens és összesítőfájl segédosztályait használja, az összefésülő rendezéshez hasonlóan ezek egyetlen menetben oldják meg a feladatot.

Egyik művelet sem interaktív, a paraméterek megadása és az eredmény vissza-jelzése is a legegyszerűbb megoldásokat használja grafikusán és parancssorban is. Ezek megvalósítása a **gui.io.command** csomagban található.

## Szegmensek és sziták ellenőrzése

Ez a két művelet a **gui.check** csomagban van megvalósítva. Mindkettő parancssorból és grafikusán is indítható, az adatbázis műveletekhez hasonlóan a paraméterek bekérése és az eredmény megjelenítése egyszerű.

Mindkét művelet az ellenőrzéseket szegmensenként végzi. Az összehasonlítás-hoz a **ReferenceSegment** osztály állítja elő a referenciának vett szitatáblát. A szitatáblát a **ReferenceSegment.SIEVE** példány Eratoszthenész szitával készíti el, a **ReferenceSegment.TEST** az erős pszeudoprím teszttel.

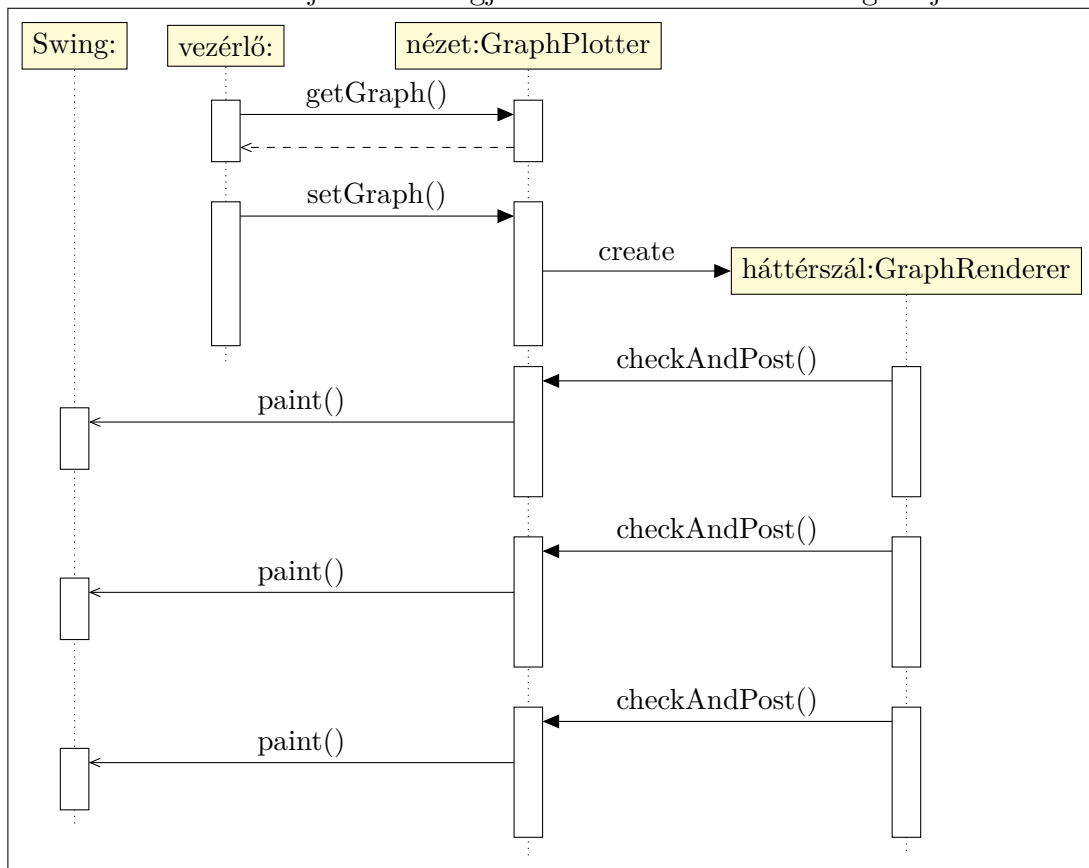
A **CheckSegments.checkSegments()** metódus végzi a szegmensek ellenőrzését, a kiválasztott szegmensek beolvasását, a referencia generálását, és az összehasonlítását. A **CheckSieve.checkSieve()** metódus ehhez hasonlóan működik, az ellenőrizni kívánt szegmenseket nem fájlból olvassa be, hanem a kiválasztott szitával állítja elő, és referenciának mindig a szitát választja.

## Grafikon ábrázolása

A grafikonok megjelenítése a program egyetlen interaktív része, ennek kezelésére a modell-nézet-vezérlő mintát használja. A modell a megjelenítendő minták és függvények, valamint a grafikon megjelenítendő intervalluma. A nézet a sztenderd Swing komponenseket használja. A vezérlő a **gui** segédosztályaira épít.

A modell alapján az megjelenítéshez szükséges grafikus utasítások előállítása időigényes, a mintákat és a függvényeket pixel széles intervallumokon mintavételezve határozza meg a képernyő egy oszlopához tartozó minimum és maximum értékeket. Az interaktivitás érzésének fenntartásához az oszlopokra bontást a program egy háttérszámban végzi, és közben a részeredményeket rendszeresen átadja a nézetnek megjelenítésre. A felhasználó a háttérszámítások közben a modellt módosítva az addigi előszámításokat elveszíti, és egy új háttérszám indul.

3.3. ábra. Új modell megjelenítésének szekvencia diagramja



## Modell

A modell osztályai a `gui.graph` csomagban vannak. A modellt a `Graph` osztály írja le. Ez egy nem módosítható konténerosztály, az ábrázoláshoz szükséges összes paramétert tartalmazza:

- a mintákat, ezeket a `Sample` osztály írja le
- a függvényeket, ezeket a `Function` osztály írja le
- a grafikon ábrázolására használható képernyőrész szélességét és magasságát pixelekből
- a grafikon megjelenített részét, amit mintatér koordináta rendszerében lehet megadni egy téglalap sarkaival
- a megjelenítéshez használt színeket

A `RenderedGraph` osztály tartalmazza a minták és függvények oszlopokra bontását. A `Graph` osztályhoz hasonlóan ennek az osztálynak a példányai se módosíthatóak. A `RenderedGraph` tartalmazza a hozzá tartozó `Graph` referenciáját is, ezzel mindig ellenőrizhető, hogy a két modell objektum egymáshoz tartozik.

A **RenderedSample** osztállyal reprezentált oszlopminták egybefüggő intervallumként vannak megadva a **RenderedInterval** osztály segítségével, de minta esetén mindig egy intervallum az oszlopokra bontás eredménye, mert a minták értelmezési tartománya véges, összefüggő intervallum sehol nincs rajta.

Ezeknek az osztályoknak a módosíthatatlansága megkönnyíti a helyes többszálú feldolgozást. Egy kész objektumot egy szál sem fog megváltozni látni, és a referenciák megjegyzésével könnyen követhető, hogy melyik a legújabb modell.

A grafikon felbontását a **GraphRenderer** osztály vezérli. A minták felbontásához pixel széles oszlopok intervallumán megkeresi a legkisebb és a legnagyobb mintaértéket, és az összes ilyet egyetlen mintaintervallumba teszi. A mintákat a nézet minden kitöltött oszlop között összeköti egy vékony egyenessel. Ahhoz, hogy a megjelenítés szélein túlra is be legyen húzva ez az egyenes, az oszlopmintába a megjelenített intervallumtól balra és jobbra eső mintaelem is bekerül.

Függvények előkészítésénél az eljárás ehhez nagyon hasonló. Az oszlop minimum és maximum értékét pontos módszer helyett mintavételezéssel határozza meg. Az oszlop intervallumában több, egyenletes távolságra lévő ponton kiértékeli a függvényt, és ezekből határozza meg az értékeket. A függvény oszlopmintájában az intervallumok összefüggők, a függvény a legelső oszlop bal szélétől a legutolsó oszlop jobb széléig értelmezve van. A nézet ennek megfelelően az intervallumok közötti szakaszokon semmit nem rajzol ki.

A **GraphRenderer** a nézettel a **CheckAndPost** interfészen keresztül kommunikál a pontos nézetimplementáció ismerete nélkül. A **check()** metódussal le tudja ellenőrizni, hogy a legújabb modell az-e, amit éppen feldolgoz, a **checkAndPost()** metódussal az előbbi ellenőrzés után az addig elkészült részeredményt tovább is adja megjelenítésre. Ha a modell már elavult, mindkét **check...()** metódus egy **RendererDeathException**-t dob, és a **GraphRenderer** hibajelzés nélkül leáll. Ilyenkor egy másik szálon már dolgozik egy új **GraphRenderer** az új modellen.

## Vezérlő

A grafikon megjelenítésének ablaka a **Plotter** osztály. A vezérlő ennek az ablaknak az eseménykezelői, amik a modell módosításán keresztül tudják módosítani a megjelenített grafikont. A modellen végezhető műveletek:

- a grafikon megjelenített részének mozgatása, nagyítása, kicsinyítése
- mintát hozzáadása
- minta törlése
- mintát közelítése

- minta törlése

Ezek megvalósítására a program a Swing sztenderd komponenseit használja.

A mintát a modellhez adni három forrásból lehet. Az adatbázis összesített statisztikáiból. Ezek előállítását az **Aggregates** osztály feladata. Be lehet tölteni mintát CSV fájlból, ezt a **LoadSampleProcess** oldja meg. Vagy meg lehet mérni egy minta futási idejét. A futási idő mérése a parancssorból indítható mérést használja, a **MeasureSieve** ablak a mérés paramétereinek megadására szolgál.

A minta közelítésének számítási részét a **gui.math** csomag megoldja. Az elemi függvények kiválasztását a **FunctionSelector** ablak oldja meg. Elemi függvényt JavaScript nyelven a **CustomFunctionDialog** ablakban lehet megadni.

A **GraphPlotter** komponens a grafikonon egérrel történő mozgatásáról és nagyításáról eseményeket küld a vezérlésnek, a nézet osztály is csak a vezérlésen keresztül módosítja a modellt.

## Nézet

A nézetet a **GraphPlotter** osztály reprezentálja. A két részből álló előkészített modell megjelenítése a grafikus vezérlőutasítások kiadásából áll.

Ennek a komponensnek a **graph** adattagja tartalmazza a legújabb modell értéket, a vezérlés, az előkészítés, és a megjelenítés ezen keresztül szinkronizálva kommunikál. A modell módosításakor a **GraphPlotter** objektum a **graph** adattag cseréjével biztosítja, hogy a régi modellre vonatkozó háttérszálak leállnak, és elindítja az új modell oszlopokra bontását. Az előkészített modellen a **CheckAndPost** eseménykezelőjén keresztül értesül, aminek hatására ütemezi az új grafikon kirajzolását. A tényleges kirajzolásnál még egyszer ellenőrzi, hogy a grafikon a komponens tényleges méretére készült el, majd a már elkészült grafikon-részeket kirajzolja.

## Sziták

A **gui** program tartalmazza Eratoszthenész szitájának több megvalósítását. Ezek az implementációk mind szegmentáltak, azaz csak egy rövidebb intervallumon szítálnak, és annak befejezéséig nem kezdenek bele új intervallumba. Egy intervallum szítaláshoz szükséges az intervallum négyzetgyökének végéig a prímek listája. A prímeket ismerve maradékos osztással meghatározható, hogy melyik a legkisebb többszörösük, ami legalább akkora, mint az intervallum kezdete, és onnantól összeadással az intervallum végéig meghatározható a többi többszörösük. A 3.1 algoritmus ezt írja le.

Ha több egymás utáni intervallumot kell szítálni, akkor minden prímhez eltárolható, hogy melyik pozícióban fog legközelebb szítálni. Ezzel az osztások időbeli

---

**Algoritmus 3.1** Az  $[u, v[$  intervallum szitálása

---

```
1: legyenek a számok az  $[u, v[$  intervallumban megjelöltenek
2: for  $p \in \{\text{prímek} \sqrt{v-1}\text{-ig}\}$  do
3:   for  $f \leftarrow p \left\lceil \frac{u}{p} \right\rceil; v > f; f \leftarrow f + p$  do
4:     legyen  $f$  megjelölve
5:   end for
6: end for
7: for  $f \in [u, v[$  do
8:   if  $f$  nincs megjelölve then
9:      $f$  prím
10:  end if
11: end for
```

---

költsége tárhely költségre váltható. A legegyszerűbb megoldás a prím-pozíció párok tárolása egy tömbben. A 3.2 algoritmus az  $[u, v[$  intervallum szitálását írja le, ahol  $v = u + kd$ , és  $d$  hosszú részintervallumonként szitál.

---

**Algoritmus 3.2** Az  $[u, v[$  intervallum szitálása,  $v = u + kd$ 

---

```
1: legyen  $t$  egy tömb
2:  $s \leftarrow 0$  ▷ a tömb elemeinek száma
3: for  $p \in \{\text{prímek} \sqrt{v-1}\text{-ig}\}$  do
4:    $t[s] \leftarrow (p : p, f : p \left\lceil \frac{u}{p} \right\rceil)$ 
5:    $s \leftarrow s + 1$ 
6: end for
7: for  $\ell \leftarrow 1, v \leftarrow u + d; k \geq \ell; \ell \leftarrow \ell + 1, u \leftarrow u + d, v \leftarrow v + d$  do
8:   legyenek a számok az  $[u, v[$  intervallumban megjelöltenek
9:   for  $i \in [1, s]$  do
10:    for ;  $v > t[i].f; t[i].f \leftarrow t[i].f + t[i].p$  do
11:      legyen  $t[i].f$  megjelölve
12:    end for
13:  end for
14:  for  $f \in [u, v[$  do
15:    if  $f$  nincs megjelölve then
16:       $f$  prím
17:    end if
18:  end for
19: end for
```

---

A gui programban "Eratoszthenész szitája" ezt az algoritmust valósítja meg. Ezzel a módszerrel a program minden rövid szegmensben a tömb minden elemét sorra veszi, azokat is, amik a szegmensben nem szitálnak. A többi implementált eratoszthenészi szita összetettebb struktúrákkal próbálja kiválasztani a ténylegesen szitáló elemeket. A többi implementált szita prioritásos sort használ az elemek részleges rendezéséhez, és sor elejének feldolgozásával választja ki a szegmensben szitáló elemeket. A bináris kupac és az edénysort használó szita mellett a Cache



Optimalizált Lineáris Szita is soron alapul.

## Prioritásos sorok

A prioritásos soron alapuló szita algoritmus nagyon hasonló a 3.2 algoritmusához. Az elemek sorban feldolgozása helyett a sor elejét addig veszi ki, amíg az éppen szitált szegmensbe esik. Ezekkel az elemekkel a szitálást elvégzi a szegmensben, majd a sorba visszahelyezi, már az új, mostani szegmensnél nagyobb pozícióval. Az elemek sorrendjét a legközelebbi szitált pozíció határozza meg, a sor eleje az éppen szitált szegmenshez legközelebbi elem. Az elemek sorrendjénél a szegmensen belüli sorrend vizsgálata felesleges. A prioritásos soron alapuló szitálás algoritmusát a 3.3 adja meg, a sor adatszerkezet meghatározása nélkül. A tényleges megvalósítás az eltávolít-hozzáad sorműveletpár helyett a helyben módosítást és a struktúra invariánsának visszaállítását is választhatja.

---

**Algoritmus 3.3** Az  $[u, v = u + kd[$  intervallum szitálása, prioritásos sorral

---

```
1: legyen  $q$  egy üres sor
2: for  $p \in \{\text{prímek} \sqrt{v-1}\text{-ig}\}$  do
3:   hozzáad( $q, (p : p, f : p \lceil \frac{u}{p} \rceil)$ )
4: end for
5: for  $\ell \leftarrow 1, v \leftarrow u + d; k \geq \ell; \ell \leftarrow \ell + 1, u \leftarrow u + d, v \leftarrow v + d$  do
6:   legyenek a számok az  $[u, v[$  intervallumban megjelöltenek
7:   while  $v > \min(q).f$  do
8:      $(p, f) \leftarrow \text{eltávolít-min}(q)$ 
9:     for ;  $v > f; f \leftarrow f + p$  do
10:      legyen  $f$  megjelölve
11:    end for
12:    hozzáad( $q, (p, f)$ )
13:  end while
14:  for  $f \in [u, v[$  do
15:    if  $f$  nincs megjelölve then
16:       $f$  prím
17:    end if
18:  end for
19: end for
```

---

A gui program szitái közül ezt az algoritmust követi az edénysor szita, a bináris kupac szita, és a COLS körkörös listája is. A bináris kupac szitánál az eltávolítás-hozzáadás és a helyben megjavítás között a felhasználó választhat.

## Edénysor

A edénysor egy monoton prioritásos sor. Minden állapotához tartozik egy érték, a sor aktuális pozíciója, aminél kisebb vagy egyenlő pozíciójú értéket a sor nem tar-

talmazhat. A sor pozíciójának megnövelése a sor elejének eltávolításával is együtt jár. A sor edények egy végtelen sorozatát is tárolja, a sor elemei ezekbe az edényekbe kerülnek. Egy eltárolt elem helyét az edények között az elem pozíciójának és a sor aktuális pozíciójának távolsága határozza meg. Két szám távolságát egy számrendszerben két érték határozza meg, a legnagyobb helyiérték, amiben eltérnek a számjegyeik, és a nagyobbik szám számjegye ezen a helyiértéken.

Legyen  $a \in \mathbb{N}, a > 1$  a sor számrendszerének alapszáma.  $x$   $i$ -edik számjegyét  $a$  alapú számrendszerben jelölje  $x_i$ , ahol  $x \in \mathbb{N}_0$ . Legyen  $h(x, y)$  a legnagyobb helyiérték, ahol  $x$  és  $y$  eltér. A sor által használt  $d(x, y)$  távolságfüggvény az eltérés diszkrét logaritmusát követi.

$$x = \sum_{i=0}^{\infty} x_i a^i \quad (i \in \mathbb{N}_0, x_i \in \mathbb{N}_0, x_i < a) \quad (3.1)$$

$$h(x, y) = \max \{i \in \mathbb{N}_0 | x_i \neq y_i\} \quad (x, y \in \mathbb{N}_0, x < y) \quad (3.2)$$

$$d(x, y) = (a - 1)h(x, y) + y_{h(x, y)} - 1 \quad (3.3)$$

Kettes számrendszerben a távolság logaritmussal és a bitenkénti kizáró-vagy művelettel is megadható,

$$d(x, y) = \lfloor \log_2(x \oplus y) \rfloor$$

Legyen  $q$  egy edénysor

- $a$  alapszámmal
- $p(q) \in \mathbb{N}_0$  a  $q$  sor pozíciója
- $b(q, i)$  a  $q$  sor  $i$ . edénye ( $i \in \mathbb{N}_0$ )
- $p(r) \in \mathbb{N}$  a  $q$  sor egy lehetséges  $r$  elemének pozíciója.

Ekkor a  $q$  sor invariánsa:

$$\forall r \in q :$$

$$p(q) < p(r)$$

$$\forall i \in \mathbb{N}_0 : r \in b(q, i) \iff i = d(p(q), p(r))$$

$$\forall r \notin q : \forall i \in \mathbb{N}_0 : r \notin b(q, i)$$

Új sor létrehozásához az edények listáját kell létrehozni, a sor kezdő pozíciója bármi lehet. Új elemet a sorhoz a távolságfüggvény kiértékelése után az elem edénybe

szúrásával lehet adni, az új  $r$  elem az  $b(q, d(p(q), p(r)))$  edénybe kell kerüljön. Ez a művelet az invariánst fenntartja.

A sor pozíciójának eggyel megnövelésével a sor invariánsa kétféleképpen válhat hamissá:

- egy elem pozíciója és a sor pozíciója egyenlő lesz. A pozíció növelése művelet ezeket az elemeket a sorból eltávolítja, és a művelet eredményeként visszaadja.
- az elem és a sor pozíciójának távolsága csökken. Az invariáns visszaállításához a művelet ezeket az elemeket áthelyezi az új távolság szerinti edénybe.

A sor elejének eltávolítsa az edénysornál tetszőleges számú elemet távolíthat el a sorból, és nem üres sor esetén is előfordulhat, hogy a sor elejének eltávolítása nem ad vissza elemeket.

---

**Algoritmus 3.4** A  $q$  edénysor elejének eltávolítása

---

```

1: function ELTÁVOLÍT-MIN( $q$ )
2:    $l \leftarrow$  üres lista
3:    $i \leftarrow d(p(q), p(q) + 1)$ 
4:    $p(q) \leftarrow p(q) + 1$ 
5:   while  $b(q, i)$  nem üres do
6:      $r \leftarrow$  eltávolít( $b(q, i)$ )
7:     if  $p(r) = p(q)$  then
8:       hozzáad( $l, r$ )
9:     else
10:      hozzáad( $b(q, d(p(q), p(r)))$ )
11:    end if
12:  end while
13:  return  $l$ 
14: end function

```

---

### Funkcionális megvalósítás

Az edénysoron alapuló szitára tisztán funkcionális megoldás adható. Az edénysor műveleteit láncolt listákkal ábrázolt számok összeadására lehet visszavezetni, ezzel a műveletigény akkor is a számjegyenkénti összeadást követi, ha a számítási modellben az összeadás konstans idejű. Az  $x + y$  összeadás elvégzésével a  $h(x, x + y)$  és a  $d(x, x + y)$  is kiszámítható,  $h(x, x + y)$  a helyiérték, ahol összeadás közben a kisebbik szám számjegyei és az átvitel is elfogy. A  $d(x, x + y)$  meghatározásához az eredményből még a  $h(x, x + y)$  helyiértéken lévő számjegyet is ki kell olvasni.

Az edénysor pozíciójának növelése algoritmus a kiválasztott edény újrendezését számjegyenként végzi egy gyors modellben is. Egy elem a sorból kikerüléséig  $h(x, x+y)$  edényt bejárhat, és ez az érték legalább annyi, mint  $y$  számjegyeinek száma,  $\log_a y$ .

A CD mellékleten megtalálható az edénysor szita egy Haskell megvalósítása a `hs` könyvtárban. A program kettes számrendszerű edénysort használ, és a szitalás minden műveletét bit-listák feldolgozására vezeti vissza.

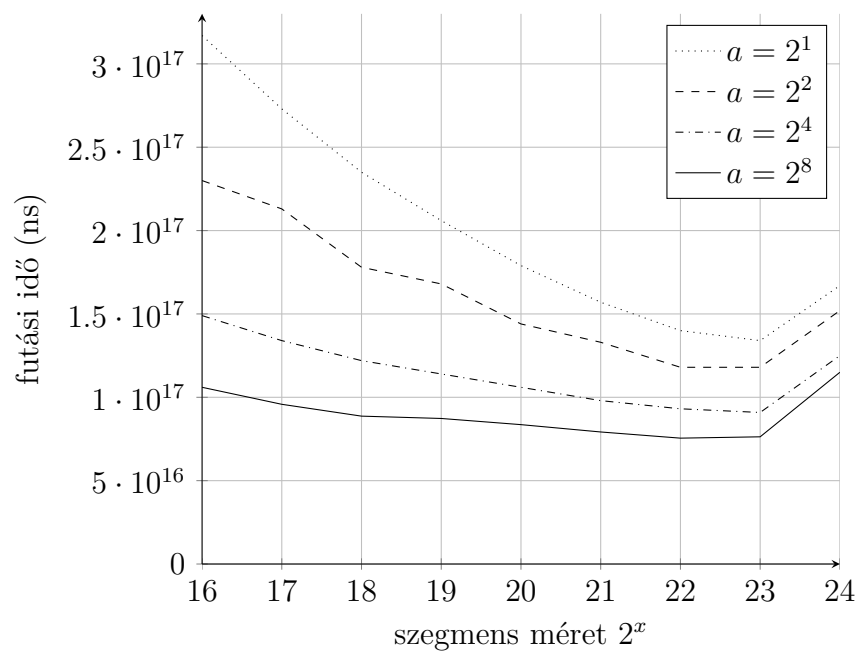
## Sebesség

A `generator` program az edénysoron alapuló szita optimalizált változata. Az optimalizálások közül a szegmensméret, a sor számrendszerének alapja tág határok között megválasztható. A szegmens hosszánál kisebb prímek függenek a szegmens méretétől, és a 64-nél kisebb prímek a felhasznált 64 bites C típustól függenek, ezek szabadon nem választhatók meg. A `measure-generator` szkript a szegmensméret és számrendszer választások sebességét méri meg. Mindkét paramétert 2 hatványainak választja, a szegmensméretet  $2^{16}$ -tól  $2^{24}$ -ig, a számrendszert  $2^1$ -től  $2^8$ -ig.

3.1. táblázat. A generator futási ideje (ns), szitatábla  $[2^{63}, 2^{63} + 2^{34}[$

szegmens / alap	$2^1$	$2^2$	$2^4$	$2^8$
$2^{16}$	$3.17 \times 10^{11}$	$2.30 \times 10^{11}$	$1.49 \times 10^{11}$	$1.06 \times 10^{11}$
$2^{17}$	$2.73 \times 10^{11}$	$2.13 \times 10^{11}$	$1.34 \times 10^{11}$	$9.58 \times 10^{10}$
$2^{18}$	$2.35 \times 10^{11}$	$1.78 \times 10^{11}$	$1.22 \times 10^{11}$	$8.87 \times 10^{10}$
$2^{19}$	$2.06 \times 10^{11}$	$1.68 \times 10^{11}$	$1.14 \times 10^{11}$	$8.73 \times 10^{10}$
$2^{20}$	$1.79 \times 10^{11}$	$1.44 \times 10^{11}$	$1.06 \times 10^{11}$	$8.36 \times 10^{10}$
$2^{21}$	$1.57 \times 10^{11}$	$1.33 \times 10^{11}$	$9.80 \times 10^{10}$	$7.92 \times 10^{10}$
$2^{22}$	$1.40 \times 10^{11}$	$1.18 \times 10^{11}$	$9.31 \times 10^{10}$	$7.55 \times 10^{10}$
$2^{23}$	$1.34 \times 10^{11}$	$1.18 \times 10^{11}$	$9.09 \times 10^{10}$	$7.63 \times 10^{10}$
$2^{24}$	$1.67 \times 10^{11}$	$1.52 \times 10^{11}$	$1.25 \times 10^{11}$	$1.15 \times 10^{11}$

3.4. ábra. A generator futási ideje (ns), szitatábla  $[2^{63}, 2^{63} + 2^{34}[$   
 $\cdot 10^{-6}$



## Az edénysor helyessége

Egy szám számjegyenkénti felírásából, és  $h$  definíciójából következik, hogy

$$x_{h(x,y)} < y_{h(x,y)} \quad (x, y \in \mathbb{N}_0, x < y) \quad (3.4)$$

A sor pozíciójának pontosan eggyel növelésének következménye, hogy a legnagyobb megváltozott helyiértéken eggyel nő a számjegy, és az összes kisebb helyiértéken a legnagyobb számjegy 0-ra vált. Ha  $x = p(q)$ ,  $y = p(q) + 1$ ,  $d' = d(x, y)$ ,  $k = h(x, y)$ , akkor

$$x_k + 1 = y_k \quad (3.5)$$

$$\forall i \in \mathbb{N}, i > k : x_k = y_k \quad (3.6)$$

$$\forall i \in \mathbb{N}, i < k : x_k = a - 1 \text{ és } y_k = 0 \quad (3.7)$$

A sor elejének eltávolítása algoritmus helyessége következik abból, hogy a  $d'$ -nél kisebb indexű edények üresek, a  $d'$ -nél nagyobb indexű edények elemeinek távolsága nem változik a sor pozíciójának növelésével, és a  $d'$  edény elemeit az algoritmus újrarendezi.

$$\forall i \in \mathbb{N}_0, i < d' : b(q, i) = \emptyset \quad (3.8)$$

$$\forall i \in \mathbb{N}_0, i > d' : \forall r \in b(q, i) : i = d(p(q), p(r)) = d(p(q) + 1, p(r)) \quad (3.9)$$

3.8 bizonyításához tegyük fel indirekt, hogy  $\exists i < d' : \exists r \in b(q, i)$ . Legyen  $k' = h(p(q), p(r))$ . Az indirekt feltevés szerint

$$d(p(q), p(r)) = i < d' = d(p(q), p(q) + 1)$$

Ha  $k' \neq k$ , 3.4 és 3.7 ellentmondáshoz vezet:

$$p(q)_{k'} = a - 1 < p(r)_{k'} < a$$

Ha  $k' = k$ , akkor  $i < d'$  és 3.3 miatt  $p(r)_k < (p(q) + 1)_k$ . Ekkor 3.4 és 3.5 ellentmond egymásnak:

$$p(q)_k < p(r)_k < (p(q) + 1)_k = p(q)_k + 1$$

Tehát az indirekt feltevés ellentmondáshoz vezet, azaz 3.8 igaz, a  $d'$ -nél kisebb indexű edények üresek.

Legyen  $i \in \mathbb{N}_0, i > d', r \in b(q, i), k' = h(p(q), p(r))$ . Ha  $k' = k$ , akkor

$$d(p(q), p(r)) = (a - 1)k' + p(r)_k - 1 = d(p(q) + 1, p(r))$$

Ha  $k \neq k'$ , akkor 3.3 szerint  $k < k'$ . 3.6 szerint

$$p(q)_{k'} = (p(q) + 1)_{k'}$$

$$h(p(q), p(r)) = k' = h(p(q) + 1, p(r))$$

$$d(p(q), p(r)) = (a - 1)k' + p(r)_k - 1 = d(p(q) + 1, p(r))$$

Ezek alapján 3.9 igaz, az elem távolsága nem változik a sor elejéhez képest.

## A program ellenőrzése

Az megoldás szitáinak ellenőrzése a `gui` program feladata. Az `init` és `generator` program szegmensfájllai, és a `gui` szitáinak kimenetei is a `ReferenceSegment` osztály szitájával összehasonlíthatóak. Ezen keresztül a sziták kimenetének egyenlősége szűrőpróbaszerűen ellenőrizhető. Ezeket az időigényes ellenőrzéseket végzi el a `test-sieves` szkript minden szitával több intervallumon. A teszt ellenőrzéséhez a szkript egy szándékosan hibás szegmens, és egy hibás szita ellenőrzését is elvégzi.

A `gui` program `gui.math` és `gui.util` csomagjaira a program összes többi része épít, az adatbázis műveletek, a sziták, és a megjelenítés is. Erre a két csomagra a JUnit keretrendszerrel és a JaCoCo lefedettségmérővel teljes lefedettségű egységtesztek készültek.

Az összesített statisztikák ellenőrzésére a `test-prime-counts` szkript a prímek számát a `samples/test/prime-counts.csv` fájlba írja az adatbázis alapján kettő hatványai helyeken, míg a `test-max-prime-gaps` szkript a maximális prímhézagokat a `samples/test/max-prime-gaps.csv` fájlba menti. Ezek a publikált értékekkel[3][4] összevethetőek. Ezeket a fájlokat a `gui` program `gui.test` csomagjának osztályai készítik el. A program tesztelésénél ez a két statisztika  $2^{47} (\approx 1,4 \cdot 10^{14})$ -ig egyezett.

A minták és közelítő függvények megjelenítésének tesztelése ismert grafikonú mintákkal való összevetéssel történt. Táblázatkezelő programokkal, mint például a LibreOffice, formula alapján generált minták grafikonja ábrázolható, és a `gui` program által olvasott CSV formátumú fájlba menthető. A formulákkal véletlenszerű zaj is hozzáadható a mintához. A `samples/test/graph` könyvtárban megtalálható néhány táblázatminta a teszteléshez.

## Numerikus pontosság

A legkisebb négyzetek módszerének megvalósítása a mátrixműveletek pontosságát két paraméterrel tudja szabályozni, a mátrixszorzás összegzéseit három különböző algoritmussal lehet végezni, és a lineáris egyenletrendszereket meg lehet oldani QR felbontással, és Gauss-eliminációval részleges sorcserével és teljes sor és oszlop cserével. Ezeknek a paramétereknek a megválasztása a tesztek alapján sem egyértelmű.

A `test-measure-sums` közelítések pontosságáról készít statisztikákat három változóban, és az eredményt a `samples/test/measure-sums.csv` fájlba menti. A három változó:

- az összegzést tömbbel, kupaccal, vagy egyetlen változóval végezze
- az egyenletrendszerek megoldását QR felbontással, részleges vagy teljes Gauss-eliminációval végezze



- $x$  szerint rendezett-e a minta.

A három paraméter mindegyik lehetőségéhez a mérés elemi függvények alapján egy mintát generál, majd ezt a mintát ugyanezekkel az elemi függvényekkel közelíti. Egy paraméter-kombinációhoz az összes közelítés négyzetes eltérését, és a közelítés idejét összesíti. A minták generálásához használt elemi függvények a **MeasureSums** osztályból kiolvashatóak.

A mérés eredményéből látszik, hogy a kiválasztott minták közelítésénél:

- legjobb, és leglassabb választás a kupac összegzés teljes cserét végrehajtó Gauss-eliminációval
- a kupac összegzés tízszer annyi időt vesz igénybe, mint a másik két összegzés
- az egy változóban összegzés hibája több nagyságrenddel nagyobb, mint a másik két összegzés hibája
- a sor és oszlop csere nem számít egy változóban összegezve, a teljes csere tömbben valamivel pontosabb, és a kupac összegzésnél a teljes csere lényegesen pontosabb eredmény ad
- a QR felbontás általában nagyobb hibát eredményez, mint a Gauss-elimináció
- a kupac összegzés a minta rendezettségére nem érzékeny
- a rendezetlen minta összegzése a változóval és tömbbel pontosabb, mint a rendezett.

Ezek alapján a program a közelítésekre a kupac alapú összegzést használja, és a lineáris egyenletrendszerek megoldására Gauss-eliminációt használ teljes sor és oszlopcserével, ezt a felhasználó nem tudja megváltoztatni.

3.2. táblázat. Numerikus algoritmusok összehasonlítása

Összegzés	LER-m.o.	Rendezett	$\sum \text{hiba}^2$	Idő (ns)
változó	részleges G.E.	igen	$2.53 \times 10^{25}$	$1.27 \times 10^9$
		nem	$4.24 \times 10^{23}$	$1.83 \times 10^9$
	teljes G.E.	igen	$2.53 \times 10^{25}$	$1.28 \times 10^9$
		nem	$4.21 \times 10^{23}$	$1.84 \times 10^9$
	QR	igen	$9.47 \times 10^{24}$	$1.34 \times 10^9$
		nem	$2.58 \times 10^{23}$	$1.83 \times 10^9$
tömb	részleges G.E.	igen	$2.62 \times 10^{20}$	$1.46 \times 10^9$
		nem	$9.75 \times 10^{19}$	$2.43 \times 10^9$
	teljes G.E.	igen	$1.87 \times 10^{20}$	$1.53 \times 10^9$
		nem	$6.76 \times 10^{19}$	$2.37 \times 10^9$
	QR	igen	$6.41 \times 10^{23}$	$1.61 \times 10^9$
		nem	$3.63 \times 10^{20}$	$2.43 \times 10^9$
kupac	részleges G.E.	igen	$1.12 \times 10^{21}$	$1.40 \times 10^{10}$
		nem	$1.12 \times 10^{21}$	$1.76 \times 10^{10}$
	teljes G.E.	igen	$3.12 \times 10^{19}$	$1.39 \times 10^{10}$
		nem	$3.12 \times 10^{19}$	$1.77 \times 10^{10}$
	QR	igen	$1.30 \times 10^{22}$	$1.44 \times 10^{10}$
		nem	$1.30 \times 10^{22}$	$1.75 \times 10^{10}$

## 4. fejezet

# Irodalomjegyzék

- [1] A. O. L. Atkin, D. J. Bernstein: Prime sieves using binary quadratic forms, Mathematics of Computation, Volume 73 (2004) 1023–1030
- [2] A. Járαι, E. Vatai: Cache optimized linear sieve, Acta Univ. Sapientiae, Inform. 3,2 (2011) 205-223
- [3] Tomás Oliveira e Silva, Siegfried Herzog and Silvio Pardi: Empirical verification of the even Goldbach conjecture and computation of prime gaps up to  $4 \cdot 10^{18}$ , Math. Comp. 83 (2014), 2033-2060
- [4] Tomás Oliveira e Silva: Tables of values of  $\pi(x)$  and of  $\pi_2(x)$ , <http://sweet.ua.pt/tos/primes.html>, 2018.11.30.
- [5] David M. Bressoud: Factorization and Primality Testing, Springer Verlag, 1989, 0-387-97040-1
- [6] Wojciech Izykowski, Jim Sinclair: Deterministic variants of the Miller-Rabin primality test, <https://miller-rabin.appspot.com/>, 2018.11.30.