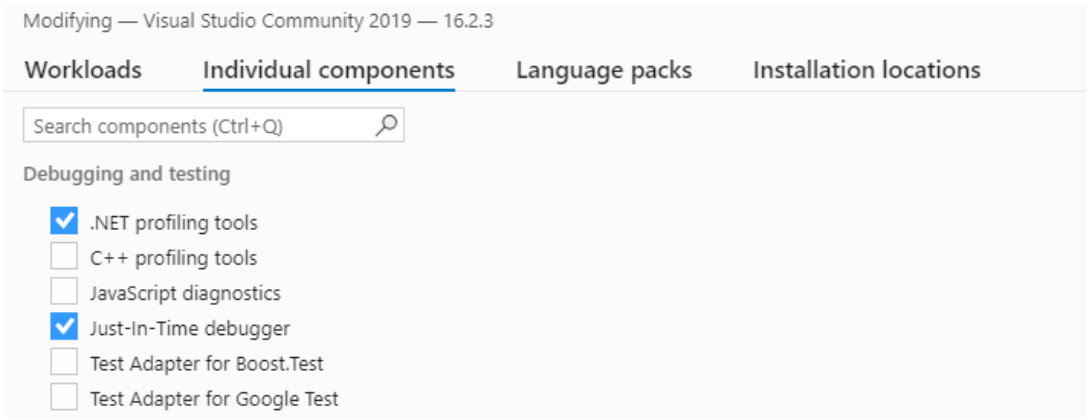
Szoftvertechnológia és -technikák
11. Gyakorlat

A tesztelés és teljesítménymérés alapjai

Bevezetés

A gyakorlat két részből áll, az első a szoftvertesztelésről, a második az alkalmazások teljesítményprofiljának méréséről szól. Mind a két rész egy adott pontig vezetett, majd önálló munkát igényel. Mellékelünk egy kiinduló projektet is. Ezt célszerű letölteni a tantárgy honlapjáról.

A kiadott kód Visual Studio 2019 alatt működik. Szükséges hozzá a „.NET Profiling tools” nevű individual component telepítése (a Visual Studio Installer segítségével). Ha laptopról dolgoznál, ezt most telepítsd. A laborgépeken telepítve van.



Bevezető: a szofvertesztelésről dióhéjban

Az amerikai statisztikai hivatal (National Institute of Standards and Technology) 2002-ben végzett egy kutatást, amely alapján a szoftverhibák – bugok – által okozott károkat évi $59.5 milliárdra becsülte, ez nagyjából az Egyesült Államok akkori GDP-jének 0.6%-ának felelt meg. Ugyanezen kutatás szerint ennek a költségnek nagyjából a harmada elkerülhető lett volna standard, könnyen elérhető teszteszközök, eljárások alkalmazásával. Nem csoda tehát, hogy a komolyabb szoftverfejlesztő cégek rendelkeznek különböző tesztelési protokollokkal és megkövetelik alkalmazottjaiktól, hogy kövessék ezeket.

A szoftvertesztelést vagy maguk a fejlesztők, vagy kinevezett tesztmérnökök végzik. Fontos ugyanakkor, hogy egy megbízható tesztkörnyezet kialakításához és tesztek írásához a szokásos fejlesztői képességek mellé továbbiak szükségesek. Egy több éves tapasztalattal rendelkező senior fejlesztő se tudja ellátni egy tesztmérnök feladatait külön felkészülés és tesztelői gyakorlat nélkül. Ettől függetlenül, minden alkalmazást érdemes tesztelni, célszerűen még a fejlesztési folyamat részeként. Már egy témalabor vagy szakdolgozat keretében elkészült alkalmazás is elég összetett ahhoz, hogy a tesztelés hiánya komoly kockázatot jelentsen a sikeres elkészülésére. A legjobb esetben a tesztelés hiánya csak a fejlesztési folyamatot lassítja, ahogy a váratlan hibák/bugok folyamatosan felfedik magukat. Minél később fedezünk fel egy hibát, annál nehezebb megtalálni az okát a kódban és annál bonyolultabb a kijavítása. Tesztelőként célunk, hogy megpróbáljuk ‘hibára bírni’ az alkalmazást. Fontos, hogy a hibákat mi fedezzük fel, ne a felhasználók. Ennek megfelelően nem szabad félteni a kódot, tesztjeink könyörtelenül meg kell követeljék az alkalmazástól, hogy megfeleljen a specifikációnak.

Ez a labor bevezet a tesztelés világába, ízelítőül szolgál. Később egy teljes specializáció (Tesztelés és üzemeltetés) épül erre négy tantárggyal. Ezek a tárgyak sokkal bővebben, formálisabban és rigorózusabban tárgyalják ezt a témakört.

1. A tesztelés módjai, fogalmai

Itt az ideje, hogy gyorsan definiáljuk a témakörben gyakran előforduló fogalmakat. Ezeket a gyakorlatban általában angolul, vagy ‘magyarosított angolul’ szoktuk használni. A laboron gyakorlatilag csak a unit tesztelésre jut idő.

user testing – manuális tesztelési módszer, mely során egy felhasználó végigpróbálja az alkalmazás funkcióit. Gyakori, hogy a teszt tervezője előírja, hogy melyik funkciókat kell ellenőrizni, vagy akár egy teljes forgatókönyvet is készíthet.
system testing – a rendszer egészének automatikus ellenőrzése. Célja, hogy verifikáljuk, hogy a működés az előzetesen elkészült specifikációnak megfelel.
unit testing – gyakran használt teszttípus, a rendszer egy kis komponensét vizsgálja a többitől önállóan. Az egység(unit) relatív szabadon változhat, gyakori osztályonként vagy függvényenként végezni a unit tesztelést.
integration testing – tesztelési feladatkör. Miután meggyőződtünk, hogy az alkalmazás komponensei önállóan helyesen működnek, az integrációs teszt ellenőrizi, hogy együtt is helyesen működnek.
regression testing – tesztelési feladatkör. Új funkciók bevezetésénél fontos, hogy a meglévők továbbra is helyesen működjenek. A regressziós tesztelés ezt ellenőrzi.
test-driven development – rövid fejlesztési ciklusokon alapozott szoftverfejlesztési módszertan. Lényege, hogy az adott specifikáció alapján először írjuk meg a teszteket, mint a kódot. Időt nyerhetünk hiányos/ellentmondásos specifikáció esetén, hiszen még az implementáció megkezdése előtt kijavíthatjuk ezt.
pair programming – agilis szoftverfejlesztési módszertan, mely szerint két fejlesztő felváltva készít egymásnak teszteseteket és ír kódot.

continuous development – agilis szoftverfejlesztési módszertan, mely felismeri, hogy egy alkalmazás fejlesztése egy organikus, iteratív folyamat. Kapcsolata a teszteléssel, hogy minden iterációban a tesztek eredményessége segít eldönteni a következő iteráció ütemezését, tervezését. A tesztek tipikusan automatizáltan futnak valamilyen continuous deployment/continuous integration eszköz segítségével.

2. White-box vs black-box

Aszerint, hogy a tesztet tervező mérnök belelát-e a tesztelendő rendszer/komponens belső működésébe, megkülönböztetünk black-box és white-box tesztelést. Mindkettőnek megvannak az előnyei, white-box tesztelés esetén, ha rendelkezésre áll a forráskód, könnyebben kiderül, hogy melyikek a kritikus tesztelendő részek. Ennek megfelelően inkább a kisebb komponensek (tipikusan unit tesztelésnél) tesztelésénél alkalmazzák. A black-box tesztelés nem igényli a szoftver belső működésének ismeretét, teljesen elfüggetlenül ettől. Tipikusan egy teljes rendszer/alrendszer vizsgálatakor alkalmazzuk, összehasonlítva a rendszer kívülről megfigyelhető működését a specifikációjával.

3. Tesztesetek dokumentálása

A teszteseteket is dokumentálni kell! Ez lehetővé teszi, hogy egy sikertelen teszt esetén tudjuk, hogy mi volt a célja, hol keressük a hibát. Továbbá a dokumentáció készítésekor, vagy utólagos tanulmányozása során felfedezhetjük, hogy kimaradt egy fontos teszteset.

Amikor elkezdjük a tesztesetek tervezését, célszerű leírni a stratégiánkat, beazonosítva a tesztelendő részeket és módszertanokat. Mivel a gyakorlatban gyakran nem fogunk elkészülni időben minden tesztesettel, célszerű prioritásokat felállítani közöttük. Célszerű esetleg csoportosítani a teszteseteket és felelősöket kinevezni. A dokumentumot, amely ezeket tartalmazza, teszttervnek nevezzük.

1.3.1 Kód lefedettség (code coverage)

A tesztelés minőségét is célszerű ellenőrizni, ez alatt elsősorban a kód lefedési szintjét értjük. Pontosabban: a kódelemek (utasítás, elágazás, útvonalak) mekkora részét érinti legalább egy teszteset.

- utasítás (statement) szintű: gyakran cél a 100%-os lefedettség megcélzása, de ritkán érhető el (például azért mert lehet olyan defenzív kód, assertek, amiket a gyakorlatban nem lehet előidézni)
- elágazás (branch) szintű: minden feltétel egy elágazáshoz vezet a kódban. Általában ezt szokták mérni és tipikusan a teljes lefedettséget a cél.
- útvonal (path) szintű: egy program konkrét futására gondolhatunk úgy is, mint különböző elágazásokon választott útja. Az összetettebb alkalmazások esetében a lehetséges útvonalak száma annyira megnő, hogy a 100%-os lefedettség a gyakorlatban nem érhető el.

1. FELADAT: Bemelegítés

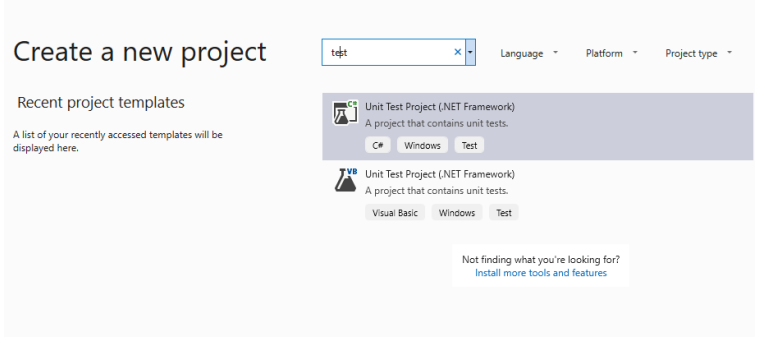
Nyisd meg a tárgy honlapjáról letöltött solutions. Három projektet találhatsz benne, ebből most a TesztelesLabort fogjuk használni. Közösen fogjuk áttekinteni a forráskódot.

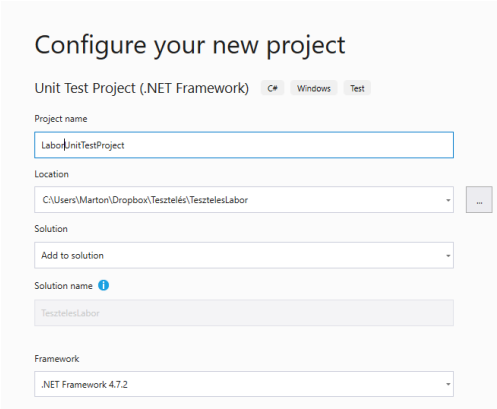
Első lépésként készítsd el a hiányzó `UppercaseChar` metódust a `StringMagic` osztályban! Ez egy egyszerű stringeket manipuláló metódus, amely nagybetűre konvertálja a megadott string egy bizonyos karakterét. A metódus szignatúráján nem szabad változtatni. Első célunk legyen az, hogy az alkalmazásunk forduljon le sikeresen. Ha még nem működik helyesen a `Main` függvényben megadott példán, nem baj. Hamarosan tesztek segítségével kiderítjük az esetleges hibák helyét.

Tippek:

- C#-ban a stringek nem módosíthatóak közvetlenül (immutable), vagyis ezt a feladatot úgy célszerű megoldani, hogy létrehozunk egy új stringet.
- Használd a + operátort két string konkatenálására:
  ```
  string str = "Hello " + " world"; //str = "Hello world";
  ```
- Nagybetűre konverzió: `char.ToUpper(char c)`
- Egy string egy substringje:
  ```
  string substr = str.Substring(startFrom, length);
  ```

Az alkalmazásunk kész van, de meg kell bizonyosodjunk róla, hogy helyesen működik. Hozzunk létre ehhez teszteket. Pontosabban már rendelkezésünkre állnak a tesztek, de szükségünk van egy projektre, ami magába foglalja ezeket. Hozzunk létre tehát a File->New menüből egy Unit Test Project-et. Figyeljünk oda, hogy a C# kódhoz tartozót válasszuk:



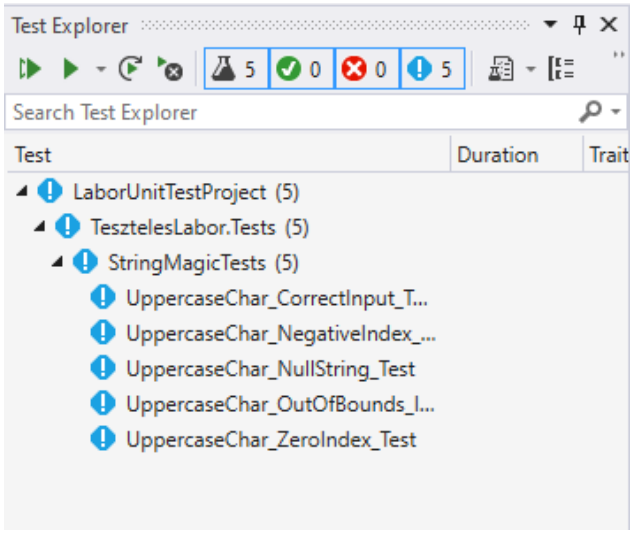


Állítsuk be, hogy Add to solution, nem szeretnénk egy új solutiont létrehozni.

Tekintsük át a teszteseteket, majd másoljuk be az alábbi kódot a `UnitTest1.cs` fájlba! A `StringMagic` osztályt aláhúzza, mert nem ismeri. Ezt könnyen javíthatjuk, vegyük fel a `LaborUnitTestProject` projekt referenciái közé a `TesztelesLabor` projektet. Jobb-klikk a References elemre, majd Add Reference…

```
namespace TesztelesLabor.Tests
{
    [TestClass()]
    public class StringMagicTests
    {
        [TestMethod()]
        public void UppercaseChar_CorrectInput_Test()
        {
            StringMagic sm = new StringMagic();
            string str = "Hello world";
            string result = sm.UppercaseChar(str, 6);
            Assert.IsTrue(string.Compare(result, "Hello World") == 0);
        }

        [TestMethod()]
        public void UppercaseChar_NullString_Test()
        {
            StringMagic sm = new StringMagic();
            sm.UppercaseChar(null, 0);
        }

        [TestMethod()]
        public void UppercaseChar_NegativeIndex_Test()
        {
            StringMagic sm = new StringMagic();
            string str = "Hello world";
            sm.UppercaseChar(str, -1);
        }

        [TestMethod()]
        public void UppercaseChar_ZeroIndex_Test()
        {
            StringMagic sm = new StringMagic();
            string str = "Hello world";
            sm.UppercaseChar(str, 0);
            Assert.IsTrue(string.Compare(sm.UppercaseChar(str, 0), "Hello world") == 0);
        }

        [TestMethod()]
        public void UppercaseChar_OutOfBounds_IndexTest()
        {
            StringMagic sm = new StringMagic();
            string str = "Hello world";
            sm.UppercaseChar(str, 42);
        }
    }
}
```

Ha ez megvan, akkor ezentúl könnyen kezelhetjük a teszteseteinket a Test Explorer ablakból. Ha nem jelent meg automatikusan, a Test->Windows->Test Explorer menüből csalogatható elő. Gyorsan leolvasható, hogy melyik tesztjeink futottak sikeresen. Az itt található gombokkal újrafuttatjuk a teszteket.



Ha valamelyik teszt sikertelen volt, könnyen megnézhetjük közelebbről is, hogy mi lehet a gond. Jobb-klikkelve a kérdéses tesztre, válasszuk a debug opciót a legördülő menüből. Ezt most együtt fogjuk megnézni.

Ha maradtak sikertelen tesztjeid, javítsd ki az implementációd!

Ezt a típusú tesztelést black-box testingnek nevezzük, mivel a tesztet készítő egyén csak a függvény szignatúráját ismerte. Ennek a hátrányát, hogy könnyen lehet, hogy az alkalmazás még mindig nem hibátlan. Például lehet, hogy valamilyen okból 1000 karakteres string felett null-al térsz vissza. Ezt nem ellenőrzi egyik tesztünk se, de ha a tesztet készítő rendelkezésére bocsátjuk a forráskódot, akkor írhat tesztesetet, amely figyelembe veszi ezt az if ágat is. A legjobb természetesen az, ha úgy a tesztek készítője, mint mi, az alkalmazás fejlesztői egy közös, részletes specifikációból dolgozunk.

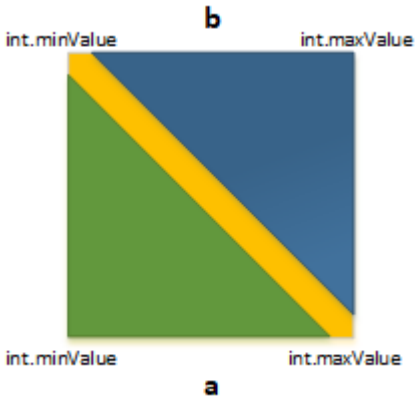
2. FELADAT: Tesztesetek tervezése közösen

Eddig készen kaptuk a teszteket, most mi fogjuk megtervezni közösen ezeket. A feladat az `IntMagic` osztály `calculateMax` metódusának tesztelése. Ez a .NET Math könyvtárát használja, ennek fogjuk a helyességét ellenőrizni. A módszer két egész számot fogad paraméterként: a, b.

Feladatunk egy olyan listát definiálni bemeneti (a,b) párokkal, amely hatékonyan lefedi a metódus bemeneti terét. A bemeneti tér fogalma a céljainknak megfelelően megegyezik a matematikából megismert függvény *értelmezési tartományával*. Mivel itt két változóval van dolgunk itt a bemeneti tér két halmaz Descartes-szorzata. Mindkét parameter az `int.MinValue` és `int.MaxValue` között vehet fel értékeket. A tér teljes lefedése nem megvalósítható, hiszen ez $(\texttt{int.MaxValue-int.MinValue})^2$ különböző tesztet eredményezne. Ez már ennél az egyszerű feladatnál is $\sim 4.6*10^{18}$, ez közelítőleg a Tejút szélessége méterben számolva. Osszuk fel (particionáljuk) tehát a teret kisebb részekre és úgy válasszuk meg a bemeneti párokat, hogy mindegyik partícióból szerepeljen legalább egy.

Első körben hagyjuk figyelmen kívül a és b lehetséges értékeit és elemezzük a viszonyukat. A mi esetünkben a és b viszonya alapján három alapvető esetet különböztetünk meg:

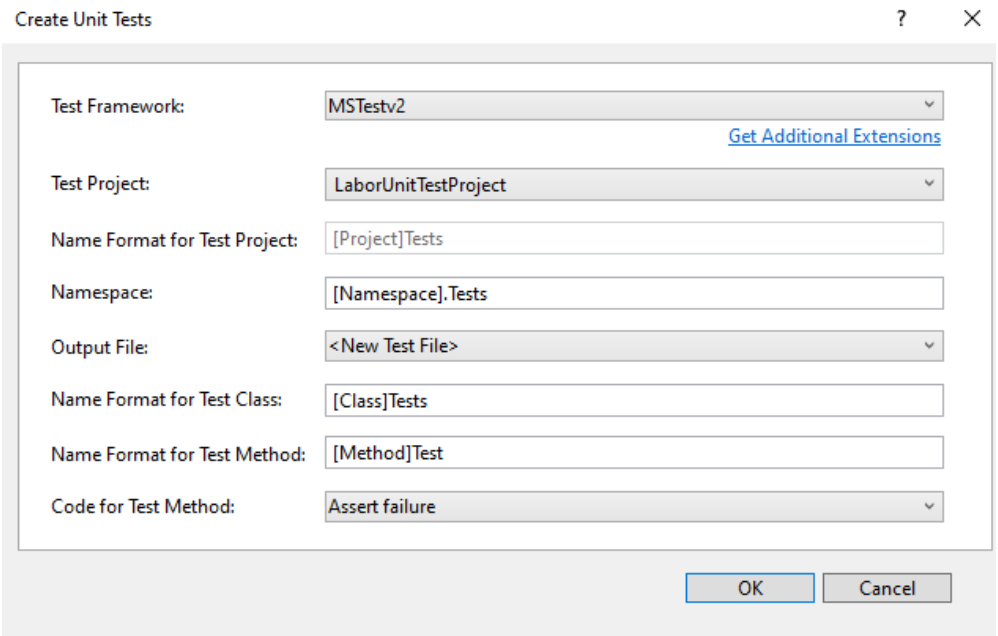
- a > b
- a < b
- a = b



A fenti ábrán a kék partíció az a < b, a zöld az a > b, valamint a sárga az a = b esetet jelzi, kissé eltúlzott arányokkal. Mindhárom esetet le tudjuk fedni például a következő bementi párokkal:

- a > b: (5, 2):
- a < b: (-123, -32)
- a = b: (0,0)

Vegyünk fel tehát három tesztesetet. Jobb-klikk a `calculateMax` metódus belsejében, majd Create Unit Tests. Hibát kapunk, csak publikus osztályok publikus metódusait unit tesztelhetünk. Állítsuk át a metódust publikussá, ezt biztos véletlenül elnézte a kód írója. A felugró ablakot a következő módon paraméterezzük.




```
[TestMethod()]
public void calculateMaxTestALargerThanB()
{
   IntMagic im = new IntMagic();
   int a = 5;
   int b = 2;
   Assert.AreEqual(a, im.calculateMax(a, b));
}
```

Az első tesztesetet még közösen írjuk meg, a másik kettőt mindenki saját maga készítse el.

Ha ezzel megvagyunk, gondolkodjunk el azon, hogy mik lehetnek a bugok tipikus forrásai. Nagyon gyakori például eggyel elrontani egy tömb indexelését, vagy egy for ciklus kilépő feltételében <= helyett < viszonyt vizsgálni. Ezek ú.n. *off-by-one* hibák és ezek lefedésére újabb teszteseteket fogunk felvenni. A partícióink határairól (boundaries) kell megválasszuk a bemeneti párokat. Ha belegondolunk, az eddigi tesztjeink közül az a = b esettel pont ezt csináltuk. Nem véletlenül bontottuk így fel a teret, a határvonalakat gyakran speciális kód kell, hogy kezelje, ezért könnyű hibát véteni itt a kódban vagy akár teljesen megfeledkezni ezekről az esetekről. Egészítsük most ki a bemeneti tér vizsgálatát az a és b potenciális értékeivel és készítsünk két új partíciót ez alapján:

a és b viszonya alapján:
- a > b
- a < b
- a = b

a értéke alapján:
- a = `int.MinValue`
- a < 0
- a = 0
- a > 0
- a = `int.MaxValue`

b értéke alapján:
- b = `int.MinValue`
- b < 0
- b = 0
- b > 0
- b = `int.MaxValue`

A nagyon kicsi/nagyon nagy értékek mellett a 0 körül is érdemes vizsgálódni. Ebben az esetben nehezebb látni, hogy miért célszerű külön vizsgálni a 0-st, de sok programozói hiba köthető a pozitív/negatív számok illetve az előjeles/előjel nélküli változók kezelésében. A 0 mint speciális eset gyakran fényt világíthat ezekre (pl. 8 bites unsinged int 0-1 = 255.). Kezdett megnőni az esetek száma, de szerencsére választhatjuk a bemeneti párainkat úgy, hogy egyszerre több esetet lefedjenek. Például, egy jó kompromisszum:
- (5, 2): a>b, a>0, b>0
- (-123, -32): a<b, a<0, b<0
- (0, 0): a=b, a=0, a=b
- (`int.MinValue, int.MaxValue`): a<b, a=`int.MinValue`, b= `int.MaxValue`
- (`int.MaxValue, int.MinValue`): a<b, a=`int.MaxValue`, b= `int.MinValue`

Implementáld a két új tesztesetet!

Bevezető: egy alkalmazás teljesítményprofilja

Miután megbizonyosodtunk róla, hogy a függvény helyesen működik, a következő lépésben áttérünk a labor második témakörére, a teljesítmény tesztelésére. Nagyon fontos, hogy az általunk készített alkalmazások helyesen működjenek, de tipikusan nem ez az egyetlen követelmény. Gyakori, hogy a megrendelő kiköti, hogy az alkalmazás meg kell, hogy feleljen bizonyos teljesítmény kritériumoknak. Gondoljunk arra, hogy ha egy webes áruház lassan jeleníti meg a termékeket, a felhasználók egy része dönthet úgy, hogy inkább máshol vásárol. Vagy ha a metróban a beléptető kapunak pár másodperc kell, hogy ellenőrizze egy bérlet érvényességét, akkor hányan fognak elkésni a reggeli csúcsban.

Az alkalmazás memóriaigénye is kulcskérdés, különösen beágyazott rendszerekben, vagy ha az alkalmazás egy olyan környezetben kell fusson, ahol szűkös erőforrásokon osztozkodik. Ez utóbbi esetben ha elfogy a fizikai memória, az operációs rendszer valamelyik lassú háttértáron található swap fájlt kezdi el használni, ami drasztikusan megnövekedett futási időt eredményezhet.

Másik, a memóriához kapcsolódó probléma lehet az ún. *memory leak*, azaz memóriaszivárgás. Ez akkor figyelhető meg, ha az alkalmazás nem jól szabadítja fel a memóriát és ezért hosszabb idejű használat esetén mind több és több memóriára van szüksége. A memóriaszivárgás javítása nehéz feladat és nagy segítséget jelent benne a profiling eszközök használata.

A labor során kizárólag az alkalmazásunk futási idejének csökkentésére fogunk koncentrálni. Mérni fogjuk a *CPU időt (CPU time)*, azaz, hogy alkalmazásunk mennyi ideig hajtott végre műveleteket valamelyik CPU magon. Az alkalmazás valódi futási ideje lehet ennél rövidebb is, ha párhuzamosítjuk és egyszerre több szálon fut. Lehet hosszabb is, ha az alkalmazásnak időnként várakoznia kell, például fájl- vagy hálózati műveletekre, esetleg az operációs rendszer ütemezőjére.

A mérést a Visual Studio részét képező CPU Profilerrel fogjuk végezni. Többféleképp lehet mérni a CPU időt, esetünkben a profiler mintavételezéssel fogja megállapítani, hogy melyik függvény mennyi ideig futott. Másképp: valamilyen gyakorisággal mintát fog venni az alkalmazás belső képéről és le fogja jegyezni, hogy épp melyik függvény futott. A végső becslés gyakorlatilag a minták összesítése egy táblázatban.

Szerencsére, ha nem fektetünk különös hangsúlyt az alkalmazásunk teljesítményére, a fordító akkor is próbálja optimalizálni a kódunkat. Például:

- A kódban többször kiszámított értékeket csak egyszer számolja ki.
- Azt a kódot, aminek az eredményét nem használjuk fel sehol nem is fordítja le.
- Függvényhívásokat küszöböl ki kódmásolással (inlining).
- Ciklusokat fejt ki, megspórolva ezzel a cikluskezelő utasításokat (loop unrolling).
- Ha sokszor kell elvégezni ugyanazt a műveletet több változón, és van megfelelő CPU utasítás hozzá, egyszerre több változón végzi el (auto-vectorization).

Sőt, maga a CPU is törekszik a hatékonyságra: próbálja kitalálni, hogy alkalmazásunknak melyik utasításai fognak a későbbiekben végrehajtódni és előre számol (branch prediction). A gyakran használt adatokat és kódrészeket gyorsítótárban tárolja (caching).

Fontos kiemelni: az optimalizációnak minden esetben kötelessége a funkcionalitást érintetlenül hagynia! A hibás, de gyors működés nem jó megoldás.

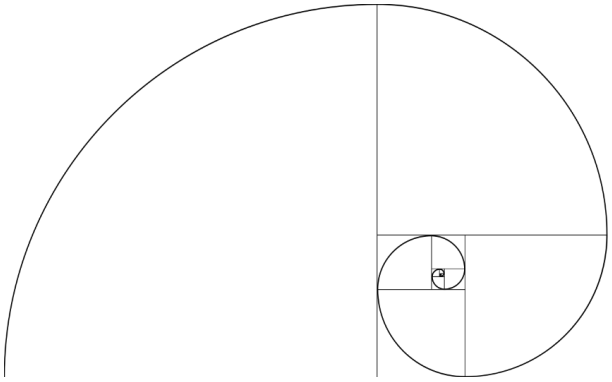
Ezeket a technikákat itt csak érdekességként említjük.

A teszteléses rész kidolgozását a következő MIT OpenCourseWare tanagyag ihlette.
Forrás: https://ocw.mit.edu/ans7870/6/6.005/s16/classes/03-testing/#reading_3_testing

3. Feladat – Teljesítménymérés

Nézzük meg, hogy miként tudjuk a Visual Studiot felhasználni arra, hogy elkészítsük alkalmazásunk teljesítményprofilját.

A gyakorlás kedvéért most saját magunk készítsünk egy egyszerű alkalmazást, amely kiszámítja az aranymetszést (golden ratio). Matematikai érdekesség, hogy az egymást közvetlenül követő Fibonacci számok aránya jól közelíti a mágikus 1.61803398875-as értéket. Ahogy a Fibonacci számok tartanak a végtelenbe, úgy a közelítés hibája tart 0 fele. Ezt fogjuk mi is kihasználni.



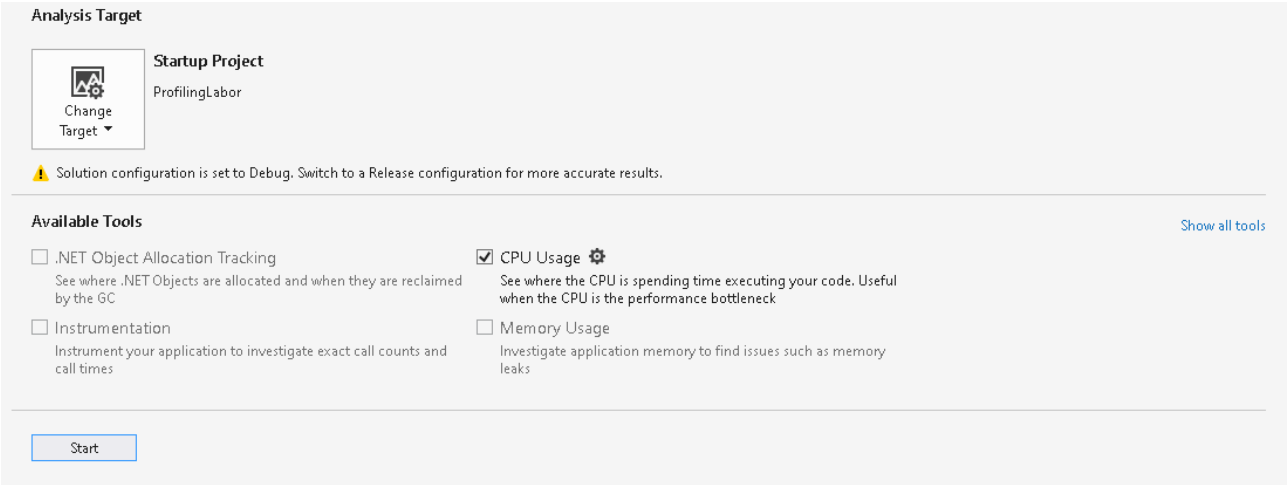
Forrás:https://es.wikipedia.org/wiki/Archivo:Fibonacci_spiral.png

Térjünk át a TesztelesLabor projektről a Profiling_Fibonacci projektre! Ezt a projekten jobb klikkelve a Set as StartUp Project opcióval tehetjük meg.

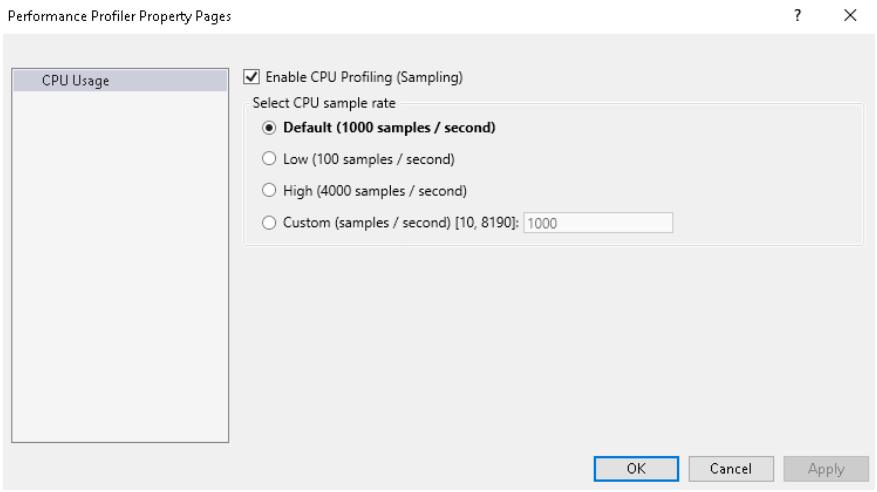
Futassuk az alkalmazást Ctrl+F5 billentyűvel és figyeljük meg, hogy míg az ötöd rendű Fibonacci számokkal végzett közelítést gyorsan kiszámította, a 40-ed rendűre sokat kellett várjunk! Igaz, ez már 10 tizedes pontosságú közelítést adott.

Viszgáljuk meg a kiadott forráskódot, értsük meg a számítás logikáját, a rekurzív függvény működését!

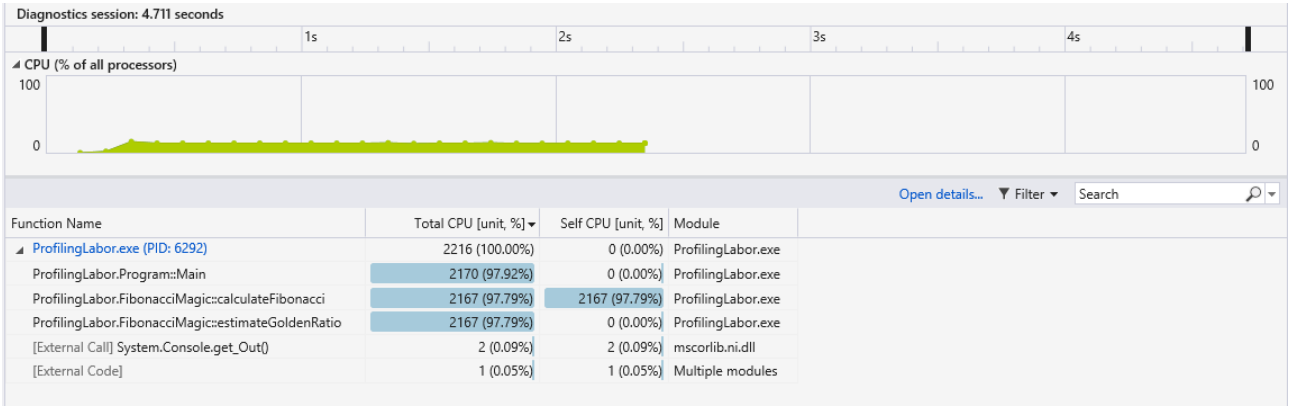
Mérjük meg, hogy mi a leglassabb része az alkalmazásnak! Ezt a legegyszerűbben úgy tehetjük meg, ha az Analyze menüből kiválasztjuk a Performance Profiler opciót (Alt+F2):



Itt tudjuk beállítani, hogy melyik projektet szeretnénk megvizsgálni, illetve az analízis módját. Egyszerre csak egy mód lehet kiválasztva, ezért van a másik három szürkével jelölve. Az alapértelmezett beállítás megfelel a célunknak, hiszen alkalmazásunk sebességét szeretnénk növelni, ezért azt szeretnénk megvizsgálni, hogy a CPU-nk melyik függvényekben tölti a legtöbb időt. Ha a CPU Usage melletti fogaskerékre klikkelünk, beállíthatjuk a mintavételezés rátáját. Az alapértelmezett megfelel a céljainknak. Ha egy olyan alkalmazást teljesítményprofilját szeretnénk mérni, ami sok, gyors lefutású műveletből áll, célszerű lenne egy gyakoribb mintavételezést beállítani.

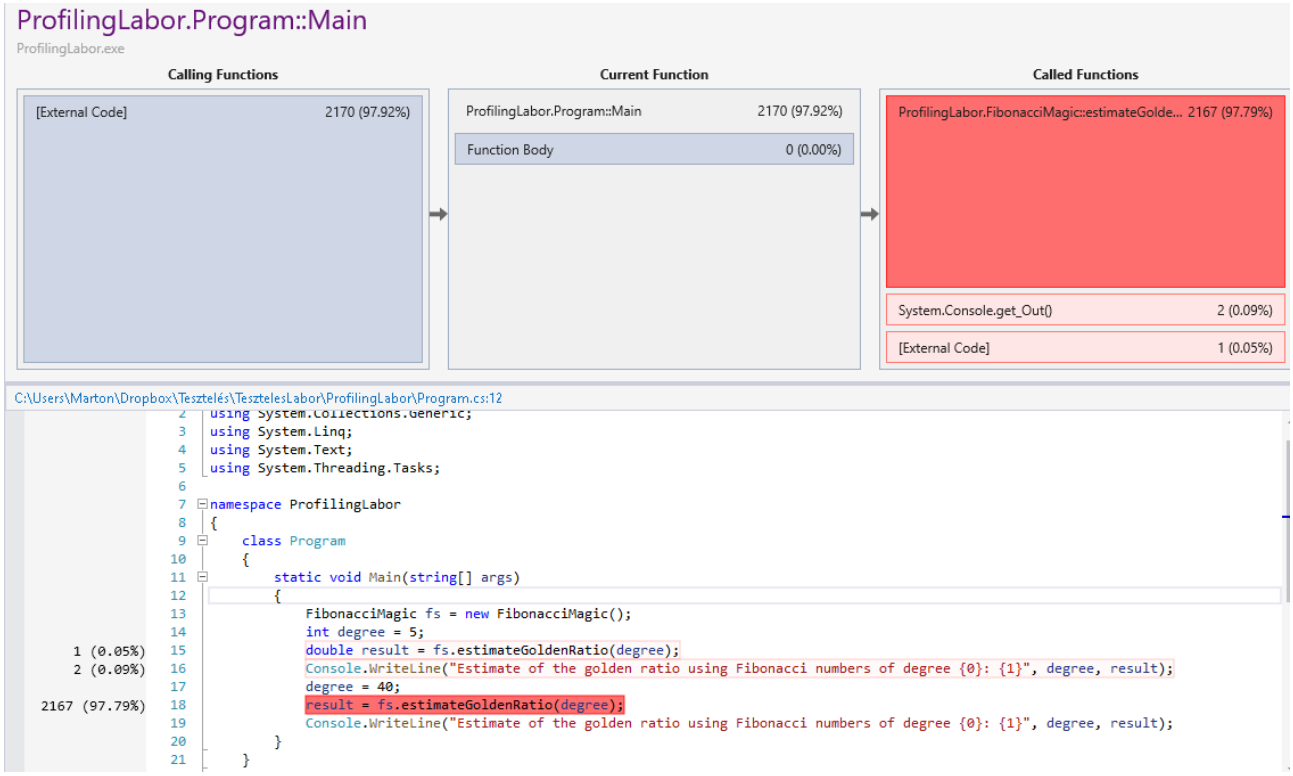


A start gomb megnyomásával elindul a mérés, majd a Visual Studio összesíti a mérés eredményeit. Egy példa erre az alábbi ábra:

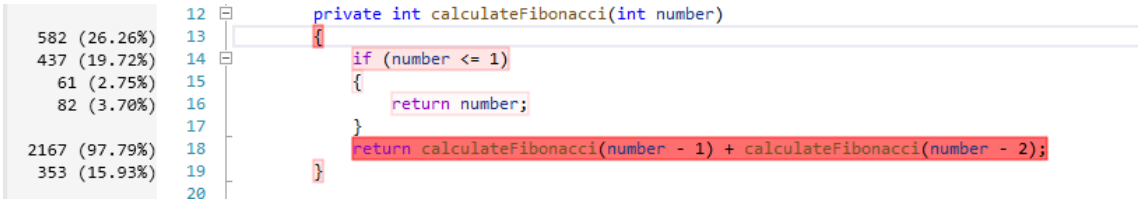


A felső sor szerint nagyjából összesen 5 másodpercig tartott a mérés. Ennél érdekesebb a gráf alatt levő lista, ahol függvényekre lebontva látjuk, hogy az alkalmazásunk hol mennyit időzött. Eszerint összesen 2216 egységnyi CPU időt vett igénybe. Az egység ms, vagyis közelítőleg 2.2 másodpercig terhelte le az alkalmazás egy virtuális CPU magunkat 100%-ban.

Két oszlopra bontva láthatjuk az eredményeket úgy abszolút értékben, mint százalékosan arányosítva a teljes futási időre. Az első a Total CPU időt mutatja, vagyis, hogy a függvényben mennyi ideig tartózkodott az alkalmazás. A második a Self CPU időt, ez az előző csökkentve a meghívott függvényekben töltött idővel. Ez érthetőbb lesz, ha ráklikkelünk az Open details opcióra, amely faként jeleníti meg, hogy melyik függvény melyik függvényt hívja és hogyan viszonyul ehhez a futási idejük. Az alábbi ábrán jól látható, hogy a `Main` függvényben töltötte az alkalmazás a CPU idő 97.92%-át az és ebből az `estimateGoldenRatio` függvény tette ki a 97.79%-át. Közben a kódban is jelöli alul a kérdéses sorokat.

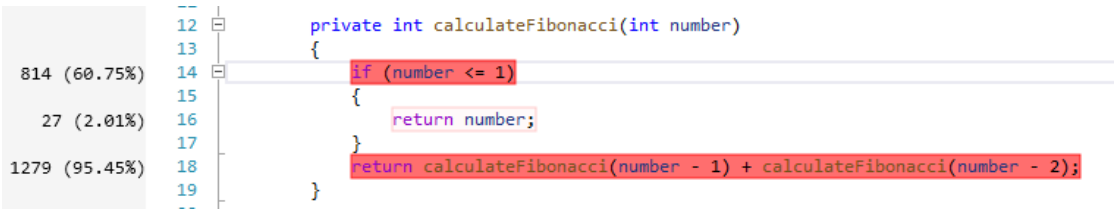


Ha ráklikkelünk az `estimateGoldenRatio` függvényt reprezentáló piros téglalapra, akkor megmutatja, hogy ez milyen függvényeket hívott és hol mennyi idő ment el. Nem meglepő módon a `calculateFibonacci` függvény a leglassabb része az alkalmazásunknak, klikkeljünk ezért a neki megfelelő téglalapra. Az alul megjelenő kódban soronként követhetjük, hogy a mintavételezett pontok mekkora hányadáért felelnek.



Ez itt egy különös eset, hiszen rekurzióval van dolgunk. Azt várnánk, hogy a sorok összege 97.79%-ot adjon ki. Ehhez képest a 18-as sor egyedül felel ennyiért. Ez annak tudható be, hogy ez egy rekurzív hívás, ezért gyakorlatilag az összeg így külön még egyszer szerepel. Hagyjuk figyelmen kívül. Szintén különleges, hogy arányaiban nézve sok időt tölt az alkalmazás a 13. és 19. sorokban. Ezek az értékek közelítik a függvényhívás költségét, a stack karbantartásához szükséges műveletekkel eltöltött időt. A rekúrziónak pont ez a hátránya, nagyon mély hívási hierarchiákat eredményez. A `calculateFibonacci` metódusban a rekurzió lecserélése egy nem-rekurzív algoritmusra tehát egy jó kiindulási irány lehet a teljesítmény növelése érdekében.

A többi sor összegére is elvárhatnánk, hogy 97.79%-ot adjon, de ez se igaz, a 68,36% jócskán ez az érték alatt marad. Ez részben betudható annak, hogy Debug módban mérjük az alkalmazásunkat. Ilyenkor számtalan, a nyomkövetéshez szükséges, további rejtett instrukciókat helyez el a fordító a kódba. A tanulság tehát, hogy a pontos mérések érdekében Release módban futtassuk a kódunkat. Ismételjük tehát meg a mérést Release módban. Az eredmény a következő ábrához hasonlóan kell kinézzen:



Így már sajnos nem látjuk, hogy mennyi idő megy el a függvényhíváshoz köthető műveletekkel, de a többi sor eredménye pontosabban követi a valóságot. Az alkalmazás futási ideje is mérséklődik, 2.2 másodpercről 1.3 másodpercre csökkent a felhasznált CPU idő.

4. Feladat – Teljesítményoptimalizálás

A harmadik projekt szintén a teljesítménymérés témaköréhez kapcsolódik, de kicsit másképpen, mint a Fibonacci algoritmus. Ebben az esetben az algoritmus lassúságát nem a rekurzió, hanem a rosszul megírt és feleslegesen annotált, debug információkkal feldúsított forráskód okozza. Állítsuk át a kezdőprojektet a Profiling_Students projektre!

A programban diákok (Student) adatait (keresztnév, családnév, születési év és számított mezőként a név és az életkor) kezeljük. A név és az életkor számításánál a bonyolult művelet szimulációjaként bevezettünk egy Thread.Sleep utasítást, ami lassítja a visszatérési érték kiszámítását (a lassítás a SleepTime statikus változó megadásával szabályozható). Az egyszerűbb hibakeresés érdekében a Student osztályban felülírtuk a ToString metódust, így a diák kiíratásakor minden adatát láthatjuk.

A diákok listáját a StudentManager osztály kezeli. Ennek főbb részei:

- AddStudent: hozzáad egy új diákot a meglévőkhöz
- RemoveStudent: törli a megadott diákot a listából
- GenerateRandomStudents: néhány keresztnév és vezetéknév kombinálásával véletlenszerű korú diákokat generál
- GetStudentByName: visszaad név alapján egy diákot
- GetOldestStudentByAge: visszaadja a legöregebb diákot
- ListStudentsByAge: kilistázza a diákokat életkor szerint

A Program.cs-ben lévő logika képes mind az életkor szerinti listázást, mind véletlenszerű emberek többszöri keresését megvalósítani. Érdemes az egyik funkciót mindig kicommentezni, hogy a teljesítménnyel kapcsolatos eredmények kevésbé torzuljanak. A teljesítménymérést segítendő a program az elején elindít egy StopWatch-ot, amit a végén leállít és kiírja az eltelt időt. A StopWatch bevált módszer a futási idő mérésére.

Futtassuk le az alkalmazást, próbáljuk ki a két funkciót külön-külön! (Használjuk a Ctrl+F5 billentyűkombinációt a futtatáshoz!) Elemezzük ki a teljesítménymérés adatait! Látszik, hogy mindkét funkció esetében jelentős lassulást okoz a számított property-k lekérdezése, ahogy arra számítani is lehetett. Sőt, kissé váratlan helyeken is beleszámít a lassulásba a származtatásnál bevezetett késleltetés. Nézzük meg pl. a GetStudentByName függvényben a foreach-en belüli if feltételt! Ez az „ártatlan” feltétel igen sokszor lefut, mivel a listánk nem rendezett és minden keresésnél iterálunk rajta, amíg meg nem találjuk az adott elemet.

Érdemes kipróbálni a Student.SleepTime változót különböző értékekre beállítani és újra lefuttatni a méréseket. A késleltetés növekedésével teljesen átalakul(hat) a teljesítményprofil! Mi lenne a legegyszerűbb gyorsítási lehetőség?

A kódon látszik, hogy aki írta, nagyon szerette volna, ha részletes információkat lehetne kapni a futás alatt minden adatról, feltételezhetően az algoritmusban történő hibakeresés érdekében. Ennek fényében került be pl. a Debug.WriteLine utasítás is, ami a debuggolás alatt a Visual Studio Output ablakában írja ki az üzeneteit. Ezekre éles alkalmazásban ebben a formában nincsen már szükség főként azért, mert a kiírás hagyományosan nagyon időigényes művelet (a matematikai operációkhoz képest). A profiling sokat segíthet abban, hogy ezek a kód rejtett részein megbújó hibákat megtaláljuk.

Önálló feladatok

1. Tesztelés: gyökvonás

A feladat az *IntMagic* osztály *calculateRoot* metódusának implementálása és unit tesztelése. A metódus visszaadja egy szám valahányadik gyökét. Ahogy az előző esetben, most is használhatod a Math könyvtár függvényeit az implementációhoz.

Rajzold fel a bemeneti teret, oszd fel partíciókra, állapítsd meg a határokat! Válassz ezekhez olyan bemeneti értékeket (ismét párokat), amelyek valamennyit lefedik legalább egyszer. Ne feledd, hogy negatív számból nem minden esetben tudunk gyököt vonni. Nagyon kis vagy nagy értékek esetén a művelet pontosságát is ellenőrizheted.

Ha ezzel megvagy, készítsd el a teszteseteket és győződj meg, hogy sikeresen lefutnak!

2. Fibonacci optimalizálás

A feladat a korábban átbeszélt Fibonacci algoritmus optimalizálása. Kiderítettük, hogy miért lassú az alkalmazásunk, már csak ki kell javítanod!

Tippek:

- Egy különleges jelenség, amiről lehet, hogy már tanultatok: egyes algoritmusok gyorsíthatóak a memóriaigényük növelésével. A kiinduló alkalmazásunk minden Fibonacci számot többször is kiszámít.
- A rekurzív algoritmusok gyakran elegánsak, de nem mindig tartoznak a leghatékonyabb megoldások közé.

Ha sikeresen kiküszöbölöd az alkalmazás lassúságát, lehet, hogy növelned kell a mintavételezési rátát, hogy pontos eredményt kapj. Sőt, lehet, hogy olyan gyorsan befejezi a futást, hogy nem sikerül elég mérési pontot gyűjtsön a pontos eredményhez. Egy lehetséges megoldás erre az lenne, ha egy ciklusban sokszor egymás után ismételten meghívnád a kérdéses függvényt. Valahogy így:

```
static void Main(string[] args){

FibonacciMagic fs = new FibonacciMagic();
int degree = 40;
double result = fs.estimateGoldenRatio(degree);
List<double> results =  new List<double>();
for (int i = 0; i < 10000; i++)
   {
         results.Add(fs.estimateGoldenRatio(degree));
   }
Console.WriteLine("Estimate of the golden ratio using Fibonacci numbers of degree {0}:
{1}", degree, result);
}
```

Ez sajnos ebben az esetben nem célravezető, mert a fordító optimalizál és észreveszi, hogy már kiszámította ezt az értéket. Sőt, meg tudja állapítani, hogy a results listát egyszer se használjuk, ezért lehet, hogy a ciklushoz tartozó kód egyszer se fut le. Örülhetünk, hogy a fordító számtalan automatizmussal gondoskodik az alkalmazásunk hatékonyságának növelésére. Ha ennek ellenére lassú alkalmazást készítenénk, most már tudjuk, hogy miként lehet megmérni, hogy miért lassú.

Optimalizáld a végrehajtást, a rekurzív függvény átírásával hagyományos iteratív megoldásra, vagy a korábbi eredmények eltárolásával csökkentsd a rekurzió mélységét!

3. Hallgatók kezelése - optimalizálás (plusz pontért)

Az alkalmazás teljesítményérésének eredményei a birtokunkban vannak. A feladat ennek fényében a teljesítmény optimalizálása. Optimalizáld a kiadott forráskódot figyelembe véve a következőket:

- Nincs szükség Debug jellegű információkra a futás alatt (az algoritmusról tudjuk, hogy jól működik)
- A Student osztályt tilos bármilyen tekintetben módosítani
- Linq használata nem engedélyezett az átírás során
- Észszerű mértékben engedélyezett extra memória felhasználása a műveletek gyorsítása érdekében

Érdemes meghagyni az eredeti StudentManager osztályt és készíteni egy másolatot belőle, azon elvégezni a módosításokat, mert így könnyebb lesz összehasonlítani az eredetivel a teljesítmény javulását. Használd a StopWatch-ot, hogy teszteld a gyorsulást a profiler alkalmazás nélkül is!

A feladat sikeres elvégzéséhez legalább 3 különböző optimalizálást kell alkalmaznod!