**Szegedi Tudományegyetem**

**Informatikai Intézet**

**Szakdolgozat**

**Horváth Bendegúz**

**2022Szegedi Tudományegyetem**

**Informatikai Intézet**

**Ládapakolás túltöltéssel**

Szakdolgozat

Készítette: Témavezető:

**Horváth Bendegúz Dr. Balogh János**

Programtervező informatikus egyetemi docens

szakos hallgató

**Szeged**

**2022**

**Feladatkiírás**

A ládapakolási feladat egy (erősen) NP-nehéz probléma, amelynek klasszikus egydimenziós változatában azonos kapacitású ládákba kell (a ládakapacitást nem meghaladó méretű) input elemeket bepakolni. A cél, hogy a lehető legkevesebb számú ládába. (Átfedés és túlpakolás nem megengedett a klasszikus változatban.)

Ennek egy olyan változatát tanulmányozták a közelmúltban egy publikációban (Kelin Luo, Frits C. R. Spieksma: Online Bin Packing with Overload Cost, CALDAM 2021: 3-15), ahol a ládák túlpakolása, túltöltése megengedett, azaz a ládakapacitást meghaladhatja a pakoló (algoritmus). Azonban ennek ára, plusz költsége van: a túltöltésért egy, a túltöltés mértékével (azaz a kapacitás fölötti résszel) egyenesen arányos, egységenként c költséget kell fizetnünk. (Itt c előre adott, fix konstans, és a túltöltés költsége a célfüggvényben hozzáadódik a felhasznált ládák 1-1 költségéhez. Így a minimalizálandó célfüggvény két komponensből áll, nevezetesen a felhasznált ládák száma és a túltöltések költsége teszi ki együttesen.)

A hallgató feladata egyrészt feldolgozni a cikk alapján a szükséges mértékben a kapcsolódó szakirodalmat, megérteni a cikkben elemzett First-Fit Algorithm with Fixed Overload (FFO) algoritmust, amely a First-Fit algoritmusnak a feladatra adaptált változata.

A másik feladat kiválasztani egy benchmark példahalmazt, és ennek feladatain keresztül elemezni az FFO algoritmus teljesítményét, és összehasonlítani más, kiválasztott klasszikus algoritmusokéval. Azaz implementálni azokat, és a példahalmazon végzett futtatások, tesztek alapján összehasonlításokat eszközölni, és mindezt elemezni.

**Tartalmi összefoglaló**

* A téma megnevezése:

Ládapakolás túltöltéssel

* A megadott feladat megfogalmazása:

A ládapakolási feladat egy NP-nehéz probléma, amelynek klasszikus egy-dimenziós változatában azonos kapacitású ládákba kell input elemeket bepakolni. A cél, hogy a lehető legkevesebb számú ládát használjuk fel. A feladat egy túltöltéses változatát a szakirodalomban nemrég vizsgálták (Kelin Luo, Frits C. R. Spieksma: Online Bin Packing with Overload Cost, CALDAM 2021: 3-15), A feladat ennek feldolgozása volt, az ottani algoritmus megértése, implementálása, más algoritmusokkal való összehasonlítása. A ládapakolási feladatot Python nyelven megírva.

* A megoldási mód:

A ládapakolási feladatra a cikkben tárgyalt First-Fit algoritmust Python nyelven megírva implementáltam. A feladatra különböző más algoritmusok segítségével is mevalósítottam, implementáltam és ezek eredményeit összahasonlítottam: ezek a First-Fit, Best-Fit és Worst-Fit algorimtusok. Ezek a módszerek ugyanarra a bemenetre vannak letesztelve (ehhez egy ún, benchmark példahalmazt választottam), így össze lehet őket hasonlítani hatékonyság alapján (pl. grafikonok segítségével is).

* Alkalmazott eszközök, módszerek:

Az általam írt kód Python 3.10.4 –ben készült, a program elkészítéséhez először az OS és a SYS könyvtárakat használtam fel. A matematikai műveletekhez a NumPy könyvtárat és a Math csomagot használtam fel. Az eredmény diagramokon való ábrázolásához a MatPlotLib könyvtárat használtam. Fejlesztő környezetben a JetBrains által forgalmazott PyCharm Professional IDE-t [2] használtam. Verziókövetéshez Git-et használtam, a kódot egy publikus GitHub Repository-ban [3] tároltam.

* Elért eredmények:

A fent említett ládapakolási algoritmusok implementálása, majd egymással való összehasonlítása, e célból pl. grafikonok segítségével.

* Kulcsszavak:

ládapakolás, ládák túltöltése, First-Fit, Best-Fit, Worst-Fit, bináris kresés, összehasonltás, grafikon, Python

Tartalomjegyzék

[**1. Bevezetés** 6](#_Toc103329804)

[**2. Felhasznált algoritmusok elemzése** 7](#_Toc103329805)

[**2.1. Motiváció** 7](#_Toc103329806)

[**2.2. Versenyképességi arány** 8](#_Toc103329807)

[**2.3. Ládapakolási algoritmusok** 10](#_Toc103329808)

[**2.4. First-Fit** 11](#_Toc103329809)

[**2.5. Best-Fit** 14](#_Toc103329810)

[**2.6. Worst-Fit** 15](#_Toc103329811)

[**3. Fejlesztői dokumentáció** 16](#_Toc103329812)

[**3.1. Fejlesztői környezet** 16](#_Toc103329813)

[**3.2. A futtatáshoz szükséges bemeneti adatok – python könyvtárak** 17](#_Toc103329814)

[**3.3. Implementáció** 20](#_Toc103329815)

[**3.3.1. First-Fit algoritmus implementációja** 20](#_Toc103329816)

[**3.3.2. Best-Fit algoritmus implementációja** 21](#_Toc103329817)

[**3.3.3. Worst-Fit algoritmus implementációja** 22](#_Toc103329818)

[**3.3.4. Beszúró rendezés bináris kereséssel** 24](#_Toc103329819)

[**3.3.5. Túltöltés meghatározása** 24](#_Toc103329820)

[**3.4. Eredmények** 25](#_Toc103329821)

[**3.5. Bővítési lehetőségek** 30](#_Toc103329822)

[**4. Forrásjegyzet** 31](#_Toc103329823)

[**4.1. Saját hivatkozások** 31](#_Toc103329824)

[**4.2. Szakirodalmi hivatkozások** 31](#_Toc103329825)

# **1. Bevezetés**

A szakdolgozat a modern ládapakolási algoritmusok bemutatását összefoglaló konzolos program, mely python nyelven lett megírva.

Egyetemi tanulmányaim során mindig érdekeltek az algoritmusok, úgy gondolom, hogy sokszor fontos problémára nyújtanak megoldást. Az emberek célja általában, hogy minél hatékonyabban – gyorsan, és kevés erőforrást felhasználva - elérjék a kívánt eredményt.

A szakdolgozatomban három darab ládapakolási algoritmust mutatok be: First-Fit, Best-Fit és Worst-Fit.

A klasszikus online ládapakolás problémában a beérkező tárgyak egyenként jelennek meg, egymás után. Ezeket az egynél nem nagyobb méretű, érkező tárgyakat kell bepakolni egységnyi méretű ládákba úgy, hogy az egyes ládákba pakolt elemek teljes mérete ne haladja meg annak kapacitását. A cél az, hogy minimalizáljuk a felhasznált (nemüres) ládák számát.

Ennek a klasszikus problémának egy változatát, az online ládapakolási problémát vizsgáljuk túltöltési költséggel. Ebben a feladatban megengedhetjük, hogy a ládákba pakolt elemek összegzett mérete meghaladja az -et.

Az online feladat esetén fontos, hogy a tárgyat végleges helyére (ládájába) kell rakni a soron következő elem megjelenése előtt. A szakdolgozat célja, hogy a dokumentációban megemlített rendező algoritmusokat részletesen bemutassa és összehasonlítsa.

# **2. Felhasznált algoritmusok elemzése**

A két legismertebb, vagy talán legelterjettebb online ládapakolási algoritmusok a *First-Fit* és a *Best-Fit* algoritmus.

A First-Fit (FF) algoritmus a következőnek érkező elemet az első olyan tárolóba teszi (a ládák létrehozásának sorrendjében), ahova belefér.

A Best-Fit (BF) algoritmus a következőnek érkező elemet abba a legjobban teli ládába teszi, amelyikbe az elemet még be lehet helyezni. Ha nincs ilyen láda, ahova egy elem belefér, akkor mindkét algoritmus nyit egy új ládát, és abba pakolja az érkező elemet.

Ezen kívül a Worst-Fit (WF) algoritmus jelenik még meg, ez tulajdonképpen a Best-Fit algoritmus alapötletének fordítottját végzi, mivel a legkevésbé teli ládát választja több pakolási lehetőség esetén. Abban az esetben, ha nem talál olyat, amelybe beleférne az elem, szintén nyit egy új tárolót és abba helyezi bele az elemet.

## **2.1. Motiváció**

Tekintsük a következő lehetséges alkalmazást. Egy többprocesszoros rendszerben a tárolást igénylő, azaz tárolókapacitást használó feladatok egyenként lépnek be a rendszerbe, és minden egyes feladatot hozzá kell rendelni a processzorok egyikéhez.

Minden processzornak van egy adott, szolgáltatási kapacitása és fix bekapcsolási/kikapcsolási költséget számol fel. Túlterhelési (túltöltési) költséget akkor kell felszámolni, ha egy processzorhoz rendelt feladat által igényelt kapacitás összege meghaladja a kiszolgáló kapacitását.

Célunk az összes feladat összköltségének minimalizálása. Ez a motivációja a túlterhelési költséggel járó online ládapakolási problémának: azaz ott minden egyes láda egy adott kapacitással rendelkező processzort képvisel, és akkor van túlterhelési költség, ha az adott ládába csomagolt elemek összmérete meghaladja a láda kapacitását.

Vagyis a túltöltéses (túlterheléses) ládapakolás esetében az egy tárolóba pakolt elemek összértéke meghaladhatja a láda kapacitását. Minden ládának van egy költsége, amelyet a használatakor fizetni kell, mely két részből tevődhet össze:

1. Minden ládának van egy fix költsége: Az 1 kapacitású tároló megnyitásának , azaz használatának ára 1 - ez nem függ a belepakolt elemek méretétől és darabszámától, akár ha egyetlen elemet is rakunk bele, ezt ki kell fizetni.
2. A láda kapacitásának túltöltése, vagyis a láda kapacitásának túllépése előírt költséggel jár. Az előző költségen kívül fizetünk minden olyan ládáért, amely túlterhelt, túltöltött. Az -et meghaladó minden töltésért arányos költséget fizetünk. Tehát a túlterhelés költsége lineárisan függ a túlterhelés méretétől.

Megjegyezzük, hogy nem túltöltött ládáknál csak az első, fix költséget kell kifizetnünk, mint ládahasználati díjat, míg a második költség csak akkor adódik hozzá a láda költségéhez, ha a benne lévő elemek mérete meghaladja a láda kapacitását, azaz -et.

A cél a felhasznált tárolók összköltségének minimalizálása - két költség összegét minden ládára összegezve.

A feladat egy konkrét példánya olyan, hogy kapunk egy elemből álló sorozatot, ahol nem tudjuk előre, hogy mekkora értéke. Minden elemet visszavonhatatlanul be kell pakolni egy ládába mielőtt a következő elem megérkezik. Végtelen számú egyforma méretű tároló áll a rendelkezésünkre, minden egyes tárolónak a kapacitása . Minden tároló megnyitásának, így használatának a költsége . Ezen kívül van még egy adott, racionális túltöltési költség, amely a túltöltés minden egyes egységére vonatkozik.

Egy láda akkor kerül felhasználásra vagy megnyitásra, ha legalább egy tárgyat tartalmaz. Egy már felhasznált láda költsége ahol az ládában lévő elemek összmérete. A cél az, hogy pakoljunk minden egyes tárgyat a ládába úgy, hogy minimalizáljuk a felhasznált ládák költségének összegét. Ezt a problémát lineáris túlterheléses online ládapakolásnak nevezzük, az angol megfelelője „online bin packing problem with linear overload cost”, röviden *BPOC*.

## **2.2. Versenyképességi arány**

A fent említett algoritmusoknak az elemzése egészen Ullman munkájáig [[17]] nyúlik vissza, továbbá Sgall cikkében [[16]] is megtalálható az online ládapakolásról egy összefoglaló áttekintés.

Egy online algoritmus teljesítménye mérhető az aszimptotikus versenyképességi arány, valamint az abszolút versenyképességi arány alapján. Johnson és szerzőtársai [[12]] bebizonyították, hogy mind az FF, mind a BF aszimptotikus versenyképességi aránya 1,7.

A jelenlegi legjobb ismert alsó korlát az aszimptotikus versenyképességi arányra vonatkozóan, amely bármely online algoritmusra teljesül: 1,54278 [[2]]. A legjobb ismert versenyképességi arányú algoritmus versenyképességi aránya pedig 1,578 [[1]].

Az aszimptotikus versenyképességi eredményhez hasonlóan az FF és a BF abszolút versenyképességi aránya is 1,7 ezt bizonyítják a [[7]] és [[8]] forrásokban.

A közelmúltban Balogh és szerzőtársai [[3]] egy olyan online ládapakoló algoritmust terveztek, amelynek abszolút versenyképességi aránya 5/3, ami a lehető legjobb. Az Öt-harmad (Five-Third) algoritmusnak nevezett algoritmusuk fő gondolata az FF használata, és néhány láda fenntartása kifejezetten 1/2-nél nagyobb méretű elemek számára.

Ebben a dolgozatban a BPOC abszolút versenyképességi arányát vizsgáljuk.

Az [[LS21]] cikkben a szerzők a versenyképességi arány alsó korlátokat mutatták meg bármely determinisztikus BPOC algoritmushoz, és felső korlátokat - algoritmust is megadva - is bizonyítottak. Összefoglalva a következő tételt bizonyították az alsó korlátokra:

Legyen a következőképpen definiálva:

Az [[LS21]] cikkben azt bizonyították, hogy ha a fenti intervallumokba esik, akkor nincsen jobb abszolút versenyképességi arányú online algoritmus a BPOC-feladatra, vagyis ezek alsó korlátok a c különböző eseteire.

A felső korlátokhoz a cikkben a következő, *First-Fit Algoritmus Fix Túltöltési Költséggel* nevű algoritmust definiálták. (First-Fit Algorithm with Fixed Overload, röviden FFO). Hasonlóan a klasszikus online ládapakolási probléma First-Fit algoritmusához (FF), az FFO minden egyes elemet az első olyan nyitott ládába pakolja, amelybe belefér; ha nem fér bele egyetlen jelenleg megnyitott tárolóba sem, akkor nyit egy új tárolót és belepakolja.

A különbség az, hogy az FFO-ban a bármelyik ládához rendelt elemek teljes mérete meghaladhatja a tároló 1 kapacitását.

Legyen a következőképpen definiálva:

Az [[LS21]] cikkben azt bizonyították a szerzők, hogy az FFO algoritmus abszolút versenyképességi aránya a fenti , ha *c* a fenti intervallumokba esik. Tehát ezzel felső korlátokat is megadtak a BPOC feladatra, bárhová is essen . pedig alsó korlát, bárhova is essen .

## **2.3. Ládapakolási algoritmusok**

A klasszikus feladaton kívül online ládapakolási feladat számos változatát vizsgálták. Az egyik változat az, amikor többféle ládaméret, azaz ládakapacitás is lehet - az 1-en kívül, de minden láda legfeljebb 1 kapacitású, és az algoritmus dönti el, hogy melyiket használja. Ez az úgynevezett online változó méretű ládapakolási probléma. Kinnerly és Langston [[13]] egy módosított FF-típusú algoritmust használt erre, az FF-et a felhasználó által meghatározott kitöltési tényező (FFf) segítségével, és bizonyítják, hogy ez az algoritmus -versenyképes, amikor .

Csirik [[5]] javasolta a Variable Harmonic (VH) algoritmust, és megmutatta, hogy az aszimptotikusan -versenyképes (bizonyos ládaméretekre), lásd még Seiden cikkében [[15]] ennek pontos elemzését.

Egy további, másik változata a ládapakolási feladatnak a nyitott végű ládapakolási probléma, amely lehetővé teszi, hogy felülírjuk a kapacitást egy meghatározott módon. Yang és Leung [[18]] az online rendezett nyitott végű ládapakolás feladatot vizsgálja (OOBP), ahol megengedett a kapacitás megsértése oly módon, hogy az egyes ládákban lévő elemek mérete -nél kisebb legyen, miután eltávolítjuk a ládabeli a legnagyobb elemet.

Epstein és Levin [[9]] a továbbiakban a nyitott végű tárolók csomagolási problémájának két másik változatát vizsgálja. Az egyik az erős nyitott végű ládapakolási probléma (strong open-end bin packing problem, SOBP). Itt az egyes ládákban lévő tárgyak súlyának kisebbnek kell lennie 1-nél, ha a legkönnyebb elemet - legkisebb méretűt - eltávolítanánk. Valamint a másik az ún. lusta ládafedési probléma (lazy bin covering problem, LBC), amely tartalmaz egy további megkötést, hogy a legkönnyebb elem eltávolítása után az egyes ládákban lévő tételek összmérete nem lehet kisebb, mint , kivéve egyetlen láda esetén.

Vannak olyan cikkek is a szakirodalomban, amelyek a felhasznált ládák költségére összpontosítanak. Li és Chen [[14]], Epstein és Levin [[10, 11]], Cambazard és szerzőtársai [[7]] cikkeinek mindegyike a tárolóba pakolt elemek különböző költségszerkezeteit vizsgálja. Különösen Epstein és Levin [[11]] munkája releváns a problémánk offline változata szempontjából.

Ők egy olyan általános környezettel foglalkoznak, ahol különböző költségű és méretű tárolók fordulnak elő, valamint az elemek egy halmaza adott, ahol a cél a használt tárolók költségének minimalizálása.

## **2.4. First-Fit**

A First-Fit algoritmus minden újonnan érkező elemet az első olyan ládába pakolja, amelybe befér vagy, ha nem fér be sehová, akkor nyit egy új ládát.

Az FFO algoritmus annyiban más, mint az FF algoritmus, hogy megengedi a túltöltést egy O(c) méretű, előre meghatározott értékben. Ez annyit jelent, hogy bár az elemeink mérete 0 és 1 között van, a láda mérete lehet 1-nél magasabb. Egészen pontosan helyett méretű ládába pakolunk. Bár nagyon hasonlóan néz ki a nagy O jelölés, itt nem a nagy ordót jelenti, ez az O(c) egyszerűen egy c-től függő szám:

Tehát ez azt jelenti, hogy a ládába tölthetünk elemeket 1 méret fölött, azonban az 1 szintet meghaladó elemméretekért annak c-szeresét kell fizetnünk. Azaz a láda költsége nem 1 lesz, hanem 1 + c-szer a „túllógás” mérete, ha mondjuk s szintig pakoljuk a ládát, és s > 1, akkor 1 + c \* (s - 1) lesz a láda költsége a klasszikus feladat 1 költségével szemben.

Például:

* Ha c = 1,25, akkor az első ágon vagyunk (a fenti definíció első sorában), egyetlen ládánk lesz, amelyet az „égig” pakolhatunk.

Ha mondjuk s = 100 magasságig pakolunk, akkor a láda költsége 1 + (s - 1) \* c = 1 + (100 - 1) \* 1,25 = 1 + 99 \* 1,25 = 124,75 lesz.

Csak egyetlen ládánk lesz ekkor, minden elemet ebbe pakolunk bele. Az 1 fölötti túllógásért fizetjük az 1,25-ször a túllógás költséget. Viszont ez „megéri” nekünk, mivel c legfeljebb 1,5 ezen az ágon, így egy legfeljebb 1,5 versenyképés algoritmust kapunk. Azaz a mi algoritmusunk, FFO összes költsége legfeljebb 1,5-szerese lesz az optimális pakolás költségének, OPT-nak.

* Ha c = 1,6, akkor a második ágon vagyunk a definíció második sorában, tehát ekkor az algoritmusunk úgy fog kinézni, mint a klasszikus FF algoritmus, de 1 magasságú ládák helyett 1 + 1/c = 1 + 1/1,6 = 1,625 magas ládákba fog pakolni. Azaz minden egyes láda amit tekintünk, 1,625 magas lesz ebben az esetben. Úgyis mondhatjuk, hogy minden egyes tekintett láda kapacitás 1,625 lesz, 1 helyett, és ilyen szabályok mellett pakolunk FF szabállyal. Persze kevesebb, mint 1,625 összmennyiségű elemet rakhatunk bele, sőt, a legtöbb ládát vélhetően nem tudjuk 1,625 magasságig megpakolni, mint ahogy a klasszikus esetben sem tudjuk 1 magasságig pakolni a ládáinkat, általában csak kisebb szintig.

Ennek ellenére az 1 szint feletti, túllógó elemekért fizetnünk kell.

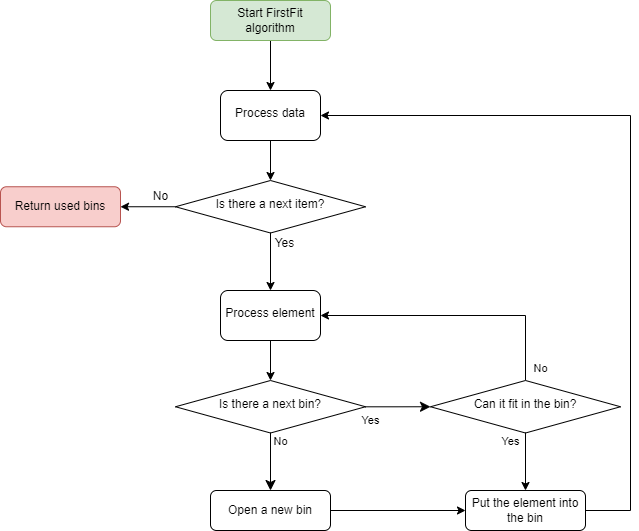
Ha mondjuk c = 1,6 és egy ládába s = 1,3 szintig rakja az elemeket az FFO algoritmus, akkor 1 + c \* (s - 1) = 1 + 1,6 \* 0,3 = 1,48 lesz a láda költsége. Így fizetünk plusz 0,48 költséget, 1 helyett 1,48-ot a ládáért cserében viszont 1,3 szintig pakolhatjuk, nem kell új ládát nyitni az 1 fölött lévő elemekhez.

Természetesen c értékét előre ismerjük. Azaz így az algoritmusunk először eldönti, hogy „melyik” ágon vagyunk, és annak megfelelően 1+O(c) magas ládákba pakol, minden láda ilyen kapacitású lesz.

Tehát az algoritmus a túltöltési költség esetén végtelen mennyiségű elemet pakol egy ládába. Azaz csak egyetlen ládát használ ebben az esetben az összes elem pakolására.

Ha c értéke meghaladja a értékét, de nem nagyobb, mint , akkor a túltöltés költsége lesz, vagyis az 1-et meghaladó értékék plusz költsége lesz, azaz -vel kell szorozni az 1-et meghaladó elemeket. Ezenkivűl ha a intervallumba esik, akkor ez a költség lesz, míg ha , akkor pedig ez az érték, a túltöltés költsége.

Ahogy az első ábrán látható az alábbi Flowchart diagram. (1. ábra) Ez szemlélteti az algoritmus működését:



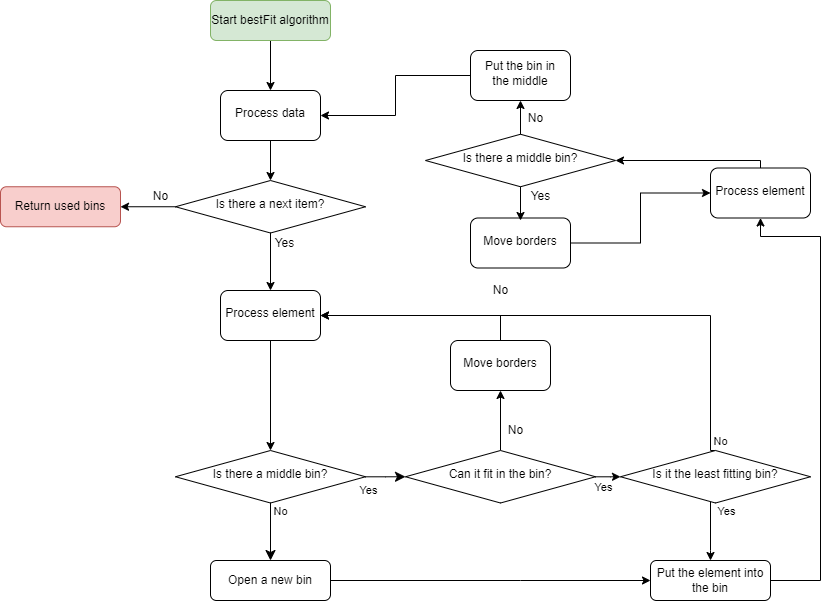
1. ábra A First-Fit algoritmus flowchart diagramja

## **2.5. Best-Fit**

A Best-Fit algoritmus minden az adott sorrend szerint érkező elemet belepakol abba a ládába, amelybe belefér, és ezen felül a láda szintje a lehető legnagyobb lesz az elem pakolása utána. Más szóval abba a ládába teszi az elemet, amelybe még belefér úgy, hogy minél kisebb üres hely maradjon az adott ládában. Abban az esetben, ha nincsen olyan láda, amelyre ez teljesül, akkor nyit egy új ládát és abba pakolja bele az elemet. Az algoritmus célja, hogy ezzel a módszerrel minimalizáljuk a felhasznált ládák számát, úgy, hogy minden elemet felhasználunk.

Túltöltés esetén itt is a First-Fit-hez hasonlóan a legfontosabb eltérés az, hogy az eddigi ládaméret 1 volt, azonban a túltöltés miatt egy függvény által meghatározott túltöltéssel növekszik a méret 1 + O(c)-re. A [2.4 First-Fit](#_2.4._First-Fit) alogritmusnál leírt O(c) függvényhez hasonlóan meghatározható egy függvény a Best-Fit algoritmushoz is.

A kettes ábrán látható Flowchart diagram szemlélteti az algoritmus működését:



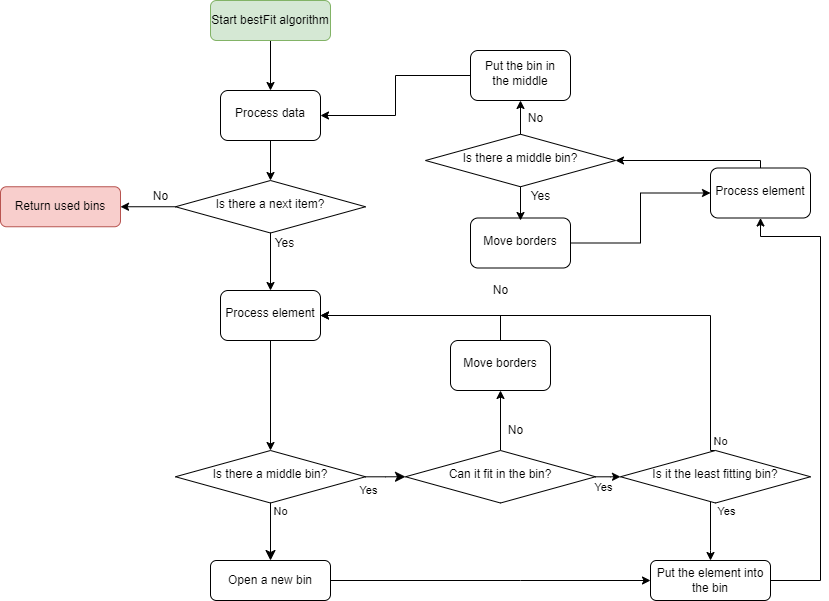
2. ábra: BestFit algoritmus flowchart diagramja

## **2.6. Worst-Fit**

A Worst-Fit algoritmus a Best-Fithez hasonlóan a láda fentmaradó része alapján helyezi el az elemet, azonban ez az algoritmus a legnagyobb fennmaradó részt tekinti optimálisnak. Tehát itt is az adott sorrend szerint érkeznek az elemek, ezeket belepakolja abba a ládába, amelybe belefér, és ezen felül a láda szintje a lehető legkisebb lesz. Az elemeket minél kisebb magassággal próbálja elhelyezni, a célja a minél nagyobb üres rész megtartása. Ez az jelenti, hogy az algoritmus mindig azt a tároló ládát választja, amiben a legnagyobb lesz a fennmaradó rész, úgy, hogy az elemet beleteszi.

Az algoritmus célja, hogy minden elemet felhasználva minimalizáljuk a felhasznált ládák számát.

Az alábbi Flowchart diagram (3. ábra) szemlélteti az algoritmus működését:



3. ábra: Worst-Fit algoritmus flowchart diagramja

# **3. Fejlesztői dokumentáció**

## **3.1. Fejlesztői környezet**

A szakdolgozat megírásához a Python [1] programozási nyelvet választottam. Guido van Rossuam holland programozó kezdte el fejleszteni a technológiát 1989-ben, majd 1991-ben hozta nyilvánosságra. Választásom legfőbb szempontja a nyelv magasszintű adatstruktúrái, valamint a kevésbé komplex szintaxisa volt. Rövid programok írása gyors, egyszerű és jól áttekinthető, ezenkívül a Python egy folyamatosan fejlődő, gazdag felhasználó és fejlesztő bázissal rendelkező teljesen ingyenesen, és korlátozások nélkül használható programozási nyelv. Megemlítendő még, hogy a Python dinamikus adattípusokat használ, tehát a progamozó által használt minden objektumnak a végrehajtáskor jól definiált típusa van - nem kell előre definiálni, így nagyban megkönnyíti a nyelv használhatóságát.

Az általam írt kód Python 3.10.4 –ben készült, a programban felhasznált adatok előkészítéséhez az OS és a SYS nyújtanak segítséget. Az előbbi különböző funkciókkal teszi lehetővé az operációs rendszerrel való kommunikálást. E könyvtár segítségével operációs rendszertől függően tudunk műveleteket végrehajtani. Ilyenek például a fájlok és mappák tartalmának lekérdezése, törlés, létrehozás, átnevezés funkciók. Ezen kívül tudunk a mappaszerkezetben lépegetni, a fájlok méretét meghatározni. Az utóbbi segítségével pedig a futási környezet különböző részeihez kapunk hozzáférést, például lehetővé teszi interpreterrel való műveletek végrehajtását.

A matematikai műveletek végrehajtását a Matplotlib könyvtár és a Math csomag teszi lehetővé. Ezekben találhatóak különböző matematikai függvények, melyek megkönnyítik az algoritmusok végrehajtását. Az eredmény diagramokon való ábrázolásához a Matplotib könyvtárral van kivitelezve, mivel ez a legelterjedtebb könyvtár erre célra. A könyvtár jól van dokumentálva, mely nagyban megkönnyíti a program áttekinthetőségét.

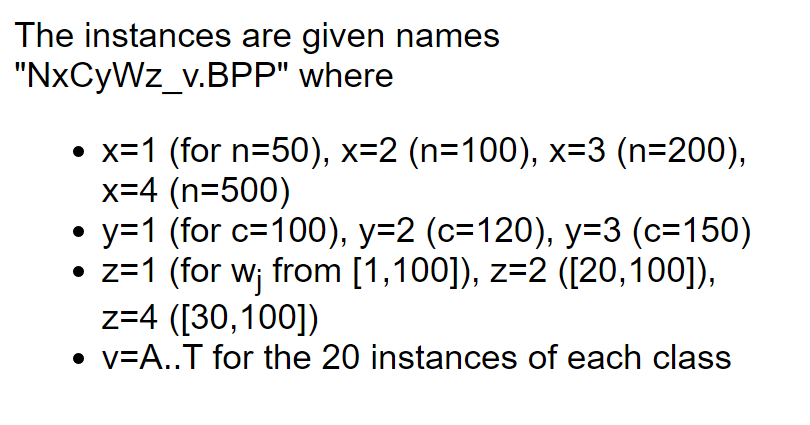
A fejlesztő környezet a JetBrains által forgalmazott PyCharm Professional IDE [2], mely nagyban megkönnyíti a kód karbantartását.

Verziókövetéshez Git-et alkalmazok, mivel ez egy elterjedt és egyszerűen használható rendszer, a kódot egy publikus GitHub Repository-ban [3] tárolom.

## **3.2. A futtatáshoz szükséges bemeneti adatok – python könyvtárak**

A megírt algoritmusok megfelelő teszteléséhez többféle bemeneti fájlra van szükség, melyeket külső forrásokból gyűjtöttem össze. Konzulensem tanácsára az M. Delorme, M. Iori és S. Martello BPPLIB ládapakolás probléma könyvtárában kerestem megfelelő bemeneti csomagot, a választás a SCHOLL 1 [4] nevezetű könyvtárra esett, mivel ez egy sokszínű példatár. Ezen kívül a könyvtár rendelkezik optimális megoldással minden egyes bemeneti fájlhoz.

A könyvtár 720 darab .BPP terjesztésű fájlból áll, ezeknek mérete(n) lehet 50, 100, 200 vagy 500 sor, soronként 1 darab elemmel. Az elemek mérete(w) fájlonként a [1,100], [20,200], [30,100] intervallumok valamelyikébe esik. A harmadik érték a tároló ládák kapacitása(c), ezek lehetnek 100, 120 vagy 150 méretűek. Ezek az adatok kiolvashatóak már a fájlok elnevezéséből is, melyet a 4-es ábra szemléltet:



4. ábra: Bemeneti fájlok elnevezésének rendszere

Itt „x” értéke jelöli a sorszámot, „y” a ládák kapacitását, valamint „z” azt, hogy az elemek mely intervallumból származnak. A feladattípus sorszámát pedig „v” jelöli.

A fenti meghatározások alapján minden típusból 20 eset van, összesen 36 feladattípus van a könyvtárban, 720db feladattal.

A kód legvégén történik a fájlok beolvasása, mely két függvény segítségével történik meg:

1. *read\_text(fpath, x):*

A függvény beolvassa az adott fájl első két sorából az elemek darabszámát és a maximum méretüket, majd beolvassa az értékeiket. Ezután a beolvasott értékeken egy for ciklus segítségével végig megyünk és átalakítjuk őket szám formátumúvá. Ezután a számolást végző függvények visszatérési értékét egy-egy változóban tároljuk, majd az eredményüket kiíratjuk a konzolra. A függvény egy listával tér vissza, mely tartalmazza mind a három végrehajtott algoritmus végeredményeit.

def read\_text(fpath, x):  
 i = 0  
 maxHeight = overload(x)  
 with open(fpath, 'r') as f:  
 f.readline()  
 f.readline()  
 data = f.readlines()  
 dataInt = []  
 for i in range(len(data)):  
 dataInt.append(int(data[i]))  
 resultFirstFit = firstFit(maxHeight, dataInt)  
 resultBestFit = bestFit(maxHeight, dataInt)  
 resultWorstFit = worstFit(maxHeight, dataInt)  
 print(resultFirstFit,' - ',resultBestFit, ' - ', resultWorstFit)  
 return [resultFirstFit, resultBestFit, resultWorstFit]

1. *readDirectory(x):*

A függvény elején deklarálva vannak a későbbiekben használatos listák, majd a program bemeneti fájlok mappáján megy végig egy for ciklus segítségével, ezen belül megvizsgálja, hogy az adott iterációban kapott fájl megfelel-e a bemeneti fájlok típusának. Ha ez a feltétel igaz, akkor meghívja a fentebb leírt függvényt és elvégzi rajtuk a ládapakolási algoritmusokat. Valamint feltölti az éppen kapott értékekkel az ezeket tároló változókat. Ha nem igaz a feltétel, akkor megvizsgálja, hogy „.csv” típusú-e a vizsgált elem. Ha második elágazás igaz, akkor meghívja az optimumot beolvasó függvényt, amely végrehajtja a beolvasást. Mivel a mappában csak ez a két féle típus van, így elég volt két elágazást kezelni. Az ezt követő részekben történik a használt értékeket diagramon való ábrázolása, valamint a végeredményeken történő elemzések kiíratása a konzolra.

Az alábbi függvényrészlet szemlélteti a fájlok beolvasását, majd az adatok listába foglalását.

def readDirectory(x):  
 firstFitList = []  
 bestFitList = []  
 worstFitList = []  
 for file in os.listdir():  
 if file.endswith('.BPP'):  
 fpath = f"{path}\{file}"  
 data = read\_text(fpath, x)  
 firstFitList.append(data[0])  
 bestFitList.append(data[1])  
 worstFitList.append(data[2])  
  
 elif file.endswith('.csv'):  
 fpath = f"{path}\{file}"  
 optimal = readOptimal(fpath)  
  
 avgOfFirstFit = average(firstFitList)  
 avgOfBestFit = average(bestFitList)  
 avgOfWorstFit = average(worstFitList)

A példa könyvtárban megtalálható még a felhasznált SCHOLL 1 könyvtár optimális megoldása is. Mivel a weboldalon egyszerű szövegként szerepel, így Microsoft Excelbe való másolással „csv” formátummá alakítottam a szöveget. Ebben a fájlban szerepel minden bemeneti fájl neve és a hozzá tartozó optimális megoldás.

Az alábbi függvényben látható ennek a beolvasása:

def readOptimal(fpath):  
 with open(fpath, 'r') as f:  
 optimals = []  
 optimalsInt = []  
 a = True  
 while a:  
 line = f.readline()  
 if(not line):  
 a = False  
 break  
 v = line.split(sep=";")  
 optimals.append(v[1])  
 optimals.append(v[3])  
 optimals.append(v[5])  
 for i in range(len(optimals)):  
 optimalsInt.append(int(optimals[i]))  
 return optimalsInt

Mivel a név-érték párosokból csak az értékeket kellett kinyernem, ezért minden 1., 3. és 5. elemét olvastam be a fájlnak. Ezzel létrejött egy számokat tartalmazó tömb, mely ugyanolyan sorrendben tartalmazza az optimális eredményeket, mint a beolvasott mintakönyvtár. A függvény ezzel az értékekkel teli tömbbel tér vissza.

## **3.3. Implementáció**

A kód 3 darab algoritmust tartalmaz, ezek a First-Fit, Best-Fit és Worst-Fit algoritmusok. Legelőször egy minta könyvtár kerül beolvasásra, ez többféle típusú és hosszúságú példa bemeneteket tartalmaz.

Mindhárom algoritmus ugyan ezzel a könyvtárral dolgozik, így a végrehajtás után könnyedén összehasonlíthatóak az eredmények. Az algoritmusok mindegyike 2 függvényre van osztva. Az első függvény –a First-Fit algoritmus kivételével- végrehajt egy beszúró rendezést bináris kereséssel, mely segítségével a ládák telítettség szerinti csökkenő sorrendbe kerülnek, miközben az adott ládapakolási algoritmus is végrehajtódik rajtuk. Ez a végrehajtás a második függvényben van meghívva, ahol megkapja a túltöltés értékét, valamint a bemeneti adatokat. Ez a két lépés minden bemeneti fájl esetén újra lejátszódik, ezzel biztosítva, hogy minden fájlból minden adatot beolvas és ezen az adaton végre van hatjva a rendezés, valamint a ládapakolási algoritmus is.

Miután az összes bemeneti fájlon végigment a program, a ládapakolási algoritmusok végeredményét diagramokon szemlélteti. Mivel a túltöltés mértéke nem állandó, hanem ciklikusan növelve van, így minden túltöltési értékhez készül 3 diagram, mely 1-1 ládapakolási algoritmust ábrázol, valamint készül egy 4. ábra, melyen összesítve látható a 3 algoritmus eredménye. Minden algoritmus más színe, valamint vonaltípusa van, ezzel megkönnyítve a vizualációt.

### **3.3.1. First-Fit algoritmus implementációja**

A *firstFitForOneItem* függvény végrehajta az algoritmust mindegy egyes elemre. Egy *while ciklus* segítségével végig megyünk az eddig már felhasznált ládákon. Minden ládánál megvizsgáljuk, hogy az éppen kapott eleme belefér-e, ha igen, akkor beletesszük és kilépünk a cilkusból. Ellenkező esetben nyitunk egy új ládát és abba helyezzük el az elemet. Végül visszetérünk az eleme elhelyezése után felhasznált ládák számával.

Alább látható a függvény implementációja:

def firstFitForOneItem(bins, remainingSpace, item, maxHeight):  
 j = 0  
 while j < bins:  
 if remainingSpace[j] >= item:  
 remainingSpace[j] -= item  
 break  
 j += 1  
 if j == bins:  
 remainingSpace.append(maxHeight - item)  
 bins += 1  
 return bins

A *usedBins* változó kezdőértéke 1, létrehozzuk a változót a ládák tárolásához, majd az elsőnek definiáljuk a maximum magasságát. Ezt követően egy *for ciklussal* végig megyünk a feldolgozandó adathalmazon, minden elemre meghívjuk a végrehajtó függvényt, majd ennek a visszatérési értékét átadjuk a *usedBins* változónak. Végezetül a függvényünk vissza adja a First-Fit algoritmus által felhasznált ládák számát.

def firstFit(maxHeight, data):  
 usedBins = 1  
 remainingSpace = []  
 remainingSpace.append(maxHeight)  
 for i in range(len(data)):  
 usedBins = firstFitForOneItem(usedBins, remainingSpace, data[i], maxHeight)  
 return usedBins

### **3.3.2. Best-Fit algoritmus implementációja**

A Best-Fit algoritmus implementációja nagyban hasonlít a First-Fit algoritmus implementációjához. A lényeges különbség a végrehajtásban van. A függvény az utóbbihoz hasonlóan 4 értéket kap, melyeket a futás során használ fel. A függvény elején deklaráljuk a futáshoz szükséges változókat, a *minIndex* és a *min* változók segítségével tudjuk megállapítani az adott ládáról, hogy az megfelel-e a Best-Fit feltételeinek.

Ezután [beszúró rendezéssel](#_3.3.4._Beszúró_rendezés) végrehajtjuk a Best-Fit algoritmust, megpróbáljuk megkeresni a megfelelő ládát. Ha nem talál ilyet az algoritmus, akkor nyit egy új ládát és belehelyezi az újonnan érkezett elemet.

def bestFitForOneItem(bins, remainingSpace, item, maxHeight):  
 a = 0  
 b = len(remainingSpace) - 1  
 minIndex = -1  
 min = sys.maxsize - 1  
 while a <= b:  
 if remainingSpace[math.floor((a + b) / 2)] >= item:  
 if min > remainingSpace[math.floor((a + b) / 2)] - item:  
 min = remainingSpace[math.floor((a + b) / 2)] - item  
 minIndex = math.floor((a + b) / 2)  
 b = math.floor((a + b) / 2) - 1  
 else:   
 a = math.floor((a + b) / 2) + 1  
 if minIndex != -1:  
 del remainingSpace[minIndex]  
 binary(remainingSpace, min)  
 else:  
 remainingSpace.append(maxHeight - item)  
 bins += 1  
 toInsert = remainingSpace[len(remainingSpace) - 1]  
 del remainingSpace[len(remainingSpace) - 1]  
 binary(remainingSpace, toInsert)  
 return bins

Ebben a függvényben a fő lépések megegyeznek a First-Fit algoritmuséban láthatóakkal. Itt további számolási műveletek miatt kétszer hozunk létre maximális magasságú ládát. A számolás elvégzése után megvizsgáljuk, hogy az utolsó láda teljesen üres-e, ha igen, akkor visszatérünk a ládák számával, ha nem, akkor pedig növeljük eggyel a felhasznált ládák számát, majd visszatérünk vele.

def bestFit(maxHeight, data):  
 usedBins = 1  
 remainingSpace = []  
 remainingSpace.append(maxHeight)  
 remainingSpace.append(maxHeight)  
 for i in range(len(data)):  
 usedBins = bestFitForOneItem(usedBins, remainingSpace, data[i], maxHeight)  
 if remainingSpace[-1] == maxHeight:  
 return usedBins  
 else:  
 return usedBins + 1

### **3.3.3. Worst-Fit algoritmus implementációja**

Ennek az algoritmusnak a kódja eltér az eddig bemutatott algoritmusoktól, mivel a Worst-Fit algoritmus esetében a legnagyobb üres résszel rendelkező ládát tekintjük a legoptimálisabbnak. Egy if-else elágazással végre tudjuk hajtani a rendezést. Az utolsó ládába behelyezzük az elemet, majd az előzőekhez hasonlóan meghívjuk a kereső függvényt és a megfelelő helyre átrakjuk a rekeszt. Egyébként nyitunk egy új rekeszt és a szokásos módon elhelyezzük a listában.

def worstFitForOneItem(bins, remainingSpace, item, maxHeight):  
 if remainingSpace[len(remainingSpace) - 1] - item > 0:  
 toInsert = remainingSpace[len(remainingSpace) - 1] - item  
 del remainingSpace[len(remainingSpace) - 1]  
 binary(remainingSpace, toInsert)  
 else:  
 remainingSpace.append(maxHeight - item)  
 bins += 1  
 toInsert = remainingSpace[len(remainingSpace) - 1]  
 del remainingSpace[len(remainingSpace) - 1]  
 binary(remainingSpace, toInsert)  
 return bins

Ennek bemutatását lásd a firstFit függvény jellemzésénél.

def worstFit(maxHeight, data):  
 usedBins = 1  
 remainingSpace = []  
 remainingSpace.append(maxHeight)  
 remainingSpace.append(maxHeight)  
 for i in range(len(data)):  
 usedBins = worstFitForOneItem(usedBins, remainingSpace, data[i], maxHeight)  
 if remainingSpace[-1] == maxHeight:  
 return usedBins  
 else:  
 return usedBins + 1

### **3.3.4. Beszúró rendezés bináris kereséssel**

Az algoritmus lényege az elemek sorrendbe rendezése, majd ezekkel az elemekkel való számolás. Jelen esetben csökkenő sorrendbe vannak rendezve az elemek, majd az újonnan érkező elemet megvizsgálja a függvény, ha nagyobb, mint a sor középső eleme, akkor csak a sor felétől nagyobb részében számol. Ha kisebb, akkor csak a felénél kisebb felével számol. Ezt követően a kiválasztott félen ismétli el a felezést, ezt addig folytatja, amíg meg nem találja az elem helyét.

Ennek a keresésnek az előnye, hogy ahelyett, hogy végig keresnénk az egész tömböt, mindig felezéssel közelítjük az beszúrandó elem helyét. A matematikai korlátok miatt kerekítést kellet alkalmaznom a felezések kiszámolásánál, ezért mindig a lefelé kerekített egész értéket vettem figyelembe.

def binary(remainingSpace, toInsert):  
 a = 0  
 b = len(remainingSpace) - 1  
 while a <= b:  
 if remainingSpace[math.floor((a + b) / 2)] > toInsert:  
 b = math.floor((a + b) / 2) - 1  
 else:  
 a = math.floor((a + b) / 2) + 1  
 remainingSpace.insert(a, toInsert)

### **3.3.5. Túltöltés meghatározása**

Az *overload(c)* függvényben 4 elágazás található, mindegyik ágban más-más ládaméret kerül meghatározásra. A függvény egy számot vár, ennek az értéke alapján végrehajtja az adott ágon lévő számolást, majd visszatér a láda értékének a 100 szorosával. Ez a szorzás azért szükséges, mivel a bemeneti könyvtárban szereplő értékek a [0;1] intervallum helyett a [0; 100] intervallumból kerülnek ki. Az első ág esetén a ládaméret a lehető legnagyobb értéket kapja, így minden algoritmus csak egy ládát használ fel, ebbe pakolja bele az összes érkező elemet.

def overload(c):  
 if (0 <= c) and (c < 3 / 2):  
 s = sys.maxsize - 2  
 return s  
  
 elif (3 / 2 <= c) and (c < 9 / 5):  
 s = 1 + (1 / c)  
  
 elif (9 / 5 <= c) and (c <= 14 / 3):  
 s = 1 + 2 / (3 \* c)  
  
 else:  
 s = 1 + 1 / (3 \* c)  
 return s \* 100

Az alább látható kódrészlet egy egyszerű *for ciklus,* melyben úgy állítom be az *x* értékét, hogy az *overload(x)* függvény minden ága le legyen fedve. Ez mindössze a szemléltetés céljából történik, valamint ellenőrizhető, hogy megfelelően történik-e az összes ládaméret kiszámolása.

for x in range(1, 5):  
 temp = x  
 if x == 2:  
 temp = 1.65  
 readDirectory(temp)  
 elif x == 4:  
 temp = 5  
 readDirectory(temp)  
 else:  
 readDirectory(temp)

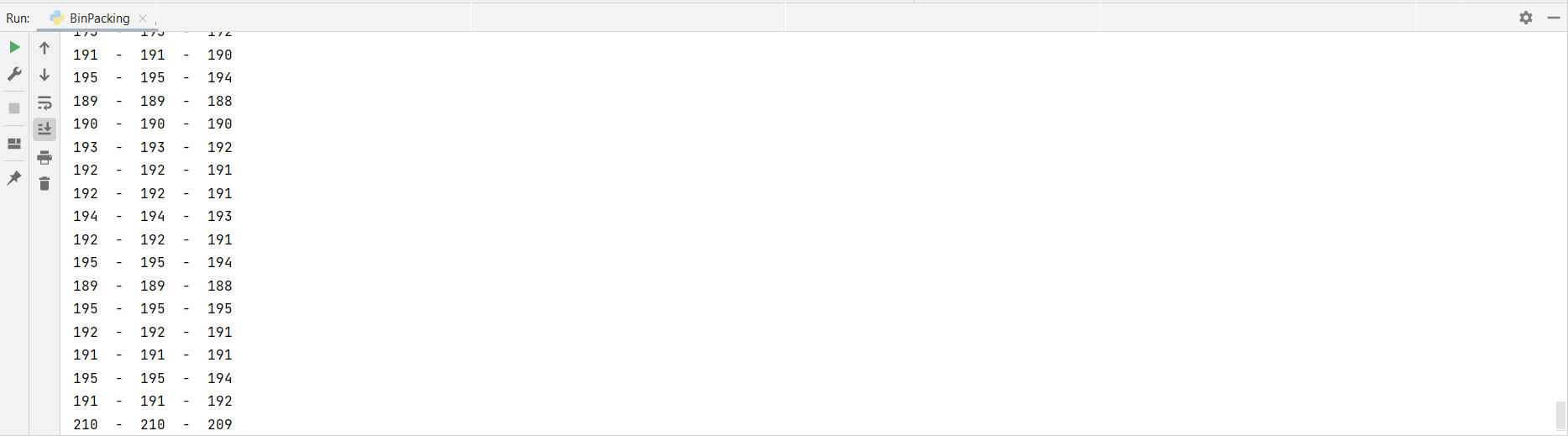
Az *x* értéket minden iteráció során kimentem egy átmeneti *temp* változóban, majd a 2. és a 4. iteráció esetében ezt az értéket átállítom úgy, hogy megfeleljen a túltöltést számoló függvény elágazásai kritériumainak. Ezt követően meghívom a *readDirectory()* függvényt a *temp* változóra.

## **3.4. Eredmények**

Miután az adatok be lettek olvasva a bemeneti fájlokból, át lettek konvertálva megfelelő típusúvá, valamint végre lett rajtuk hajtva az összes rendező algoritmus, a végeredmény konzolon, valamit grafikonon kerül kiírásra.

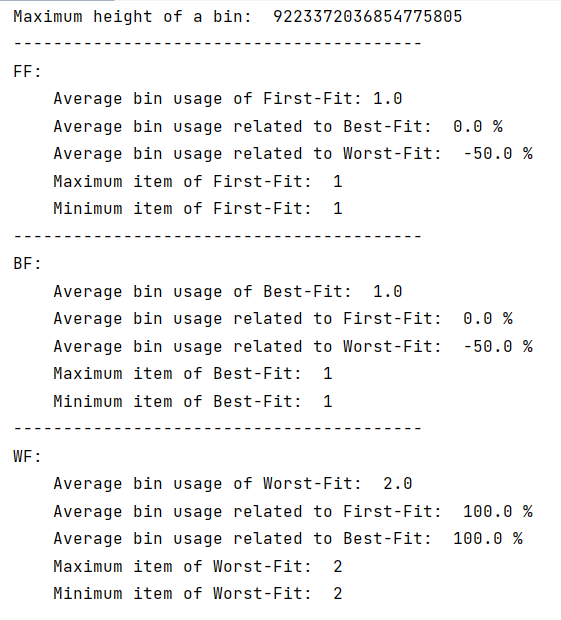
Miután az összes algoritmus lefutott az adott bementre, adott túltöltési értékkel, azután kiíródik a konzolra egy rövid összefoglaló, melyben az adatokról látható néhány információ. Itt leolvasható az adott iterációban használt ládamagasság, az átlag ládafelhasználás, többi algoritmushoz hasonlított ládafelhasználás, maximum, illetve minimum elem, ezek mindegyike algoritmusonként van kiírva.

Az alábbi kép szemlélteti az eredmények konzolra írt formátumát, ezekből az adatokból több ezer szerepel, ezért egy kis része szerepel csak a képen.



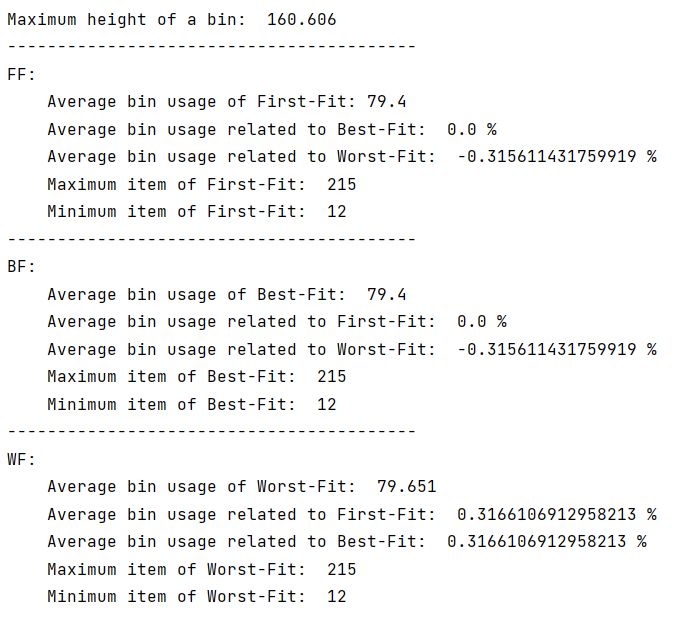
5. ábra Adatok konzolon megjelenítve

A 6. ábra szemlélteti az első iteráció eredményein végrehajtott elemzés eredményeit. Látható, hogy az *overload(x)* meghatározása alapján a ládamagasság a python által használt lehető legnagyobb egész értéket kapta, ezzel biztosítva, hogy az összes beérkező elem egy ládába lesz pakolva. Ennek köszönetően az algoritmusok eredményének elemzése között nem fedezhetőek fel számottevő különbségek.



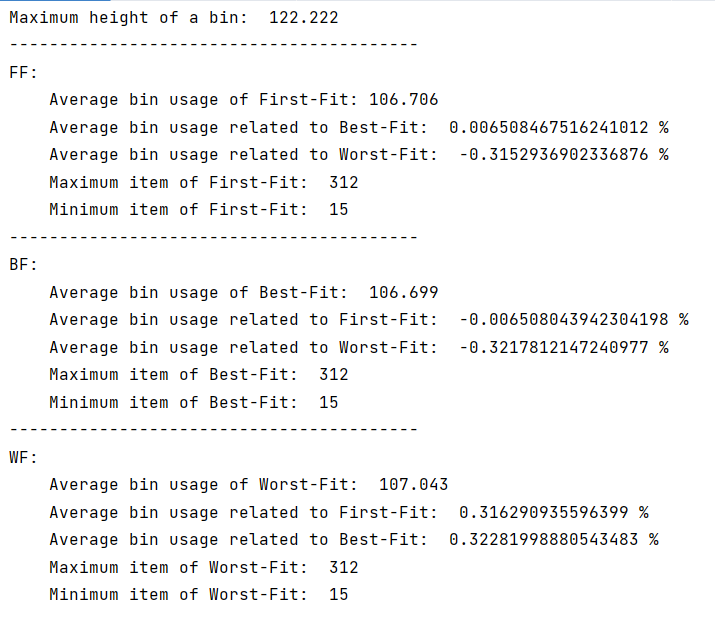
6. ábra Első iteráció elemzései

Az 7.ábra szemlélteti a második iteráció eredményein végrehajtott elemzés eredményeit. A ládamagasságot meghatározó függvény második ágában vagyunk, tehát a ládamagasság értéke 160.606. Ez a képlettel lett kiszámolva, ahol *c* = 1.65, tehát a magasság. Az első ággal ellentétnem ott már felfedezhetőek különbségek.



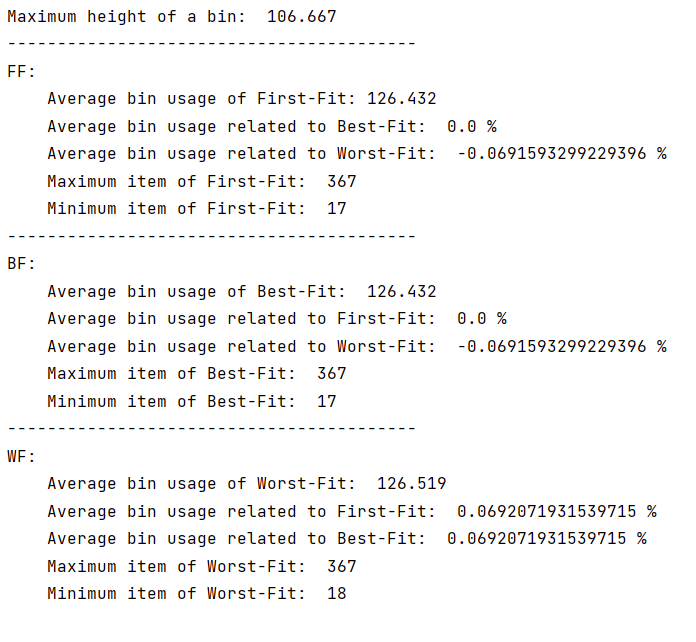
7. ábra Második iteráció elemzései

A 8.ábra szemlélteti az harmadik iteráció eredményein végrehajtott elemzés eredményeit.



8. ábra Harmadik iteráció elemzései

Az 9. ábra szemlélteti a negyedik iteráció eredményein végrehajtott elemzés eredményeit.



9. ábra Negyedik iteráció elemzései

A diagramok x tengelye jelöli a bemeneti fájlokon történő előrehaladást, ez az érték maximum 720 lehet,mivel ennyi fájlból történik a beolvasás. A y tengely szemlélteti a felhasznált ládák számát, ez az érték nagyban változik, mivel sem a ládaméretek, sem a felhasznált ládák száma nem állandó.

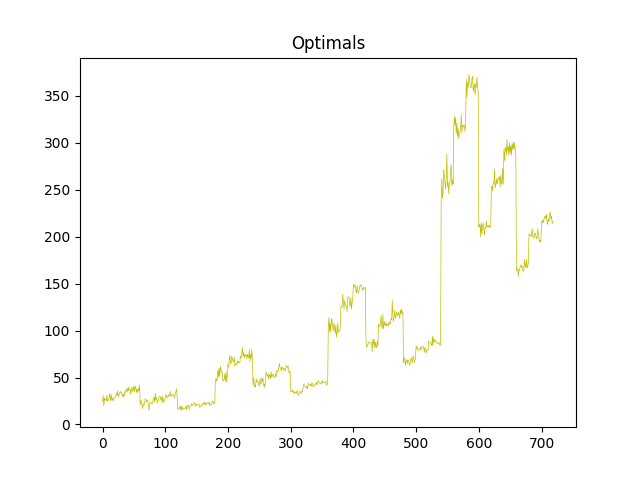
A lefuttatott algoritmusok eredményei túltöltési értékenként ábrázolva vannak egy-egy diagramon. Mivel a túltöltési érték egy ciklus segítségével van növelve a program futása során, így az algoritmusok is több alkalommal futnak le, mindig más eredményt adva. Ezért a diagramok generálása is minden túltöltési érték esetén újra lejátszódik. Tehát minden túltöltési értékre létrejön mind a három algoritmusból egy-egy példány.

Minden algoritmus különöbző színnel, valamint vonaltípussal van ábrázolva. A First-Fit algoritmus diagramjain az értékek zöld színnel, a Best-Fit algoritmus értékei piros szaggatott vonallal, míg a Worst-Fit által létrehozott értékek kék, szaggatott vonallal lettek megjelenítve. Miután lefutott minden algoritmus ábrázolása, elkészül egy összegző diagram, melyben mindhárom eredmény látható, ezek a diagramok szintén túltöltési értékenként generálódnak.

Az alábbi kódrészlet a *readDirectory(x)* függvényből lett kimásolva, a diagramok létrehozást szemlélteti.

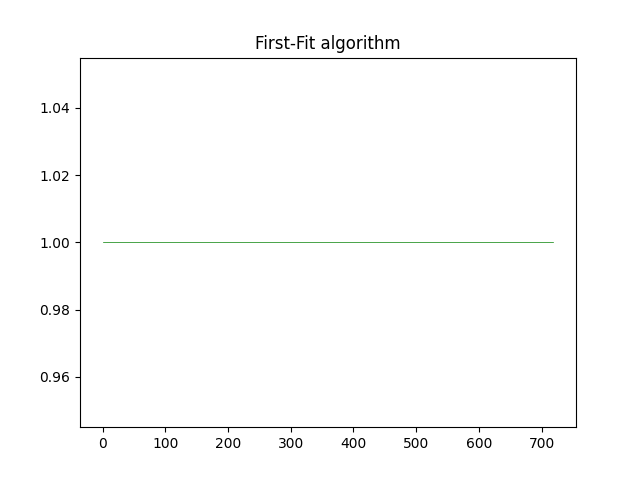
fig, ax = plt.subplots()  
fig, ax.plot(optimal,'y', linewidth=0.5)  
ax.set\_title('Optimals')  
plt.show()  
  
fig, ax = plt.subplots()  
fig, ax.plot(firstFitList, 'g', linewidth=0.5)  
ax.set\_title('First-Fit algorithm')  
plt.show()  
  
fig, ax = plt.subplots()  
ax.plot(bestFitList, 'r--', linewidth=0.5)  
ax.set\_title('Best-Fit algorithm')  
plt.show()  
  
fig, ax = plt.subplots()  
ax.plot(worstFitList, 'b-', linewidth=0.5)  
ax.set\_title('Worst-Fit algorithm')  
plt.show()  
  
fig, ax = plt.subplots()  
ax.plot(firstFitList, 'g', bestFitList, 'r--', worstFitList, 'b-', optimal,'y', linewidth=0.5)  
ax.set\_title('Summarized')  
plt.show()

A 10. ábrán látható az optimális eredmények diagramon történő ábrázolása. Ebben az esetben a ládaméret 1-re volt meghatározva, mely jelen esetben 100-at jelentene, mivel a használt elemek 100 szorosai a minta bemenet értékének. A diagramon látszódik, hogy a kezdetben a bemeneti adatok mérete kisebb, majd ez egészen a kezdeti érték több, mint hétszeresére növekszik. Ez is bizonyítja, hogy a mintkönyvtár alkalmas az algoritmusok többoldaló tesztelésére.



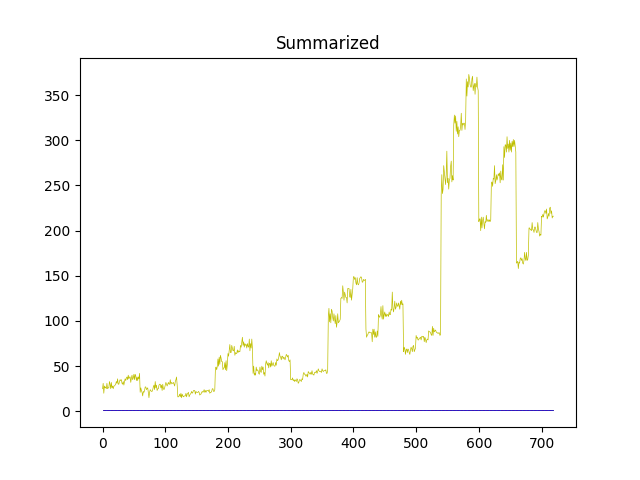
10. ábra Optimális eredmények

Mivel az első iterációban a ládaméret a Python által feldogozott maximum érték, így az algoritmusok akármennyi bejövő elem esetén mindössze egy ládát nyitnak és ebbe pakolják bele az összes érkező elemet. Ennek az iterációja nem túl látványos, mindössze egy konstans vízszintes vonal az y tengely 1 értékénel, mivel ez jelzi a felhasznált ládaszámot. A Best-Fit, valamint a Worst-Fit algoritmus diagramja megegyezik a 10.ábrán látható függvénnyel, emiatt csak egy diagramot mutatok be, azonban a forrásfájlokban megtalálható az összes létrehozott diagram.



11. ábra First-Fit algoritmus maxiumáls ládamérettel

Mivel mind három algoritmus 1 ládát használ fel és a diagramuk is megegyezik, ezért az optimális adatokhoz való viszonyításuk is teljesen megegyezik. Ezért a 12. ábrán mindössze egy x tengellyel párhuzamos kék vonal látszódik, ezzel megegyezik First-Fit, a Best-Fit, valamint a Worst-Fit algoritmus függvénye is. A sárga függvény jelzi az optimális láda felhasználási számot.



12. ábra Maximum ládamagasságú pakolás összehasonlítása az optimális értékekkel

A második iteráció során a ládamagasság 1 helyett, . Ebben az esetben az algoritmusok között felfedezhető eltérés, azonban a nagy elemszámú bemenet miatt az eltérés minimális.

## **3.5. Bővítési lehetőségek**

# **4. Forrásjegyzet**

## **4.1. Saját hivatkozások**

**[1]** Python programozási nyelv hivatalos weboldal:

<https://www.python.org>

**[2]** PyCharm hivatalos weboldal:

<https://www.jetbrains.com/pycharm/>

**[3]** Saját GitHub repository:

<https://github.com/boss1203/thesis>

**[4]** A. Scholl, R. Klein és C. Jürgens in Bison: egydimenziós ládapakolási köyvtár

<https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm>

## **4.2. Szakirodalmi hivatkozások**

Szerző: Luo, K. and Spieksma, F.C.R.

Cím: Online Bin Packing with Overload Cost

A szakirodalmi hivatkozások a fent megemlített cikkből származnak.

**[[LS21]]** DOI: <https://doi.org/10.1007/978-3-030-67899-9_1>

**[[1]]** A legjobb ismert versenyképességi hányadosú online algoritmus:

Balogh, J., Békési, J., Dósa, G., Epstein, L., Levin, A.: A new and improved algorithm for online bin packing. In: Azar, Y., Bast, H., Herman, G. (eds.) 26th Annual European Symposium on Algorithms, ESA 2018, 20–22 August 2018, Helsinki, Finland. LIPIcs, vol. 112, pp. 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)

**[[2]]** A legjobb ismert alsó korlát az online algoritmusok asszimptotikus versenyképességi arányára:

Balogh, J., Békési, J., Dósa, G., Epstein, L., Levin, A.: A new lower bound for classic online bin packing. In: Bampis, E., Megow, N. (eds.) WAOA 2019. LNCS, vol. 11926, pp. 18–28. Springer, Cham (2020).

https://link.springer.com/chapter/10.1007%2F978-3-030-39479-0\_2

**[[3]]** A legjobb elérhető, 5/3-os abszolút versenyképességi arányú online algoritmus:

Balogh, J., Békési, J., Dósa, G., Sgall, J., van Stee, R.: The optimal absolute ratio for online bin packing. J. Comput. Syst. Sci. 102, 1–17 (2019)

**[[5]]** Variable Harmonic algoritmus:

Csirik, J.: An on-line algorithm for variable-sized bin packing. Acta Inf. 26(8), 697–709 (1989)

**[[7]]** FF és BF abszolút versenyképességi aránya is 1,7:

Dósa, G., Sgall, J.: First fit bin packing: a tight analysis. In: 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)

**[[8]]** FF és BF abszolút versenyképességi aránya is 1.7:Dósa, G., Sgall, J.: Optimal analysis of best fit bin packing. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp.429–441. Springer, Heidelberg (2014).

https://link.springer.com/chapter/10.1007%2F978-3-662-43948-7\_36

**[[9]]** Epstein and Levin - strong open end és lazy bin covering:

Epstein, L., Levin, A.: Asymptotic fully polynomial approximation schemes for variants of open-end bin packing. Inf. Process. Lett. 109(1), 32–37 (2008)

**[[10]]** Epstein, L., Levin, A.: Bin packing with general cost structures. Math. Program. 132(1–2), 355–391 (2012)

**[[11]]** Epstein, L., Levin, A.: An AFPTAS for variable sized bin packing with general activation costs. J. Comput. Syst. Sci. 84, 79–96 (2017)

**[[12]]** BF és FF aszimptotikus teljesítményaránya 1,7:

Johnson, D.S., Demers, A.J., Ullman, J.D., Garey, M.R., Graham, R.L.: Worstcase performance bounds for simple one-dimensional packing algorithms. SIAM J. Comput. 3(4), 299–325 (1974)

**[[13]]** Változó ládaméretű ládapakolás:

Kinnersley, N.G., Langston, M.A.: Online variable-sized bin packing. Discrete Appl. Math. 22(2), 143–148 (1989)

**[[14]]** Li, C.L., Chen, Z.L.: Bin-packing problem with concave costs of bin utilization. Naval Res. Logist. (NRL) 53(4), 298–308 (2006)

**[[15]]** Seiden Harmonic Variable algoritmus elemzése:

Seiden, S.S.: An optimal online algorithm for bounded space variable-sized bin packing. SIAM J. Discrete Math. 14(4), 458–470 (2001)

**[[16]]** Egy áttekintés az online ládapakolásról:

Sgall, J.: Online bin packing: old algorithms and new results. In: Beckmann, A., Csuhaj-Varjú, E., Meer, K. (eds.) CiE 2014. LNCS, vol. 8493, pp. 362–372. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08019-2 38](https://doi.org/10.1007/978-3-319-08019-2%2038)

**[[17]]** First Fit és Best Fist algoritmus elemzése egészen eddig a cikkig nyúlik vissza:

Ullman, J.D.: The performance of a memory allocation algorithm. Technical report 100, Princeton University, Princeton, NJ (1971)

**[[18]]** Open-end bin. packing:

Yang, J., Leung, J.Y.: The ordered open-end bin-packing problem. Oper. Res.51(5), 759–770 (2003)

## Nyilatkozat

Alulírott ………………..………… szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet ……………………….. Tanszékén készítettem, …………………….…….…… diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Dátum

Aláírás