Tátrai János Norbert - Szabó Norbert - Kaczur Sándor[871]: *Szálkiosztó stratégiák optimalizálási lehetőségeinek elemzése*

Absztrakt: Cikkünk témája egy szálakat is kezelő alkalmazás megtervezése, megvalósítása és az eredmények összehasonlítása. Témánk a prímszámokhoz kapcsolódik, több program segítségével keressük azt az intervallum felosztó stratégiát, amivel a legjobban kioszthatjuk a részfeladatokat a szálak között. Jelenleg hét fajta módszert támogatnak a programjaink, de még a programozási nyelveket is versenyeztetni kívántuk, így négy programozási nyelven is elkészítettük a szoftvert. A prímszámok kutatása még ma is fontos feladatnak bizonyul, így a témánk még ezen a ponton is aktuális. Mi egy egyszerű naiv módszeren alapuló prímtesztet alkalmazunk, de ez könnyen kicserélhető akár sokkal intelligensebbre is, így elég jó sebességgel tudja a programunk kigyűjteni egy adott intervallumból a prímszámokat. Először ismertetjük a prímteszteket, az egyszerű algoritmusoktól a valószínűségen alapuló következtetésig. A párhuzamosítást is áttekintjük, hogy egy általános képet kapjunk a hardver és szoftver biztosította technikákról, illetve a problémákat is megemlítjük. Ezután a megvalósított szálkiosztáshoz kötődő stratégiákat ismertetjük, sejtéseket adunk a várható eredményekről. Majd még mindig nyelvtől függetlenül a tervezési folyamatot is dokumentáljuk. Ezután egy gyakorlati fejezet következik, melyben ismertetjük az eltéréseket a négy programozási nyelven megvalósított változat között. A futás közben fontos eredmények keletkeznek, melyet tárolunk egy CSV fájlban, hogy támogassuk az eredmények elemzését. Ismertetjük az állomány felépítését és arról is írunk, miért éppen ez mellett döntöttünk. Levonjuk a következtetéseket: a leggyorsabb módszer, legjobb teljesítményt nyújt programozási nyelv és leghatékonyabbnak bizonyuló szálkiosztáshoz kötődő stratégia szerinti felbontásban is. A fejlesztés és tesztelés során felmerültek bennünk továbbfejlesztésre lehetőséget adó ötletek, így ezek ismertetése sem maradhat el. Végül a felvetett problémákat megválaszoljuk.

Bevezetés

Jelen világunkban igyekszünk törekedni a hardver és szoftver legjobb kihasználtságára. Specifikusan nagy hatékonyságot és teljesítményt szeretnénk kinyerni egy adott rendszerből. Manapság ez az igény megtalálható a szoftveriparban, és az azon belüli fejlesztések között is. Ennek egy részét próbálja hatékonyabbá tenni egyrészt az általunk választott téma, a programozás területén alkalmazott szálkezelés, másrészt annak különböző módszereivel kiosztott feladatmennyiségek. Megvizsgáljuk, hogy egyes esetek kombinációjaként hogyan változhat az adott elvégzendő feladat lebonyolítása idő és hatékonyság szempontjából. Tehát például a különbségekre leszünk majd figyelmesek akkor, amikor egy bizonyos implementált program eltérő darabszámú szálakat futtat, kitérve a különböző programozási nyelvek közötti különbségekre is, amelyeket a megvalósítás szempontjából és az eredmények alapján is szemléltetjük, időbeliségük szerint.

A kitűzött cél egy optimalizálási feladat. A megközelítés újdonsága abban rejlik, hogy ahelyett, hogy egy adott algoritmust próbálnánk matematikailag levezetve, kevesebb lépésszámúvá – úgymond gyorsabbá – tenni, inkább azt vizsgáljuk, hogyan lehet változatlan algoritmus mellett, szálkezelést segítségül hívva, gyorsabb, hatékonyabb implementációs készíteni.

Tehát ez a fajta módszer olyan körülmények között lesz segítség számunkra és mások számára is, amikor már nem nyílik további lehetőségünk egy-egy eljárás, illetve egy algoritmus további lépésszámának csökkentésére semmilyen féle további módon. Ebben az esetben lehetőség kínálkozik arra is, hogy több szálra bontva gyorsítsuk fel az adott feladat végrehajtását. Ez kiváló lehetőség, hiszen manapság több processzoros rendszerekkel rendelkezünk. Nemes egyszerűséggel a többprocesszoros rendszereket – valamint ezek mellett a processzorokon belüli párhuzamosításokat

[871] Gábor Dénes Főiskola tatraijanos@hotmail.com, mind.in.dark@gmail.com, kaczur@gdf.hu

– igyekszünk kihasználni egy nagyobb feladat több szálra bontásával, így hatékonyabb megoldást elérve. Ezt a módszert a szoftveriparban már régóta használják, viszont tanulmányunk inkább arra vállalkozik, hogy a vizsgált szálkezelési módszerekkel többféle programozási nyelven implementált algoritmusok futási eredményeiként kapott eredményeket összegezzük és következtetéseket adjunk a további fejlesztésekre vonatkozóan. Ezek után az adatok alapján szeretnénk a majd kialakuló alkalmazások teljesítményének fokozásában segédkezni, amennyiben ez összekapcsolható a szálkezelés témájával.
Ennek érdekében a konzulensünk által javasolt feladatot vettük alapul, aminél szemléletesebb eredményeket tudtunk kimutatni szálkezeléssel való optimalizálás során. Ez a feladat nem más, mint a prímszámok egy adott intervallumban való kiválogatása. A feladat akkor válik érdekessé, amikor nagyobb intervallumokban szeretnénk prímszámok után kutatni. Ugyanis nagyobb számok esetén egyértelműen megnövekszik a feladat lépésszáma. Így az intervallumok eltérő hosszúságú részekre osztása és ezek önálló szálakkal való feldolgozása látványos különbségeket eredményez, amelyekből következtetéseket vonunk le.

Prímszámokról és alkalmazásukról

Prímszámoknak azokat a természetes számokat nevezzük, amelyeknek pontosan két pozitív osztójuk van a természetes számok halmazán belül. Ez maga a szám és az egy. Akkor tekinthetjük prímnek a számot az egész számok halmazán belül, ha az abszolútértéke is prím.
A számelmélet alaptétele szerint minden nem prímszám (azaz összetett) egyértelműen felírható a prímszámok szorzataként. Ezt nevezzük kanonikus alaknak. Az egyet nem tekintjük prímszámnak. A másik megfogalmazás az már inkább történelmi okra vezethető vissza miszerint a görögök nem tekintették számnak az egyet, így prímszám sem lehetett.
Euklidész (Kr. e. 300 körül) görög matematikushoz kapcsolódik annak legrégebbi bizonyítása, hogy végtelen sok prímszám van [1].
Prímszámok keresésére és kiválogatására számos ismert algoritmus született. A mi programunkban a könnyű párhuzamosítás végett a legegyszerűbbet választottuk, miszerint a vizsgált számot egytől a számig elosztjuk az összes sorra kerülő számmal. Végül megszámoljuk az osztók számát, ha ez kettő, akkor a számot prímszámnak tekintjük. Tehát **az algoritmusunk a naiv módszeren alapszik**. Ha a dolgozatunkban nem a szálkezelésre fektettük volna a fő hangsúlyt, akkor a fenti algoritmust tovább optimalizálhattuk volna.
A matematika még kidolgozott számos olyan algoritmust, illetve indeterminisztikus módszert is, amelyek ismeretében eldönthetjük egy adott számról, hogy prímszám-e. Ezeket a módszereket nevezzük **prímteszt**eknek. Terjedelmi okok miatt nem ismertetjük a módszereket, csak felsorolunk néhányat: AKS-teszt, BPSW-teszt, Fermat-prímteszt, Miller-Rabin teszt, Solovay-Strassen teszt illetve Wilson-prímteszt. Ezek között van olyan módszer is, amit a XXI. században mutattak be, így látszik, hogy az emberiséget még a mai napig foglalkoztatja, hogy minél több prímszámot megismerjen a lehető leghatékonyabb módszert használva [2]. A módszerek többsége a programunkba is könnyen beépíthető.
Elterjedtek azonban prímszámkereső algoritmusok is egyik legismertebb módszer az **Eratoszthenész szitája** (Kr. e. 200 körül). Ezt az algoritmust például papíron a legegyszerűbb használni. Írjuk fel kettőtől a számokat a maximum értékig és tekintsük első prímszámunknak a kettőt és húzzuk ki a listáról mindazokat a számokat, amelyek oszthatóak kettővel. Az első ki nem húzott szám lesz a következő prím, ezután húzzuk ki ennek a többszörösét is. Ha ezeket a lépéseket ismételjük, akkor minden nem prímszámot áthúzunk és így megtaláljuk a prímszámokat. Másik egyszerű algoritmus, hogy a háromnál nagyobb számokat megpróbáljuk elosztani az addigi prímszámokkal. Ha valamelyikkel sikerült az osztás, akkor a számot nem tekintjük prímnek [3].
Ma már 17 millió számjegyből álló prímszámok is ismertek így kialakultak a **nevezetes prímszámtípusok** listái. Ma már csoportosítjuk is a prímszámokat, legismertebbek talán az ikerprímek, mely ma is egy megoldatlan probléma. Ikerprímnek tekintjük azokat a prím párokat

melyre igaz, hogy *p* és *p+2* is prím. Ikerprím-sejtés kimondja, hogy végtelen számú ikerprím létezik, de ezt még ma sem sikerült bizonyítani. 2013-ban megjelent egy cikk miszerint áttörést értek el az ikerprím-sejtés bizonyításában [4], de még így is messze vagyunk a két eltéréstől. Jitang *„Csang azt bizonyította be, hogy végtelen sok olyan prímszámpár létezik, amelyek között a különbség kevesebb, mint 70 millió. Bár a 70 millió nagyon soknak tűnik a 2-höz képest, valójában nagyon nagy eredmény, mert egy véges szám."*
A minél nagyobb prímszámok keresése még a mai napig nem állt meg, de mi szükségünk van manapság a prímszámokra?
Ma rendkívüli **nagy prímszámokat használnak nyílt kulcsú titkosítási algoritmusokban**. Ebben a kriptográfiai eljárásban két, egymással szorosan összefüggő kulccsal rendelkezik a felhasználó. Az egyik kulcs a nyilvános, mindenki számára hozzáférhető, míg a másik a titkos. Fontos, hogy a nyilvános kulcsból ne lehessen visszafejteni a titkosat. A nyilvános kulcsot el lehet terjeszteni széles körben és bárki kódolhat vele üzenetet. De kibontani viszont csak a fogadó saját titkos kulcsával lehet és az ilyen módon kódolt üzenetet a feladó sem tudja dekódolni [5].
A fenti elvi alapokra építve 1976-ban Ron Rivest, Adi Shamir és Len Adleman fejlesztette ki az **RSA-eljárás**t, mely ma is az egyik leggyakrabban használt titkosítási algoritmus [6]. Az eljárás lényege, hogy válaszunk két nagy prímszámot, legyen ez *p* és *q*, szorzatuk pedig *n*. Válaszunk még egy *n*-nél kisebb *e* számot. Keressük meg *e* inverzét *d*-t, erre teljesül *ed=mod(p-1)(q-1)*. Így már *p* és *d* nem szükséges, semmisítsük meg. Ilyenkor a nyilvános kulcsunk az *e* és *n* számpár, titkos a *d* és *n* számpár [7].
Az 1980-as években az **elektronikus aláírás** megjelenésével szükségessé vált, hogy bármilyen hosszú adatot adott hosszúságúra lehessen leképezni. Ezt a technológiát nevezzük **hash függvény**nek.
Az SHA egyik legelterjedtebb **kriptográfiai hasító algoritmus**. Ennek első változatát 1993-ban készítették NIST és NSA felügyelete alatt. 2005-ben felfedezték, hogy ütközés fordulhat elő (két egyforma SHA hash értékű fájl) ezért ma már nem használják, bejelentette a NIST, hogy 2010-ig át kell térni a biztonságosabb SHA-2-re. A több változatot, hosszabb hash értékkel működő algoritmust 2002-ben mutatta be a NIST, SHA-2 néven. Három, bithossz szerint különböző változat terjedt el ezek: SHA-256; SHA-348; SHA-512 [8].
A prímszámokat a kriptográfiától eltérő területen is használják, például **véletlenszámok előállítása**. Létezik hardveres megvalósítás és szoftveres álvéletlenszám-generátor. Működésében a kiindulási számot megszorozzuk egy *a* számmal, majd ezt elosztjuk egy *b* számmal, ahol *a* és *b* relatív prímek. Az osztás maradékát tekintjük az új kiindulási értéknek. A lépések tetszőlegesen ismételhetők.

Párhuzamosítás

Számítástechnikai értelemben akkor beszélünk párhuzamosításról, mikor adott feldolgozásra váró feladatokat úgy bontjuk részekre, hogy azok egyszerre, egymástól függetlenül feldolgozható legyen. Ehhez olyan feladat szükséges, amely felbontható ily módon. Mi olyan módon valósítjuk meg tesszük ezt a szétbontást, hogy különböző szálaknak kiosztott intervallumok számtartományokba eső számok diszjunktak legyenek. Mivel a hardver önmagában nem elég a párhuzamos folyamat, ill. műveletek elvégzésére, ezért a párhuzamosítás szoftveres környezetben vett értelmezéseit is szemügyre kell vennünk. Továbbá ki kell térnünk azokra az akadályokra, amely párhuzamos tevékenységeknél szóba jöhetnek.

Hardveres megoldások

A párhuzamosításnak sok formája létezik, és többféle osztályozási rendszert alkothatunk rá, például Flynn-féle architektúrális osztályozás. Négy különböző architektúrájú számítógép típust tudunk definiálni. Ezek adatfolyam és utasításfolyam szerinti csoportosítások, röviden: **SISD, SIMD, MISD és MIMD architektúrájú számítógépek** [9].

A SISD (Single Instruction Single Data) architektúrájú gépek esetében nem beszélhetünk párhuzamosításról, mivel egyetlen adatfolyam csak egy utasításfolyam dolgozhat. Nyilván ilyen esetben nem dolgozhatunk fel több adatot egyszerre egy időben.
A SIMD (Single Instruction Multiple Data) esetén viszont beszélhetünk erről, hiszen ezek a számítógépek jellemzően több adatfolyamon hajtanak végre egy utasítás folyamot. Tehát igazolhatjuk, hogy párhuzamos műveletvégzésre alkalmas. Ilyenek például azok a gépek, amik több aritmetikai és logikai műveletvégző egységet (ALU) tartalmaznak. Ezek két altípussal rendelkeznek, a közös tárolójú (shared memory) gépek, valamint az osztott tárolóval (distributed memory) rendelkező gépek.
A MISD (Multiple Instruction Single Data) architektúrájú gépek jellemzői, hogy több utasításfolyammal egyetlen adatfolyamot dolgoz fel.
A MIMD (Multiple Instruction Multiple Data) architektúrájú gépek képesek több adatfolyam feldolgozására több utasításfolyam szerint. Jellemzően ilyenek a többprocesszoros rendszerek. Itt szintén valójában beszélhetünk párhuzamos adatfeldolgozásról, amik egymástól függetlenül mehetnek végbe.
Megemlíthetjük azt is hogy többprocesszoros rendszereknél megkülönböztetünk **szimmetrikus és aszimmetrikus felépítésű rendszerek**et is. Szimmetrikus rendszerek (SMP - Symmetric MultiProcessing) esetében a processzorok egyenrangúak egymáshoz viszonyítva, a rajtuk futtatott operációs rendszer egységesen mindegyik processzorra rá tudja bízni a kiszabott feladatot. Aszimmetrikus rendszerek esetében nincs egyenrangúság a processzorokat illetően, egyik vélhetően magasabb szintet fog képviselni a másiknál [9].
Bárhogyan is legyen a több processzor csak korlátozott számban áll majd rendelkezésünkre. Előfordulhat, hogy nem áll rendelkezésünkre több processzor. Ilyenkor pár különböző technológia közül választhatunk, párhuzamosítás szempontjából.
Egyik nagyobb segítségünk a **pipeline** (futószalag) szervezés. Ez azt jelenti, hogy processzoron belül várakozó folyamatokat, úgy tudjuk párhuzamosítani, hogy az elemi gépi utasítások feldolgozását különböző párhuzamosan működtethető alrendszerek tudják feldolgozni és végrehajtani [9]. Továbbá lehetőségünk van **szuperskalár processzorok**at is használni, amelyek megvalósításánál a processzorokban található végrehajtó hardveregységeket is megtöbbszörözik. Így több futószalagot kapunk, továbbá jellemzőjük, hogy egy gépi ciklus alatt több utasítást is képesek elvégezni. Ezek kihasználása érdekében bevezették a **többszálas feldolgozás**t. Így biztosított a folyamatok szálainak párhuzamos futása [9].

Szoftveres megoldások

Szoftveres értelemben a folyamatoknak nevezzük azokat a futás alatt lévő programokat, amelyek már az operációs rendszer által kaptak folyamatleíró blokkot (PCB - Process Control Block), amellyel az operációs rendszer általi ütemezők képesek őt azonosítani. A **folyamatok** ezek után erőforrásokat kaphatnak meg és szabadíthatnak fel. Különbséget teszünk megosztható és nem megosztható, valamint nem megszakítható (non-preemptív) és megszakítható (preemptív) erőforrások között. Megosztható erőforrás például a processzor és a memória, ugyanakkor nem megosztható például a nyomtató [10].
Az erőforrások esetében a rendszermag **erőforrás-kezelő** része fog majd gondoskodni a folyamatunk erőforráshoz jutásáról. Az erőforrás-kezelő fog dönteni arról, hogy az adott folyamatunk a keletkező igények szerint erőforráshoz juthat-e vagy sem. Ha az erőforrás-kezelő úgy találja, hogy futhat a folyamatunk, akkor erőforrást rendel hozzá. A folyamatunk lefutása után az erőforrás kezelő felszabadítja a folyamat által használt erőforrást, és ahhoz hozzá férhet a többi folyamat is. Ha nem áll rendelkezésünkre erőforrás, akkor a folyamatunk várakozásra kényszerül, vagyis várólistára kerül, amíg az erőforrás fel nem szabadul [10]. Ha van szabad erőforrásunk, adunk a folyamatunknak, ha nincs, akkor várakozni kényszerül a folyamatunk. Ez viszont felvethet pár akadályt.

Egyik esetben a **kiéheztetés** veszélye kerül elénk. Ez olyan esetben jön létre, ha nem vagyunk biztosak abban, hogy egy adott folyamat mikor juthat hozzá az éppen neki szükséges erőforrásra. Ez erőforrás felszabadítás után is felléphet, ha egy prioritás alapján működő rendszerben folytonosan megelőzik az adott folyamatot.
A másik esetben a **holtpont** kialakulásának a veszélye áll fenn [10]. A holtpont nem más, mint amikor: *„Több folyamat egy vagy több olyan erőforrás felszabadulásra vár, amit csak egy ugyancsak várakozó folyamat tudna előidézni.”* Ez azt jelenti, hogy egyik folyamatnak sem lehet adni erőforrást, mivel mindkét folyamat egymás erőforrásaira vár.
Ezt kétféle módszerrel lehet kezelni egyik a **holtpont kialakulásának megelőzése**, a másik a **holtpont felszámolása**. A holtpont kialakulásának alapvetően négy alapfeltétele van, ezeket nem részletezzük. Időnként futhat a holtpont detektálásra alkalmas algoritmus. Ezt az operációs rendszer erőforrás kiosztásának és a felmerülő erőforrás igények nyilvántartása alapján vizsgáljuk meg. A detektálás után a holtpontok megszüntetését csak a folyamatok feláldozásával (leállításával) lehet megtenni, ezt a folyamatot nem részletezzük.
További akadály még a nem megosztható, egymást kizáró (mutual exclusion) erőforrásoknál következhet be, amelyeket több folyamat szeretne használni. Ilyenkor a kritikus erőforrást **szemaforok** segítségével tudjuk lezárni és engedélyezni a folyamatok előtt [10]. Így elkerülhetőek különböző például adatütközések, és a **versenyhelyzetek**. A versenyhelyzetnek azok az eseteket nevezzük, mikor egymástól független folyamatoknak az eltérő végrehajtási idejétől függően megváltozik a folyamatokhoz tartozó kimeneti eredményünk [11], amit nyilván nem engedhetjük meg.
A folyamatainkat jelentősen fel tudjuk gyorsítani, ha párhuzamosan végezzük. Az egyik csoportot a konkurens, egymással (erőforrásért) versengő folyamatot alkotják. A másik csoport kooperatív folyamatok halmaza, az erőforrás megszerzés mellett még az általuk elért közös cél is összeköti [10].
A **folyamatok szinkronizálása** alapvetően három módon lehetséges:

- kölcsönös kizárás, ahol egy időben egy tetszőleges folyamat futhat,
- folyamatok várakozása, majd a végrehajtásukat követően azok szétválása (ún. randevú),
- precedencia szerint, ahol egyes feladatok sorrendisége nem lehet tetszőleges.

Azoknál az utasításoknál, amelyek nem végezhetőek el adott tetszőleges sorrendben, általában valamilyen módon meg kell jelölnünk azokat az utasításokat, melyeket párhuzamosan végezhetünk. Egyik jól ismert példája a **fork-join utasítások**. A fork (szétágazás) utasítással azt a programrészletet fogjuk ellátni, amelyek párhuzamosan, egymástól tetszőleges sorrendben végezhetőek. Ha egy olyan utasítás részlethez érünk, ahol ez már nem igaz, akkor a join (egyesítés) paranccsal kell össze kapcsolnunk a szétágazott utasításokat [10].
Ezt az analógiát fogják követni majd a létrehozott szálaink is. A szálak ugyanakkor tekinthetőek folyamatnak.

Szálkiosztás stratégiája
Programunkban hét fajta szálkiosztást valósítunk meg. Célunk egy olyan arányelosztás volt, hogy a szálak körülbelül egyszerre végezzenek, így megtalálva az optimális kiosztást. Mivel a prímtesztelő algoritmusunk a legegyszerűbb, naiv módszerre alapszik, ezért nagy számok esetében jelentősen megnövekszik az időigénye, hogy eldöntse egy számról, hogy prím-e.
A **szál nélküli módszer** esetén csak a főszálon fut tovább a program, nincs benne semmi féle szálkezelés. Így a teljes intervallumot egy szál fogja számolni, tehát a leglassabb eredményt fogja hozni, viszont a legegyszerűbb leprogramozni. Ez kiinduló pontot, referenciát ad a többi módszerrel való összehasonlításra.
A **normál módszer** esetén az intervallumot felbontjuk egyforma egységekre és így minden szál ugyanannyi számot kap. A programozásban itt már kell majd figyelnünk a szálkezelésre és az azzal felmerülő problémákra, programozása így bonyolultabb, de a mögöttes matematika még így is

egyszerű. Arra számítunk, hogy gyorsabb lesz a szálak miatt az algoritmus, mint szál nélkül, de valószínű, nem a legjobb, mivel minden szál egyforma egységet kap.

A **hatvány módszer** a kettő hatványait használjuk ki és eszerint bontjuk fel az intervallumot. Így már a tartományok nem egyforma nagyságúak, nagyobb számok esetén egy szál kevesebb számot fog kapni, hogy a prímtesztünk döntést hozzon. Programozási szempontból nem nehezebb, mint a normál, de a matematika komolyabb mögötte. Várakozásunk szerint eddigi leggyorsabb algoritmus lesz.

A **Fibonacci módszer** az intervallumot a Fibonacci sorozat elemei szerint bontja fel. Fibonacci sorozat elemei egyszerűen előállíthatók [12]. A szükséges Fibonacci számokat majd a szálkiosztás előtt kell kiszámolnunk, de ez gyorsan megtörténik, mert nincs szükségünk sok számra. Mindig annyi kell, ahány szálon kívánjuk futtatni a feladatot. Így egy-egy szál futási időtartalmát nem fogják lassítani ezek a számítások, de a módszer teljes futási idejében már benne lesz. (A Fibonacci számok számítási idejét nem kell mérnünk külön). Ha a számítást kihelyezzük egy függvénybe, akkor egy tömbben adhatja majd vissza a kiszámolt számokat, amelyből képezzük az összegüket, majd a szálak a számoknak megfelelően kapják az értékeket. Tehát itt is minél nagyobb számok szerepelnek az adott tartományban, annál kevesebbet kap a szál. Programozási szempontból ez sem nehezebb, mint a többi szálkezelős módszer, főleg. A matematika itt a legbonyolultabb, de ez sem okozhat problémát. Mivel itt nem exponenciálisan növekednek a számok egy picivel jobb eredményre számítunk, mint az előző algoritmusoknál.

A **Pascal háromszög módszert** háromféleképpen is alkalmaztuk. A példányosítás során így szükségünk lesz egy azonosító értékre is, amely eldönti, hogy a három módszer közül melyik fusson. A tartománykiosztás a Pascal háromszög számaiból építkezik. Számítása egyszerű módon történik: a nulladik sorba beírunk egy egyest, a következő sorok elemeit pedig úgy kapjuk meg, hogy összeadjuk a felette balra és jobbra található számokat. Ha ott nem találunk számot, akkor pedig nullának tekintjük [13].

A programunkban itt is érdemes lesz egy külön függvényt írni, amely a számításért fog felelni. Legegyszerűbben egy többdimenziós tömb válhat segítségünkre. Minden nulladik elemet, az egyszerűség kedvéért egyel töltünk fel és a sor utolsó oszlopát is, ami megegyezik az adott sor sorszámával. A számítás után, ha egy tömbbel térünk vissza itt is jó megoldásnak tűnik. Innen már hasonló, mint az előző metódus.

Három változatra azért volt szükség, mert a sima nem megfelelő mivel a két szélsőséges tartomány egyforma mennyiségű számot fog kapni, tehát arra számítunk, hogy lassabb lesz. Ezt neveztük el **Pascal normál** módszernek.

Ezért gondoltunk arra, hogy növekvő sorrendbe helyezzük a számokat, de így a két legkisebb tartomány egyforma értéket fog kapni, és még további ismétlődések is előfordulhatnak. Tehát így sem érhetünk el olyan jó eredményeket, mint eddigi módszerekkel. De az előző fajtánál jobbnak kell lennie, ezért a **Pascal optimális** nevet kapta.

A harmadik változatot úgy terveztük meg, hogy több sort számolunk ki (szál * 2 – 1 db sor) és ez a sor a Pascal háromszög egy páratlan sorára fog mutatni, ezért a neve **Pascal páratlan**. Aminek a felére van szükségünk, így minden tartomány más mennyiségű számot fog kapni. Várakozásaink szerint ez fog a leggyorsabb metódusnak bizonyulni.

A könnyebb áttekinthetőség végett az 1. ábra segítségével kívánjuk ismertetni öt szál esetén a kiosztott tartományokat egyes szálakra levetítve. A teljes intervallum 1-10000-ig terjed. A szál nélküli módszert kihagytuk a táblázatból.

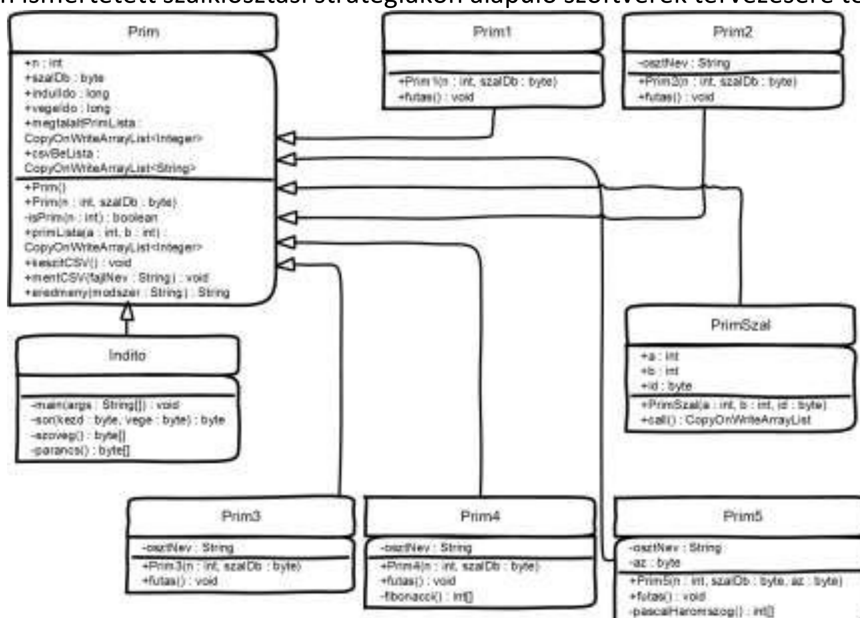
A metódusok leírásából látszik, hogy egyre jobb eredményre számítunk az újabb algoritmusokkal. Ezen kívül még a megfelelő szál szám meghatározása is fontos lehet, ez architektúra függő, de egy szintig növelhető és gyorsabb lesz az adott algoritmus. Minél több szál, annál nagyobb költség – azaz adminisztrációs és/vagy koordinációs veszteség –, így nagyon sok szál már lassú lesz. Természetesen olyan kijelentést sem tehetünk, hogy négy szál négyszer gyorsabb, mint egy szál.

Szál száma	Normál	Hatvány	Fibonacci	Pascal normál	Pascal optimális	Pascal páratlan
1.	1-2000	1-5161	1-4166	1-625	1-3750	1-4294
2.	2001-4000	5162-7741	4167-6666	626-3125	3751-6250	4295-7730
3.	4001-6000	7742-9031	6667-8333	3126-6875	6251-8750	7731-9447
4.	6001-8000	9032-9676	8334-9166	6876-9375	8751-9375	9448-9938
5.	8001-10000	9677-10000	9167-10000	9376-10000	9376-10000	9939-10000

1. ábra: Intervallumok felosztása különböző szálkiosztó stratégiák esetén

Tervezés

A 4. fejezetben ismertetett szálkiosztási stratégiákon alapuló szoftverek tervezésére térünk ki.



2. ábra: Koncepcionális osztálydiagram a Java implementációhoz

A tervezés során ötféle módszert különböztetünk meg, ezeknek az algoritmusai mind egy-egy osztályba kerültek, amelyeknek a közös ősosztálya a Prim. Ebbe az osztályba tartoznak azok a függvények, amelyeket minden osztály használ, illetve található benne egy csak az osztályból elérhető (private) függvény is. A 2. ábrán UML osztálydiagram [14] segítségével koncepcionálisan ismertetjük az alkalmazás felépítését. Az ábrát a MS Visio szoftverrel állítottuk elő [15].
Röviden kívánjuk bemutatni a Prim osztály függvények feladatát:

- isPrim(): naiv módszerrel eldönti egy adott számról, hogy prímszám-e,
- primLista(): a beérkező tartományból azokat az értékeket tárolja egy ArrayList segítségével, amelyekre az isPrim() igaz értékeket adott,
- keszitCSV(): a leendő CSV úgynevezett összefoglaló sorát állítja elő, ezt a 7.2. fejezetben ismertetjük,
- mentCSV(): a csvBeLista ArrayList tartalmát menti el a számítógépen CSV fájlformátumban,
- eredmeny(): a konzolra írja ki minden osztály lefutása után az eredményeket.

A Prim osztálydiagramjából látható, hogy a konstruktor két értékkel dolgozik. Az n az intervallum legnagyobb számát, míg a szalDb a szálak számát tartalmazza. Ezeket az értékeket az leszármaztatott osztályok példányosítása során kell definiálnunk. Ezért külön készült egy vezérlő funkciójú Indito nevű osztály is.

Az Indito nevű osztály feladata, hogy könnyű elindíthatóságot biztosítson a felhasználók számára. Jelenleg parancssorból működik a programunk, melyekben kódszámok alapján tudjuk a funkciókat kiválasztani, de erről a program a futás közben tájékoztat. Figyelünk a felmerülő problémák lekezelésére is (érvénytelen parancsszámok, más típusú beviteli adat), így hibakezelés is van benne, mivel a felhasználóval végül is ez az osztály tartja a fő kapcsolatot.
A program elindulása után a felhasználónak ki kell választania a futtatandó metódust, azután az intervallumot is meg kell határoznia (három lehetősége van). Ha nem a szál nélküli metódust választotta, akkor meghatározhatja a maximális szál számot (12 db). A későbbiekben eddig az értékig fogja a program kiosztani a szálak számát, egytől kezdődően, ciklussal inkrementálva. Ha ez is megvan, akkor pedig eldönthetjük, hogy a programunk az eredményeket tárolja-e egy CSV fájlban.
Miután ezzel is megvagyunk, a program elindul és konzolon értesíti a felhasználót minden lényeges eredményről, még egy-egy szál részeredményét is megjelenítjük. A könnyebb érthetőség kedvéért egy képernyőképet láthatunk a 4. ábrán. A megtalált/kiválogatott prímszámokat nem íratjuk ki a konzolra, mert a hangsúlyt a megfelelő szálkezelés megvalósítására helyeztük.
Ha a program lefutott, akkor ismét van lehetőségünk futtatni akár egy újabb metódust is, mintha újraindult volna a program.
Több programozási nyelven is implementáltuk a programot, hogy megtaláljuk az igazán gyors megoldást, de a fenti koncepció igaz mindegyik implementációra.

Implementáció
A fenti tervet négy ismert programozási nyelven valósítottuk meg abból a célból, hogy megállapítsuk melyik programozási nyelven implementált megoldás lesz a leggyorsabb. Mindegyik implementációnál törekszünk a teljes hasonlóságra.

Java nyelvű megvalósítás
A Java nyelvű implementáció egy részét a konzulensünktől kaptuk meg, mint kiindulási probléma és azt fejlesztettük tovább, illetve ebből készült a többi megvalósítás is.
Fejlesztési környezetként a NetBeans IDE szoftvert használtuk [16]. A nyelvnek a JDK 7-es verziójában [17] történt a programozás, néhány új függvény alkalmazhatósága miatt.
A fejlesztés a nyelv alapos ismeretét megköveteli, mivel rengeteg olyan eljárás előjön, ami csak gyakorlattal és mély ismerettel oldható meg. A program fejlesztése az objektumorientált szoftverfejlesztés elvei szerint történt. A forráskódban megtalálható az öröklés és így kialakult ős-utód viszony a függvények hozzáférésének módosításával (public, private), illetve a ősosztály változóihoz való hozzáférés (super). Így szükségessé vált, hogy ismert legyen a példányosítás is, ami miatt például egyik változónkat (csvBeLista) statikussá (static) kell tenni, hogy az adott szálkiosztó stratégia/módszer összes eredményét ki tudjuk írni a fájlba. Illetve konstruktorokat is használunk, így előkerül a példányra mutató jelző is (this).
Egy-egy feladatot metódusok látnak el, ezek vagy egy előre meghatározott fajtájú értékkel térnek vissza (return) vagy void típusúak.
Kivételkezelés is számos helyen előfordul a programban, és azon problémák javítását is megkísérli a program (akár a felhasználó bevonásával), ha van értelme.
Ami hiányzik: a program nem elosztott, ezért nem a Java EE technikán alapul, belépési pontnak így elég a public static void main() függvény. Adatbázis-kezelés sem szerepel a programban. Interfészre sincs szükségünk jelenleg, ahogyan getter(), setter() függvények használatára sem. Függvény felülírása is csak egy pontban valósul meg (@Override). Grafikus megjelenítésre (GUI) nincs igény.

Igyekeztünk megfelelően használni a primitív adattípusokat, például a szalDb (a szálak száma) nevű változó byte típusú, mivel 127 szálat a program indítója sem enged. Nem szükséges mindenhol például az alapértelmezett int egész típus.

A szálkezelést az ExecutorService osztály végzi, mely alapértelmezetten megtalálható a JDK 6-os verziójától. Optimalizáltsága végett ez a Java nyelv jelenleg leghatékonyabb szálkezelésért felelős csomagjának része. Mivel több szálon fut a program, ezért a sima ArrayList a program megfelelő, hibátlan működése miatt nem elegendő, mivel így ha egyik szál beleír, a többi észleli az adatszerkezet megváltozását (mivel Observable), ezért kivételt dobhat futás közben (java.util.ConcurrentModificationException). Ha a kivételt el is kapnánk, akkor is hibás eredményt kapnánk. Ezt elkerülendő, a több szál konkurens írási műveletét is támogató (szálbiztos) kiegészített CopyOnWriteArrayList-et használjuk, ami a JDK 7-es verzió óta áll rendelkezésünkre. A szál futási adatainak konzolra való kiíratásáért a CopyOnWriteArrayList call() nevű felülírt függvénye felel.

C# nyelvű megvalósítás

A C# nyelv nagy hasonlóságot mutat a Java nyelvvel. Ezért a fejlesztése viszonylag gyorsan történt. Fejlesztői környezetként a Visual Studio szoftvert [18], programozási nyelvként a .NET Framework 4.5-ös verzióját használtuk [19], de a program a 4.0-ás verzióval is működne. Amiért mégis a 4.5-ös verzió mellett döntöttem mert a Task Paralell Library-n komoly fejlesztéseket végeztek el, de a kódolásban mégsem kellett változtatni, de gyorsabb végrehajtás figyelhető meg [20].

A fejlesztés a Javához hasonlóan a nyelv alapos ismeretét mégis megköveteli, mert rengeteg helyen kellett a kódhoz hozzányúlni. A programozás során figyeltem a már kialakított osztályszerkezetre és igyekeztem tartani, de néhol apróbb eltérések figyelhetőek meg, így ez az implementáció is megfelel az objektumorientált szoftverfejlesztés szemléletének. Csupán a Java implementációtól való eltéréseit kívánjuk röviden bemutatni.

A C# nyelvben az ArrayList szálbiztos, így nem használunk kimondott konkurens gyűjteményt, bár a nyelv lehetőséget biztosít rá.

Eltérés figyelhető még meg az időmérésnél is. A C#-hoz egy rövid függvényt kellet írni, amely hasonlóan működik, mint a Javás currentTimeMillis(). A Prim osztályban helyeztem el ezt a függvényt és javaIdo() nevet kapta [21].

Mivel sima ArrayList-et használunk, ezért a PrimSzal osztály megtalálható a Prim2-ben és a szerepe is ugyan az, de nem implementálja a Callable interfészt és így nem szükséges a call() függvény felülírása az utódosztályban. A hasonlóság kedvéért itt is a call() nevű függvény látja el a PrimSzal nevű osztályban a szálak eredményének kiíratását.

Következő eltérésünk a szálkezelésben mutatkozik meg. Az első változatban az egyszerű Thread család függvényeit használtam, azután kiderült, hogy a Task-ok használatával [22] hasonló eredményre juthatunk, mint a Javás ExecutorService. Ezért megpróbáltuk és így gyorsabbnak bizonyult a megvalósítás. A Task Parallel Library segítségével nagyon egyszerű párhuzamos alkalmazást készítenünk, de néha szükség lehet, hogy szabályozzuk a működésüket. Ha ez nem történik meg, akkor az ütemező általában virtuális szálon próbálja meg a feladatot megoldani, ami méréseink szerint rövid intervallum esetén még gyorsabb is (például: [1, 1000] zárt intervallum), mintha jelezném az ütemezőnek, hogy külön szálat kapjon a folyamat. A hasonlóság kedvéért ez a jelzés mindenhol megtörténik, amely nagyon egyszerű: az indításnál még a TaskCreationOptions.LongRunning értéket is át kell adni a StartNew() függvénynek. Így ezzel már látványos javulást értünk el a Thread-hez képest, illetve a kezelhetősége is egyszerűbb és jobban hasonlít a Java implementációhoz, de a sebessége még így is elmaradt a Javás megvalósításhoz képest.

C++ nyelvű megvalósítás

Az implementáció elkészítéséhez a Code::Blocks nevű fejlesztőkörnyezetet használtuk [23]. A programozási nyelv tekintetében a C++ a 2011-es ISO szabványba (ISO/IEC 14882:2011) is belefoglalt verzióját használtuk [24]. Erre azért volt szükség, mert megtalálhatóak benne a 2007-es tervezetben már felvetett, de akkor még nem szabványosított szálkezeléshez kapcsolódó osztálykönyvtárak is (így

tehát a szálkezelés viszonylag csak a 2011-es szabványtól elérhető). Rengeteg előzetes tapasztalatot és még ennél is több szakirodalmi kutatási munkát igényel a nyelv elsajátítása, legfőbbképp az új elemek helyes használata.
Mivel a Java implementációnak próbál a tükörképe lenni – bár ez teljes egészében nem valósulhat meg – ezért az osztályosítás, illetve származtatásnak is nagyon hasonlónak kell lennie. A többi implementáció közötti nagy különbségek egyike, hogy egyszerre kell úgynevezett .h (header) állományokkal és magával a programkódot tartalmazó, függvényeket és változókat kifejtő .cpp állományokkal dolgoznunk. A header fájlok lesznek a rendeltetésszerű helyei az általunk deklarálni kívánt függvényeinknek és változóinknak, itt kell felsorolni a változók neveit, típusait, azok elérhetőségét más osztályok számára (public, private). Továbbá öröklődésnél és származtatásnál is ezeket az állományokat kell figyelembe vennünk. Valamint a .cpp állományokban jelezni kell, és ezzel hozzá kell kapcsolnunk (#include) a header állományokat a .cpp állományunkhoz. Ez azért is célszerű kihasználni, mert a az álalunk használt GNU GCC fordítóprogramunk relatív gyorsan tud fordítani egyszerűbb változtatásoknál. Ezenkívül megfelelő használatukkal áttekinthetőbbé tehetjük a forráskódunkat. Hátránya csupán csak annyi, hogy körültekintőbb szerkesztést igényel, hiszen a változók neveinek, függvények típusainak és azok neveinek meg kell egyeznie mindkét állományban.
A szálkezelés legegyszerűbb eléréséhez a <future> osztálykönyvtárat használtuk fel. Azon belül is az async függvénnyel való szálindítást követően annak visszatérési értékét (eredményeit) egy future típusú a C++ környezetben megszokott vector adatszerkezetben kapjuk vissza [25].
Ennek az a legfőbb tulajdonsága, hogy képes egymástól függetlenül írás és olvasási műveletet végezni és az adott függvény által visszaadott eredményeket eltárolni. Arra is lehetőségünk nyílna a paraméterek használatával, hogy a szálak egymás után (launch::deffered) indul¬janak, esetleg elinduljanak egymástól függetlenül (launch::async) vagy a kettő lehetőség között automatikusan szálkezelő szerint választva induljanak el (utóbbi esetben nem kell paraméterként megadni ezt, vagy beírhatjuk mindkét paramétert egymás után).
Továbbá különbség, hogy azt az idő függvényt, amely a szálak futási idejét hivatott mérni, nem volt lehetőségem 1970. január 1-től számolni milliszekundumokban UTC formátumban. Ennek oka valószínűleg az osztályokon keresztül való hívás okolható. Ezért alternatívaként a program indulási idejét vesszük figyelembe, ugyanúgy milliszekundumokban mérve. Ez nekünk megfelelő referenciabázis lesz, mivel csak a szálak viszonylagos indulási és végződő időpontjait vesszük figyelembe az eredmények kimutatásánál.

PHP nyelvű megvalósítás

A PHP nyelv nagyobb eltérést mutat, mint az eddigiek. Sok hasonlóság is felfedezhető bennük, az egymásra gyakorolt hatásuknak köszönhetően. De nem azonos a nyelv célja az eddigi nyelvekéivel, ezért más megközelítést kíván a nyelv. Fejlesztését elkezdtük, noha tudtuk, hogy a jelenlegi verzió nem támogatja a szálkezelést (sőt egyelőre nem is tervezik), bár béta csomag elérhető hozzá az interneten, de azt sem a nyelv készítői publikálták. Így csak a Prim és Prim1 osztályt készítettem el. Kimondott fejlesztői környezetet nem használtunk hozzá, noha például a NetBeansben is lehet fejleszteni, de nekem egyelőre azt ezt lehetővé tevő plugin még nem tűnik elég okosnak benne. A hiányosságok miatt maradtunk a Notepad++ mellett [26]. A PHP 5-ös verzióját használtuk főleg azért, mert ettől a verziótól érhető el az objektum orientált szemléletmód. Ezért a Prim és Prim1 osztály egymáshoz való viszonyában nem mutat különbséget.
Mivel egy osztály készült el, ezért nem éreztük szükségesnek az Indito osztály létrehozását. Jelenleg maga a Prim1 indítja el a feladatot. A PHP nyelv segítségével a szerverek gyorsasága úgymond tesztelhető, de a hosszú számítási igény miatt a set_time_limit() függvénnyel több időt kell hagyni az alapértelmezettnél (szerverenként eltérő). Illetve még lehetséges a php.ini fájl módosítása is, ez akár egy .php fájlból is megtehető vagy az .override fájl használata is segítséget nyújthat. Persze, sajnos nem minden szerver engedi a maximális futási idő meghosszabbítását, elsősorban biztonsági és teljesítménybeli okok miatt.

Helyi számítógépen PHP programot a legegyszerűbben a WampServer [27] környezetet használva tudunk. Telepítés után a wamp mappa www mappájába belemásolva a fájlokat elindíthatjuk azokat a böngészőnk segítségével a localhost-ról. Így lehetőség adódik összehasonlítani a többi nyelvvel, mert azonos hardveren fut a teszt. A PHP interpreter alapú nyelv, így nagyságrendekkel hosszabb futásidő szükséges neki.
A lassúság hátránya miatt mégis elég nagy szerepe van a nyelvnek a mai piaci helyzetben és a fejlesztése is még mindig folyamatos. Legegyszerűbb és így nagyon elterjedt, hogy egy weboldal dinamikus részét PHP nyelven állítsuk elő és kis számítási feladatok mellett ez nagyon gyorsan meg is történik. És ha figyelembe vesszük, hogy a PHP szerver oldali nyelv, és egy szerver nagyságrendekkel több erőforrással rendelkezhet a klienseknél, így ez a sebességkülönbség többnyire kompenzálható éles környezetben.

Teszteredmények

A programunk megvalósult mind a négy nyelven, így a futtatásokból eredményeink keletkeztek. A 3. ábrán például szál nélkül 10000-ig válogatjuk a prímszámokat, e módszer futási eredményei láthatóak.

Programozási nyelv	Java	C#	C++	PHP
Futási idő (ms)	479	774	397	25265

3. ábra: Futási eredmények szálkezelés nélküli módszernél

Ez a fejezet az elért eredményeinket kívánja ismertetni, kitérünk a szoftverünk működése közben szolgáltatott eredményekre, fontosnak tartottuk, hogy a mentett adatok felépítésére is kitérjünk. Ezek az adatok emberi szemnek átláthatatlannak tűnhetnek, ezért időközben elkészült az eredmények grafikus reprezentációja is.

Eredmények a képernyőn

A programunk konzolos kérdés-kóddal/felelek menürendszerrel működik. Indulás után pedig folyamatosan értesíti a felhasználót a lefutott szálak fontosabb paramétereiről (4. ábra). Azért döntöttünk a hagyományos megjelenítés mellett, mert jelen feladatukban a programozási nyelvek elsődleges feladata az adatok szolgáltatása és így nem láttuk szükségesnek egy grafikus felhasználói felület (GUI) megtervezését és elkészítését minden nyelven. Megjelenítésre pedig azért van szükség, mert a folyamatok nagyon időigényesek és lehetőséget kell biztosítanunk a felhasználónak, hogy paraméterezze a megoldandó feladatokat és futás közben azok állapotáról is információt szerezzen.

```
Kérem, válassza ki a C# futtatandó metódust:
1 -> Szál nélküli
2 -> Normál
3 -> Hatvány
4 -> Fibonacci
5 -> Pascal normál
6 -> Pascal optimális
7 -> Pascal páratlan
8 -> Kilépés
A futtatandó: 4
Kérem, válassza ki az intervallumot:
1 -> 1 - 1000
2 -> 1 - 10000
3 -> 1 - 100000
TILTVA 4 -> 1 - 1000000 TILTVA
Az intervallum: 2
Kérem, határozza meg a maximális szál számot (1 - 12 közé eshet): 5
Mentsük az eredményeket egy CSV fájlba?
1 -> Igen
2 -> Nem
Mentés: 2

A futtatás indul...
2. szál futott: 12, a: 1, b: 1000, prímDb: 168
1000;168;Prim4;1;22;
2. szál futott: 773, a: 1, b: 10000, prímDb: 1229
10000;1229;Prim4;1;775;
2. szál futott: 2, a: 1, b: 500, prímDb: 95
3. szál futott: 6, a: 501, b: 1000, prímDb: 73
1000;168;Prim4;2;7;
2. szál futott: 199, a: 1, b: 5000, prímDb: 669
3. szál futott: 589, a: 5001, b: 10000, prímDb: 560
10000;1229;Prim4;2;591;
2. szál futott: 2, a: 1, b: 500, prímDb: 95
3. szál futott: 2, a: 501, b: 750, prímDb: 37
4. szál futott: 4, a: 751, b: 1000, prímDb: 36
1000;168;Prim4;3;5;
2. szál futott: 198, a: 1, b: 5000, prímDb: 669
3. szál futott: 245, a: 5001, b: 7500, prímDb: 281
4. szál futott: 342, a: 7501, b: 10000, prímDb: 279
10000;1229;Prim4;3;343;
2. szál futott: 1, a: 1, b: 428, prímDb: 82
4. szál futott: 2, a: 715, b: 857, prímDb: 21
3. szál futott: 2, a: 429, b: 714, prímDb: 45
4. szál futott: 2, a: 858, b: 1000, prímDb: 20
1000;168;Prim4;4;6;
2. szál futott: 175, a: 1, b: 4285, prímDb: 588
4. szál futott: 186, a: 7143, b: 8571, prímDb: 153
4. szál futott: 226, a: 8572, b: 10000, prímDb: 162
3. szál futott: 341, a: 4286, b: 7142, prímDb: 326
10000;1229;Prim4;4;343;
2. szál futott: 2, a: 1, b: 416, prímDb: 80
3. szál futott: 3, a: 417, b: 666, prímDb: 41
4. szál futott: 3, a: 667, b: 833, prímDb: 24
5. szál futott: 1, a: 834, b: 916, prímDb: 11
5. szál futott: 1, a: 917, b: 1000, prímDb: 12
1000;168;Prim4;5;7;
4. szál futott: 127, a: 8334, b: 9166, prímDb: 91
2. szál futott: 155, a: 1, b: 4166, prímDb: 573
3. szál futott: 212, a: 6667, b: 8333, prímDb: 186
3. szál futott: 217, a: 4167, b: 6666, prímDb: 286
5. szál futott: 125, a: 9167, b: 10000, prímDb: 93
10000;1229;Prim4;5;264;
```

4. ábra: Eredmények a képernyőn (konzolosan)

Eredmények CSV állományban

Eredetileg a program TXT formátumba mentette az eredményeket, ami a későbbi feldolgozás folyamán nagyobb problémát jelentett volna, illetve ahogy bővült a program egyre átláthatatlanabb lett volna. Ezért kerestünk más tárolási módot.

Mivel mérési adatokról volt szó, így táblázatszerű elrendezés mellett döntöttünk. Szóba került a széles körben elterjedt XML formátum is, amit főleg azért vetettünk el, mert átláthatatlan lett volna a felhasználónak, mivel alapértelmezetten nem táblázatként jelenítik meg azt az operációs rendszer. Ezért a szintén ismert szabvány CSV formátum mellett döntöttünk. Szöveghatároló karakternek az idézőjelet (” ”) választottuk, míg szeparátor karakter a pontos vessző (;) lett.

Így ha a számítógépre telepítve van a Microsoft Excel vagy akár az OpenOffice Calc, akkor táblázatban jelennek meg az eredmények és így átlátható lesz még sok szál esetén is. Illetve tesztek miatt tudjuk használni az Excel függvényeket, vagy akár makrókat is írhatunk. Jelenlegi CSV formátumunk a mérésben fontos adatokon kívül még tárolja a megtalált prímszámokat is, tehát ezekben a fájlokban meg lehet tekinteni a számokat, ellenőrzés céljával. A későbbi adatfeldolgozás, és így a következtetések levonása érdekében a többi adat ehhez kapcsolódik. Ha egy feladaton belül minden szál befejezte a feladatát, akkor létrejönnek ún. összefoglaló sorok, ezeket az Összefoglaló (hetedik) oszlop igaz (true) értéke jelzi. A CSV fájl sorainak tartalma az alábbi szerkezetet követi:

- Szál: szálak maximális száma,
- IntervallumTól: az intervallum eleje, ami jelen esetben 1,

- IntervallumIg: az intervallum vége, amit a felhasználó beállított,
- Megtalált prímek száma: a megtalált prímszámok ArrayList mérete, azaz hány prímet talált meg az intervallumban az algoritmus,
- Indulási idő: amikor elindult az adott metódus, azaz a példányosítása megtörtént, UTC szerint eltelt milliszekundumok száma,
- Idő: feladat teljes lefutási ideje milliszekundumban, bele tartozik például a Fibonacci számok kiszámolási ideje is,
- Összefoglaló: jelzi, hogy ez összefoglaló sor-e,
- a megtalált prímszámok felsorolása.

Mivel egyes szálak teljesítményére is kíváncsiak vagyunk, ezért fontos, hogy a CSV-be is mentsük le az eredményeiket is az alábbiak szerint:

- Szál: az adott szál sorszáma, amely 1-től növekszik, de a CSV-ben maximum akkora az értéke, amekkorát a felhasználó engedélyezett; Összefoglaló sorok között nem lehet azonos értékű ennek a cellának a tartalma és maximum akkora lehet, mint az Összefoglaló sor e cellájának az értéke,
- IntervallumTól: a szál intervallumának eleje, ettől az értéktől indul az adott szál vizsgálata; az első szálnak 1 lesz az értéke,
- IntervallumIg: a szál intervallumának vége, ameddig történik az adott szál vizsgálata; az utolsó szálnál az érték megegyezik az Összefoglaló sor e cellájával,
- Megtalált prímek száma: az adott szál által az adott intervallumban megtalált prímek száma; ha összeadjuk ezeket az értékeket az Összefoglaló sorral azonos értéknek kell kapnunk,
- Indulási idő: előzőek szerint UTC-ben,
- Idő: a szál teljes futási ideje milliszekundumban,
- Összefoglaló: cella értéke üres.

Az 5. ábrán egy elkészült CSV-t láthatunk, amit az Excel program nyitott meg.

	A	B	C	D	E	F	G	H	I	J
1	Szál	IntervallumTól	IntervallumIg	Megtalált prímek száma	Indulási idő	Idő	Összefoglaló			
2	1	1	1000	168	1423160000000	5				
3	1	1	1000	168	1423160000000	6	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
4	1	1	10000	1229	1423160000000	489				
5	1	1	10000	1229	1423160000000	489	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
6	1	1	100000	9592	1423160000000	49003				
7	1	1	100000	9592	1423160000000	49005	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
8	1	1	500	95	1423160000000	1				
9	2	501	1000	73	1423160000000	4				
10	2	1	1000	168	1423160000000	5	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
11	1	1	5000	669	1423160000000	124				
12	2	5001	10000	560	1423160000000	364				
13	2	1	10000	1229	1423160000000	365	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
14	1	1	50000	5133	1423160000000	12182				
15	2	50001	100000	4459	1423160000000	36646				
16	2	1	100000	9592	1423160000000	36647	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
17	1	1	500	95	1423160000000	1				
18	2	501	750	37	1423160000000	2				
19	3	751	1000	36	1423160000000	3				
20	3	1	1000	168	1423160000000	4	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
21	1	1	5000	669	1423160000000	123				
22	2	5001	7500	281	1423160000000	153				
23	3	7501	10000	279	1423160000000	218				
24	3	1	10000	1229	1423160000000	219	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
25	1	1	50000	5133	1423160000000	12341				
26	2	50001	75000	2260	1423160000000	15309				
27	3	75001	100000	2199	1423160000000	21467				
28	3	1	100000	9592	1423160000000	21468	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		
29	1	1	428	82	1423160000000	1				
30	3	716	857	21	1423160000000	1				
31	4	858	1000	20	1423160000000	1				
32	2	429	714	45	1423160000000	3				
33	4	1	1000	168	1423160000000	3	true	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29		

5. ábra: Eredmények CSV állományban (Excelben)

Egy módszer egy fájlt hoz létre és abba menti az értékeit, így nem kell feltüntetni a CSV-ben a módszer nevét. Az adott fájl egyértelmű beazonosíthatósága miatt az alábbi elnevezést konvenciót (6. ábra) követjük: Nprime[módszer kódja] [programozási nyelv].csv.

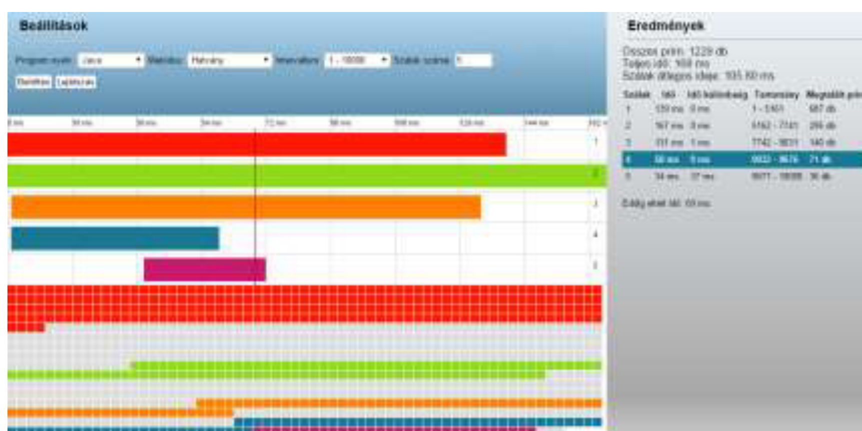
Módszer neve	Szál nélküli	Normál	Hatvány	Fibonacci	Pascal normál	Pascal optimális	Pascal páratlan
Módszer kódja	1	2	3	4	50	51	52

6. ábra: Eredmény állomány elnevezési konvenciójához tartozó kódolás

Például az Nprime3C#.csv fájlnévből kiderül, hogy a hatvány módszer C# programozási nyelven írt mérési eredményeit tartalmazza.

Eredmények grafikus reprezentációja

A programok futtatásából egységes szerkezetű CSV-k keletkeznek. A benne szereplő sok adat miatt ezért nehézkes lehet értelmezni, ezért döntöttünk egy kimondottan csak erre a CSV-k értelmezésére szolgáló program elkészítése mellett. Mivel ennek a szoftvernek már megjelenítési és értelmezési feladatai vannak, ezért ő már kapott egy grafikus felhasználói felületet (GUI). Fontosnak tartottuk, hogy külön ne keljen telepíteni szoftvert, lehető legkisebb legyen a hardver igénye és platform-független legyen, ezért egy internetes alkalmazás mellett döntöttünk, amely akár okostelefonról, tabletről is elérhető. Az üzleti logikát a PHP szolgáltatja, az animációért a JavaScript felelős, megjelenítést pedig a HTML+CSS együtt végzi el.



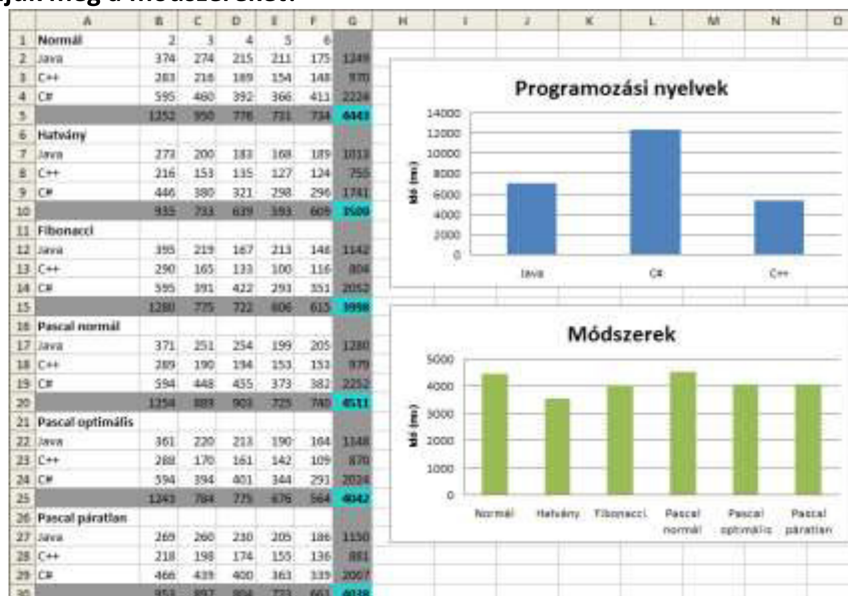
7. ábra: Eredmények grafikus implementációja (böngészőben)

Igyekeztünk úgy kialakítani, hogy könnyű legyen a használhatósága, de minden fontos dolgot beállíthassunk. A beállított adatok szerint pedig akár le is futtathatjuk az animációt a keresésről így a módszer máris látható és áttekinthető. A beállítás hasonlóan zajlik, mint a programjainkban először a programozási nyelvet kell meghatározni, utána a módszert melyet az intervallum követ azután pedig a szálat (7. ábra).

Következtetések

A PHP nyelven implementált program eredményeit nem vesszük figyelembe, mivel nagyon lassú volt, illetve az implementációnál megjegyeztem, hogy nem támogatja a szálkezelést. Tehát elemzésre három nyelv maradt (Java, C#, C++). A programok futása során keletkezett adatokból előállított táblázat és grafikonok a 8. ábrán láthatók.

Először vizsgáljuk meg a módszereket!



	A	B	C	D	E	F	G
1	Normál	2	3	4	5	6	
2	Java	374	274	215	211	175	1249
3	C++	283	216	189	154	148	970
4	C#	595	460	392	366	411	2224
5		1252	950	776	731	734	4443
6	Hatvány						
7	Java	273	200	183	168	189	1013
8	C++	216	153	135	127	124	755
9	C#	446	380	321	298	296	1741
10		935	733	639	593	609	3509
11	Fibonacci						
12	Java	395	219	167	213	148	1142
13	C++	290	165	133	100	116	804
14	C#	595	391	422	293	351	2052
15		1280	775	722	606	615	3998
16	Pascal normál						
17	Java	371	251	254	199	205	1280
18	C++	289	190	194	153	153	979
19	C#	594	448	435	373	382	2252
20		1254	889	903	725	740	4511
21	Pascal optimális						
22	Java	361	220	213	190	164	1148
23	C++	288	170	161	142	109	870
24	C#	594	394	401	344	291	2024
25		1243	784	775	676	564	4042
26	Pascal páratlan						
27	Java	269	260	230	205	186	1150
28	C++	218	198	174	155	136	881
29	C#	466	439	400	363	339	2007
30		953	897	804	723	661	4038

8. ábra: Futási eredmények a szálkiosztó stratégiák összehasonlításához

A szál nélküli módszer a leggyorsabb akkor, ha a többi algoritmust is (normál, hatvány, Fibonacci, Pascal) egy szálon vizsgáljuk. Ez egyértelmű, mivel itt nem kell az intervallumot felosztani, tartományokat számolni (normál, hatvány, Fibonacci, Pascal) illetve a szálak menedzselése sem vesz el időt szál nélküli módban. Míg ha szálat is kezelő módszert használunk, attól még létrehozzuk, elindítjuk, várunk és megszüntetjük a szálat, így nem a főszálon fut tovább a program, és ez az idő számottevően érezhető. Mivel a szálkezelésen van a dolgozat hangsúlyos része, ezért továbbiakban csak a többi módszert ismertetjük (8. ábra).

Első esetben **állapítsuk meg programozási nyelvtől és száltól függetlenül, hogy melyik módszer bizonyult a leggyorsabbnak!**

A leglassabb a Pascal normál nevű módszer, ami érthető is, mivel a legnagyobb számok közül is ugyanannyi számról kellett döntenie a prímtesztnek, mint a legkisebbeknél. Így a várt eredményeknek megfelelően alakult. Egy picit gyorsabban a normál nevű módszer teljesített, ami szintén nem meglepő, mivel a tartománykiosztás egységes. A várakozással ellentétben a Pascal optimális, majd nagyon szorosan mögötte (tízezres tartományban 4 ms-mal gyorsabban) a Pascal páratlan következik, ami érdekes, hiszen ezt a két algoritmust, különösen a Pascal páratlant gondoltuk a leggyorsabbnak. (Pascal módszerek közül a sorrend a vártnak megfelelően alakult). Szintén még szorosan mögötte a Fibonacci módszer végzett.

Elég nagy előnnyel a hatvány nevű módszer győzött, amit meglepőnek találunk, várakozásainknak nem megfelelő eredményt adott. Ez a leglassabb algoritmushoz képest 21%-kal gyorsabbnak bizonyult a program teljes futására vonatkozva.

A szálakat is érdemes megvizsgálni mérési eredmények szerint átlagban: érdemes lehet az öt-hat szálon való futtatás, de ez gépenként erősen eltérő lehet, nagyon hardverfüggő, így egyértelmű kijelentést nem tudunk tenni.

Ha a programozási nyelvet figyelembe vesszük, akkor sem tapasztalunk az eddig megállapított dolgokban változást. A módszerek gyorsasági sorrendje és a szálak számára tett kijelentések is megállják a helyüket.

Tehát ha csak a nyelvre vagyunk kíváncsiak, akkor a C# bizonyult erősen a leglassabbnak. A Java a következő, de nem sokkal bizonyult gyorsabbnak a C++ programozási nyelv.

Továbbfejlesztési lehetőségek

Jelenleg hétféle módszert támogat a programunk, ami az intervallumokat kiosztja, ezért még lehetne újfajta stratégiákat kialakítani pár nevezetes számsorozat segítségével (Bell, álprím). Jelenleg csak három nyelven készült el a teljes implementáció, így érdemes lehet új nyelveket is megvizsgálni, például Visual Basic, Python.

Nem lenne szükség minden egyes módszer futásához egyedi CSV-hez, hanem egy nagyba mehetnének bele az eredmények, így újra szükségünk lenne a CSV-be olyan oszlopra, ami jelzi a módszert, illetve a programozási nyelvet sem ártana benne tárolni. Adattárolásra adatbázist is alkalmazhatnánk.

Fájlfeltöltést lehetne biztosítani a felhasználónak, hogy a saját mérési eredményeit megjeleníthesse, hogy a grafikus reprezentáló szoftver segítségével könnyebben tanulmányozhassa. Ha feltöltünk egy fájlt, annak szükséges adatai egy adatbázisban tárolódnának és a régebben feltöltött CSV-k törlődnének automatikusan. Az új CSV szerint pedig a programnak képesnek kéne lennie automatikusan felismerni a szükséges adatokat.

A megjelenítést illetően a könnyebb megértéshez, a jelenleginél sokkal dinamikusabb GUI-t szeretnénk kifejleszteni. Támogatnunk kell a méretezhetőséget, a reszponzív megjelenést, valamint a szálakhoz tartozó képernyők nagyíthatóságát és gördíthetőségét (mozgatását). Ezenfelül az akadálymentesség kérdését is felvethetjük.

Jó ötlet lenne egy olyan ábrázolási mód is, ahol egymás mellett közvetlen tudnánk összevetni a CSV-ben található adatokat, bármely tetszőleges szempont szerint, például metódus, szálak száma, programozási nyelv.

További jó ötlet lehetne a megjelenítésért felelős algoritmus további optimalizálása is. Még nagyobb érdeklődést különböző magyarázó és egyben látványos animációkkal érhetnénk el a továbbiakban.

Befejezés

Láthattuk azt, hogy kutatásunk alatt nagyon sok témát kellett érintenünk, ezért ezt rendkívül sokrétű és egymással erősen összefüggő feladatoknak kell értenünk. Kisebb kutatómunkát és segítséget igényelt a feladat megértése, de annál nagyobb kutatásra és rengeteg próbálkozásra volt szükség például a szálkiosztás stratégiájának algoritmus szerinti megvalósítását, illetve ennek matematikai vonzatát illetően. Szükséges volt érintenünk a párhuzamosítás jelentőségét, egyrészt hardveres szemlélettel, másrészt szoftveres álláspontból is megértést és kutatást igényelt.

Mindemellett nélkülözhetetlen volt a hosszas tanulmányozás az implementációkkal összefüggésben is. Különösebb nagyobb figyelmet igényelt a programok és a különböző szálkiosztási stratégiák megtervezése és felépítése. Azonkívül elengedhetetlennek tartottuk az objektumorientált szemléletű tervezést és kivitelezést, amely könnyebben javíthatóvá és áttekinthetőbbé tette a programjainkat. Valamint az implementációk mind az egy-egy nyelvhez tartozó szintaktika elsajátítása céljából, mind a megfelelő eszközök (különböző függvények, eljárások) kiválasztásához igényelték a legnagyobb volumenű kutatást és folytonos kísérletek sorozatát.

Feltérképezhettük és összevethettük a szálkezelési stratégiák különbségeit, konklúziókat vonhattunk le az elért eredmények alapján. Megtapasztalhattuk eredményeit az egyes implementációk által reprezentált szálkezeléssel vonzatos függvények tulajdonságait, amellett ezek különbségét is.

Ezeket összevetve az elért eredmények és tapasztalatok lehetnek a legjobb hatással a későbbiek folyamán. Hozzájárulhat más fejlesztések tanulmányaként, illetve ötleteiként is. További optimalizálás ötleteket adhat más fejlesztők részére is, amely mindenképp jó támpontot jelent. Az elkészült projekt oktatási céllal is felhasználható lehetne, a probléma sokrétű szemléltetése céljából.

A kutatásokkal, tervezéssel és rengeteg kísérletezéssel töltött munka után egyértelműen képesek voltunk fejlődni számos területen. Rengeteg hasznos tapasztalattal gazdagodtunk, mivel a projekt sokrétű felvetődő problémát érintett. Így például gazdagabbak lettünk a programozási ismereteinket illetően, a kutatómunkában való hatékonyságot illetően, elméletek megértésének tekintetében és végül, de nem utolsó sorban a kooperativitási készségünket illetően. Egymást segítve találtunk

megoldásokat a folytonosan felvetődött nehézségekre. Sőt egymás ötleteit folyton javítgatva jutottunk egyre és egyre kedvezőbb eredményekre, ez hatással volt a csapatszerű, céltudatos szemléletmódunkra is.

Köszönet
Szeretnénk pár mondatban kinyilvánítani mély köszönetünket azoknak az embereknek, akik nagyban hozzájárultak ezen eredmények eléréséhez. Ez a cikk nélkülük valószínűleg nem valósulhatott volna meg. Ezért is szeretnénk őket egy lapon méltányolni és az eredményekhez hozzáadott munkásságukat megemlíteni. Szeretnénk kinyilvánítani köszönetünket konzulensünknek, Kaczur Sándor tanár úrnak, aki hosszan tartó munkával, sok konzultálással, sok egyeztetéssel és rengeteg szerkesztéssel segített utunk során. Továbbá a saját ötletéből és Java nyelvű kezdeti implementációból kiindulva juthattunk el az implementációk megvalósításához és kiegészítéséhez. Szeretnénk köszönetünket nyilvánítani Tavas Bernadett munkássága után is, aki a pályamunka lektorálását és fordítását végezte. Szeretnénk köszönetünket nyilvánítani továbbá Kasza Gergő Dániel apróbb, de annál nagyszerűbb segítségét a webes implementáció JavaScriptes megvalósítás tömbkezelővel kapcsolatos részéhez. Végül, de nem utolsó sorban szeretnénk családjaink számára is köszönetünket nyilvánítani, akik ebben az időszakban különös türelemmel és megértéssel fordultak hozzánk.

Referenciák:

[1] Sain Márton: Nincs királyi út! – Matematikatörténet, Gondolat Kiadó, Budapest, 1986, ISBN 963 281 7044, http://mek.oszk.hu/05000/05052/pdf, 2015.01.15.
[2] Primality test, http://en.wikipedia.org/wiki/Primality_test, 2015.01.15.
[3] Eratoszthenész szitája, http://hu.wikipedia.org/wiki/Eratoszthenész_szitája, 2015.01.20.
[4] Index - Tudomány - Áttörés az ikerprím-sejtés bizonyításában, http://index.hu/tudomany/2013/05/15/attores_az_ikerprim-sejtes_bizonyitasaban, 2015.01.26.
[5] Kónya László: Számítógép-hálózatok, Inok Kft., Budapest, 2006, ISBN 963 9625 16 7
[6] Informatikai biztonság I. - Kriptográfia, titkosítás napjainkban - DoclerWeb blog, http://blog.doclerweb.hu/post/informatikai-biztonsag-i-kriptografia-titkositas-napjainkban, 2015.02.10.
[7] Koltay Tibor - Szaniszló István: Kapcsolattartás e-mail útján az Interneten, Nemzeti Információs Infrastruktúra Fejlesztési Program Koordinációs Iroda, Budapest, 1996, ISBN 963 02 9946 1
[8] Németh L. Zoltán: Kriptográfia, SZTE, 2008 – Prezentáció, http://www.inf.u-szeged.hu/~zlnemeth/crypto/crypt10.pdf, 2015.02.15.
[9] GDF ILIAS-ETOR, http://ilias.gdf.hu --> Taneszköz tároló -> BSc/BA-> Számítógép architektúrák I. (BSc GI,MI) -> Számítógép-architektúrák (online tananyag) (privát)
[10] GDF ILIAS-ETOR, http://ilias.gdf.hu --> Taneszköz tároló -> BSc/BA-> Operációs rendszerek (BSc GI,MI) » Operációs rendszerek (on-line, F, BSc) (privát)
[11] Versenyhelyzet – Wikipédia, http://hu.wikipedia.org/wiki/Versenyhelyzet, 2015.01.30.
[12] Fibonacci Sequence, http://www.mathsisfun.com/numbers/fibonacci-sequence.html, 2015.01.25.
[13] Pascal's Triangle, http://www.mathsisfun.com/pascals-triangle.html, 2015.01.28.
[14] UML Class Diagram…, http://www.uml-diagrams.org/class-diagrams.html, 2015.02.05.
[15] Try Visio Professional 2013, http://www.microsoft.com/en-us/evalcenter/evaluate-visio-professional-2013, 2015.01.19.
[16] NetBeans IDE Download, https://netbeans.org/downloads, 2015.01.20.
[17] Java SE Development Kit 7 – Download, http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html, 2015.01.20.

[18] Download Microsoft Visual Studio..., http://www.microsoft.com/en-us/download/details.aspx?id=44914, 2015.01.20.
[19] Download Microsoft .NET - keretrendszer..., http://www.microsoft.com/hu-hu/download/details.aspx?id=42642, 2015.01.20.
[20] Prognyelvek portál Csharp, http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=16#section_6, http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=21, 2015.02.10.
[21] Java System.currentTimeMillis() equivalent in C# - Stack Overflow, http:// stackoverflow.com/questions/290227/java-system-currenttimemillis-equivalent-in-c-sharp, 2015.02.14.
[22] .net - C# equivalent for Java ExecutorService.newSingleThreadExecutor(), or how to serialize mulithreaded access to a resource - Stack Overflow, http://stackoverflow.com/questions/4661760/c-sharp-equivalent-for-java-executorservice-newsinglethreadexecutor-or-how-t, 2015.02.15.
[23] Code::Blocks Download, http://www.codeblocks.org/downloads, 2015.01.30.
[24] Open Browsing Platform (OBP) - ISO/IEC 14882:2011, https://www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-3:v1:en, 2015.01.30.
[25] async - C++ Reference, http://en.cppreference.com/w/cpp/thread/async, 2015.02.10.
[26] Notepad++, http://notepad-plus-plus.org/download, 2015.02.08.
[27] WampServer, http://www.wampserver.com/en, 2015.02.08.
[28] Eredmények grafikus reprezentációja, http://hallomamitlatnek.ingyenweb.info/kmd, 2015.02.22.